



A Statistical Machine Learning Perspective of Deep Learning: Algorithm, Theory, Scalable Computing

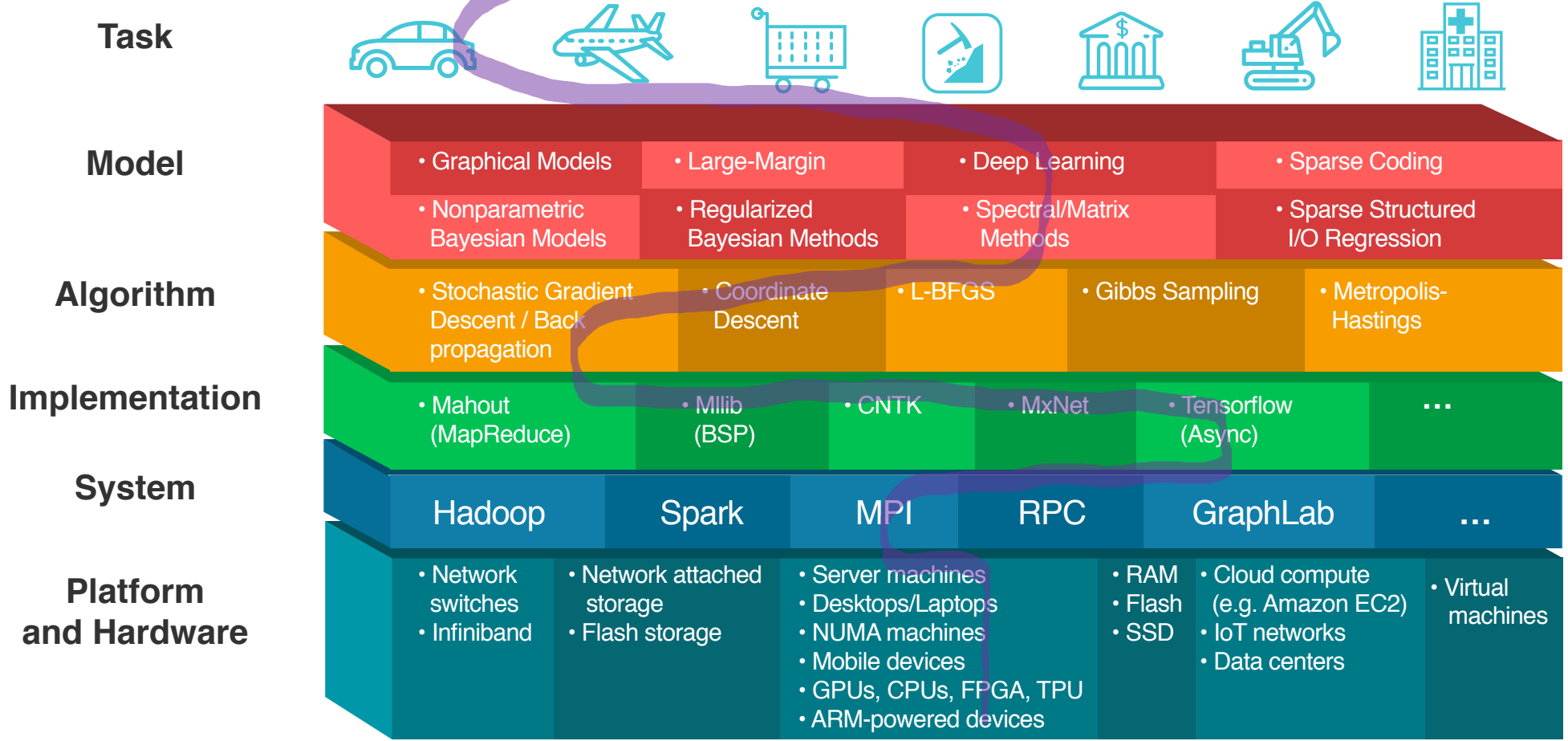
Maruan Al-Shedivat, Zhiting Hu, Hao Zhang, and *Eric Xing*

Petuum Inc

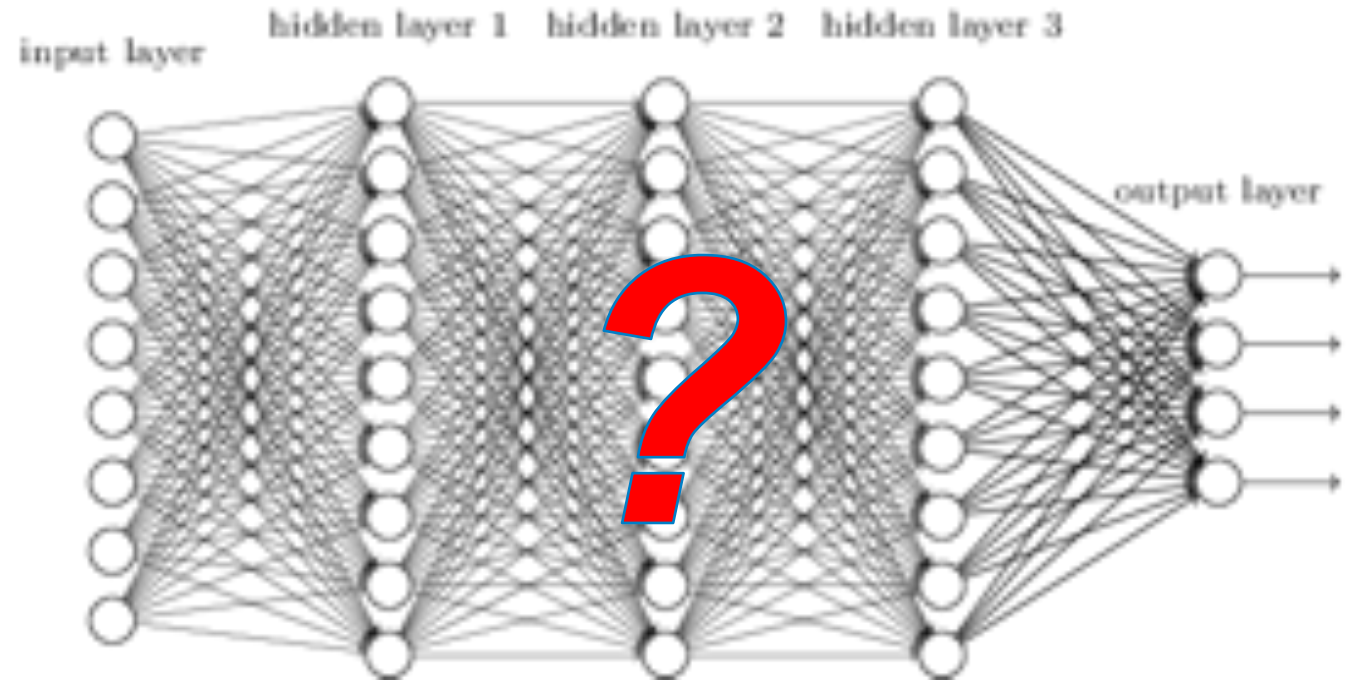
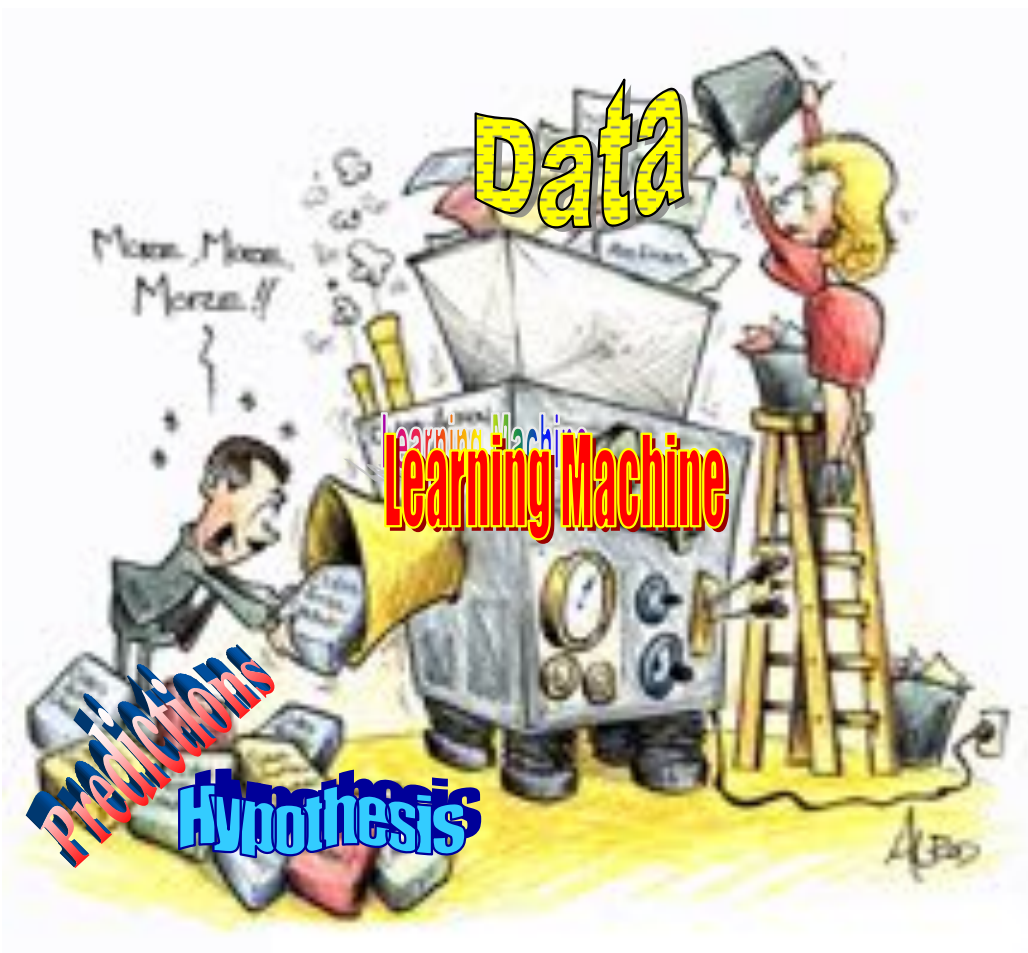
&

Carnegie Mellon University

Element of AI/Machine Learning



ML vs DL



Plan

- Statistical And Algorithmic Foundation and Insight of Deep Learning
- On Unified Framework of Deep Generative Models
- Computational Mechanisms: Distributed Deep Learning Architectures



Part-I Basics

Outline

- Probabilistic Graphical Models: Basics
- An overview of DL components
 - Historical remarks: early days of neural networks
 - Modern building blocks: units, layers, activations functions, loss functions, etc.
 - Reverse-mode automatic differentiation (aka backpropagation)
- Similarities and differences between GMs and NNs
 - Graphical models vs. computational graphs
 - Sigmoid Belief Networks as graphical models
 - Deep Belief Networks and Boltzmann Machines
- Combining DL methods and GMs
 - Using outputs of NNs as inputs to GMs
 - GMs with potential functions represented by NNs
 - NNs with structured outputs
- Bayesian Learning of NNs
 - Bayesian learning of NN parameters
 - Deep kernel learning

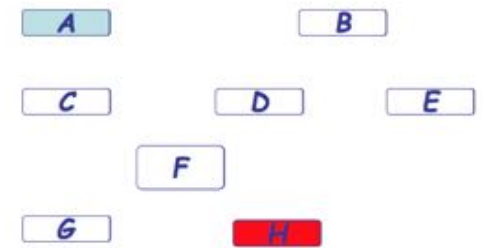
Outline

- Probabilistic Graphical Models: Basics
- An overview of DL components
 - Historical remarks: early days of neural networks
 - Modern building blocks: units, layers, activations functions, loss functions, etc.
 - Reverse-mode automatic differentiation (aka backpropagation)
- Similarities and differences between GMs and NNs
 - Graphical models vs. computational graphs
 - Sigmoid Belief Networks as graphical models
 - Deep Belief Networks and Boltzmann Machines
- Combining DL methods and GMs
 - Using outputs of NNs as inputs to GMs
 - GMs with potential functions represented by NNs
 - NNs with structured outputs
- Bayesian Learning of NNs
 - Bayesian learning of NN parameters
 - Deep kernel learning

Fundamental questions of probabilistic modeling

- **Representation:** what is the joint probability distr. on multiple variables?

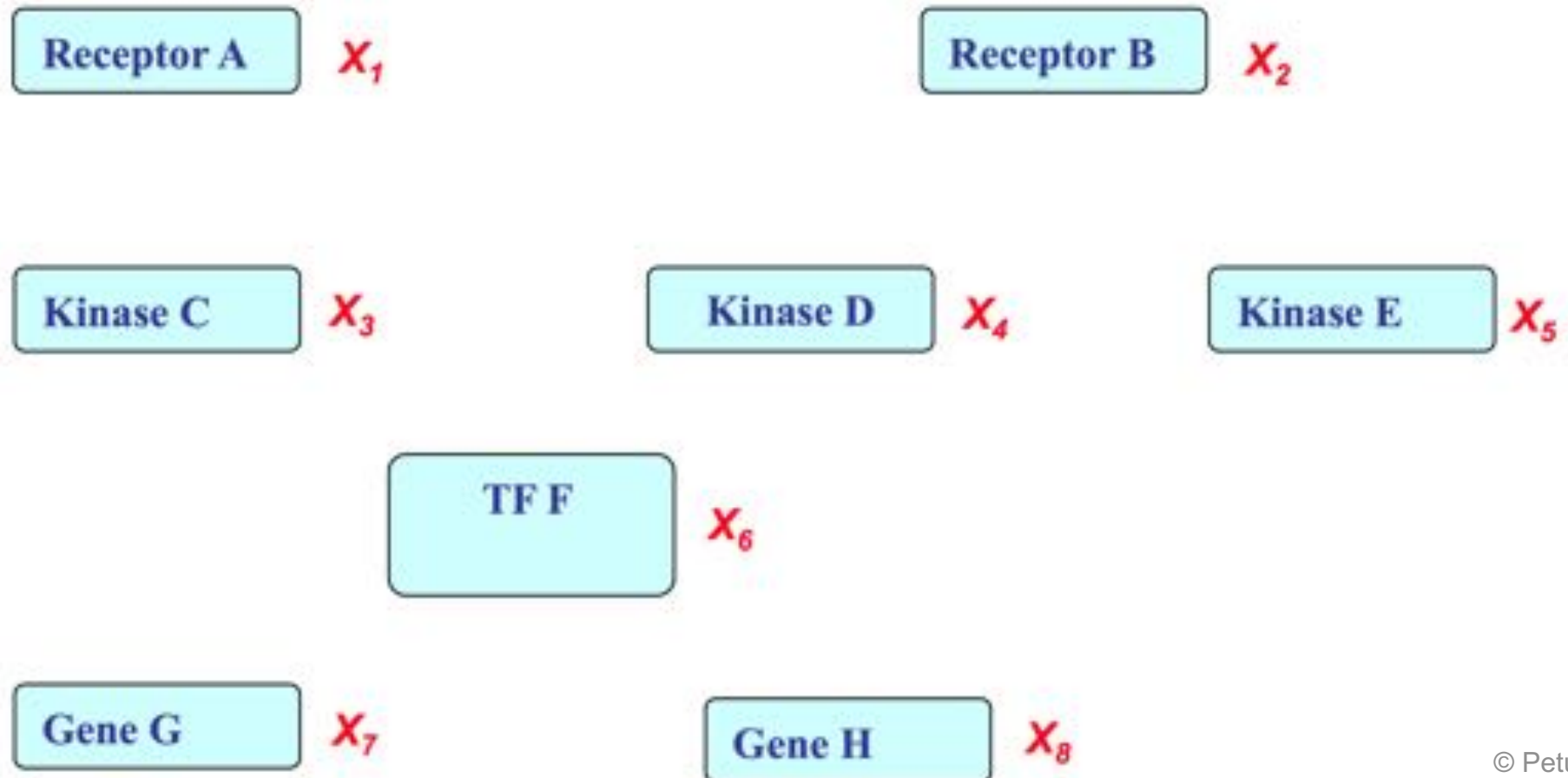
$$P(X_1, X_2, X_3, \dots, X_d)$$



- How many state configurations are there?
 - Do they all need to be represented?
 - Can we incorporate any domain-specific insights into the representation?
- **Learning:** where do we get the probabilities from?
 - Maximum likelihood estimation? How much data do we need?
 - Are there any other established principles?
 - **Inference:** if not all variables are observable, how to compute the conditional distribution of latent variables given evidence?
 - Computing $P(H|A)$ would require summing over 2^6 configurations of the unobserved variables

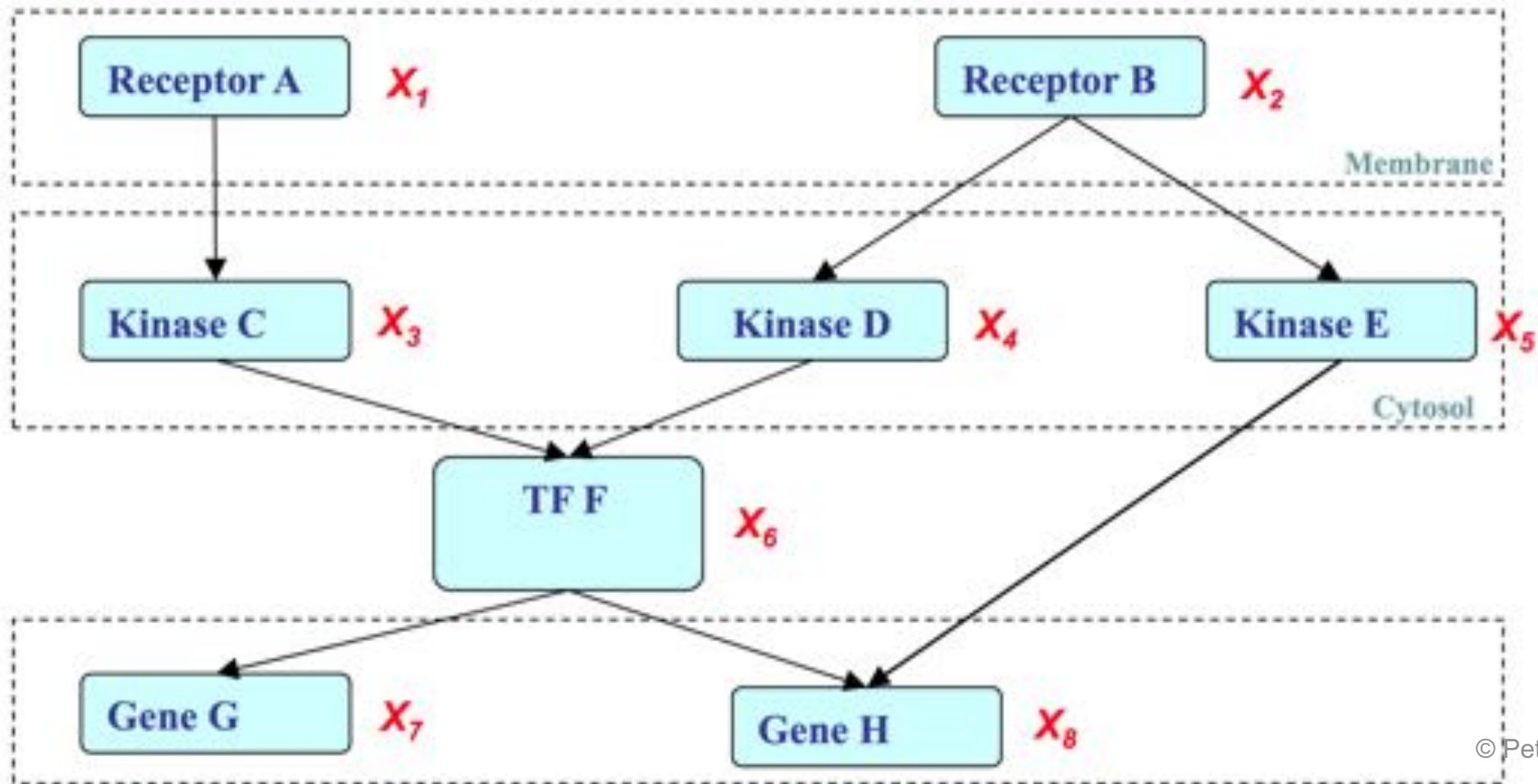
What is a graphical model?

- A possible world of cellular signal transduction



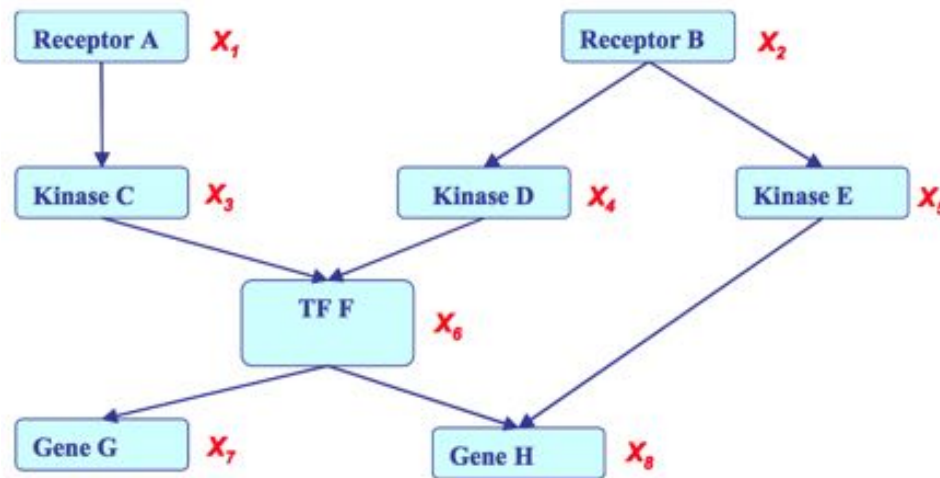
GM: structure simplifies representation

- A possible world of cellular signal transduction



Probabilistic Graphical Models

- If X_i 's are **conditionally independent** (as described by a **PGM**), then the joint can be factored into a product of simpler terms



$$\begin{aligned}
 P(X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8) = & \\
 & P(X_1)P(X_2)P(X_3|X_1)P(X_4|X_2)P(X_5|X_2) \\
 & P(X_6|X_3, X_4)P(X_7|X_6)P(X_8|X_5, X_6)
 \end{aligned}$$

- Why we may favor a PGM?
 - Easy to incorporate domain knowledge and causal (logical) structures
 - Significant reduction in representation cost (2^8 reduced down to 18)

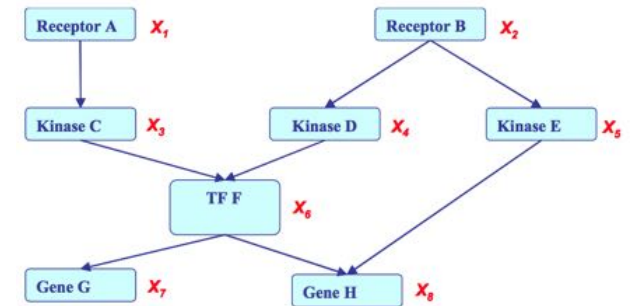
The two types of GMs

$$P(H|V)$$

$$\theta = \operatorname{argmax}_{\theta} P_{\theta}(V)$$

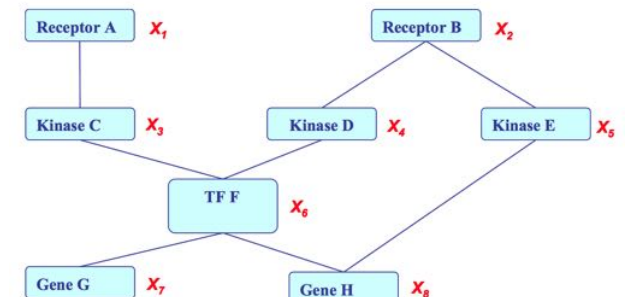
- **Directed edges** assign causal meaning to the relationships (Bayesian Networks or Directed Graphical Models)

$$P(X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8) = P(X_1)P(X_2)P(X_3|X_1)P(X_4|X_2)P(X_5|X_2)P(X_6|X_3, X_4)P(X_7|X_6)P(X_8|X_5, X_6)$$



- **Undirected edges** represent correlations between the variables (Markov Random Field or Undirected Graphical Models)

$$P(X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8) = \frac{1}{Z} \exp\{E(X_1) + E(X_2) + E(X_1, X_3) + E(X_2, X_4) + E(X_5, X_2) + E(X_3, X_4, X_6) + E(X_6, X_7) + E(X_5, X_6, X_8)\}$$

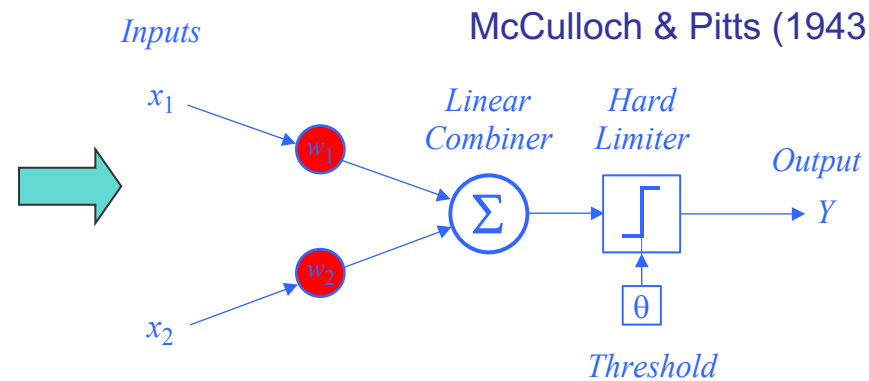
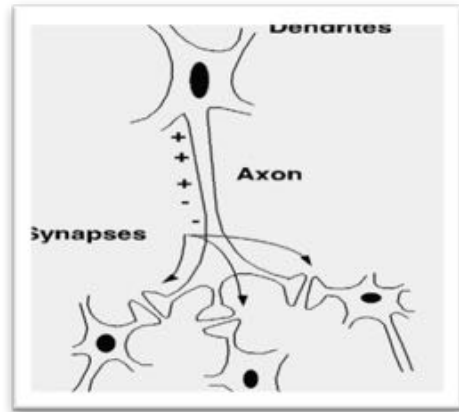


Outline

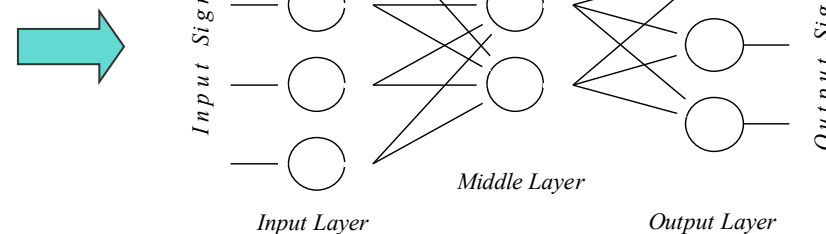
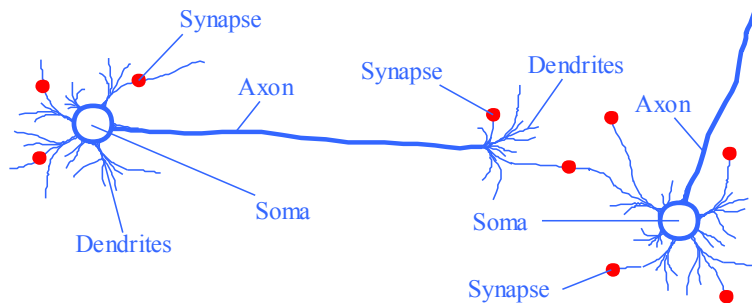
- Probabilistic Graphical Models: Basics
- An overview of DL components
 - Historical remarks: early days of neural networks
 - Modern building blocks: units, layers, activations functions, loss functions, etc.
 - Reverse-mode automatic differentiation (aka backpropagation)
- Similarities and differences between GMs and NNs
 - Graphical models vs. computational graphs
 - Sigmoid Belief Networks as graphical models
 - Deep Belief Networks and Boltzmann Machines
- Combining DL methods and GMs
 - Using outputs of NNs as inputs to GMs
 - GMs with potential functions represented by NNs
 - NNs with structured outputs
- Bayesian Learning of NNs
 - Bayesian learning of NN parameters
 - Deep kernel learning

Perceptron and Neural Nets

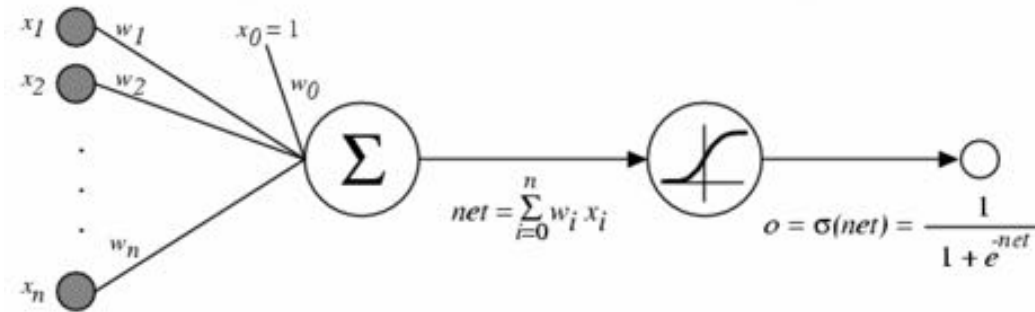
- From biological neuron to artificial neuron (perceptron)



- From biological neuron network to artificial neuron networks



The perceptron learning algorithm



- Recall the nice property of sigmoid function $\frac{d\sigma}{dt} = \sigma(1 - \sigma)$
- Consider regression problem $f: X \rightarrow Y$, for scalar Y : $y = f(x) + \epsilon$
- We used to maximize the conditional data likelihood

$$\vec{w} = \arg \max_{\vec{w}} \ln \prod_i P(y_i | x_i; \vec{w})$$

- Here ...

$$\vec{w} = \arg \min_{\vec{w}} \sum_i \frac{1}{2} (y_i - \hat{f}(x_i; \vec{w}))^2$$

The perceptron learning algorithm

$$\begin{aligned}
 \frac{\partial E_D[\vec{w}]}{\partial w_j} &= \frac{\partial}{\partial w_i} \frac{1}{2} \sum_d (t_d - o_d)^2 \\
 &= \frac{1}{2} \sum_d 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d) \\
 &= \sum_d (t_d - o_d) \left(-\frac{\partial o_d}{\partial w_i} \right) \\
 &= -\sum_d (t_d - o_d) \frac{\partial o_d}{\partial net_i} \frac{\partial net_d}{\partial w_i} \\
 &= -\sum_d (t_d - o_d) o_d (1 - o_d) x_d^i
 \end{aligned}$$

Batch mode:

Do until converge:

1. compute gradient $\nabla E_D[\vec{w}]$
2. $\vec{w} = \vec{w} - \eta \nabla E_D[\vec{w}]$

x_d = input

t_d = target output

o_d = observed output

w_i = weight i

Incremental mode:

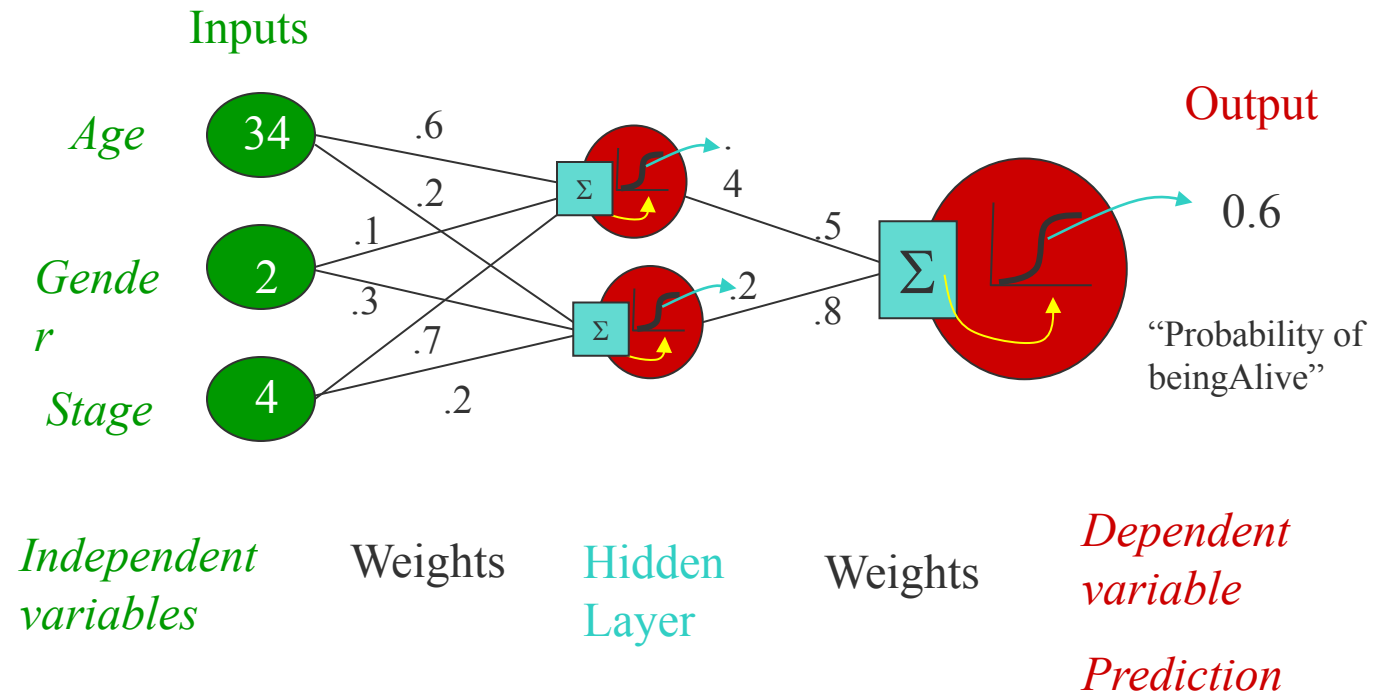
Do until converge:

- For each training example d in D
 1. compute gradient $\nabla E_d[\vec{w}]$
 2. $\vec{w} = \vec{w} - \eta \nabla E_d[\vec{w}]$

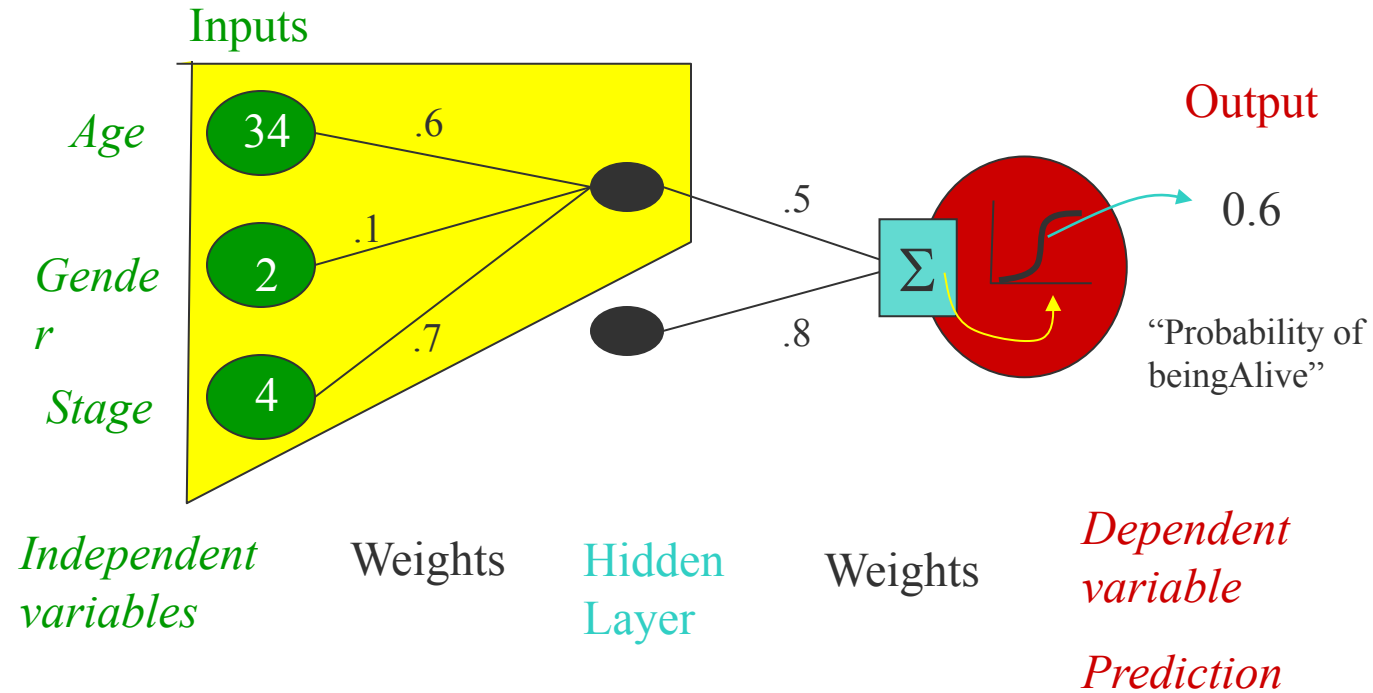
where

$$\nabla E_d[\vec{w}] = -(t_d - o_d) o_d (1 - o_d) \vec{x}_d$$

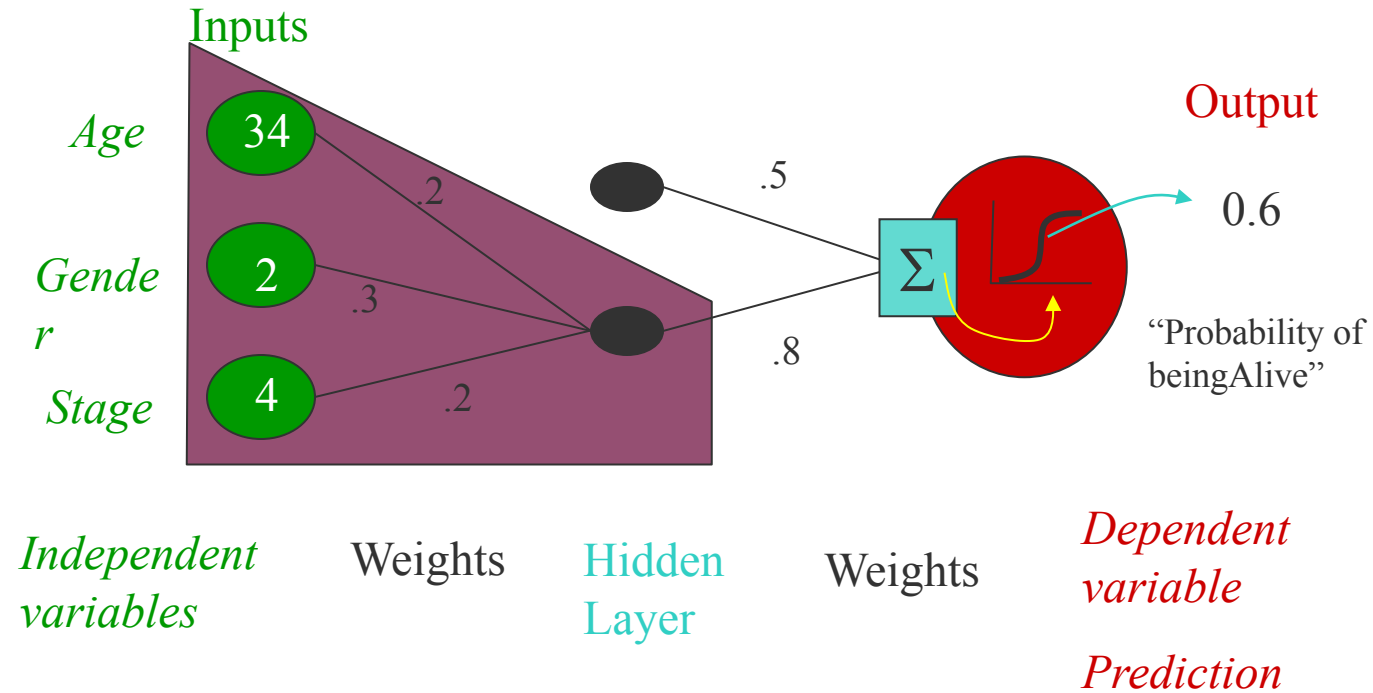
Neural Network Model



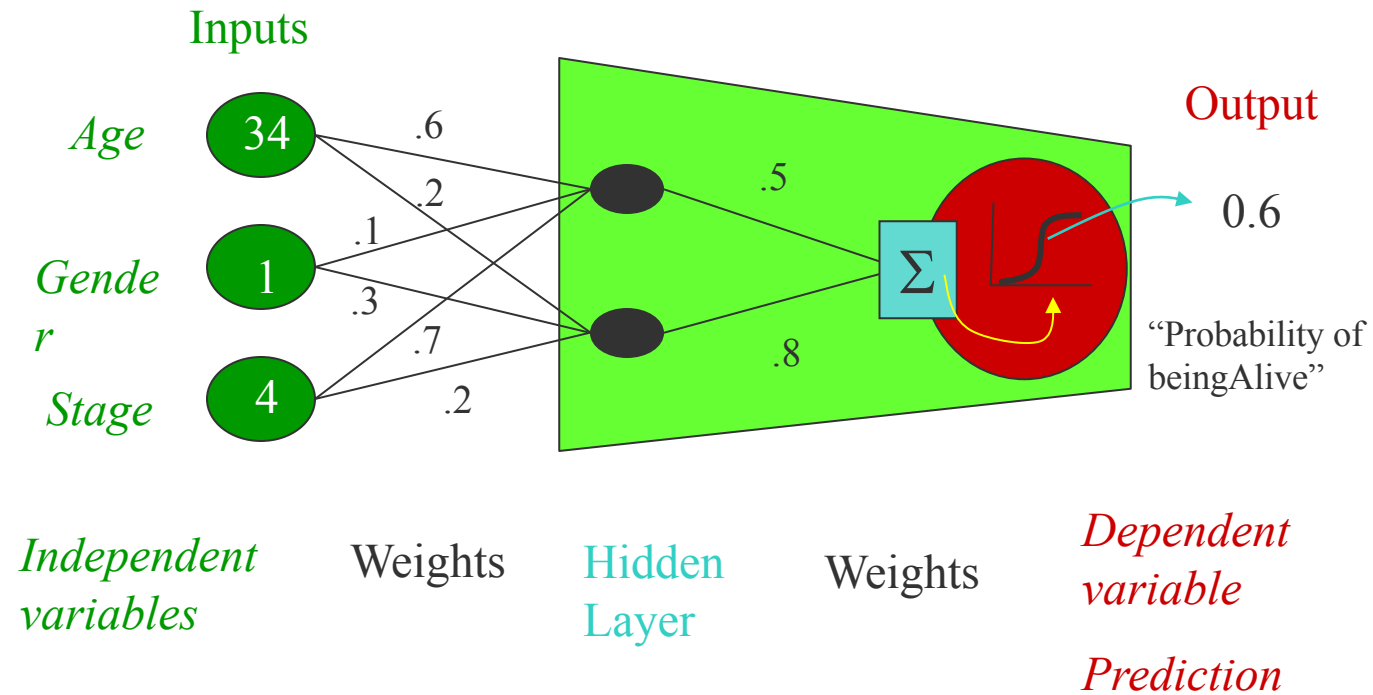
“Combined logistic models”



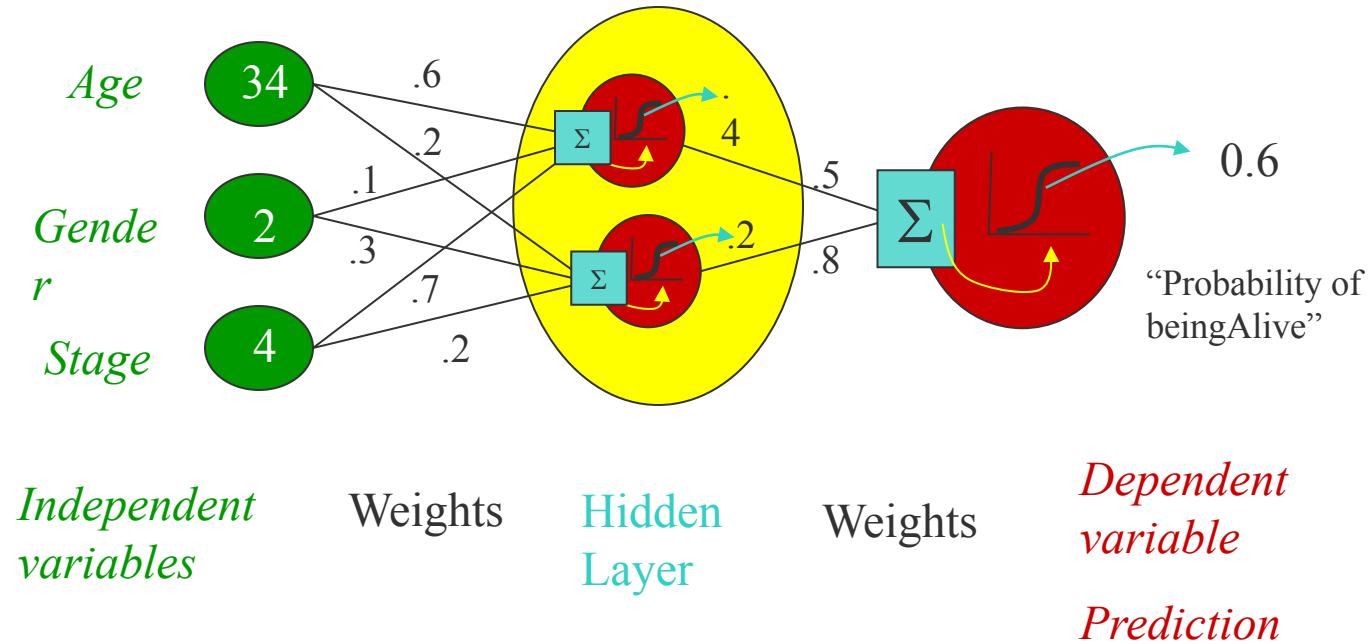
“Combined logistic models”



“Combined logistic models”

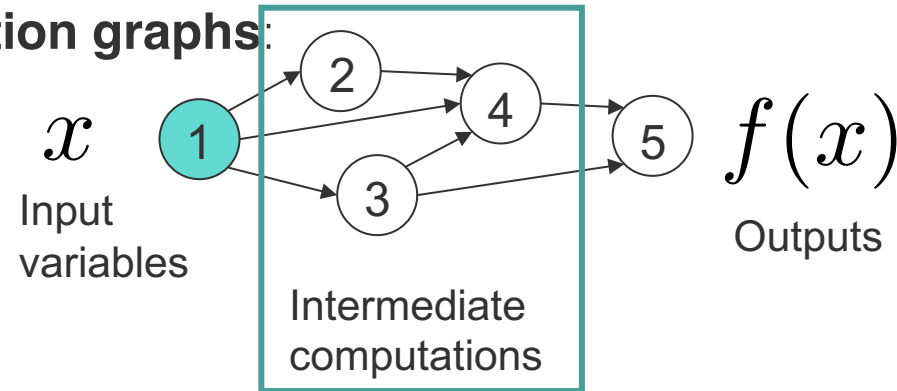


Not really, no target for hidden units...



Backpropagation: Reverse-mode differentiation

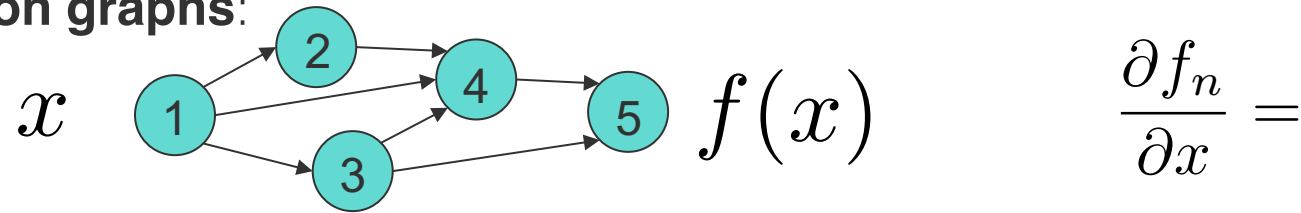
- Artificial neural networks are nothing more than complex functional compositions that can be represented by **computation graphs**:



$$\frac{\partial f_n}{\partial x} =$$

Backpropagation: Reverse-mode differentiation

- Artificial neural networks are nothing more than complex functional compositions that can be represented by **computation graphs**:



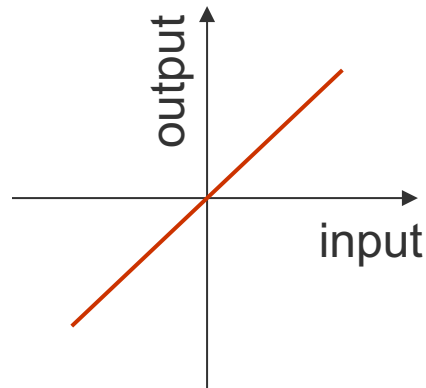
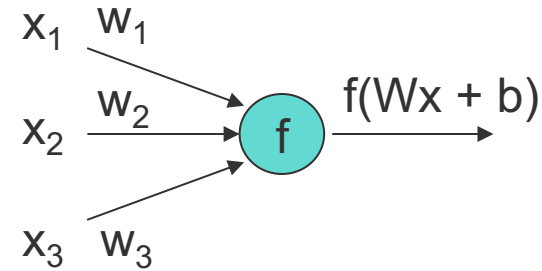
- By applying the chain rule and using reverse accumulation, we get

$$\frac{\partial f_n}{\partial x} = \sum_{i_1 \in \pi(n)} \frac{\partial f_n}{\partial f_{i_1}} \frac{\partial f_{i_1}}{\partial x} = \sum_{i_1 \in \pi(n)} \frac{\partial f_n}{\partial f_{i_1}} \sum_{i_2 \in \pi(i_1)} \frac{\partial f_{i_1}}{\partial f_{i_2}} \frac{\partial f_{i_2}}{\partial x} = \dots$$

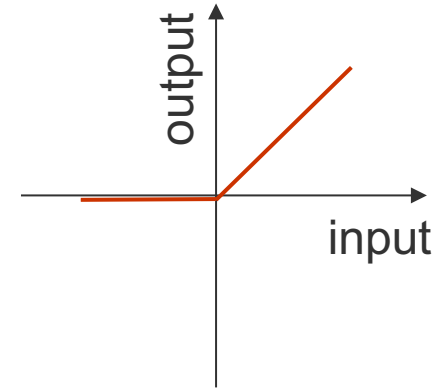
- The algorithm is commonly known as backpropagation
- What if some of the functions are stochastic?
- Then use **stochastic backpropagation!**
(to be covered in the next part)
- Modern packages can do this *automatically* (more later)

Modern building blocks of deep networks

- Activation functions
 - Linear and ReLU
 - Sigmoid and tanh
 - Etc.



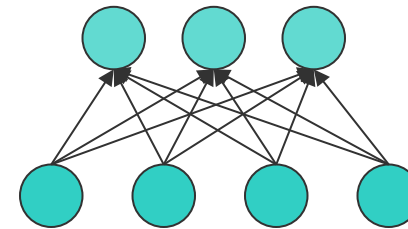
Linear



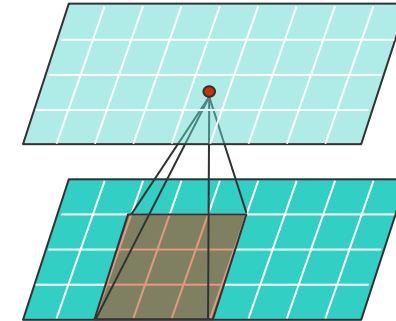
Rectified linear (ReLU)

Modern building blocks of deep networks

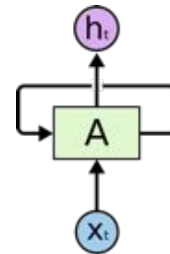
- Activation functions
 - Linear and ReLU
 - Sigmoid and tanh
 - Etc.
- Layers
 - Fully connected
 - Convolutional & pooling
 - Recurrent
 - ResNets
 - Etc.



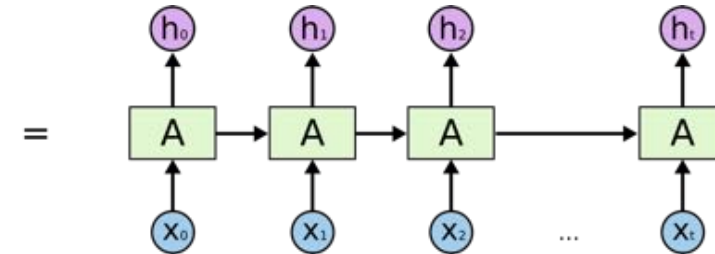
fully connected



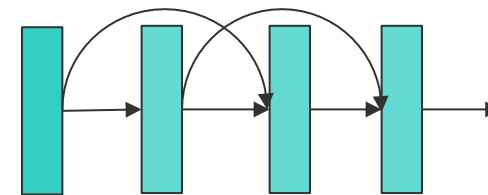
convolutional



recurrent



source: colah.github.io

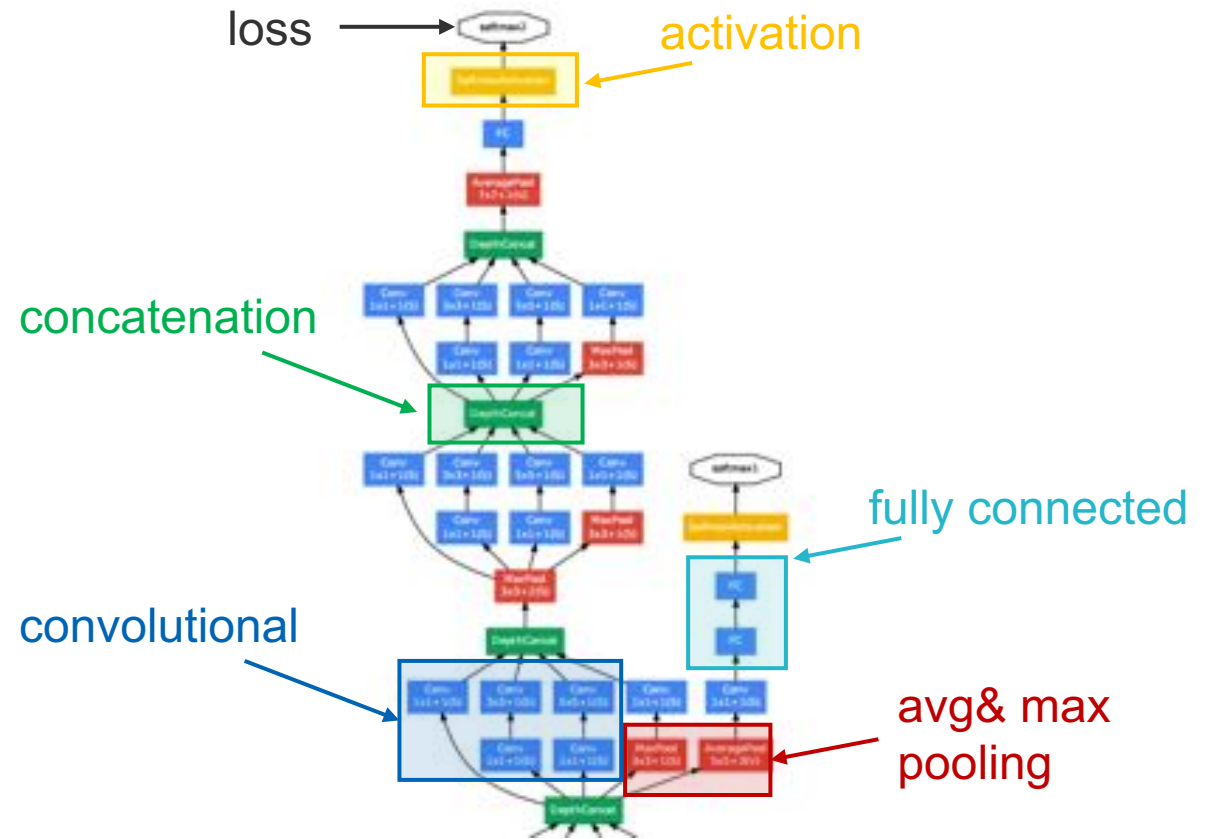


blocks with residual connections

Modern building blocks of deep networks

- Activation functions
 - Linear and ReLU
 - Sigmoid and tanh
 - Etc.
- Layers
 - Fully connected
 - Convolutional & pooling
 - Recurrent
 - ResNets
 - Etc.
- Loss functions
 - Cross-entropy loss
 - Mean squared error
 - Etc.

Putting things together:



(a part of GoogleNet)

Modern building blocks of deep networks

- Activation functions
 - Linear and ReLU
 - Sigmoid and tanh
 - Etc.
- Layers
 - Fully connected
 - Convolutional & pooling
 - Recurrent
 - ResNets
 - Etc.
- Loss functions
 - Cross-entropy loss
 - Mean squared error
 - Etc.

Putting things together:



(a part of GoogleNet)

- Arbitrary combinations of the basic building blocks
- Multiple loss functions – multi-target prediction, transfer learning, and more
- Given enough data, deeper architectures just keep improving
- Representation learning: **the networks learn increasingly more abstract representations of the data that are “disentangled,” i.e., amenable to linear separation.**

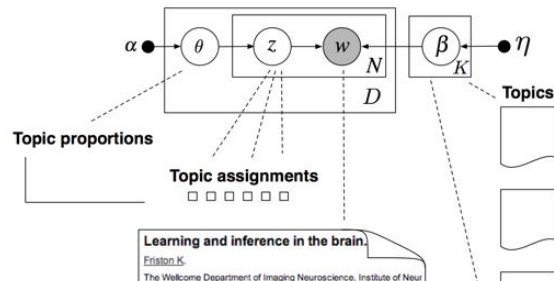
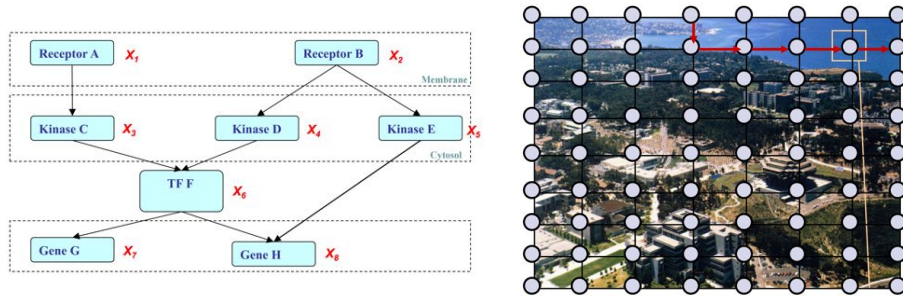
Outline

- Probabilistic Graphical Models: Basics
- An overview of the DL components
 - Historical remarks: early days of neural networks
 - Modern building blocks: units, layers, activations functions, loss functions, etc.
 - Reverse-mode automatic differentiation (aka backpropagation)
- Similarities and differences between GMs and NNs
 - Graphical models vs. computational graphs
 - Sigmoid Belief Networks as graphical models
 - Deep Belief Networks and Boltzmann Machines
- Combining DL methods and GMs
 - Using outputs of NNs as inputs to GMs
 - GMs with potential functions represented by NNs
 - NNs with structured outputs
- Bayesian Learning of NNs
 - Bayesian learning of NN parameters
 - Deep kernel learning

Graphical models vs. Deep nets

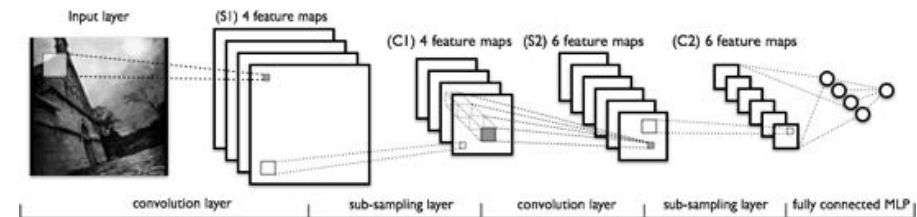
Graphical models

- Representation for encoding meaningful knowledge and the associated uncertainty in a graphical form



Deep neural networks

- Learn representations that facilitate computation and performance on the end-metric (intermediate representations are not guaranteed to be meaningful)



Graphical models vs. Deep nets

Graphical models

- Representation for encoding meaningful knowledge and the associated uncertainty in a graphical form
- Learning and inference are based on a rich toolbox of well-studied (structure-dependent) techniques (e.g., EM, message passing, VI, MCMC, etc.)
- Graphs represent models

Deep neural networks

- Learn representations that facilitate computation and performance on the end-metric (intermediate representations are not guaranteed to be meaningful)
- Learning is predominantly based on the gradient descent method (aka backpropagation); Inference is often trivial and done via a “forward pass”
- Graphs represent computation

Graphical models vs. Deep nets

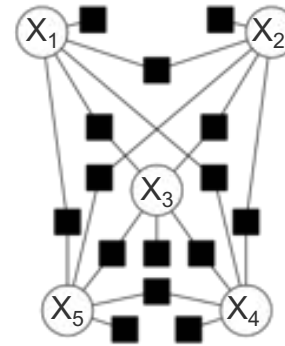
Graphical models

Utility of the graph

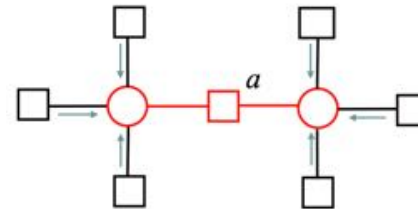
- A vehicle for synthesizing a global loss function from local structure
 - potential function, feature function, etc.
- A vehicle for designing sound and efficient inference algorithms
 - Sum-product, mean-field, etc.
- A vehicle to inspire approximation and penalization
 - Structured MF, Tree-approximation, etc.
- A vehicle for monitoring theoretical and empirical behavior and accuracy of inference

Utility of the loss function

- A major measure of quality of the learning algorithm and the model $\theta = \operatorname{argmax}_{\theta} P_{\theta}(V)$



$$\log P(X) = \sum_i \log \phi(x_i) + \sum_{i,j} \log \psi(x_i, x_j)$$



$$m_{i \rightarrow a}(x_i) = \prod_{c \in N(i) \setminus a} m_{c \rightarrow i}(x_i)$$

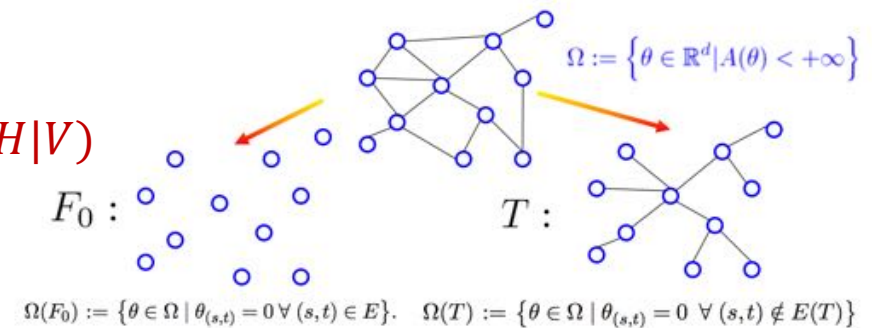
$$b_a(X_a) \propto f_a(X_a) \prod_{i \in N(a)} m_{i \rightarrow a}(x_i)$$

$$m_{a \rightarrow i}(x_i) = \sum_{X_a \setminus x_i} f_a(X_a) \prod_{j \in N(a) \setminus i} m_{j \rightarrow a}(x_j)$$

$$Q(H) \sim P(H|V)$$

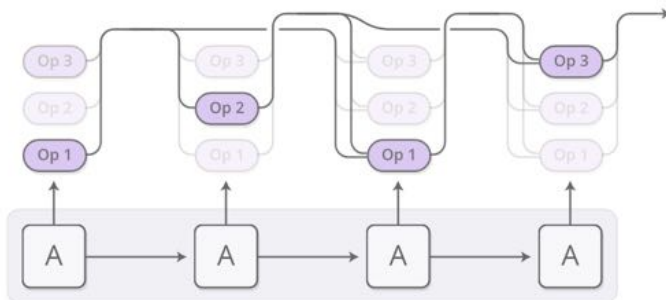
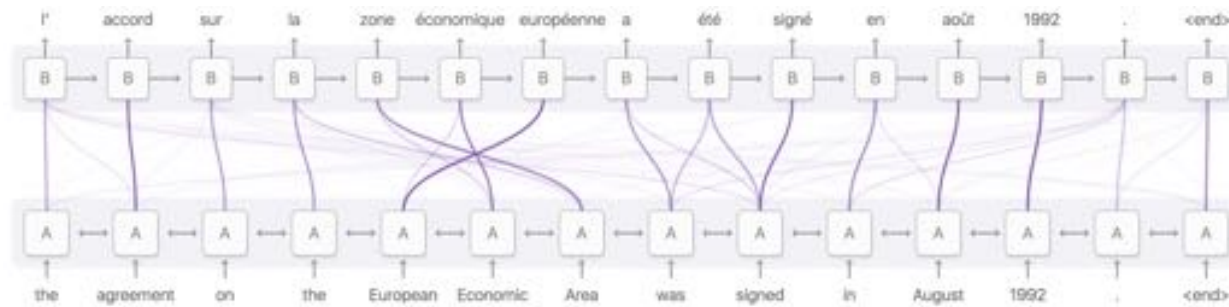
$F_0 :$

$T :$



$$\Omega(F_0) := \{\theta \in \Omega \mid \theta_{(s,t)} = 0 \forall (s,t) \in E\}. \quad \Omega(T) := \{\theta \in \Omega \mid \theta_{(s,t)} = 0 \forall (s,t) \notin E(T)\}$$

Graphical models vs. Deep nets



Deep neural networks

Utility of the network

- A vehicle to conceptually synthesize complex decision hypothesis
 - stage-wise projection and aggregation
- A vehicle for organizing computational operations
 - stage-wise update of latent states
- A vehicle for designing processing steps and computing modules
 - Layer-wise parallelization
- No obvious utility in evaluating DL inference algorithms

Utility of the Loss Function

- Global loss? Well it is complex and non-convex...

Graphical models vs. Deep nets

Graphical models

Utility of the graph

- A vehicle for synthesizing a global loss function from local structure
 - potential function, feature function, etc.
- A vehicle for designing sound and efficient inference algorithms
 - Sum-product, mean-field, etc.
- A vehicle to inspire approximation and penalization
 - Structured MF, Tree-approximation, etc.
- A vehicle for monitoring theoretical and empirical behavior and accuracy of inference

Utility of the loss function

- A major measure of quality of the learning algorithm and the model

Deep neural networks

Utility of the network

- A vehicle to conceptually synthesize complex decision hypothesis
 - stage-wise projection and aggregation
- A vehicle for organizing computational operations
 - stage-wise update of latent states
- A vehicle for designing processing steps and computing modules
 - Layer-wise parallelization
- No obvious utility in evaluating DL inference algorithms

Utility of the Loss Function

- Global loss? Well it is complex and non-convex...

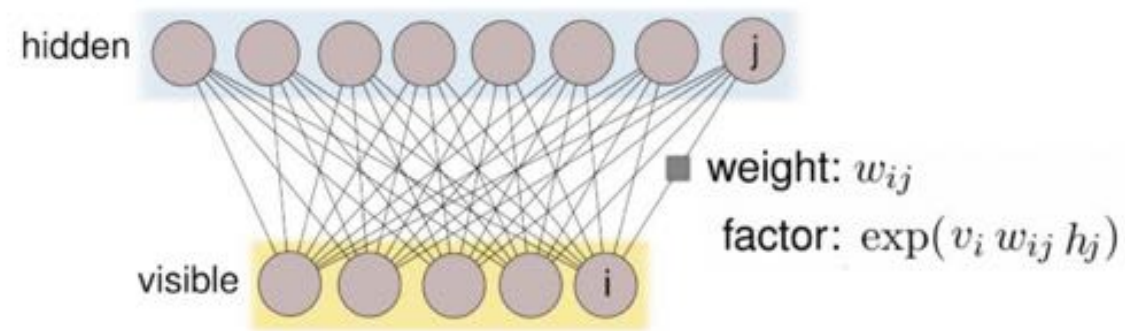
	DL	MM ? ML (e.g., GM)
Empirical goal:	e.g., classification, feature learning	e.g., latent variable inference, transfer learning
Structure:	Graphical	Graphical
Objective:	Something aggregated from local functions	Something aggregated from local functions
Vocabulary:	Neuron, activation function, ...	Variable, potential function, ...
Algorithm:	A single, unchallenged, inference algorithm – Backpropagation (BP)	A major focus of open research, many algorithms, and more to come
Evaluation:	On a black-box score – end performance	On almost every intermediate quantity
Implementation:	Many tricks	More or less standardized
Experiments:	Massive, real data (GT unknown)	Modest, often simulated data (GT known)

Graphical Models vs. Deep Nets

- So far:
 - Graphical models are representations of probability distributions
 - Neural networks are function approximators (with no probabilistic meaning)
- Some of the neural nets are in fact proper graphical models (i.e., units/neurons represent random variables):
 - Boltzmann machines (Hinton & Sejnowsky, 1983)
 - Restricted Boltzmann machines (Smolensky, 1986)
 - Learning and Inference in sigmoid belief networks (Neal, 1992)
 - Fast learning in deep belief networks (Hinton, Osindero, Teh, 2006)
 - Deep Boltzmann machines (Salakhutdinov and Hinton, 2009)
- Let's go through these models one-by-one

I: Restricted Boltzmann Machines

- RBM is a Markov random field represented with a bi-partite graph
- All nodes in one layer/part of the graph are connected to all in the other; no inter-layer connections



- Joint distribution:

$$P(v, h) = \frac{1}{Z} \exp \left\{ \sum_{i,j} w_{ij} v_i h_j + \sum_i b_i v_i + \sum_j c_j h_j \right\}$$

I: Restricted Boltzmann Machines

- Log-likelihood of a single data point (unobservables marginalized out):

$$\log L(v) = \log \sum_h \exp \left\{ \sum_{i,j} w_{ij} v_i h_i + \sum_i b_i v_i + \sum_j c_j h_j - \log(Z) \right\}$$

- Gradient of the log-likelihood w.r.t. the model parameters:

$$\frac{\partial}{\partial w_{ij}} \log L(v) = \sum_h P(h|v) \frac{\partial}{\partial w_{ij}} P(v, h) - \sum_{v,h} P(v, h) \frac{\partial}{\partial w_{ij}} P(v, h)$$

- where we have averaging over the posterior and over the joint.

I: Restricted Boltzmann Machines

- Gradient of the log-likelihood w.r.t. the parameters (alternative form):

$$\frac{\partial}{\partial w_{ij}} \log L(v) = \mathbb{E}_{P(h|v)} \left[\frac{\partial}{\partial w_{ij}} P(v, h) \right] - \mathbb{E}_{P(v, h)} \left[\frac{\partial}{\partial w_{ij}} P(v, h) \right]$$

- Both expectations can be approximated via sampling
- Sampling from **the posterior** is exact (RBM factorizes over h given v)
- Sampling from **the joint** is done via MCMC (e.g., Gibbs sampling)
- In the neural networks literature:
 - computing **the first term** is called **the clamped / wake / positive phase** (the network is “awake” since it conditions on the visible variables)
 - Computing the second term is called **the unclamped / sleep / free / negative phase** (the network is “asleep” since it samples the visible variables from the joint; metaphorically, it is “dreaming” the visible inputs)

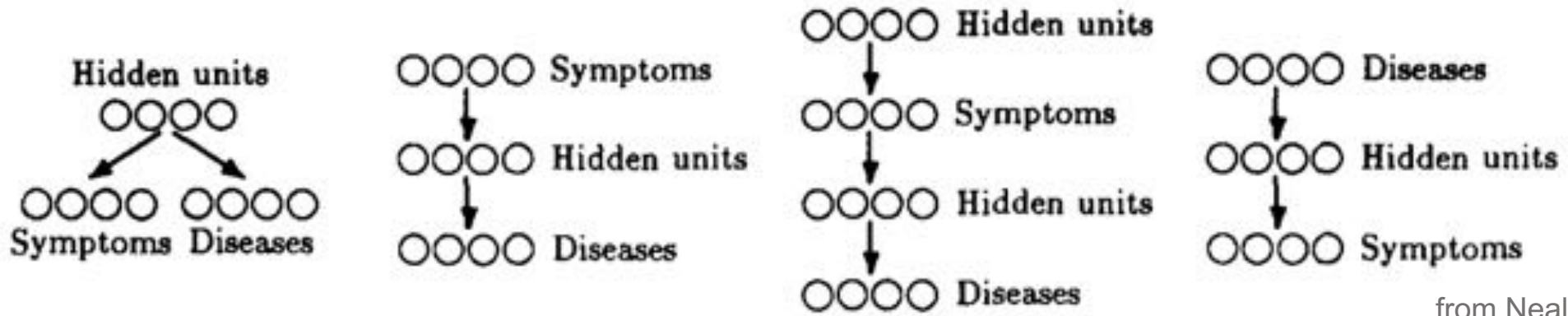
I: Restricted Boltzmann Machines

- Gradient of the log-likelihood w.r.t. the parameters (alternative form):

$$\frac{\partial}{\partial w_{ij}} \log L(v) = \mathbb{E}_{P(h|v)} \left[\frac{\partial}{\partial w_{ij}} P(v, h) \right] - \mathbb{E}_{P(v, h)} \left[\frac{\partial}{\partial w_{ij}} P(v, h) \right]$$

- Learning is done by optimizing the log-likelihood of the model for a given data via stochastic gradient descent (SGD)
- Estimation of **the second term (the negative phase)** heavily relies on the mixing properties of the Markov chain
- This often causes slow convergence and requires extra computation

II: Sigmoid Belief Networks



from Neal, 1992

- Sigmoid belief nets are simply Bayesian networks over binary variables with conditional probabilities represented by sigmoid functions:

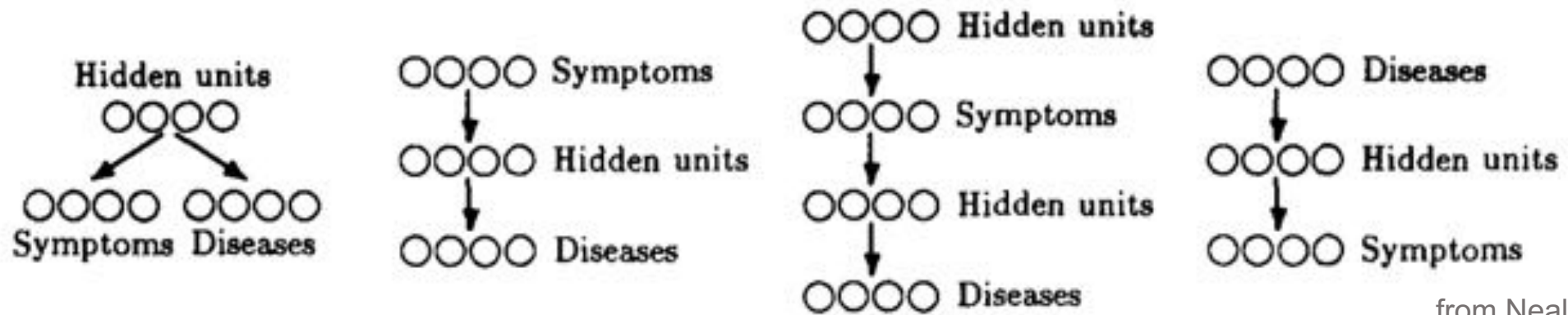
$$P(x_i | \pi(x_i)) = \sigma \left(x_i \sum_{x_j \in \pi(x_i)} w_{ij} x_j \right)$$

- Bayesian networks exhibit a phenomenon called “explain away effect”



If A correlates with C, then the chance of B correlating with C decreases. \Rightarrow A and B become correlated given C.

II: Sigmoid Belief Networks

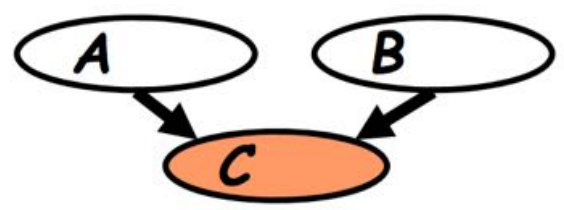


from Neal, 1992

- Sigmoid belief nets are simply Bayesian networks over binary variables with conditional probabilities represented by sigmoid functions:

$$P(x_i | \pi(x_i)) = \sigma \left(x_i \sum_{x_j \in \pi(x_i)} w_{ij} x_j \right)$$

- Bayesian networks exhibit a phenomenon called “explain away effect”



Note: Due to the “explain away effect,” when we condition on the visible layer in belief networks, hidden variables all become dependent.

Sigmoid Belief Networks: Learning and Inference

- Neal proposed Monte Carlo methods for learning and inference (Neal, 1992):

Approximated with Gibbs sampling

- Conditional distributions:

$$P(S_i = x \mid S_j = s_j : j \neq i) \propto \sigma\left(x^* \sum_{j < i} s_j w_{ij}\right) \prod_{j > i} \sigma\left(s_j^* \left(x w_{ji} + \sum_{k < j, k \neq i} s_k w_{jk}\right)\right)$$

- No negative phase as in RBM!
- Convergence is very slow, especially for large belief nets, due to the intricate “explain-away” effects...

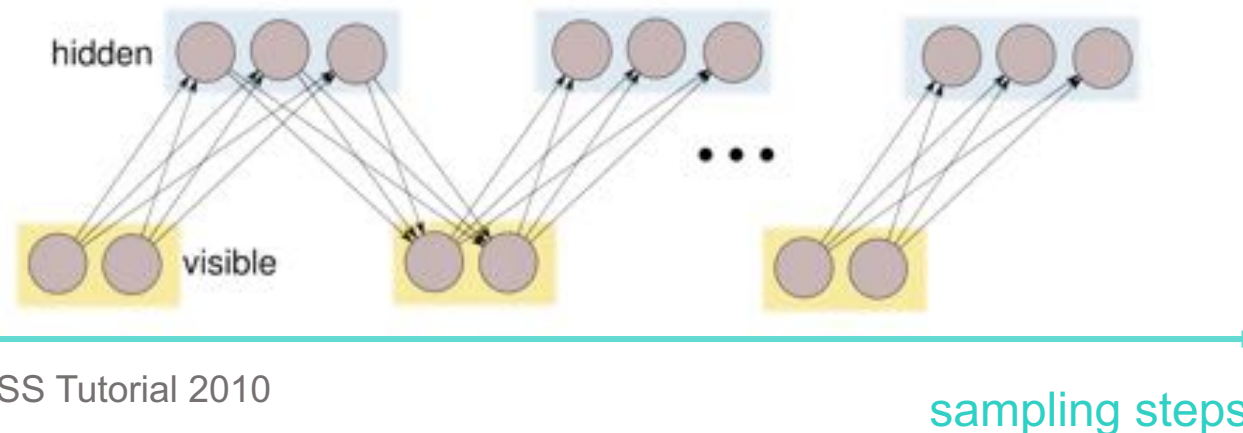
$$\begin{aligned} \frac{\partial L}{\partial w_{ij}} &= \sum_{\tilde{v} \in \mathcal{T}} \frac{1}{P(\tilde{V} = \tilde{v})} \frac{\partial P(\tilde{V} = \tilde{v})}{\partial w_{ij}} && \text{log derivative} \\ &= \sum_{\tilde{v} \in \mathcal{T}} \frac{1}{P(\tilde{V} = \tilde{v})} \sum_{\tilde{h}} \frac{\partial P(\tilde{S} = \langle \tilde{h}, \tilde{v} \rangle)}{\partial w_{ij}} && \text{prob. of the visibles via marginalization} \\ &= \sum_{\tilde{v} \in \mathcal{T}} \sum_{\tilde{h}} P(\tilde{S} = \langle \tilde{h}, \tilde{v} \rangle \mid \tilde{V} = \tilde{v}) \frac{1}{P(\tilde{S} = \langle \tilde{h}, \tilde{v} \rangle)} \frac{\partial P(\tilde{S} = \langle \tilde{h}, \tilde{v} \rangle)}{\partial w_{ij}} && \text{Bayes rule + rearrange sums} \\ &= \sum_{\tilde{v} \in \mathcal{T}} \sum_{\tilde{s}} P(\tilde{S} = \tilde{s} \mid \tilde{V} = \tilde{v}) \frac{1}{P(\tilde{S} = \tilde{s})} \frac{\partial P(\tilde{S} = \tilde{s})}{\partial w_{ij}} \\ &= \sum_{\tilde{v} \in \mathcal{T}} \sum_{\tilde{s}} P(\tilde{S} = \tilde{s} \mid \tilde{V} = \tilde{v}) \frac{1}{\sigma\left(s_i^* \sum_{k < i} s_k w_{ik}\right)} \frac{\partial \sigma\left(s_i^* \sum_{k < i} s_k w_{ik}\right)}{\partial w_{ij}} && \text{Plug-in the actual sigmoid form of the conditional prob.} \\ &= \sum_{\tilde{v} \in \mathcal{T}} \sum_{\tilde{s}} P(\tilde{S} = \tilde{s} \mid \tilde{V} = \tilde{v}) s_i^* s_j \sigma\left(-s_i^* \sum_{k < i} s_k w_{ik}\right). \end{aligned}$$

RBMMs are infinite belief networks

- Recall the expression for the gradient of the log likelihood for RBM:

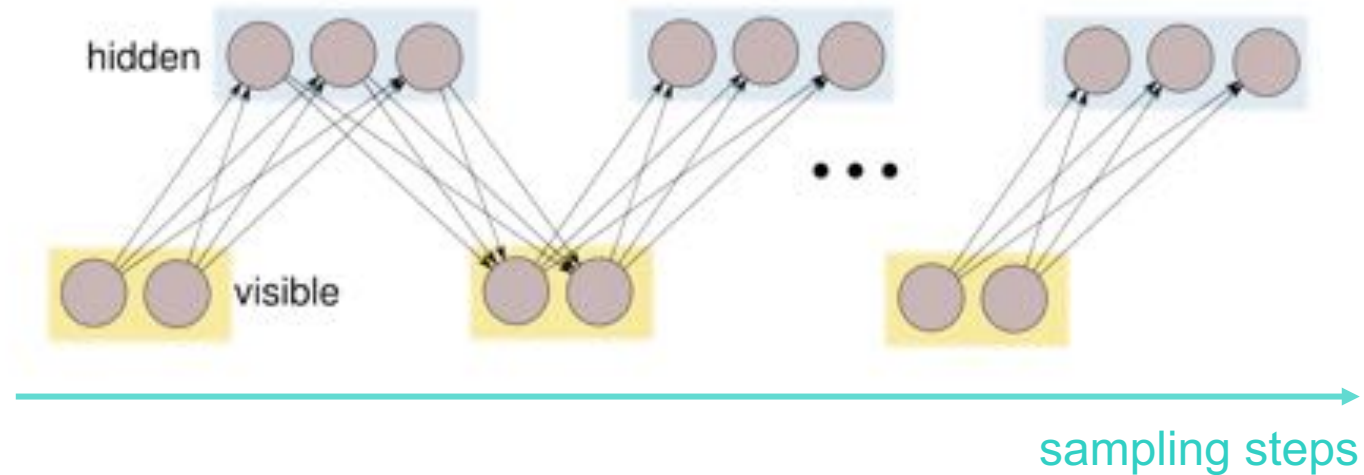
$$\frac{\partial}{\partial w_{ij}} \log L(v) = \mathbb{E}_{P(h|v)} \left[\frac{\partial}{\partial w_{ij}} P(v, h) \right] - \mathbb{E}_{P(v, h)} \left[\frac{\partial}{\partial w_{ij}} P(v, h) \right]$$

- To make a gradient update of the model parameters, we need compute the expectations via sampling.
 - We can sample exactly from the posterior in the first term
 - We run block Gibbs sampling to approximately sample from the joint distribution



RBM's are infinite belief networks

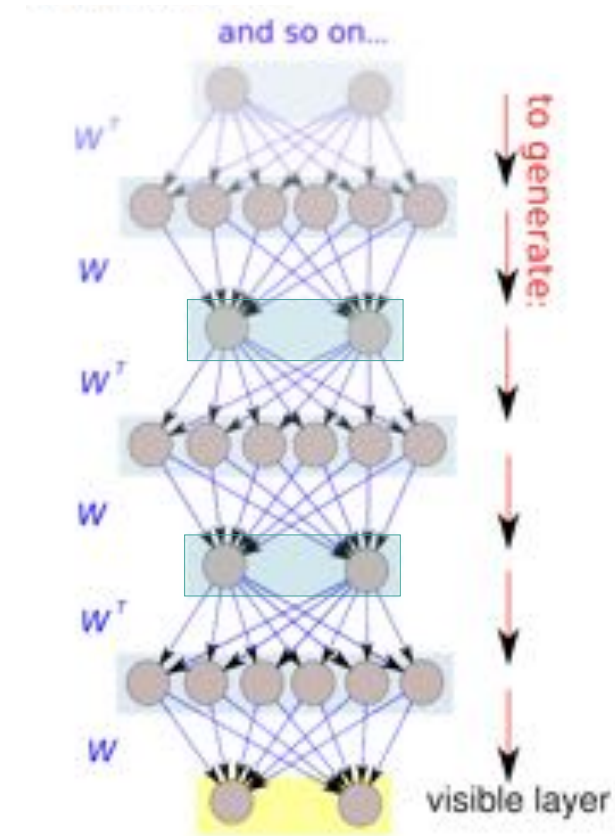
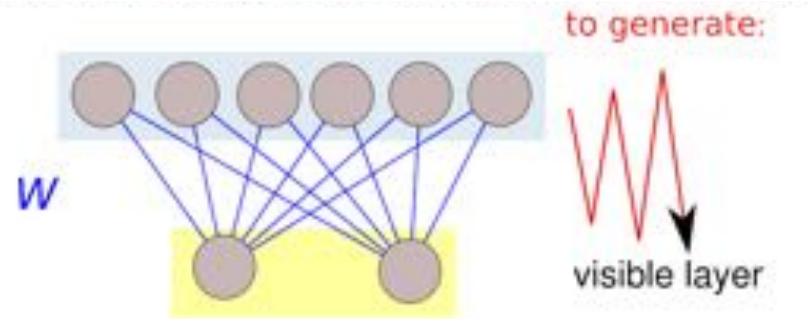
- Gibbs sampling: alternate between sampling hidden and visible variables



- Conditional distributions $P(v|h)$ and $P(h|v)$ are represented by sigmoids
- Thus, we can think of Gibbs sampling from the joint distribution represented by an RBM as a top-down propagation in an infinitely deep sigmoid belief network!

RBMMs are infinite belief networks

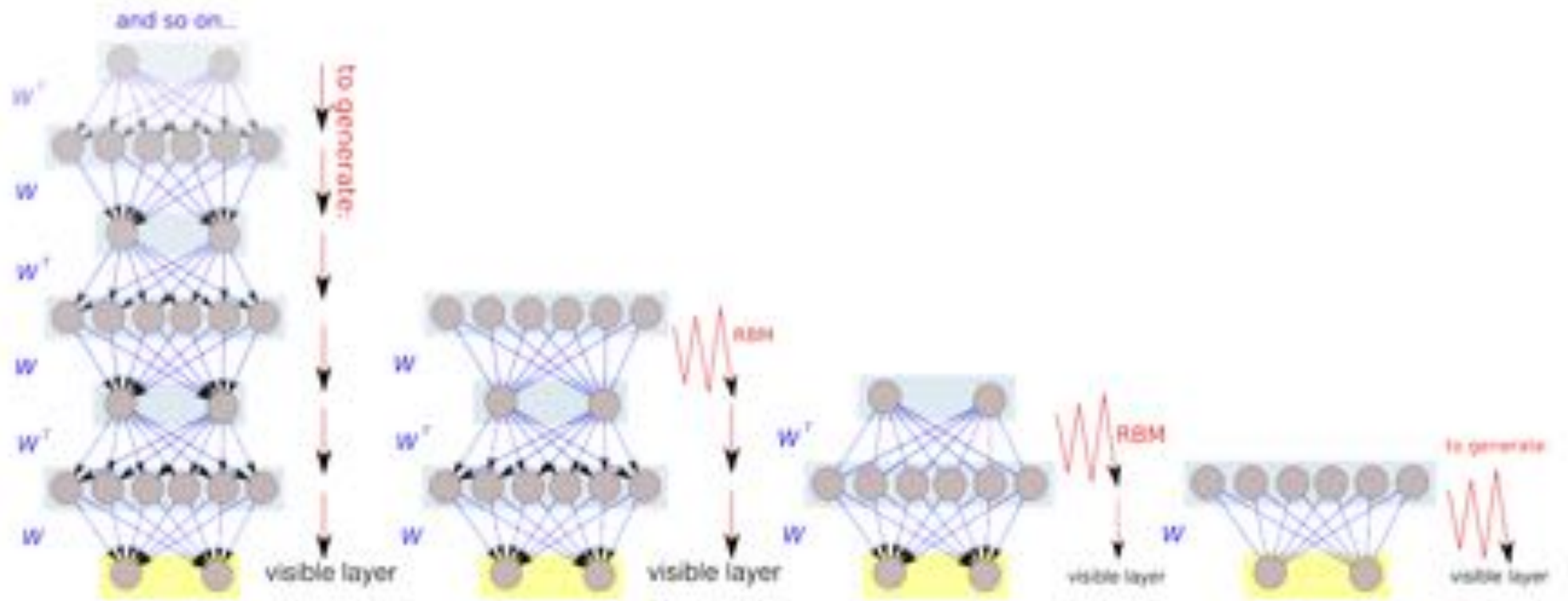
- RBMMs are equivalent to infinitely deep belief networks



- Sampling from this is the same as sampling from the network on the right

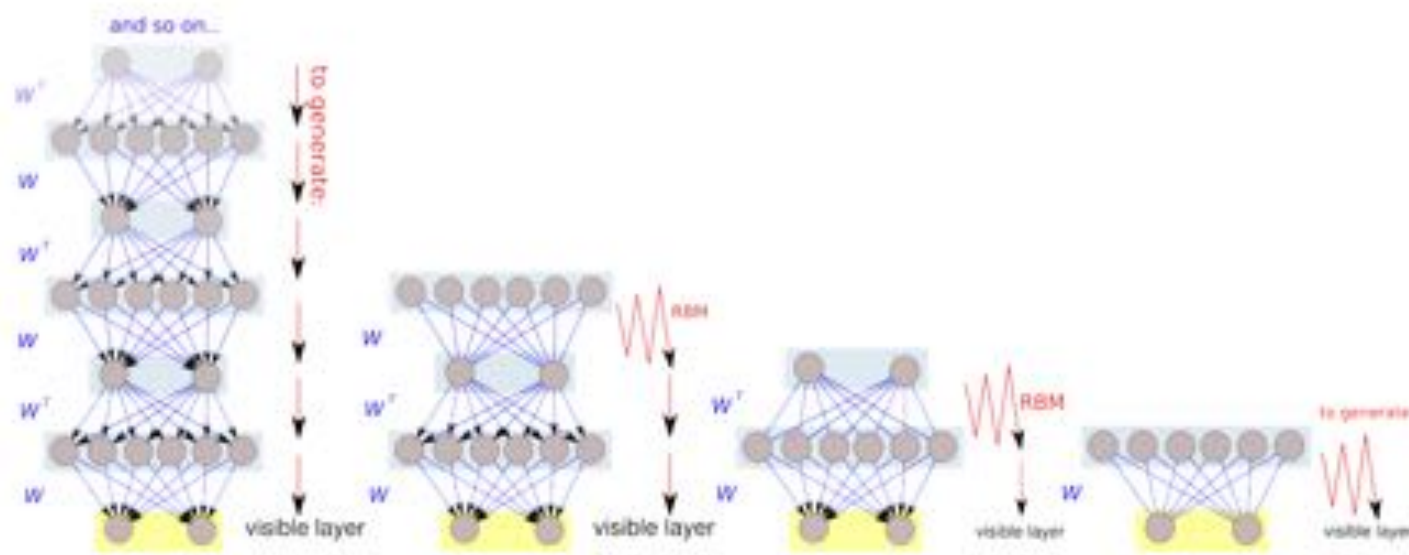
RBMMs are infinite belief networks

- RBMs are equivalent to infinitely deep belief networks



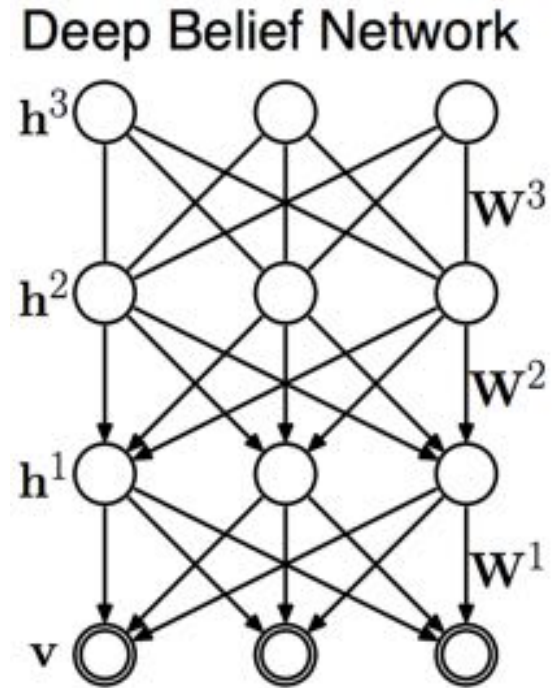
RBMMs are infinite belief networks

- RBMs are equivalent to infinitely deep belief networks



- When we train an RBM, we are really training an infinitely deep belief net!
- It is just that the weights of all layers are tied.
- If the weights are “untied” to some extent, we get a Deep Belief Network.

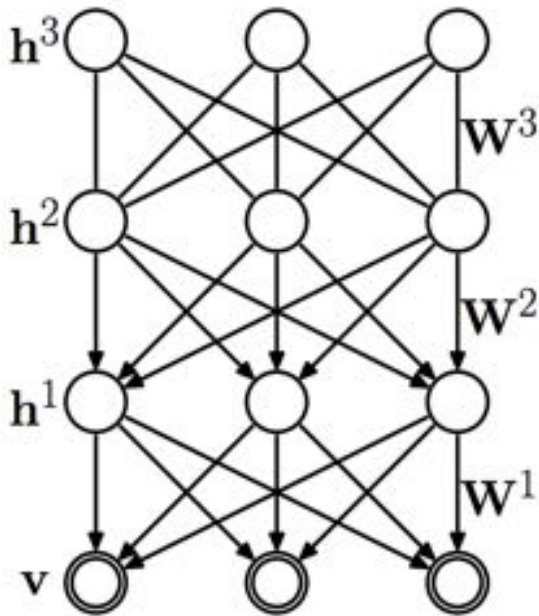
III: Deep Belief Nets



- DBNs are hybrid graphical models (chain graphs):
 - Exact inference in DBNs is problematic due to explaining away effect
 - Training: greedy pre-training + ad-hoc fine-tuning; no proper joint training
 - Approximate inference is feed-forward

Deep Belief Networks

Deep Belief Network



- DBNs represent a joint probability distribution

$$P(v, h^1, h^2, h^3) = P(h^2, h^3)P(h^1|h^2)P(v|h^1)$$

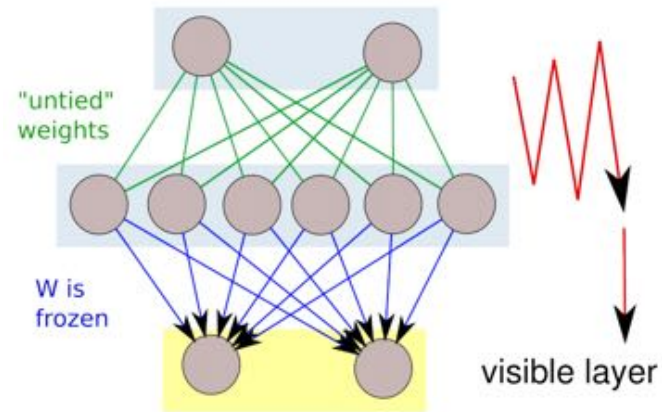
- Note that $P(h^2, h^3)$ is an RBM and the conditionals $P(h^1|h^2)$ and $P(v|h^1)$ are represented in the sigmoid form
- The model is trained by optimizing the log likelihood for a given data $\log P(v)$

Challenges:

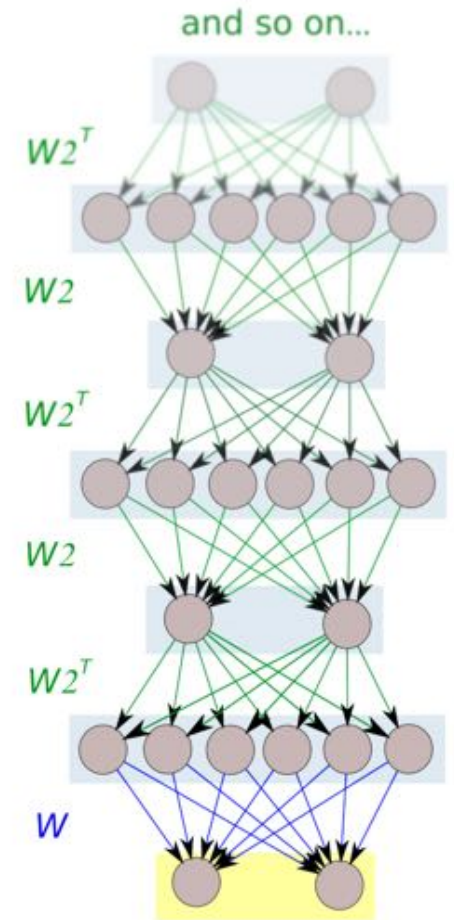
- Exact inference in DBNs is problematic due to explain away effect
- Training is done in two stages:
 - greedy pre-training + ad-hoc fine-tuning; no proper joint training
- Approximate inference is feed-forward (bottom-up)

DBN: Layer-wise pre-training

- Pre-train and freeze the 1st RBM
- Stack another RBM on top and train it

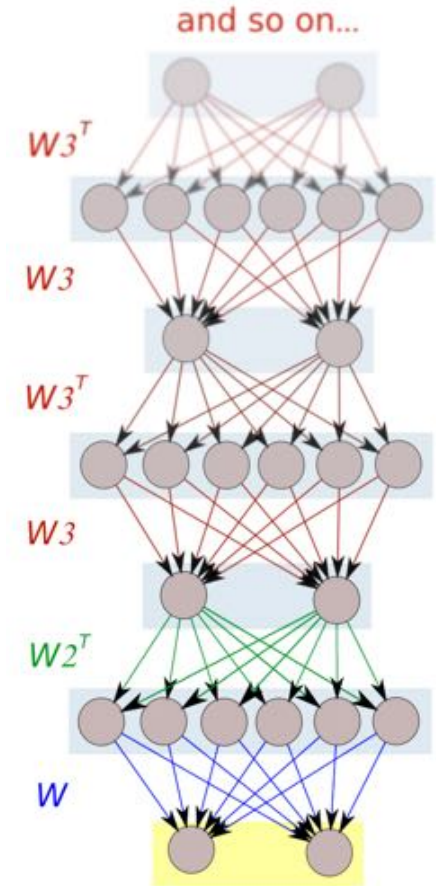
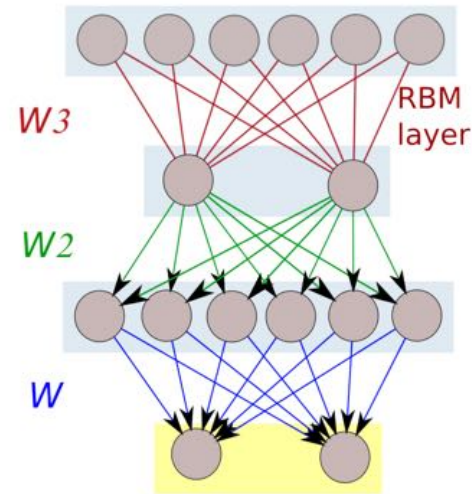


- The weights weights 2+ layers remain tied
- We repeat this procedure: pre-train and untie the weights layer-by-layer...



DBN: Layer-wise pre-training

- We repeat this procedure: pre-train and untie the weights layer-by-layer:
- The weights of 3+ layers remain tied



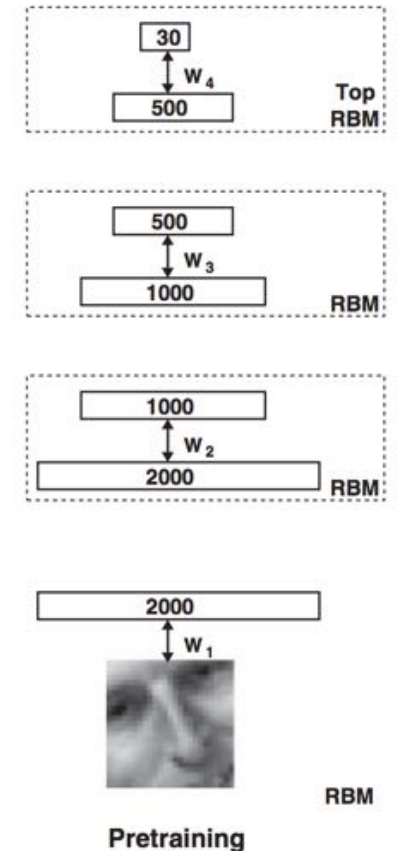
- and so forth
- *From the optimization perspective, this procedure loosely corresponds to an approximate block-coordinate ascent on the log-likelihood*

DBN: Fine-tuning

- Pre-training is quite ad-hoc and is unlikely to lead to a good probabilistic model *per se*
- However, the layers of representations could perhaps be useful for some other downstream tasks!
- We can further “fine-tune” a pre-trained DBN for some other task

Setting A: Unsupervised learning (DBN \rightarrow autoencoder)

1. **Pre-train a stack of RBMs in a greedy layer-wise fashion**
2. “Unroll” the RBMs to create an autoencoder
3. Fine-tune the parameters by optimizing the reconstruction error

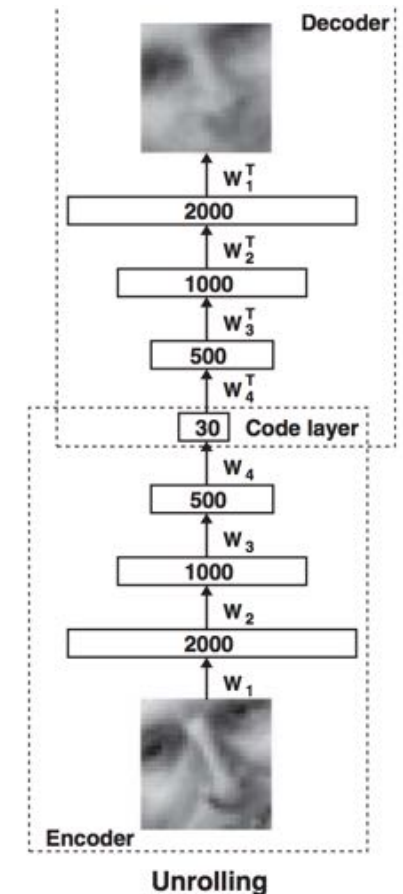


DBN: Fine-tuning

- Pre-training is quite ad-hoc and is unlikely to lead to a good probabilistic model *per se*
- However, the layers of representations could perhaps be useful for some other downstream tasks!
- We can further “fine-tune” a pre-trained DBN for some other task

Setting A: Unsupervised learning (DBN → autoencoder)

1. Pre-train a stack of RBMs in a greedy layer-wise fashion
2. **“Unroll” the RBMs to create an autoencoder**
3. Fine-tune the parameters by optimizing the reconstruction error

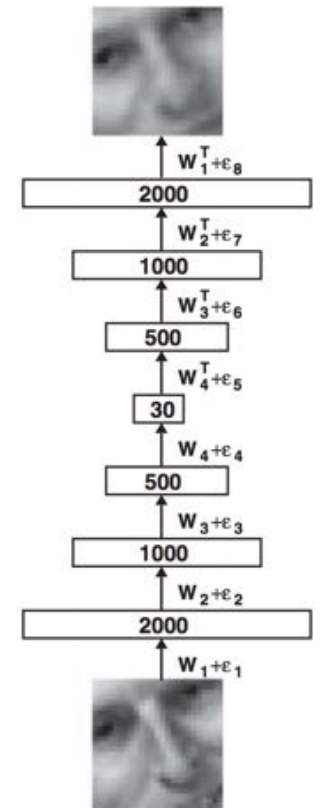


DBN: Fine-tuning

- Pre-training is quite ad-hoc and is unlikely to lead to a good probabilistic model *per se*
- However, the layers of representations could perhaps be useful for some other downstream tasks!
- We can further “fine-tune” a pre-trained DBN for some other task

Setting A: Unsupervised learning (DBN \rightarrow autoencoder)

1. Pre-train a stack of RBMs in a greedy layer-wise fashion
2. “Unroll” the RBMs to create an autoencoder
3. **Fine-tune the parameters by optimizing the reconstruction error**



Fine-tuning

DBN: Fine-tuning

- Pre-training is quite ad-hoc and is unlikely to lead to a good probabilistic model *per se*
- However, the layers of representations could perhaps be useful for some other downstream tasks!
- We can further “fine-tune” a pre-trained DBN for some other task

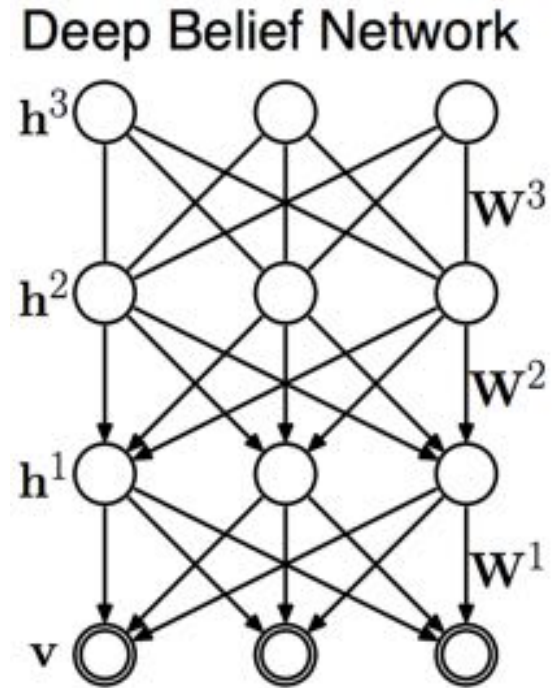
Setting B: Supervised learning (DBN → classifier)

1. Pre-train a stack of RBMs in a greedy layer-wise fashion
2. “Unroll” the RBMs to create a feedforward classifier
3. Fine-tune the parameters by optimizing the reconstruction error

Some intuitions about how pre-training works:

Erhan et al.: Why Does Unsupervised Pre-training Help Deep Learning? JMLR, 2010

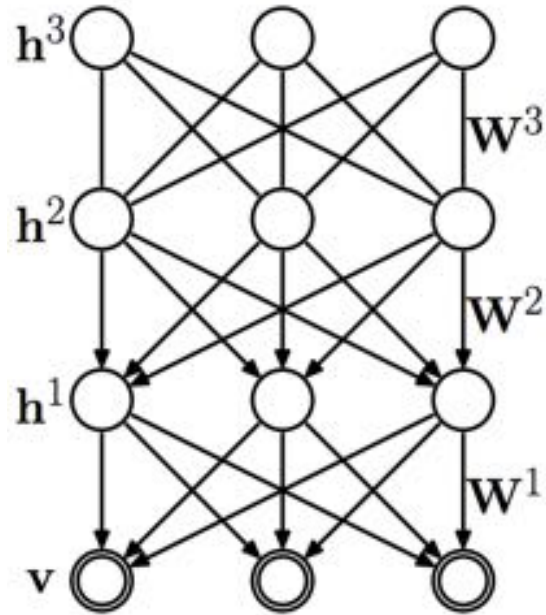
Deep Belief Nets and Boltzmann Machines



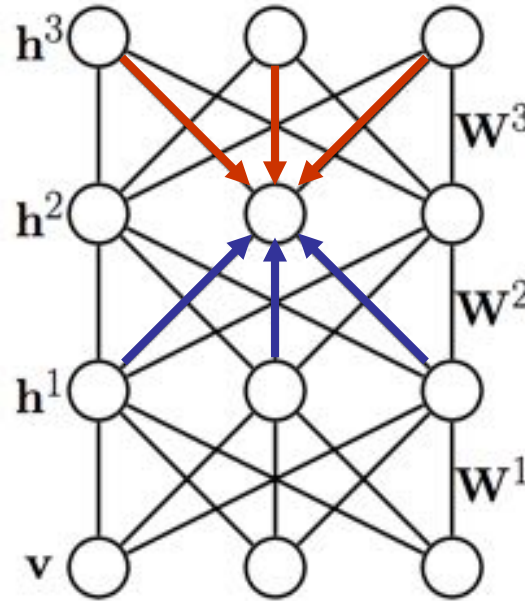
- DBNs are hybrid graphical models (chain graphs):
 - Inference in DBNs is problematic due to explaining away effect
 - Training: greedy pre-training + ad-hoc fine-tuning; no proper joint training
 - Approximate inference is feed-forward

Deep Belief Nets and Boltzmann Machines

Deep Belief Network



Deep Boltzmann Machine



- DBMs are fully un-directed models (Markov random fields):
 - Can be trained similarly as RBMs via MCMC (Hinton & Sejnowski, 1983)
 - Use a variational approximation of the data distribution for faster training (Salakhutdinov & Hinton, 2009)
 - Similarly, can be used to initialize other networks for downstream tasks

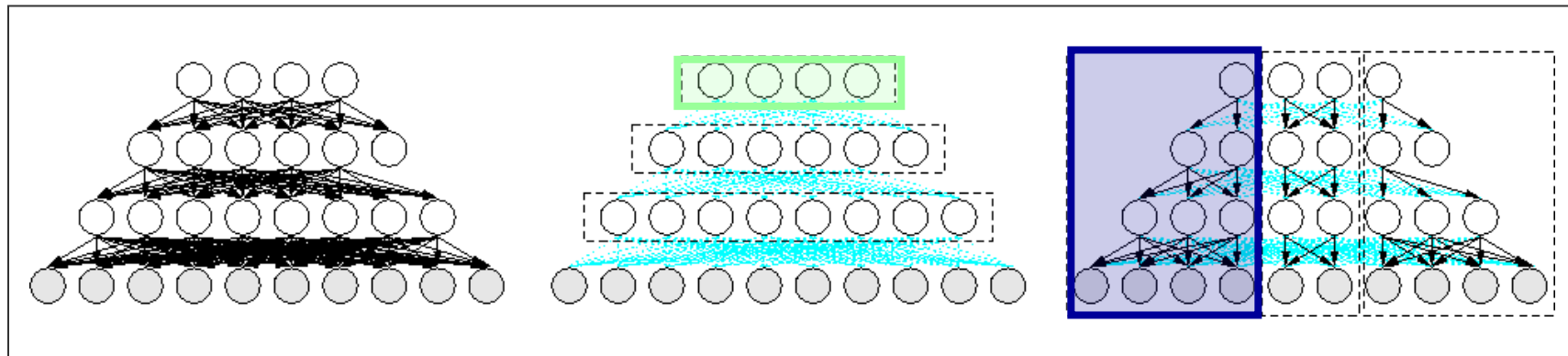
Graphical models vs. Deep networks

- A few critical points to note about all these models:
 - The primary goal of deep generative models is to represent the distribution of the observable variables. Adding layers of hidden variables allows to represent increasingly more complex distributions.
 - Hidden variables are secondary (auxiliary) elements used to facilitate learning of complex dependencies between the observables.
 - Training of the model is ad-hoc, but what matters is the quality of learned hidden representations.
 - Representations are judged by their usefulness on a downstream task (the probabilistic meaning of the model is often discarded at the end).
- In contrast, classical graphical models are often concerned with the correctness of learning and inference of all variables

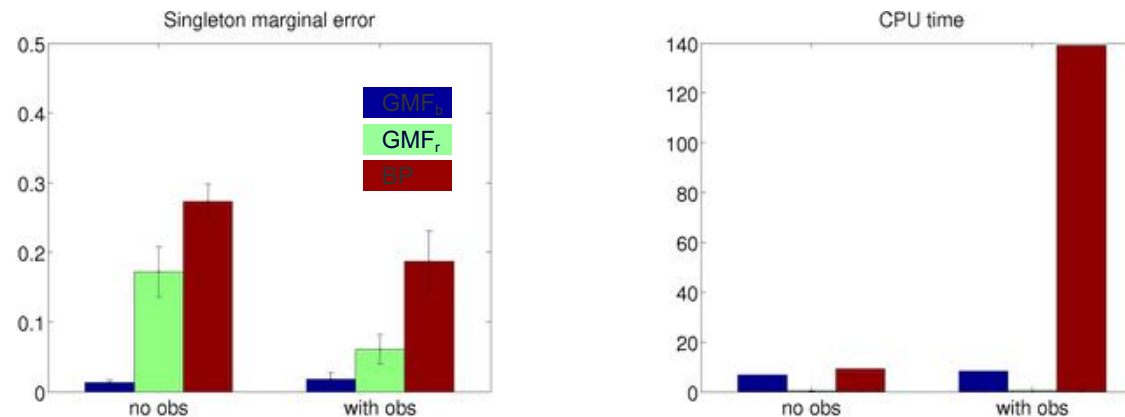
An old study of belief networks from the GM standpoint

[Xing, Russell, Jordan, UAI 2003]

Mean-field partitions of a sigmoid belief network for subsequent GMF inference



Study focused on only inference/learning accuracy, speed, and partition



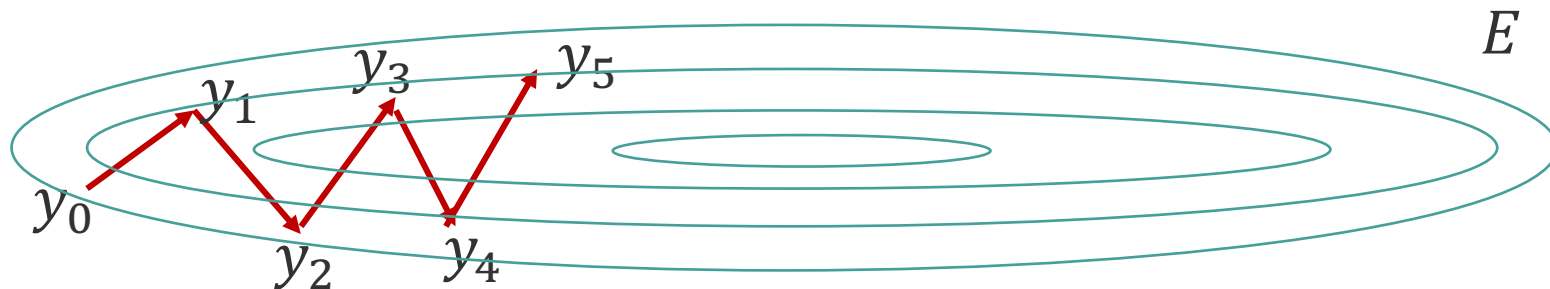
“Optimize” how to optimize via truncation & re-opt

- Energy-based modeling of the structured output (CRF)

$$\mathbf{y}^*(\mathbf{x}; \mathbf{w}) := \arg \min_{\mathbf{y}} E(\mathbf{y}, \mathbf{x}; \mathbf{w})$$

- Unroll the optimization algorithm for a fixed number of steps (Domke, 2012)

$$\mathbf{y}^*(\mathbf{x}; \mathbf{w}) = \underset{\mathbf{y}}{\text{opt-alg}} E(\mathbf{y}, \mathbf{x}; \mathbf{w})$$



We can backprop through the optimization steps since they are just a sequence of computations

Relevant recent paper:
Anrychowicz et al.: *Learning to learn by gradient descent by gradient descent*. 2016.

Dealing with structured prediction

- Energy-based modeling of the structured output (CRF)

$$\mathbf{y}^*(\mathbf{x}; \mathbf{w}) := \arg \min_{\mathbf{y}} E(\mathbf{y}, \mathbf{x}; \mathbf{w})$$

- Unroll the optimization algorithm for a fixed number of steps ([Domke, 2012](#))

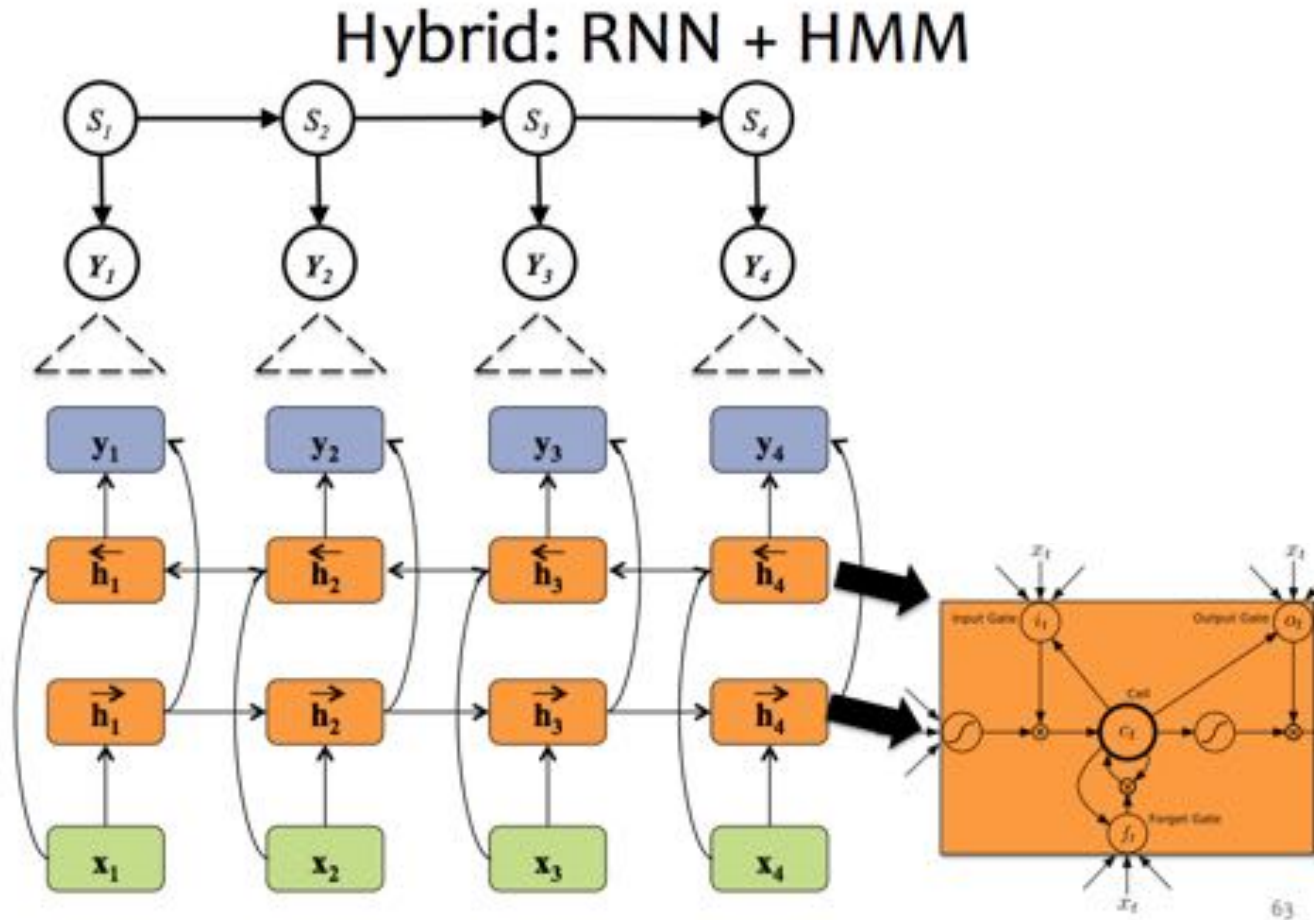
$$\mathbf{y}^*(\mathbf{x}; \mathbf{w}) = \text{opt-alg}_{\mathbf{y}} E(\mathbf{y}, \mathbf{x}; \mathbf{w})$$

- We can think of \mathbf{y}^* as some non-linear differentiable function of the inputs and weights \rightarrow impose some loss and optimize it as any other standard computation graph using backprop!
- Similarly, message passing based inference algorithms can be truncated and converted into computational graphs ([Domke, 2011](#); [Stoyanov et al., 2011](#))

Outline

- Probabilistic Graphical Models: Basics
- An overview of DL components
 - Historical remarks: early days of neural networks
 - Modern building blocks: units, layers, activations functions, loss functions, etc.
 - Reverse-mode automatic differentiation (aka backpropagation)
- Similarities and differences between GMs and NNs
 - Graphical models vs. computational graphs
 - Sigmoid Belief Networks as graphical models
 - Deep Belief Networks and Boltzmann Machines
- Combining DL methods and GMs
 - Using outputs of NNs as inputs to GMs
 - GMs with potential functions represented by NNs
 - NNs with structured outputs
- Bayesian Learning of NNs
 - Bayesian learning of NN parameters
 - Deep kernel learning

Combining sequential NNs and GMs



Combining sequential NNs and GMs

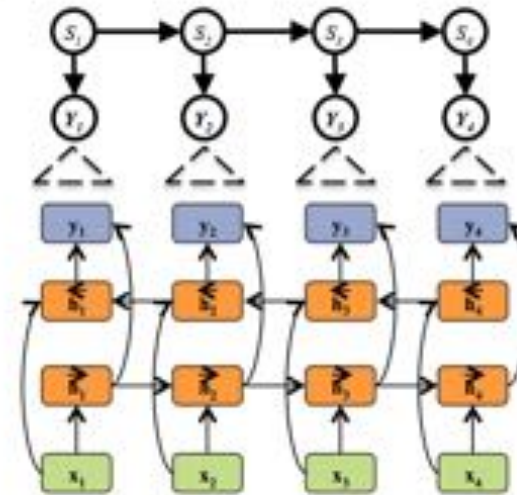
Hybrid: RNN + HMM

(Graves et al., 2013)

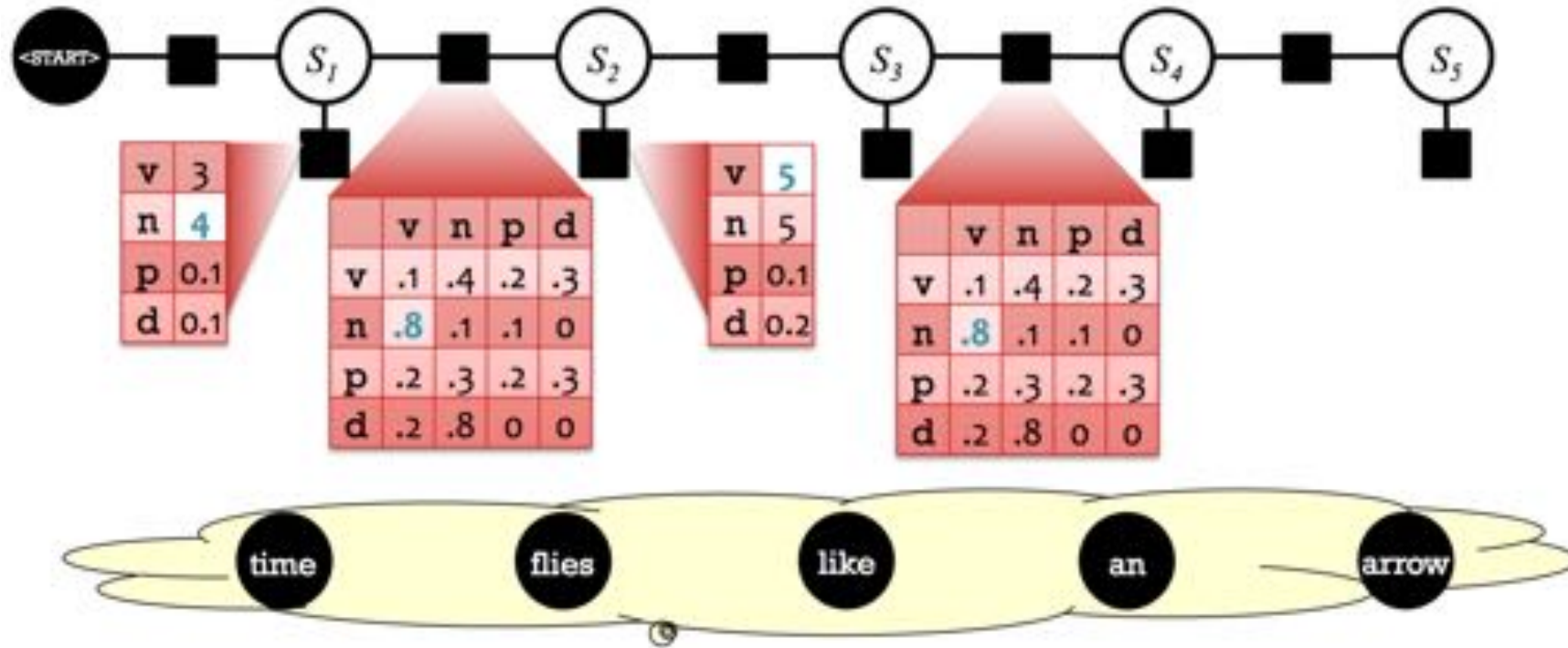


The model, inference, and learning can be **analogous** to our NN + HMM hybrid

- **Objective:** log-likelihood
- **Model:** HMM/Gaussian emissions
- **Inference:** forward-backward algorithm
- **Learning:** SGD with gradient by backpropagation



Hybrid NNs + conditional GMs

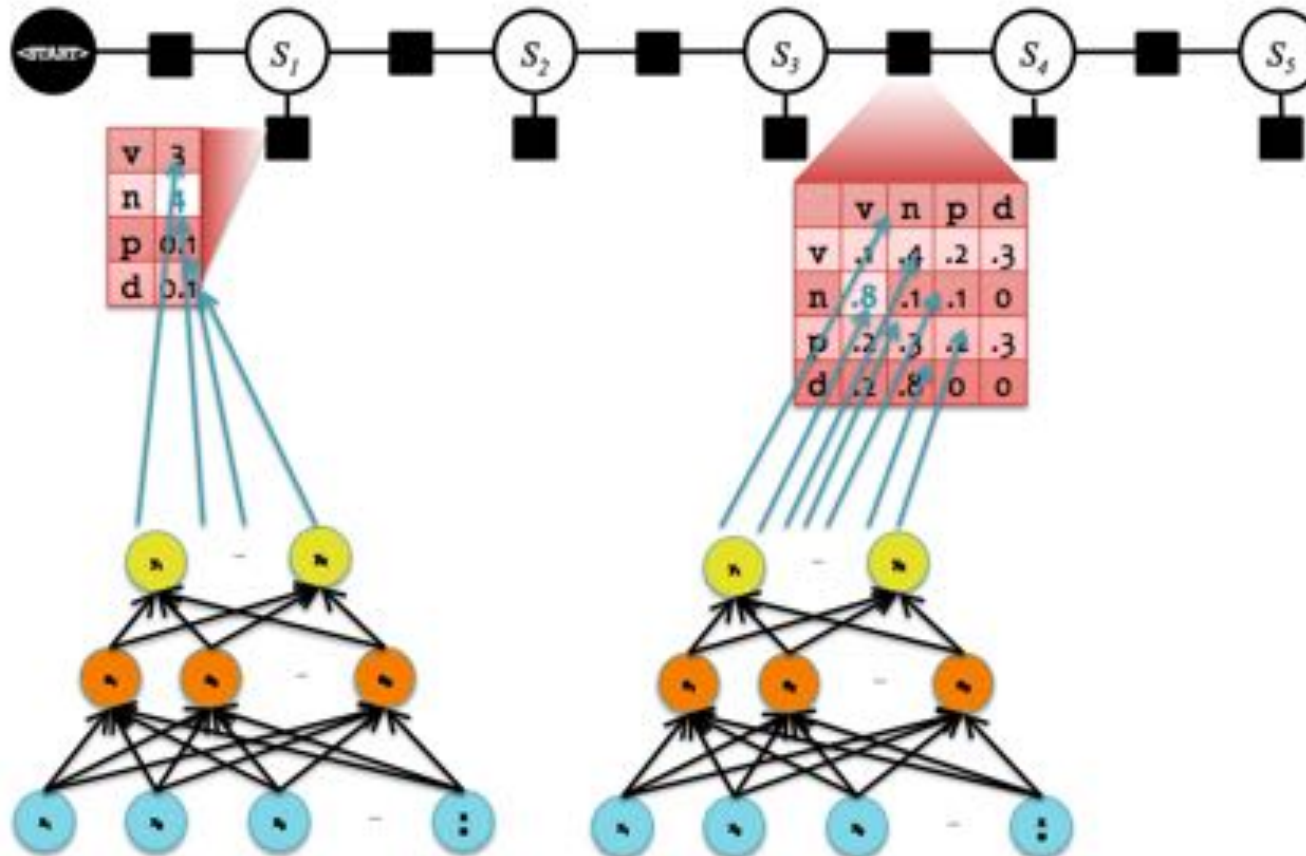


- In a standard CRF, each of the factor cells is a parameter.
- In a hybrid model, these values are computed by a neural network.

Hybrid NNs + conditional GMs

Hybrid: Neural Net + CRF

Forward computation



Hybrid NNs + conditional GMs



Hybrid: CNN + CRF

“NN + SLL”

- Model: Convolutional Neural Network (CNN) with **linear-chain CRF**
- Training objective: maximize **sentence-level likelihood (SLL)**

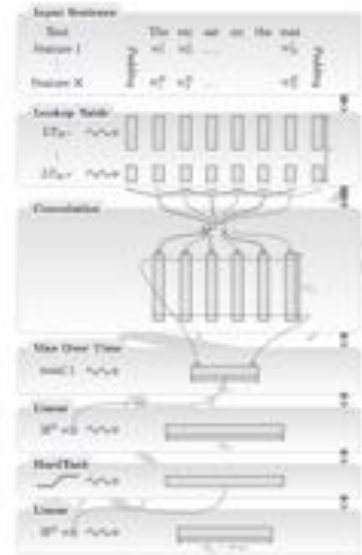
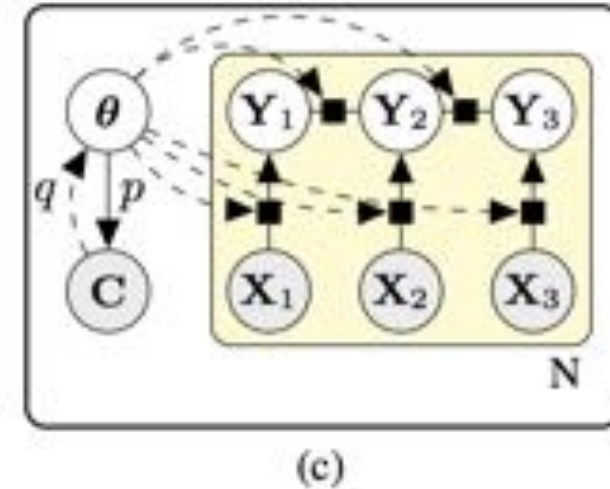
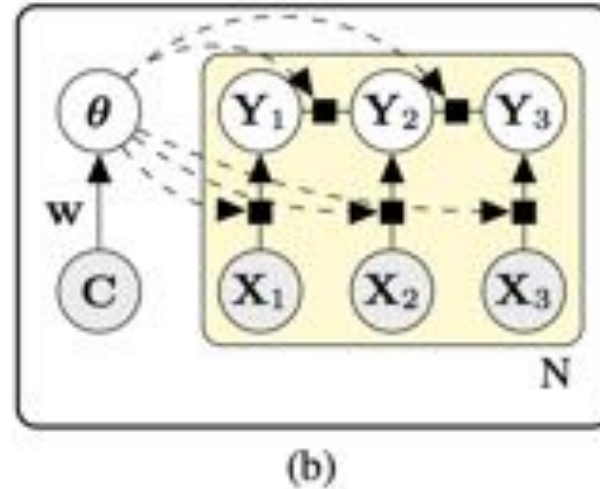
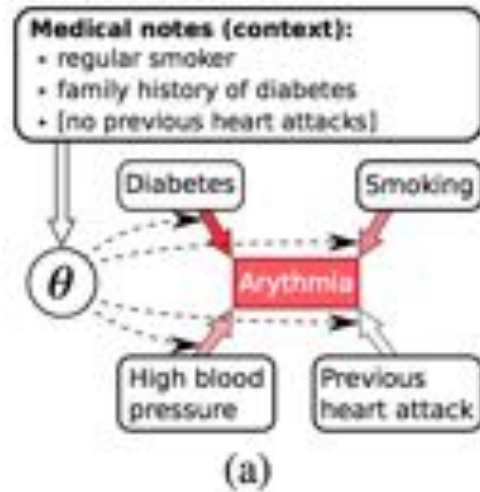


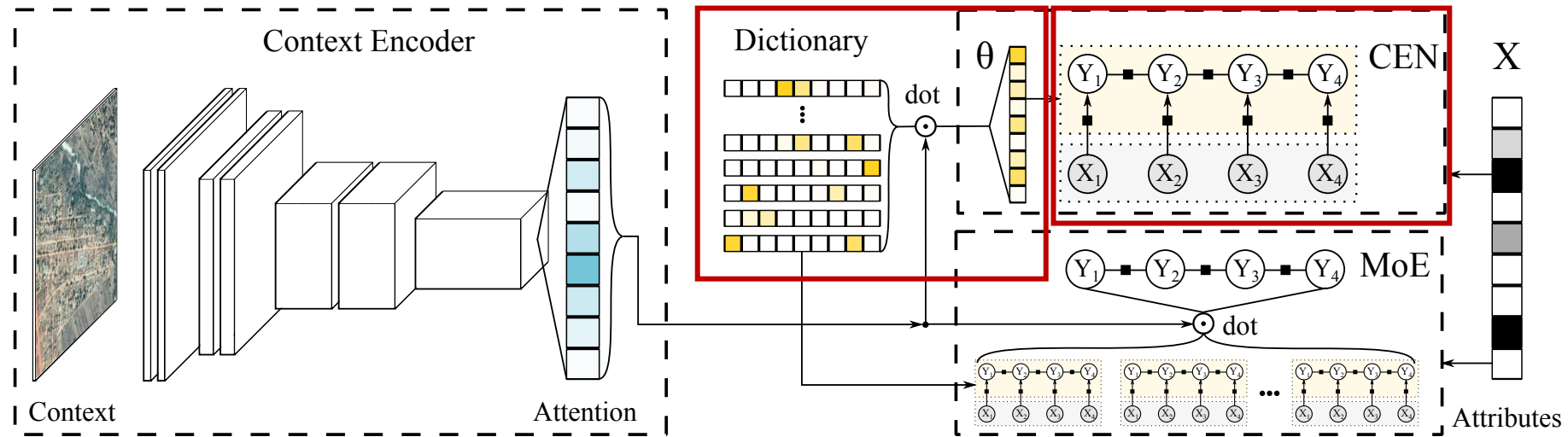
Figure from (Collobert & Weston, 2011)

Using GMs as Prediction Explanations



- **Idea:** Use deep neural nets to *generate parameters* of a graphical model for a given context (e.g., specific instance or case)
- Produced GMs are used to make the final prediction
- GMs are built on top of interpretable variables (not deep embeddings!) and can be used as *contextual explanations* for each prediction

Using GMs as Prediction Explanations



A practical implementation:

- Maintain a (sparse) dictionary of GM parameters
- Process complex inputs (images, text, time series, etc.) using deep nets; use soft attention to either *select* or *combine* models from the dictionary
- Use constructed GMs (e.g., CRFs) to make predictions
- Inspect GMs to understand the reasoning behind predictions

Outline

- An overview of the DL components
 - Historical remarks: early days of neural networks
 - Modern building blocks: units, layers, activations functions, loss functions, etc.
 - Reverse-mode automatic differentiation (aka backpropagation)
- Similarities and differences between GMs and NNs
 - Graphical models vs. computational graphs
 - Sigmoid Belief Networks as graphical models
 - Deep Belief Networks and Boltzmann Machines
- Combining DL methods and GMs
 - Using outputs of NNs as inputs to GMs
 - GMs with potential functions represented by NNs
 - NNs with structured outputs
- Bayesian Learning of NNs
 - Bayesian learning of NN parameters
 - Deep kernel learning

Bayesian learning of NNs

- A neural network as a probabilistic model:
 - Likelihood: $p(y|x, \theta)$
 - Categorical distribution for classification \Rightarrow cross-entropy loss
 - Gaussian distribution for regression \Rightarrow squared loss
 - Prior on parameters: $p(\theta)$
- Maximum a posteriori (MAP) solution:
 - $\theta^{MAP} = \operatorname{argmax}_{\theta} \log p(y|x, \theta)p(\theta)$
 - Gaussian prior \Rightarrow L2 regularization
 - Laplace prior \Rightarrow L1 regularization

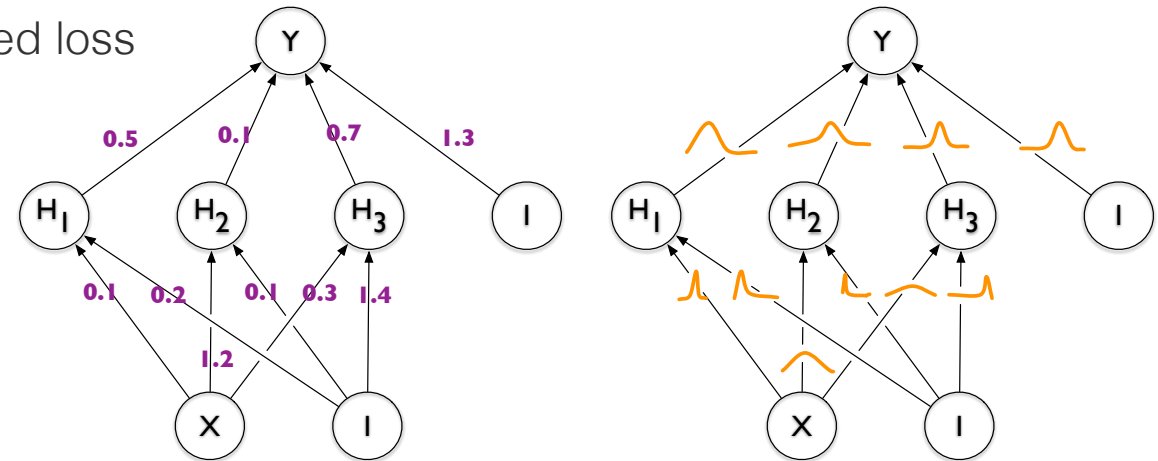


Figure courtesy: Blundell et al, 2016

- Bayesian learning [MacKay 1992, Neal 1996, de Freitas 2003]
 - Posterior: $p(\theta|x, y)$
 - Variational inference with approximate posterior $q(\theta)$

Bayesian learning of NNs

- Variational inference (in a nutshell):

$$\min_q F(D, \theta) = \text{KL}(q(\theta) || p(\theta|D)) - \mathbb{E}_{q(\theta)}[\log p(D|\theta)]$$

$$\min_q F(D, \theta) = \text{KL}(q(\theta) || p(\theta|D)) - \sum_i \log p(D|\theta_i)$$

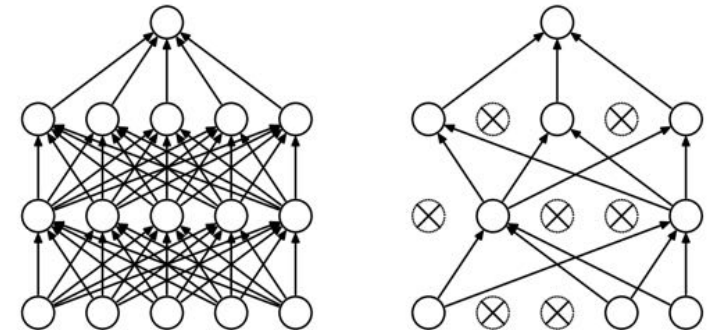
where $\theta_i \sim q(\theta)$; KL term can be approximated similarly

- We can define $q(\theta)$ as a diagonal Gaussian or full-covariance Gaussian
- Alternatively, $q(\theta)$ can be defined implicitly, e.g. via dropout [Gal & Ghahramani, 2016]

$$\theta = \mathbf{M} \cdot \text{diag}(\mathbf{z}),$$

$$\mathbf{z} \sim \text{Bernoulli}(p)$$

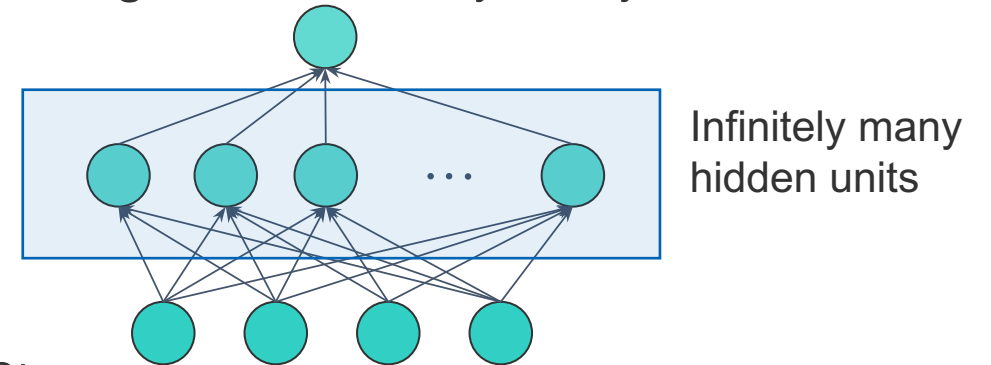
- Dropping out neurons is equivalent to zeroing out columns of the parameter matrices (i.e., weights)



- $z_i = 0$ corresponds to i -th column of \mathbf{M} being dropped out
 \Rightarrow the procedure is equivalent to dropout of unit i [Hinton et al., 2012]
- Variational parameters are $\{\mathbf{M}, \mathbf{p}\}$

“Infinitely Wide” Deep Models

- **We have seen that an “infinitely deep” network can be explained by a proper GM, How about an “infinitely wide” one?**
- Consider a neural network with a Gaussian prior on its weights an infinitely many hidden neurons in the intermediate layer.
- Turns out, if we have a certain Gaussian prior on the weights of such infinite network, it will be equivalent to a Gaussian process [Neal 1996].



- Gaussian process (GP) is a distribution over functions:

$$m(\mathbf{x}) = \mathbb{E}[f(\mathbf{x})],$$

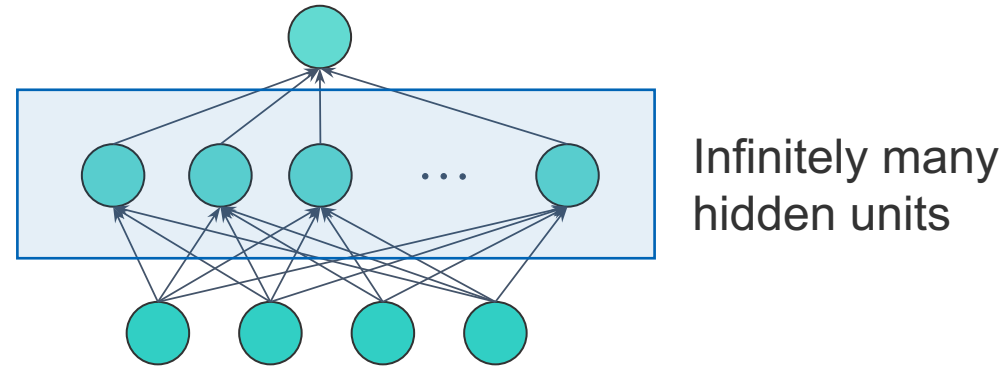
$$k(\mathbf{x}, \mathbf{x}') = \mathbb{E}[(f(\mathbf{x}) - m(\mathbf{x}))(f(\mathbf{x}') - m(\mathbf{x}'))],$$

$$f(\mathbf{x}) \sim \mathcal{GP}(m(\mathbf{x}), k(\mathbf{x}, \mathbf{x}')).$$

- When used for prediction, GPs account for correlations between the data points and can output well-calibrated predictive uncertainty estimates.

Gaussian Process and Deep Kernel Learning

- Consider a neural network with a Gaussian prior on its weights an infinitely many hidden neurons in the intermediate layer.



- Certain classes of Gaussian priors for neural networks with infinitely many hidden units converge to Gaussian processes [Neal 1996]
- Deep kernel [Wilson et al., 2016]

- Combines the inductive biases of deep model architectures with the non-parametric flexibility of Gaussian processes

$$k(x_i, x_j | \phi) \rightarrow k(g(x_i, \theta), g(x_j, \theta) | \phi, \theta) \quad \text{where } K_{ij} = k(x_i, x_j)$$

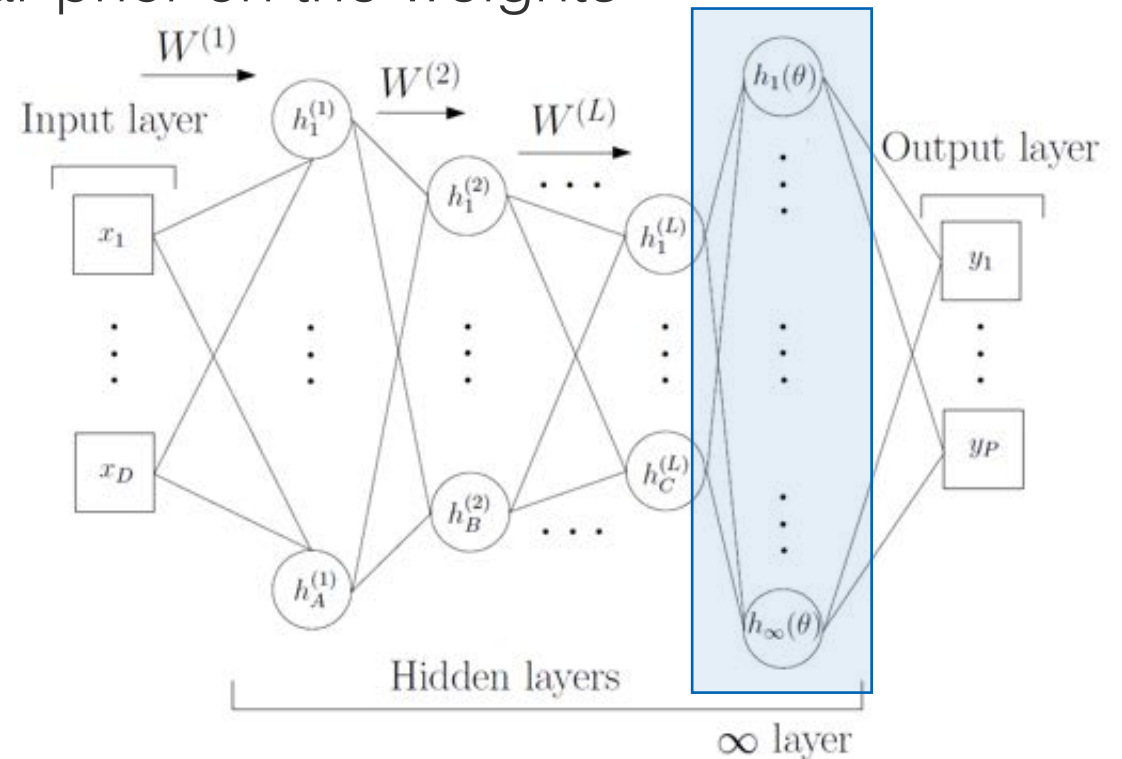
- Starting from a base kernel $k(x_i, x_j | \phi)$, transform the inputs x as

$$p(y|f) = \mathcal{N}(y|f, \beta^{-1})$$

$$p(f|\phi) = \mathcal{N}(f|m(x), K)$$
- Learn both kernel and neural parameters $\{\phi, \theta\}$ jointly by optimizing marginal log-likelihood (or its variational lower-bound).
- Fast learning and inference with local kernel interpolation, structured inducing points, and Monte Carlo approximations

Gaussian Process and Deep Kernel Learning

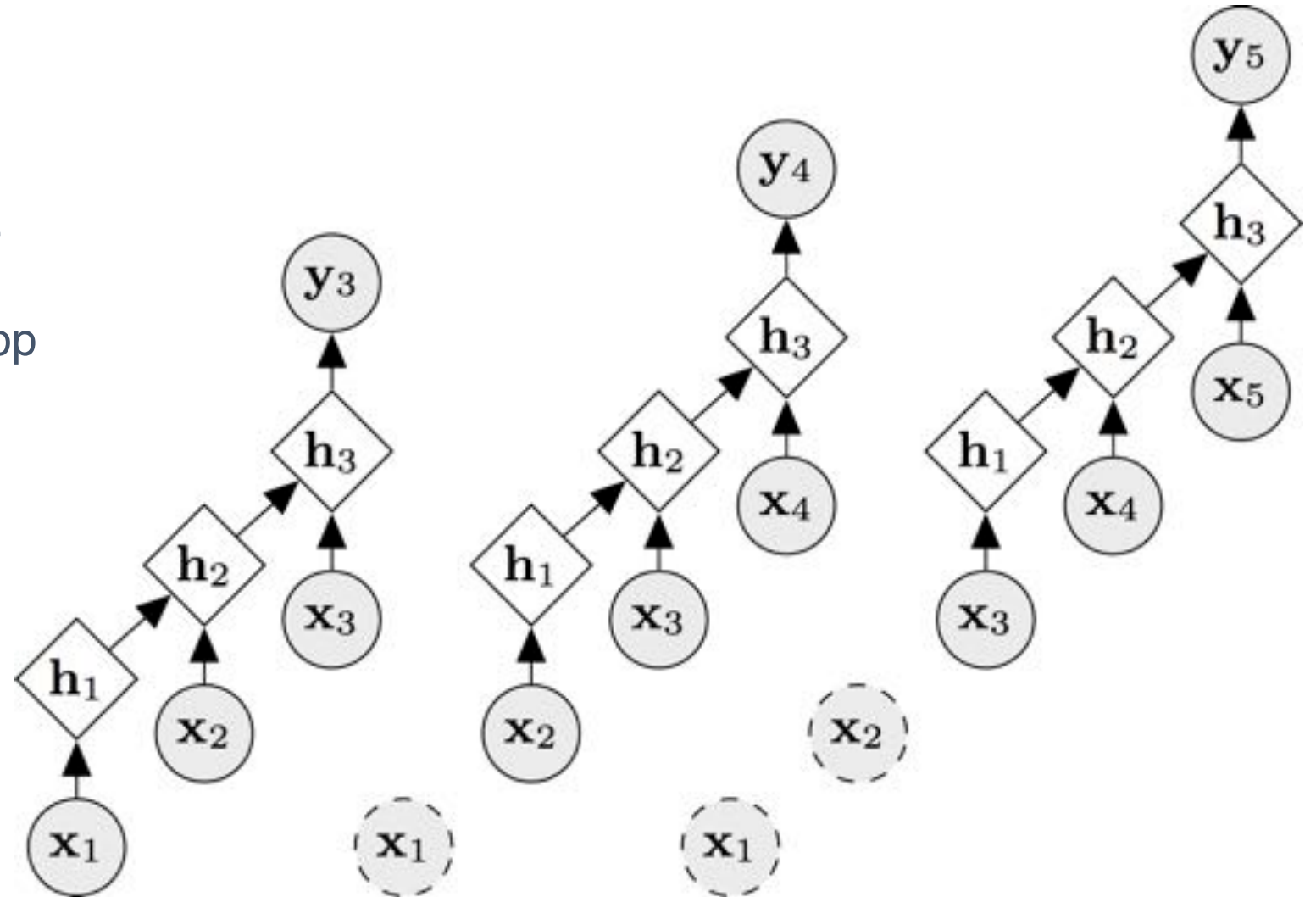
- By adding GP as a layer to a deep neural net, we can think of it as adding an infinite hidden layer with a particular prior on the weights
- Deep kernel learning [Wilson et al., 2016]
 - Combines the inductive biases of deep models with the non-parametric flexibility of Gaussian processes
 - GPs add powerful regularization to the network
 - Additionally, they provide predictive uncertainty estimates



Deep kernel learning on sequential data

What if we have data of sequential nature?

Can we still apply the same reasoning and build rich nonparametric models on top recurrent nets?

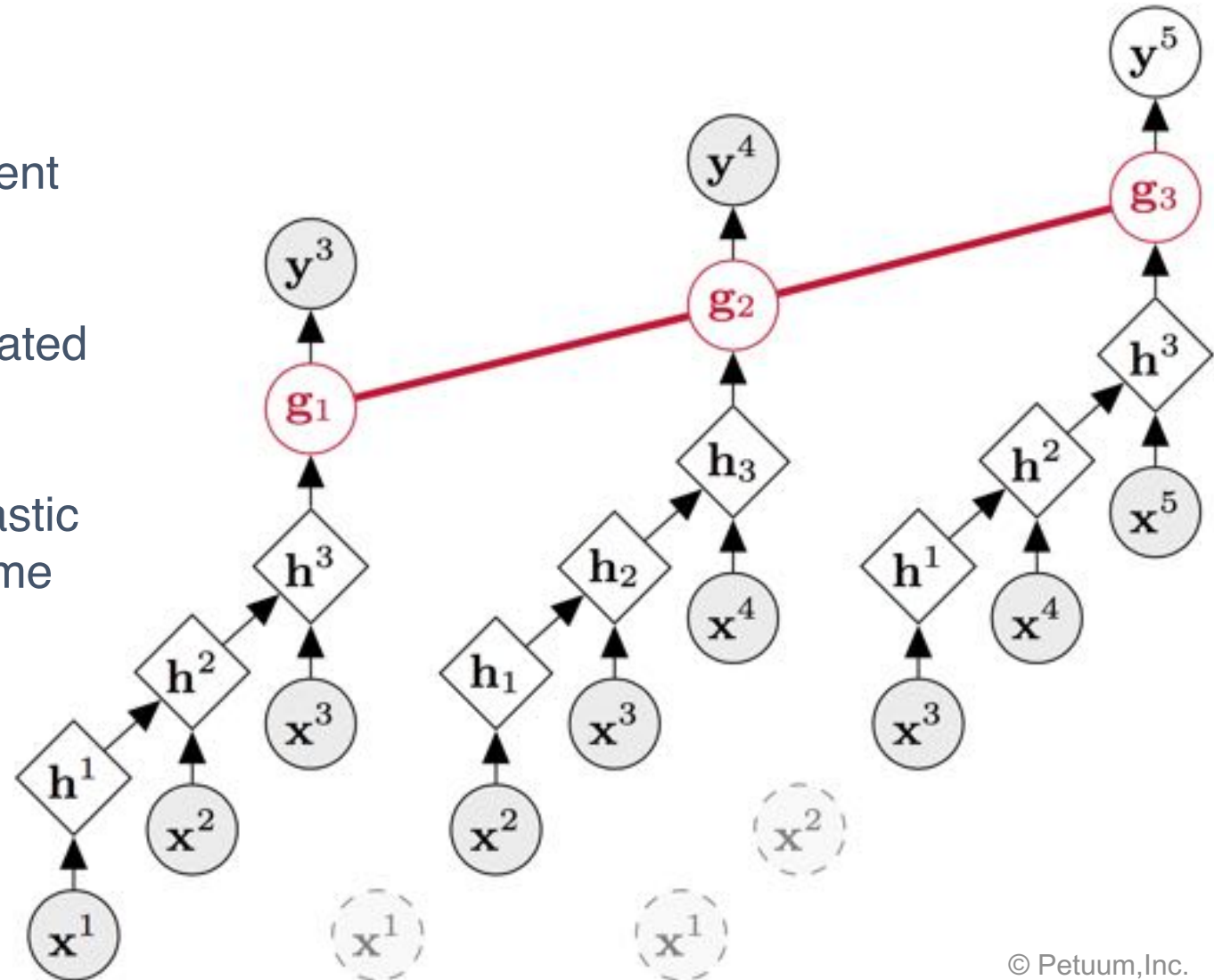


Deep kernel learning on sequential data

The answer is **YES!**

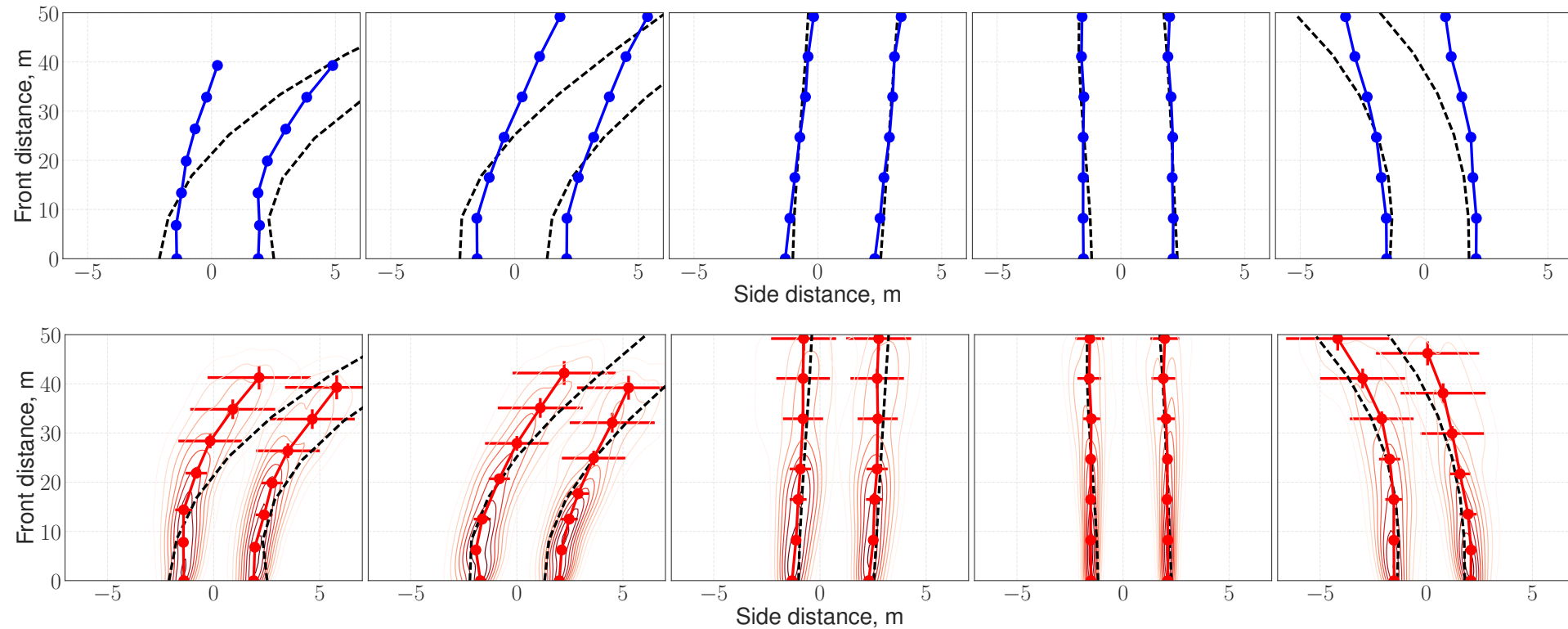
By adding a GP layer to a recurrent network, we effectively correlate samples across time and get predictions along with well calibrated uncertainty estimates.

To train such model using stochastic techniques however requires some additional care (see our paper).



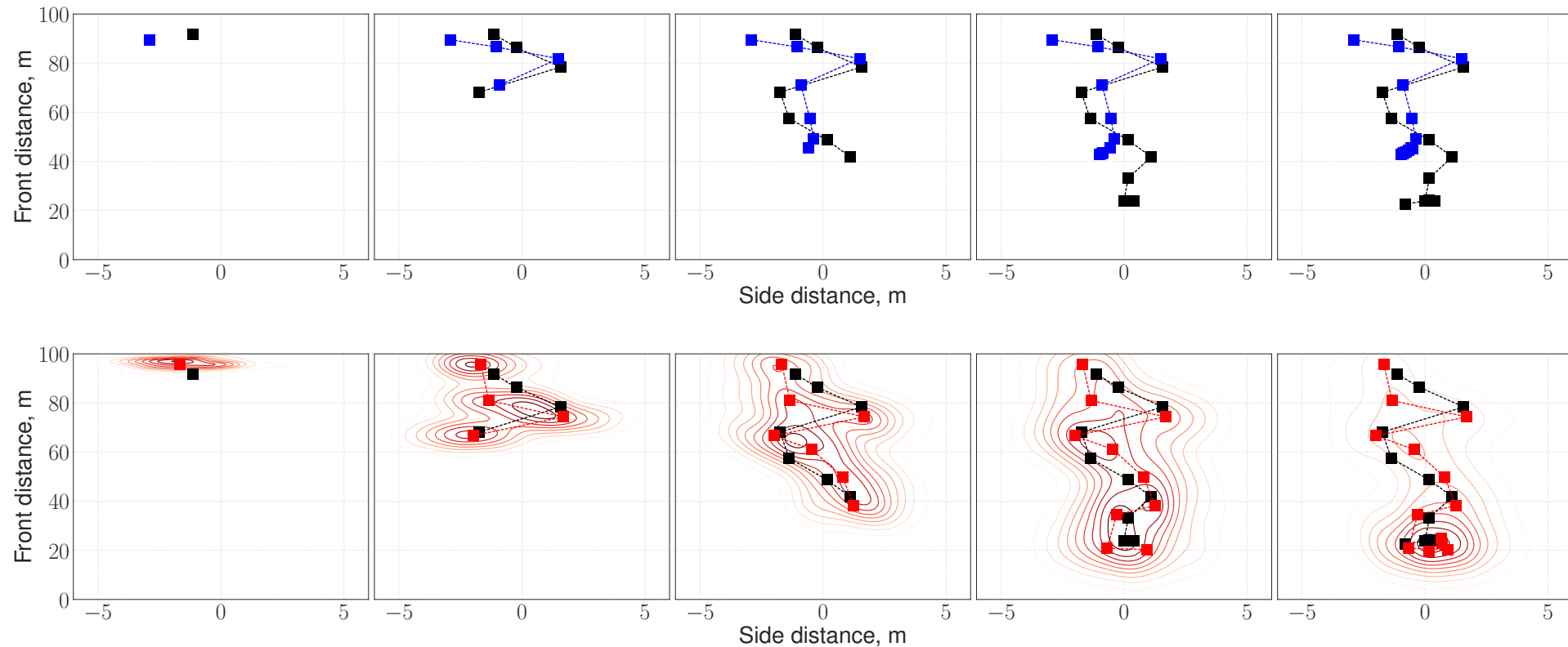
Deep kernel learning on sequential data

Lane prediction: LSTM vs GP-LSTM



Deep kernel learning on sequential data

Lead vehicle prediction: LSTM vs GP-LSTM



Conclusion

- DL & GM: the fields are similar in the beginning (structure, energy, etc.), and then diverge to their own signature pipelines
- DL: most effort is directed to comparing different architectures and their components (models are driven by evaluating empirical performance on a downstream tasks)
 - DL models are good at learning robust hierarchical representations from the data and suitable for simple reasoning (call it “low-level cognition”)
- GM: the effort is directed towards improving inference accuracy and convergence speed
 - GMs are best for provably correct inference and suitable for high-level complex reasoning tasks (call it “high-level cognition”)
- Convergence of both fields is very promising!
 - Next part: a unified view of deep generative models in the GM interpretation



Part-II

Deep Generative Models

Plan

- Statistical And Algorithmic Foundation and Insight of Deep Learning
- On Unified Framework of Deep Generative Models
- Computational Mechanisms: Distributed Deep Learning Architectures

Outline

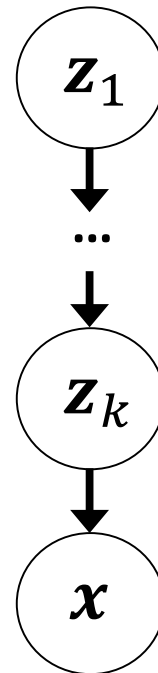
- Overview of advances in deep generative models
- Backgrounds of deep generative models
 - Wake sleep algorithm
 - Variational autoencoders
 - Generative adversarial networks
- A unified view of deep generative models
 - new formulations of deep generative models
 - Symmetric modeling of latent and visible variables

Outline

- Overview of advances in deep generative models
- Backgrounds of deep generative models
 - Wake sleep algorithm
 - Variational autoencoders
 - Generative adversarial networks
- A unified view of deep generative models
 - new formulations of deep generative models
 - Symmetric modeling of latent and visible variables

Deep generative models

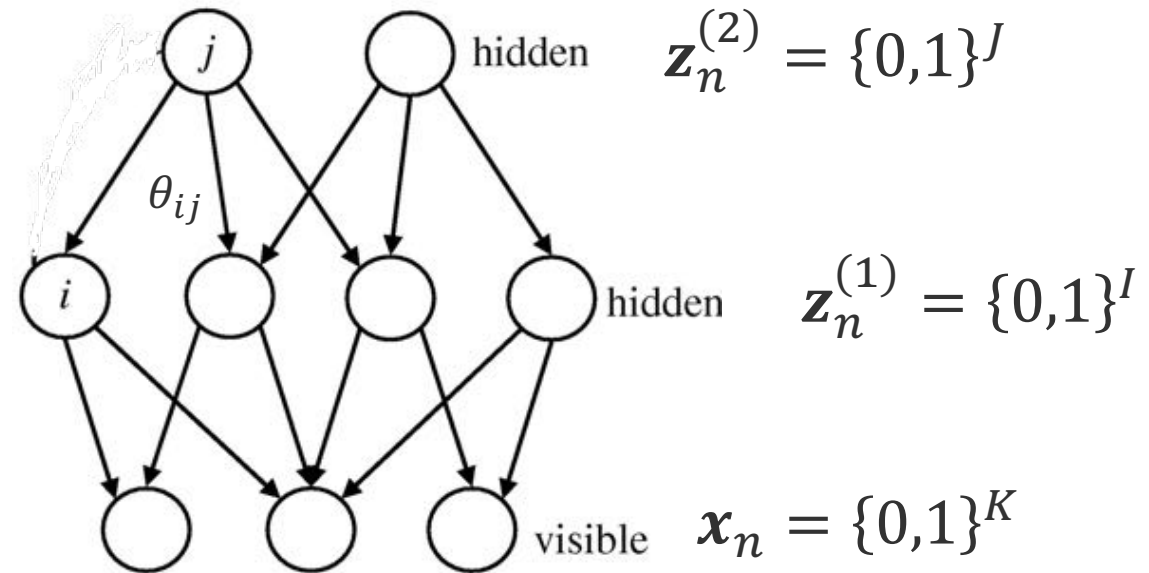
- Define probabilistic distributions over a set of variables
- "Deep" means multiple layers of hidden variables!



Early forms of deep generative models

- Hierarchical Bayesian models

- Sigmoid belief nets [Neal 1992]

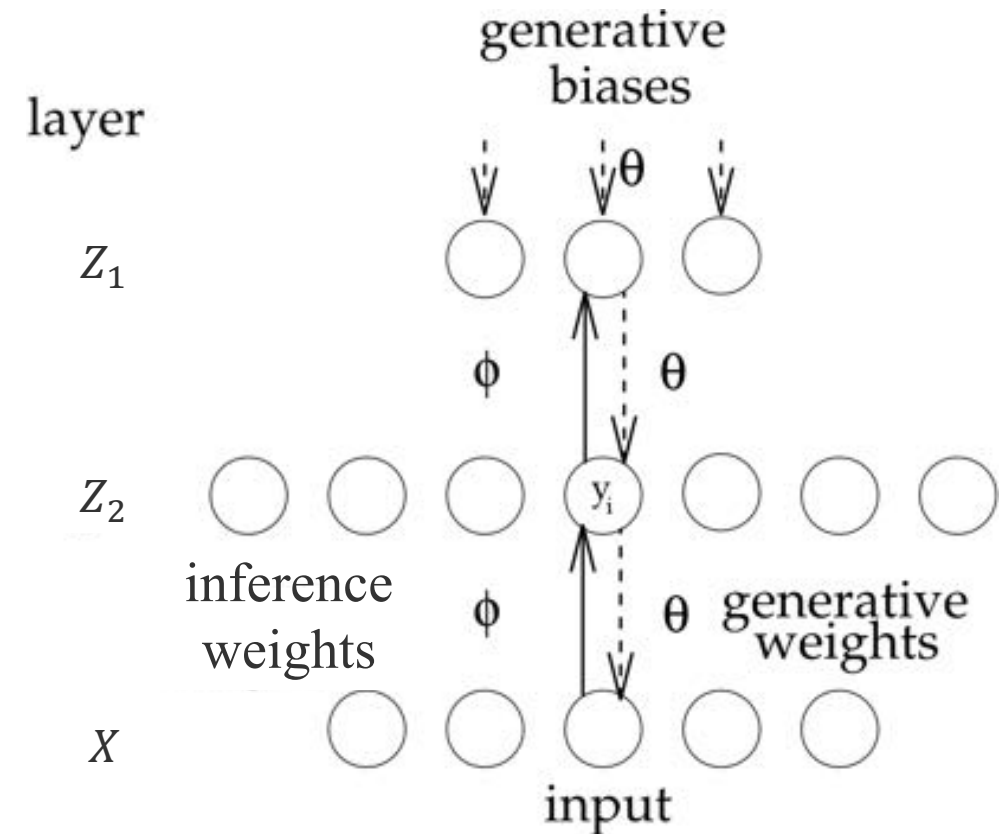


$$p\left(x_{kn} = 1 \mid \boldsymbol{\theta}_k, \mathbf{z}_n^{(1)}\right) = \sigma\left(\boldsymbol{\theta}_k^T \mathbf{z}_n^{(1)}\right)$$

$$p\left(z_{in}^{(1)} = 1 \mid \boldsymbol{\theta}_i, \mathbf{z}_n^{(2)}\right) = \sigma\left(\boldsymbol{\theta}_i^T \mathbf{z}_n^{(2)}\right)$$

Early forms of deep generative models

- Hierarchical Bayesian models
 - Sigmoid belief nets [Neal 1992]
- Neural network models
 - Helmholtz machines [Dayan et al., 1995]



[Dayan et al. 1995]

Early forms of deep generative models

- Hierarchical Bayesian models
 - Sigmoid belief nets [Neal 1992]
- Neural network models
 - Helmholtz machines [Dayan et al., 1995]
 - Predictability minimization [Schmidhuber 1995]

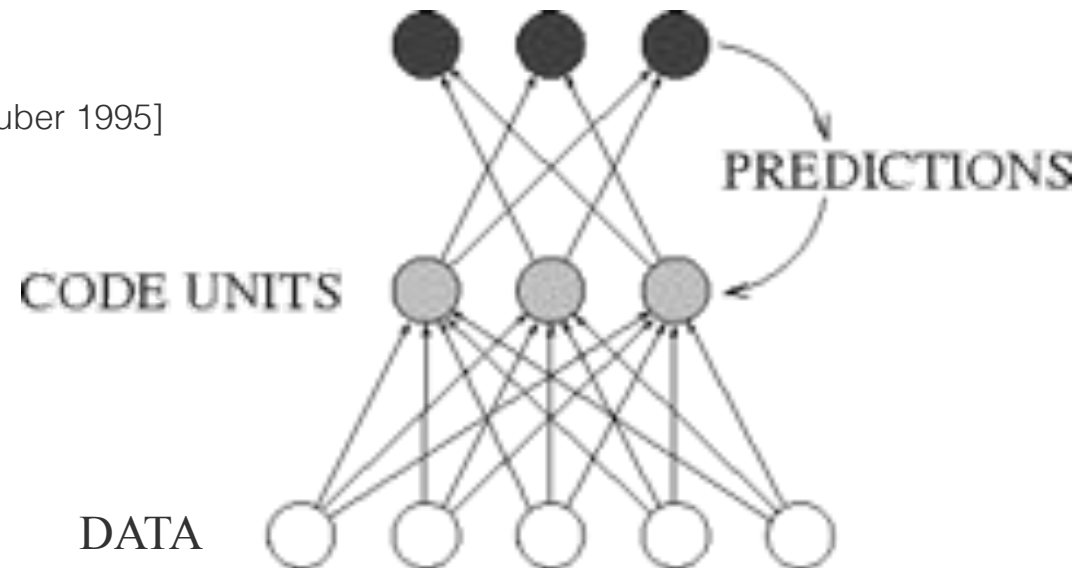


Figure courtesy: Schmidhuber 1996

Early forms of deep generative models

- Training of DGMs via an EM style framework
 - Sampling / data augmentation

$$\mathbf{z} = \{\mathbf{z}_1, \mathbf{z}_2\}$$

$$\mathbf{z}_1^{new} \sim p(\mathbf{z}_1 | \mathbf{z}_2, \mathbf{x})$$

$$\mathbf{z}_2^{new} \sim p(\mathbf{z}_2 | \mathbf{z}_1^{new}, \mathbf{x})$$

- Variational inference

$$\log p(\mathbf{x}) \geq \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}, \mathbf{z})] - \text{KL}(q_\phi(\mathbf{z}|\mathbf{x}) || p(\mathbf{z})) := \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x})$$

$$\max_{\boldsymbol{\theta}, \boldsymbol{\phi}} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x})$$

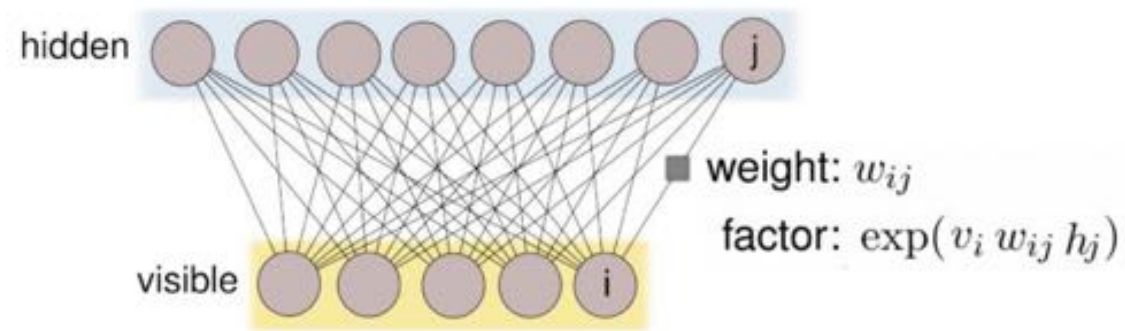
- Wake sleep

$$\text{Wake: } \min_{\boldsymbol{\theta}} \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}|\mathbf{z})]$$

$$\text{Sleep: } \min_{\boldsymbol{\phi}} \mathbb{E}_{p_\theta(\mathbf{x}|\mathbf{z})}[\log q_\phi(\mathbf{z}|\mathbf{x})]$$

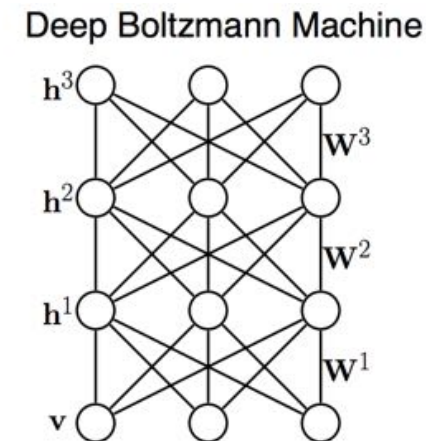
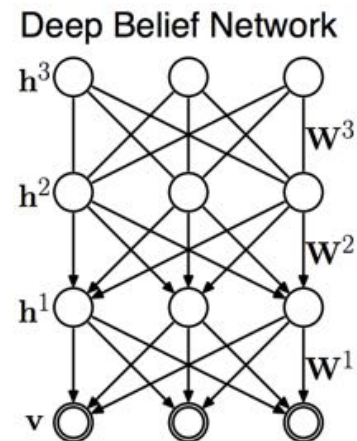
Resurgence of deep generative models

- Restricted Boltzmann machines (RBMs) [Smolensky, 1986]
 - Building blocks of deep probabilistic models



Resurgence of deep generative models

- Restricted Boltzmann machines (RBMs) [Smolensky, 1986]
 - Building blocks of deep probabilistic models
- Deep belief networks (DBNs) [Hinton et al., 2006]
 - Hybrid graphical model
 - Inference in DBNs is problematic due to explaining away
- Deep Boltzmann Machines (DBMs) [Salakhutdinov & Hinton, 2009]
 - Undirected model



Resurgence of deep generative models

- Variational autoencoders (VAEs) [Kingma & Welling, 2014]
 / Neural Variational Inference and Learning (NVIL) [Mnih & Gregor, 2014]

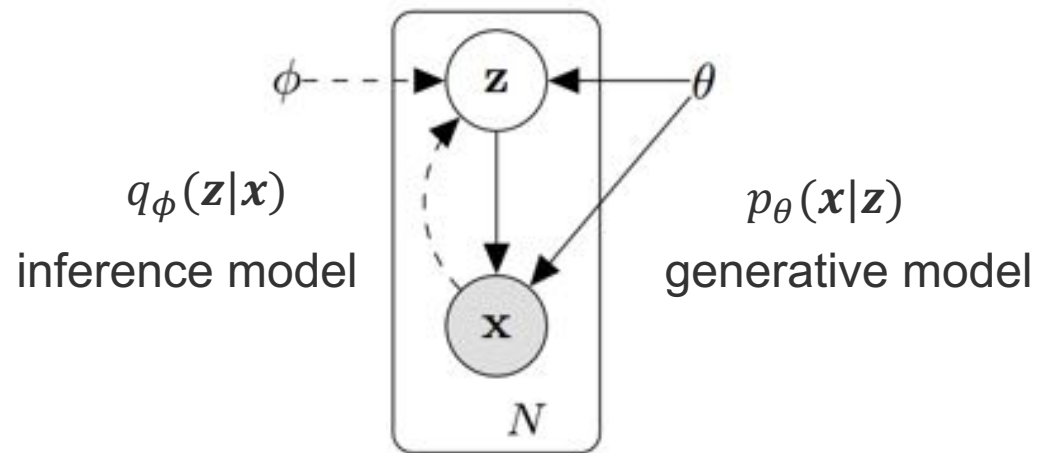
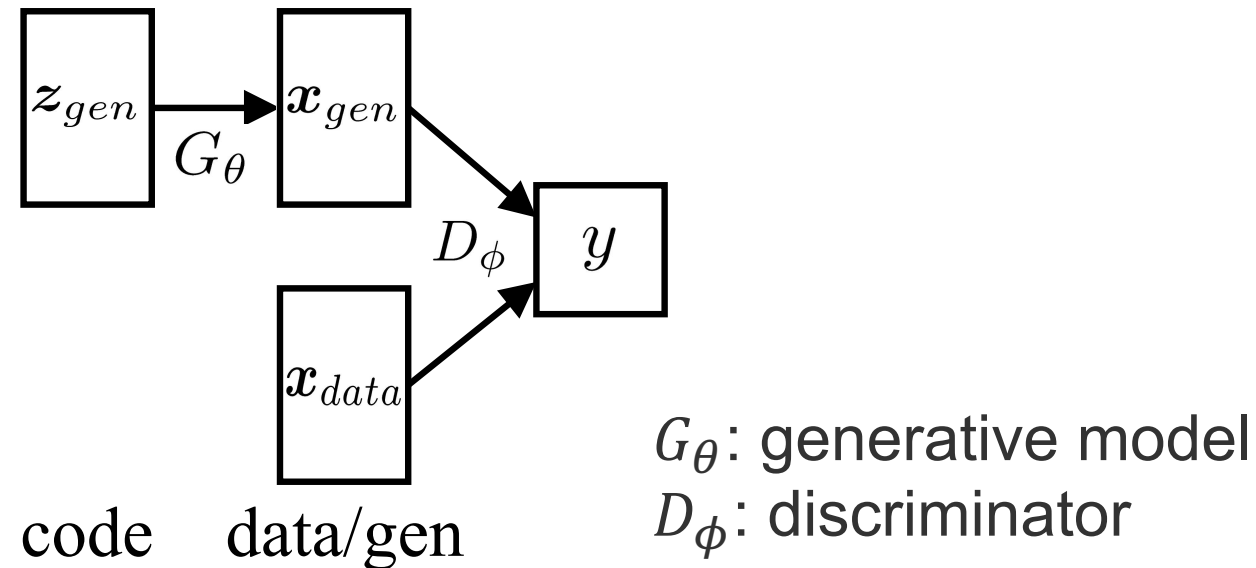


Figure courtesy: Kingma & Welling, 2014

Resurgence of deep generative models

- Variational autoencoders (VAEs) [Kingma & Welling, 2014]
/ Neural Variational Inference and Learning (NVIL) [Mnih & Gregor, 2014]
- Generative adversarial networks (GANs)

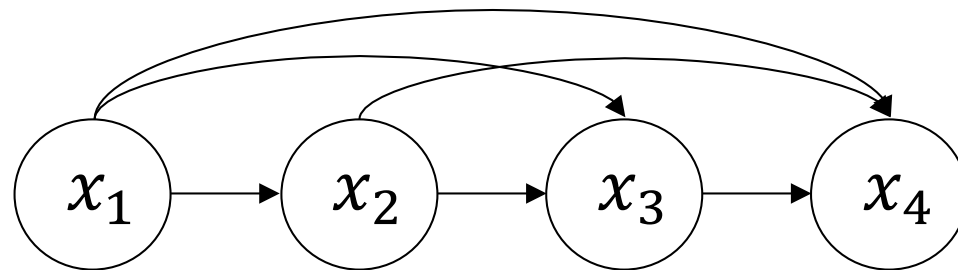


Resurgence of deep generative models

- Variational autoencoders (VAEs) [Kingma & Welling, 2014]
/ Neural Variational Inference and Learning (NVIL) [Mnih & Gregor, 2014]
- Generative adversarial networks (GANs)
- Generative moment matching networks (GMMNs) [Li et al., 2015; Dziugaite et al., 2015]

Resurgence of deep generative models

- Variational autoencoders (VAEs) [Kingma & Welling, 2014]
/ Neural Variational Inference and Learning (NVIL) [Mnih & Gregor, 2014]
- Generative adversarial networks (GANs)
- Generative moment matching networks (GMMNs) [Li et al., 2015; Dziugaite et al., 2015]
- Autoregressive neural networks



Outline

- Overview of advances in deep generative models
- Backgrounds of deep generative models
 - Wake sleep algorithm
 - Variational autoencoders
 - Generative adversarial networks
- A unified view of deep generative models
 - new formulations of deep generative models
 - Symmetric modeling of latent and visible variables

Synonyms in the literature

- Posterior Distribution -> Inference model
 - Variational approximation
 - Recognition model
 - Inference network (if parameterized as neural networks)
 - Recognition network (if parameterized as neural networks)
 - (Probabilistic) encoder
- "The Model" (prior + conditional, or joint) -> Generative model
 - The (data) likelihood model
 - Generative network (if parameterized as neural networks)
 - Generator
 - (Probabilistic) decoder

Recap: Variational Inference

- Consider a generative model $p_{\theta}(\mathbf{x}|\mathbf{z})$, and prior $p(\mathbf{z})$
 - Joint distribution: $p_{\theta}(\mathbf{x}, \mathbf{z}) = p_{\theta}(\mathbf{x}|\mathbf{z})p(\mathbf{z})$
- Assume **variational distribution** $q_{\phi}(\mathbf{z}|\mathbf{x})$
- Objective: Maximize **lower bound** for log likelihood

$$\log p(\mathbf{x})$$

$$= KL(q_{\phi}(\mathbf{z}|\mathbf{x}) || p_{\theta}(\mathbf{z}|\mathbf{x})) + \int_{\mathbf{z}} q_{\phi}(\mathbf{z}|\mathbf{x}) \log \frac{p_{\theta}(\mathbf{x}, \mathbf{z})}{q_{\phi}(\mathbf{z}|\mathbf{x})}$$

$$\geq \int_{\mathbf{z}} q_{\phi}(\mathbf{z}|\mathbf{x}) \log \frac{p_{\theta}(\mathbf{x}, \mathbf{z})}{q_{\phi}(\mathbf{z}|\mathbf{x})}$$

$$:= \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x})$$

- Equivalently, minimize **free energy**

$$F(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x}) = -\log p(\mathbf{x}) + KL(q_{\phi}(\mathbf{z}|\mathbf{x}) || p_{\theta}(\mathbf{z}|\mathbf{x}))$$

Recap: Variational Inference

Maximize the variational lower bound $\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x})$

- E-step: maximize \mathcal{L} wrt. $\boldsymbol{\phi}$ with $\boldsymbol{\theta}$ fixed

$$\max_{\boldsymbol{\phi}} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x}) = \mathbb{E}_{q_{\boldsymbol{\phi}}(z|\mathbf{x})}[\log p_{\boldsymbol{\theta}}(\mathbf{x}|z)] + KL(q_{\boldsymbol{\phi}}(z|\mathbf{x})||p(z))$$

- If with closed form solutions

$$q_{\boldsymbol{\phi}}^*(z|\mathbf{x}) \propto \exp[\log p_{\boldsymbol{\theta}}(\mathbf{x}, z)]$$

- M-step: maximize \mathcal{L} wrt. $\boldsymbol{\theta}$ with $\boldsymbol{\phi}$ fixed

$$\max_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x}) = \mathbb{E}_{q_{\boldsymbol{\phi}}(z|\mathbf{x})}[\log p_{\boldsymbol{\theta}}(\mathbf{x}|z)] + KL(q_{\boldsymbol{\phi}}(z|\mathbf{x})||p(z))$$

Recap: Amortized Variational Inference

- Variational distribution as an **inference model** $q_{\phi}(\mathbf{z}|\mathbf{x})$ with parameters ϕ
- Amortize the cost of inference by learning a single data-dependent inference model
- The trained inference model can be used for quick inference on new data
- **Maximize the variational lower bound** $\mathcal{L}(\theta, \phi; \mathbf{x})$
 - E-step: maximize \mathcal{L} wrt. ϕ with θ fixed
 - M-step: maximize \mathcal{L} wrt. θ with ϕ fixed

Deep generative models with amortized inference

- Helmholtz machines
- Variational autoencoders (VAEs) / Neural Variational Inference and Learning (NVIL)
- We will see later that adversarial approaches are also included in the list
 - Predictability minimization (PM)
 - Generative adversarial networks (GANs)

Wake Sleep Algorithm

- [Hinton et al., Science 1995]
- Train a separate inference model along with the generative model
 - Generally applicable to a wide range of generative models, e.g., Helmholtz machines
- Consider a generative model $p_{\theta}(\mathbf{x}|\mathbf{z})$ and prior $p(\mathbf{z})$
 - Joint distribution $p_{\theta}(\mathbf{x}, \mathbf{z}) = p_{\theta}(\mathbf{x}|\mathbf{z})p(\mathbf{z})$
 - E.g., multi-layer brief nets
- Inference model $q_{\phi}(\mathbf{z}|\mathbf{x})$
- Maximize data log-likelihood with **two steps of loss relaxation**:
 - Maximize the **lower bound** of log-likelihood, or equivalently, minimize the free energy

$$F(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x}) = -\log p(\mathbf{x}) + KL(q_{\phi}(\mathbf{z}|\mathbf{x}) || p_{\theta}(\mathbf{z}|\mathbf{x}))$$

- Minimize a different objective (**reversed KLD**) wrt ϕ to ease the optimization
 - Disconnect to the original variational lower bound loss

$$F'(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x}) = -\log p(\mathbf{x}) + KL(p_{\theta}(\mathbf{z}|\mathbf{x}) || q_{\phi}(\mathbf{z}|\mathbf{x}))$$

Wake Sleep Algorithm

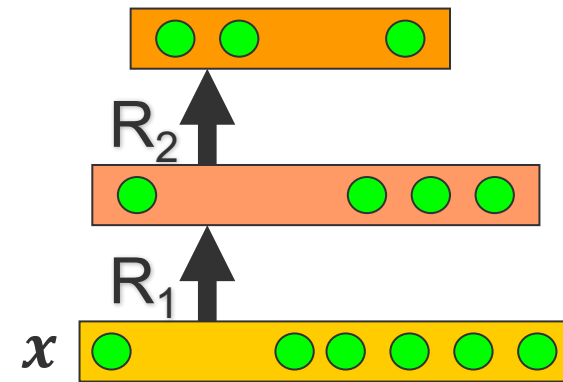
- Free energy:

$$F(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x}) = -\log p(\mathbf{x}) + KL(q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) || p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x}))$$

- Minimize the free energy wrt. $\boldsymbol{\theta}$ of $p_{\boldsymbol{\theta}}$ \rightarrow *wake* phase

$$\max_{\boldsymbol{\theta}} E_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})} [\log p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})]$$

- Get samples from $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$ through inference on hidden variables
- Use the samples as targets for updating the generative model $p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})$
- Correspond to the variational M step



Wake Sleep Algorithm

- Free energy:

$$F(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x}) = -\log p(\mathbf{x}) + KL(q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) || p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x}))$$

- Minimize the free energy wrt. $\boldsymbol{\phi}$ of $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$

- Correspond to the variational E step

- Difficulties:

- Optimal $q_{\boldsymbol{\phi}}^*(\mathbf{z}|\mathbf{x}) = \frac{p_{\boldsymbol{\theta}}(\mathbf{z}, \mathbf{x})}{\int p_{\boldsymbol{\theta}}(\mathbf{z}, \mathbf{x}) d\mathbf{z}}$ intractable

- High variance of direct gradient estimate $\nabla_{\boldsymbol{\phi}} F(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x}) = \dots + \nabla_{\boldsymbol{\phi}} \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}[\log p_{\boldsymbol{\theta}}(\mathbf{z}, \mathbf{x})] + \dots$

- Gradient estimate with the log-derivative trick:

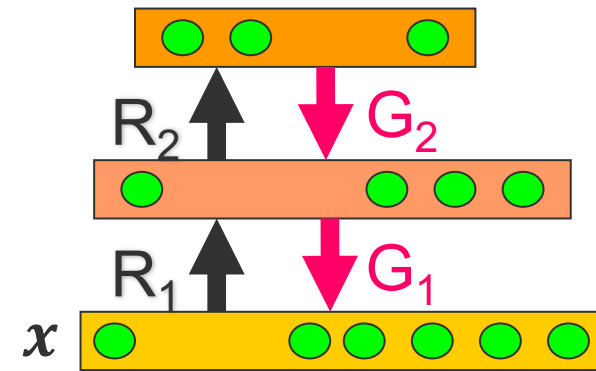
$$\nabla_{\boldsymbol{\phi}} \mathbb{E}_{q_{\boldsymbol{\phi}}}[\log p_{\boldsymbol{\theta}}] = \int \nabla_{\boldsymbol{\phi}} q_{\boldsymbol{\phi}} \log p_{\boldsymbol{\theta}} = \int q_{\boldsymbol{\phi}} \log p_{\boldsymbol{\theta}} \nabla_{\boldsymbol{\phi}} \log q_{\boldsymbol{\phi}} = \mathbb{E}_{q_{\boldsymbol{\phi}}}[\log p_{\boldsymbol{\theta}} \nabla_{\boldsymbol{\phi}} \log q_{\boldsymbol{\phi}}]$$

- Monte Carlo estimation:

$$\nabla_{\boldsymbol{\phi}} \mathbb{E}_{q_{\boldsymbol{\phi}}}[\log p_{\boldsymbol{\theta}}] \approx \mathbb{E}_{z_i \sim q_{\boldsymbol{\phi}}}[\log p_{\boldsymbol{\theta}}(\mathbf{x}, z_i) \nabla_{\boldsymbol{\phi}} \log q_{\boldsymbol{\phi}}(z_i|\mathbf{x})]$$

- The scale factor $\log p_{\boldsymbol{\theta}}$ of the derivative $\nabla_{\boldsymbol{\phi}} \log q_{\boldsymbol{\phi}}$ can have arbitrary large magnitude

Wake Sleep Algorithm



- Free energy:

$$F(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x}) = -\log p(\mathbf{x}) + KL(q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) || p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x}))$$

- WS works around the difficulties with the sleep phase approximation
- Minimize the following objective \rightarrow *sleep* phase

$$F'(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x}) = -\log p(\mathbf{x}) + KL(p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x}) || q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}))$$

$$\max_{\boldsymbol{\phi}} E_{p_{\boldsymbol{\theta}}(\mathbf{z}, \mathbf{x})} [\log q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})]$$

- “Dreaming” up samples from $p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})$ through top-down pass
- Use the samples as targets for updating the inference model
- (Recent approaches other than sleep phase is to reduce the variance of gradient estimate: slides later)

Wake Sleep Algorithm

Wake sleep

- Parametrized inference model $q_\phi(\mathbf{z}|\mathbf{x})$
- Wake phase:
 - minimize $KL(q_\phi(\mathbf{z}|\mathbf{x}) || p_\theta(\mathbf{z}|\mathbf{x}))$ wrt. θ
 - $\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})} [\nabla_\theta \log p_\theta(\mathbf{x}|\mathbf{z})]$
- Sleep phase:
 - minimize $KL(p_\theta(\mathbf{z}|\mathbf{x}) || q_\phi(\mathbf{z}|\mathbf{x}))$ wrt. ϕ
 - $\mathbb{E}_{p_\theta(\mathbf{z},\mathbf{x})} [\nabla_\phi \log q_\phi(\mathbf{z}, \mathbf{x})]$
 - low variance
 - Learning with generated samples of \mathbf{x}
- Two objective, not guaranteed to converge

Variational EM

- Variational distribution $q_\phi(\mathbf{z}|\mathbf{x})$
- Variational M step:
 - minimize $KL(q_\phi(\mathbf{z}|\mathbf{x}) || p_\theta(\mathbf{z}|\mathbf{x}))$ wrt. θ
 - $\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})} [\nabla_\theta \log p_\theta(\mathbf{x}|\mathbf{z})]$
- Variational E step:
 - minimize $KL(q_\phi(\mathbf{z}|\mathbf{x}) || p_\theta(\mathbf{z}|\mathbf{x}))$ wrt. ϕ
 - $q_\phi^* \propto \exp[\log p_\theta]$ if with closed-form
 - $\nabla_\phi \mathbb{E}_{q_\phi} [\log p_\theta(\mathbf{z}, \mathbf{x})]$
 - need variance-reduce in practice
 - Learning with real data \mathbf{x}
- Single objective, guaranteed to converge

Variational Autoencoders (VAEs)

- [Kingma & Welling, 2014]
- Use variational inference with an inference model
 - Enjoy similar applicability with wake-sleep algorithm
- Generative model $p_{\theta}(\mathbf{x}|\mathbf{z})$, and prior $p(\mathbf{z})$
 - Joint distribution $p_{\theta}(\mathbf{x}, \mathbf{z}) = p_{\theta}(\mathbf{x}|\mathbf{z})p(\mathbf{z})$
- Inference model $q_{\phi}(\mathbf{z}|\mathbf{x})$

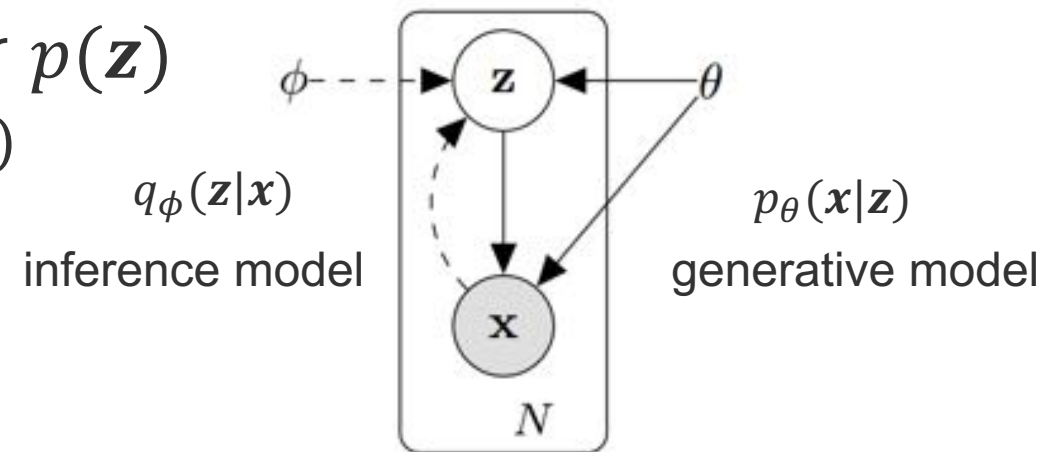


Figure courtesy: Kingma & Welling, 2014

Variational Autoencoders (VAEs)

- Variational lower bound

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x}) = \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}[\log p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})] - \text{KL}(q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) || p(\mathbf{z}))$$

- Optimize $\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x})$ wrt. $\boldsymbol{\theta}$ of $p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})$
 - The same with the wake phase
- Optimize $\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x})$ wrt. $\boldsymbol{\phi}$ of $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$

$$\nabla_{\boldsymbol{\phi}} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x}) = \dots + \nabla_{\boldsymbol{\phi}} \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}[\log p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})] + \dots$$

- Use *reparameterization trick* to reduce variance
- Alternatives: use control variates as in reinforcement learning [Mnih & Gregor, 2014; Paisley et al., 2012]

Reparametrized gradient

- Optimize $\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x})$ wrt. $\boldsymbol{\phi}$ of $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$
 - Recap: gradient estimate with log-derivative trick:

$$\nabla_{\boldsymbol{\phi}} \mathbb{E}_{q_{\boldsymbol{\phi}}} [\log p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})] = \mathbb{E}_{q_{\boldsymbol{\phi}}} [\log p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z}) \nabla_{\boldsymbol{\phi}} \log q_{\boldsymbol{\phi}}]$$
 - High variance: $\nabla_{\boldsymbol{\phi}} \mathbb{E}_{q_{\boldsymbol{\phi}}} [\log p_{\boldsymbol{\theta}}] \approx \mathbb{E}_{z_i \sim q_{\boldsymbol{\phi}}} [\log p_{\boldsymbol{\theta}}(\mathbf{x}, z_i) \nabla_{\boldsymbol{\phi}} q_{\boldsymbol{\phi}}(z_i|\mathbf{x})]$
 - The scale factor $\log p_{\boldsymbol{\theta}}(\mathbf{x}, z_i)$ of the derivative $\nabla_{\boldsymbol{\phi}} \log q_{\boldsymbol{\phi}}$ can have arbitrary large magnitude
 - gradient estimate with *reparameterization trick*

$$\mathbf{z} \sim q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) \Leftrightarrow \mathbf{z} = \mathbf{g}_{\boldsymbol{\phi}}(\boldsymbol{\epsilon}, \mathbf{x}), \quad \boldsymbol{\epsilon} \sim p(\boldsymbol{\epsilon})$$

$$\nabla_{\boldsymbol{\phi}} \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})} [\log p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})] = \mathbb{E}_{\boldsymbol{\epsilon} \sim p(\boldsymbol{\epsilon})} \left[\nabla_{\boldsymbol{\phi}} \log p_{\boldsymbol{\theta}} \left(\mathbf{x}, \mathbf{z}_{\boldsymbol{\phi}}(\boldsymbol{\epsilon}) \right) \right]$$

- (Empirically) lower variance of the gradient estimate
- E.g., $\mathbf{z} \sim N(\boldsymbol{\mu}(\mathbf{x}), \mathbf{L}(\mathbf{x})\mathbf{L}(\mathbf{x})^T) \Leftrightarrow \boldsymbol{\epsilon} \sim N(0,1), \mathbf{z} = \boldsymbol{\mu}(\mathbf{x}) + \mathbf{L}(\mathbf{x})\boldsymbol{\epsilon}$

VAEs: algorithm

Algorithm 1 Minibatch version of the Auto-Encoding VB (AEVB) algorithm. Either of the two SGVB estimators in section 2.3 can be used. We use settings $M = 100$ and $L = 1$ in experiments.

$\theta, \phi \leftarrow$ Initialize parameters

repeat

$\mathbf{X}^M \leftarrow$ Random minibatch of M datapoints (drawn from full dataset)

$\epsilon \leftarrow$ Random samples from noise distribution $p(\epsilon)$

$\mathbf{g} \leftarrow \nabla_{\theta, \phi} \tilde{\mathcal{L}}^M(\theta, \phi; \mathbf{X}^M, \epsilon)$ (Gradients of minibatch estimator (8))

$\theta, \phi \leftarrow$ Update parameters using gradients \mathbf{g} (e.g. SGD or Adagrad [DHS10])

until convergence of parameters (θ, ϕ)

return θ, ϕ

[Kingma & Welling, 2014]

VAEs: example results

- VAEs tend to generate blurred images due to the mode covering behavior (more later)



Celebrity faces [Radford 2015]

- Latent code interpolation and sentences generation from VAEs [Bowman et al., 2015].

“ i want to talk to you . ”
“i want to be with you . ”
“i do n’t want to be with you . ”
i do n’t want to be with you .
she did n’t want to be with him .

Generative Adversarial Nets (GANs)

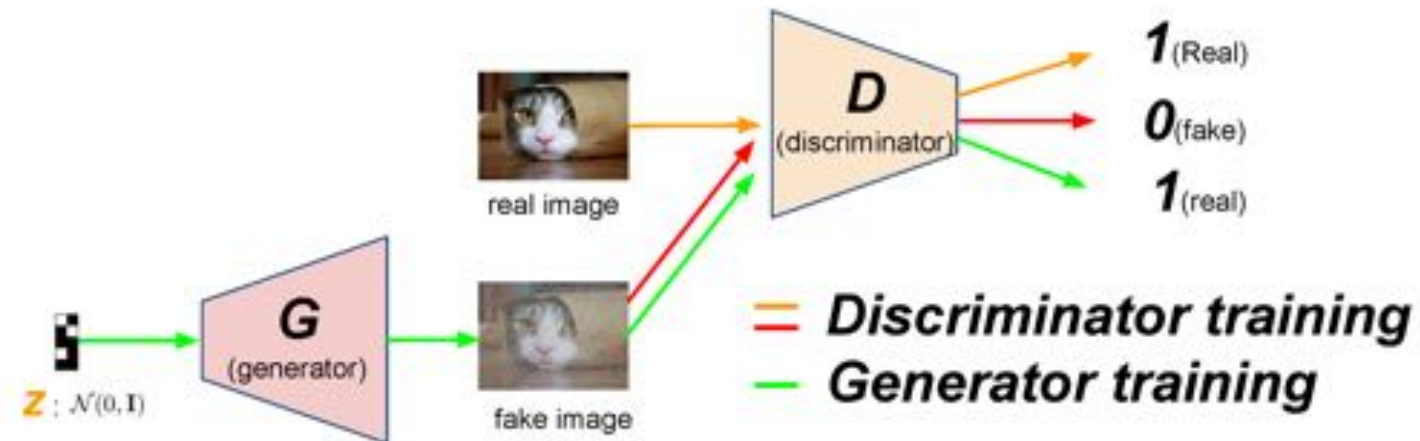
- [Goodfellow et al., 2014]
- Generative model $\mathbf{x} = G_{\theta}(\mathbf{z})$, $\mathbf{z} \sim p(\mathbf{z})$
 - Map noise variable \mathbf{z} to data space \mathbf{x}
 - Define an **implicit distribution** over \mathbf{x} : $p_{g_{\theta}}(\mathbf{x})$
 - a stochastic process to simulate data \mathbf{x}
 - Intractable to evaluate likelihood
- Discriminator $D_{\phi}(\mathbf{x})$
 - Output the probability that \mathbf{x} came from the data rather than the generator
- No explicit inference model
- No obvious connection to previous models with inference networks like VAEs
 - We will build formal connections between GANs and VAEs later

Generative Adversarial Nets (GANs)

- Learning
 - A minimax game between the generator and the discriminator
 - Train D to maximize the probability of assigning the correct label to both training examples and generated samples
 - Train G to fool the discriminator

$$\max_D \mathcal{L}_D = \mathbb{E}_{\mathbf{x} \sim p_{data}(\mathbf{x})} [\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{x} \sim G(\mathbf{z}), \mathbf{z} \sim p(\mathbf{z})} [\log(1 - D(\mathbf{x}))]$$

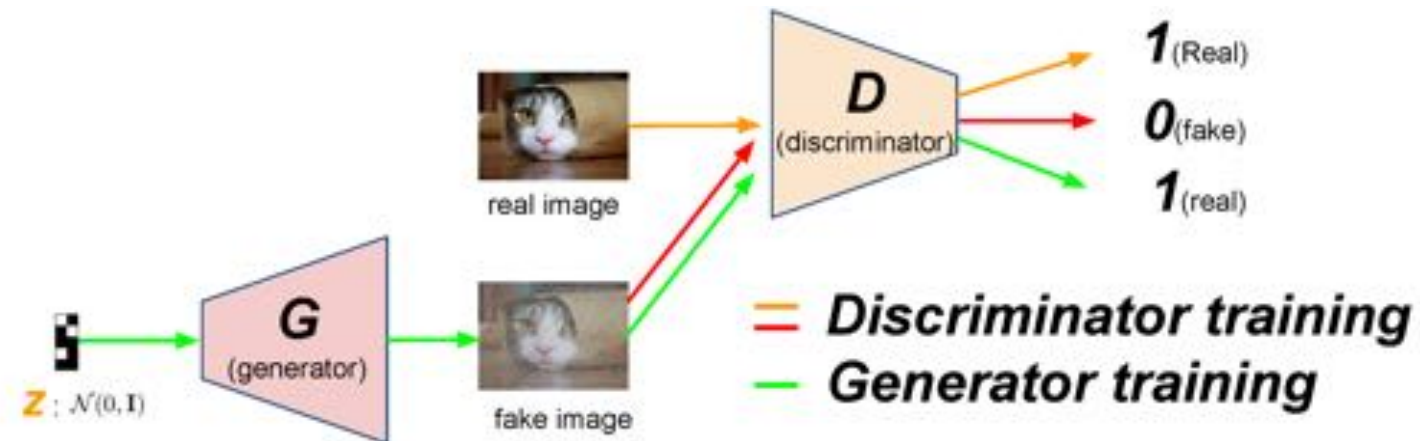
$$\min_G \mathcal{L}_G = \mathbb{E}_{\mathbf{x} \sim G(\mathbf{z}), \mathbf{z} \sim p(\mathbf{z})} [\log(1 - D(\mathbf{x}))].$$



Generative Adversarial Nets (GANs)

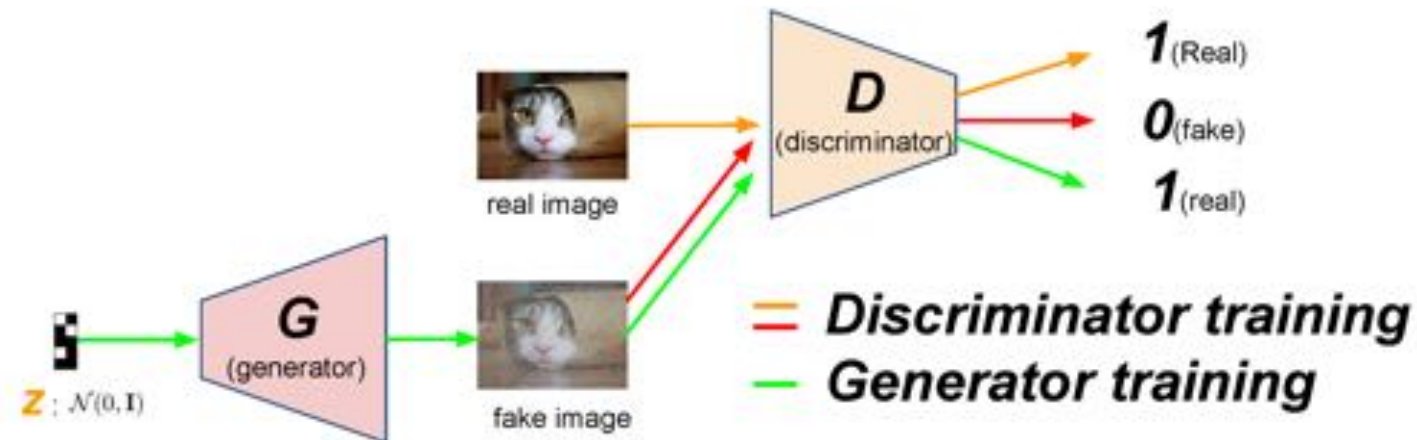
- Learning
 - Train G to fool the discriminator
 - The original loss suffers from vanishing gradients when D is too strong
 - Instead use the following in practice

$$\max_G \mathcal{L}_G = \mathbb{E}_{\mathbf{x} \sim G(\mathbf{z}), \mathbf{z} \sim p(\mathbf{z})} [\log D(\mathbf{x})]$$



Generative Adversarial Nets (GANs)

- Learning
 - Aim to achieve equilibrium of the game
 - Optimal state:
 - $p_g(\mathbf{x}) = p_{data}(\mathbf{x})$
 - $D(\mathbf{x}) = \frac{p_{data}(\mathbf{x})}{p_{data}(\mathbf{x}) + p_g(\mathbf{x})} = \frac{1}{2}$

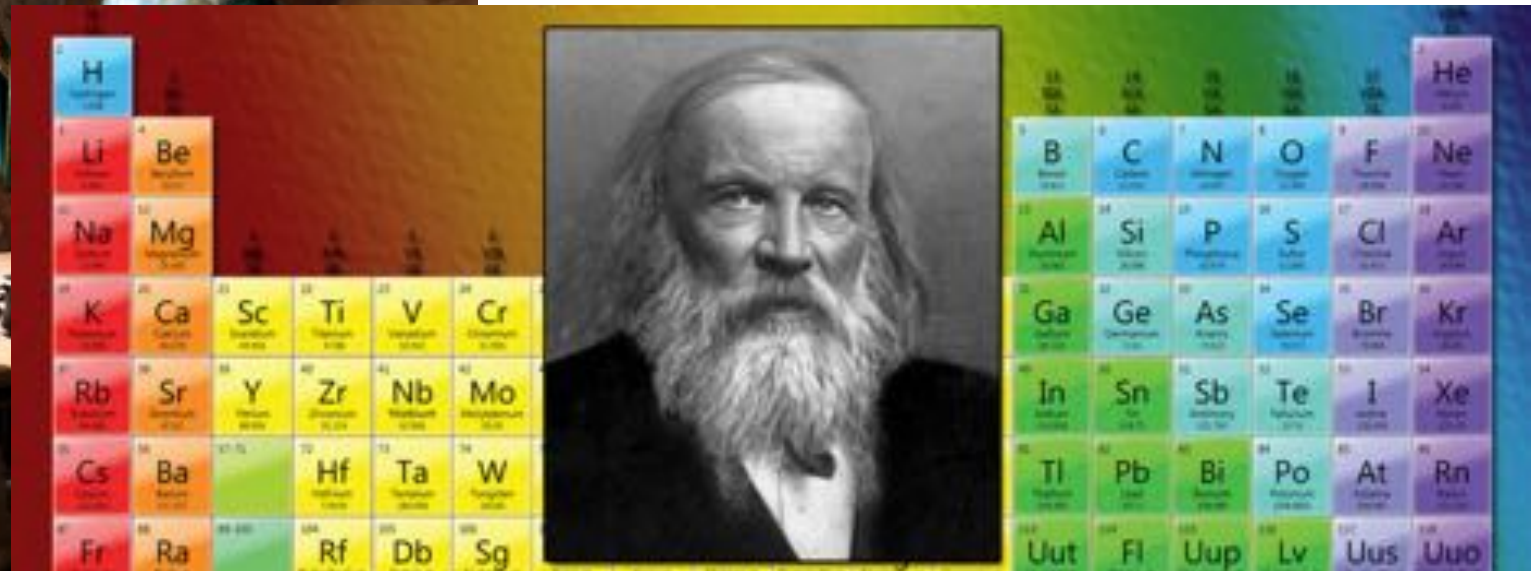
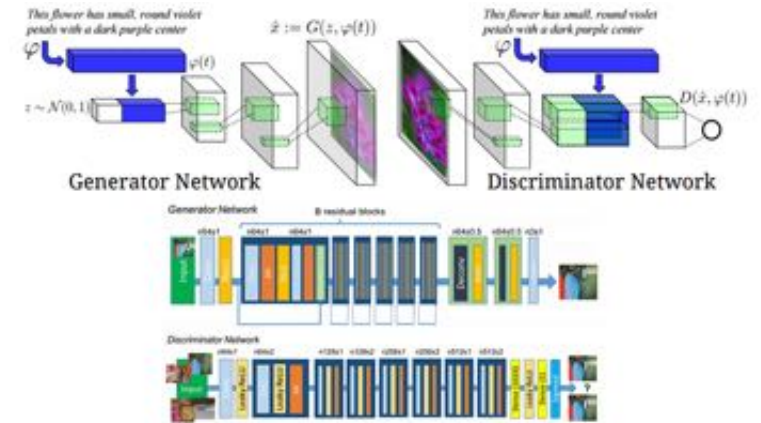
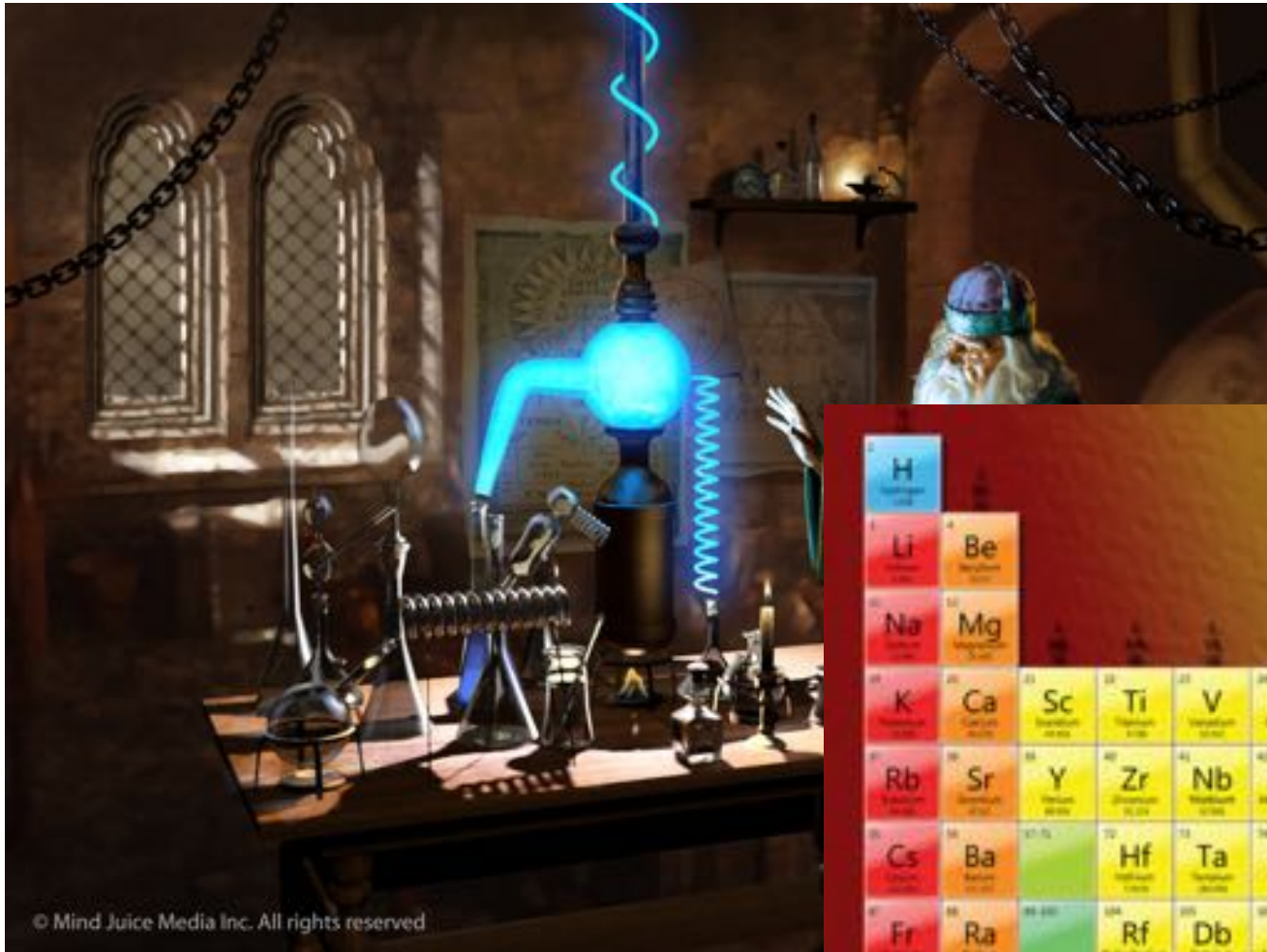


GANs: example results



Generated bedrooms [Radford et al., 2016]

Alchemy Vs Modern Chemistry



Outline

- Overview of advances in deep generative models
- Backgrounds of deep generative models
 - Wake sleep algorithm
 - Variational autoencoders
 - Generative adversarial networks
- A unified view of deep generative models
 - new formulations of deep generative models
 - Symmetric modeling of latent and visible variables

Z Hu, Z YANG, R Salakhutdinov, E Xing,
“On Unifying Deep Generative Models”, arxiv 1706.00550

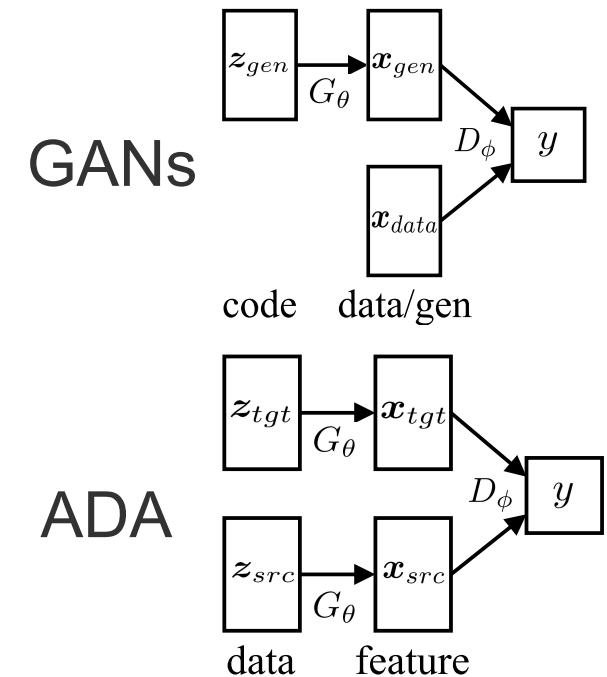
A unified view of deep generative models

- Literatures have viewed these DGM approaches as distinct model training paradigms
 - GANs: achieve an equilibrium between generator and discriminator
 - VAEs: maximize lower bound of the data likelihood
- Let's study a new formulation for DGMs
 - Connects GANs, VAEs, and other variants, under a unified view
 - Links them back to inference and learning of Graphical Models, and the wake-sleep heuristic that approximates this
 - Provides a tool to analyze many GAN-/VAE-based algorithms
 - Encourages mutual exchange of ideas from each individual class of models

Adversarial domain adaptation (ADA)

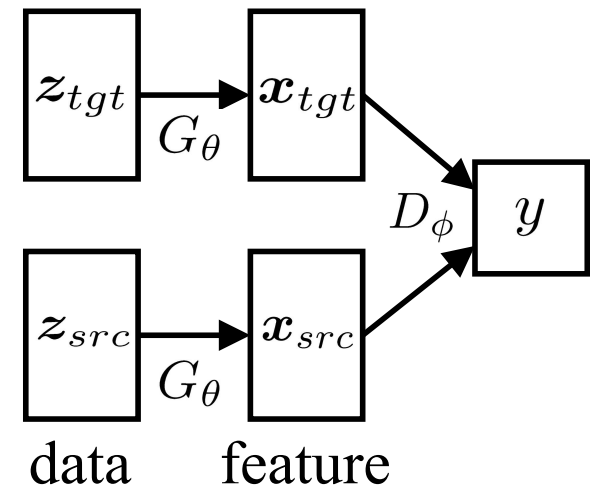
- Let's start from ADA
 - The application of adversarial approach on domain adaptation
 - We then show GANs can be seen as a special case of ADA
 - Correspondence of elements:

Elements	GANs	ADA
x	data/generation	features
z	code vector	Data from src/tgt domains
y	Real/fake indicator	Source/target domain indicator



Adversarial domain adaptation (ADA)

- Data z from two domains indicated by $y \in \{0,1\}$
 - Source domain ($y = 1$)
 - Target domain ($y = 0$)
- ADA transfers prediction knowledge learned from the source domain to the target domain
 - Learn a feature extractor $G_\theta: \mathbf{x} = G_\theta(\mathbf{z})$
 - Wants \mathbf{x} to be indistinguishable by a domain discriminator: $D_\phi(\mathbf{x})$
- Application in classification
 - E.g., we have labels of the source domain data
 - Train classifier over \mathbf{x} of source domain data to predict the labels
 - \mathbf{x} is domain invariant $\Rightarrow \mathbf{x}$ is predictive for target domain data



ADA: conventional formulation

- Train D_ϕ to distinguish between domains

$$\max_{\phi} \mathcal{L}_{\phi} = \mathbb{E}_{\mathbf{x}=G_{\theta}(\mathbf{z}), \mathbf{z} \sim p(\mathbf{z}|y=1)} [\log D_{\phi}(\mathbf{x})] + \mathbb{E}_{\mathbf{x}=G_{\theta}(\mathbf{z}), \mathbf{z} \sim p(\mathbf{z}|y=0)} [\log(1 - D_{\phi}(\mathbf{x}))]$$

- Train G_{θ} to fool D_{ϕ}

$$\max_{\theta} \mathcal{L}_{\theta} = \mathbb{E}_{\mathbf{x}=G_{\theta}(\mathbf{z}), \mathbf{z} \sim p(\mathbf{z}|y=1)} [\log(1 - D_{\phi}(\mathbf{x}))] + \mathbb{E}_{\mathbf{x}=G_{\theta}(\mathbf{z}), \mathbf{z} \sim p(\mathbf{z}|y=0)} [\log D_{\phi}(\mathbf{x})]$$

ADA: new formulation

- To reveal the connections to conventional variational approaches, let's rewrite the objectives in a format that resembles variational EM

- Implicit distribution over $\mathbf{x} \sim p_{\theta}(\mathbf{x}|y)$

$$\mathbf{x} = G_{\theta}(\mathbf{z}), \mathbf{z} \sim p(\mathbf{z}|y)$$

- Discriminator distribution $q_{\phi}(y|\mathbf{x})$

$$q_{\phi}^r(y|\mathbf{x}) = q_{\phi}(1 - y|\mathbf{x})$$

- Rewrite the objective in the new form (up to constant scale factor)

$$\max_{\phi} \mathcal{L}_{\phi} = \mathbb{E}_{p_{\theta}(\mathbf{x}|y)p(y)} [\log q_{\phi}(y|\mathbf{x})]$$

$$\max_{\theta} \mathcal{L}_{\theta} = \mathbb{E}_{p_{\theta}(\mathbf{x}|y)p(y)} [\log q_{\phi}^r(y|\mathbf{x})]$$

- \mathbf{z} is encapsulated in the implicit distribution $p_{\theta}(\mathbf{x}|y)$

$$\max_{\phi} \mathcal{L}_{\phi} = \mathbb{E}_{p_{\theta}(\mathbf{x}|y=0)p(y=0)} [\log q_{\phi}(y = 0|\mathbf{x})] + \mathbb{E}_{p_{\theta}(\mathbf{x}|y=1)p(y=1)} [\log q_{\phi}(y = 1|\mathbf{x})]$$

$$= \frac{1}{2} \mathbb{E}_{\mathbf{x}=G_{\theta}(\mathbf{z}), \mathbf{z} \sim p(\mathbf{z}|y=0)} [\log(1 - D_{\phi}(\mathbf{x}))] + \frac{1}{2} \mathbb{E}_{\mathbf{x}=G_{\theta}(\mathbf{z}), \mathbf{z} \sim p(\mathbf{z}|y=1)} [\log D_{\phi}(\mathbf{x})]$$

- (Ignore the constant scale factor 1/2)

ADA: new formulation

- New formulation

$$\max_{\phi} \mathcal{L}_{\phi} = \mathbb{E}_{p_{\theta}(\mathbf{x}|y)p(y)} [\log q_{\phi}(y|\mathbf{x})]$$

$$\max_{\theta} \mathcal{L}_{\theta} = \mathbb{E}_{p_{\theta}(\mathbf{x}|y)p(y)} [\log q_{\phi}^r(y|\mathbf{x})]$$

- The only **difference** between θ and ϕ : q vs. q^r
- This is where the **adversarial** mechanism comes about

ADA vs. Variational EM

Variational EM

- Objectives

$$\max_{\phi} \mathcal{L}_{\phi, \theta} = \mathbb{E}_{q_{\phi}(z|x)} [\log p_{\theta}(x|z)] + KL(q_{\phi}(z|x) || p(z))$$

$$\max_{\theta} \mathcal{L}_{\phi, \theta} = \mathbb{E}_{q_{\phi}(z|x)} [\log p_{\theta}(x|z)] + KL(q_{\phi}(z|x) || p(z))$$

- Single objective for both θ and ϕ
- Extra prior regularization by $p(z)$

ADA

- Objectives

$$\max_{\phi} \mathcal{L}_{\phi} = \mathbb{E}_{p_{\theta}(\mathbf{x}|y)p(y)} [\log q_{\phi}(y|\mathbf{x})]$$

$$\max_{\theta} \mathcal{L}_{\theta} = \mathbb{E}_{p_{\theta}(\mathbf{x}|y)p(y)} [\log q_{\phi}^r(y|\mathbf{x})]$$

- Two objectives
- Have global optimal state in the game theoretic view

ADA vs. Variational EM

Variational EM

- Objectives

$$\max_{\phi} \mathcal{L}_{\phi, \theta} = \mathbb{E}_{q_{\phi}(z|x)} [\log p_{\theta}(x|z)] + KL(q_{\phi}(z|x) || p(z))$$

$$\max_{\theta} \mathcal{L}_{\phi, \theta} = \mathbb{E}_{q_{\phi}(z|x)} [\log p_{\theta}(x|z)] + KL(q_{\phi}(z|x) || p(z))$$
 - Single objective for both θ and ϕ
 - Extra prior regularization by $p(z)$
- The reconstruction term: maximize the conditional log-likelihood of x with the generative distribution $p_{\theta}(x|z)$ conditioning on the latent code z inferred by $q_{\phi}(z|x)$



- $p_{\theta}(x|z)$ is the generative model
- $q_{\phi}(z|x)$ is the inference model

ADA

- Objectives

$$\max_{\phi} \mathcal{L}_{\phi} = \mathbb{E}_{p_{\theta}(x|y)p(y)} [\log q_{\phi}(y|x)]$$

$$\max_{\theta} \mathcal{L}_{\theta} = \mathbb{E}_{p_{\theta}(x|y)p(y)} [\log q_{\phi}^r(y|x)]$$
 - Two objectives
 - Have global optimal state in the game theoretic view
- The objectives: maximize the conditional log-likelihood of y (or $1 - y$) with the distribution $q_{\phi}(y|x)$ conditioning on latent feature x inferred by $p_{\theta}(x|y)$

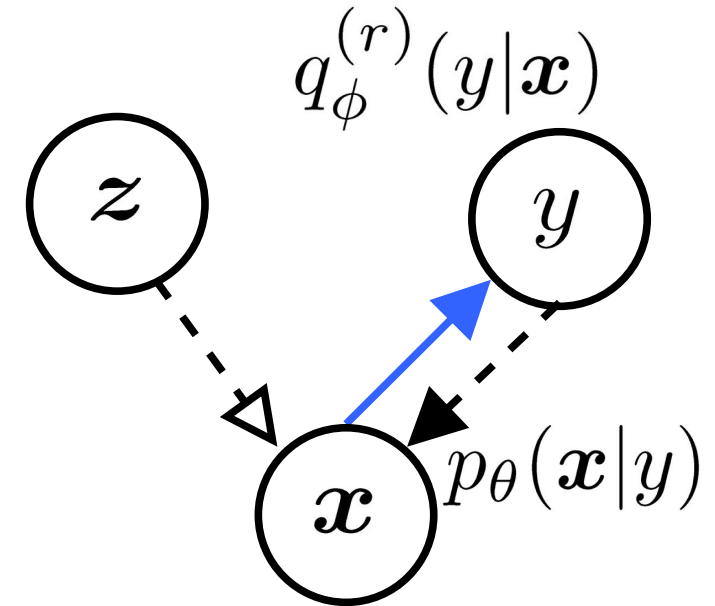


- Interpret $q_{\phi}(y|x)$ as the generative model
- Interpret $p_{\theta}(x|y)$ as the inference model

ADA: graphical model

Define:

- Solid-line arrows ($x \rightarrow y$):
 - generative process
- Dashed-line arrows ($y, z \rightarrow x$):
 - inference
- Hollow arrows ($z \rightarrow x$):
 - deterministic transformation
 - leading to implicit distributions
- Blue arrows ($x \rightarrow y$):
 - adversarial mechanism
 - involves both $q_\phi(y|\mathbf{x})$ and $q_\phi^r(y|\mathbf{x})$

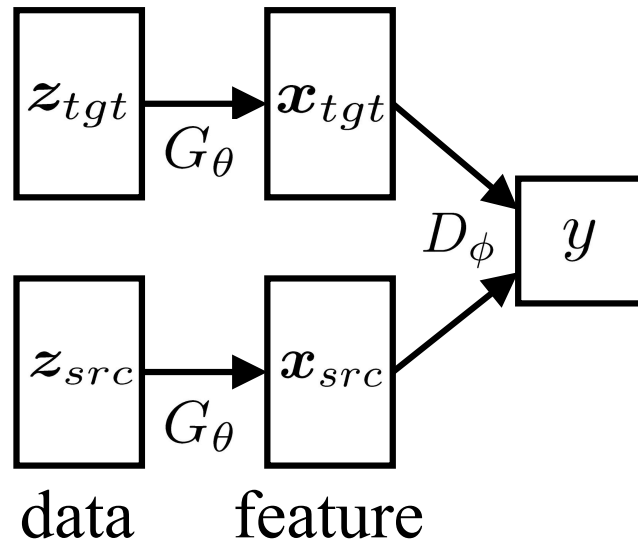


$$\max_{\phi} \mathcal{L}_{\phi} = \mathbb{E}_{p_{\theta}(\mathbf{x}|y)p(y)} [\log q_{\phi}(y|\mathbf{x})]$$

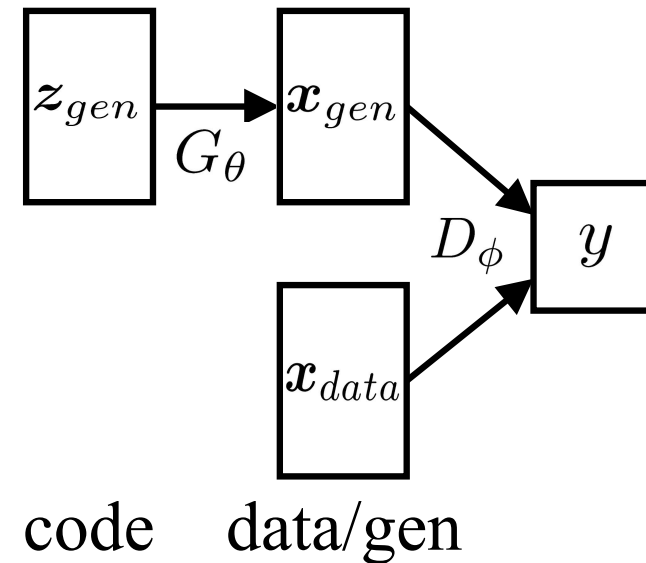
$$\max_{\theta} \mathcal{L}_{\theta} = \mathbb{E}_{p_{\theta}(\mathbf{x}|y)p(y)} [\log q_{\phi}^r(y|\mathbf{x})]$$

GANs: a variant of ADA

- Transfer the properties of source domain to target domain
 - Source domain: e.g. real image, $y = 1$
 - Target domain: e.g. generated image, $y = 0$



ADA



GANs

GANs: a variant of ADA

- Implicit distribution over $\mathbf{x} \sim p_{\theta}(\mathbf{x}|y)$

$$p_{\theta}(\mathbf{x}|y) = \begin{cases} p_{g_{\theta}}(\mathbf{x}) & y = 0 \\ p_{data}(\mathbf{x}) & y = 1. \end{cases} \begin{array}{l} \text{(distribution of generated images)} \\ \text{(distribution of real images)} \end{array}$$

- $\mathbf{x} \sim p_{g_{\theta}}(\mathbf{x}) \Leftrightarrow \mathbf{x} = G_{\theta}(\mathbf{z}), \mathbf{z} \sim p(\mathbf{z}|y = 0)$
- $\mathbf{x} \sim p_{data}(\mathbf{x})$
 - the code space of \mathbf{z} is degenerated
 - sample directly from data

GANs: new formulation

- Again, rewrite GAN objectives in the "variational-EM" format
- Recap: conventional formulation:

$$\max_{\phi} \mathcal{L}_{\phi} = \mathbb{E}_{\mathbf{x}=G_{\theta}(\mathbf{z}), \mathbf{z} \sim p(\mathbf{z}|\mathbf{y}=0)} [\log(1 - D_{\phi}(\mathbf{x}))] + \mathbb{E}_{\mathbf{x} \sim p_{data}(\mathbf{x})} [\log D_{\phi}(\mathbf{x})]$$

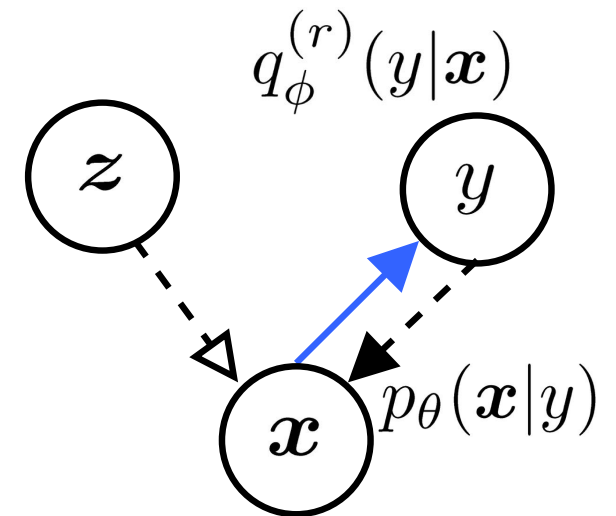
$$\begin{aligned} \max_{\theta} \mathcal{L}_{\theta} &= \mathbb{E}_{\mathbf{x}=G_{\theta}(\mathbf{z}), \mathbf{z} \sim p(\mathbf{z}|\mathbf{y}=0)} [\log D_{\phi}(\mathbf{x})] + \mathbb{E}_{\mathbf{x} \sim p_{data}(\mathbf{x})} [\log(1 - D_{\phi}(\mathbf{x}))] \\ &= \mathbb{E}_{\mathbf{x}=G_{\theta}(\mathbf{z}), \mathbf{z} \sim p(\mathbf{z}|\mathbf{y}=0)} [\log D_{\phi}(\mathbf{x})] \end{aligned}$$

- Rewrite in the new form

$$\max_{\phi} \mathcal{L}_{\phi} = \mathbb{E}_{p_{\theta}(\mathbf{x}|\mathbf{y})p(\mathbf{y})} [\log q_{\phi}(\mathbf{y}|\mathbf{x})]$$

$$\max_{\theta} \mathcal{L}_{\theta} = \mathbb{E}_{p_{\theta}(\mathbf{x}|\mathbf{y})p(\mathbf{y})} [\log q_{\phi}^r(\mathbf{y}|\mathbf{x})]$$

- Exact the same with ADA !
- The same correspondence to variational EM !



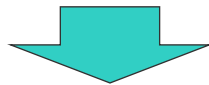
GANs vs. Variational EM

Variational EM

- Objectives

$$\max_{\phi} \mathcal{L}_{\phi, \theta} = \mathbb{E}_{q_{\phi}(z|x)} [\log p_{\theta}(x|z)] + KL(q_{\phi}(z|x) || p(z))$$

$$\max_{\theta} \mathcal{L}_{\phi, \theta} = \mathbb{E}_{q_{\phi}(z|x)} [\log p_{\theta}(x|z)] + KL(q_{\phi}(z|x) || p(z))$$
 - Single objective for both θ and ϕ
 - Extra prior regularization by $p(z)$
- The reconstruction term: maximize the conditional log-likelihood of x with the generative distribution $p_{\theta}(x|z)$ conditioning on the latent code z inferred by $q_{\phi}(z|x)$



- $p_{\theta}(x|z)$ is the generative model
- $q_{\phi}(z|x)$ is the inference model

GAN

- Objectives

$$\max_{\phi} \mathcal{L}_{\phi} = \mathbb{E}_{p_{\theta}(x|y)p(y)} [\log q_{\phi}(y|x)]$$

$$\max_{\theta} \mathcal{L}_{\theta} = \mathbb{E}_{p_{\theta}(x|y)p(y)} [\log q_{\phi}^r(y|x)]$$
 - Two objectives
 - Have global optimal state in the game theoretic view
- The objectives: maximize the conditional log-likelihood of y (or $1 - y$) with the distribution $q_{\phi}(y|x)$ conditioning on data/generation x inferred by $p_{\theta}(x|y)$



- Interpret $q_{\phi}(y|x)$ as the generative model
- Interpret $p_{\theta}(x|y)$ as the inference model

GANs vs. Variational EM

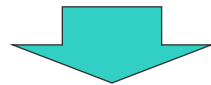
- Interpret x as latent variables
- Interpret generation of x as performing inference over latent

Variational EM

- Objectives

$$\max_{\phi} \mathcal{L}_{\phi, \theta} = \mathbb{E}_{q_{\phi}(z|x)} [\log p_{\theta}(x|z)] + KL(q_{\phi}(z|x) || p(z))$$

$$\max_{\theta} \mathcal{L}_{\phi, \theta} = \mathbb{E}_{q_{\phi}(z|x)} [\log p_{\theta}(x|z)] + KL(q_{\phi}(z|x) || p(z))$$
 - Single objective for both θ and ϕ
 - Extra prior regularization by $p(z)$
- The reconstruction term: maximize the conditional log-likelihood of x with the generative distribution $p_{\theta}(x|z)$ conditioning on the latent code z inferred by $q_{\phi}(z|x)$



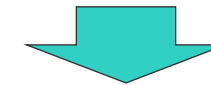
- $p_{\theta}(x|z)$ is the generative model
- $q_{\phi}(z|x)$ is the inference model

GAN

- Objectives

$$\max_{\phi} \mathcal{L}_{\phi} = \mathbb{E}_{p_{\theta}(x|y)p(y)} [\log q_{\phi}(y|x)]$$

$$\max_{\theta} \mathcal{L}_{\theta} = \mathbb{E}_{p_{\theta}(x|y)p(y)} [\log q_{\phi}^r(y|x)]$$
 - Two objectives
 - Have global optimal state in the game theoretic view
- The objectives: maximize the conditional log-likelihood of y (or $1 - y$) with the distribution $q_{\phi}(y|x)$ conditioning on data/generation x inferred by $p_{\theta}(x|y)$



- Interpret $q_{\phi}(y|x)$ as the generative model
- Interpret $p_{\theta}(x|y)$ as the inference model

GANs: minimizing KLD

- As in Variational EM, we can further rewrite in the form of **minimizing KLD** to reveal more insights into the optimization problem
- For each optimization step of $p_{\theta}(\mathbf{x}|y)$ at point $(\theta = \theta_0, \phi = \phi_0)$, let
 - $p(y)$: uniform prior distribution
 - $p_{\theta=\theta_0}(\mathbf{x}) = \mathbb{E}_{p(y)}[p_{\theta=\theta_0}(\mathbf{x}|y)]$
 - $q^r(\mathbf{x}|y) \propto q_{\phi=\phi_0}^r(y|\mathbf{x})p_{\theta=\theta_0}(\mathbf{x})$

- **Lemma 1:** The updates of θ at θ_0 have

$$\nabla_{\theta} \left[- \mathbb{E}_{p_{\theta}(\mathbf{x}|y)p(y)} [\log q_{\phi=\phi_0}^r(y|\mathbf{x})] \right] \Big|_{\theta=\theta_0} =$$

$$\nabla_{\theta} \left[\mathbb{E}_{p(y)} [KL(p_{\theta}(\mathbf{x}|y) \| q^r(\mathbf{x}|y))] - JSD(p_{\theta}(\mathbf{x}|y=0) \| p_{\theta}(\mathbf{x}|y=1)) \right] \Big|_{\theta=\theta_0}$$

- KL: KL divergence
- JSD: Jensen-shannon divergence

Proof of Lemma 1

Proof.

$$\begin{aligned} \mathbb{E}_{p_{\theta}(\mathbf{x}|y)p(y)} [\log q^r(y|\mathbf{x})] = \\ - \mathbb{E}_{p(y)} [\mathbf{KL}(p_{\theta}(\mathbf{x}|y) \| q^r(\mathbf{x}|y)) - \mathbf{KL}(p_{\theta}(\mathbf{x}|y) \| p_{\theta_0}(\mathbf{x}))], \end{aligned} \quad (3)$$

where

$$\begin{aligned} \mathbb{E}_{p(y)} [\mathbf{KL}(p_{\theta}(\mathbf{x}|y) \| p_{\theta_0}(\mathbf{x}))] \\ = p(y=0) \cdot \mathbf{KL} \left(p_{\theta}(\mathbf{x}|y=0) \| \frac{p_{\theta_0}(\mathbf{x}|y=0) + p_{\theta_0}(\mathbf{x}|y=1)}{2} \right) \\ + p(y=1) \cdot \mathbf{KL} \left(p_{\theta}(\mathbf{x}|y=1) \| \frac{p_{\theta_0}(\mathbf{x}|y=0) + p_{\theta_0}(\mathbf{x}|y=1)}{2} \right). \end{aligned} \quad (4)$$

Note that $p_{\theta}(\mathbf{x}|y=0) = p_{g_{\theta}}(\mathbf{x})$, and $p_{\theta}(\mathbf{x}|y=1) = p_{data}(\mathbf{x})$. Let $p_{M_{\theta}} = \frac{p_{g_{\theta}} + p_{data}}{2}$. Eq.(4) can be simplified as:

$$\mathbb{E}_{p(y)} [\mathbf{KL}(p_{\theta}(\mathbf{x}|y) \| p_{\theta_0}(\mathbf{x}))] = \frac{1}{2} \mathbf{KL}(p_{g_{\theta}} \| p_{M_{\theta_0}}) + \frac{1}{2} \mathbf{KL}(p_{data} \| p_{M_{\theta_0}}). \quad (5)$$

Proof of Lemma 1 (cont.)

On the other hand,

$$\begin{aligned}
 \text{JSD}(p_{g\theta} \| p_{data}) &= \frac{1}{2} \mathbb{E}_{p_{g\theta}} \left[\log \frac{p_{g\theta}}{p_{M\theta}} \right] + \frac{1}{2} \mathbb{E}_{p_{data}} \left[\log \frac{p_{data}}{p_{M\theta}} \right] \\
 &= \frac{1}{2} \mathbb{E}_{p_{g\theta}} \left[\log \frac{p_{g\theta}}{p_{M_{\theta_0}}} \right] + \frac{1}{2} \mathbb{E}_{p_{g\theta}} \left[\log \frac{p_{M_{\theta_0}}}{p_{M\theta}} \right] \\
 &\quad + \frac{1}{2} \mathbb{E}_{p_{data}} \left[\log \frac{p_{data}}{p_{M_{\theta_0}}} \right] + \frac{1}{2} \mathbb{E}_{p_{data}} \left[\log \frac{p_{M_{\theta_0}}}{p_{M\theta}} \right] \tag{6} \\
 &= \frac{1}{2} \mathbb{E}_{p_{g\theta}} \left[\log \frac{p_{g\theta}}{p_{M_{\theta_0}}} \right] + \frac{1}{2} \mathbb{E}_{p_{data}} \left[\log \frac{p_{data}}{p_{M_{\theta_0}}} \right] + \mathbb{E}_{p_{M\theta}} \left[\log \frac{p_{M_{\theta_0}}}{p_{M\theta}} \right] \\
 &= \frac{1}{2} \text{KL}(p_{g\theta} \| p_{M_{\theta_0}}) + \frac{1}{2} \text{KL}(p_{data} \| p_{M_{\theta_0}}) - \text{KL}(p_{M\theta} \| p_{M_{\theta_0}}).
 \end{aligned}$$

Note that

$$\nabla_{\theta} \text{KL}(p_{M\theta} \| p_{M_{\theta_0}}) |_{\theta=\theta_0} = 0. \tag{7}$$

Taking derivatives of Eq.(5) w.r.t θ at θ_0 we get

$$\begin{aligned}
 &\nabla_{\theta} \mathbb{E}_{p(y)} [\text{KL}(p_{\theta}(\mathbf{x}|y) \| p_{\theta_0}(\mathbf{x}))] |_{\theta=\theta_0} \\
 &= \nabla_{\theta} \left(\frac{1}{2} \text{KL}(p_{g\theta} \| p_{M_{\theta_0}}) |_{\theta=\theta_0} + \frac{1}{2} \text{KL}(p_{data} \| p_{M_{\theta_0}}) \right) |_{\theta=\theta_0} \tag{8} \\
 &= \nabla_{\theta} \text{JSD}(p_{g\theta} \| p_{data}) |_{\theta=\theta_0}.
 \end{aligned}$$

Taking derivatives of the both sides of Eq.(3) at w.r.t θ at θ_0 and plugging the last equation of Eq.(8), we obtain the desired results. \square

GANs: minimizing KLD

- **Lemma 1:** The updates of θ at θ_0 have

$$\nabla_{\theta} \left[-\mathbb{E}_{p_{\theta}(\mathbf{x}|y)p(y)} [\log q_{\phi=\phi_0}^r(y|\mathbf{x})] \right] \Big|_{\theta=\theta_0} =$$

$$\nabla_{\theta} \left[\mathbb{E}_{p(y)} [\mathbf{KL}(p_{\theta}(\mathbf{x}|y) \| q^r(\mathbf{x}|y))] - \mathbf{JSD}(p_{\theta}(\mathbf{x}|y=0) \| p_{\theta}(\mathbf{x}|y=1)) \right] \Big|_{\theta=\theta_0}$$

- Connection to variational inference
 - See \mathbf{x} as latent variables, y as visible
 - $p_{\theta=\theta_0}(\mathbf{x})$: prior distribution
 - $q^r(\mathbf{x}|y) \propto q_{\phi=\phi_0}^r(y|\mathbf{x})p_{\theta=\theta_0}(\mathbf{x})$: posterior distribution
 - $p_{\theta}(\mathbf{x}|y)$: variational distribution
 - Amortized inference: updates model parameter θ
- Suggests relations to VAEs, as we will explore shortly

GANs: minimizing KLD

- **Lemma 1:** The updates of θ at θ_0 have

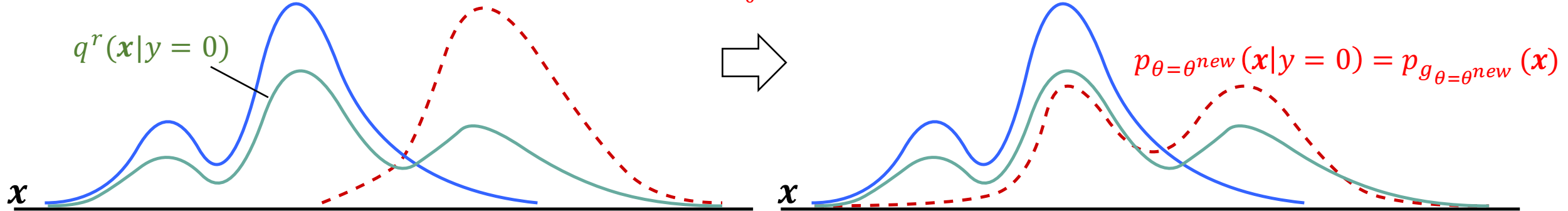
$$\nabla_{\theta} \left[-\mathbb{E}_{p_{\theta}(\mathbf{x}|y)p(y)} [\log q_{\phi=\phi_0}^r(y|\mathbf{x})] \right] \Big|_{\theta=\theta_0} =$$

$$\nabla_{\theta} \left[\mathbb{E}_{p(y)} [\text{KL}(p_{\theta}(\mathbf{x}|y) || q^r(\mathbf{x}|y))] - \text{JSD}(p_{\theta}(\mathbf{x}|y=0) || p_{\theta}(\mathbf{x}|y=1)) \right] \Big|_{\theta=\theta_0}$$

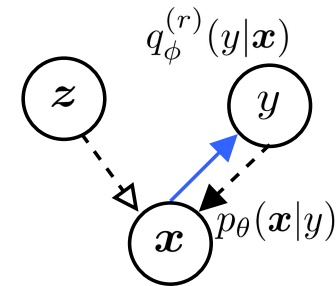
- Minimizing the KLD drives $p_{g_{\theta}}(\mathbf{x})$ to $p_{data}(\mathbf{x})$
 - By definition: $p_{\theta=\theta_0}(\mathbf{x}) = \mathbb{E}_{p(y)} [p_{\theta=\theta_0}(\mathbf{x}|y)] = (p_{g_{\theta=\theta_0}}(\mathbf{x}) + p_{data}(\mathbf{x})) / 2$
 - $\text{KL}(p_{\theta}(x|y=1) || q^r(x|y=1)) = \text{KL}(p_{data}(x) || q^r(x|y=1))$: constant, no free parameters
 - $\text{KL}(p_{\theta}(x|y=0) || q^r(x|y=0)) = \text{KL}(p_{g_{\theta}}(x) || q^r(x|y=0))$: parameter θ to optimize
 - $q^r(\mathbf{x}|y=0) \propto q_{\phi=\phi_0}^r(y=0|\mathbf{x})p_{\theta=\theta_0}(\mathbf{x})$
 - seen as a mixture of $p_{g_{\theta=\theta_0}}(\mathbf{x})$ and $p_{data}(\mathbf{x})$
 - mixing weights induced from $q_{\phi=\phi_0}^r(y=0|\mathbf{x})$
 - Drives $p_{g_{\theta}}(\mathbf{x}|y)$ to mixture of $p_{g_{\theta=\theta_0}}(\mathbf{x})$ and $p_{data}(\mathbf{x})$
 \Rightarrow Drives $p_{g_{\theta}}(\mathbf{x})$ to $p_{data}(\mathbf{x})$

GANs: minimizing KLD

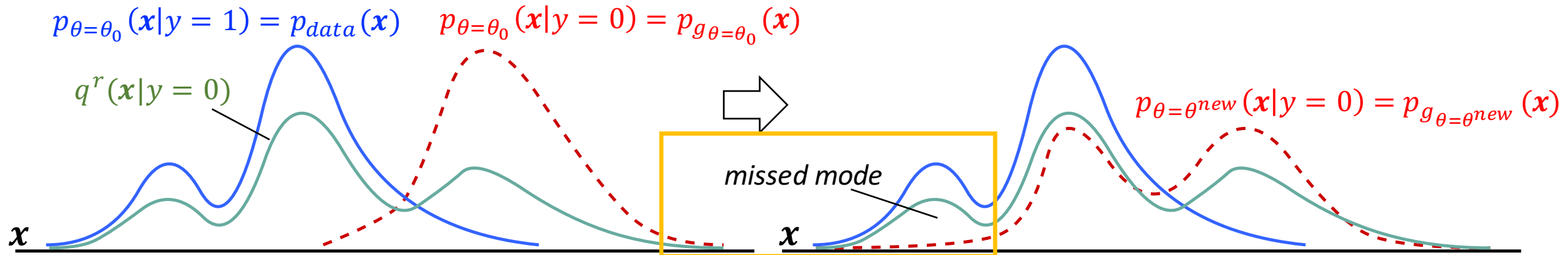
$$p_{\theta=\theta_0}(\mathbf{x}|y=1) = p_{data}(\mathbf{x}) \quad p_{\theta=\theta_0}(\mathbf{x}|y=0) = p_{g_{\theta=\theta_0}}(\mathbf{x})$$



- Minimizing the KLD drives $p_{g_{\theta}}(\mathbf{x})$ to $p_{data}(\mathbf{x})$
 - By definition: $p_{\theta=\theta_0}(\mathbf{x}) = E_{p(y)}[p_{\theta=\theta_0}(\mathbf{x}|y)] = (p_{g_{\theta=\theta_0}}(\mathbf{x}) + p_{data}(\mathbf{x})) / 2$
 - $KL(p_{\theta}(x|y=1) || q^r(x|y=1)) = KL(p_{data}(x) || q^r(x|y=1))$: constant, no free parameters
 - $KL(p_{\theta}(x|y=0) || q^r(x|y=0)) = KL(p_{g_{\theta}}(x) || q^r(x|y=0))$: parameter θ to optimize
 - $q^r(\mathbf{x}|y=0) \propto q_{\phi=\phi_0}^r(y=0|\mathbf{x})p_{\theta=\theta_0}(\mathbf{x})$
 - seen as a mixture of $p_{g_{\theta=\theta_0}}(\mathbf{x})$ and $p_{data}(\mathbf{x})$
 - mixing weights induced from $q_{\phi=\phi_0}^r(y=0|\mathbf{x})$
 - Drives $p_{g_{\theta}}(\mathbf{x}|y)$ to mixture of $p_{g_{\theta=\theta_0}}(\mathbf{x})$ and $p_{data}(\mathbf{x})$
- ⇒ Drives $p_{g_{\theta}}(\mathbf{x})$ to $p_{data}(\mathbf{x})$



GANs: minimizing KLD



- Missing mode phenomena of GANs
 - Asymmetry of KLD
 - Concentrates $p_{\theta}(x|y=0)$ to large modes of $q^r(x|y)$
 - $\Rightarrow p_{g_{\theta}}(x)$ misses modes of $p_{data}(x)$
 - Symmetry of JSD
 - Does not affect the behavior of mode missing

$$\begin{aligned}
 & \text{KL}(p_{g_{\theta}}(x) || q^r(x|y=0)) \\
 &= \int p_{g_{\theta}}(x) \log \frac{p_{g_{\theta}}(x)}{q^r(x|y=0)} dx
 \end{aligned}$$

- Large positive contribution to the KLD in the regions of x space where $q^r(x|y=0)$ is small, unless $p_{g_{\theta}}(x)$ is also small
- $\Rightarrow p_{g_{\theta}}(x)$ tends to avoid regions where $q^r(x|y=0)$ is small

GANs: minimizing KLD

- **Lemma 1:** The updates of θ at θ_0 have

$$\nabla_{\theta} \left[-\mathbb{E}_{p_{\theta}(\mathbf{x}|y)p(y)} [\log q_{\phi_0}^r(y|\mathbf{x})] \right] \Big|_{\theta=\theta_0} =$$

$$\nabla_{\theta} \left[\mathbb{E}_{p(y)} [\text{KL}(p_{\theta}(\mathbf{x}|y) \| q^r(\mathbf{x}|y))] - \text{JSD}(p_{\theta}(\mathbf{x}|y=0) \| p_{\theta}(\mathbf{x}|y=1)) \right] \Big|_{\theta=\theta_0}$$

- No assumption on optimal discriminator $q_{\phi_0}^r(y|\mathbf{x})$
 - Previous results usually rely on (near) optimal discriminator
 - $q^*(y=1|\mathbf{x}) = p_{data}(\mathbf{x}) / (p_{data}(\mathbf{x}) + p_g(\mathbf{x}))$
 - Optimality assumption is impractical: limited expressiveness of D_{ϕ} [Arora et al 2017]
 - Our result is a generalization of the previous theorem [Arjovsky & Bottou 2017]
 - Plug the optimal discriminator into the above equation, we recover the theorem

$$\nabla_{\theta} \left[-\mathbb{E}_{p_{\theta}(\mathbf{x}|y)p(y)} [\log q_{\phi_0}^r(y|\mathbf{x})] \right] \Big|_{\theta=\theta_0} = \nabla_{\theta} \left[\frac{1}{2} \text{KL}(p_{g_{\theta}} \| p_{data}) - \text{JSD}(p_{g_{\theta}} \| p_{data}) \right] \Big|_{\theta=\theta_0}$$

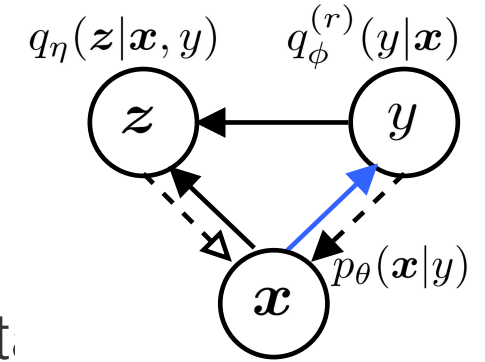
- Give insights on the generator training when discriminator is optimal

GANs: minimizing KLD

In summary:

- Reveal connection to variational inference
 - Build connections to VAEs (slides soon)
 - Inspire new model variants based on the connections
- Offer insights into the generator training
 - Formal explanation of the missing mode behavior of GANs
 - Still hold when the discriminator does not achieve its optimum at each iteration

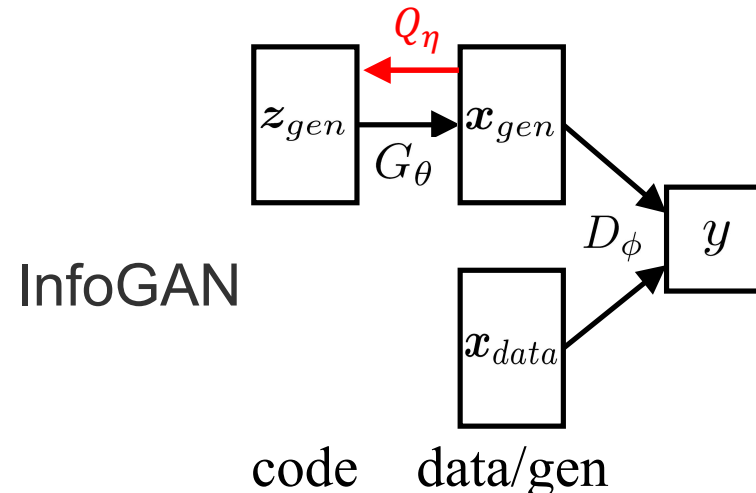
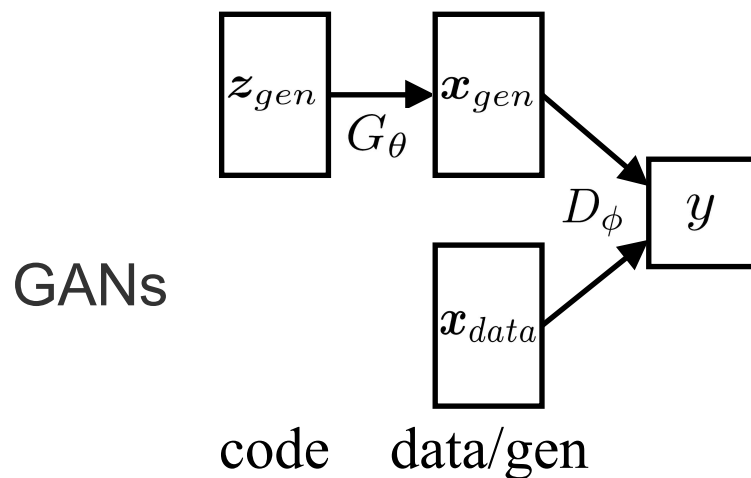
Variant of GAN: InfoGAN



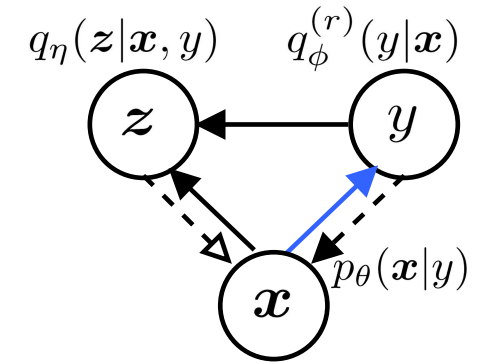
- GANs don't offer the functionality of inferring code \mathbf{z} given data.
- InfoGAN [Chen et al., 2016]
 - Introduce inference model $Q_\eta(\mathbf{z}|\mathbf{x})$ with parameters η
 - Augment the objectives of GANs by additionally inferring \mathbf{z}

$$\max_D \mathcal{L}_D = \mathbb{E}_{\mathbf{x} \sim p_{data}(\mathbf{x})} [\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{x} \sim G(\mathbf{z}), \mathbf{z} \sim p(\mathbf{z})} [\log(1 - D(\mathbf{x}))],$$

$$\max_{G, Q} \mathcal{L}_{G, Q} = \mathbb{E}_{\mathbf{x} \sim G(\mathbf{z}), \mathbf{z} \sim p(\mathbf{z})} [\log D(\mathbf{x}) + \log Q(\mathbf{z}|\mathbf{x})].$$



InfoGAN: new formulation

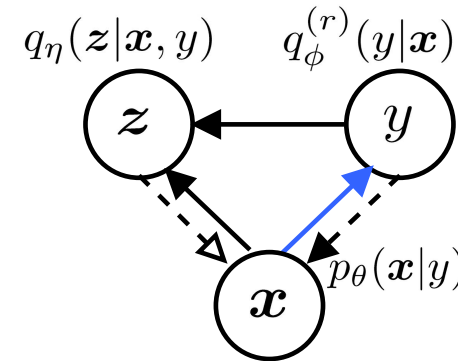
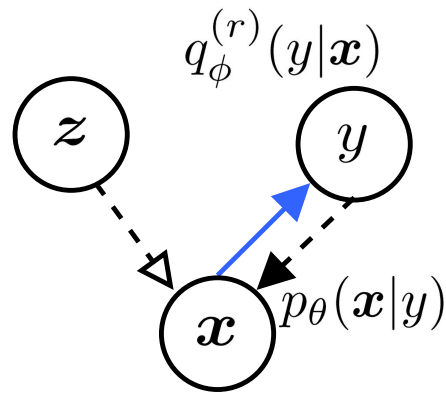
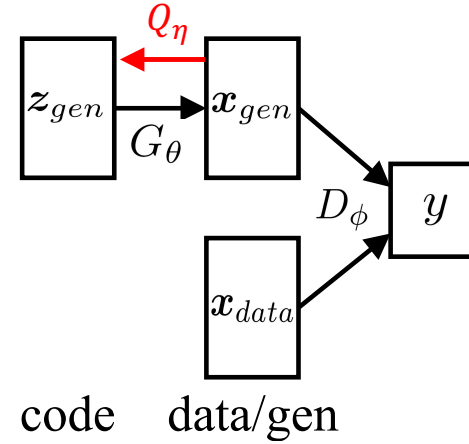
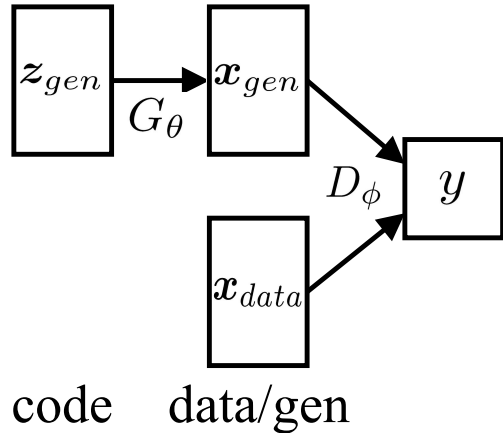


- Defines conditional $q_\eta(\mathbf{z}|\mathbf{x}, y)$
 - $q_\eta(\mathbf{z}|\mathbf{x}, y = 1)$ is fixed without free parameters to learn
 - As GANs assume the code space of real data is degenerated
 - Parameters η are only associated with $q_\eta(\mathbf{z}|\mathbf{x}, y = 0)$
- Rewrite in the new form:

$$\max_{\phi} \mathcal{L}_{\phi} = \mathbb{E}_{p_{\theta}(\mathbf{x}|y)p(y)} [\log q_{\eta}(\mathbf{z}|\mathbf{x}, y)q_{\phi}(y|\mathbf{x})]$$

$$\max_{\theta, \eta} \mathcal{L}_{\theta, \eta} = \mathbb{E}_{p_{\theta}(\mathbf{x}|y)p(y)} [\log q_{\eta}(\mathbf{z}|\mathbf{x}, y)q_{\phi}^r(y|\mathbf{x})]$$

GANs vs InfoGAN



$$\max_{\phi} \mathcal{L}_{\phi} = \mathbb{E}_{p_{\theta}(\mathbf{x}|y)p(y)} [\log q_{\phi}(y|\mathbf{x})]$$

$$\max_{\theta} \mathcal{L}_{\theta} = \mathbb{E}_{p_{\theta}(\mathbf{x}|y)p(y)} [\log q_{\phi}^r(y|\mathbf{x})]$$

$$\max_{\phi} \mathcal{L}_{\phi} = \mathbb{E}_{p_{\theta}(\mathbf{x}|y)p(y)} [\log q_{\eta}(z|\mathbf{x}, y)q_{\phi}(y|\mathbf{x})]$$

$$\max_{\theta, \eta} \mathcal{L}_{\theta, \eta} = \mathbb{E}_{p_{\theta}(\mathbf{x}|y)p(y)} [\log q_{\eta}(z|\mathbf{x}, y)q_{\phi}^r(y|\mathbf{x})]$$

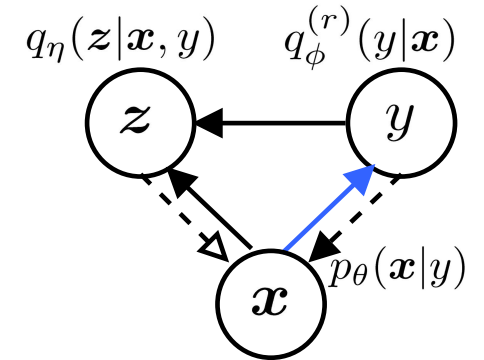
InfoGAN: new formulation

- Similar results as in GANs hold:
 - Let $q^r(\mathbf{x}|\mathbf{z}, y) \propto q_{\eta=\eta_0}(\mathbf{z}|\mathbf{x}, y)q_{\phi=\phi_0}^r(y|\mathbf{x})p_{\theta=\theta_0}(\mathbf{x})$
 - We have:

$$\nabla_{\theta} \left[- \mathbb{E}_{p_{\theta}(\mathbf{x}|y)p(y)} \left[\log q_{\eta_0}(\mathbf{z}|\mathbf{x}, y)q_{\phi_0}^r(y|\mathbf{x}) \right] \right] \Big|_{\theta=\theta_0} =$$

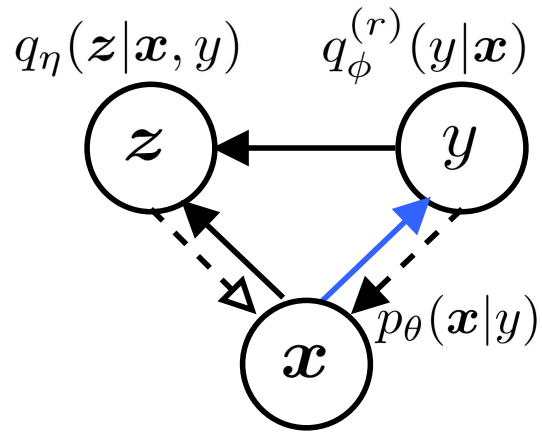
$$\nabla_{\theta} \left[\mathbb{E}_{p(y)} \left[\text{KL} (p_{\theta}(\mathbf{x}|y) \| q^r(\mathbf{x}|\mathbf{z}, y)) \right] - \text{JSD} (p_{\theta}(\mathbf{x}|y = 0) \| p_{\theta}(\mathbf{x}|y = 1)) \right] \Big|_{\theta=\theta_0}$$

- Next we show correspondences between GANs/InfoGAN and VAEs



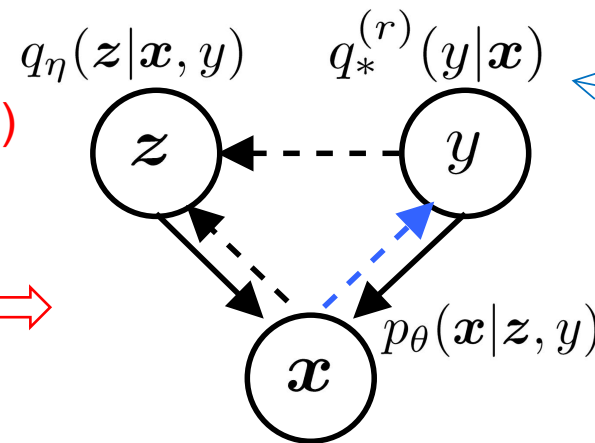
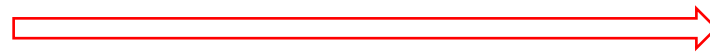
Relates VAEs with GANs

- Resemblance of GAN generator learning to variational inference
 - Suggest strong relations between VAEs and GANs
- Indeed, VAEs are basically minimizing **KLD with an opposite direction**, and with **a degenerated adversarial discriminator**



InfoGAN

swap the generation (solid-line) and inference (dashed-line) processes of InfoGAN



VAEs

degenerated discriminator

Recap: conventional formulation of VAEs

- Objective:

$$\max_{\theta, \eta} \mathcal{L}_{\theta, \eta}^{\text{vae}} = \mathbb{E}_{p_{data}(\mathbf{x})} \left[\mathbb{E}_{\tilde{q}_{\eta}(\mathbf{z}|\mathbf{x})} [\log \tilde{p}_{\theta}(\mathbf{x}|\mathbf{z})] - \text{KL}(\tilde{q}_{\eta}(\mathbf{z}|\mathbf{x}) \parallel \tilde{p}(\mathbf{z})) \right]$$

- $\tilde{p}(\mathbf{z})$: prior over \mathbf{z}
 - $\tilde{p}_{\theta}(\mathbf{x}|\mathbf{z})$: generative model
 - $\tilde{q}_{\eta}(\mathbf{z}|\mathbf{x})$: inference model
 - Only uses real examples from $p_{data}(\mathbf{x})$, lacks adversarial mechanism
- To align with GANs, let's introduce the real/fake indicator y and adversarial discriminator

VAEs: new formulation

- Assume a *perfect* discriminator $q_*(y|\mathbf{x})$
 - $q_*(y = 1|\mathbf{x}) = 1$ if \mathbf{x} is real examples
 - $q_*(y = 0|\mathbf{x}) = 1$ if \mathbf{x} is generated samples
 - $q_*^r(y|\mathbf{x}) := q_*(1 - y|\mathbf{x})$

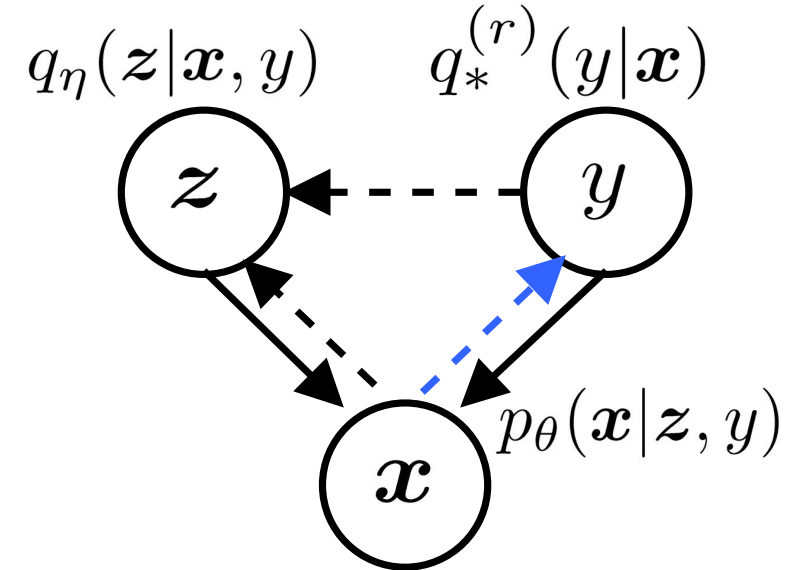
- Generative distribution

$$p_\theta(\mathbf{x}|\mathbf{z}, y) = \begin{cases} p_\theta(\mathbf{x}|\mathbf{z}) & y = 0 \\ p_{data}(\mathbf{x}) & y = 1. \end{cases}$$

- Let $p_\theta(\mathbf{z}, y|\mathbf{x}) \propto p_\theta(\mathbf{x}|\mathbf{z}, y)p(\mathbf{z}|y)p(y)$

- **Lemma 2**

$$\begin{aligned} \mathcal{L}_{\theta, \eta}^{vae} &= 2 \cdot \mathbb{E}_{p_{\theta_0}(\mathbf{x})} \left[\mathbb{E}_{q_\eta(\mathbf{z}|\mathbf{x}, y)q_*^r(y|\mathbf{x})} [\log p_\theta(\mathbf{x}|\mathbf{z}, y)] - KL(q_\eta(\mathbf{z}|\mathbf{x}, y)q_*^r(y|\mathbf{x}) || p(\mathbf{z}|y)p(y)) \right] \\ &= 2 \cdot \mathbb{E}_{p_{\theta_0}(\mathbf{x})} [-KL(q_\eta(\mathbf{z}|\mathbf{x}, y)q_*^r(y|\mathbf{x}) || p_\theta(\mathbf{z}, y|\mathbf{x}))]. \end{aligned}$$



Lemma 2: sketch of proof

- **Lemma 2**

$$\begin{aligned} \mathcal{L}_{\theta, \eta}^{vae} &= 2 \cdot \mathbb{E}_{p_{\theta_0}(\mathbf{x})} \left[\mathbb{E}_{q_{\eta}(\mathbf{z}|\mathbf{x}, y) q_*^r(y|\mathbf{x})} [\log p_{\theta}(\mathbf{x}|\mathbf{z}, y)] - KL(q_{\eta}(\mathbf{z}|\mathbf{x}, y) q_*^r(y|\mathbf{x}) \| p(\mathbf{z}|y)p(y)) \right] \\ &= 2 \cdot \mathbb{E}_{p_{\theta_0}(\mathbf{x})} [-KL(q_{\eta}(\mathbf{z}|\mathbf{x}, y) q_*^r(y|\mathbf{x}) \| p_{\theta}(\mathbf{z}, y|\mathbf{x}))]. \end{aligned}$$

- Proof

- 1) Expand $\mathbb{E}_{p_{\theta_0}(\mathbf{x})}[\cdot] = \frac{1}{2} \mathbb{E}_{p_{\theta_0}(\mathbf{x}|y=1)}[\cdot] + \frac{1}{2} \mathbb{E}_{p_{\theta_0}(\mathbf{x}|y=0)}[\cdot]$

- 2) $\frac{1}{2} \mathbb{E}_{p_{\theta_0}(\mathbf{x}|y=0)}[\cdot]$ is constant

- Due to the perfect discriminator $q_*^r(y|\mathbf{x})$
- Blocks out generated samples in the training loss

- 3) $\frac{1}{2} \mathbb{E}_{p_{\theta_0}(\mathbf{x}|y=1)}[\cdot] = \frac{1}{2} \mathbb{E}_{p_{data}(\mathbf{x})}[\cdot]$

- Recovers the conventional formulation

Proof of Lemma 2

Proof. For the reconstruction term:

$$\begin{aligned}
 & \mathbb{E}_{p_{\theta_0}(\mathbf{x})} \left[\mathbb{E}_{q_{\eta}(\mathbf{z}|\mathbf{x},y)q_*^r(y|\mathbf{x})} [\log p_{\theta}(\mathbf{x}|\mathbf{z}, y)] \right] \\
 &= \frac{1}{2} \mathbb{E}_{p_{\theta_0}(\mathbf{x}|y=1)} \left[\mathbb{E}_{q_{\eta}(\mathbf{z}|\mathbf{x},y=0),y=0 \sim q_*^r(y|\mathbf{x})} [\log p_{\theta}(\mathbf{x}|\mathbf{z}, y = 0)] \right] \\
 &+ \frac{1}{2} \mathbb{E}_{p_{\theta_0}(\mathbf{x}|y=0)} \left[\mathbb{E}_{q_{\eta}(\mathbf{z}|\mathbf{x},y=1),y=1 \sim q_*^r(y|\mathbf{x})} [\log p_{\theta}(\mathbf{x}|\mathbf{z}, y = 1)] \right] \\
 &= \frac{1}{2} \mathbb{E}_{p_{data}(\mathbf{x})} \left[\mathbb{E}_{\tilde{q}_{\eta}(\mathbf{z}|\mathbf{x})} [\log \tilde{p}_{\theta}(\mathbf{x}|\mathbf{z})] \right] + const,
 \end{aligned} \tag{25}$$

where $y = 0 \sim q_*^r(y|\mathbf{x})$ means $q_*^r(y|\mathbf{x})$ predicts $y = 0$ with probability 1. Note that both $q_{\eta}(\mathbf{z}|\mathbf{x}, y = 1)$ and $p_{\theta}(\mathbf{x}|\mathbf{z}, y = 1)$ are constant distributions without free parameters to learn; $q_{\eta}(\mathbf{z}|\mathbf{x}, y = 0) = \tilde{q}_{\eta}(\mathbf{z}|\mathbf{x})$, and $p_{\theta}(\mathbf{x}|\mathbf{z}, y = 0) = \tilde{p}_{\theta}(\mathbf{x}|\mathbf{z})$.

For the KL prior regularization term:

$$\begin{aligned}
 & \mathbb{E}_{p_{\theta_0}(\mathbf{x})} [\mathbf{KL}(q_{\eta}(\mathbf{z}|\mathbf{x}, y)q_*^r(y|\mathbf{x})||p(\mathbf{z}|y)p(y))] \\
 &= \mathbb{E}_{p_{\theta_0}(\mathbf{x})} \left[\int q_*^r(y|\mathbf{x}) \mathbf{KL}(q_{\eta}(\mathbf{z}|\mathbf{x}, y)||p(\mathbf{z}|y)) dy + \mathbf{KL}(q_*^r(y|\mathbf{x})||p(y)) \right] \\
 &= \frac{1}{2} \mathbb{E}_{p_{\theta_0}(\mathbf{x}|y=1)} [\mathbf{KL}(q_{\eta}(\mathbf{z}|\mathbf{x}, y = 0)||p(\mathbf{z}|y = 0)) + const] + \frac{1}{2} \mathbb{E}_{p_{\theta_0}(\mathbf{x}|y=1)} [const] \\
 &= \frac{1}{2} \mathbb{E}_{p_{data}(\mathbf{x})} [\mathbf{KL}(\tilde{q}_{\eta}(\mathbf{z}|\mathbf{x})||\tilde{p}(\mathbf{z}))].
 \end{aligned} \tag{26}$$

Combining Eq.(25) and Eq.(26) we recover the conventional VAE objective in Eq.(7) in the paper. \square

GANs vs VAEs side by side

	GANs (InfoGAN)	VAEs
Generative distribution	$p_{\theta}(\mathbf{x} y) = \begin{cases} p_{g_{\theta}}(\mathbf{x}) & y = 0 \\ p_{data}(\mathbf{x}) & y = 1. \end{cases}$	$p_{\theta}(\mathbf{x} \mathbf{z}, y) = \begin{cases} p_{\theta}(\mathbf{x} \mathbf{z}) & y = 0 \\ p_{data}(\mathbf{x}) & y = 1. \end{cases}$
Discriminator distribution	$q_{\phi}(y \mathbf{x})$	$q_{*}(y \mathbf{x}), \text{ perfect, degenerated}$
z-inference model	$q_{\eta}(\mathbf{z} \mathbf{x}, y) \text{ of InfoGAN}$	$q_{\eta}(\mathbf{z} \mathbf{x}, y)$
KLD to minimize	$\min_{\theta} \text{KL}(p_{\theta}(\mathbf{x} y) \parallel q^r(\mathbf{x} \mathbf{z}, y))$ $\sim \min_{\theta} \text{KL}(P_{\theta} \parallel Q)$	$\min_{\theta} \text{KL}(q_{\eta}(\mathbf{z} \mathbf{x}, y)q_{*}^r(y \mathbf{x}) \parallel p_{\theta}(\mathbf{z}, y \mathbf{x}))$ $\sim \min_{\theta} \text{KL}(Q \parallel P_{\theta})$

Link back to wake sleep algorithm

- Denote
 - Latent variables \mathbf{h}
 - Parameters λ
- Recap: wake sleep algorithm
 - Wake : $\max_{\theta} \mathbb{E}_{q_{\lambda}(\mathbf{h}|\mathbf{x})p_{data}(\mathbf{x})} [\log p_{\theta}(\mathbf{x}|\mathbf{h})]$
 - Sleep : $\max_{\lambda} \mathbb{E}_{p_{\theta}(\mathbf{x}|\mathbf{h})p(\mathbf{h})} [\log q_{\lambda}(\mathbf{h}|\mathbf{x})]$

VAEs vs. Wake-sleep

- Wake sleep algorithm

$$\text{Wake : } \max_{\boldsymbol{\theta}} \mathbb{E}_{q_{\lambda}(\mathbf{h}|\mathbf{x})p_{data}(\mathbf{x})} [\log p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{h})]$$

$$\text{Sleep : } \max_{\lambda} \mathbb{E}_{p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{h})p(\mathbf{h})} [\log q_{\lambda}(\mathbf{h}|\mathbf{x})]$$

- Let \mathbf{h} be \mathbf{z} , and λ be $\boldsymbol{\eta}$

$$\Rightarrow \max_{\boldsymbol{\theta}} \mathbb{E}_{q_{\boldsymbol{\eta}}(\mathbf{z}|\mathbf{x})p_{data}(\mathbf{x})} [\log p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})], \text{ recovers VAE objective of optimizing } \boldsymbol{\theta}$$

- VAEs extend wake phase by also learning the inference model ($\boldsymbol{\eta}$)

$$\max_{\boldsymbol{\theta}, \boldsymbol{\eta}} \mathcal{L}_{\boldsymbol{\theta}, \boldsymbol{\eta}}^{\text{vae}} = \mathbb{E}_{q_{\boldsymbol{\eta}}(\mathbf{z}|\mathbf{x})p_{data}(\mathbf{x})} [\log p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})] - \mathbb{E}_{p_{data}(\mathbf{x})} [\text{KL}(q_{\boldsymbol{\eta}}(\mathbf{z}|\mathbf{x})||p(\mathbf{z}))]$$

- Minimize the KLD in the original variational free energy wrt. $\boldsymbol{\eta}$
- Stick to minimizing the wake-phase KLD wrt. both $\boldsymbol{\theta}$ and $\boldsymbol{\eta}$
- Do not involve sleep-phase objective
- Recall: sleep phase minimizes the *reverse* KLD in the variational free energy

GANs vs. Wake-sleep

- Wake sleep algorithm

$$\text{Wake : } \max_{\theta} \mathbb{E}_{q_{\lambda}(\mathbf{h}|\mathbf{x})p_{data}(\mathbf{x})} [\log p_{\theta}(\mathbf{x}|\mathbf{h})]$$

$$\text{Sleep : } \max_{\lambda} \mathbb{E}_{p_{\theta}(\mathbf{x}|\mathbf{h})p(\mathbf{h})} [\log q_{\lambda}(\mathbf{h}|\mathbf{x})]$$

- Let \mathbf{h} be y , and λ be ϕ

$$\Rightarrow \max_{\phi} \mathbb{E}_{p_{\theta}(\mathbf{x}|y)p(y)} [\log q_{\phi}(y|\mathbf{x})], \text{ recovers GAN objective of optimizing } \phi$$

- GANs extend sleep phase by also learning the generative model (θ)

- Directly extending sleep phase: $\max_{\theta} \mathcal{L}_{\theta} = \mathbb{E}_{p_{\theta}(\mathbf{x}|y)p(y)} [\log q_{\phi}(y|\mathbf{x})]$

- GANs: $\max_{\theta} \mathcal{L}_{\theta} = \mathbb{E}_{p_{\theta}(\mathbf{x}|y)p(y)} [\log q_{\phi}^r(y|\mathbf{x})]$

- The only difference is replacing q_{ϕ} with q_{ϕ}^r

- This is where adversarial mechanism come about !

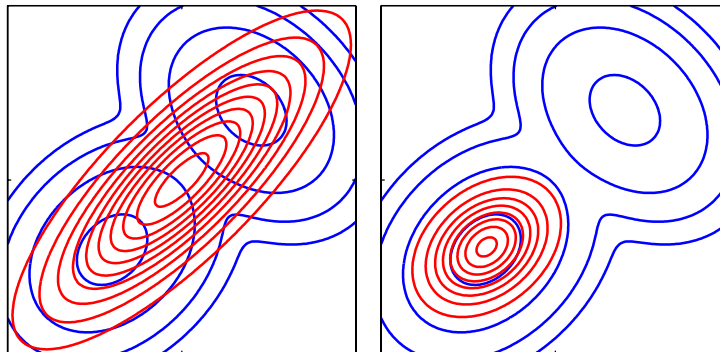
- GANs stick to minimizing the sleep-phase KLD

- Do not involve wake-phase objective

Mutual exchanges of ideas: augment the loss functions

	GANs (InfoGAN)	VAEs
KLD to minimize	$\min_{\theta} \text{KL}(p_{\theta}(\mathbf{x} y) \parallel q^r(\mathbf{x} \mathbf{z}, y))$ $\sim \min_{\theta} \text{KL}(P_{\theta} \parallel Q)$	$\min_{\theta} \text{KL}(q_{\eta}(\mathbf{z} \mathbf{x}, y)q_{*}^r(y \mathbf{x}) \parallel p_{\theta}(\mathbf{z}, y \mathbf{x}))$ $\sim \min_{\theta} \text{KL}(Q \parallel P_{\theta})$

- Asymmetry of KLDs inspires combination of GANs and VAEs
 - GANs: $\min_{\theta} \text{KL}(P_{\theta} \parallel Q)$ tends to missing mode
 - VAEs: $\min_{\theta} \text{KL}(Q \parallel P_{\theta})$ tends to cover regions with small values of p_{data}



Mode covering

Mode missing

Mutual exchanges of ideas: augment the loss functions

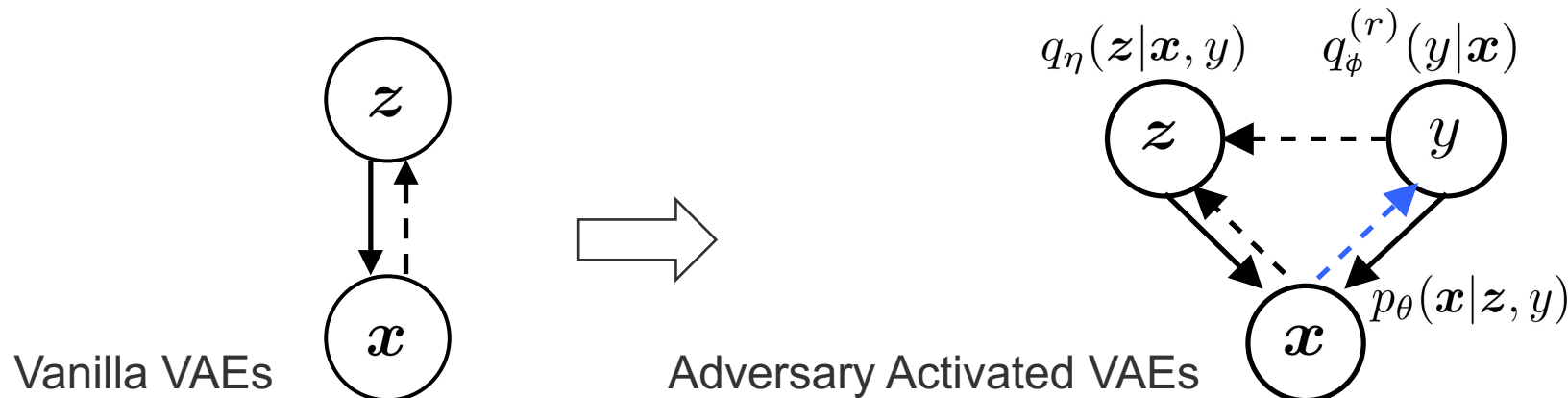
	GANs (InfoGAN)	VAEs
KLD to minimize	$\min_{\theta} \text{KL}(p_{\theta}(\mathbf{x} y) \parallel q^r(\mathbf{x} \mathbf{z}, y))$ $\sim \min_{\theta} \text{KL}(P_{\theta} \parallel Q)$	$\min_{\theta} \text{KL}(q_{\eta}(\mathbf{z} \mathbf{x}, y)q_{*}^r(y \mathbf{x}) \parallel p_{\theta}(\mathbf{z}, y \mathbf{x}))$ $\sim \min_{\theta} \text{KL}(Q \parallel P_{\theta})$

- Asymmetry of KLDs inspires combination of GANs and VAEs
 - GANs: $\min_{\theta} \text{KL}(P_{\theta} \parallel Q)$ tends to missing mode
 - VAEs: $\min_{\theta} \text{KL}(Q \parallel P_{\theta})$ tends to cover regions with small values of p_{data}
 - Augment VAEs with GAN loss [Larsen et al., 2016]
 - Alleviate the mode covering issue of VAEs
 - Improve the sharpness of VAE generated images
 - Augment GANs with VAE loss [Che et al., 2017]
 - Alleviate the mode missing issue of GANs

Mutual exchanges of ideas: augment the graphical model

	GANs (InfoGAN)	VAEs
Discriminator distribution	$q_\phi(y \mathbf{x})$	$q_*(y \mathbf{x})$, perfect, degenerated

- Activate the adversarial mechanism in VAEs
 - Enable adaptive incorporation of fake samples for learning
 - Straightforward derivation by making symbolic analog to GANs



Adversary Activated VAEs (AAVAE)

- Vanilla VAEs:

$$\max_{\theta, \eta} \mathcal{L}_{\theta, \eta}^{\text{vae}} = \mathbb{E}_{p_{\theta_0}(\mathbf{x})} \left[\mathbb{E}_{q_{\eta}(\mathbf{z}|\mathbf{x}, y)} q_*^r(\mathbf{y}|\mathbf{x}) [\log p_{\theta}(\mathbf{x}|\mathbf{z}, y)] - \text{KL}(q_{\eta}(\mathbf{z}|\mathbf{x}, y) q_*^r(\mathbf{y}|\mathbf{x}) \| p(\mathbf{z}|y)p(y)) \right]$$

- Replace $q_*(\mathbf{y}|\mathbf{x})$ with learnable one $q_{\phi}(\mathbf{y}|\mathbf{x})$ with parameters ϕ
 - As usual, denote reversed distribution $q_{\phi}^r(\mathbf{y}|\mathbf{x}) = q_{\phi}(\mathbf{y}|\mathbf{x})$

$$\max_{\theta, \eta} \mathcal{L}_{\theta, \eta}^{\text{aavae}} = \mathbb{E}_{p_{\theta_0}(\mathbf{x})} \left[\mathbb{E}_{q_{\eta}(\mathbf{z}|\mathbf{x}, y)} q_{\phi}^r(\mathbf{y}|\mathbf{x}) [\log p_{\theta}(\mathbf{x}|\mathbf{z}, y)] - \text{KL}(q_{\eta}(\mathbf{z}|\mathbf{x}, y) q_{\phi}^r(\mathbf{y}|\mathbf{x}) \| p(\mathbf{z}|y)p(y)) \right]$$

AAVAE: adaptive data selection

$$\max_{\theta, \eta} \mathcal{L}_{\theta, \eta}^{\text{avae}} = \mathbb{E}_{p_{\theta_0}(\mathbf{x})} \left[\mathbb{E}_{q_{\eta}(\mathbf{z}|\mathbf{x}, y) q_{\phi}^r(y|\mathbf{x})} [\log p_{\theta}(\mathbf{x}|\mathbf{z}, y)] - \text{KL}(q_{\eta}(\mathbf{z}|\mathbf{x}, y) q_{\phi}^r(y|\mathbf{x}) \| p(\mathbf{z}|y)p(y)) \right]$$

- An effective data selection mechanism:
 - Both generated samples and real examples are weighted by $q_{\phi}^r(y = 0|\mathbf{x}) = q_{\phi}(y = 1|\mathbf{x})$
 - Only samples that resembles real data and fool the discriminator will be used for training
 - A real example receiving large weight $q_{\phi}^r(y|\mathbf{x})$
 - ⇒ Easily recognized by the discriminator as real
 - ⇒ Hard to be simulated from the generator
 - ⇒ Hard examples get larger weights

AAVAE: discriminator learning

- Use the binary classification objective as in GAN

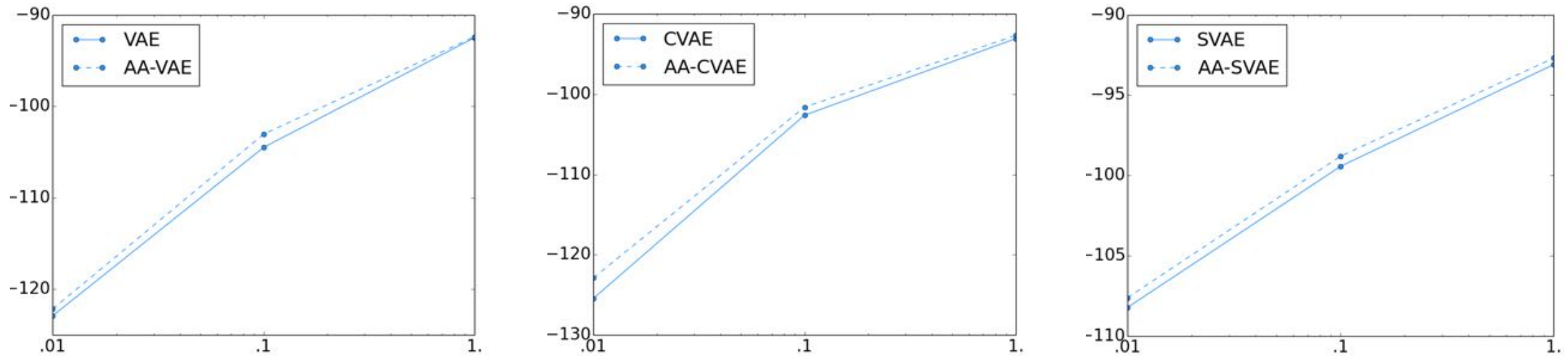
$$\max_{\phi} \mathcal{L}_{\phi}^{\text{aavae}} = \mathbb{E}_{p_{\theta}(\mathbf{x}|\mathbf{z},y)p(\mathbf{z}|y)p(y)} [\log q_{\phi}(y|\mathbf{x})]$$

AAVAE: empirical results

- Applied the adversary activating method on
 - vanilla VAEs
 - class-conditional VAEs (CVAE)
 - semi-supervised VAEs (SVAE)

AAVAE: empirical results

- Evaluated test-set variational lower bound on MNIST
 - The higher the better



- X-axis: the ratio of training data for learning (0.01, 0.1, 1.)
- Y-axis: value of test-set lower bound

AAVAE: empirical results

- Evaluated classification accuracy of SVAE and AA-SVAE

	1%	10%
SVAE	0.9412±.0039	0.9768±.0009
AASVAE	0.9425±.0045	0.9797±.0010

- Used 1% and 10% data labels in MNIST

Mutual exchanges of ideas

- AAVAE enhances VAEs with ideas from GANs
- We can also enhance GANs with ideas from VAEs
- VAEs maximize a variational lower bound of log likelihood
- Importance weighted VAE (IWAE) [Burda et al., 2016]
 - Maximizes a tighter lower bound through importance sampling
- The variational inference interpretation of GANs allows the importance weighting method to be straightforwardly applied to GANs
 - Just copy the derivations of IWAE side by side with little adaptations!

Importance weighted GANs (IWGAN)

- Generator learning in vanilla GANs

$$\max_{\theta} \mathbb{E}_{\mathbf{x} \sim p_{\theta}(\mathbf{x}|y)p(y)} [\log q_{\phi_0}^r(y|\mathbf{x})]$$

- Generator learning in IWGAN

$$\max_{\theta} \mathbb{E}_{\mathbf{x}_1, \dots, \mathbf{x}_k \sim p_{\theta}(\mathbf{x}|y)p(y)} \left[\sum_{i=1}^k \frac{q_{\phi_0}^r(y|\mathbf{x}_i)}{q_{\phi_0}(y|\mathbf{x}_i)} \log q_{\phi_0}^r(y|\mathbf{x}_i) \right]$$

- Assigns higher weights to samples that are more realistic and fool the discriminator better

IWGAN: empirical results

- Applied the importance weighting method to
 - vanilla GANs
 - class-conditional GANs (CGAN)
 - CGAN adds one dimension to code z to represent the class label
 - The derivations of the IW extension remain the same as in vanilla GANs

IWGAN: empirical results

- Evaluated on MNIST and SVHN
- Used pretrained NN to evaluate:
 - Inception scores of samples from GANs and IW-GAN
 - Confidence of a pre-trained classifier on generated samples + diversity of generated samples

	MNIST	SVHN
GAN	8.34±.03	5.18±.03
IWGAN	8.45±.04	5.34±.03

- Classification accuracy of samples from CGAN and IW-CGAN

	MNIST	SVHN
CGAN	0.985±.002	0.797±.005
IWCGAN	0.987±.002	0.798±.006

Recap: Variational Inference

Maximize the variational lower bound $\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x})$, or equivalently, minimize **free energy**

$$F(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x}) = -\log p(\mathbf{x}) + KL(q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) || p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x}))$$

- E-step: maximize \mathcal{L} wrt. $\boldsymbol{\phi}$ with $\boldsymbol{\theta}$ fixed

$$\max_{\boldsymbol{\phi}} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x}) = \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})} [\log p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})] + KL(q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) || p(\mathbf{z}))$$

- If with closed form solutions

$$q_{\boldsymbol{\phi}}^*(\mathbf{z}|\mathbf{x}) \propto \exp[\log p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})]$$

- M-step: maximize \mathcal{L} wrt. $\boldsymbol{\theta}$ with $\boldsymbol{\phi}$ fixed

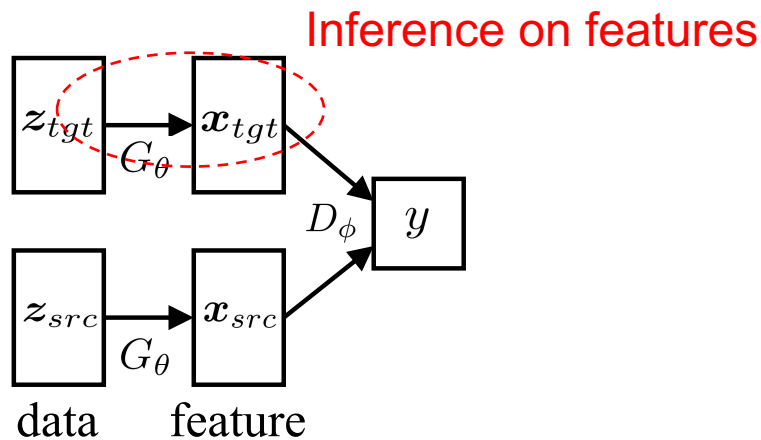
$$\max_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x}) = \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})} [\log p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})] + KL(q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) || p(\mathbf{z}))$$

Discussion: Modeling latent vs. visible variables

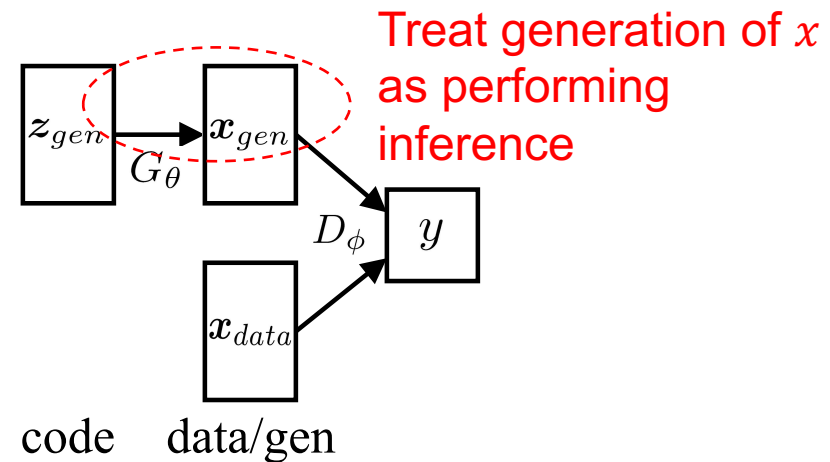
- Latent and visible variables are traditionally distinguished clearly and modeled in very different ways
- A key thought in the new formulation:
 - Not necessary to make clear boundary between **latent** and **visible** variables,
 - And between **inference** and **generation**
 - Instead treat them as a symmetric pair

Symmetric modeling of latent & visible variables

- Help with modeling and understanding:
 - Treating the generation space \mathbf{x} in GANs as latent
 - reveals the connection between GANs and ADA
 - provides an variational inference interpretation of generation



ADA



GANs

Symmetric modeling of latent & visible variables

- Help with modeling and understanding:
 - Treating the generation space \mathbf{x} in GANs as latent
 - reveals the connection between GANs and ADA
 - provides an variational inference interpretation of generation
 - Wake sleep algorithm
 - wake phase reconstructs **visible** variables based on **latents**
 - sleep phase reconstructs **latent** variables based on **visibles**
 - latent and visible variables are treated in a completely symmetric way

$$\text{Wake: } \max_{\theta} E_{q_{\phi}(\mathbf{z}|\mathbf{x})} [\log p_{\theta}(\mathbf{x}, \mathbf{z})]$$

$$\text{Sleep: } \max_{\phi} E_{p_{\theta}(\mathbf{z}, \mathbf{x})} [\log q_{\phi}(\mathbf{z}|\mathbf{x})]$$

Symmetric modeling of latent & visible variables

- New modeling approaches narrow the gap

Empirical distributions over visible variables

- Impossible to be explicit distribution
 - The only information we have is the observe data examples
 - Do not know the true parametric form of data distribution
- Naturally an implicit distribution
 - Easy to sample from, hard to evaluate likelihood

Prior distributions over latent variables

- Traditionally defined as explicit distributions, e.g., Gaussian prior distribution
 - Amiable for likelihood evaluation
 - We can assume the parametric form according to our prior knowledge
- New tools to allow implicit priors and models
 - GANs, density ratio estimation, approximate Bayesian computations
 - E.g., adversarial autoencoder [Makhzani et al., 2015] replaces the Gaussian prior of vanilla VAEs with implicit priors

Symmetric modeling of latent & visible variables

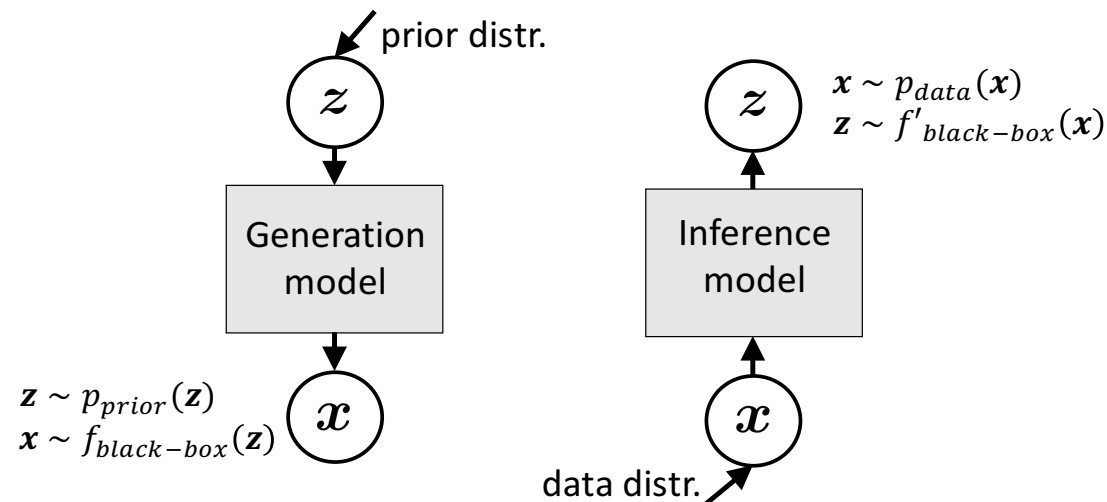
- No difference in terms of formulations
 - with implicit distributions and black-box NN models
 - just swap the symbols x and z

$$z \sim p_{\text{prior}}(z)$$

$$x \sim f_{\text{black-box}}(z)$$

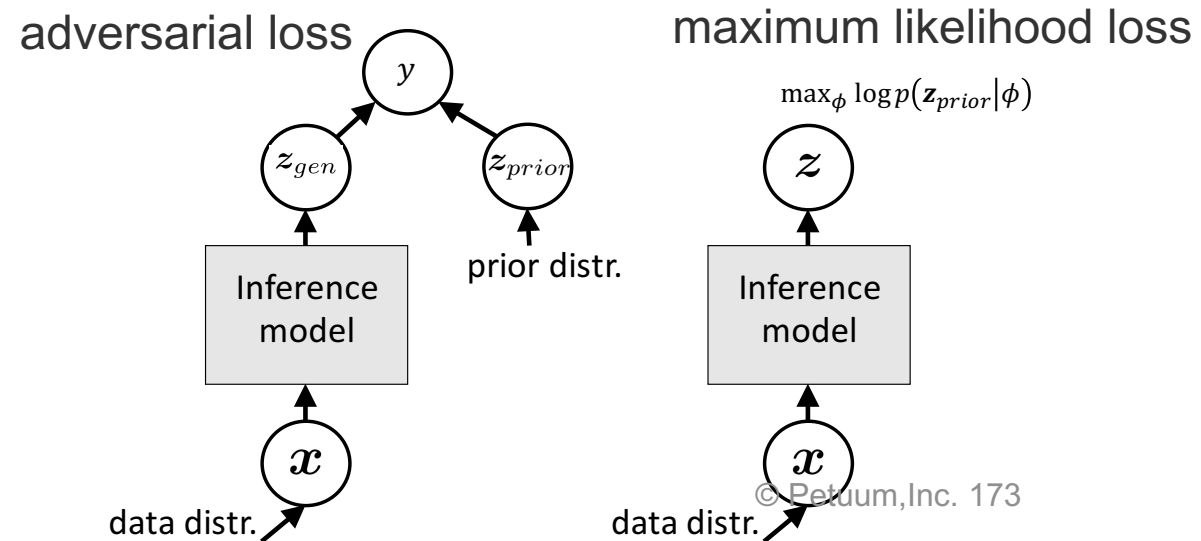
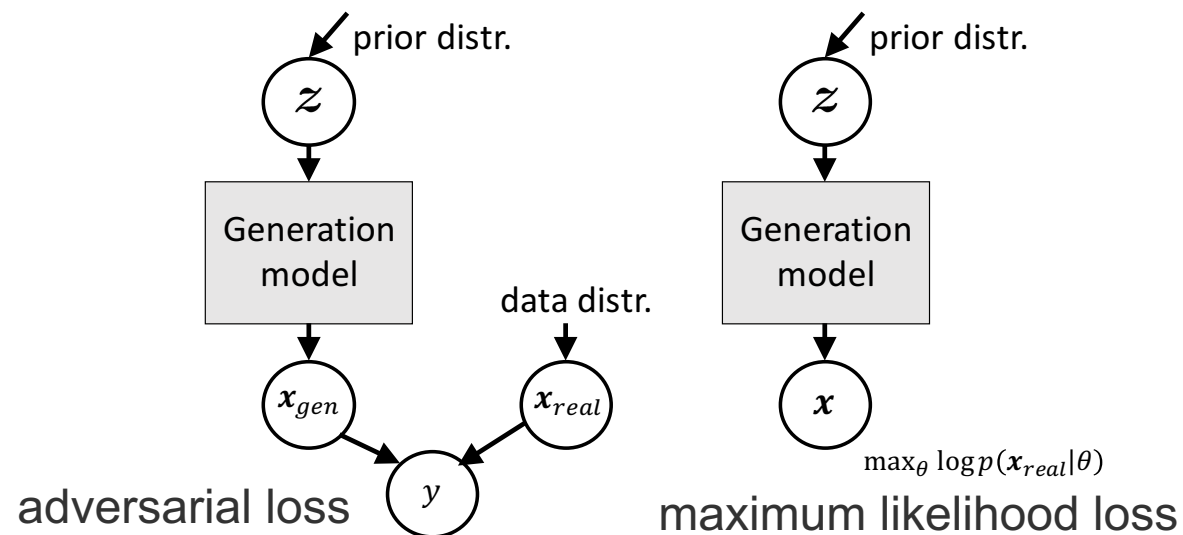
$$x \sim p_{\text{data}}(x)$$

$$z \sim f'_{\text{black-box}}(x)$$



Symmetric modeling of latent & visible variables

- No difference in terms of formulations
 - with implicit distributions and black-box NN models
- Difference in terms of space complexity
 - depend on the problem at hand
 - choose appropriate tools:
 - implicit/explicit distribution, adversarial/maximum-likelihood optimization, ...



Part-II: Conclusions

Z Hu, Z YANG, R Salakhutdinov, E Xing,
“**On Unifying Deep Generative Models**”, arxiv 1706.00550

- Deep generative models research have a long history
 - Deep belief nets / Helmholtz machines / Predictability Minimization / ...
- Unification of deep generative models
 - GANs and VAEs are essentially minimizing KLD in opposite directions
 - Extends two phases of classic wake sleep algorithm, respectively
 - A general formulation framework useful for
 - Analyzing broad class of existing DGM and variants: ADA/InfoGAN/Joint-models/...
 - Inspiring new models and algorithms by borrowing ideas across research fields
- Symmetric view of latent/visible variables
 - No difference in formulation with implicit prior distributions and black-box NN transformations
 - Difference in space complexity: choose appropriate tools

Plan

- Statistical And Algorithmic Foundation and Insight of Deep Learning
- On Unified Framework of Deep Generative Models
- Computational Mechanisms: Distributed Deep Learning Architectures

Part-III (1)

Inference and Learning

Outline

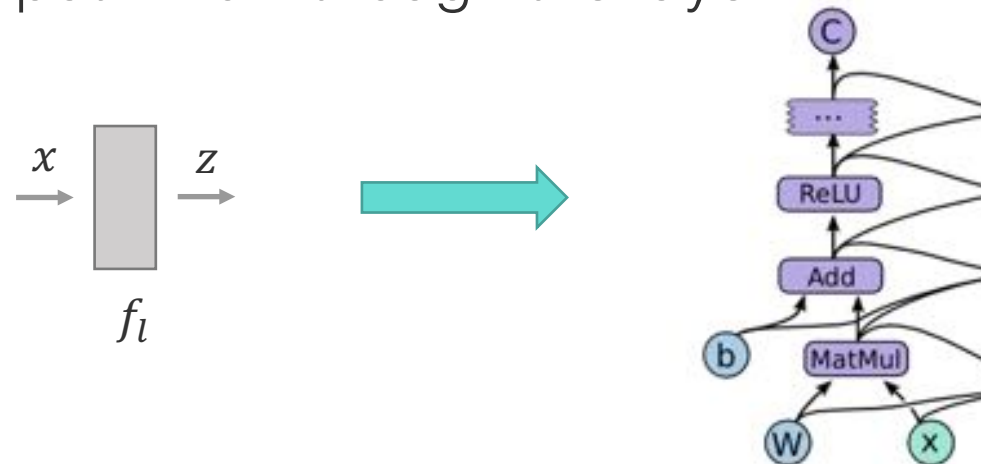
- Deep Learning as Dataflow Graphs
- Auto-differentiable Libraries

Outline

- **Deep Learning as Dataflow Graphs**
- Auto-differentiable Libraries

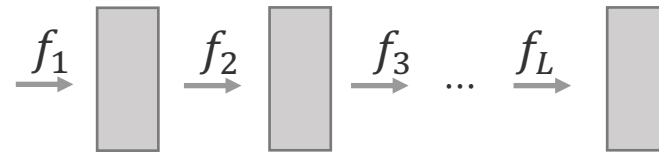
A Computational Layer in DL

- A layer in a neural network is composed of a few finer computational operations
 - A layer l has input x and output z , and transforms x into z following:
 $y = Wx + b, z = \text{ReLU}(y)$
 - Denote the transformation of layer l as f_l , which can be represented as a dataflow graphs: the input x flow through the layer



From Layers to Networks

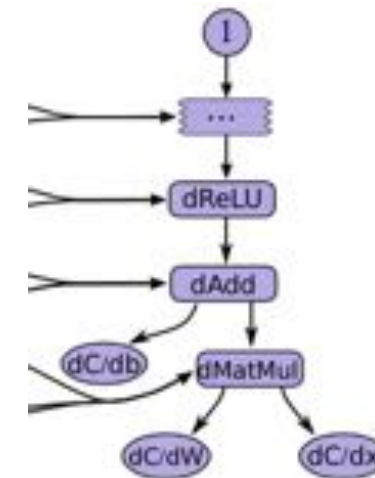
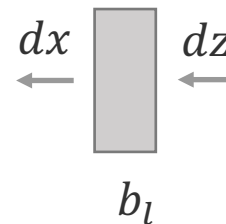
- A neural network is thus a few stacked layers $l = 1, \dots, L$, where every layer represents a function transform f_l
 - The forward computation proceeds by sequentially executing $f_1, f_2, f_3, \dots, f_L$



- Training the neural network involves deriving the gradient of its parameters with a backward pass (next slides)

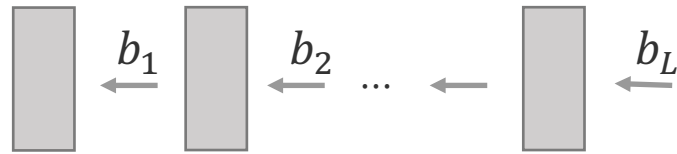
A Computational Layer in DL

- Denote the backward pass through a layer l as b_l
 - b_l derives the gradients of the input $x(dx)$, given the gradient of z as dz , as well as the gradients of the parameters W, b
 - dx will be the backward input of its previous layer $l - 1$
 - Backward pass can be thought as a backward dataflow where the gradient flow through the layer



Backpropagation through a NN

- The backward computation proceeds by sequentially executing $b_L, b_{L-1}, b_{L-2}, \dots, b_1$



A Layer as a Dataflow Graph

- Give the forward computation flow, gradients can be computed by auto differentiation
 - Automatically derive the backward gradient flow graph from the forward dataflow graph

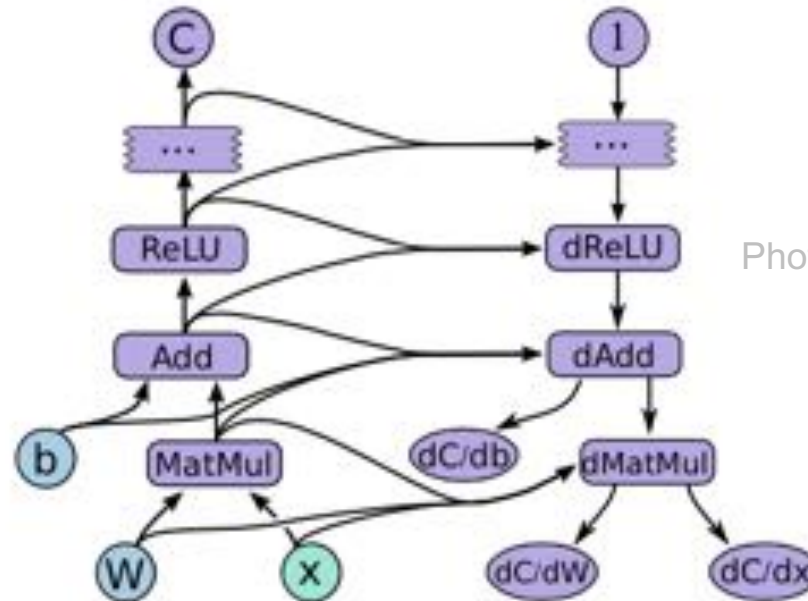


Photo from TensorFlow website

A Network as a Dataflow Graph

- Gradients can be computed by auto differentiation
 - Automatically derive the gradient flow graph from the forward dataflow graph

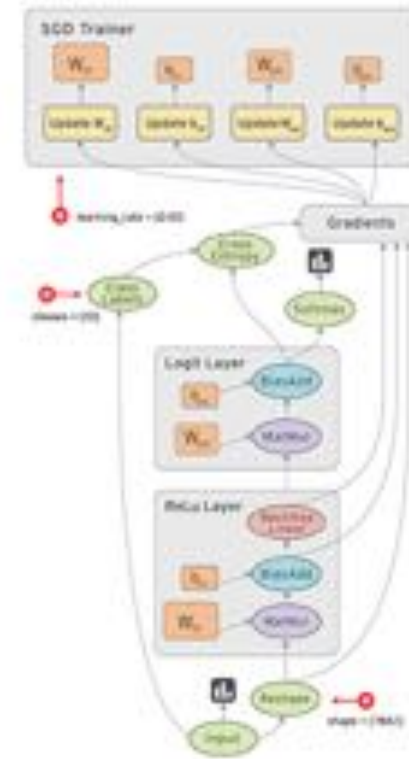
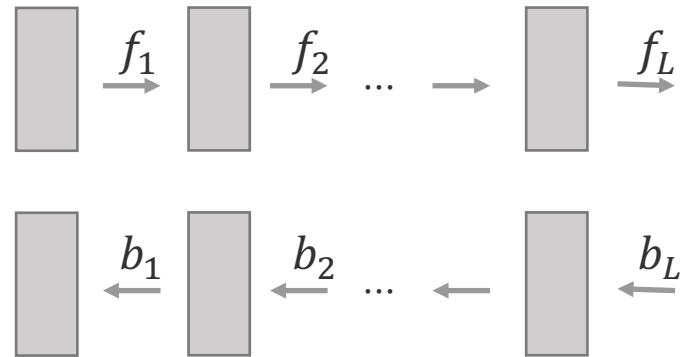
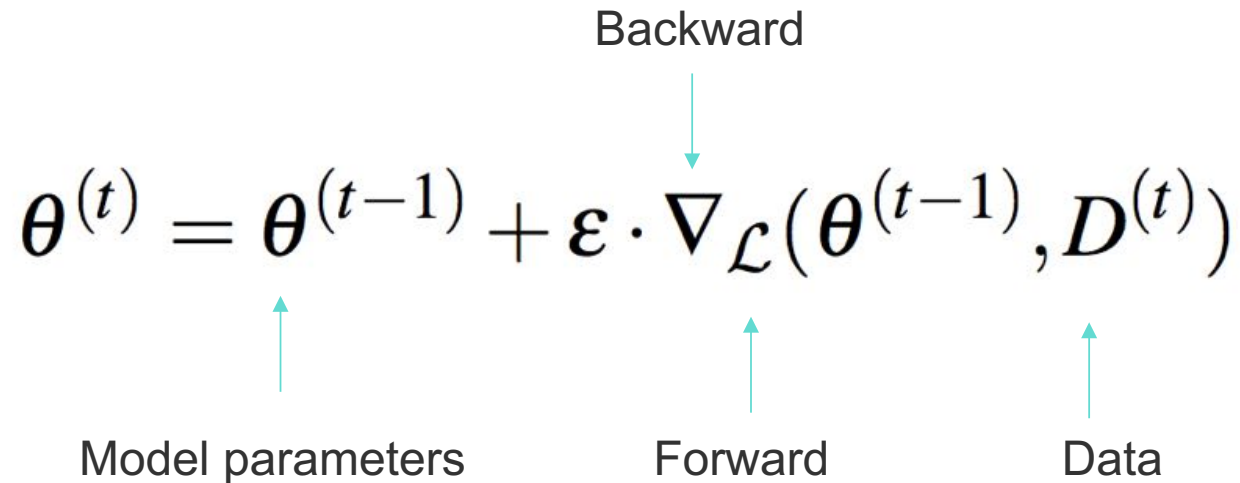


Photo from TensorFlow website

Gradient Descent via Backpropagation

- The computational workflow of deep learning
 - Forward, which we usually also call inference: forward dataflow
 - Backward, which derives the gradients: backward gradient flow
 - Apply/update gradients and repeat

- Mathematically,

$$\theta^{(t)} = \theta^{(t-1)} + \varepsilon \cdot \nabla_{\mathcal{L}}(\theta^{(t-1)}, D^{(t)})$$


Model parameters
Forward
Data

Program a neural network

- Define a neural network
 - Define operations and layers: fully-connected? Convolution? Recurrent?
 - Define the data I/O: read what data from where?
 - Define a loss function/optimization objective: L2 loss? Softmax? Ranking Loss?
 - Define an optimization algorithm: SGD? Momentum SGD? etc
- Auto-differential Libraries will then take over
 - Connect operations, data I/O, loss functions and trainer.
 - Build forward dataflow graph and backward gradient flow graphs.
 - Perform training and apply updates

Outline

- Deep Learning as Dataflow Graphs
- **Auto-differentiable Libraries**

Auto-differential Libraries

- Auto-differential Library automatically derives the gradients following the back-propagation rule.
- A lot of auto-differentiation libraries have been developed:
- So-called Deep Learning toolkits

Caffe  Caffe2


TensorFlow

 Microsoft
CNTK

 torch

PYTORCH

DyNet

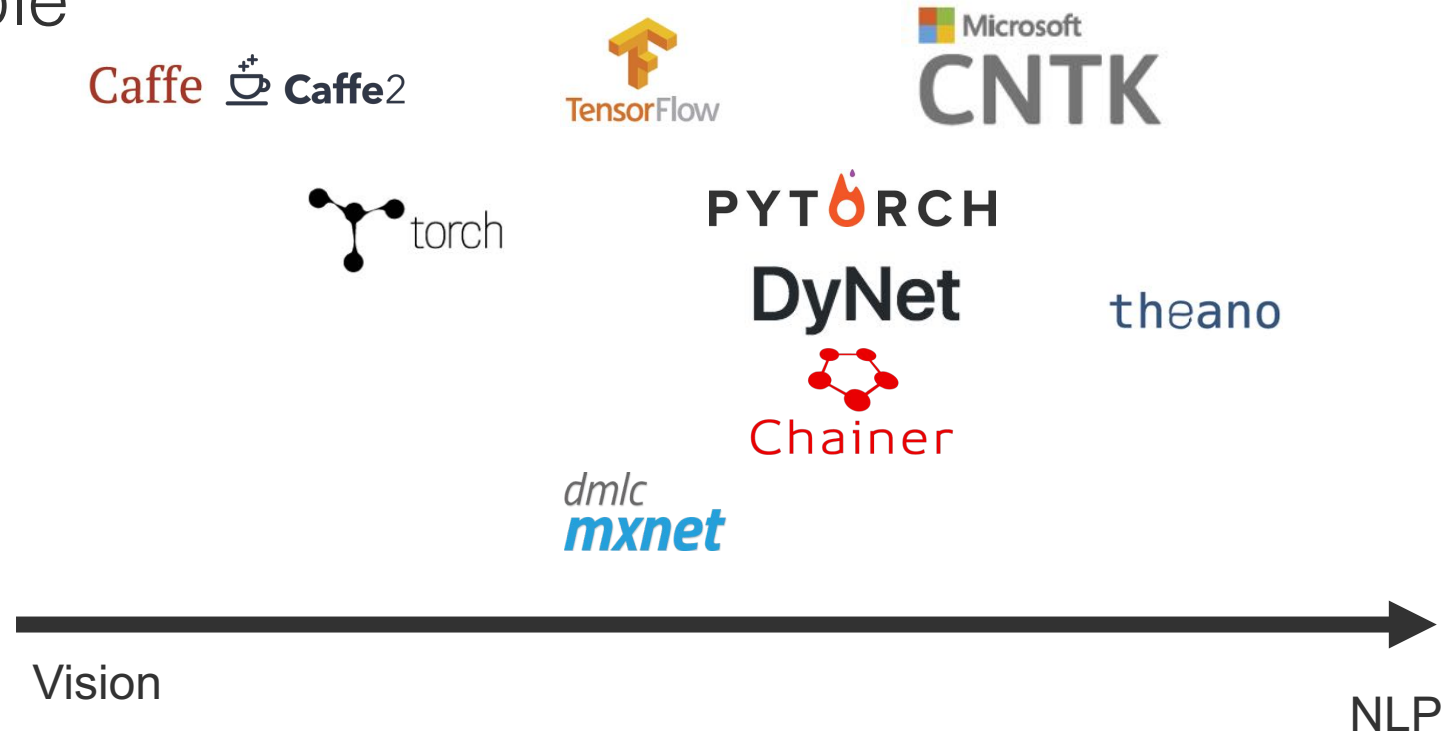
theano


Chainer

dmlc
mxnet

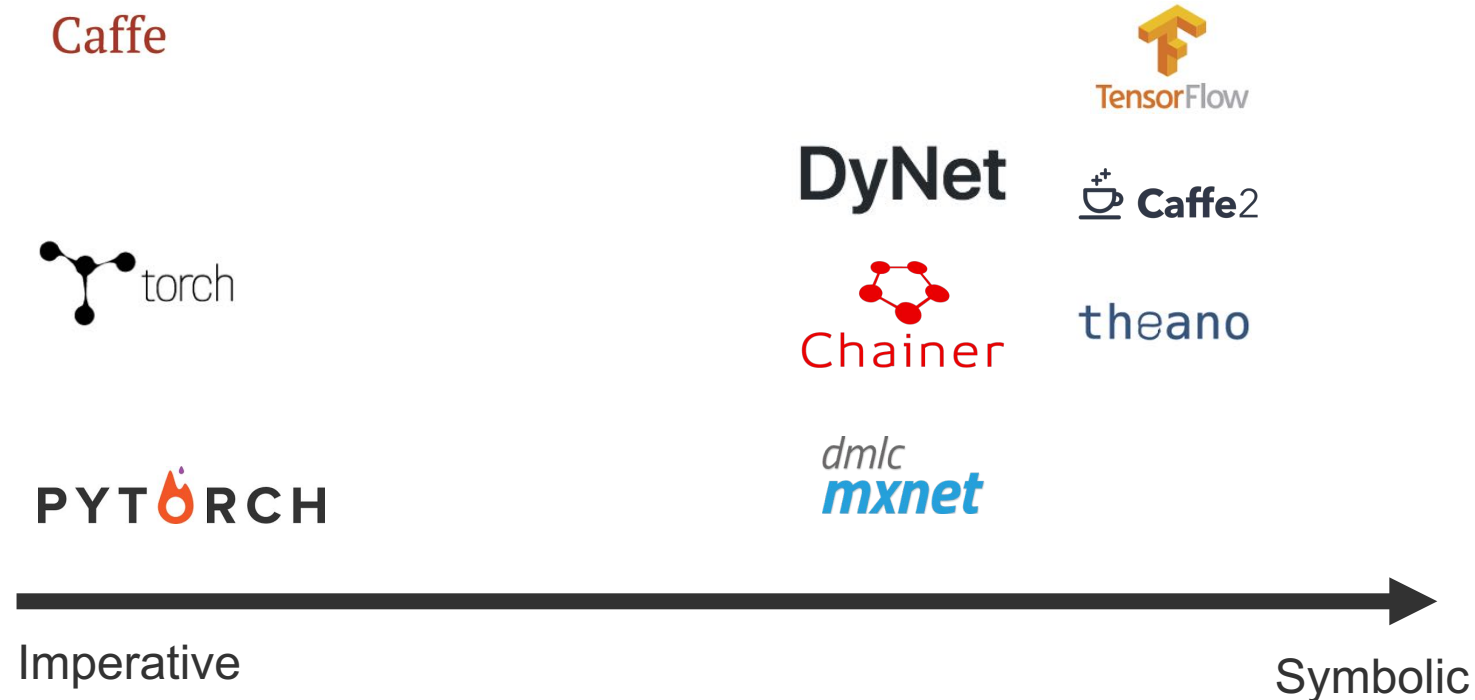
Deep Learning Toolkits

- They are adopted differently in different domains
- For example



Deep Learning Toolkits

- They are also designed differently
 - Symbolic v.s. imperative programming



Deep Learning Toolkits

- Symbolic vs. imperative programming
 - Symbolic: write symbols to assemble the networks first, evaluate later
 - Imperative: immediate evaluation

```
A = Variable('A')
B = Variable('B')
C = B * A
D = C + Constant(1)
# compiles the function
f = compile(D)
d = f(A=np.ones(10), B=np.ones(10)*2)
```

Symbolic

```
import numpy as np
a = np.ones(10)
b = np.ones(10) * 2
c = b * a
d = c + 1
```

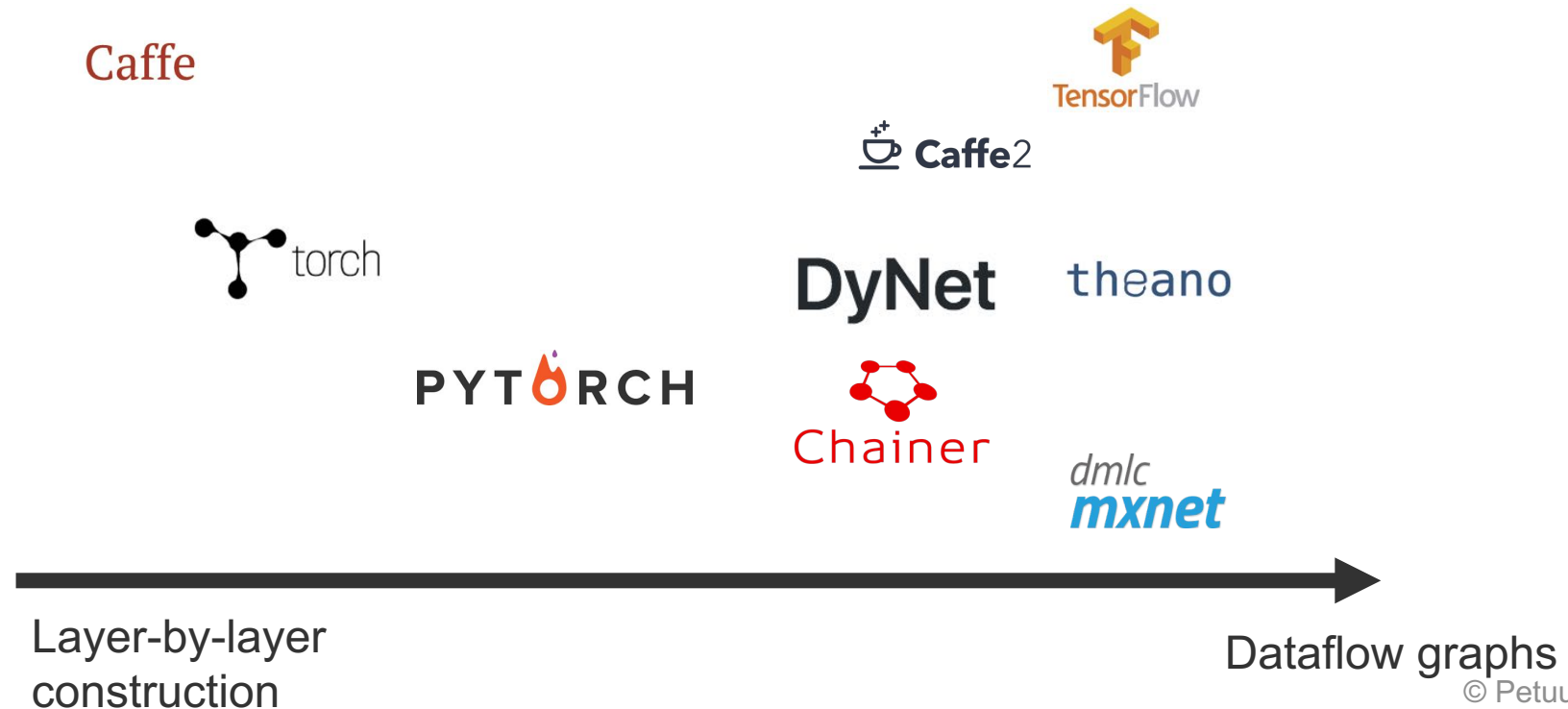
Imperative

Deep Learning Toolkits

- Symbolic
 - Good
 - easy to optimize (e.g. distributed, batching, parallelization) for developers
 - More efficient
 - Bad
 - The way of programming might be counter-intuitive
 - Hard to debug for user programs
 - Less flexible: you need to write symbols before actually doing anything
- Imperative:
 - Good
 - More flexible: write one line, evaluate one line
 - Easy to program and easy to debug: because it matches the way we use C++ or python
 - Bad
 - Less efficient
 - More difficult to optimize

Deep Learning Toolkits

- They are also designed differently
 - For another example, dataflow graphs v.s. layer-by-layer construction



Good and Bad of Dataflow Graphs

- Dataflow graphs seems to be a dominant choice for representing deep learning models
 - What's good for dataflow graphs
 - Good for **static** workflows: define once, run for arbitrary batches/data
 - Programming convenience: **easy to program once you get used to it.**
 - Easy to parallelize/batching for a fixed graph
 - Easy to optimize: a lot of off-the-shelf optimization techniques for graph
 - What's bad for dataflow graphs
 - Not good for **dynamic** workflows: need to define a graph for every training sample - > overheads
 - Hard to program dynamic neural networks: how can you define dynamic graphs using a language for static graphs? (e.g. LSTM, tree-LSTM).
 - Not easy for debugging.
 - Difficult to parallelize/batching across multiple graphs: every graph is different, no natural batching.

Static vs. Dynamic Dataflow Graphs

- Static Dataflow graphs
 - Define once, execute many times
 - For example: convolutional neural networks
 - Execution: Once defined, all following computation will follow the defined computation
 - Advantages
 - No extra effort for batching optimization, because it can be by nature batched
 - It is always easy to handle a static computational dataflow graphs in all aspects, because of its fixed structure
 - Node placement, distributed runtime, memory management, etc.
 - Benefit the developers

Static vs. Dynamic Dataflow Graphs

- Dynamic Dataflow graphs
 - When do we need?
 - In all cases that static dataflow graphs do not work well
 - Variably sized inputs
 - Variably structured inputs
 - Nontrivial inference algorithms
 - Variably structured outputs
 - Etc.

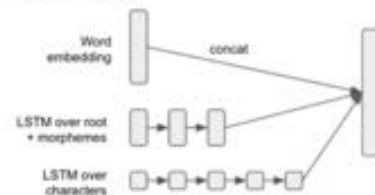
Static vs. Dynamic Dataflow Graphs

- Can we handle dynamic dataflow graphs? Using static methods (or declaration) will have a lot of problems
 - Difficulty in expressing complex flow-control logic
 - Complexity of the computation graph implementation
 - Difficulty in debugging

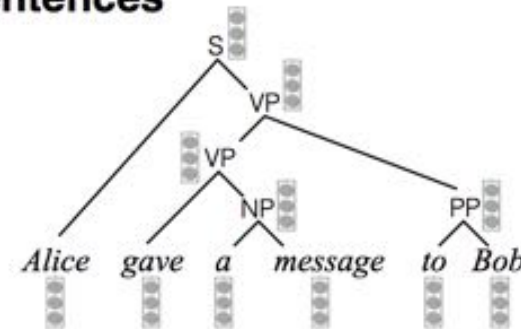
Introducing DyNet

- Designed for dynamic deep learning workflow, e.g.
 - Tree-LSTM for neural machine translation, where each sentence defines a structure that corresponds to the computational flow
 - Graph-LSTM for image parsing, where each image has a specific connection between segments
 - etc.

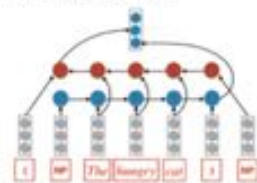
Words



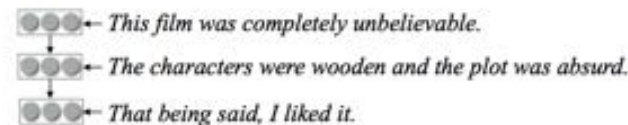
Sentences



Phrases



Documents



Key Ingredients in DyNet

- Concept
 - Separate parameter declaration and graph construction
 - Declare trainable parameters and construct models first
 - Parameters, e.g. the weight matrices in an LSTM unit.
 - Construct a model as a collection of trainable parameters
 - Construct computation graphs
 - Allocate a few nodes for our computation (node can be seen as layers in NN)
 - Specify the dataflow graph by connecting nodes together
 - If necessary, different graphs for different input samples
 - Conclusion: Define parameter once, but define graphs dynamically depending on inputs

```
model = dy.Model()

pW = model.add_parameters((20,4))
pb = model.add_parameters(20)

dy.renew_cg()
x = dy.inputVector([1,2,3,4])
W = dy.parameter(pW) # convert params to expression
b = dy.parameter(pb) # and add to the graph

y = W * x + b
```


Key Ingredients in DyNet

- Backend and programming model
 - Graph construction
 - In TensorFlow, constructing a graph has a considerable overhead.
 - TensorFlow users avoid defining graphs repeatedly
 - DyNet: highly optimized graph definition
 - Little overhead defining a graph: good for dynamic neural networks.
 - Easy to write recursive programs to define graphs (very effective for many dynamic networks, such as tree-LSTM or graph-LSTM).

Key Ingredients in DyNet

- A visual comparison

```

1  class TreeRNNBuilder(object):
2      def __init__(self, model, word_vocab, hdim):
3          self.W = model.add_parameters(hdim, 2*hdim)
4          self.E = model.add_lookup_parameters((len(word_vocab), hdim))
5          self.w2i = word_vocab
6
7      def encode(self, tree):
8          if tree.isleaf():
9              return self.E[self.w2i.get(tree.label, 0)]
10             elif len(tree.children) == 1: # unary node, skip
11                 expr = self.encode(tree.children[0])
12                 return expr
13             else:
14                 assert(len(tree.children) == 2)
15                 e1 = self.encode(tree.children[0])
16                 e2 = self.encode(tree.children[1])
17                 W = dy.parameter(self.W)
18                 expr = dy.tanh(W*dy.concatenate([e1,e2]))
19                 return expr
20
21     model = dy.Model()
22     U_p = model.add_parameters((2,50))
23     tree_builder = TreeRNNBuilder(model, word_vocabulary, 50)
24     trainer = dy.AdamTrainer(model)
25     for epoch in xrange(10):
26         for in_tree, out_label in read_examples():
27             dy.renew_cg()
28             U = dy.parameter(U_p)
29             loss = dy.pickneglogsoftmax(U+tree_builder.encode(in_tree), out_label)
30             loss.backward()
31             trainer.update()
32

```

DyNet TreeLSTM (30 LoC)

```

# Copyright 2017 Google Inc. All Rights Reserved.
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
# http://www.apache.org/licenses/LICENSE-2.0
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
# Wasteful neural network for sentence analysis using TensorFlow.

def read_examples(filename):
    """Read examples from a file. Returns a generator that yields (in_tree, out_label)
    pairs for each line in the file. The file is assumed to be a CSV file with
    the following format:
    in_tree,out_label
    The first column is a string representing the tree and the second column is
    an integer representing the label. The file is read lazily, so it can be
    used for large datasets.
    """
    with open(filename, 'r') as f:
        for line in f:
            in_tree, out_label = line.strip().split(',')
            yield in_tree, out_label

def main():
    """Main function that runs the model. It takes the following arguments:
    --train_data: Path to the training data file.
    --dev_data: Path to the development data file.
    --num_epochs: Number of training epochs.
    """
    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument('--train_data', type=str, required=True)
    parser.add_argument('--dev_data', type=str, required=True)
    parser.add_argument('--num_epochs', type=int, default=10)
    args = parser.parse_args()

    # Load the data
    train_data_loader = read_examples(args.train_data)
    dev_data_loader = read_examples(args.dev_data)

    # Build the model
    model = dy.Model()
    W = model.add_parameters([40, 80])
    E = model.add_lookup_parameters([10000, 40])
    U = model.add_parameters([2, 40])

    # Build the tree builder
    tree_builder = TreeRNNBuilder(model, dev_data_loader.get_vocab(), 40)

    # Build the trainer
    trainer = dy.AdamTrainer(model)

    # Train the model
    for epoch in range(args.num_epochs):
        for in_tree, out_label in train_data_loader:
            dy.renew_cg()
            U = dy.parameter(U)
            loss = dy.pickneglogsoftmax(U + tree_builder.encode(in_tree), out_label)
            loss.backward()
            trainer.update()

        # Evaluate on the dev set
        dev_loss = 0.0
        dev_count = 0
        for in_tree, out_label in dev_data_loader:
            dy.renew_cg()
            U = dy.parameter(U)
            dev_loss += dy.pickneglogsoftmax(U + tree_builder.encode(in_tree), out_label)
            dev_count += 1

        dev_loss /= dev_count
        print('Epoch %d, dev loss: %f' % (epoch, dev_loss))

if __name__ == '__main__':
    main()

```

TensorFlow TreeLSTM (200 LoC)

```

def __init__(self, num_units, target_size, num_embeddings, num_filters, num_filters_size, num_filters_stride):
    """Initialize the model.
    Args:
        num_units: The number of units in the LSTM cell.
        target_size: The size of the target vector.
        num_embeddings: The number of embeddings.
        num_filters: The number of filters.
        num_filters_size: The size of the filter.
        num_filters_stride: The stride of the filter.
    """
    self._num_units = num_units
    self._target_size = target_size
    self._num_embeddings = num_embeddings
    self._num_filters = num_filters
    self._num_filters_size = num_filters_size
    self._num_filters_stride = num_filters_stride

    # Create the input embedding
    self._input_embedding = tf.nn.embedding_lookup(tf.nn.embedding_lookup(self._embeddings, self._input), self._input)

    # Create the LSTM cell
    self._lstm_cell = tf.nn.rnn_cell.LSTMCell(self._num_units)

    # Create the LSTM layer
    self._lstm_layer = tf.nn.rnn_cell.LSTMBlockCellWrapper(self._lstm_cell)

    # Create the LSTM layer with input embedding
    self._lstm_layer_with_embedding = tf.nn.rnn_cell.LSTMBlockCellWrapper(self._lstm_cell, self._input_embedding)

    # Create the LSTM layer with input embedding and target size
    self._lstm_layer_with_embedding_and_target_size = tf.nn.rnn_cell.LSTMBlockCellWrapper(self._lstm_cell, self._input_embedding, self._target_size)

    # Create the LSTM layer with input embedding, target size, and num_filters
    self._lstm_layer_with_embedding_and_target_size_and_num_filters = tf.nn.rnn_cell.LSTMBlockCellWrapper(self._lstm_cell, self._input_embedding, self._target_size, self._num_filters)

    # Create the LSTM layer with input embedding, target size, num_filters, and num_filters_size
    self._lstm_layer_with_embedding_and_target_size_and_num_filters_and_num_filters_size = tf.nn.rnn_cell.LSTMBlockCellWrapper(self._lstm_cell, self._input_embedding, self._target_size, self._num_filters, self._num_filters_size)

    # Create the LSTM layer with input embedding, target size, num_filters, num_filters_size, and num_filters_stride
    self._lstm_layer_with_embedding_and_target_size_and_num_filters_and_num_filters_size_and_num_filters_stride = tf.nn.rnn_cell.LSTMBlockCellWrapper(self._lstm_cell, self._input_embedding, self._target_size, self._num_filters, self._num_filters_size, self._num_filters_stride)

```

Part-III (2)

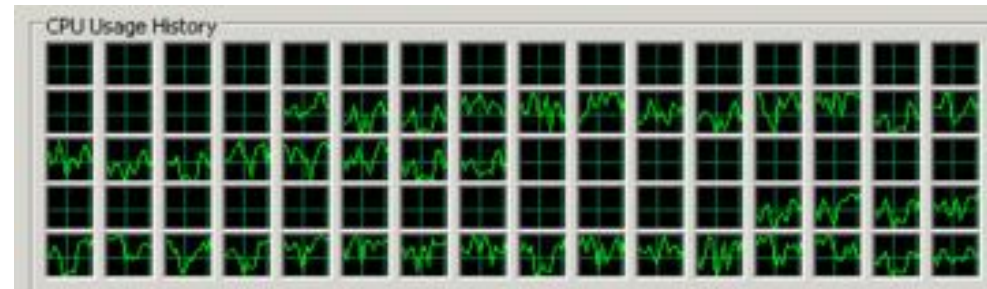
Distributed Deep Learning

Outline

- **Overview: Distributed Deep Learning on GPUs**
- Challenges 1: Addressing the communication bottleneck
- Challenges 2: Handling the limited GPU memory

Review – DL toolkits on single machine

- Using GPU is a must
 - A small number of GPU-equipped machines could achieve satisfactory speedup compared to CPU clusters with thousands of cores

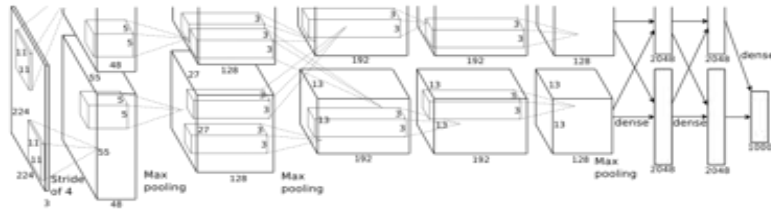


- A cluster of 8 GPU-equipped machines
- A cluster of 2000 CPU cores

More readily
available to
researchers

Review – DL toolkits on single machine

- However, using a single GPU is far from sufficient
 - average-sized deep networks can take days to train on a single GPU when faced with 100s of GBs to TBs of data
 - Demand faster training of neural networks on ever-larger datasets



AlexNet, 5 – 7 days



GoogLeNet, 10+ days

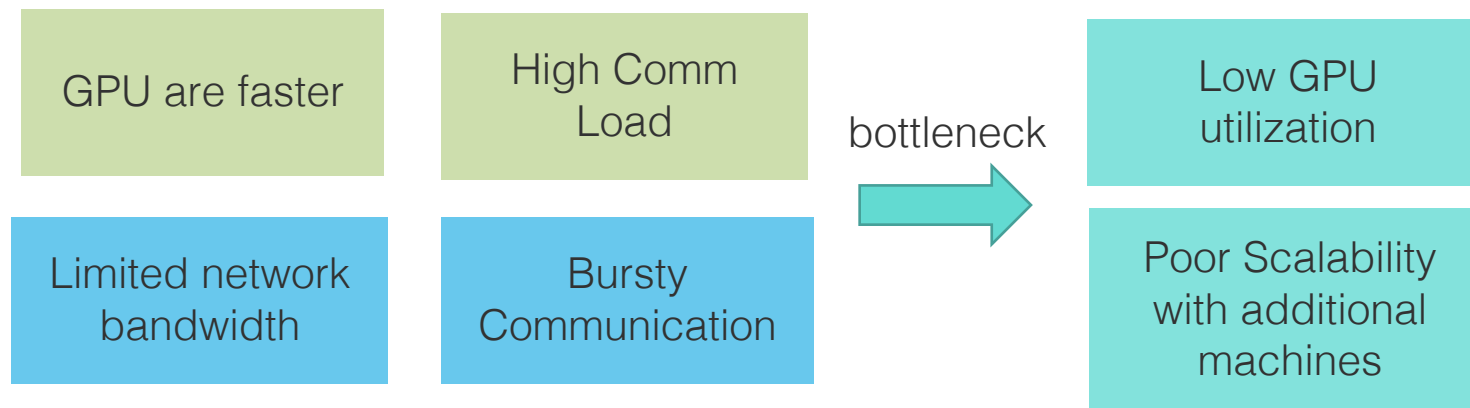
- However, current distributed DL implementations (e.g. in TensorFlow) can scale poorly due to substantial parameter synchronization over the network (we will show later)

Outline

- Overview: Distributed Deep Learning on GPUs
- **Challenges 1: Addressing the communication bottleneck**
- Challenges 2: Handling the limited GPU memory

Challenges

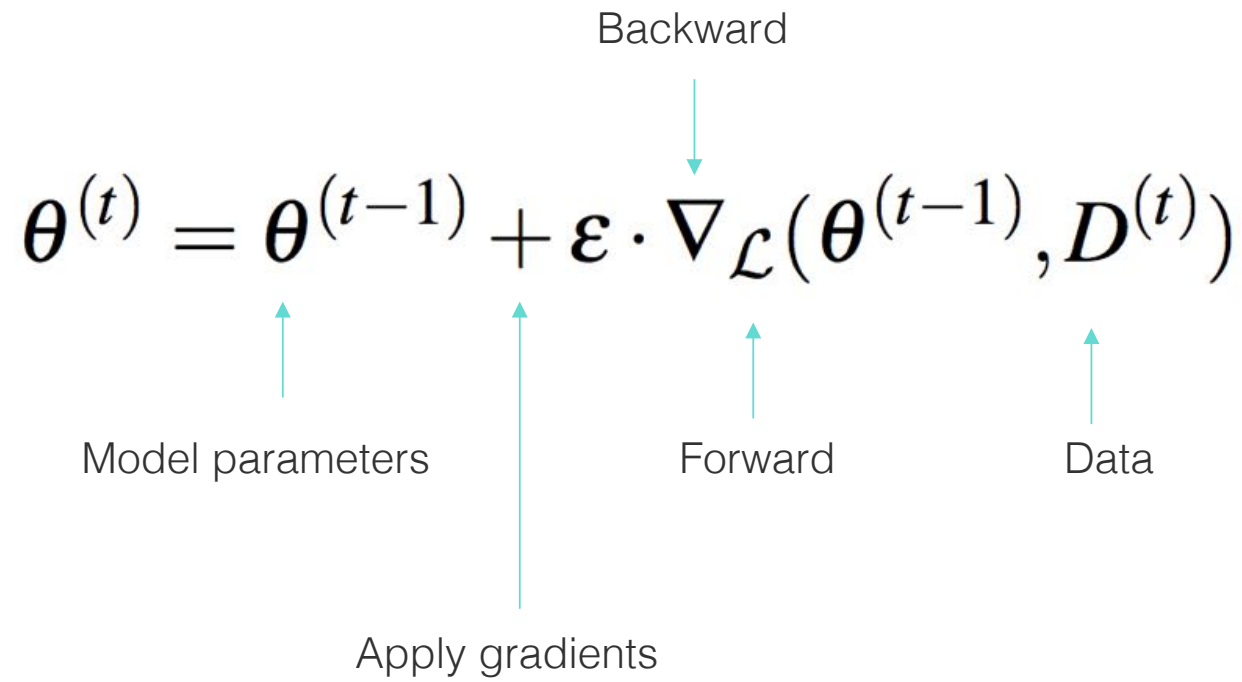
- Communication challenges
 - GPUs are at least one order of magnitude faster than CPUs



- High communication load raises the network communication as the main bottleneck given limited bandwidth of commodity Ethernet
- Managing the computation and communication in a distributed GPU cluster often complicates the algorithm design

Let's see what causes the problem

- Deep Learning on a single node – an iterative-convergent formulation

$$\boldsymbol{\theta}^{(t)} = \boldsymbol{\theta}^{(t-1)} + \varepsilon \cdot \nabla_{\mathcal{L}}(\boldsymbol{\theta}^{(t-1)}, \mathbf{D}^{(t)})$$


Backward

Model parameters

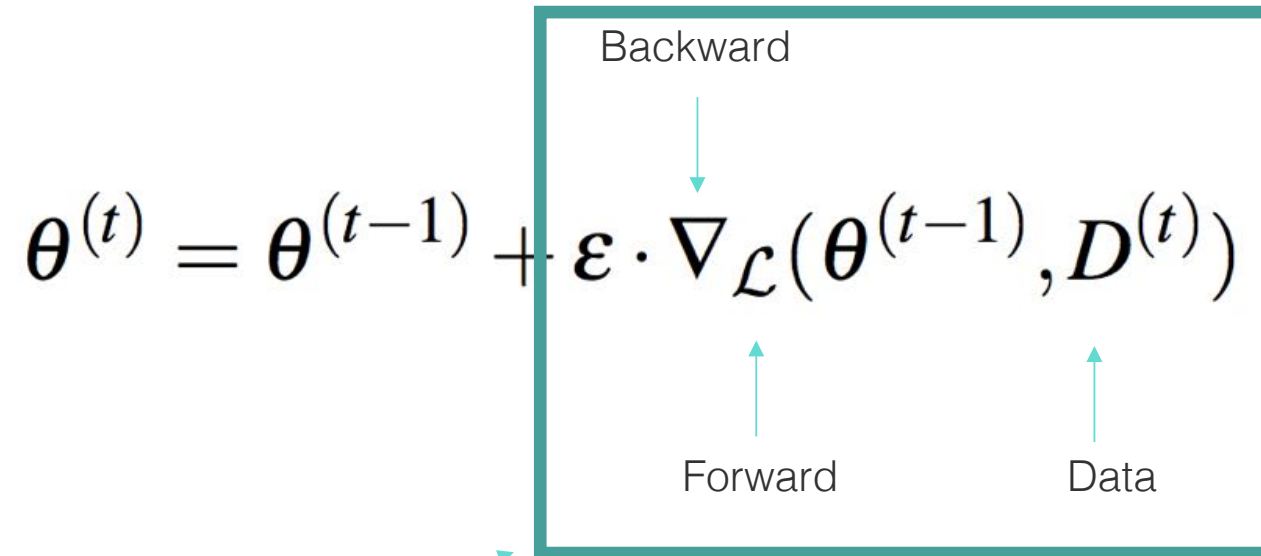
Forward

Data

Apply gradients

Let's see what causes the problem

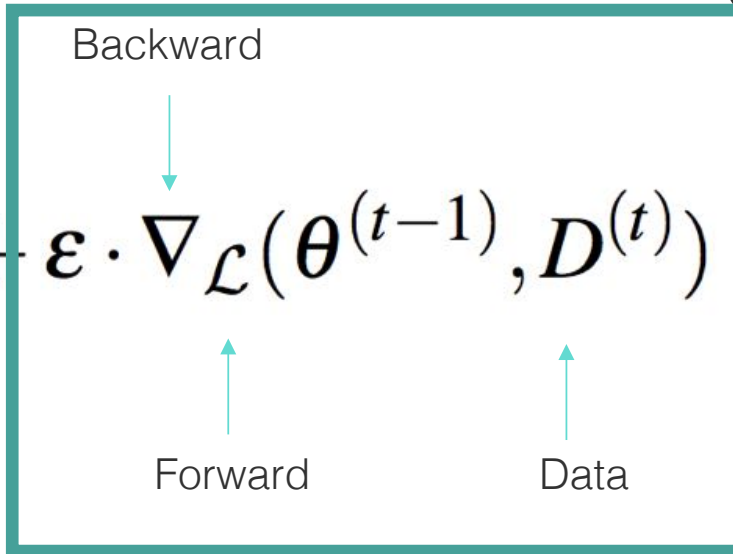
- Deep Learning on a single node – an iterative-convergent formulation

$$\theta^{(t)} = \theta^{(t-1)} + \varepsilon \cdot \nabla_{\mathcal{L}}(\theta^{(t-1)}, D^{(t)})$$


Forward and backward are the main computation (99%) workload of deep learning programs.

Distributed Deep Learning

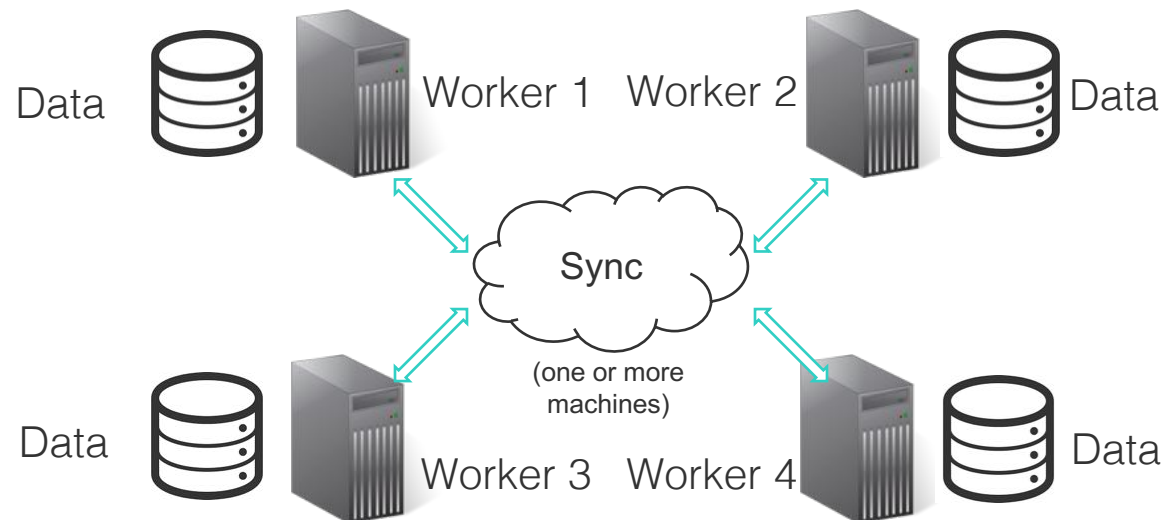
- Distributed DL: parallelize DL training using multiple machines.
- i.e. we want to accelerate the heaviest workload (in the box) to multiple machines

$$\boldsymbol{\theta}^{(t)} = \boldsymbol{\theta}^{(t-1)} + \varepsilon \cdot \nabla_{\mathcal{L}}(\boldsymbol{\theta}^{(t-1)}, \mathbf{D}^{(t)})$$


Forward and backward are the main computation (99%) workload of deep learning programs.

Data parallelism with stochastic gradient descent

- We usually seek a parallelization strategy called data parallelism, based on SGD
 - We partition data into different parts
 - Let different machines compute the gradient updates on different data partitions
 - Then aggregate/sync.



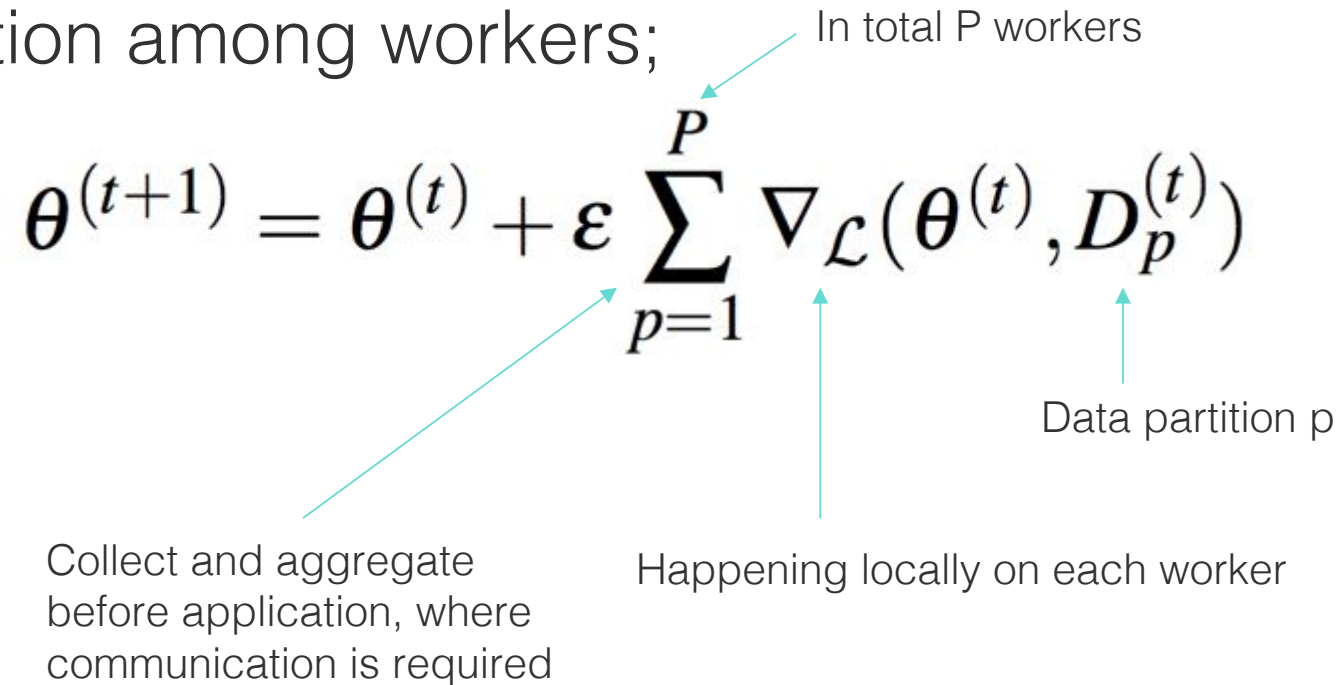
Data Parallel SGD

- Data parallel stochastic gradient descent
- Data-parallelism requires every worker to have read and write access to the shared model parameters θ , which causes communication among workers;

$$\theta^{(t+1)} = \theta^{(t)} + \varepsilon \sum_{p=1}^P \nabla_{\mathcal{L}}(\theta^{(t)}, D_p^{(t)})$$

In total P workers
Data partition p

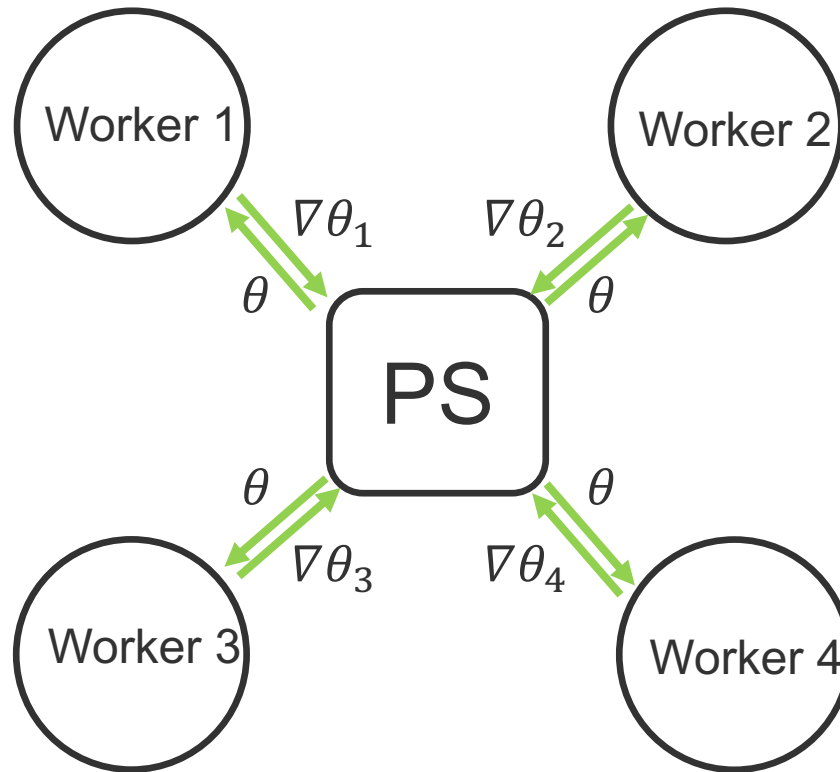
Collect and aggregate before application, where communication is required
Happening locally on each worker



How to communicate

- Parameter server, e.g. Bosen, SSP
 - A parameter server (PS) is a shared memory system that provides a shared access for the global model parameters θ
- Deep learning can be trivially data-parallelized over distributed workers using PS by 3 steps:
 - Each worker computes the gradients (∇L) on their own data partition (D_p) and send them to remote servers;
 - servers receive the updates and apply (+) them on globally shared parameters;
 - Each worker pulls back the updated parameters (θ_t)

How PS works



Parameter Server

- Parameter server has been successful for CPU-based deep learning
 - Google Distbelief, Dean et al. 2012
 - Scale up to thousands of CPU machines and 16000 CPU cores
 - SSPTable, Ho et al, 2013
 - Stale-synchronous parallel consistency model
 - Microsoft Adam, Chilimbi et al. 2014
 - 63 machines, state-of-art results on ImageNet 22K
 - Bosen, Wei et al. 2015
 - Managed communication

Parameter Server on GPUs

- Directly applying parameter server for GPU-based distributed deep learning will underperform (as will show later).
 - GPU is too fast
 - Ethernet bandwidth is limited, and has latency
- For example
 - AlexNet: 61.5M float parameters, 0.25s/iteration on Geforce Titan X (batchsize = 256)
 - Gradient generation rate: 240M float/(s*GPU)
 - Parallelize it over 8 machines each w/ one GPU using PS.
 - To ensure the computation not blocked on GPU (i.e. linear speed-up with additional nodes)
 - As a worker: send 240M floats/s and pull back 240M floats/s (at least)
 - As a server: receive 240M * 8 floats/s and send back 240M * 8/s (at least)

Parameter Server on GPUs

- Let's see where we are

This is what the GPU workstation in your lab has

Ethernet standards

Ethernet	Rate(GBit/s)	Rate (Mb/s)	Rate (# floats/s)
1 GbE	1	125	31.25M
10 GbE	10	1250	312.5M
Infiniband	40	5000	1250M

One of the most expensive instances AWS could provide you (18\$/h?)

Specialized hardware! Non-commodity anymore, inaffordable

Parameter Server on GPUs

The problem is more severe than described above

- We only use 8 nodes (which is small). How about 32, 128, or even 256?
- We haven't considered other issues (which might be also troublesome), e.g.
 - Memory copy between DRAM and GPU will have a non-trivial cost
 - The Ethernet might be shared with other tasks, i.e. available bandwidth is even less.
 - Burst communication happens very often on GPUs (which will explain later).

Address the Communication Bottleneck

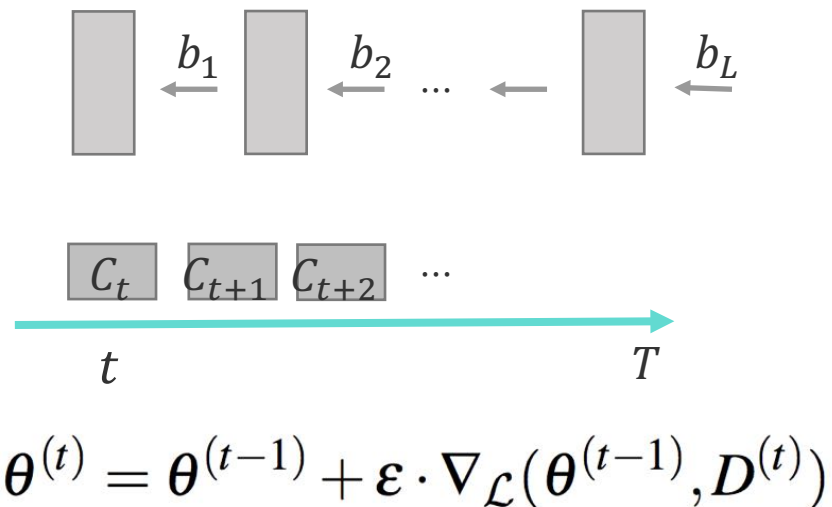
- A simple fact:
 - Communication time may be reduced, but cannot be eliminated (of course)
- Therefore, **possible** ideas to address the communication bottleneck
 - Hide the communication time by overlapping it with the computation time
 - Reduce the size of messages needed to be communications

Address the Communication Bottleneck

- A simple fact:
 - Communication time may be reduced, but cannot be eliminated (of course).
- Therefore, possible ideas to address the communication bottleneck
 - **Hide the communication time by overlapping it with the computation time**
 - Reduce the size of messages needed to be communications

Overlap Computation and Communication

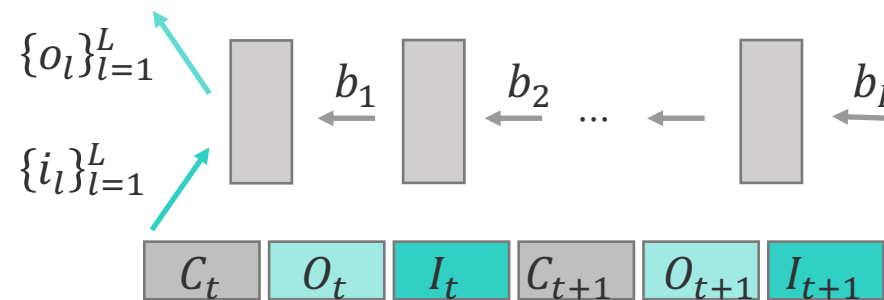
- Revisit on a single node the computation flow of BP
 - b_l : backpropagation computational through layer l
 - C_t : forward and backward computation at iteration t



Overlap Computation and Communication

- On multiple nodes, when communication is involved
- Introduce two communication operations
 - o_l : send out the gradients in layer l to the remote
 - i_l : pull back the globally shared parameters of layer l from the remote
 - O_t : the set $\{o_l\}_{l=1}^L$ at iteration t
 - I_t : the set $\{i_l\}_{l=1}^L$ at iteration t

$$\theta^{(t)} = \theta^{(t-1)} + \varepsilon \cdot \nabla_{\mathcal{L}}(\theta^{(t-1)}, D^{(t)})$$



Computation and communication happen sequentially!

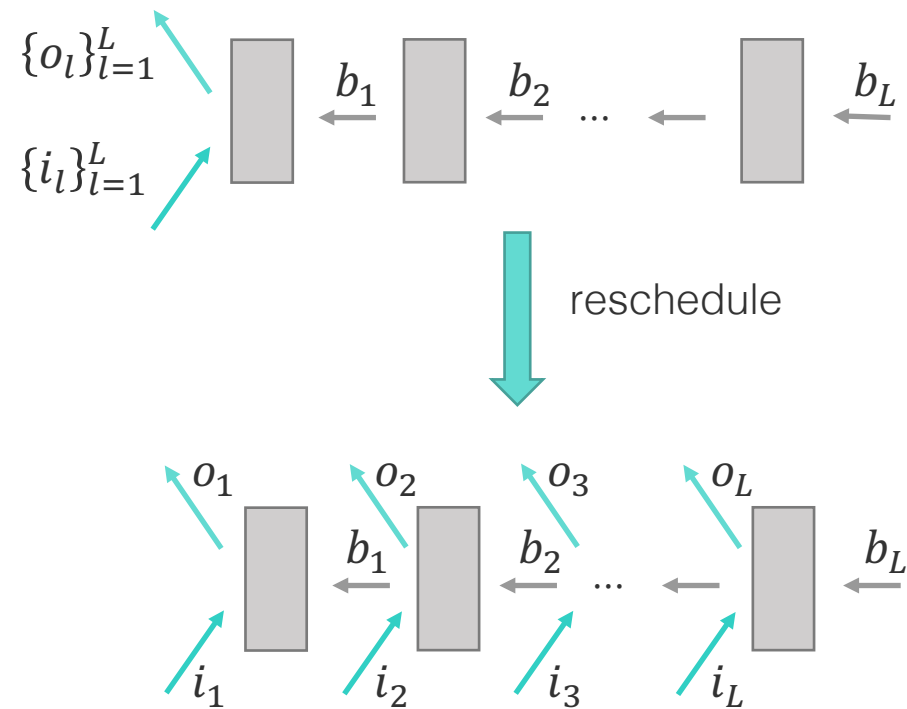
Overlap Computation and Communication

- Note the following independency
 - The send-out operation o_l is independent of backward operations
 - The read-in operation i_l could update the layer parameters as long as b_l was finished, without blocking the subsequent backward operations b_i ($i < l$)
- Idea: overlap computation and communication by utilizing concurrency
 - Pipelining the updates and computation operations

$$\theta^{(t)} = \theta^{(t-1)} + \varepsilon \cdot \nabla_{\mathcal{L}}(\theta^{(t-1)}, D^{(t)})$$

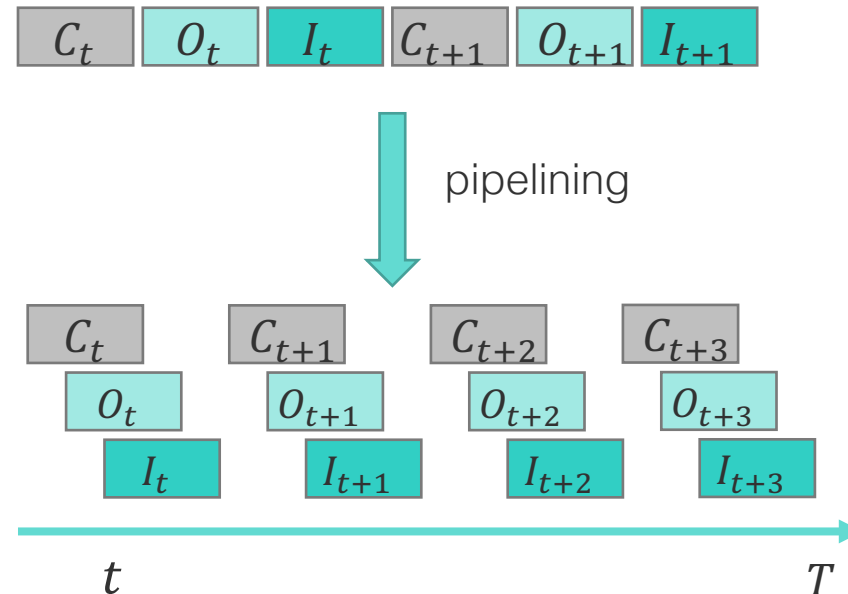
WFBP: Wait-free backpropagation

- Idea: overlap computation and communication by utilizing concurrency
 - Pipelining the updates and computation operations



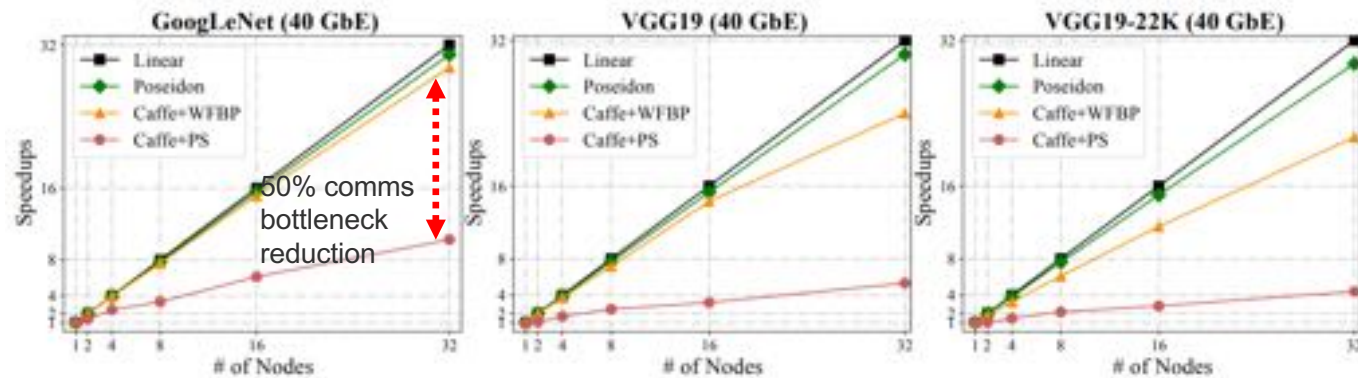
WFBP: Wait-free backpropagation

- Idea: overlap computation and communication by utilizing concurrency
 - Communication overhead is hidden under computation
 - Results: more computations in unit time



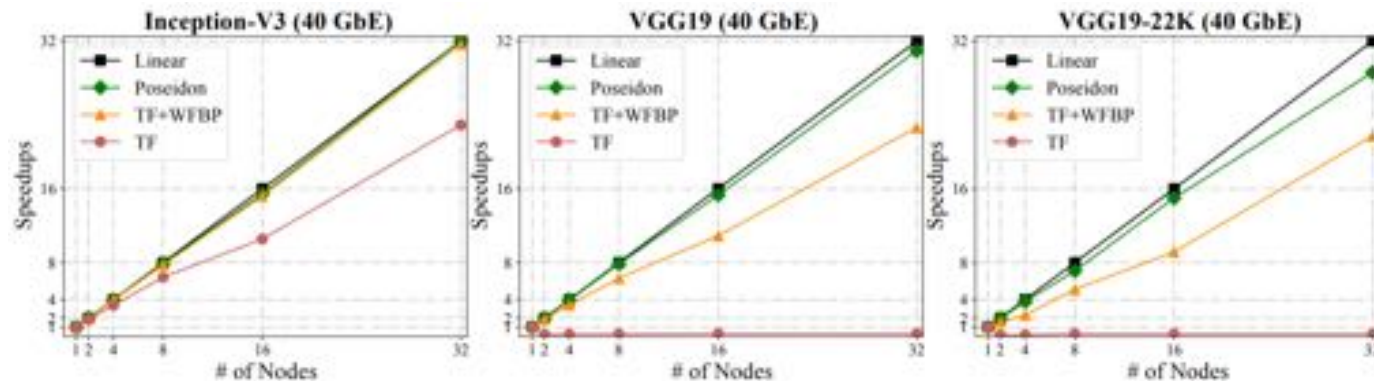
WFBP: Distributed Wait-free backpropagation

- How does WFBP perform?
 - Using Caffe as an engine:



Zhang et al. 2017

- Using TensorFlow as engine:



WFBP: Distributed Wait-free backpropagation

- Observation: Why DWBP would be effective
 - More statistics of modern CNNs

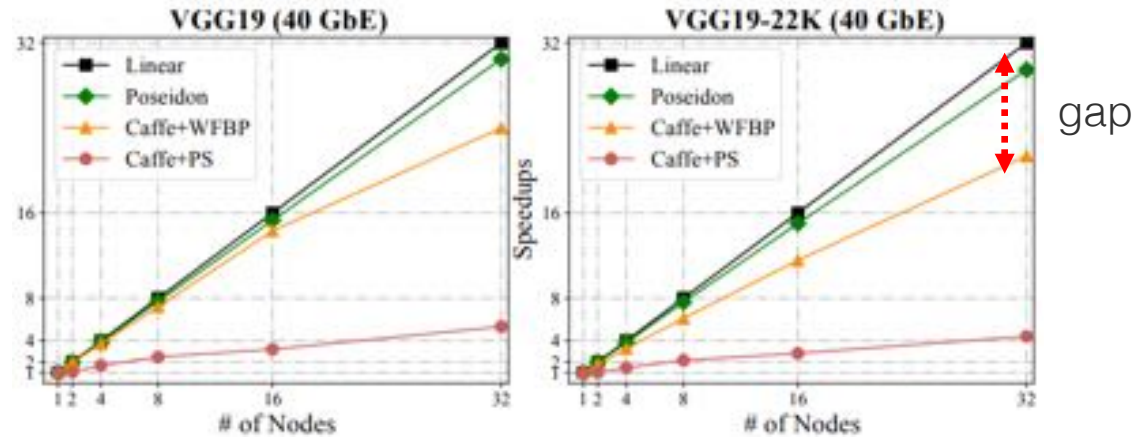
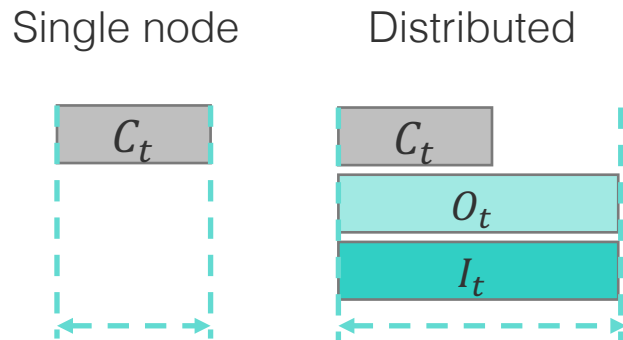
Params/FLOP distribution of modern CNNs

Parameters	CONV Layers (#/%)	FC Layers (#/%)
AlexNet	2.3M / 3.75	59M / 96.25
VGG-16	7.15M / 5.58	121.1M / 94.42
FLOPs	CONV Layers (#/%)	FC Layers (#/%)
AlexNet	1,352M / 92.0	117M / 8.0
VGG-16	10,937M / 91.3	121.1M / 8.7

- 90% computation happens at bottom layers
- 90% communication happens at top layers
- WFBP overlaps 90% and 90%

WFBP: Wait-free Backpropagation

- Does overlapping communication and computation solve all the problems?
 - When communication time is longer than computation, no (see the figure below).
 - Say, if communication and computation are perfectly overlapped, how many scalability we can achieve?



Address the communication bottleneck

- Note a simple fact:
 - Communication time may be reduced, but cannot be eliminated (of course).
- Therefore, possible ideas to address the communication bottleneck
 - Hide the communication time by overlapping it with the computation time – which we have described before.
 - **Reduce the size of messages needed to be communications**
 - **While without compromising statistical convergence**

Introducing Sufficient Factor Broadcasting

- Matrix-parametrized models

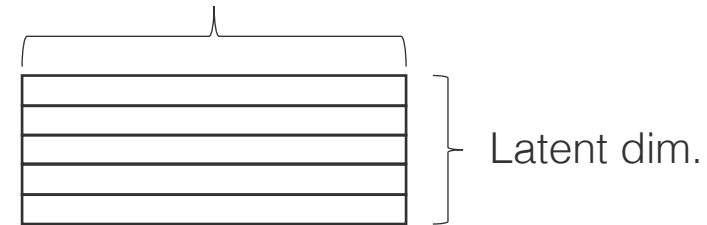
Multiclass Logistic Regression

Feature dim.



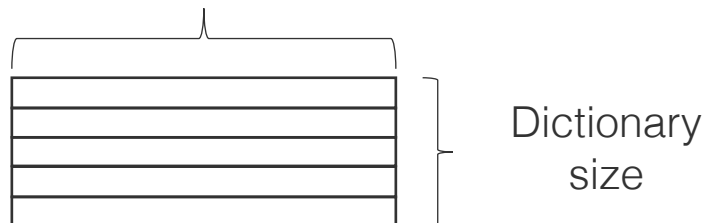
Distance Metric Learning

Feature dim.



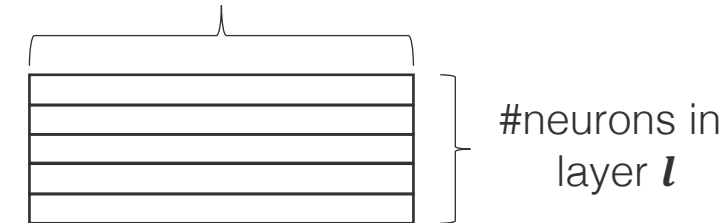
Sparse Coding

Feature dim.



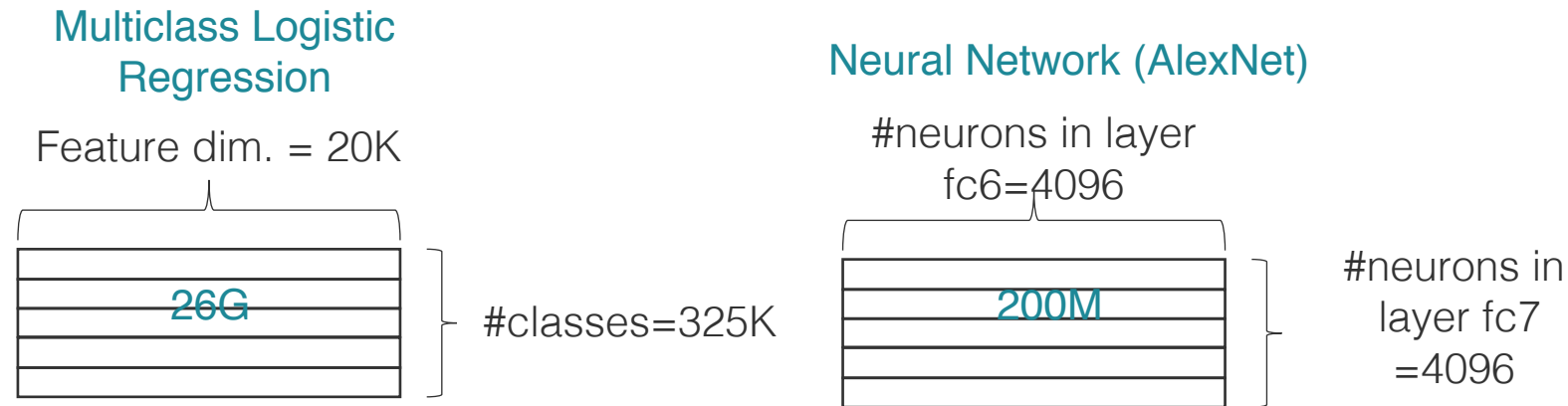
Neural Network

#neurons in layer $l - 1$



Distributed Learning of MPMs

- Learning MPMs by communicating parameter matrices between server and workers
 - Dean and Ghemawat, 2008; Dean et al, 2012; Sindhvani and Ghoting, 2012; Gopal and Yang, 2013; Chilimbi et al, 2014, Li et al, 2015
- High communication cost and large synchronization delays



Contents:

Sufficient Factor (SF) Updates

Full parameter matrix update ΔW can be computed as **outer product of two vectors uv^T** (called sufficient factors)

- Example: Primal stochastic gradient descent (SGD)

$$\min_W \frac{1}{N} \sum_{i=1}^N f_i(Wa_i; b_i) + h(W)$$

$$\Delta W = uv^T \quad u = \frac{\partial f(Wa_i, b_i)}{\partial (Wa_i)} \quad v = a_i$$

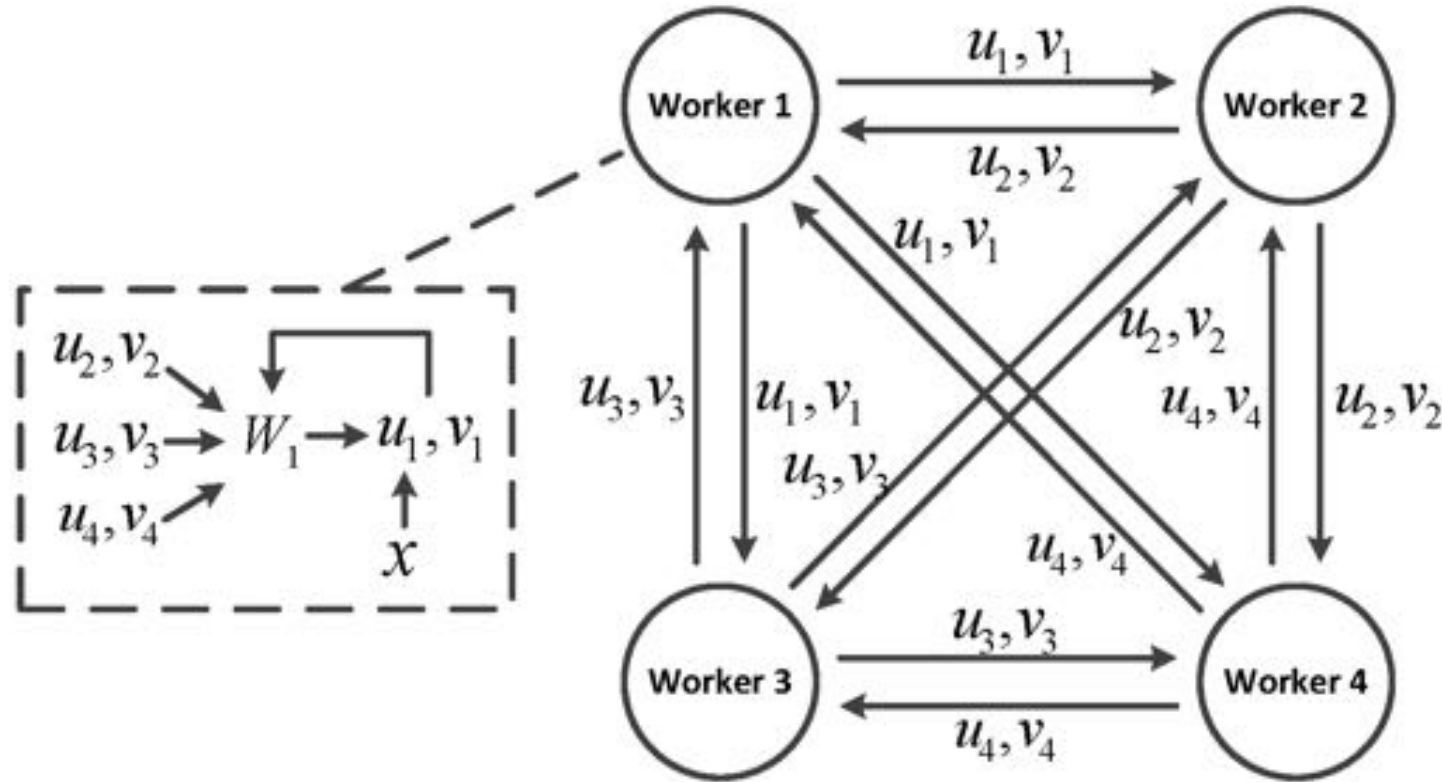
- Example: Stochastic dual coordinate ascent (SDCA)

$$\min_Z \frac{1}{N} \sum_{i=1}^N f_i^*(-z_i) + h^*\left(\frac{1}{N} ZA^T\right)$$

$$\Delta W = uv^T \quad u = \Delta z_i \quad v = a_i$$

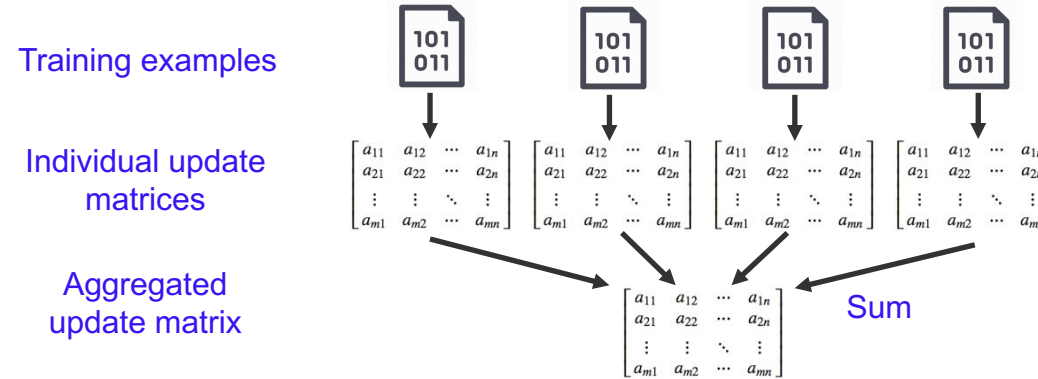
Send lightweight SF updates (u, v) , instead of expensive full-matrix ΔW updates!

Sufficient Factor Broadcasting: P2P Topology + SF Updates

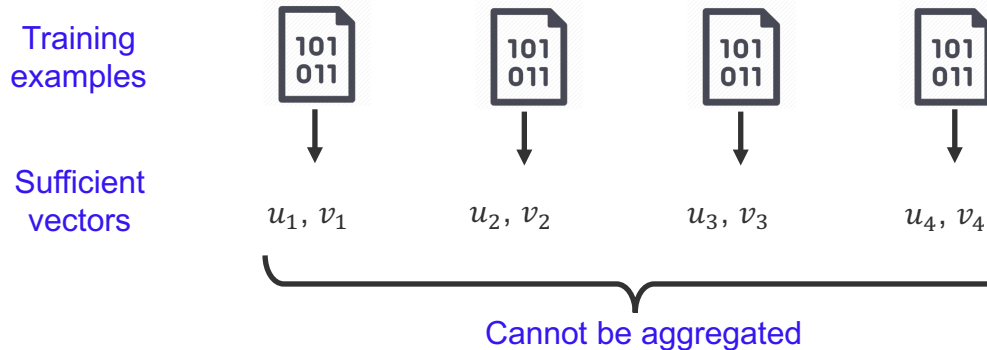


A computing & communication tradeoff

- Full update:



- Pre-update

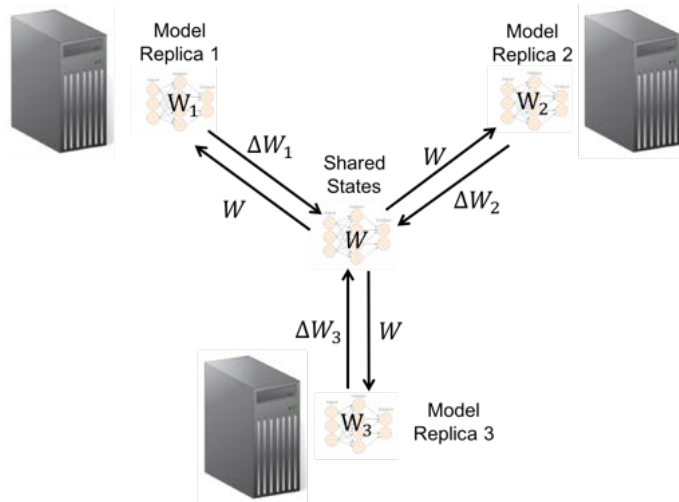


- Stochastic algorithms
 - Mini-batch: C samples

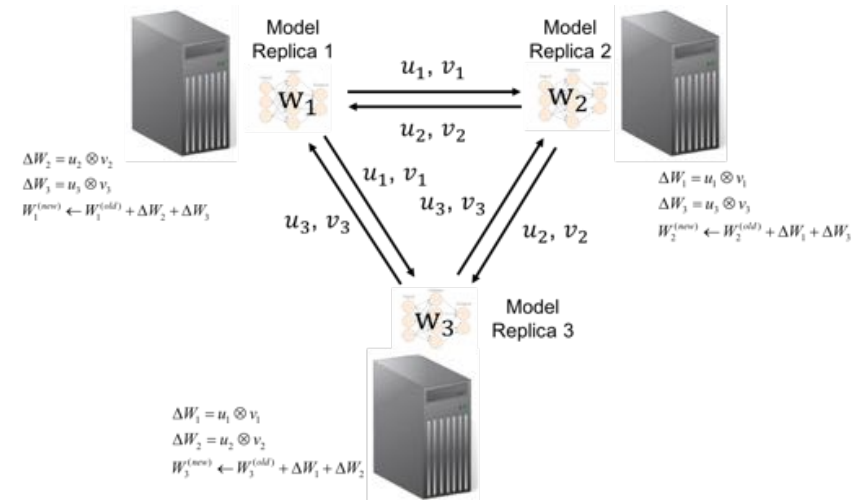
Matrix Representation	$O(JK)$
SV Representation	$O((J + K)C)$

Synchronization of Parameter Replicas

parameter server



Transfer SVs instead of ΔW



- **A Cost Comparison**

	Size of one message	Number of messages	Network Traffic
P2P SV-Transfer	$O(J + K)$	$O(P^2)$	$O((J + K)P^2)$
Parameter Server	$O(JK)$	$O(P)$	$O(JKP)$

Convergence Speedup



- 3 Benchmark ML Programs
 - Big parameter matrices with 6.5-8.6b entries (30+GB), running on 12- & 28-machine clusters
- 28-machine SFB finished in 2-7 hours
 - Up to 5.6x faster than 28-machine PS, 12.3x faster than 28-machine Spark
- PS cannot support SF communication, which requires decentralized storage

Convergence Guarantee

- Assumptions
 - Bridging model
 - Staleness Synchronous Parallel (SSP) with staleness parameter s
 - Bulk Synchronous Parallel is a special case of SSP when $s = 0$
 - Communication methods
 - Partial broadcast (PB): sending messages to a subset of Q ($Q < P - 1$) machines

• Full broadcast is a special case of PB when $Q = P - 1$

Assumption 1. (1) For all j , f_j is continuously differentiable and F is bounded from below; (2) $\nabla F, \nabla F_p$ are Lipschitz continuous with constants L_F and L_p , respectively, and let $L = \sum_{p=1}^P L_p$; (3) There exists G, σ^2 such that for all p and c , we have (almost surely) $\|U_p(\mathbf{W}_p^c, I_p^c)\| \leq G\eta$ and $\mathbb{E} \| |S_p| \sum_{j \in I_p} \nabla f_j(\mathbf{W}) - \nabla F_p(\mathbf{W}) \|_2^2 \leq \sigma^2$.

Convergence Guarantee

- Results

Theorem 1. *Let Assumption 1 hold, and let $\{\mathbf{W}_p^c\}$, $p = 1, \dots, P$, $\{\mathbf{W}^c\}$ be the local sequences and the auxiliary sequence, respectively.*

Under full broadcasting (i.e., $Q = P - 1$) and set the learning rate $\eta := \eta_c = O(\sqrt{\frac{1}{L\sigma^2 P s c}})$, we have

- $\liminf_{c \rightarrow \infty} \mathbb{E} \|\nabla F(\mathbf{W}^c)\| = 0$, hence there exists a subsequence of $\nabla F(\mathbf{W}^c)$ that almost surely vanishes;
- $\lim_{c \rightarrow \infty} \max_p \|\mathbf{W}^c - \mathbf{W}_p^c\| = 0$, i.e., the maximal disagreement between all local sequences and the auxiliary sequence converges to 0 (almost surely);
- There exists a common subsequence of $\{\mathbf{W}_p^c\}$ and $\{\mathbf{W}^c\}$ that converges almost surely to a stationary point of F , with the rate $\min_{c \leq C} \mathbb{E} \|\sum_{p=1}^P \nabla F_p(\mathbf{W}_p^c)\|_2^2 \leq O\left(\sqrt{\frac{L\sigma^2 P s}{C}}\right)$

Under partial broadcasting (i.e., $Q < P - 1$) and set a constant learning rate $\eta = \frac{1}{CLG(P-Q)}$, where C is the total number of iterations. Then we have

$$\min_{c \leq C} \mathbb{E} \left[\|\sum_{p=1}^P \nabla F_p(\mathbf{W}_p^c)\|_2^2 \right] \leq O\left(LG(P-Q) + \frac{P(sG + \sigma^2)}{CG(P-Q)} \right).$$

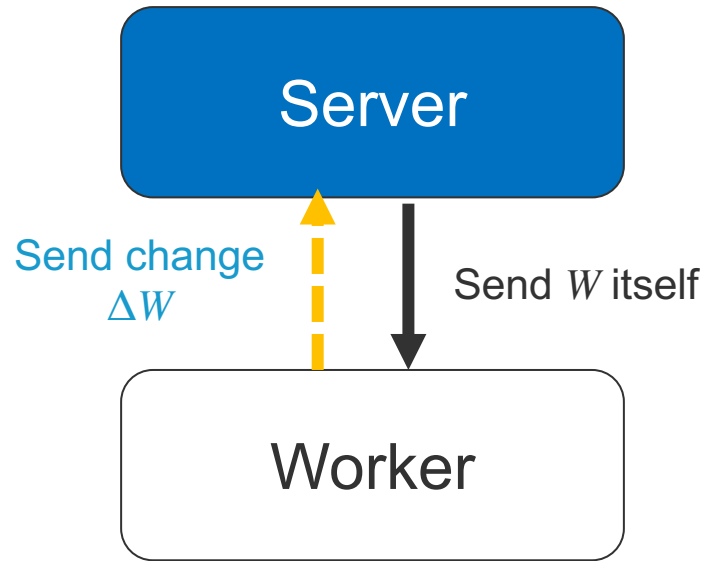
Hence, the algorithm converges to a $O(LG(P-Q))$ neighbourhood if $C \rightarrow \infty$.

Convergence Guarantee

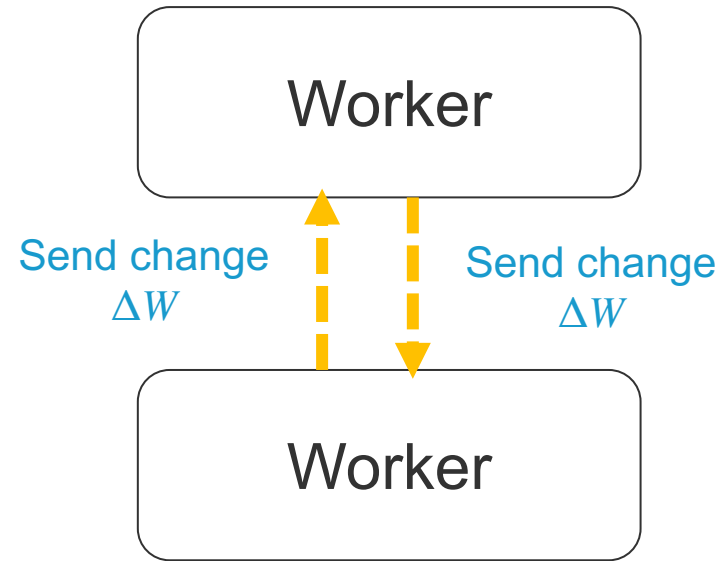
- Take-home message:
 - Under full broadcasting, given a properly-chosen learning rate, all local worker parameters W_p^c eventually converge to stationary points (i.e. local minima) of the objective function, despite the fact that SV transmission can be delayed by up to s iterations.
 - Under partial broadcasting, the algorithm converges to a $O(LG(P - Q))$ neighbourhood if $C \rightarrow \infty$.

Parameter Storage and Communication Paradigms

Centralized Storage

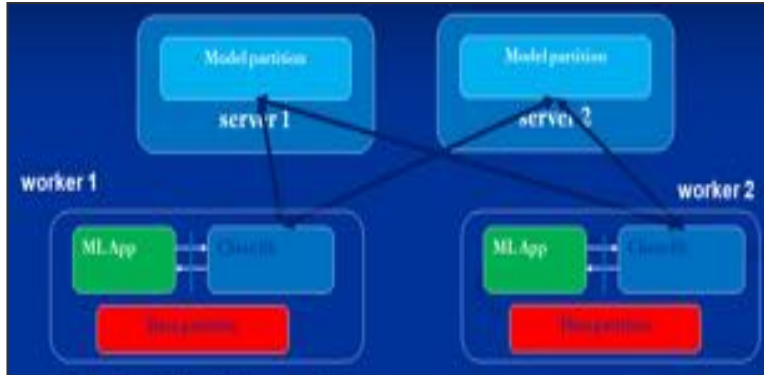


Decentralized Storage



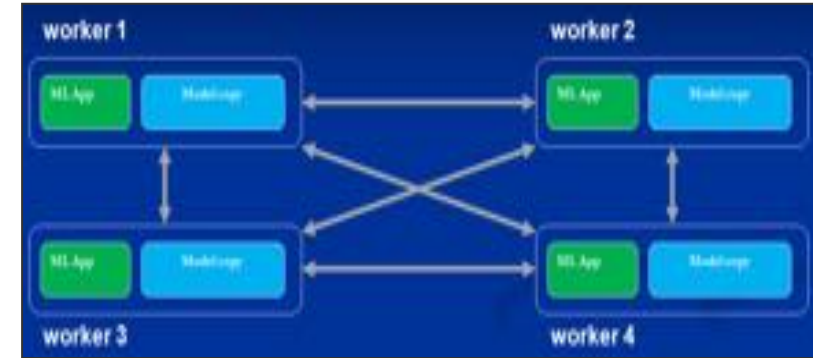
- **Centralized:** send parameter W itself from server to worker
 - Advantage: allows compact comms topology, e.g. bipartite
- **Decentralized:** always send changes ΔW between workers
 - Advantage: more robust, homogeneous code, low communication (?)

Topologies: Master-Slave versus P2P?



Master-slave

- Used with **centralized storage** paradigm
- **Disadvantage:** need to code/manage clients and servers separately
- **Advantage:** bipartite topology is comms-efficient
- Popular for Parameter Servers: Yahoo LDA, Google DistBelief, Petuum PS, Project Adam, Li&Smola PS, ...

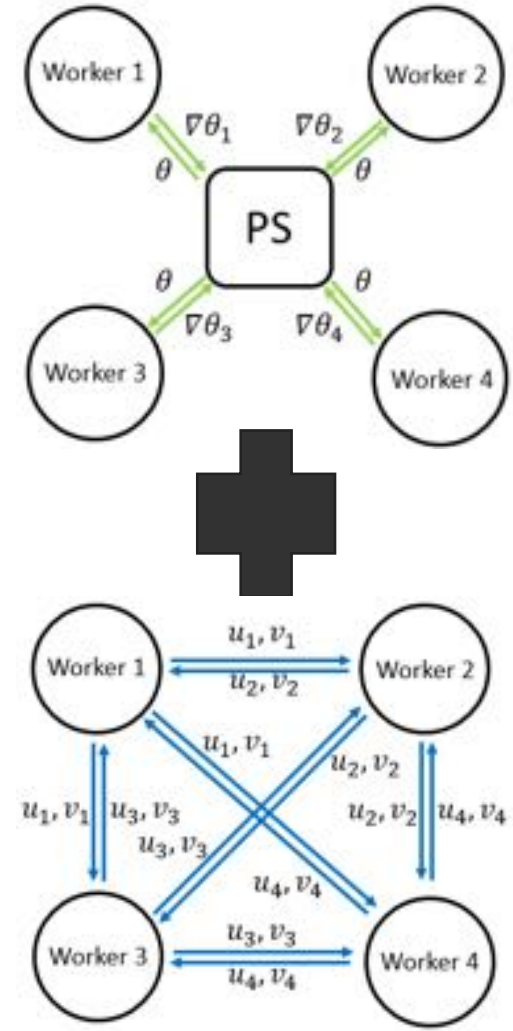


P2P

- Used with **decentralized storage**
- **Disadvantage (?):** high comms volume for large # of workers
- **Advantage:** same code for all workers; no single point of failure, high elasticity to resource adjustment
- Less well-explored due to perception of high communication overhead?

Hybrid Updates: PS + SFB

- Hybrid communications: Parameter Server + Sufficient Factor Broadcasting
 - Parameter Server: Master-Slave topology
 - Sufficient factor broadcasting: P2P topology
- For problems with a mix of large and small matrices,
 - Send small matrices via PS
 - Send large matrices via SFB



Hybrid example: CNN

Hao Zhang, Zhiting Hu, Jinliang Wei, Pengtao Xie, Gunhee Kim, Qirong Ho, Eric P. Xing. Poseidon: A System Architecture for Efficient GPU-based Deep Learning on Multiple Machines. USENIX ATC 2016.

- Example: AlexNet CNN model
 - Final layers = 4096 * 30000 matrix (120M parameters)
 - Use SFB to communicate
 - 1. Decouple into two 4096 vectors: u, v
 - 2. Transmit two vectors
 - 3. Reconstruct the gradient matrix

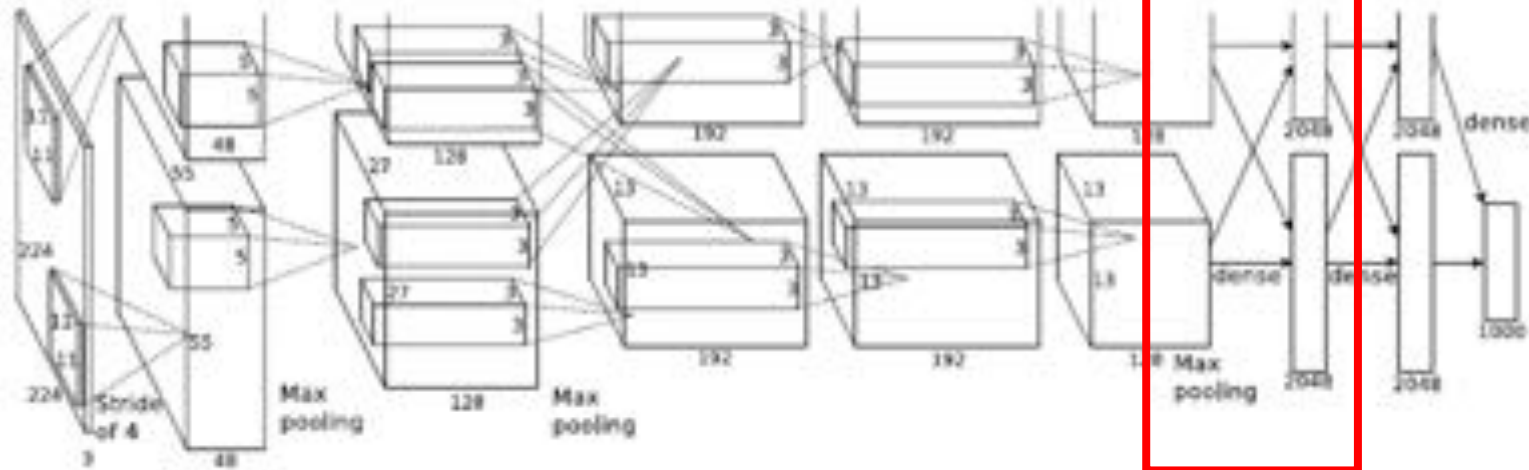


Figure from Krizhevsky et al. 2012

Hybrid example: CNN

Hao Zhang, Zhiting Hu, Jinliang Wei, Pengtao Xie, Gunhee Kim, Qirong Ho, Eric P. Xing. Poseidon: A System Architecture for Efficient GPU-based Deep Learning on Multiple Machines. USENIX ATC 2016.

- Example: AlexNet CNN model
 - Convolutional layers = e.g. $11 * 11$ matrix (121 parameters)
 - Use Full-matrix updates to communicate
 - 1. Send/receive using Master-Slave PS topology

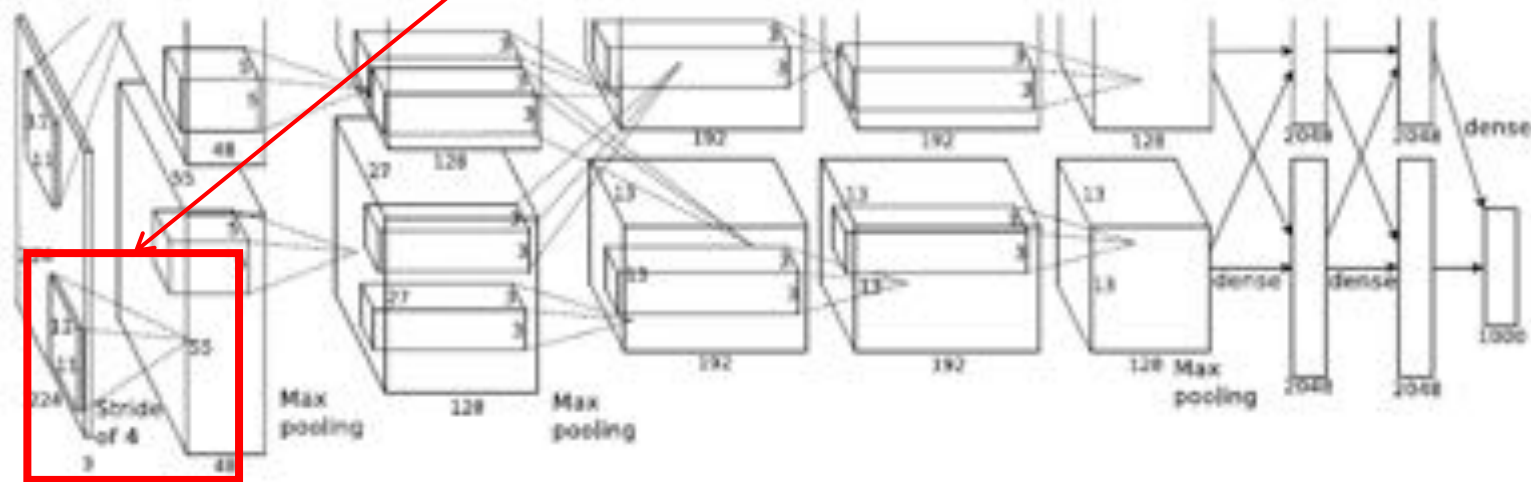
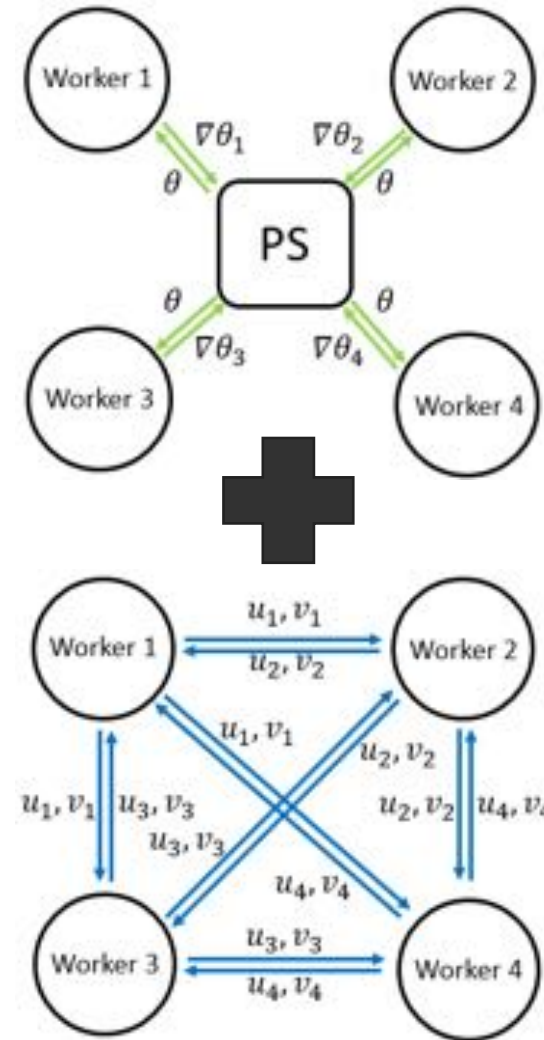


Figure from Krizhevsky et al. 2012

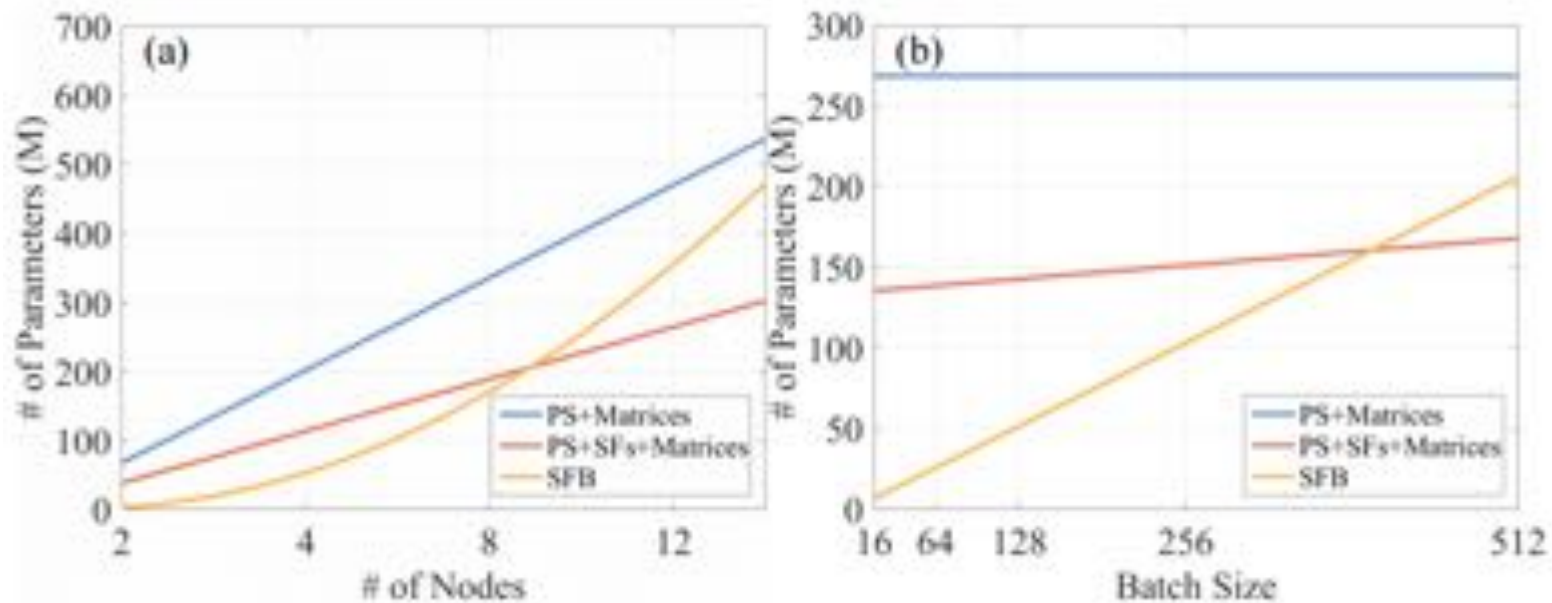
Hybrid Communication

- Idea
 - Sync FC layers using SFB
 - Sync Conv layer using PS
- Effectiveness
 - It directly reduces the size of messages in many situations
- Is SFB always optimal?
 - No, its communication load increases quadratically
 - The right strategy: choose PS whenever it results in less communication



Hybrid Communication

- A best of both worlds strategy
- For example, AlexNet parameters between FC6 and FC7
- Tradeoff between PS and SFB communication



Hybrid Communication

- How to choose? Where is the threshold?
- Determine the best strategy depending on
 - Layer type: CONV or FC?
 - Layer size
 - Batch size
 - # of Cluster nodes

Method	Server	Worker	Server & Worker
PS	$2P_1MN/P_2$	$2MN$	$2MN(P_1 + P_2 - 2)/P_2$
SFB	N/A	$2K(P_1 - 1)(M + N)$	N/A
Adam (max)	$P_1MN + P_1K(M + N)$	$K(M + N) + MN$	$(P_1 - 1)(MN + KM + KN)$

Table 1: Estimated communication cost of PS, SFB and Adam for synchronizing the parameters of a $M \times N$ FC layer on a cluster with P_1 workers and P_2 servers, when batchsize is K .

Hybrid Communication

- Hybrid communication algorithm

Determine the best strategy depending on

- Layer type: CONV or FC?
- Layer size: M, N
- Batch size: K
- # of Cluster nodes: P_1, P_2

Algorithm 1 Get the best comm method of layer l

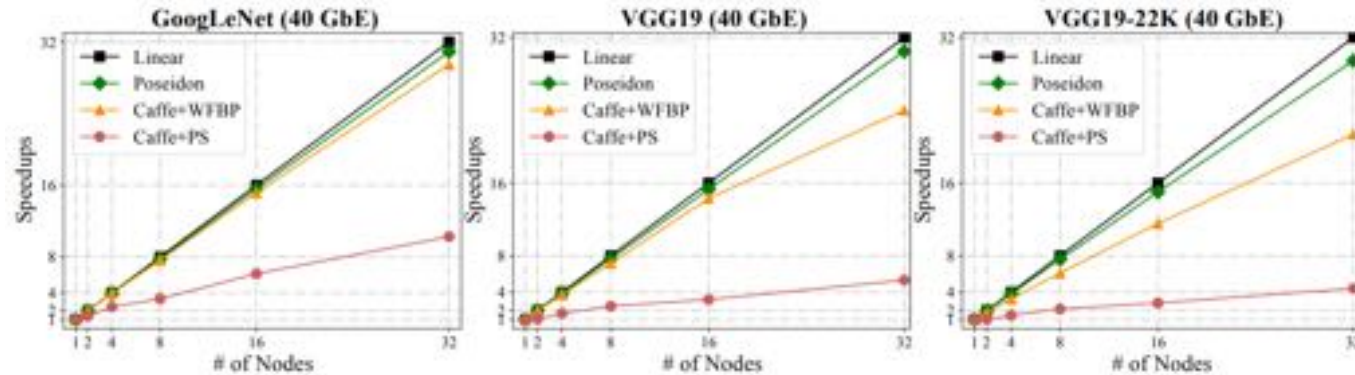
```

1: function BESTSCHEME( $l$ )
2:    $layer\_property = \text{Query}(l.name)$ 
3:    $P_1, P_2, K = \text{Query}('n\_worker', 'n\_server', 'batchsize')$ 
4:   if  $layer\_property.type == 'FC'$  then
5:      $M = layer\_property.width$ 
6:      $N = layer\_property.height$ 
7:     if  $2K(P_1 - 1)(M + N) \leq \frac{2MN(P_1 + P_2 - 2)}{P_2}$  then
8:       return 'SFB'
9:     end if
10:  end if
11:  return 'PS'
12: end function

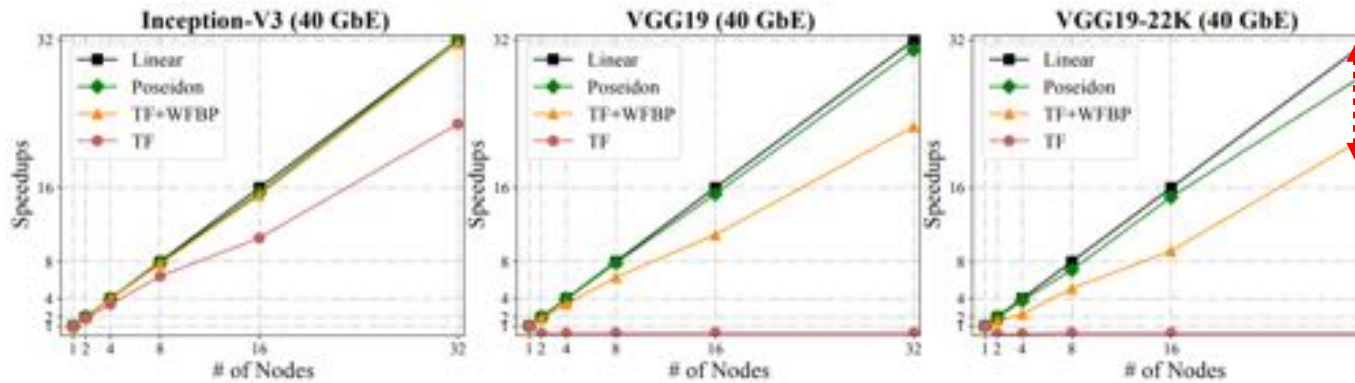
```

Hybrid Communication

- Results: achieve linear scalability across different models/data with 40GbE bandwidth
 - Using Caffe as an engine:



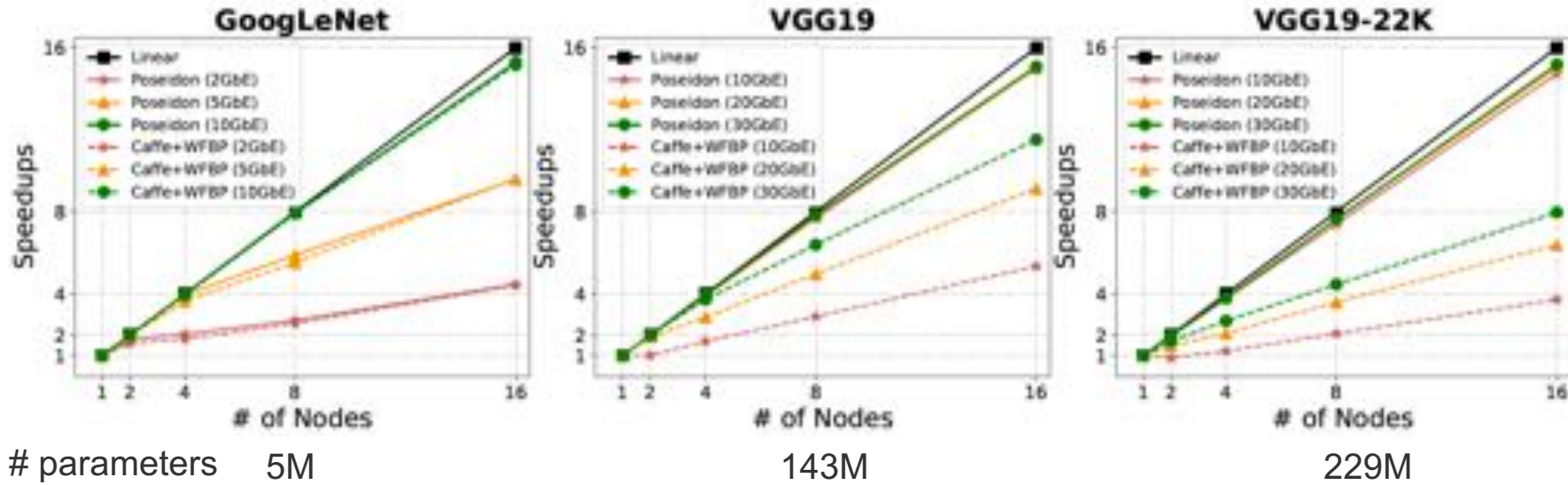
- Using TensorFlow as engine



Improve over WFBP

Hybrid Communication

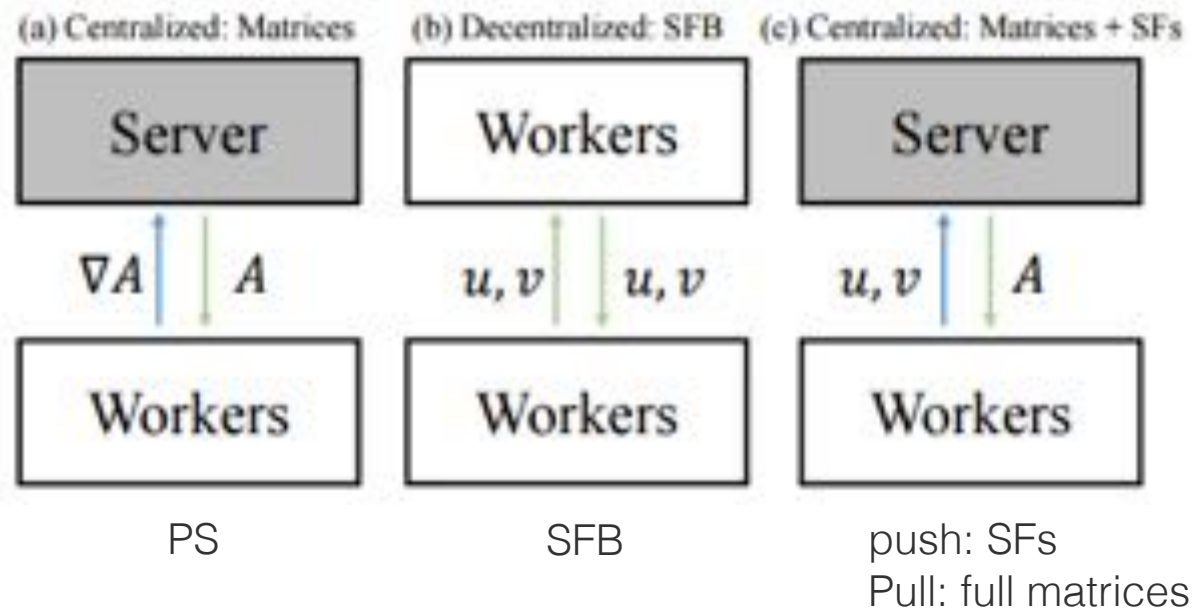
- Linear scalability on throughput, even with limited bandwidth!
 - Make distributed deep learning affordable



Ethernet	Rate(GBit/s)	Rate (Mb/s)	Rate (# floats/s)
1 GbE	1	125	31.25M
10 GbE	10	1250	312.5M
Infiniband	40	5000	1250M

Hybrid Communication

- Discussion: Utilizing SFs is not a new idea, actually
 - Microsoft Adam uses the third strategy (c)



Hybrid Communication

- Adam's strategy leads to communication bottleneck
 - Pushing SFs to server is fine
 - Pulling full matrices back will create a bottleneck on the server node.

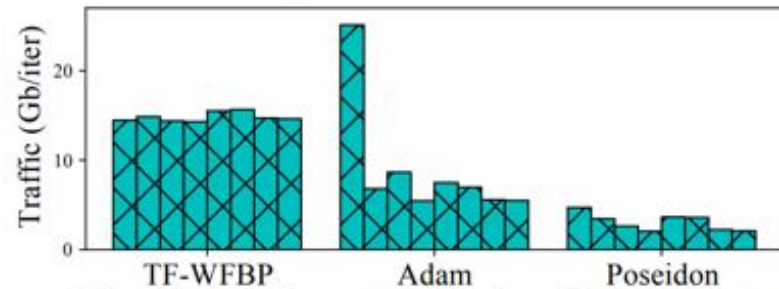
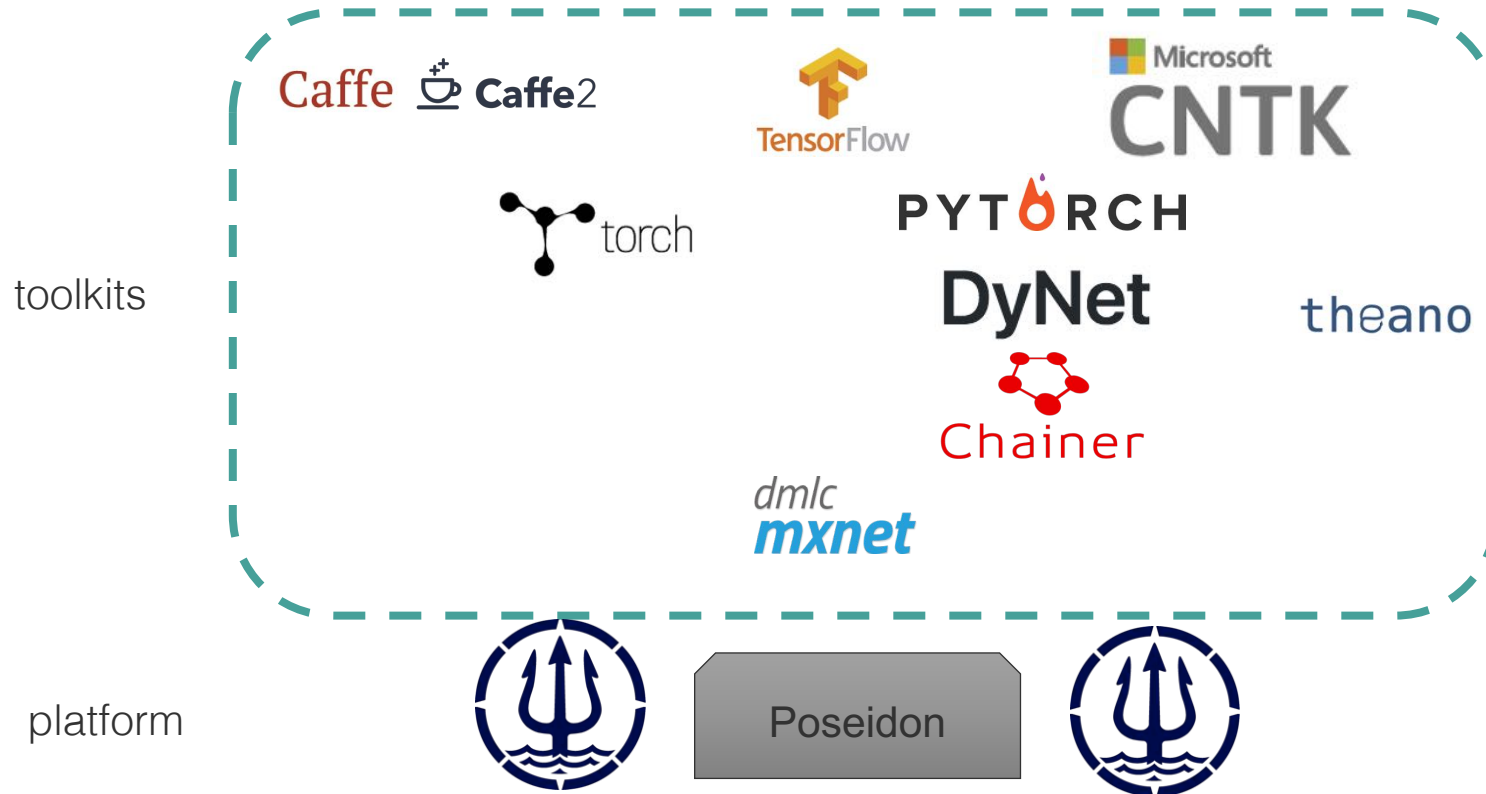


Figure 10: Averaged communication load when training VGG19 using *TF-WFBP*, *Adam* and *Poseidon* with TensorFlow engine. Each bar represents the network traffic on a node.

- Hybrid communication yields communication load balancing
 - Which is important to address the problem of burst communication.

Introducing Poseidon

- Poseidon: An efficient communication architecture
 - A distributed platform to amplify existing DL toolkits

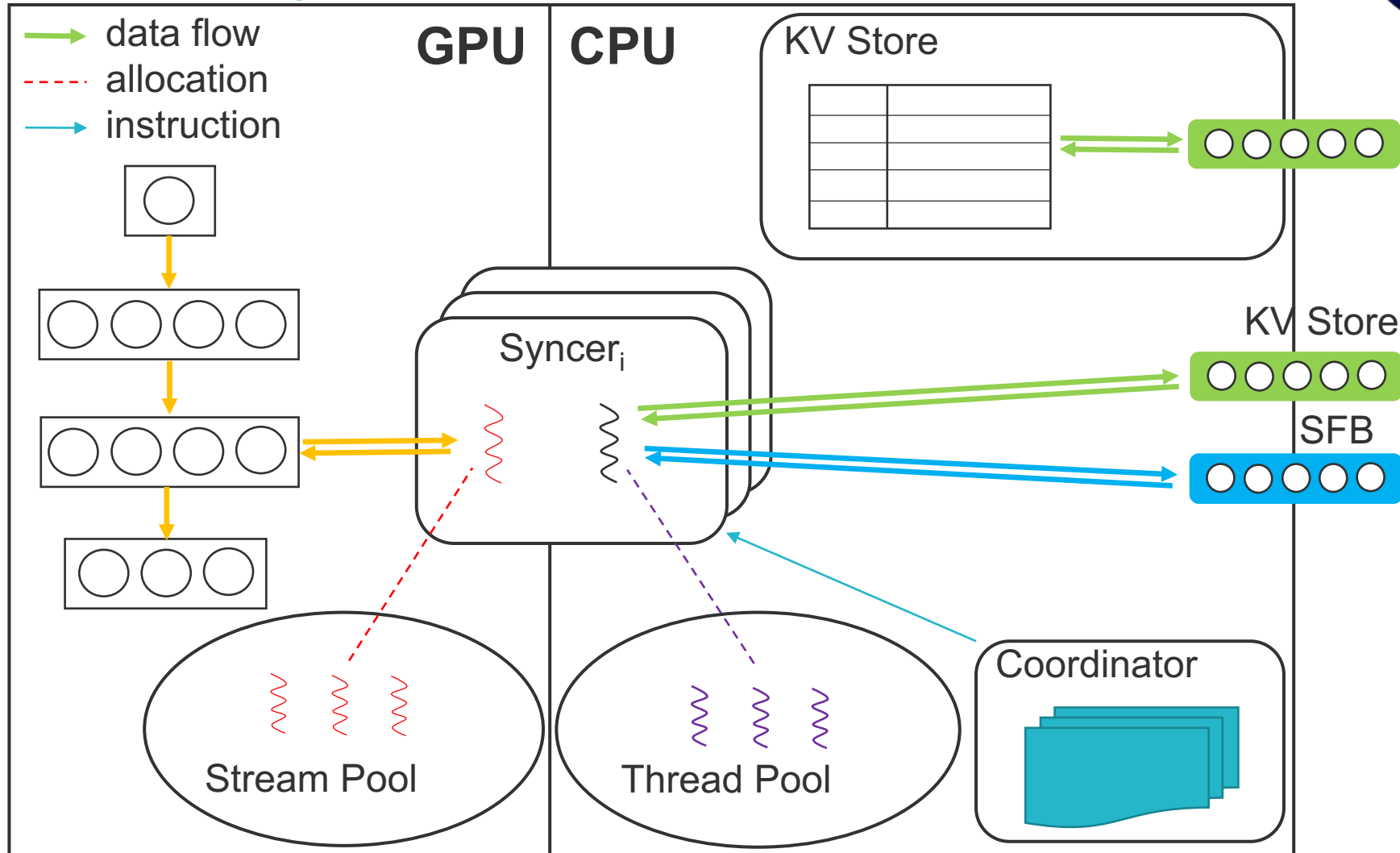


Poseidon's position

- Design principles
 - Efficient distributed platform for amplifying any DL toolkits
 - Preserve the programming interface for any high-level toolkits
 - i.e. distribute the DL program without changing any line of code
 - Easy deployment, easy adoption.



Poseidon System Architecture



Poseidon APIs

- KV Store, Syncer and Coordinator
- Standard APIs similar to parameter server
 - *Push/Pull* API for parameter synchronization
 - *BestScheme* method to return the best communication method

Method	Owner	Arguments	Description
BestScheme	Coordinator	A layer name or index	Get the best communication scheme of a layer
Query	Coordinator	A list of property names	Query information from coordinators' information book
Send	Syncer	None	Send out the parameter updates of the corresponding layer
Receive	Syncer	None	Receive parameter updates from either parameter server or peer workers
Move	Syncer	A GPU stream and an indicator of move direction	Move contents between GPU and CPU, do transformations and application of updates if needed
Send	KV store	updated parameters	Send out the updated parameters
Receive	KV store	parameter buffer of KV stores	Receive gradient updates from workers

Amplify DL toolboxes Using Poseidon

- For developers: plug Poseidon API into the backpropagation code, all you need to do is:
 - Back propagate through layer l
 - Sync parameters of layer l
 - Wait for finishing
- Amplifying Google TensorFlow
 - 250 line of code
- Amplifying Caffe
 - 150 line of code

Algorithm 2 Parallelize a DL library using Poseidon

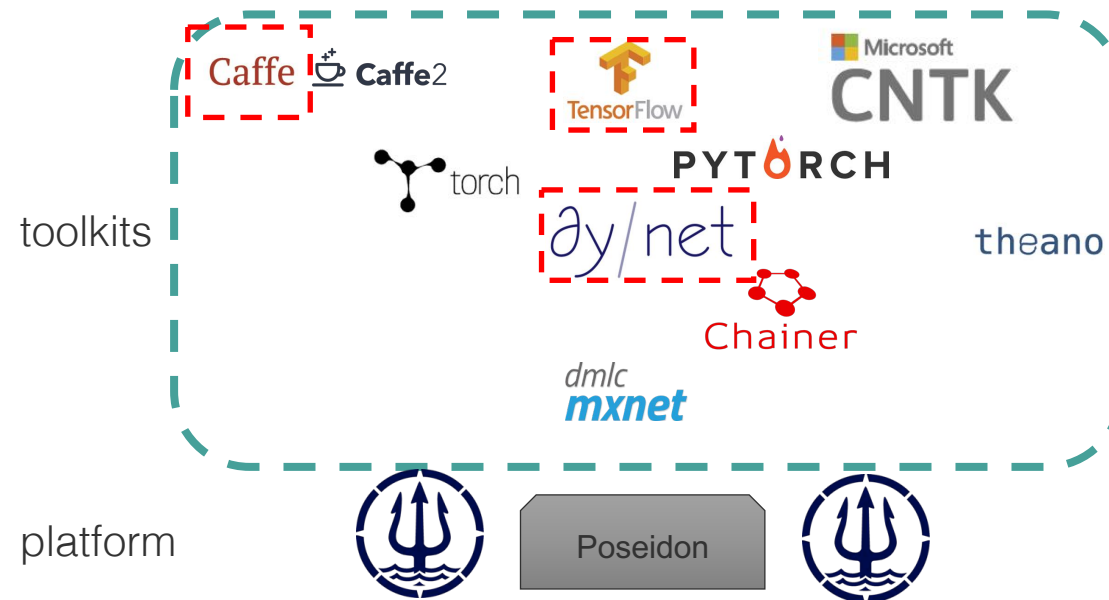
```

1: function TRAIN(net)
2:   for iter = 1 → T do
3:     sync_count = 0
4:     net.Forward()
5:     for l = L → 1 do
6:       net.BackwardThrough(l)
7:       thread_pool.Schedule(sync(l))
8:     end for
9:     wait_until(sync_count == net.num_layers)
10:  end for
11: end function
12: function SYNC(l)
13:  stream = stream_pool.Allocate()
14:  syncers[l].Move(stream, GPU2CPU)
15:  syncers[l].method = coordinator.BestScheme(l)
16:  syncers[l].Send()
17:  syncers[l].Receive()
18:  syncers[l].Move(stream, CPU2GPU)
19:  sync_count++
20: end function

```

Using Poseidon

- Poseidon: An efficient communication architecture
 - Preserve the programming interface for any high-level toolkits
 - i.e. distribute the DL program **without changing any line of application code**



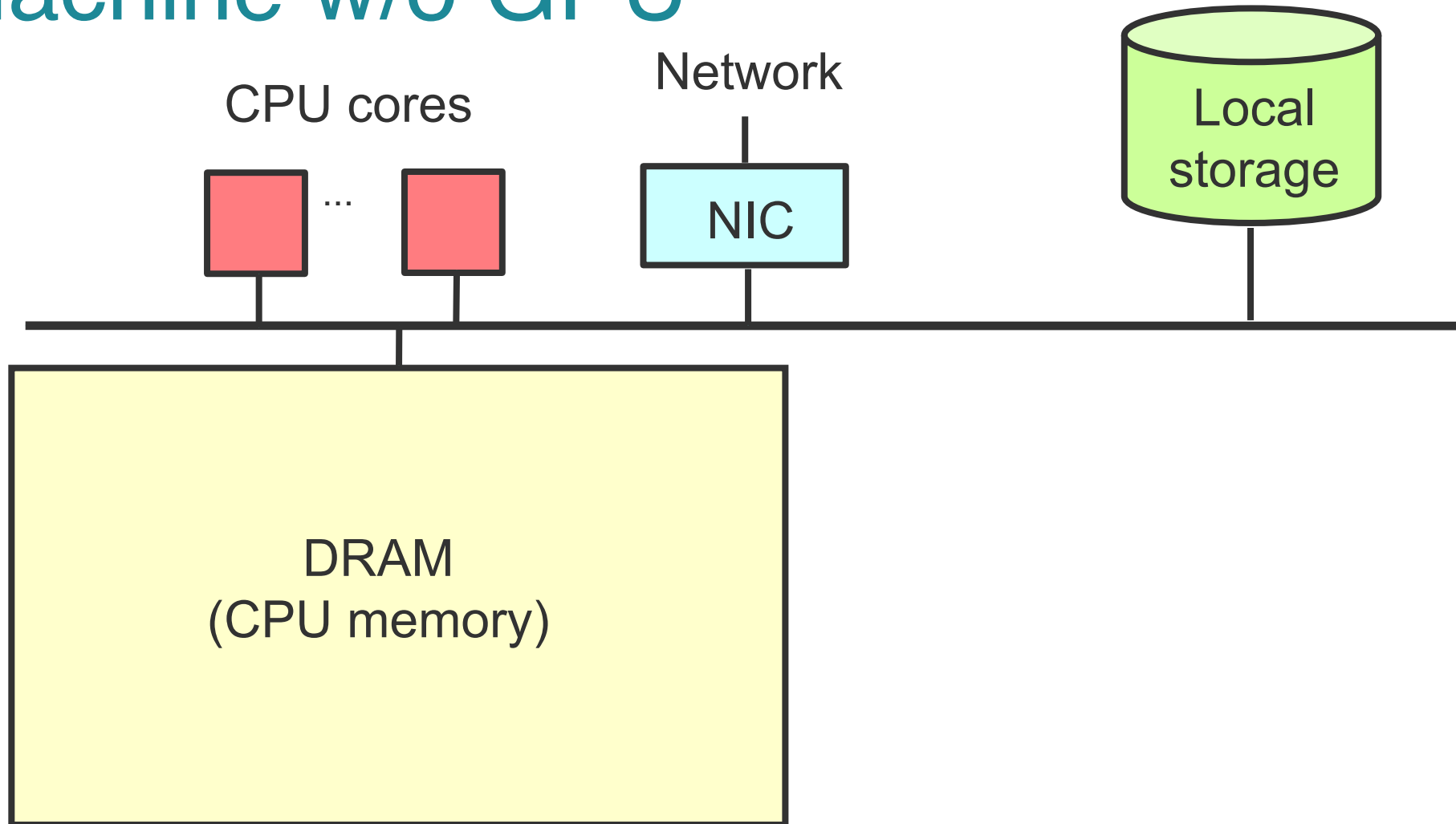
Outline

- Overview: Distributed Deep Learning on GPUs
- Challenges 1: Addressing the communication bottleneck
- **Challenges 2: Handling the limited GPU memory**

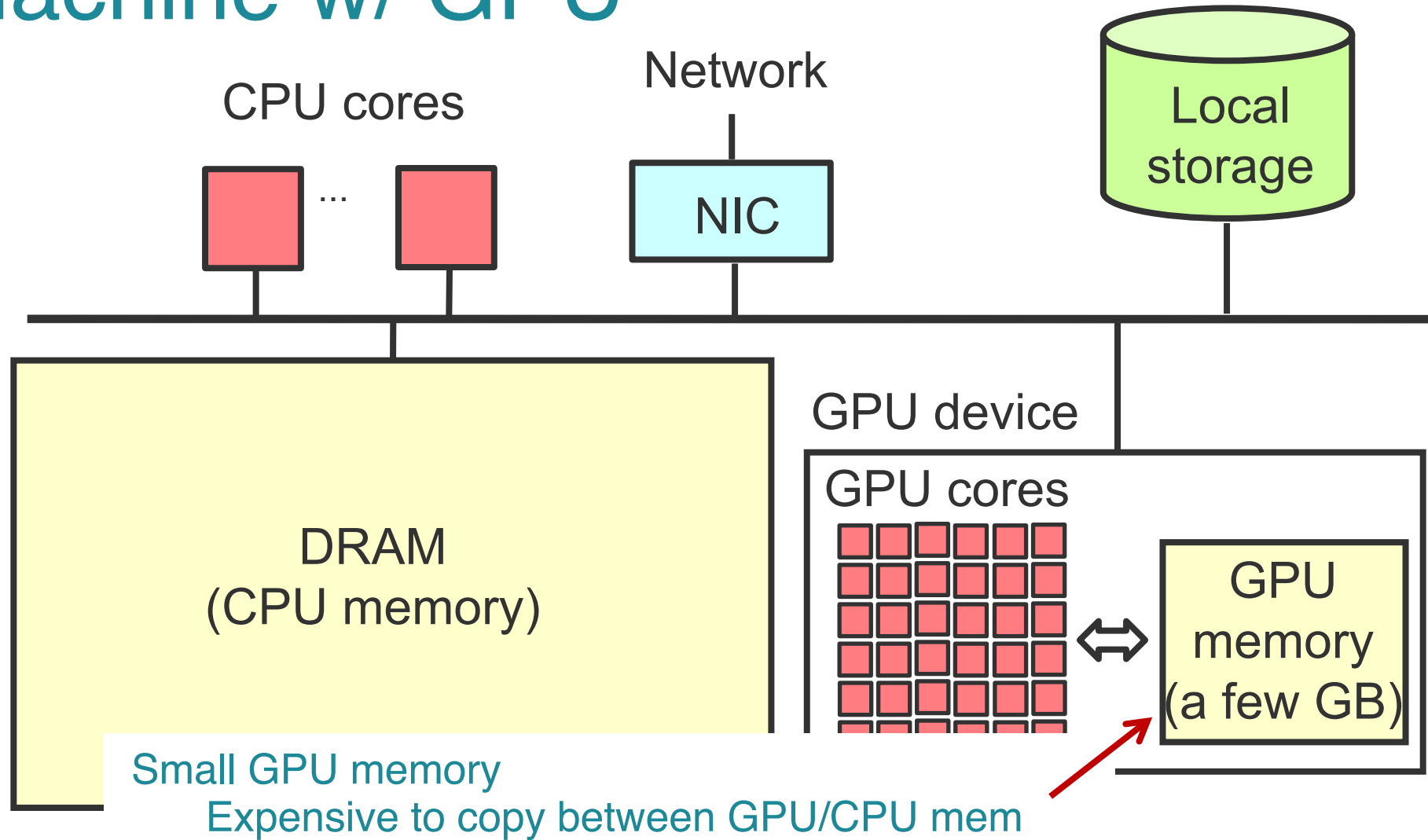
What is the Issue

- Memory
 - GPUs have dedicate memory
 - For a DL training program to be efficient, its data must be placed on GPU memory
 - GPU memory is limited, compared to CPU, e.g. maximally 12Gb
 - Memcpy between CPU and GPU is expensive – a memcpy takes the same time as launching a GPU computation kernel
- Problems to be answered
 - How to Avoid memcpy overhead between CPU and GPU?
 - How to proceed the training of a gigantic network with very limited available memory?

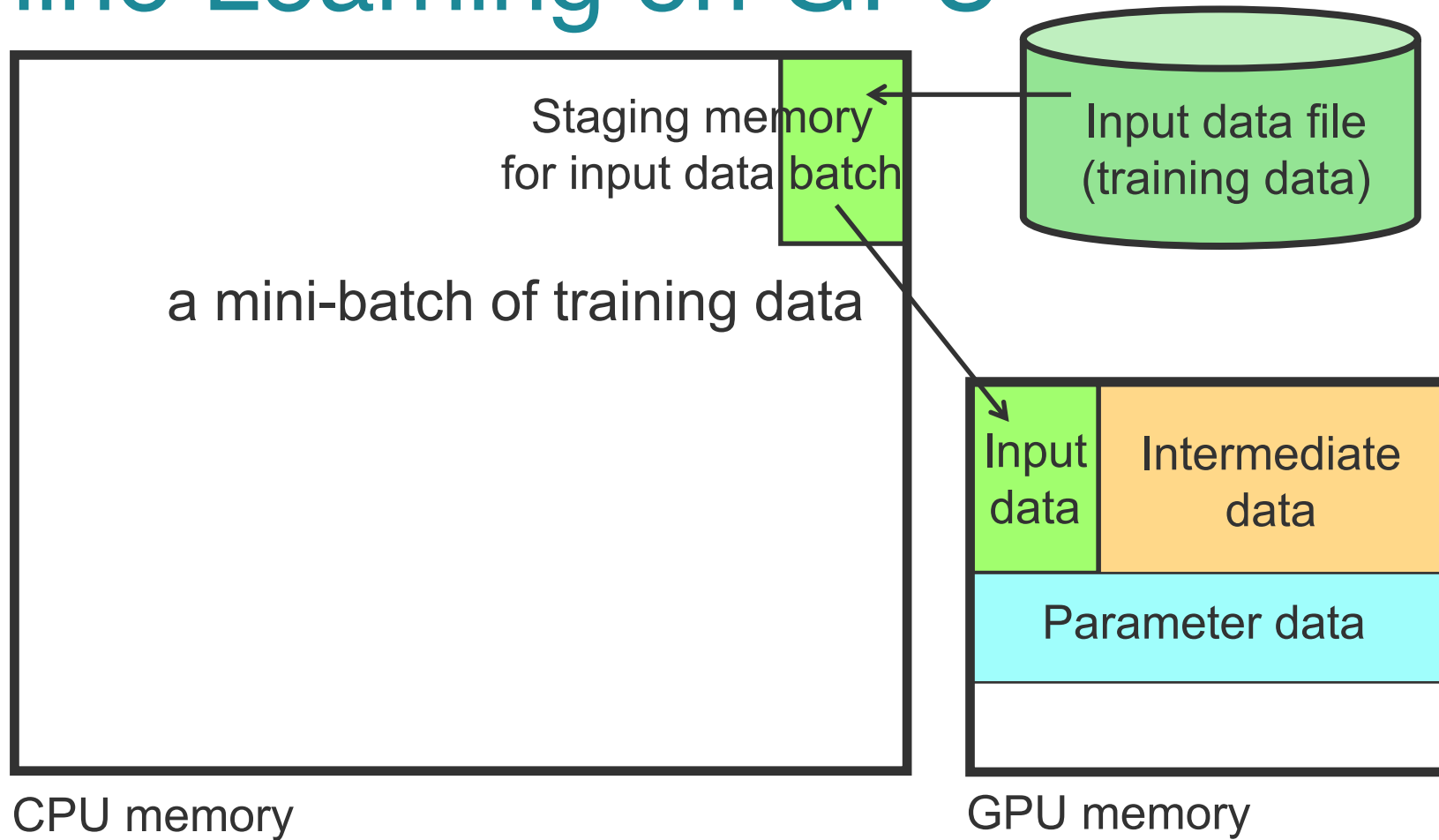
A Machine w/o GPU



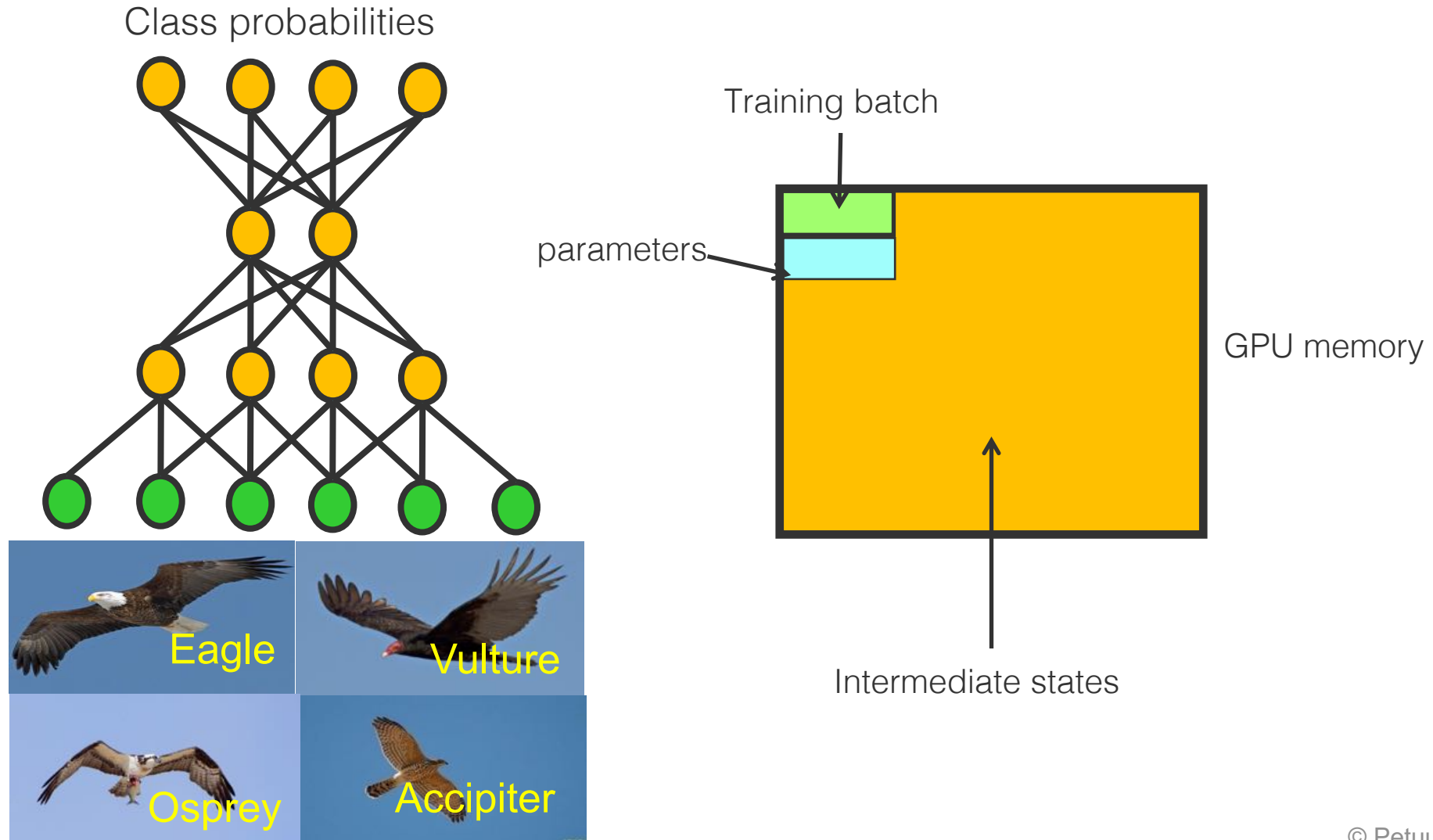
A Machine w/ GPU



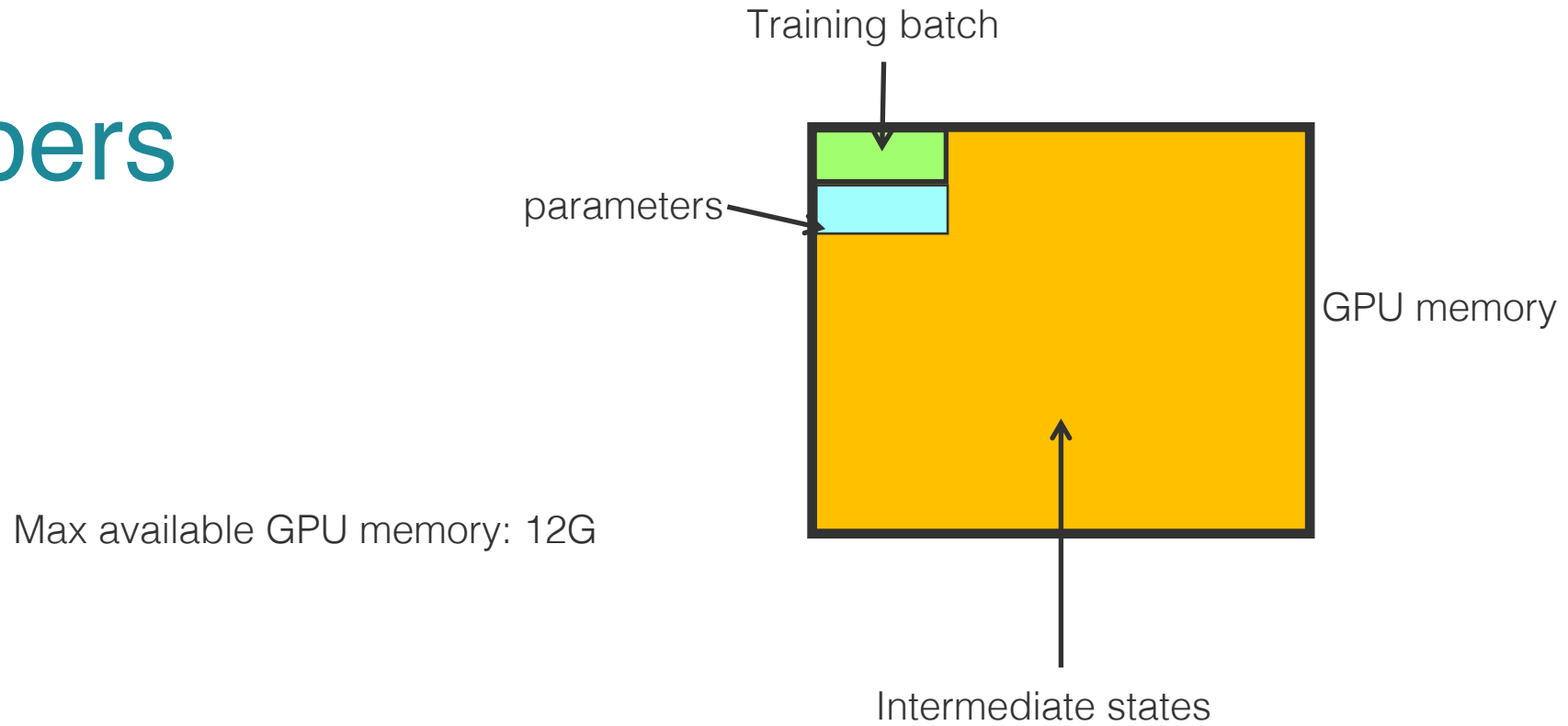
Machine Learning on GPU



Deep Learning on GPU



Numbers



Network	Batch size	Input size	Parameters + grads	Intermediate states
AlexNet	256	150MB	<500M	4.5G
GoogLeNet	64	19MB	<40M	10G
VGG19	16	10MB	<1.2G	10.8G

Why Memory is an Issue?

- Intermediate states occupy 90% of the GPU memory
- Intermediate states is proportional to input batch size

- However,
 - If you want high throughput, you must have large batch size (because of the SIMD nature of GPUs)
 - If you have large batch size, your GPU will be occupied by intermediate states, which thereby limits your model size/depth

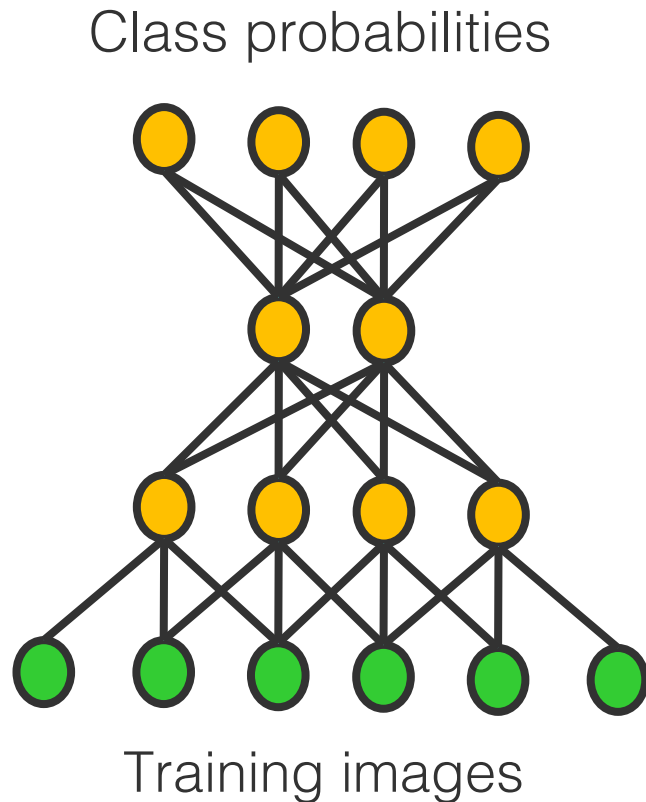
Saving Memory: A Simple Trick

- Basic idea
 - The fact: intermediate states are proportional to the batch size K
 - Idea: achieve large batch size by accumulating gradients generated by smaller batch sizes which are affordable in the GPU memory
- Solution:
 - Partition K into M parts, every part has K/M samples
 - For iter = 1: M
 - Train with mini-batchsize K/M
 - Accumulate the gradient on GPU w/o updating model parameters
 - Update the model parameter all together when all M parts finished
- Drawbacks
 - What if the GPU still cannot afford the intermediate states even if $K=1$?
 - Small batch size usually leads to insufficient use of GPUs' computational capability

Memory Management using CPU Memory

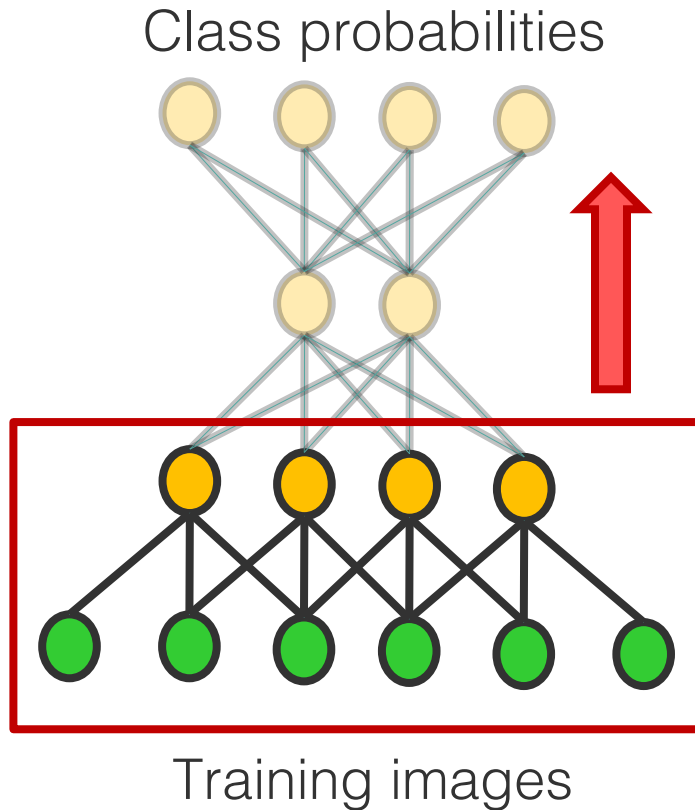
- Core ideas
 - **If the memory is limited, trade something for memory**
 - Trade extra computations for memory
 - **Trade other cost (e.g. memory exchange) for more available memory**
 - **If the memory is limited, then get more**
 - model parallel
 - **CPU memory**

Memory Management using CPU Memory



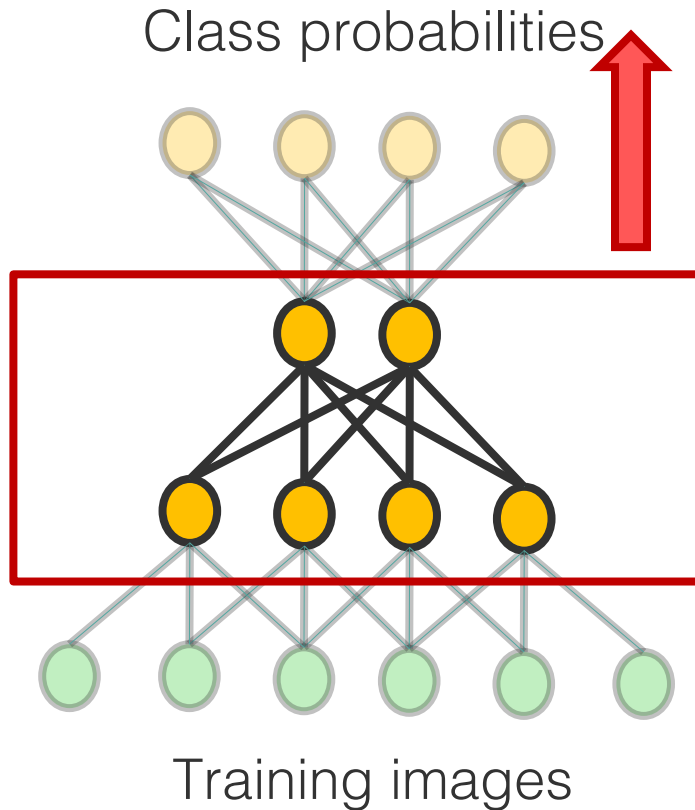
- For each iteration (mini-batch)
 - A forward pass
 - Then a backward pass
- Each time only data of two layers are used

Memory Management using CPU Memory



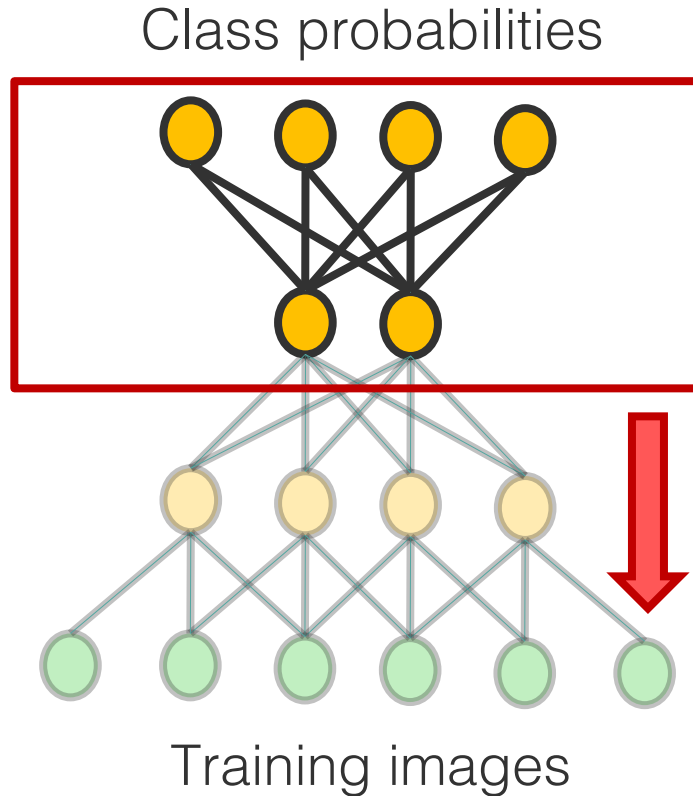
- For each iteration (mini-batch)
 - A forward pass
 - Then a backward pass
- Each time only data of two layers are used

Memory Management using CPU Memory



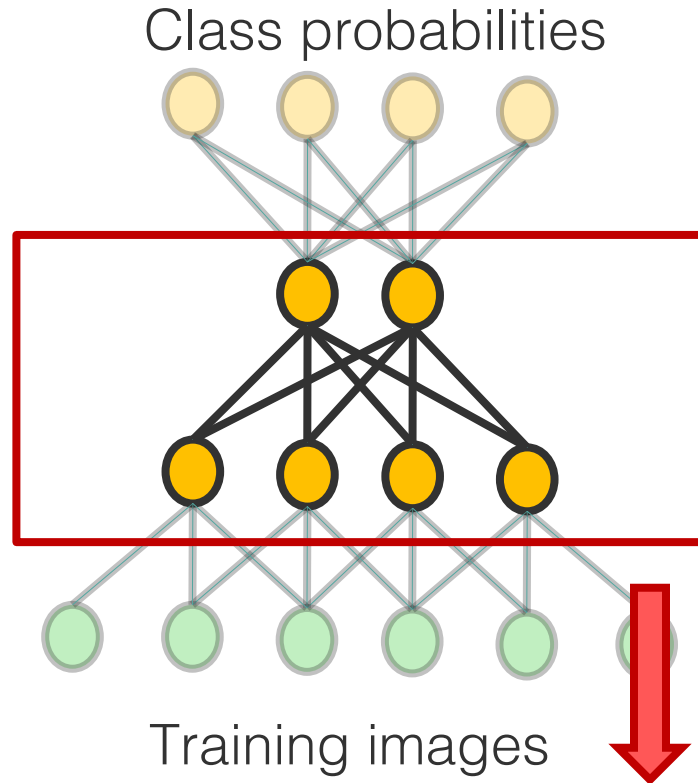
- For each iteration (mini-batch)
 - A forward pass
 - Then a backward pass
- Each time only data of two layers are used

Memory Management using CPU Memory



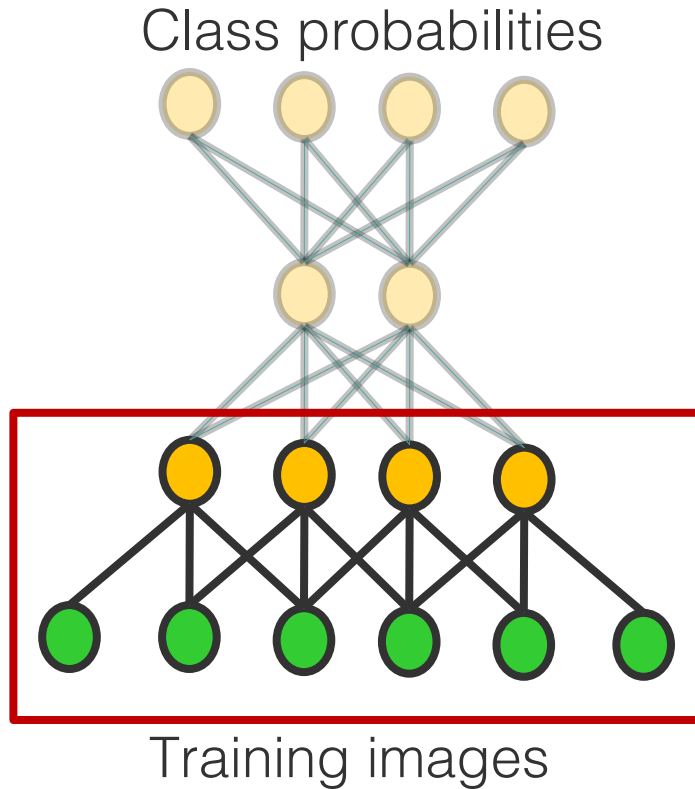
- For each iteration (mini-batch)
 - A forward pass
 - Then a backward pass
- Each time only data of two layers are used

Memory Management using CPU Memory



- For each iteration (mini-batch)
 - A forward pass
 - Then a backward pass
- Each time only data of two layers are used

Memory Management using CPU Memory

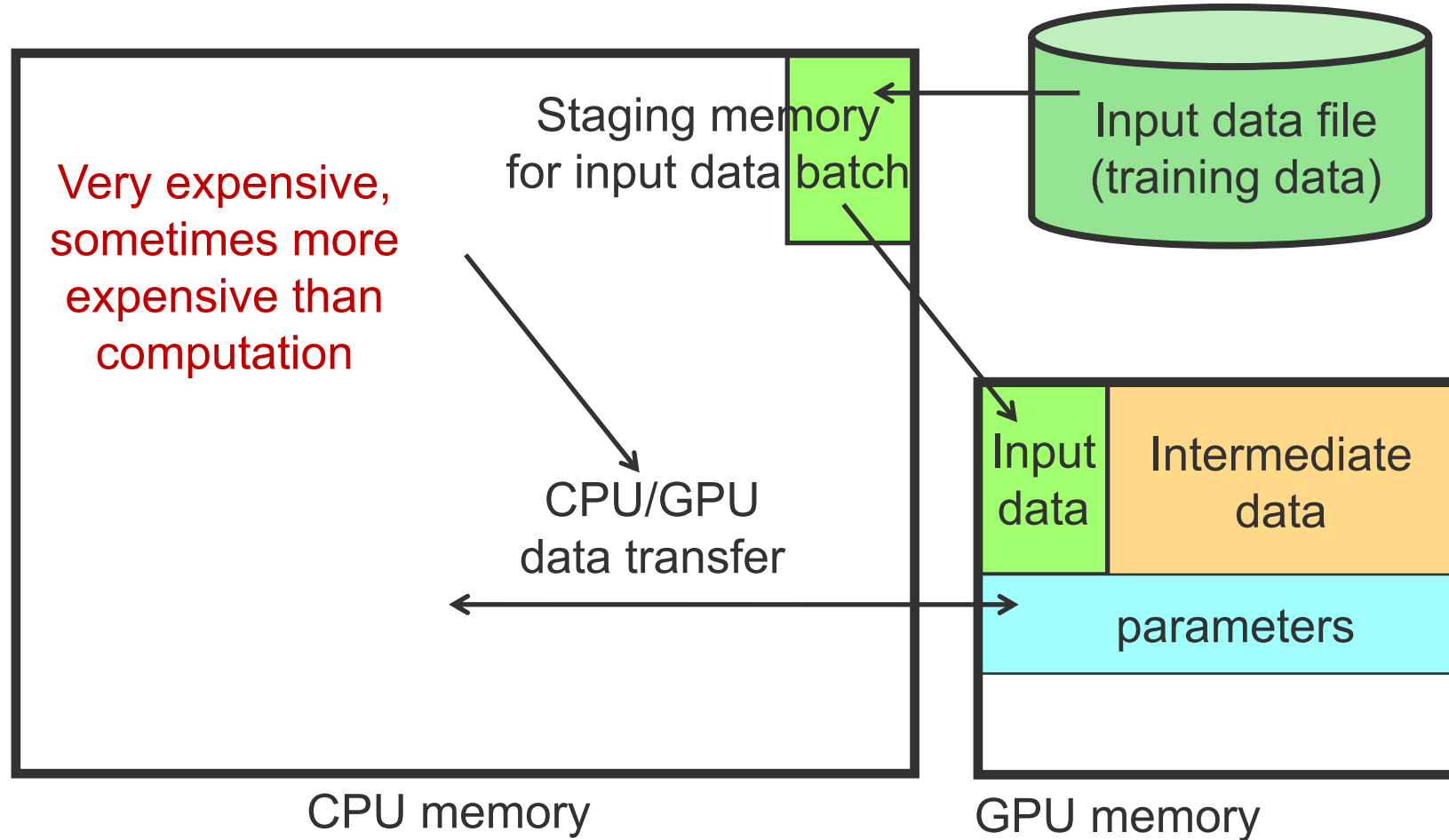


- For each iteration (mini-batch)
 - A forward pass
 - Then a backward pass
- Each time only data of two layers are used

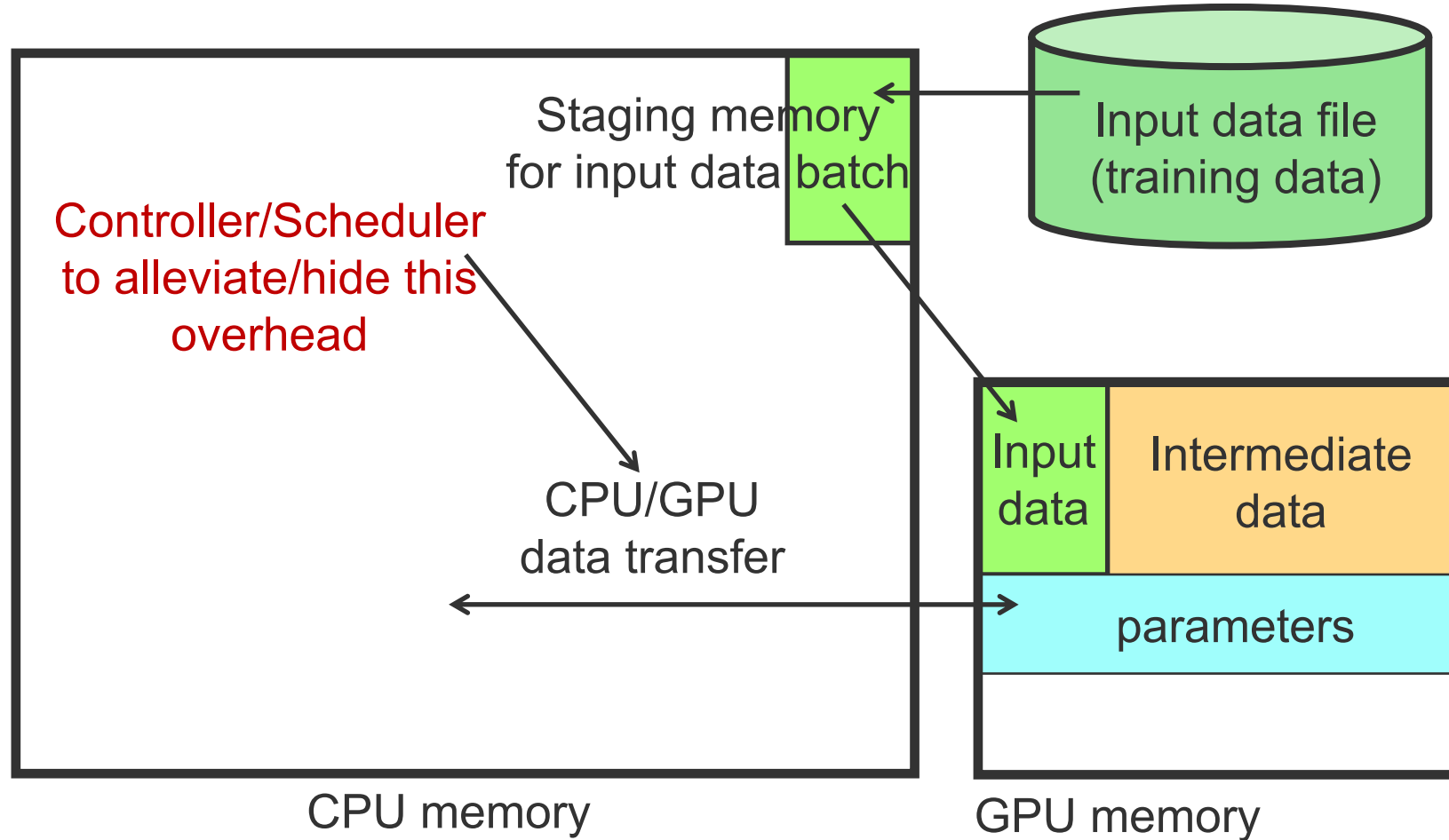
The idea

- Use GPU mem as a cache to keep actively used data
- Store the remaining in CPU memory

Memory Management using CPU Memory



Memory Management using CPU Memory

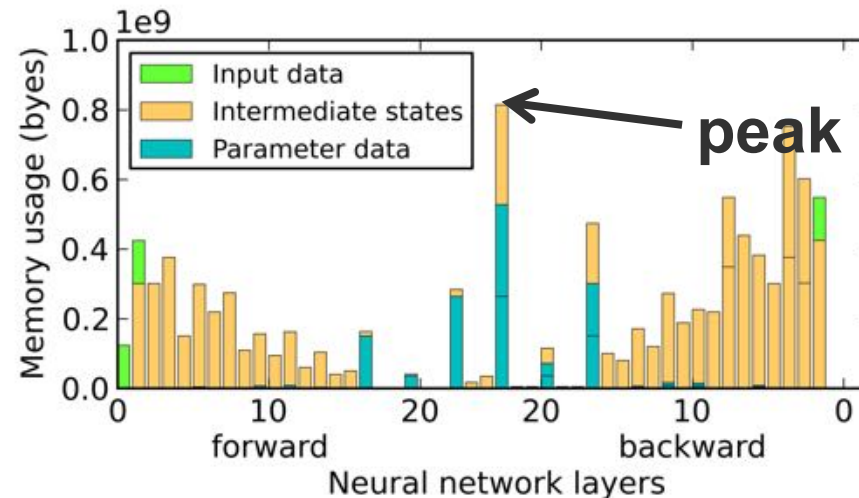


Memory Management using CPU Memory

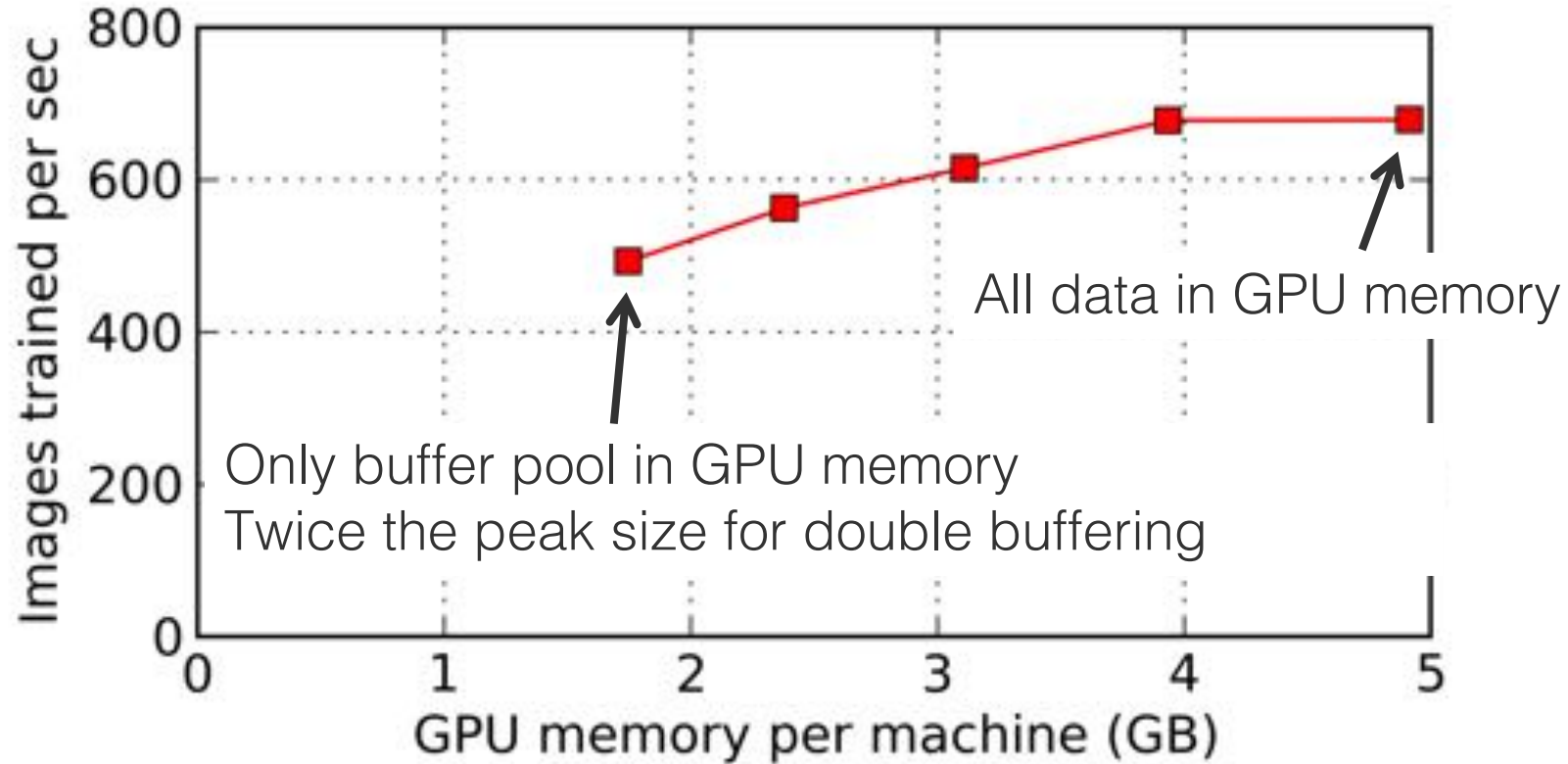
- Controller
 - The fact: the memory access order is deterministic and can be exactly known by a single forward and backward pass
 - Idea:
 - Obtain the memory access order by a virtual iteration
 - Pre-fetch memory blocks from CPU to GPU
 - Overlap memory swap overhead with computation

Memory Management using CPU Memory

- What's the best we can do with this strategy
 - We only need 3 memory blocks (peak size) on GPU for:
 - Input, Parameters, Output
 - The whole training can process with ONLY these three blocks by
 - Scheduling memcopy between CPU and GPU to be overlapped with computation
 - Move in and out for each layer's computation as training proceeds

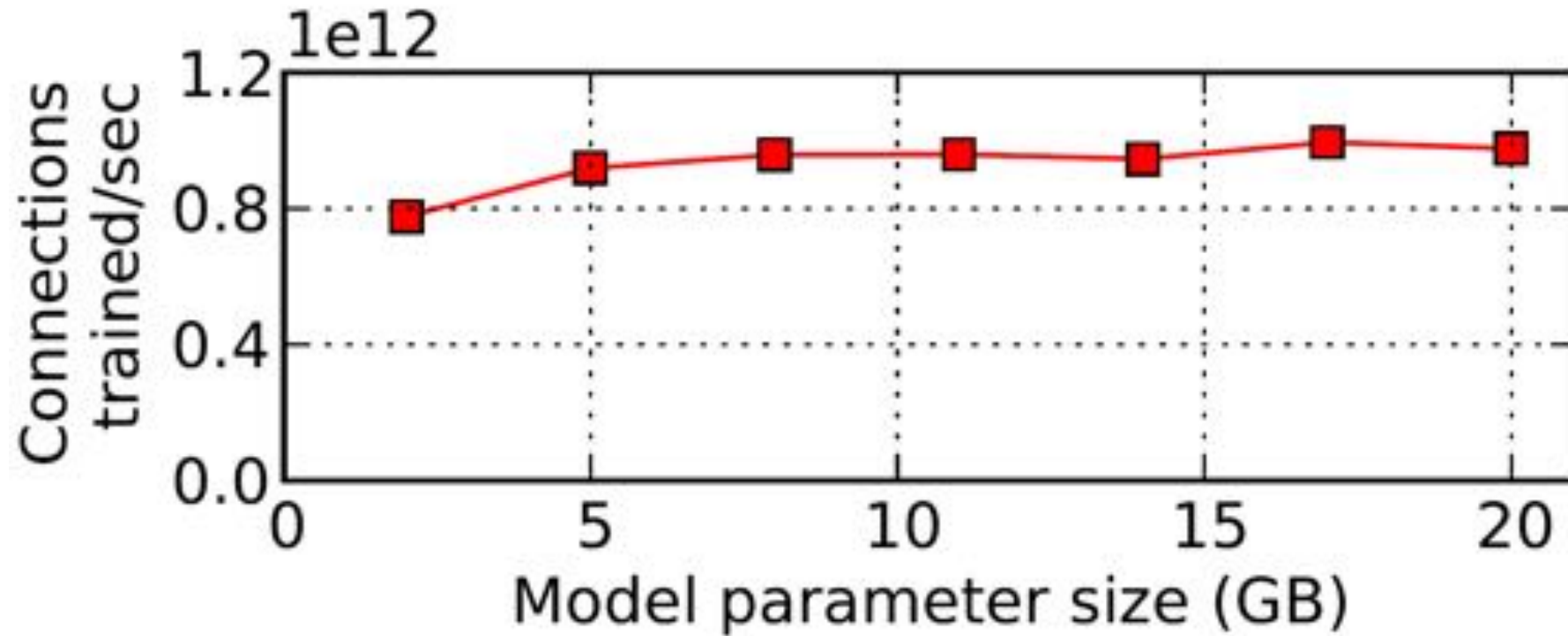


Throughput vs. memory budget



- Only 27% reduction in throughput with 35% memory
- Can do 3x bigger problems with little overhead

Larger models

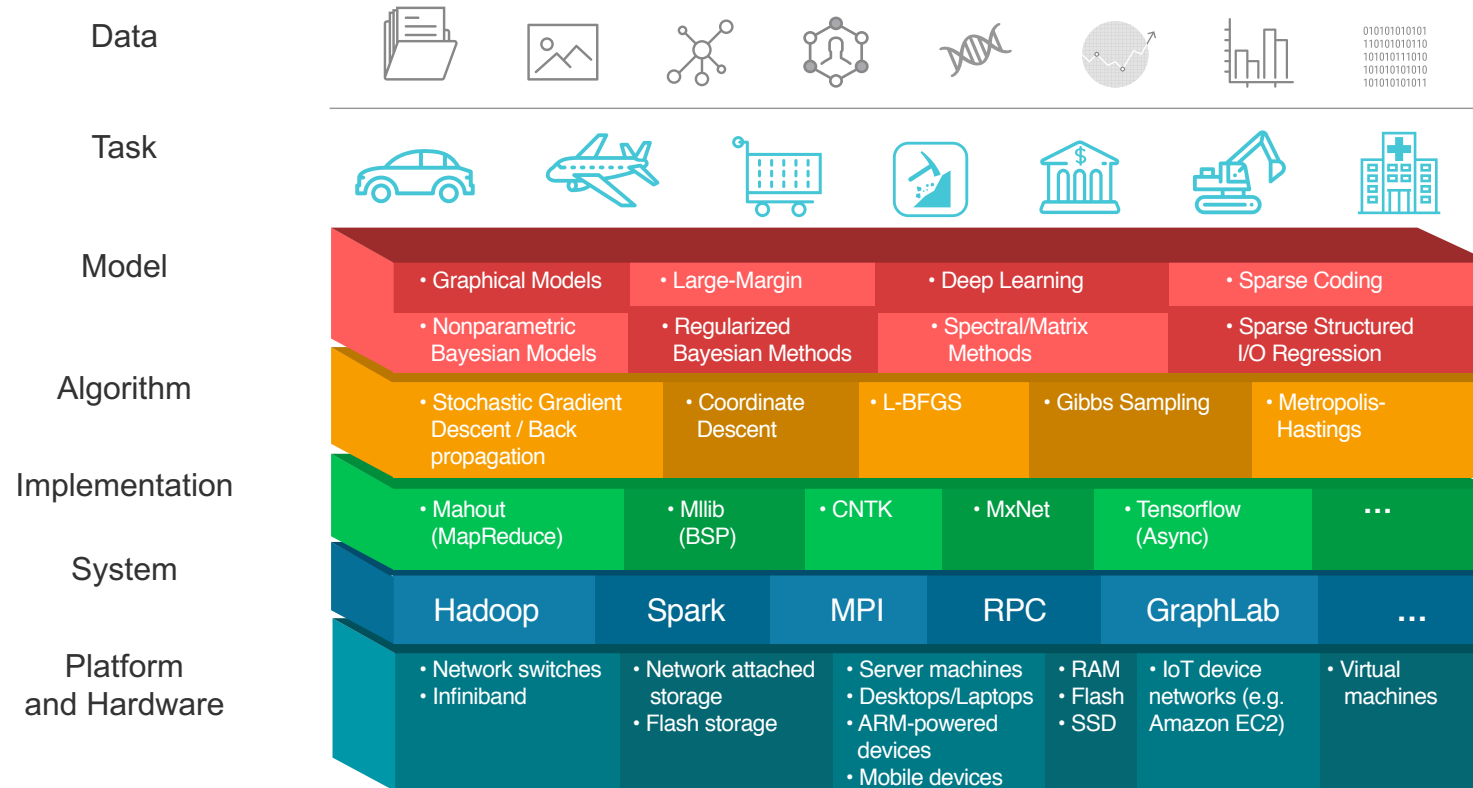


- Models up to 20 GB

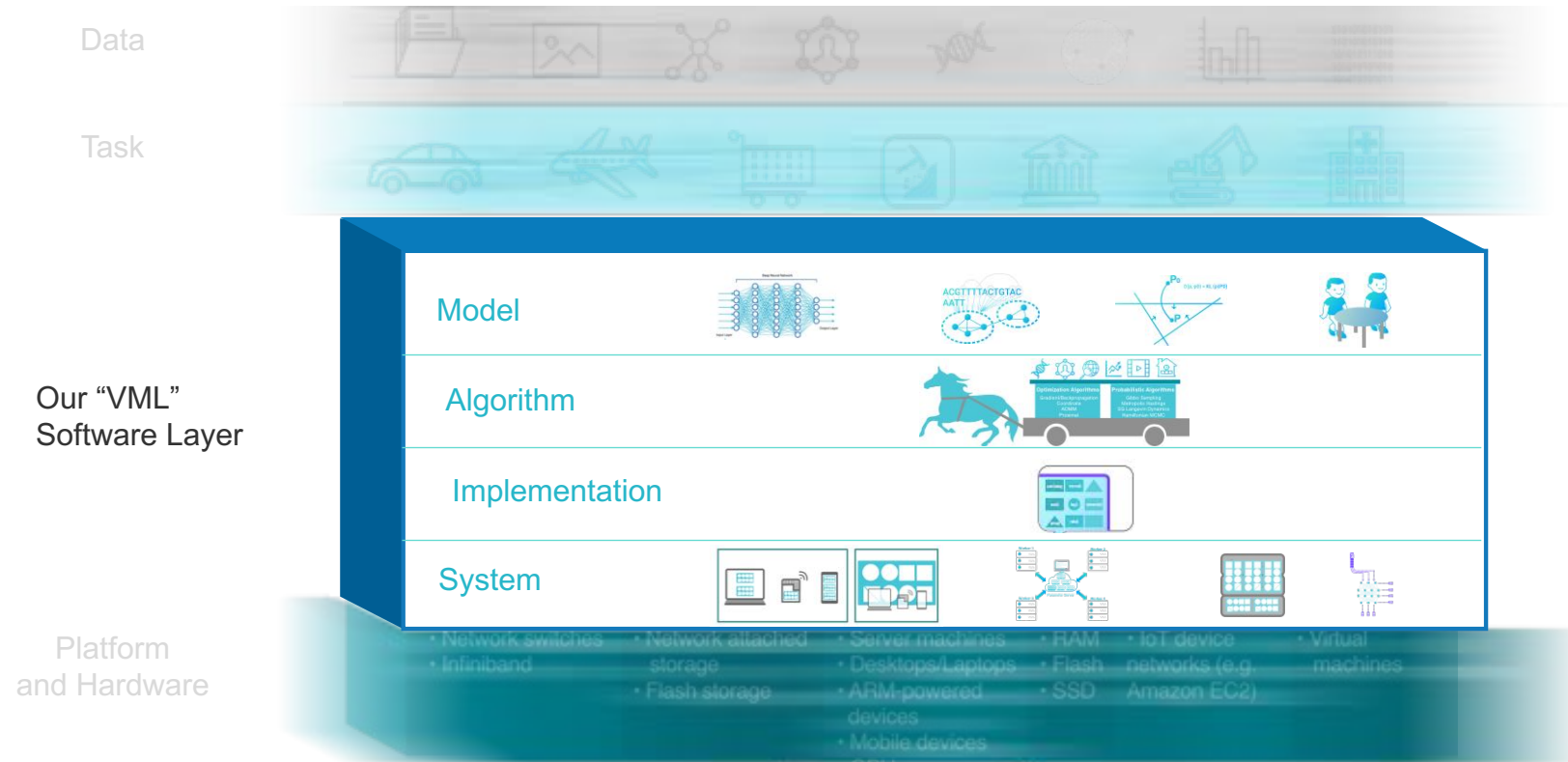
Summary

- Deep learning as dataflow graphs
- A lot of auto-differentiation libraries have been developed to train NNs
 - Different adoption, advantages, disadvantages
 - DyNet is a new framework for next-wave dynamic NNs
- Difficulties arise when scaling up DL using distributed GPUs
 - Communication bottleneck
 - Memory limit
- Poseidon as a platform to support and amplify different kinds of DL toolboxes

Elements of Modern AI



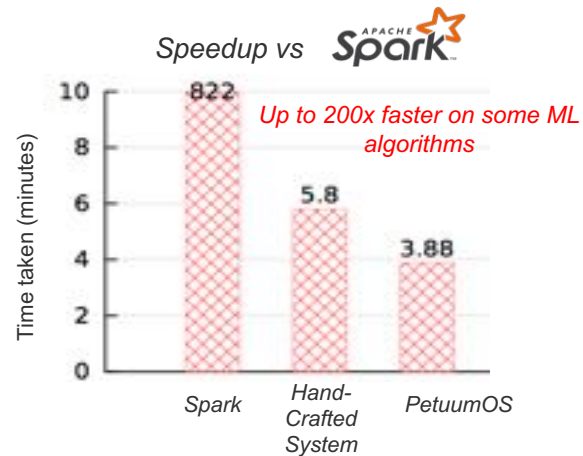
Sys-Alg Co-design Inside!



Better Performance

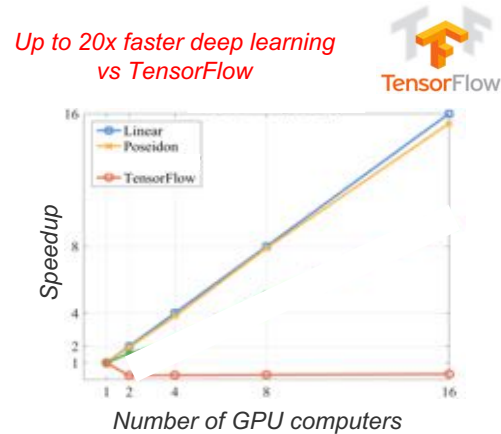
- **Fast and Real-Time**

- Orders of magnitude faster than Spark and TensorFlow
- As fast as hand-crafted systems



- **Any Scale**

- Perfect straight-line speedup with more computing devices
- Spark, TensorFlow can slow down with more devices



- **Low Resource**

- **Turning a regular cluster into a super computer:**

- Achieve AI results with much more data, but using fewer computing devices
- Google brain uses ~1000 machines whereas Petuum uses ~10 for the same job



A Petuum Vision

