Deep Learning and Optimization, With Applications to Speech Recognition

Tara N. Sainath
June 15, 2015

Acknowledgements

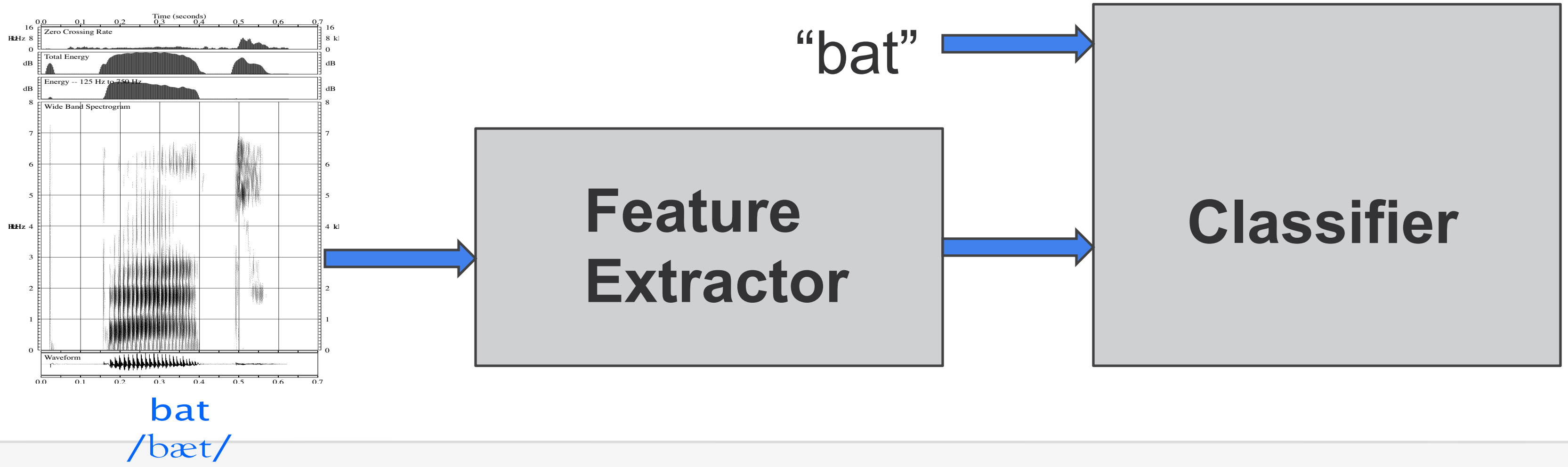
- Google
  - Michiel Bacchiani
  - Hasim Sak
  - Andrew Senior
- IBM
  - Brian Kingsbury
  - George Saon
- Microsoft
  - Li Deng
  - Frank Seide
  - Dong Yu

Outline

- Why Neural Networks
- Training Neural Networks
- Making Neural Networks Work for Speech Recognition
- Optimization challenges: Training time
- Optimization challenges: New architectures

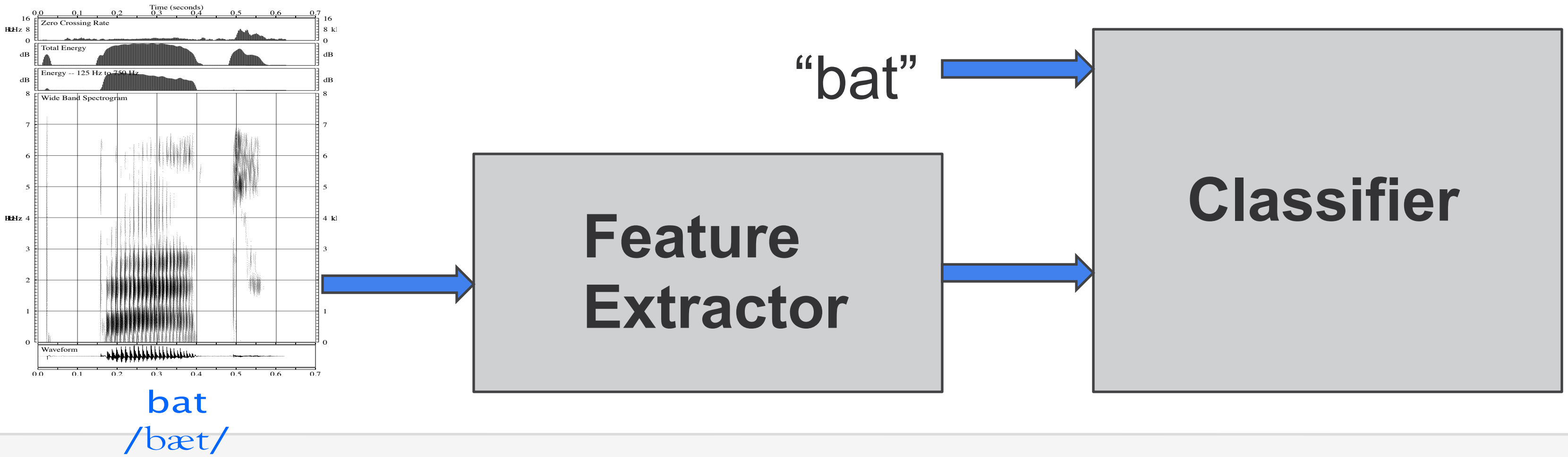
Pattern Recognition System

- The goal of any pattern recognition system is to
  - determine an appropriate feature representation
  - classify these features effectively



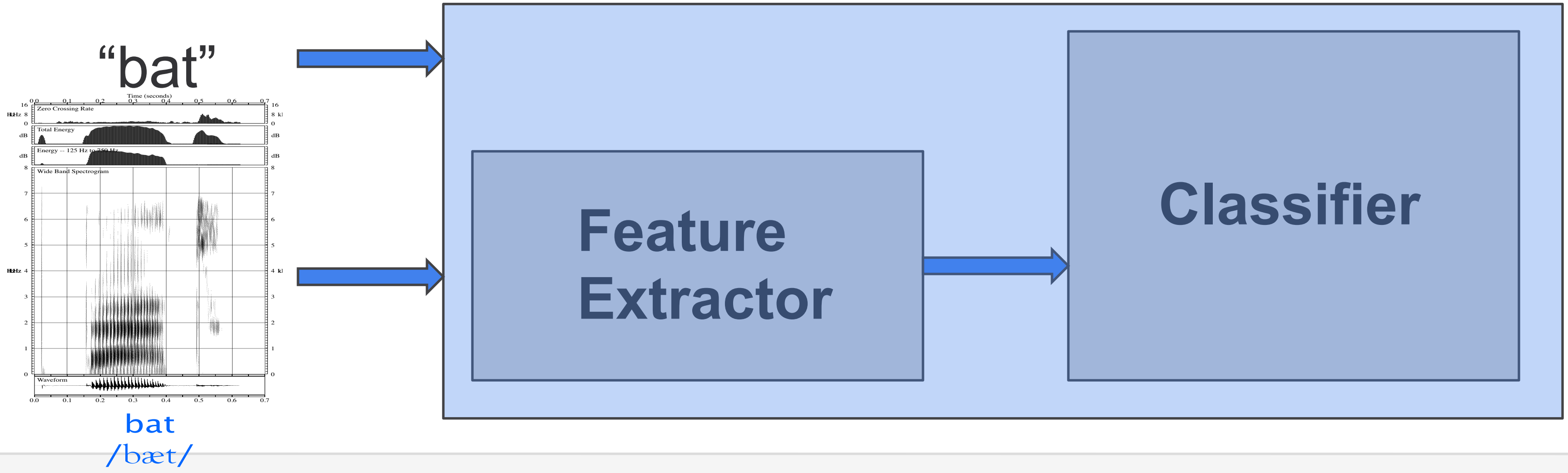
Example from Speech Recognition

- For example, in speech recognition, we first create features by hand
- Then we build a discriminative classifier (Gaussian Mixture Model) to distinguish between classes
- Features are not directly designed for classification objective



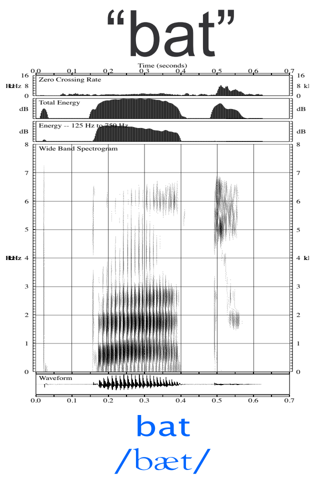
End-to-End Recognition System

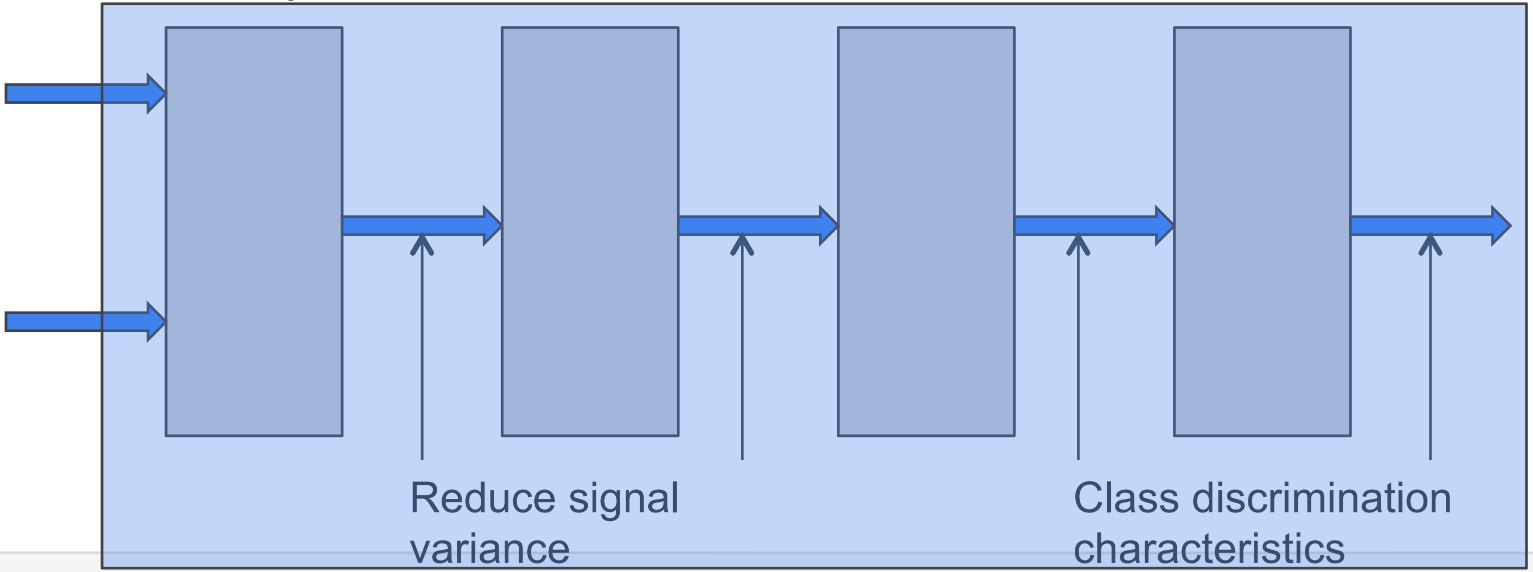
- Black box which takes simple features + labels does the feature extraction **jointly** with the classification
- Features are trained to the classification objective
- Big non-linear system trained to map from simple features to labels



Intuition Behind Deep Neural Networks

- Each block produces a higher level feature representation and better classifier than its input
- By combining simple building blocks, we can design more and more complex, non-linear systems



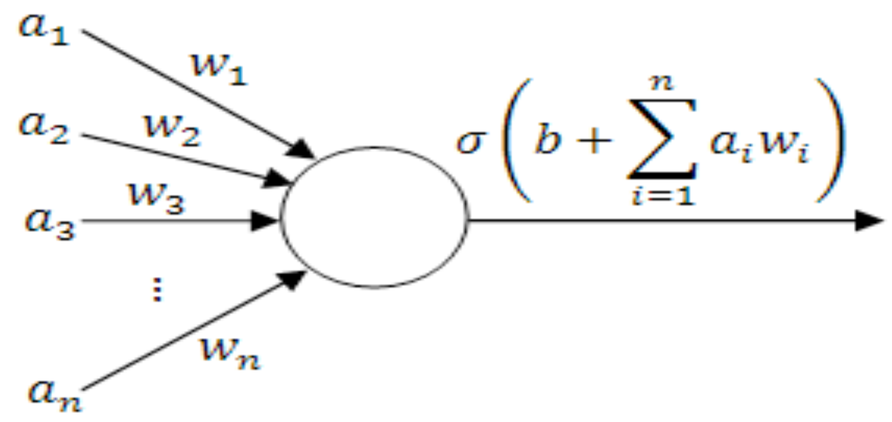


Representation at Each Stage

- Within each building block, we want the following properties:
  - Create a higher level representation of input
  - Better separate input into classes
  - Can be combined with previous layers
  - Can be trained jointly with other layers
- Using a mathematical model of a biological neuron is an appropriate choice

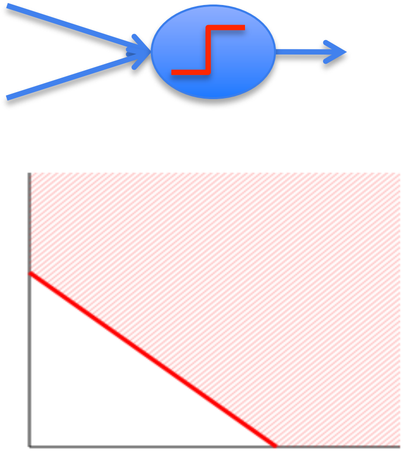
Neurons

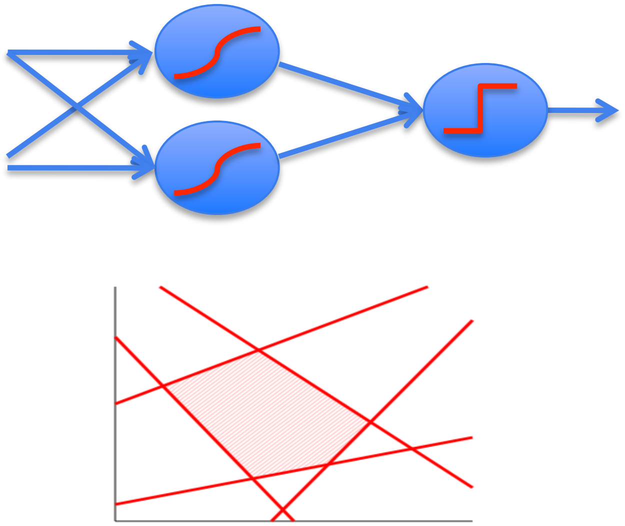
- A neuron takes a weighted sum of inputs ***a*** and feeds the result through an activation function ***σ***
- Output of the activation function produces decision boundary which can be used in classification



Combining Neurons

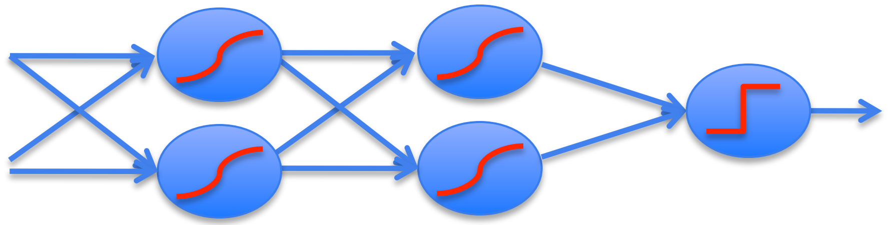
- Each neuron splits the feature space with a hyperplane
- 1-layer of trainable weights cannot handle XOR
- 2-layers of trainable weights gives a convex polygon region

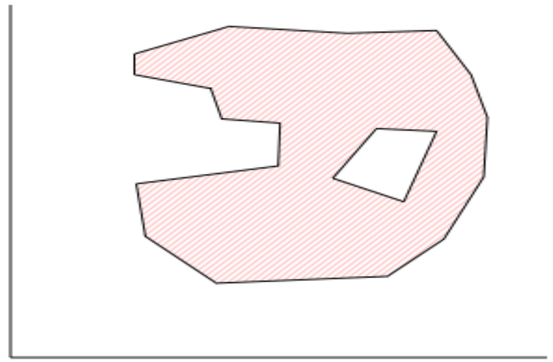




Combining Neurons

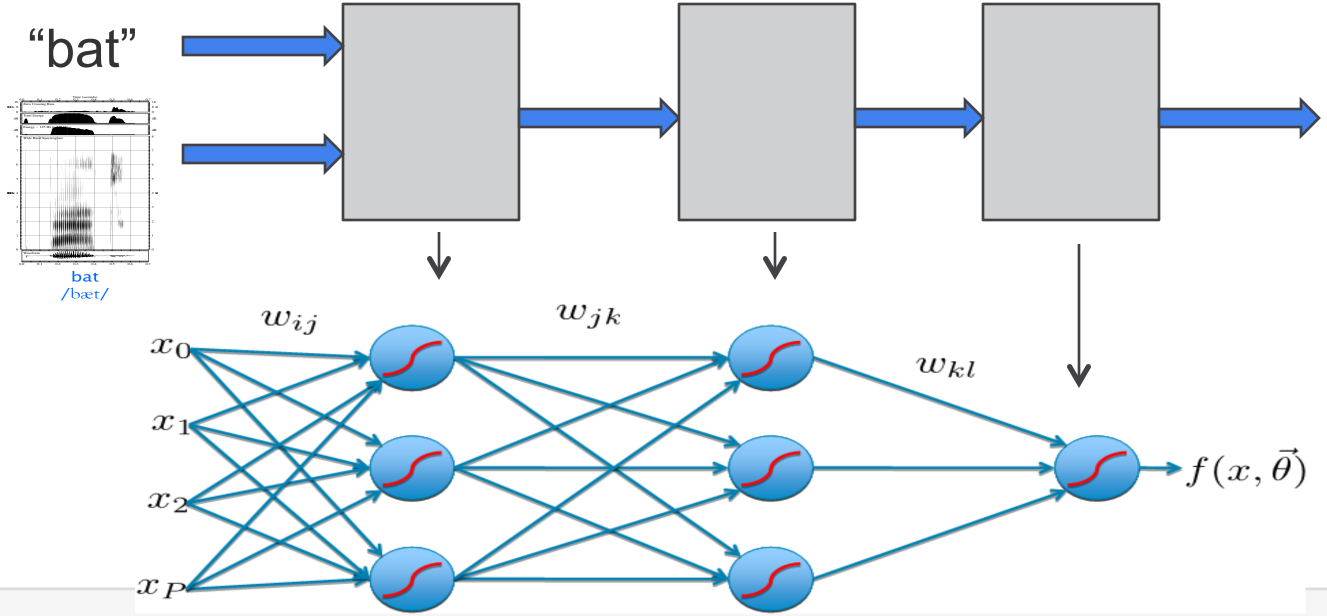
- 3 layers of trainable weights gives a composition of polygons: convex regions
- More layers can handle more complicated spaces – but require more parameters





Multi-Layer Neural Network

- Each simple building block is a connection of neurons which produces a higher-order, more complex representation of the input
- Neurons in one layer are connected to neurons in the next layer

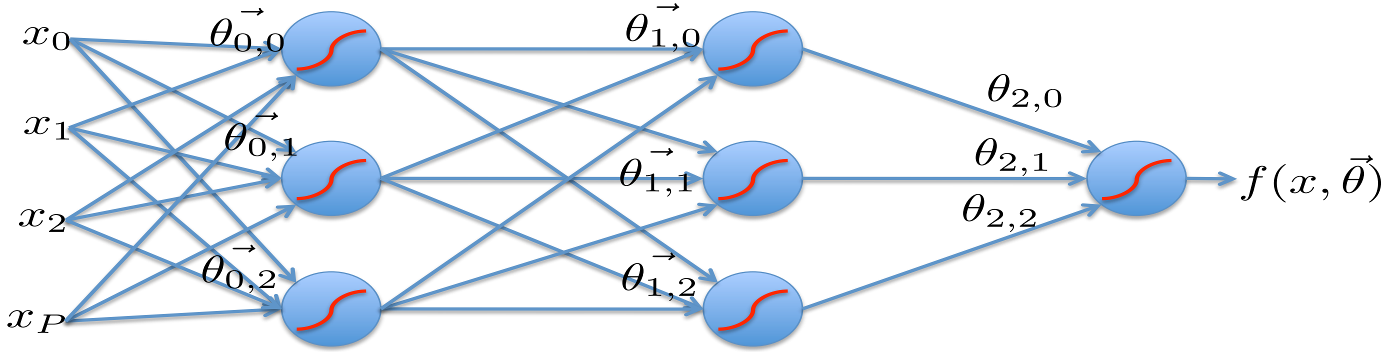


Outline

- Why Neural Networks
- Training Neural Networks
- Making Neural Networks Work for Speech Recognition
- Optimization challenges: Training time
- Optimization challenges: New architectures

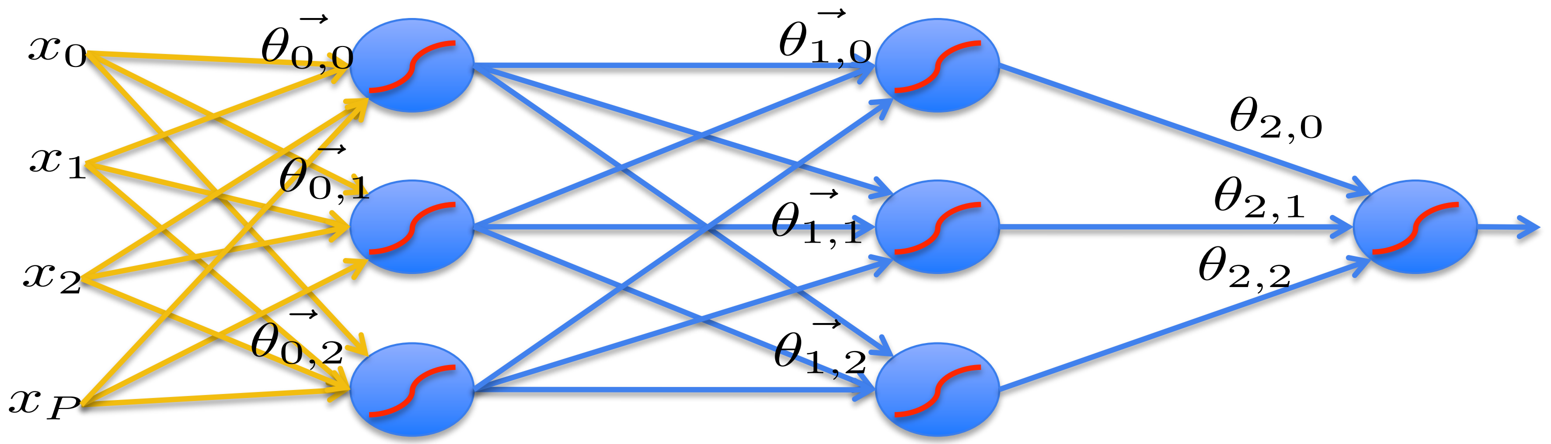
Training Neural Networks

- Most common approach to train neural networks is via stochastic gradient descent
    - Propagate input forward
    - Compute gradient of objective function
    - Propagate error gradient backwards



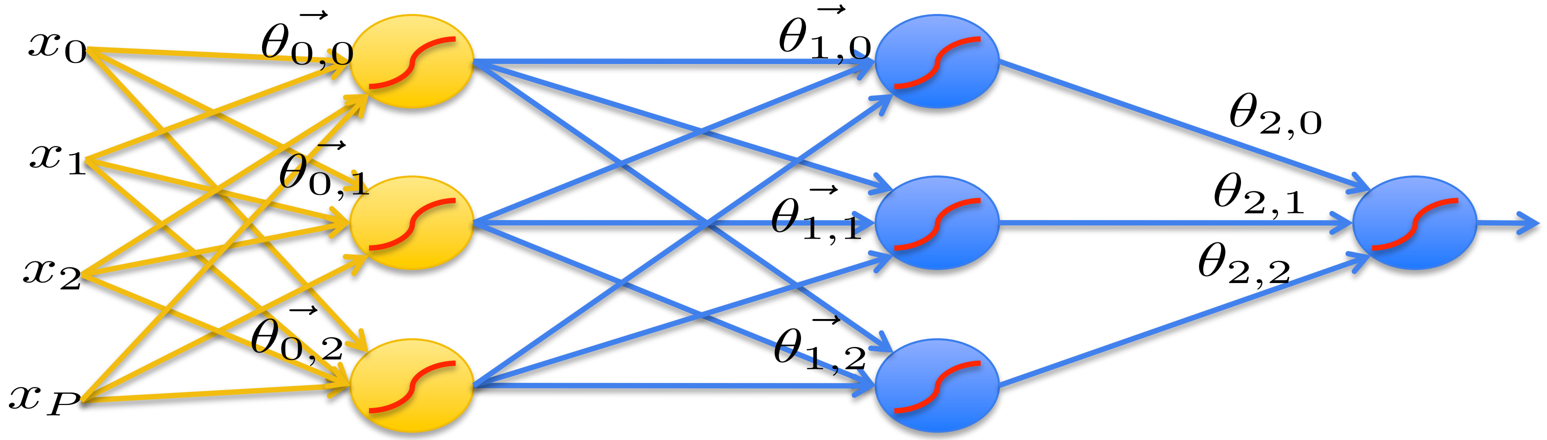
Feed-Forward Networks

- Predictions are fed forward through the network to classify



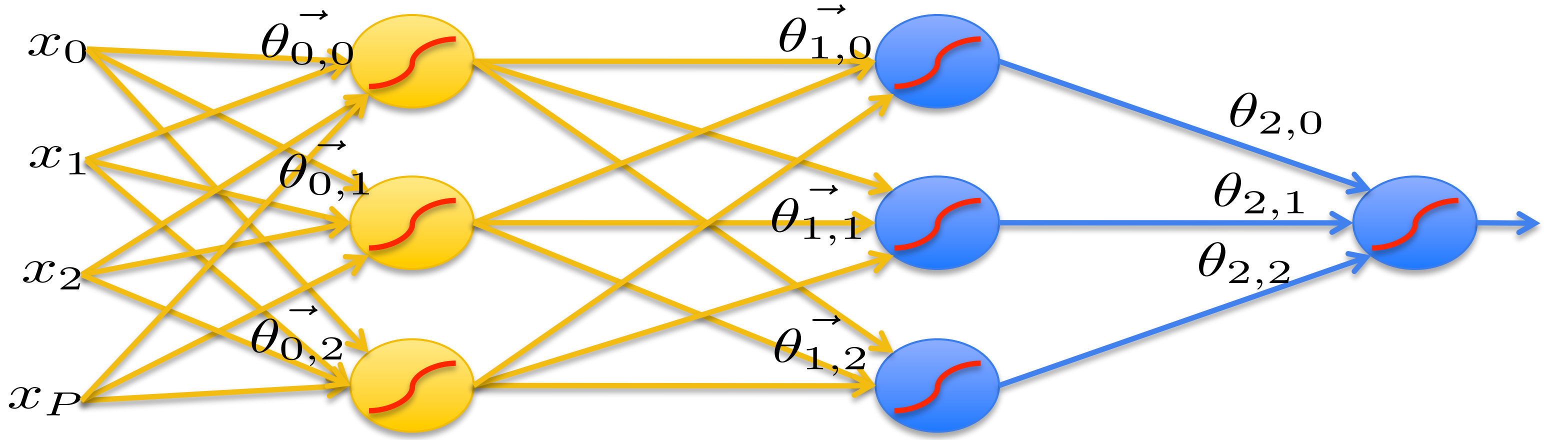
Feed-Forward Networks

- Predictions are fed forward through the network to classify



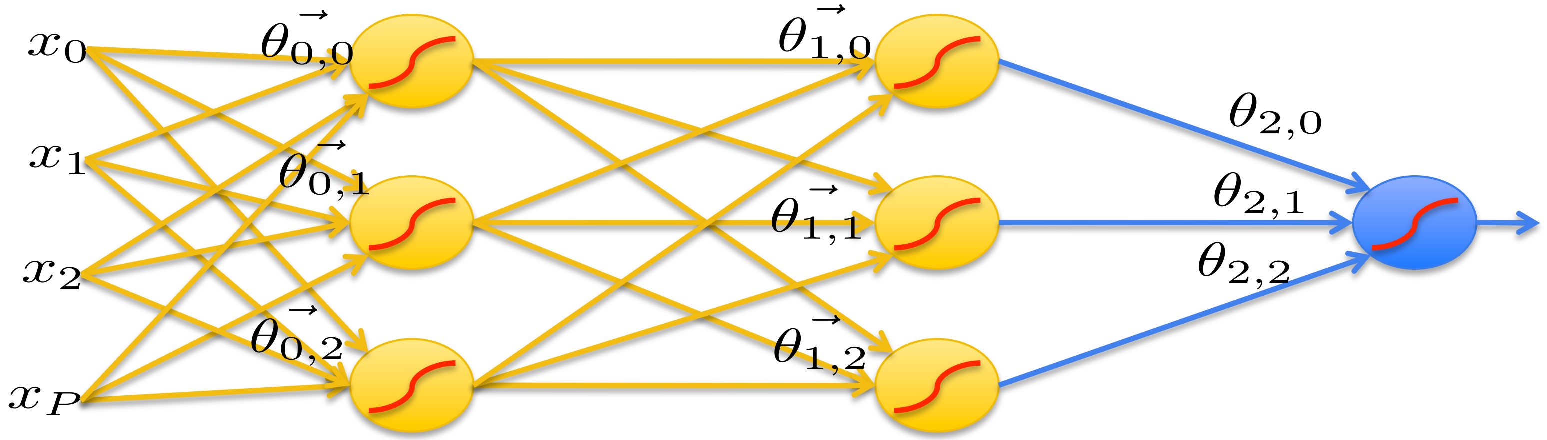
Feed-Forward Networks

- Predictions are fed forward through the network to classify



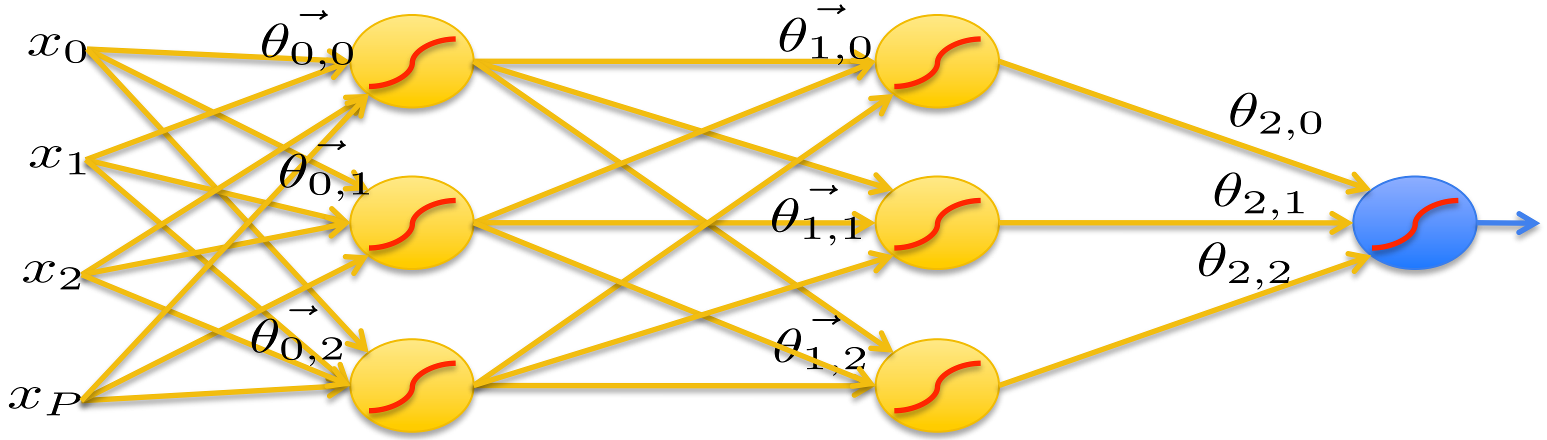
Feed-Forward Networks

- Predictions are fed forward through the network to classify



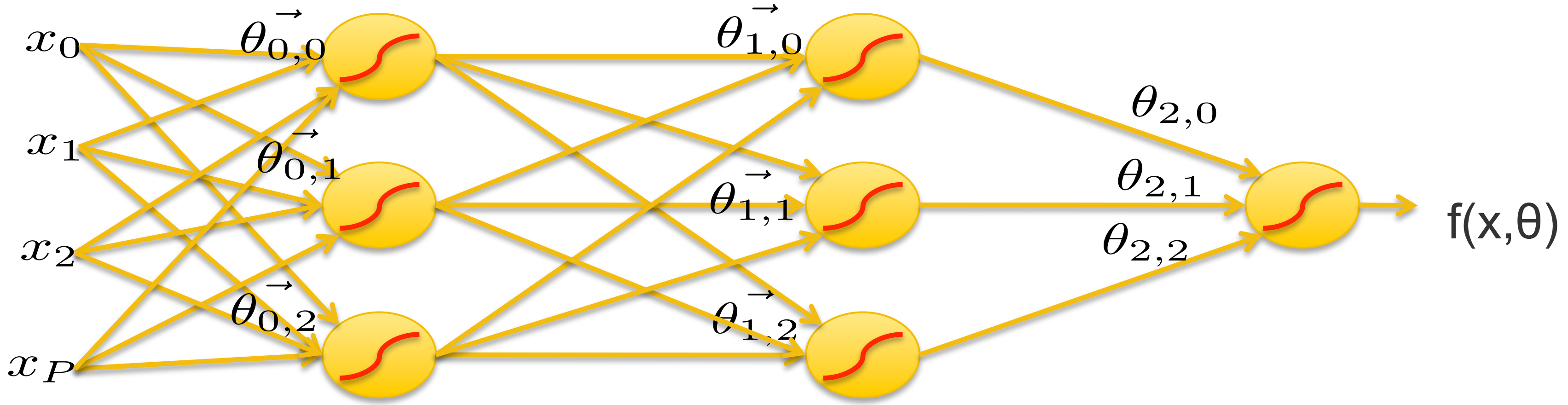
Feed-Forward Networks

- Predictions are fed forward through the network to classify

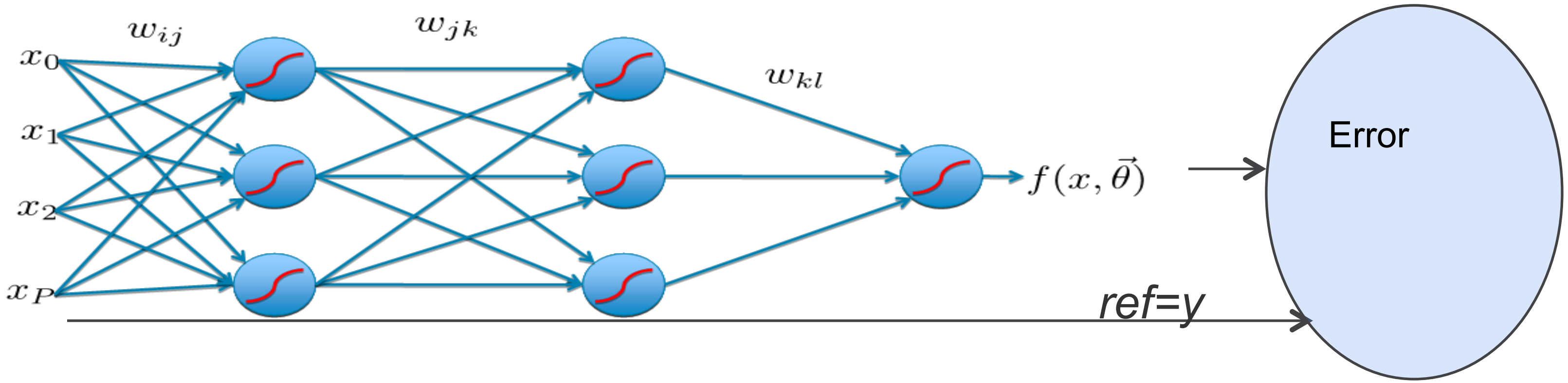


Feed-Forward Networks

- Predictions are fed forward through the network to classify



Define Objective Function



- For regression problems, sum-of-squared error is used

$$y, f(x,\theta) \in R^N \qquad L = \frac{1}{2}\sum_{n=1}^{N}(y_n - f_n(x,\theta))^2$$

- For classification problems, cross-entropy is used

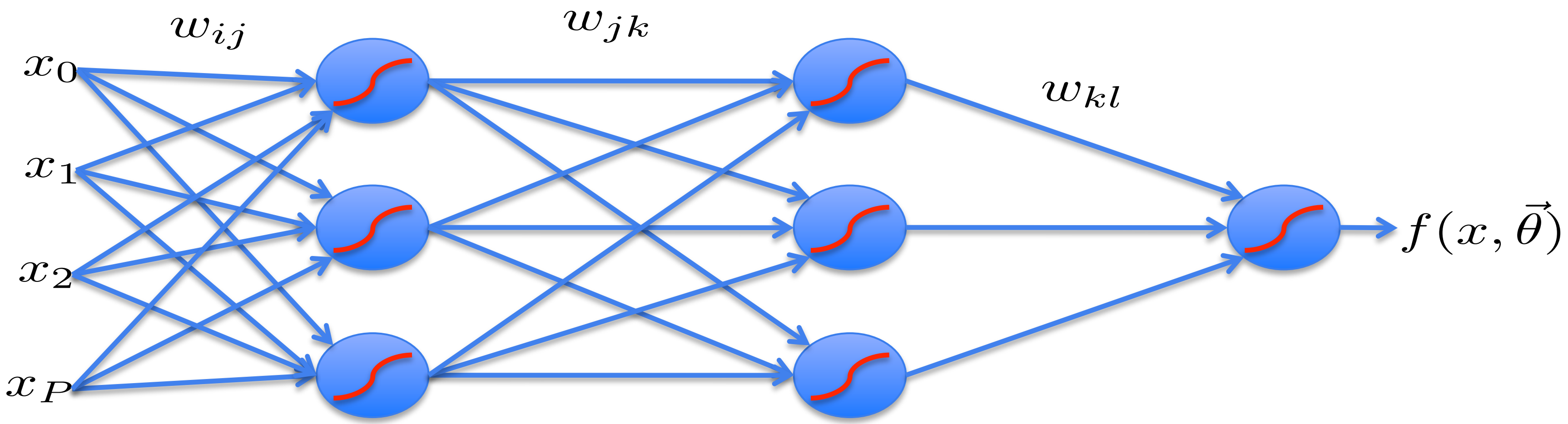
$$y, f(x,\theta) \in [0,1]^N$$
$$\sum_{n=1}^{N} y_n = 1$$
$$\sum_{n=1}^{N} f_n(x,\theta) = 1$$

$$L = -\sum_{n=1}^{N} y_n \log f_n(x,\theta)$$

Error Backpropagation

- Introduce variables over the neural network

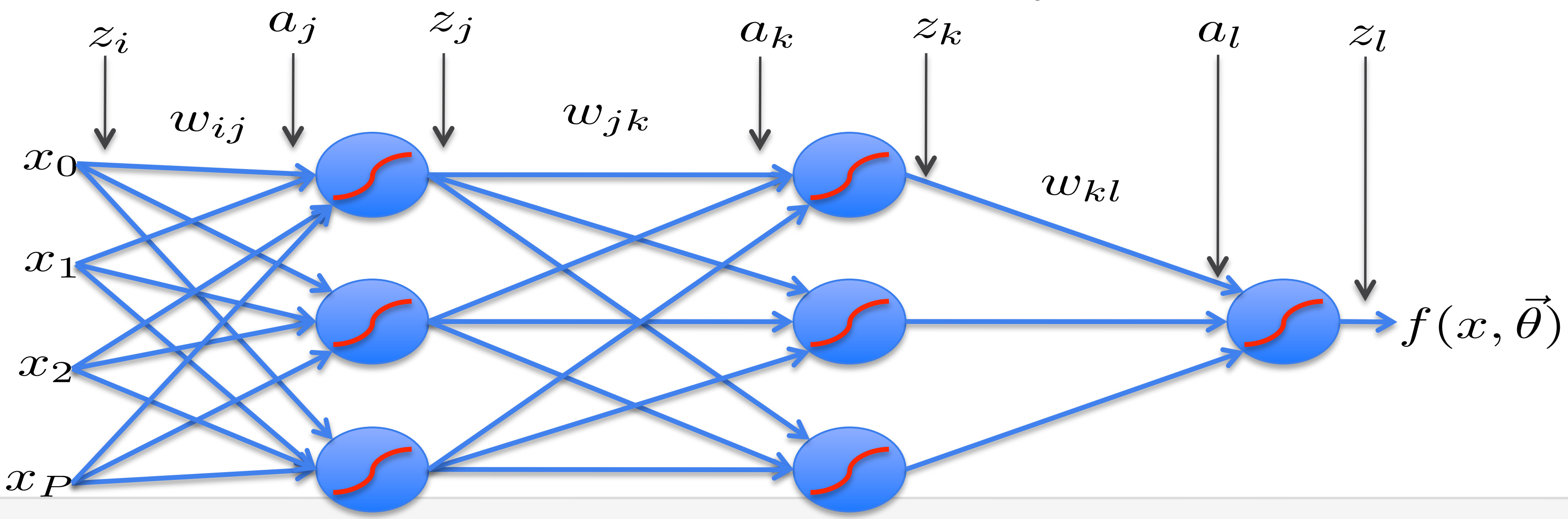
$$\vec{\theta} = \{w_{ij}, w_{jk}, w_{kl}\}$$



Error Backpropagation

$$\vec{\theta} = \{w_{ij}, w_{jk}, w_{kl}\}$$

- Introduce variables over the neural network
  - Define *a* to be the input of each non-linearity
  - Define *z* to be the output of each non-linearity

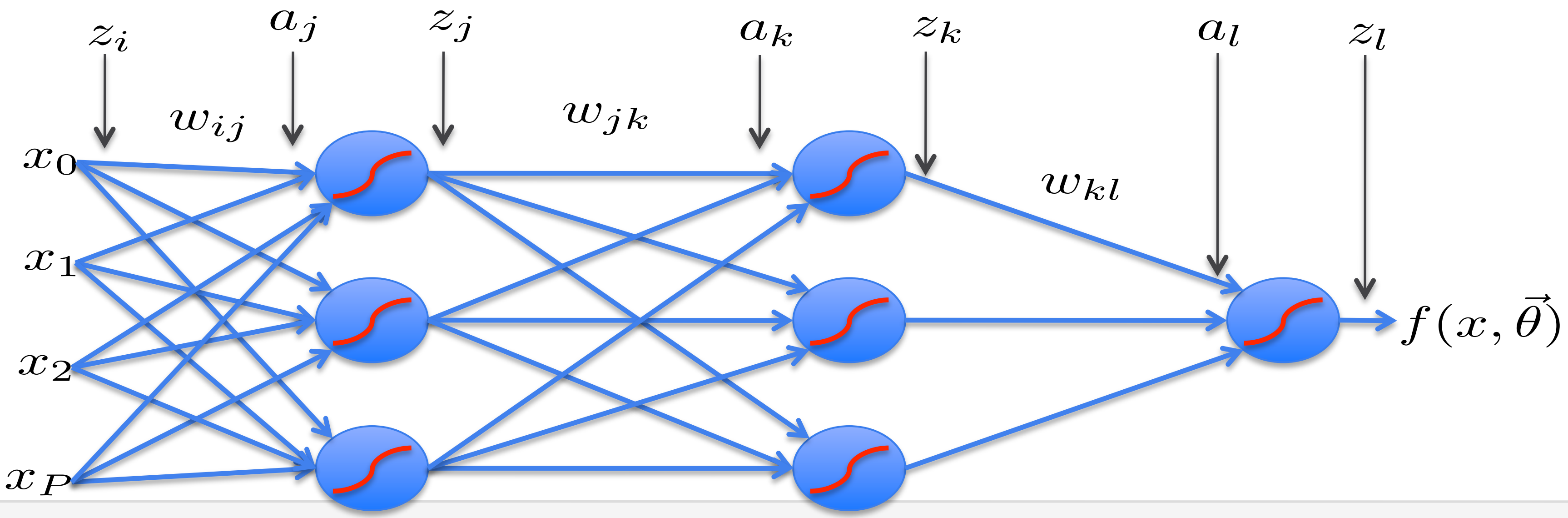


Error Backpropagation

$$\vec{\theta} = \{w_{ij}, w_{jk}, w_{kl}\}$$

$$a_j = \sum_i w_{ij} z_i \qquad a_k = \sum_j w_{jk} z_j \qquad a_l = \sum_k w_{kl} z_k$$

$$z_j = g(a_j) \qquad z_k = g(a_k) \qquad z_l = g(a_l)$$



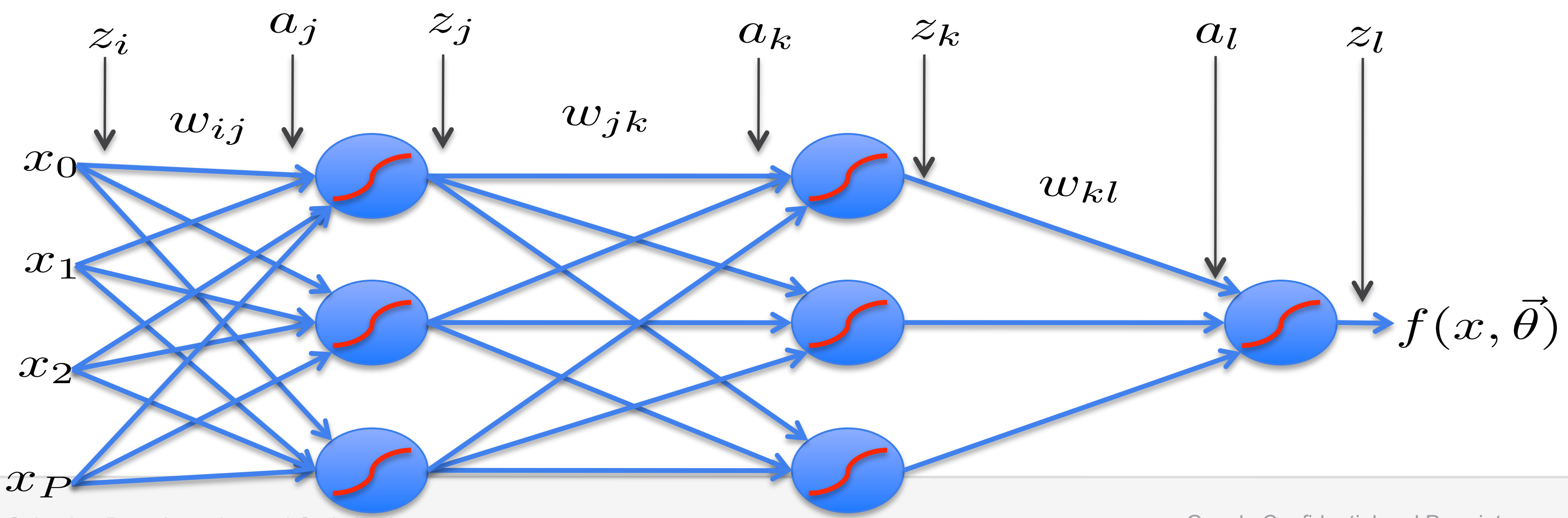
Error Backpropagation

$$\vec{\theta} = \{w_{ij}, w_{jk}, w_{kl}\}$$

Training: Take the gradient of the last component and iterate backwards

$$a_j = \sum_i w_{ij} z_i \qquad a_k = \sum_j w_{jk} z_j \qquad a_l = \sum_k w_{kl} z_k$$

$$z_j = g(a_j) \qquad z_k = g(a_k) \qquad z_l = g(a_l)$$



Error Backpropagation

Empirical Risk Function

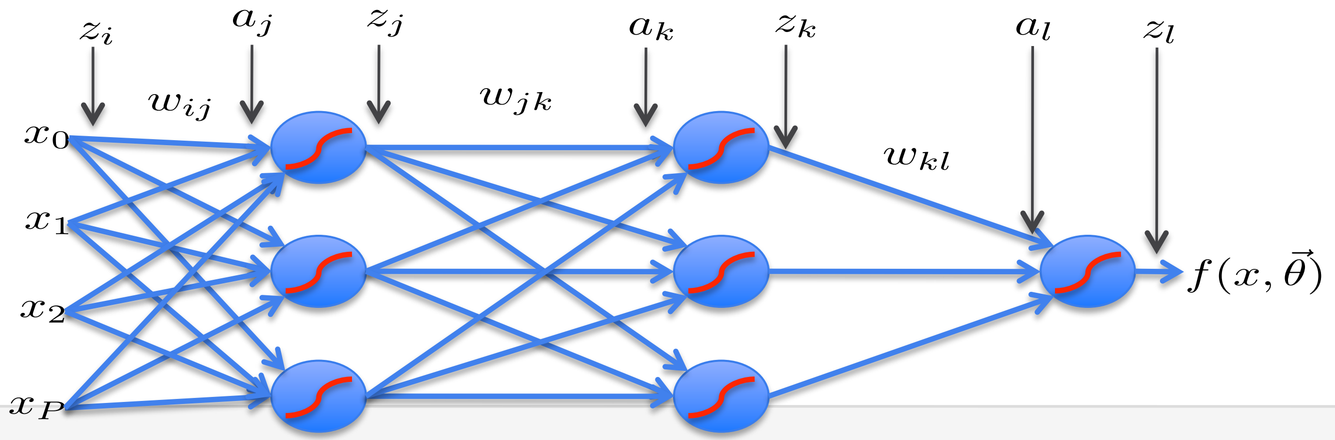
n is number of training points

$$R(\theta) = \frac{1}{N}\sum_{n=1}^{N} L_n(y_n, f(x_n,\theta))$$

$$= \frac{1}{N}\sum_{n=1}^{N}\frac{1}{2}(y_n - f(x_n,\theta))^2$$

$$= \frac{1}{N}\sum_{n=1}^{N}\frac{1}{2}\left(y_n - g\left(\sum_k w_{k1}\, g\left(\sum_j w_{jk}\, g\left(\sum_i w_{ij} x_{n,i}\right)\right)\right)\right)^2$$

Composition of weights + nonlinearity creates a nonconvex objective function



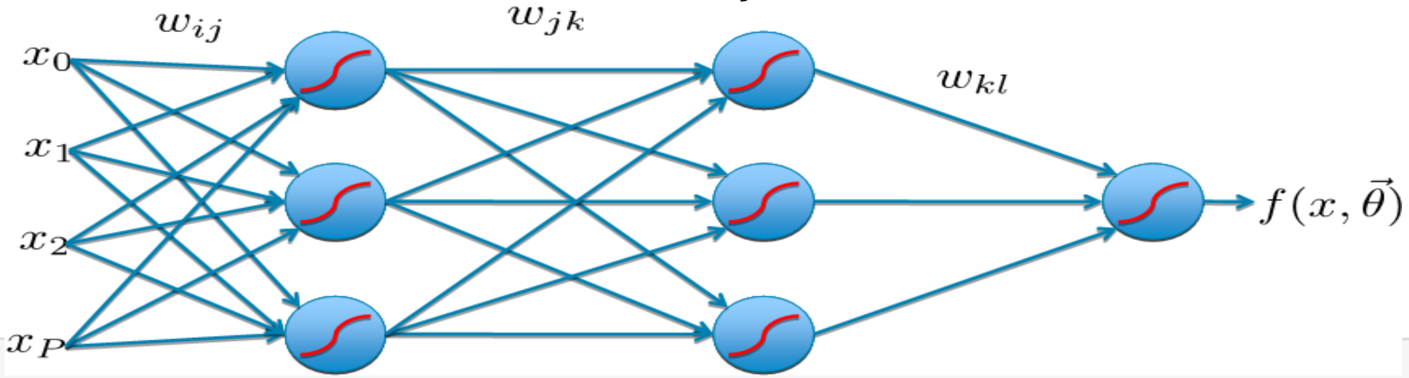
Error Backpropagation

- Compute the gradient with respect to objective function and propagate the gradient backwards to update each layer
- Stochastic Gradient Descent (SGD) is the most popular optimization strategy

$$w_{kl} = w_{kl} - \eta \frac{\partial L}{\partial w_{kl}}$$

$$w_{jk} = w_{jk} - \eta \frac{\partial L}{\partial w_{jk}}$$

$$w_{ij} = w_{ij} - \eta \frac{\partial L}{\partial w_{ij}}$$



Outline

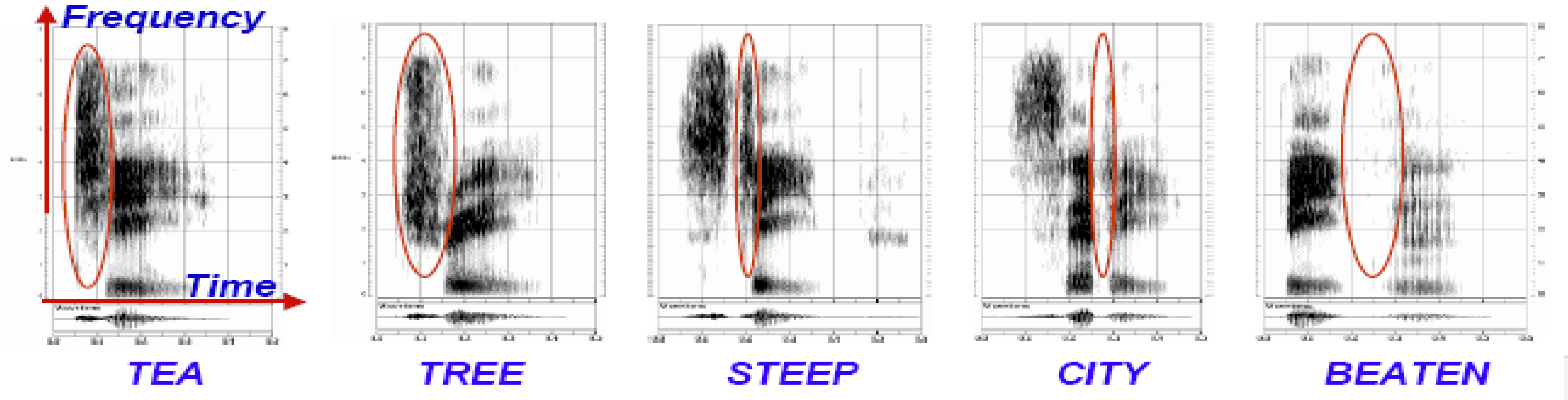
- Why Neural Networks
- Training Neural Networks
- Making Neural Networks Work for Speech Recognition
- Optimization challenges: Training time
- Optimization challenges: New architectures

Acoustic Modeling for Speech Recognition

- Speech recognition problem characterized as follows:

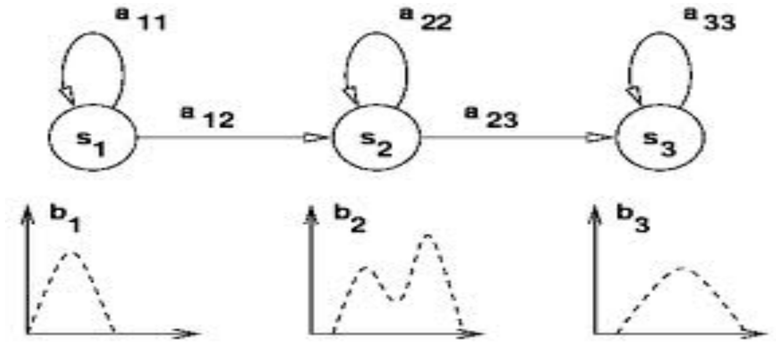
$$\widehat{W} = \arg\max_{W} P(W|A) = \arg\max_{W} P(A|W)P(W)$$

- Acoustic modeling is the process of modeling a set of sub-word units which make up words
- Acoustic realization of a phoneme depends strongly on context
- We model sub-word units as triphones (context-dependent states)



Acoustic Modeling for Speech Recognition

- Each sub-word unit is modeled by a 3-state Hidden Markov Model



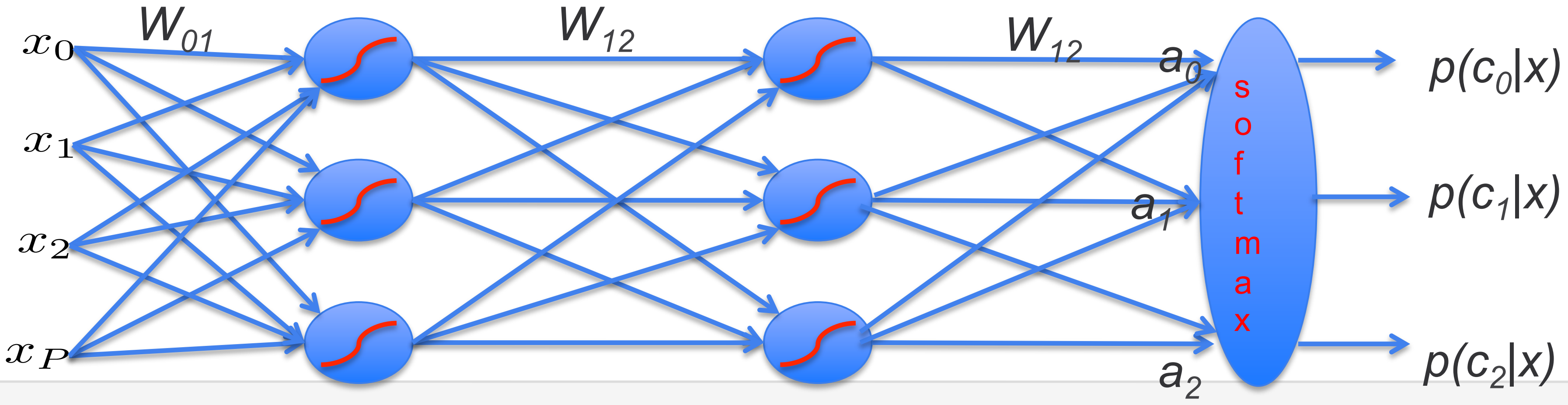
- 5 years ago, a popular acoustic modeling technique is to model the output distribution in each state by Gaussian Mixture Models (GMM)
- Neural Networks (alternatively called Multi Layer Perceptrons – MLPs) can also be used for acoustic modeling

Neural Network Acoustic Models

- Final non-linearity is represented by softmax

$$p(c_i|x) = \frac{\exp(-a_i)}{\sum_{j=1}^{n} \exp(-a_j)}$$

- Each class c_i will be the same sub-word units we build GMMs for



Neural Network Acoustic Models

- Neural networks are trained to minimize cross-entropy objective function (i.e. frame error rate)

$$y_i^{ref}, p(c_i|x) \in [0,1]^N$$
$$\sum_{i=1}^{N} y_i^{ref} = 1$$
$$\sum_{i=1}^{N} p(c_i|x) = 1$$

$$L = -\sum_{i=1}^{N} y_i^{ref} \log p(c_i|x)$$

- NN gives posterior *p(c$_i$|x)* so divide by class prior to get likelihood

$$p(x|c_i) = \frac{\mathrm{p}(c_i|x)}{p(c_i)}$$

- NN likelihood replaces GMM likelihood as output distribution in HMM

Early Performance of Neural Networks

- Previous LVCSR performance with MLPs
    - shallow network (3 layers), small output targets (46) - [Zhu et al, ICSLP 2004]
- On a Switchboard telephony task, gains with MLPs only observed when combined with baseline

Method	WER
Baseline GMM/HMM	30.8
+ MLP	28.6

- Training neural networks is difficult!
    - Objective function is non-convex
    - Training is done SGD serially one one machine, can be slow on CPUs
- These difficulties pose challenges to have deep networks with many output targets

Making DNNs Successful for Acoustic Modeling

3 advances made DNNs successful for acoustic modeling

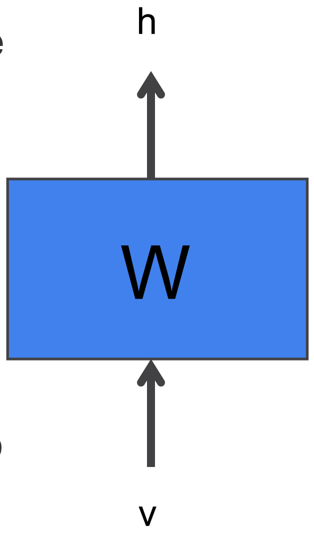
1. Pre-training
2. Improved Hardware with GPUs
3. Sequence training [B. Kingsbury et al, Interspeech 2012]

This encouraged

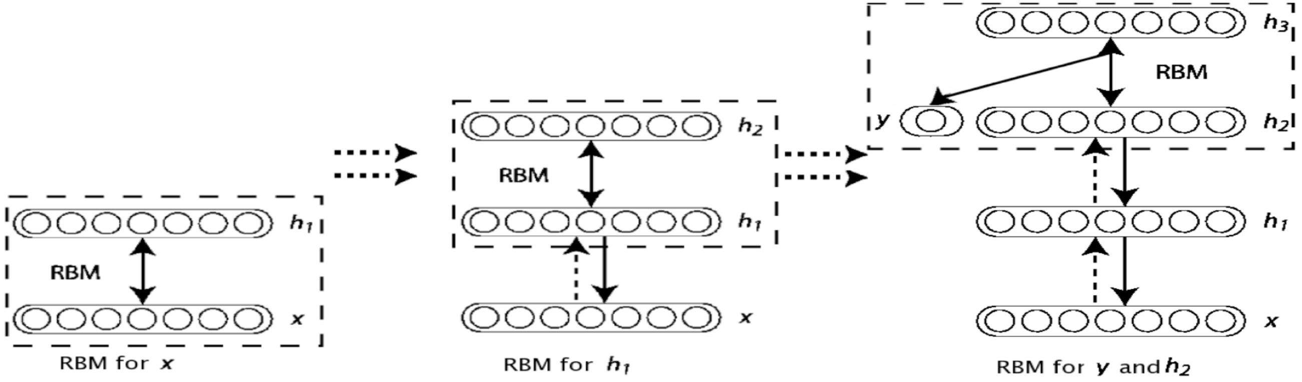
- Deeper networks
- Networks with more output targets (i.e., classes)

(1) Pre-Training via Unsupervised Learning

- The goal of unsupervised learning is to put the weights in a good initial space to encourage deeper and larger networks during superivsed fine-tuning
- Unsupervised learning systems can be designed using the encoder-decoder paradigm
  - Encoder: transform input *v* into code representation *h*
  - Decoder: reconstructs input from the code by minimizing reconstruction error
- Encoder-decoder paradigm learns weights such that the code captures higher-order relevant information from input signal, these weights are used to initialize network for fine-tuning
- Unsupervised learning
  - Restricted Boltzmann Machine (RBM) [Hinton – Toronto]
  - Sparse Encoding Symmetric Machine (SESM) [Lecun – NYU]
  - De-noising Auto-Encoder [Bengio – Montreal]



Greedy Layer-wise Pre-training



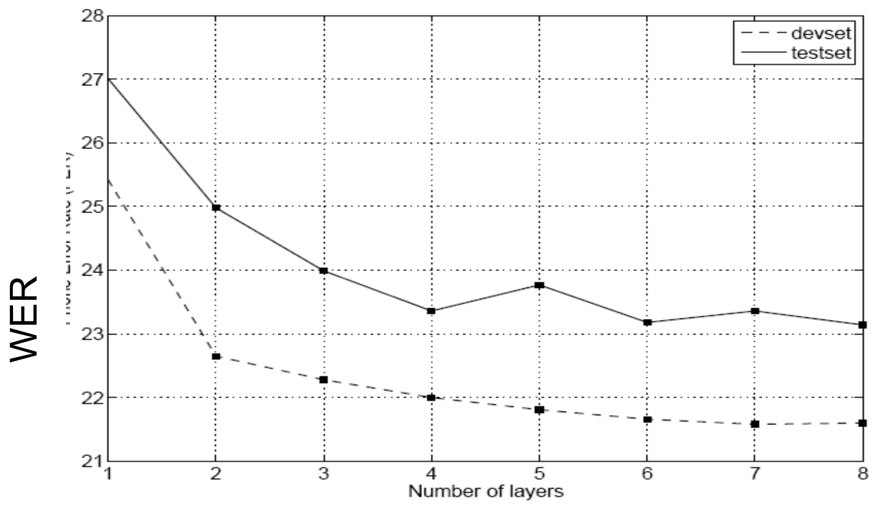
- First train a layer of features that receive input directly from the speech features
- Then treat the activations of the trained features as if they were speech features and learn features of features in a second hidden layer.
- After pre-training is done, use weights and train network using cross-entropy objective

(2) GPU training

- DNN training is slow is due to the large number of dense matrix multiplications and large amount of training data
- GPUs help SGD DNN training by parallelizing this matrix multiplication over thousands of cores
- GPU training can achieve over a 9x speedup with a K20x speedup to a compared to a 8-core CPU

Method	WER (50-hr BN)	Training Time (hrs)
SGD (CPU)	17.8	35
SGD (GPU) – K20x	17.8	3.8

Results with PT+CE: Deepness



- Experimentally, we see that network depth improves WER
- Generally 6-7 hidden layers is used for speech tasks

Results with PT+CE: Increased Output Targets

- We know with GMM/HMMs, increasing the number of context-dependent states (i.e., classes) improves performance
- In the past, MLPs typically trained with small number of outputs
  - increasing output targets becomes a harder optimization problem and does not always improve WER
  - increases parameters → increases training time
- With DNNs, **pre-training** putting weights in better space, and thus we can **increase output targets** effectively

Number of Targets	WER
384	21.3
512	20.8
1024	19.4
2,220	**18.5**

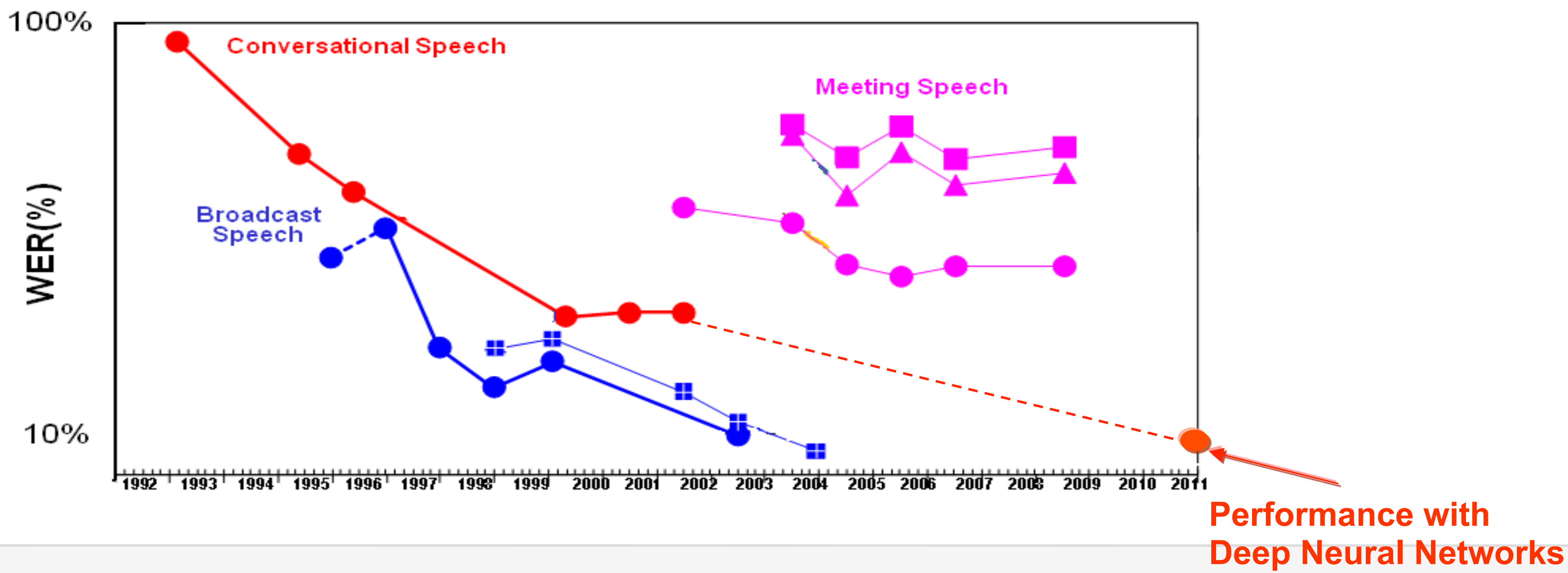
DNN Acoustic Modeling Results

- DNNs provide between a 8-25% relative improvement in word error rate over GMM/HMM systems across a variety of tasks and languages
- Results confirmed by many, many research labs

	300 hour SWB Conversational Telephony	400 hour Broadcast News	2000 hour Voice Search
GMM/HMM	14.3	16.5	16.0
DNN	12.2	15.2	12.2
% Relative Improvement	**14.7**	**7.9**	**23.8**

Historical Performance in Speech Recognition

- Few techniques we explore ***consistently*** show gains of this ***magnitude***



Outline

- Why Neural Networks
- Training Neural Networks
- Making Neural Networks Work for Speech Recognition
- Optimization challenges: Training time
- Optimization challenges: New architectures

The Revolution

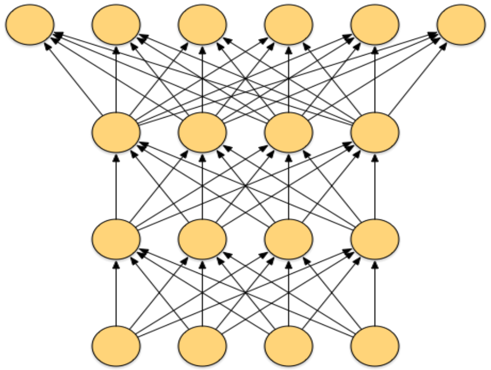
- The 2007 launch of smartphones (iPhone and Android) was a revolution and dramatically changed the status of speech processing
- Google’s current suite of mobile applications is launched in 48 languages and processes about a decade of speech each day

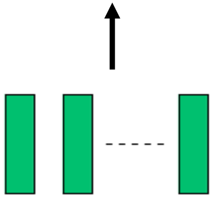
Mobile Speech Will Grow

- Speech becomes the primary input modality
- Training data will continue to grow
- With this, we need efficient algorithms to train these networks

Stochastic Gradient Descent

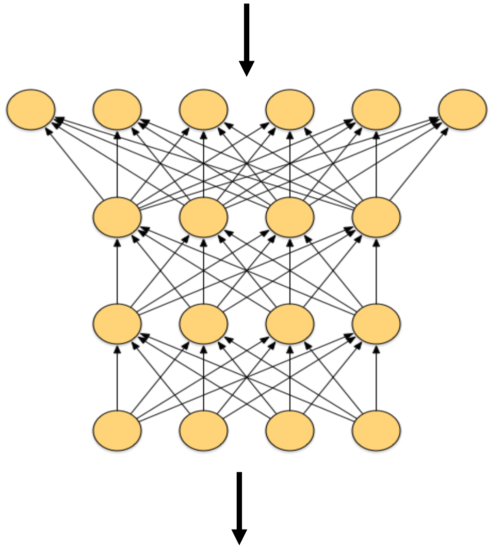
$$P(\hat{S}_1^N \mid \Lambda, x_1^N)$$





$$(x_1, S_1) \dots (x_N, S_N)$$

$$\text{Error}(P(\hat{S}_1^N \mid \Lambda, x_1^N), S_1^N)$$



$$\Lambda' = \Lambda - \eta \nabla_\Lambda \text{Error}$$

GPU training

- DNN training is slow is due to the large number of dense matrix multiplications and large amount of training data
- GPUs help SGD DNN training by parallelizing this matrix multiplication over thousands of cores
- GPU training can achieve over a 9x speedup with a K20x speedup to a compared to a 8-core CPU
- In reality applications of speech, text and NLP thousands of hours of labeled training data, even more unlabeled data
- It is critical for DNN success to speed up training

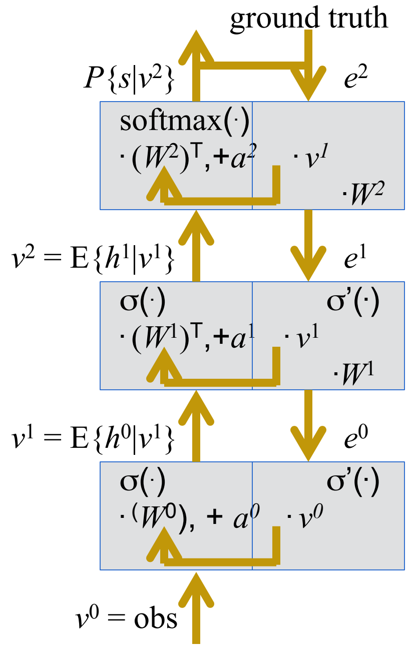
Method	WER (50-hr BN)	Training Time (hrs)
SGD (CPU)	17.8	35
SGD (GPU) – K20x	17.8	3.8

Approaches to Speed up DNN Training

- Parallel SGD on GPUs [Microsoft]
- Asynchronous SGD [Google]
- Hessian-free training on Blue Gene [IBM]

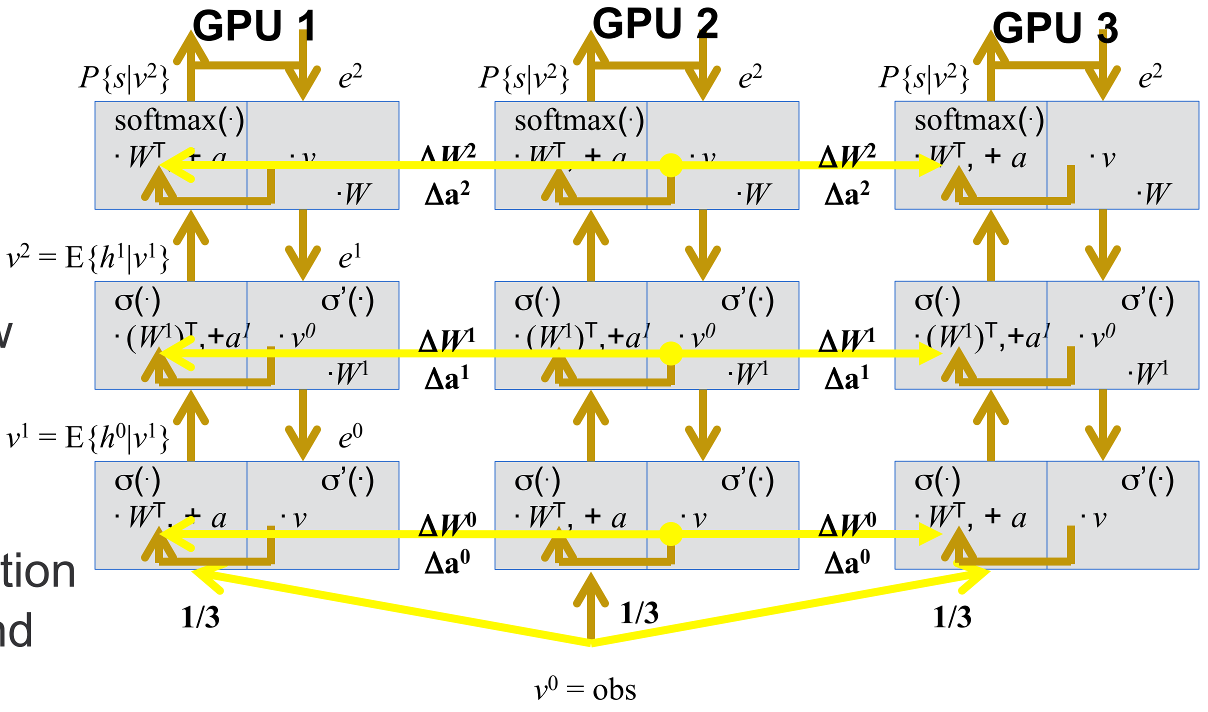
(1) Parallel SGD

- BP steps:
  - forward propagation
  - error back propagation
  - model update



Parallel SGD

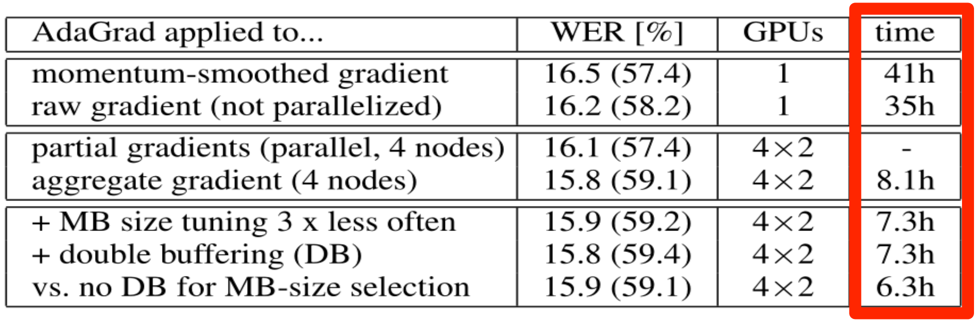
- BP steps:
  - forward propagation
  - error back propagation
  - model update
- To improve efficiency of data parallelism, reduce how much data gets exchanged by quantizing sub-gradients to one bit/value
- Key trick is to keep quantization error from one mini-batch and add it to the next



Results

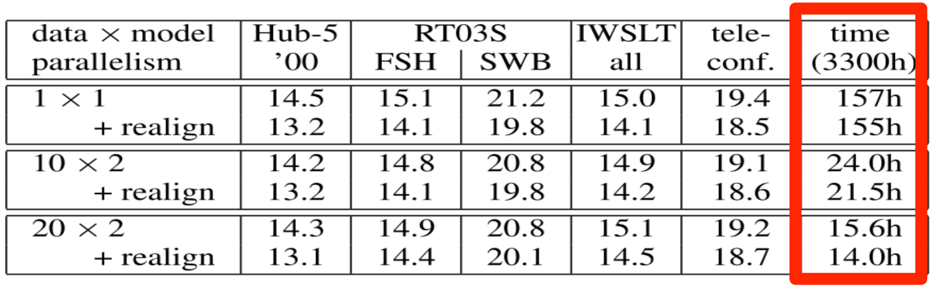
[F. Seide et al, Interspeech 2014]

- SWBD 309h (46M): total reduction from 41 to 6.3h → **6.5 x** (8-GPU server)



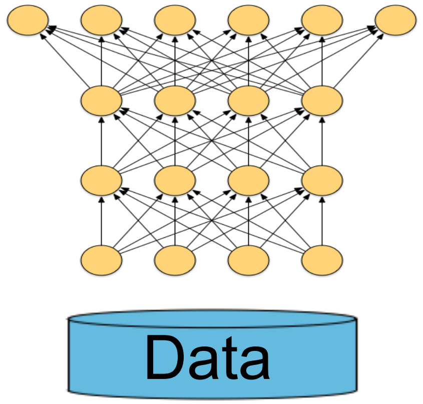
AdaGrad applied to...	WER [%]	GPUs	time
momentum-smoothed gradient	16.5 (57.4)	1	41h
raw gradient (not parallelized)	16.2 (58.2)	1	35h
partial gradients (parallel, 4 nodes)	16.1 (57.4)	4×2	-
aggregate gradient (4 nodes)	15.8 (59.1)	4×2	8.1h
+ MB size tuning 3 x less often	15.9 (59.2)	4×2	7.3h
+ double buffering (DB)	15.8 (59.4)	4×2	7.3h
vs. no DB for MB-size selection	15.9 (59.1)	4×2	6.3h

- 3300h S2S/MAVIS model (160M): 312 to 45.5h → **6.9 x** (20 GPUs GCD)

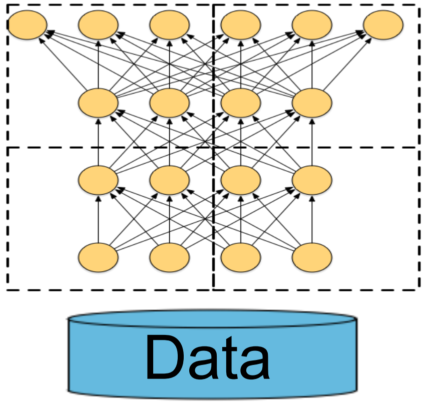


data × model parallelism	Hub-5 '00	RT03S FSH	RT03S SWB	IWSLT all	tele-conf.	time (3300h)
1 × 1	14.5	15.1	21.2	15.0	19.4	157h
+ realign	13.2	14.1	19.8	14.1	18.5	155h
10 × 2	14.2	14.8	20.8	14.9	19.1	24.0h
+ realign	13.2	14.1	19.8	14.2	18.6	21.5h
20 × 2	14.3	14.9	20.8	15.1	19.2	15.6h
+ realign	13.1	14.4	20.1	14.5	18.7	14.0h

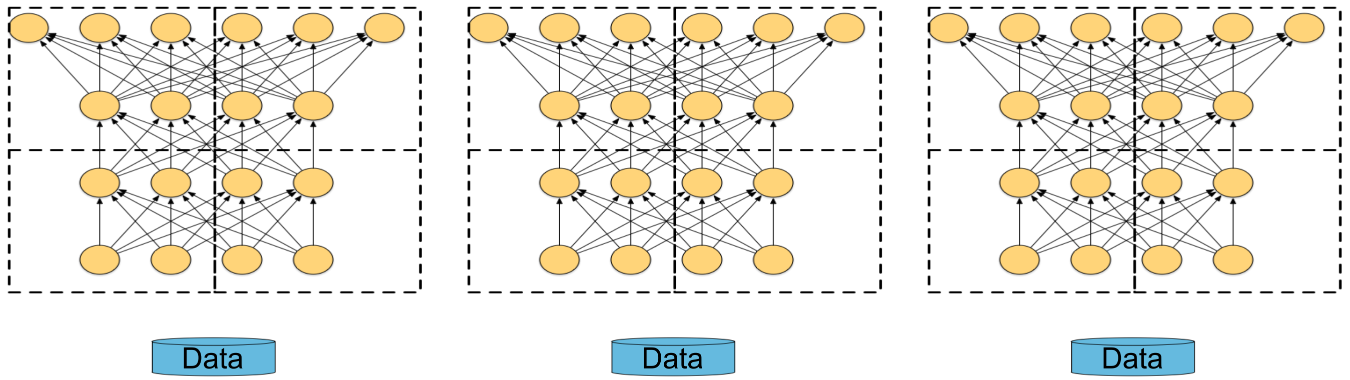
(2) Asynchronous SGD - DistBelief Parallel Trainer



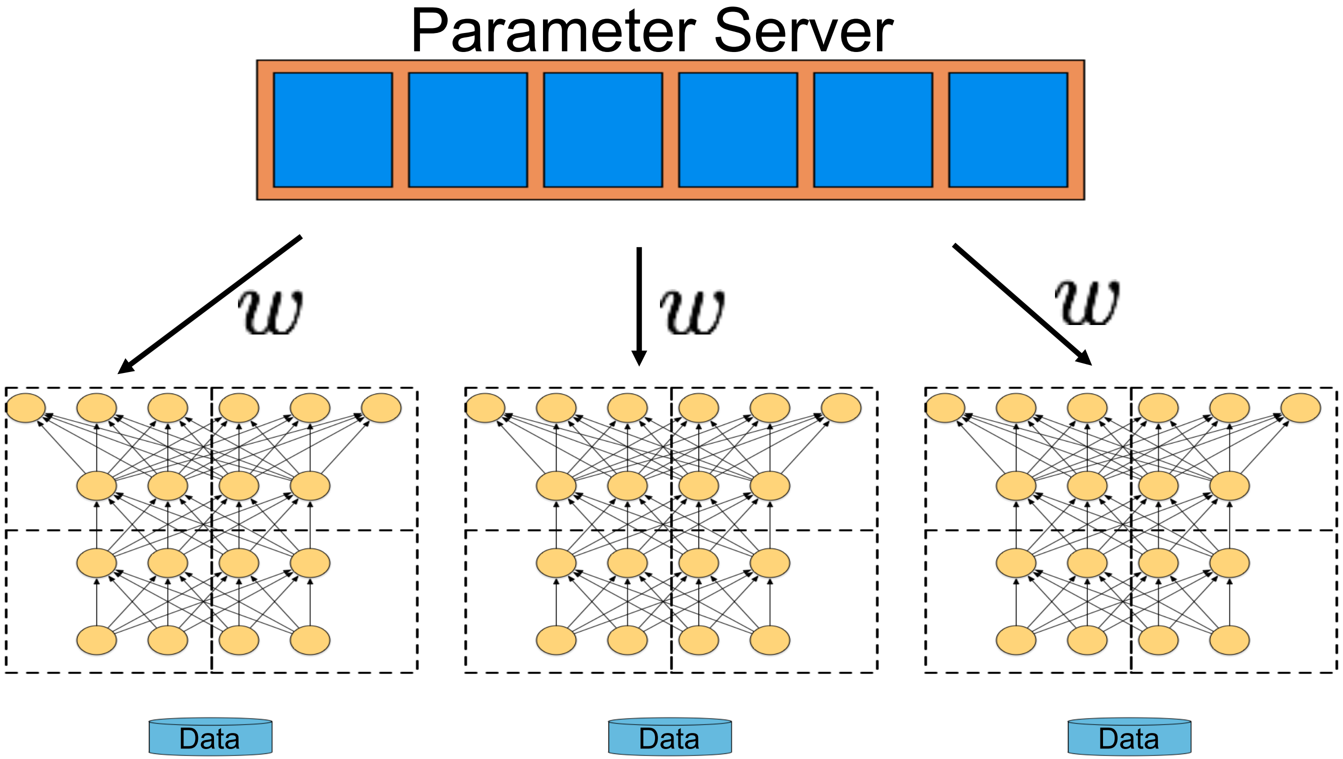
Asynchronous SGD - DistBelief Parallel Trainer



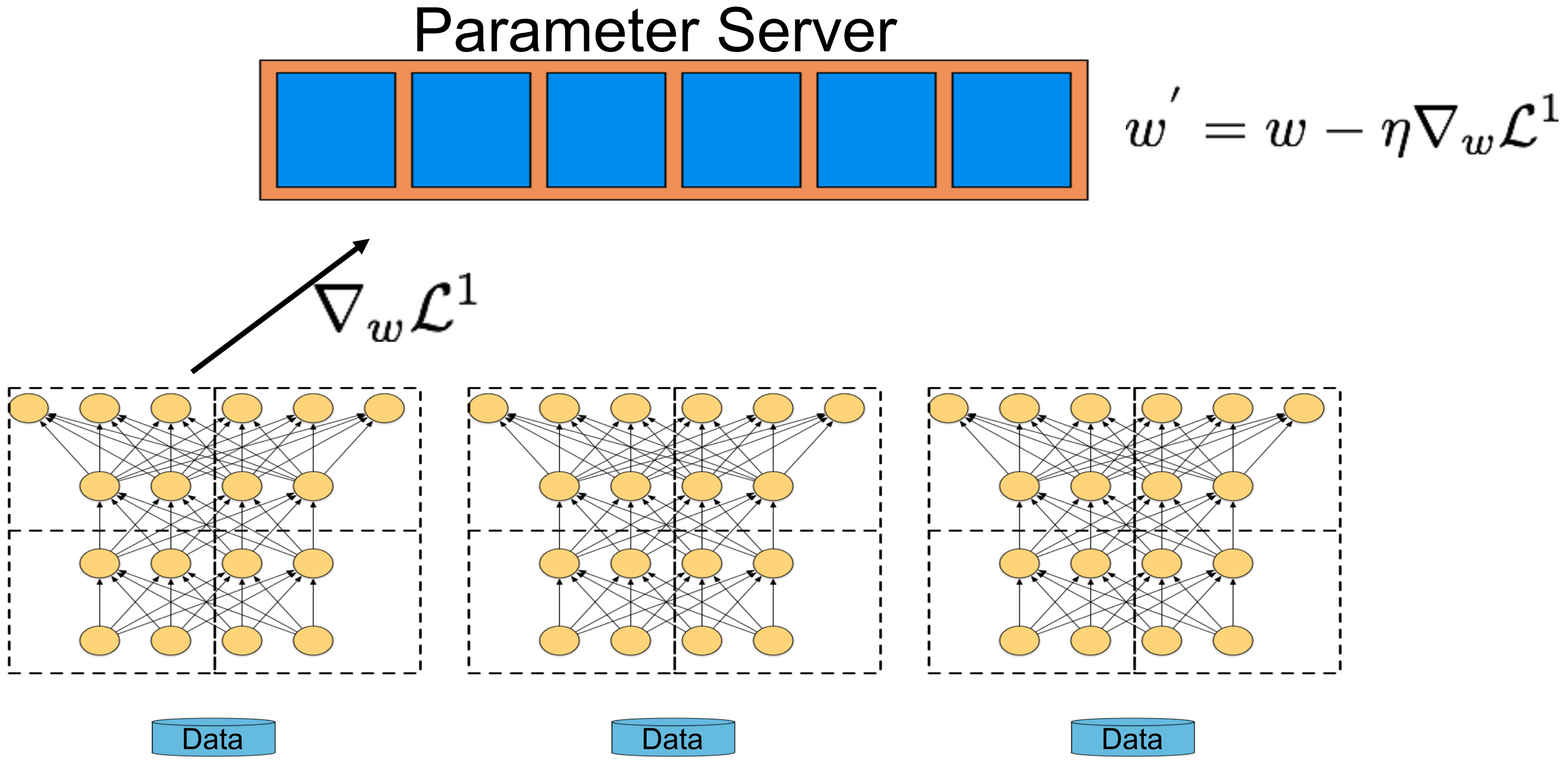
Asynchronous SGD - DistBelief Parallel Trainer



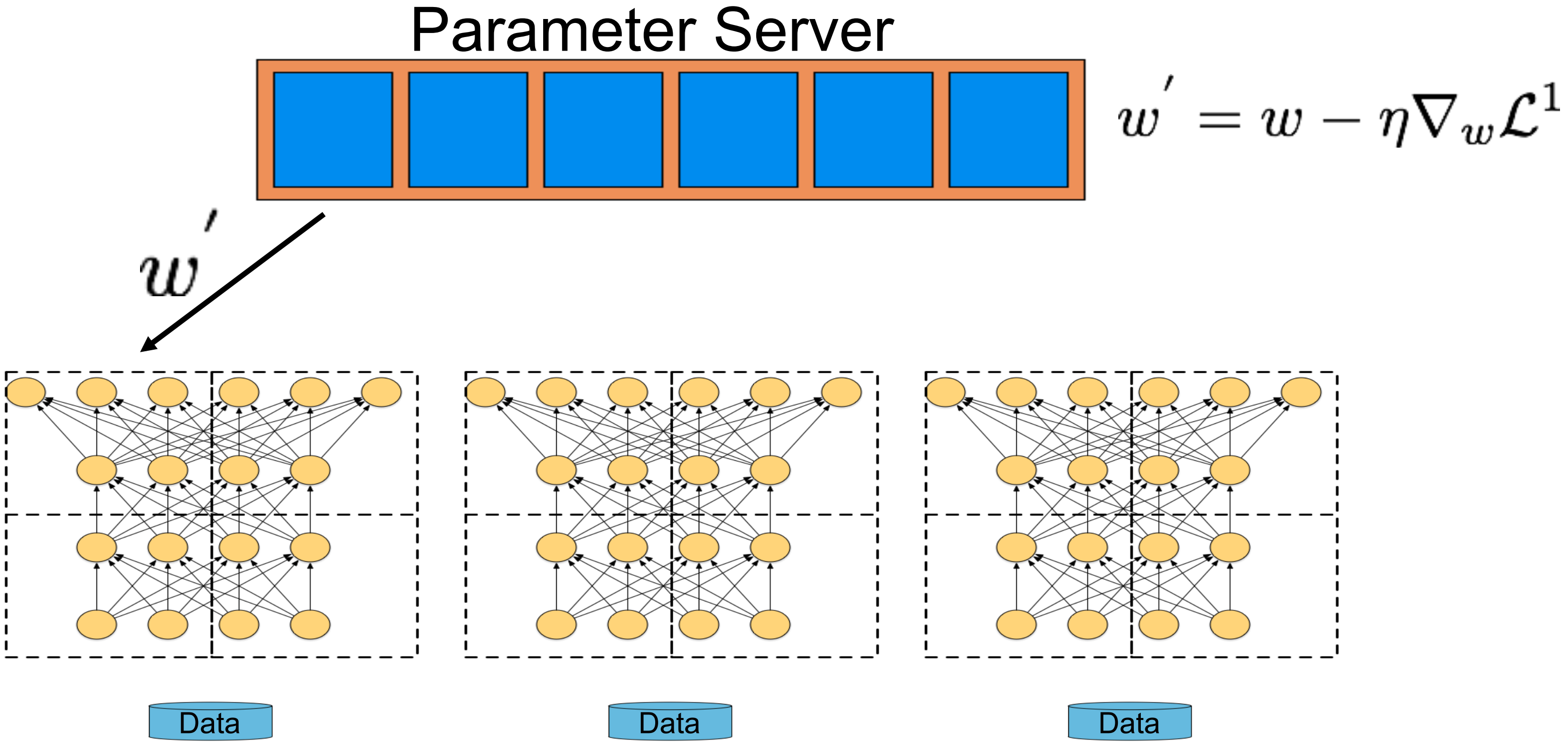
Asynchronous SGD - DistBelief Parallel Trainer



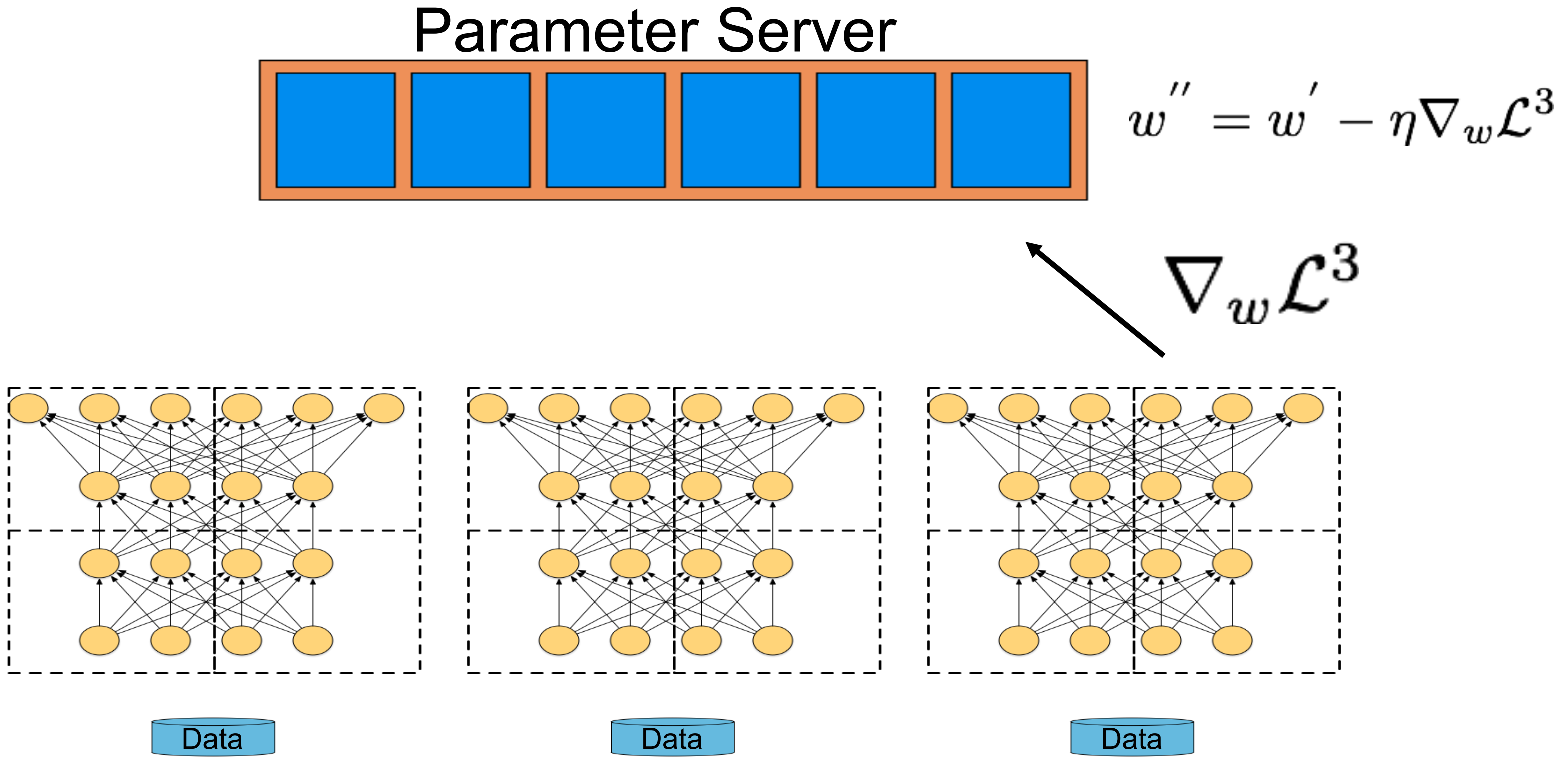
Asynchronous SGD - DistBelief Parallel Trainer



Asynchronous SGD - DistBelief Parallel Trainer



Asynchronous SGD - DistBelief Parallel Trainer



ASGD Training Time

[J. Dean et al, NIPS 2012]

Task	Model Type	WER	Training Size (hours)	GPU Training Time (hours /epoch)	Hidden Layers	Number of States
Voice Search	GMM	16.0	5780	321	4x2560	7969
	DNN	12.2				
You Tube	GMM	52.3	1400	55	4x2560	17552
	DNN	46.2				

- DistBelief CPU training allows speed ups of 70 times over a single CPU and 5 times over a GPU.
- Train a 85M parameter system on 2,000 hours, 10 epochs in about 10 days.

(3) 2nd Order Optimization via Hessian-free

[B. Kingsbury et al, Interspeech 2012]

- Distributed optimization techniques, such as 2nd order methods, use large data batches for gradient and curvature information, which can be parallelized across machines

1. Minimize the following objective function

$$\mathcal{L}(\mathbf{w}_n + \mathbf{d}_n) - \mathcal{L}(\mathbf{w}_n) \approx \nabla\mathcal{L}(\mathbf{w}_n)^T\mathbf{d}_n + \frac{1}{2}\mathbf{d}_n^T\mathbf{B}(\mathbf{w}_n)\mathbf{d}_n$$
$$\equiv q_{\mathbf{w_n}}(\mathbf{d}_n)$$

2. Find the best search direction d_n

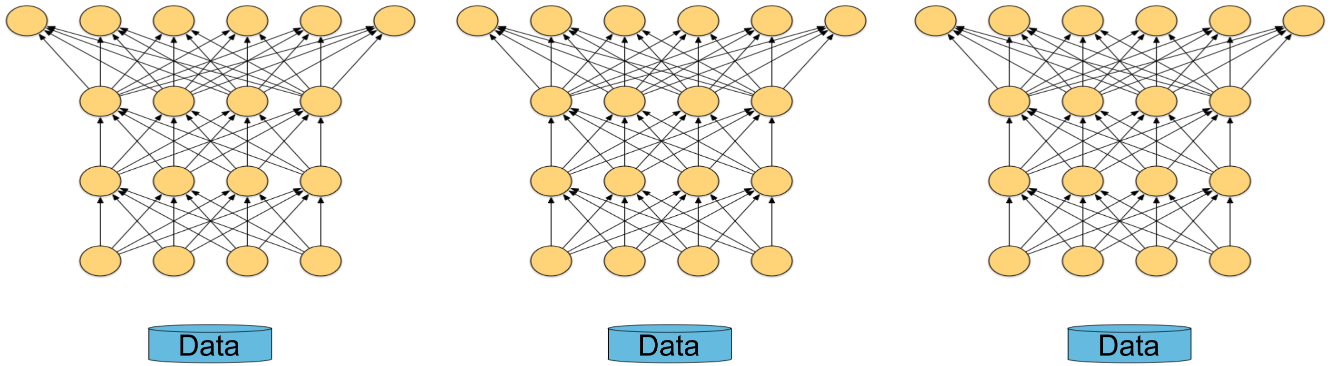
$$\text{Find } \mathbf{d}_n \text{ such that } q_{\mathbf{w_n}}(\mathbf{d}_n) < q_{\mathbf{w_n}}(\mathbf{0}).$$

3. Update parameters

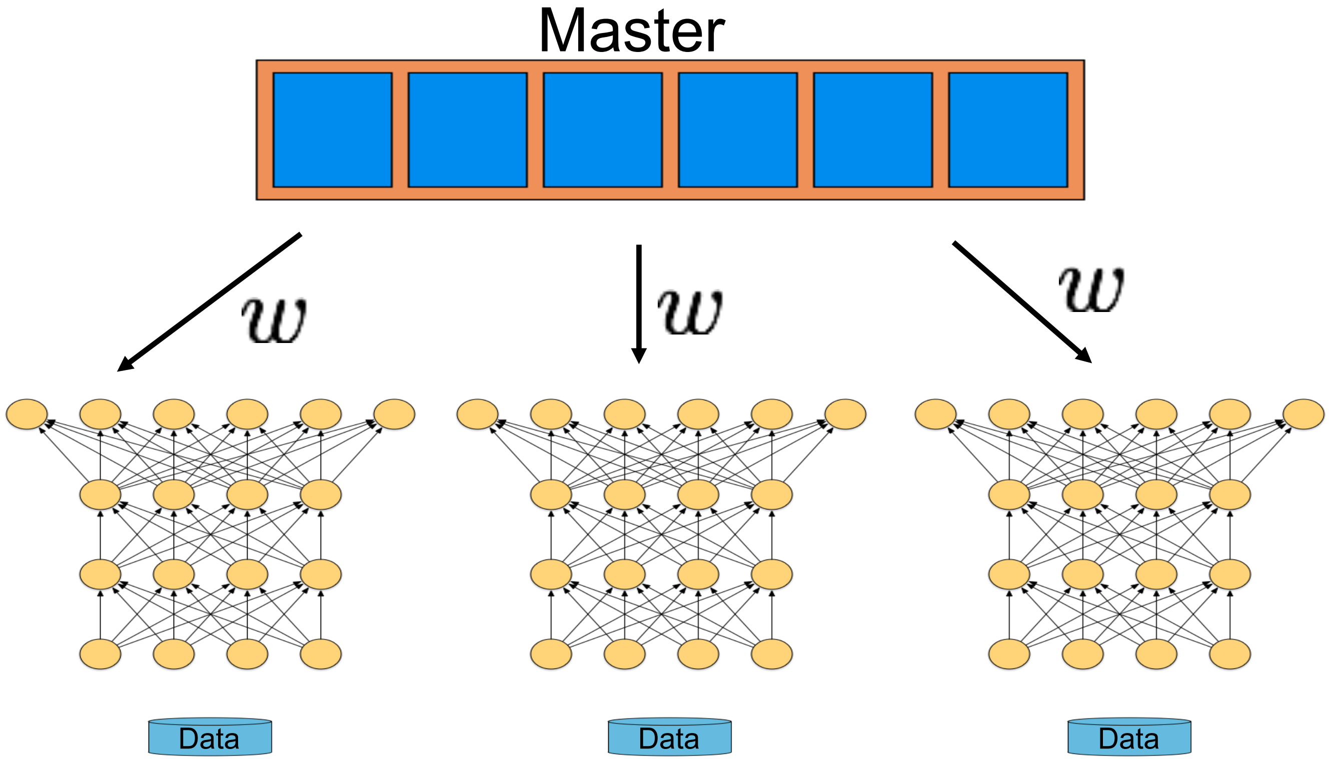
$$\mathbf{w}_{n+1} \leftarrow \mathbf{w}_n + \alpha\mathbf{d}_n$$

4. Iterate

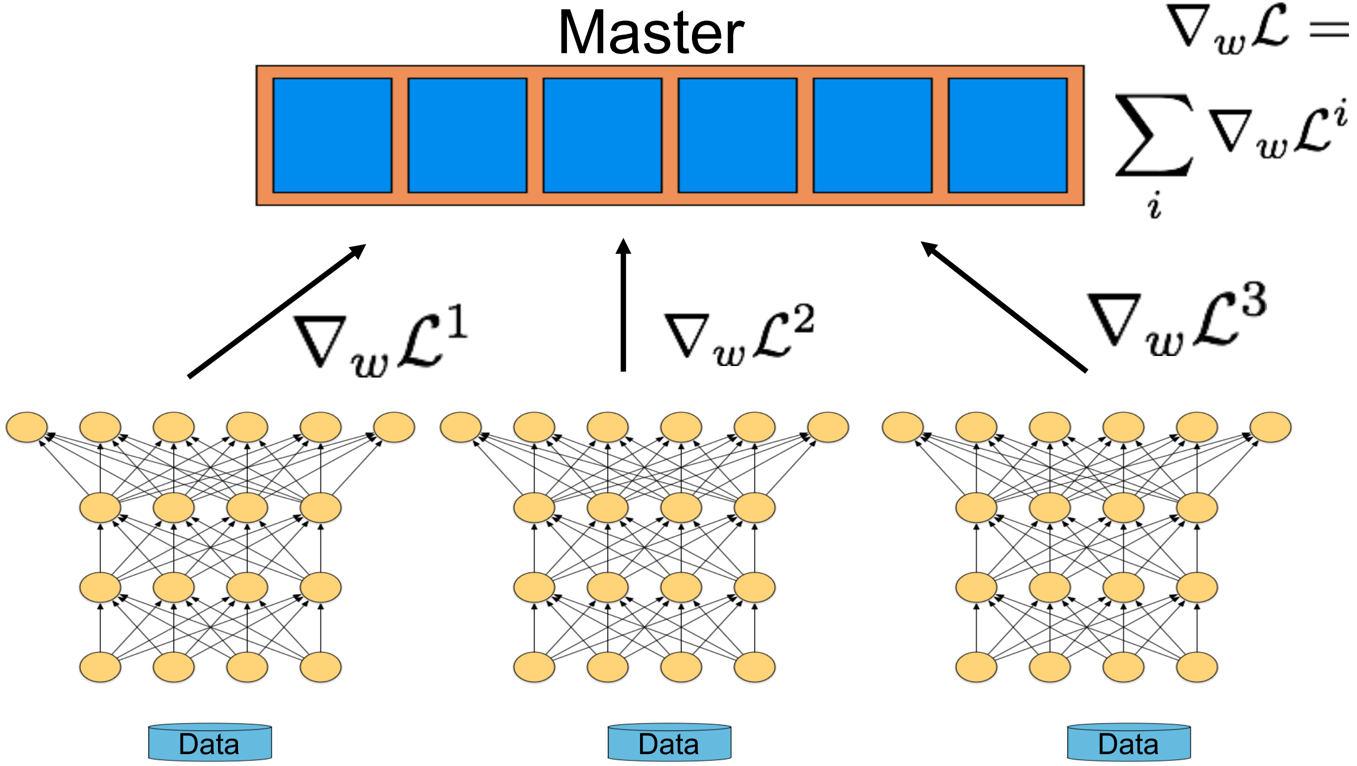
Hessian-free Training



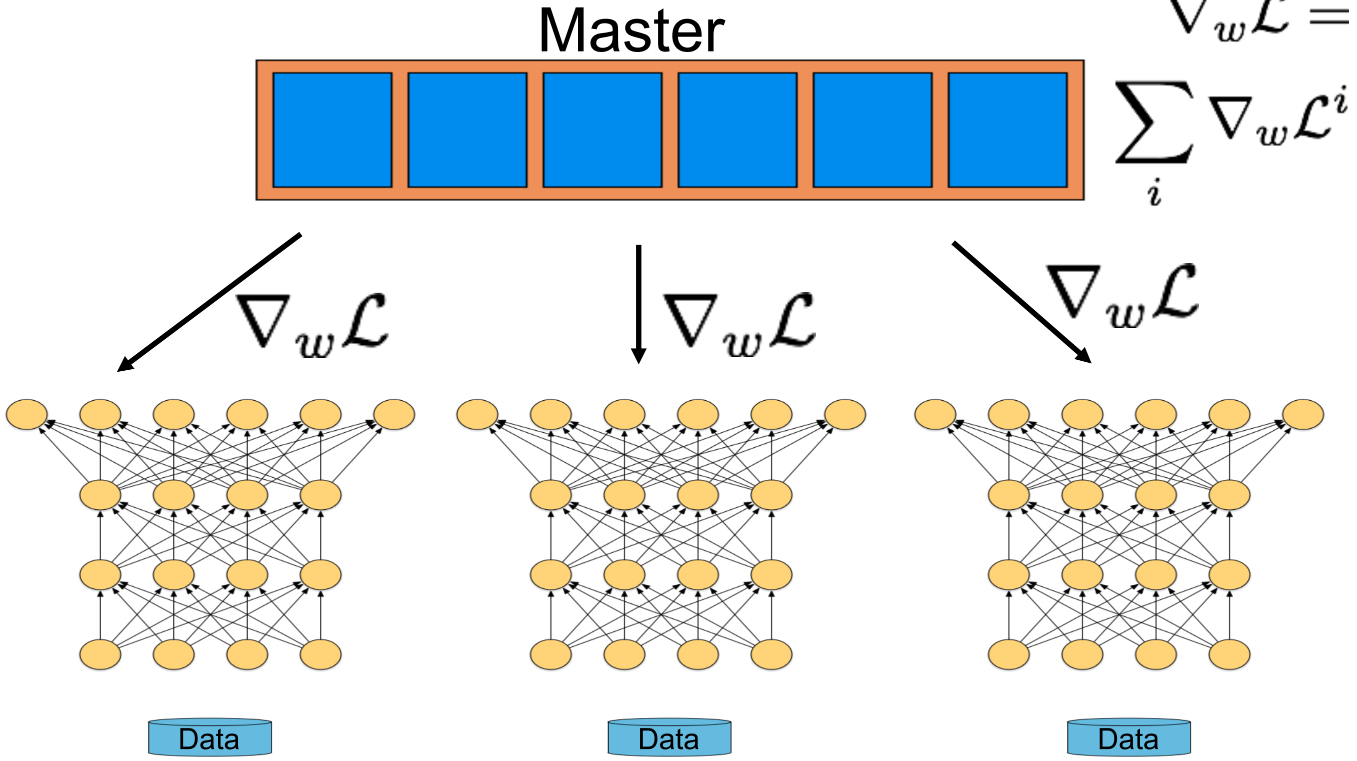
Hessian-free Training



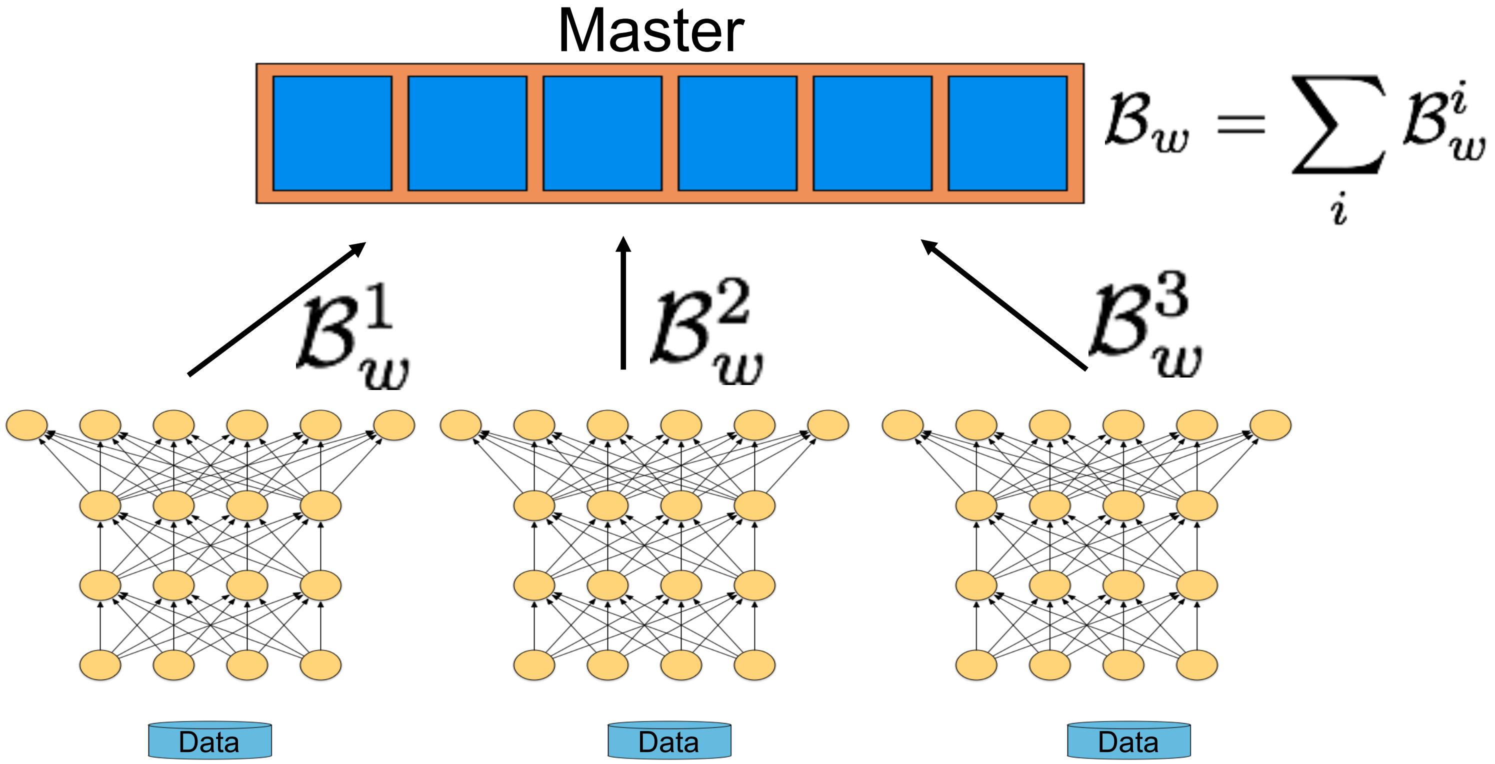
Hessian-free Training



Hessian-free Training



Hessian-free Training



$$\mathcal{B}_w = \sum_i \mathcal{B}_w^i$$

$\mathcal{B}_w^1$ $\mathcal{B}_w^2$ $\mathcal{B}_w^3$

Further Speedups with Blue Gene/Q

- A major problem with parallel architectures is communication bottlenecks between workers
- Having a specialized hardware/software architecture to minimize these bottlenecks is critical
- The Blue Gene/Q architecture is perfect for 2nd order parallel HF training
  - Massively parallel architecture with thousands of cores
  - Minimal communication cost between processors

Overall Speedups with BG/Q – 300 hrs SWB

[T. N. Sainath et al, Interspeech 2014]

- Experiments run on two BG/Q rack (2,048 nodes, 16 cores/node, 4 threads/core)
- On 300-hr SWB, BG/Q is
  - 4x faster for CE compared to SGD GPU
  - 10x faster for ST compared to SGD GPU
  - No loss in WER compared to HF CPU

Algorithm	**Cross-Entropy (CE)**		**Sequence Training (ST)**	
	Time (hrs)	WER	Time (hrs)	WER
HF CPU	-	-	-	12.4
SGD GPU	121.5	14.1	47.6	12.7
HF BG/Q	**28.0**	**14.1**	**4.6**	**12.4**
Speedup	4	-	**10.3**	

Overall Speedups with BG/Q – 400 hrs BN

- On 400 hours BN with 2 racks, HF BG/Q shows
  - 3x faster for CE compared to SGD GPU
  - 11.6x faster for ST compared to SGD GPU
  - No loss in WER compared to HF CPU
- A specialized architecture such as BG/Q makes HF **the fastest approach** for CE and Sequence training

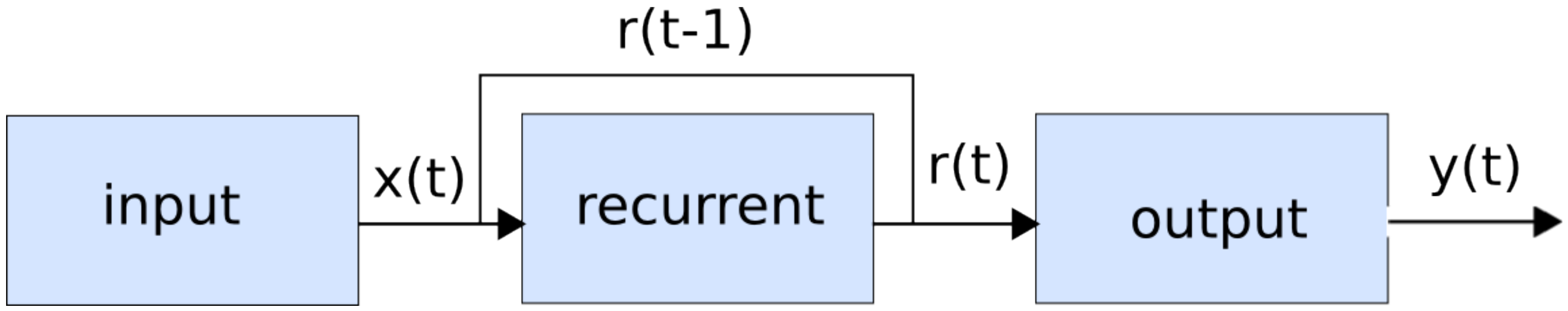
Hardware	Cross-Entropy		Sequence Training	
	Time (hrs)	WER	Time (hrs)	WER
HF CPU	-	-	-	15.1
SGD GPU	77.9	16.5	42.1	15.8
BG/Q	**21.7**	**16.5**	**3.6**	**15.1**
Speedup	3	-	11.6	

Outline

- Why Neural Networks
- Training Neural Networks
- Making Neural Networks Work for Speech Recognition
- Optimization challenges: Training time
- Optimization challenges: New architectures

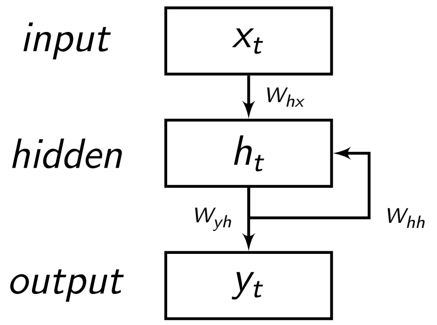
(1) Recurrent Neural Networks

- Activation from previous time step is fed as input to network at current time step
- Recurrent layer encodes “state” and can encode long-term temporal information
- RNNs good at modeling non-linear temporal sequence data [Robinson, 1993]

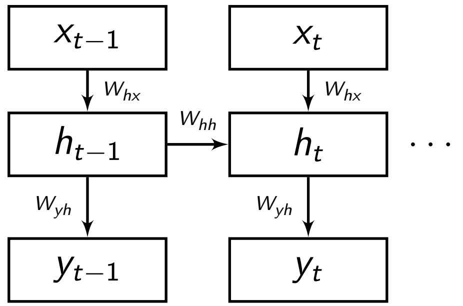


Training RNNs

- An RNN can be made to look like a feed forward network by unrolling the RNN through time
- During training, activations are forward propagated for a fixed time-step *T*
- Gradients are computed and then backpropagated to start (backpropagation through time)



(a) RNN

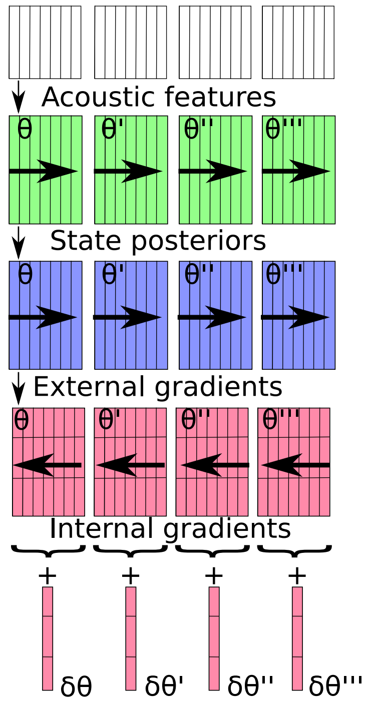


(b) RNN unrolled in time

$$h_t = \sigma(W_{hx}x_t + W_{hh}h_{t-1} + b_h)$$

$$y_t = \phi(W_{yh}h_t + b_y)$$

Backpropagation Through Time



RNN Architectures

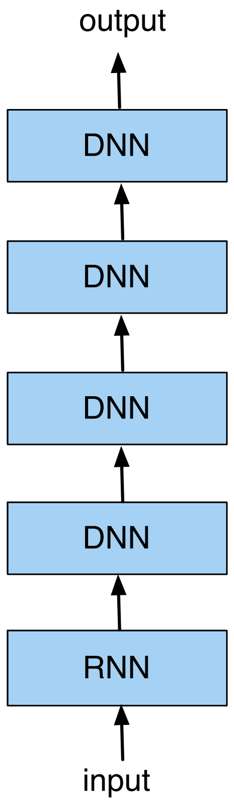
[G. Saon et al, Interspeech 2014]

- Explore using RNNs for temporal modeling and DNNs for depth in a unified framework
- RNN is unfolded for 6 time steps

Model	WER
RNN – unfolded 6	13.5
RNN – unfolded 11	13.8

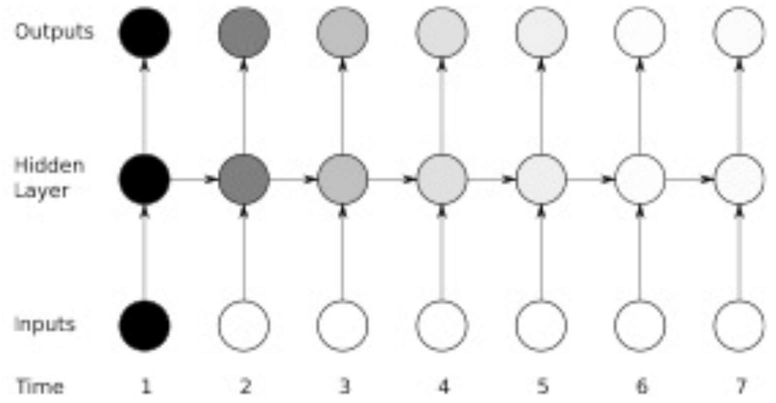
- RNN can achieve a 4% relative improvement over DNN

Model	SWB-300
Baseline GMM/HMM	14.5
DNN	12.5
RNN	**12.0**



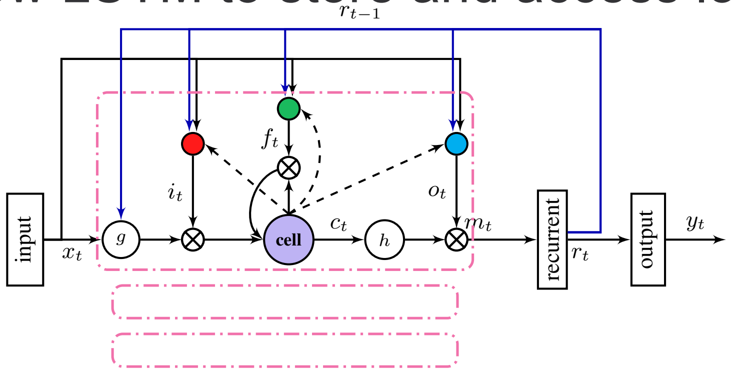
(2) Long Short-Term Memory RNNs

- Modeling long-term dependencies with RNNs is difficult due to vanishing gradient problem
- This limits modeling capability of RNNs to small time steps (5-10)
- LSTMs were developed to address these issues [Hochreiter and Schmidhuber, 1997]



LSTM architecture

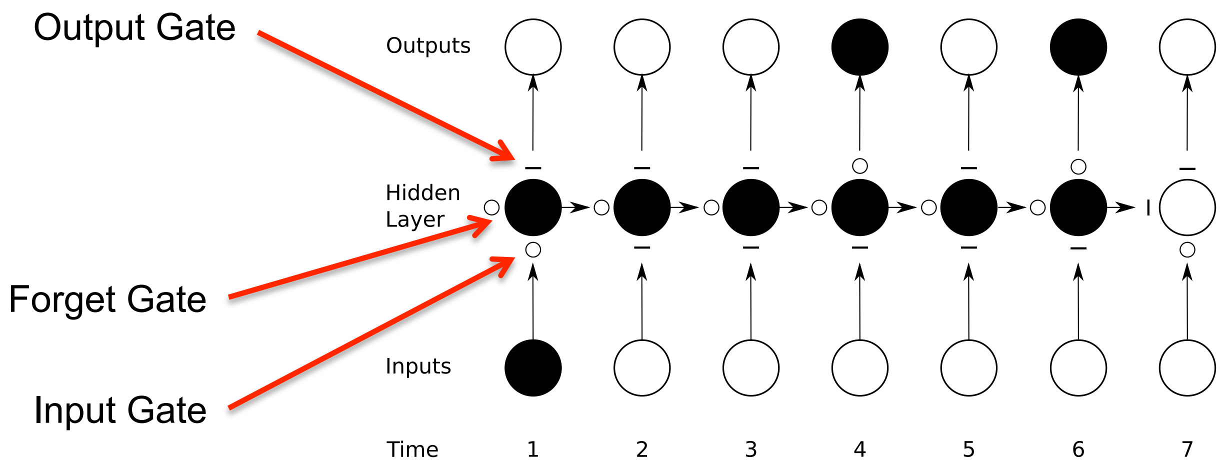
- Memory cells store temporal state of network
- Multiplicative gates control information flow
  - Input gate: controls flow of input activations into cell
  - Output gate: controls output flow of cell activations
  - Forget gate: Process continuous input streams
- These gates allow LSTM to store and access long-term information



LSTM memory blocks

Preserving Gradient Information with LSTMs

- Memory cell remembers first input as long as the forget gate is open and then input gate is closed



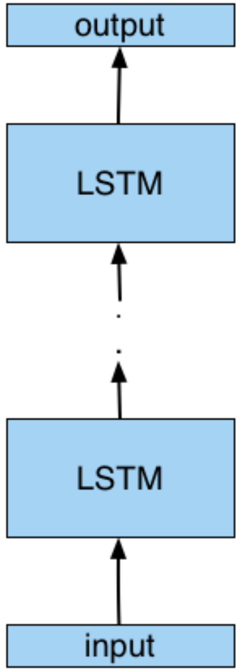
LSTM Results

[H. Sak et al, Interspeech 2014]

- Explore LSTMs on 2,000 Voice Search Task
- LSTM gives a 10% relative improvement over the DNN
- Optimization challenges:
    - LSTM is unrolled for 20 time steps
    - Performance seems to saturate after 2 LSTM layers

Model Training	WER-DNN	WER-LSTM
Cross-Entropy	11.1	**10.0**
Sequence	10.0	**8.9**

Number of Layers	WER, CE
1	11.3
2	**10.7**
3	10.7



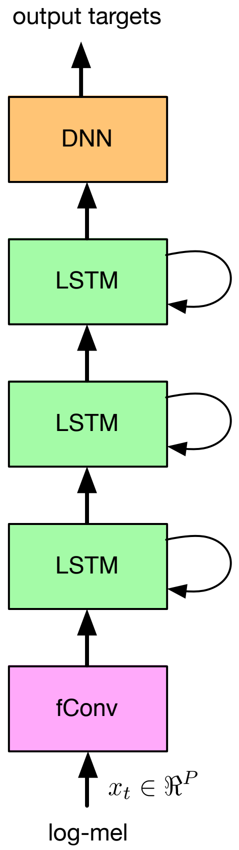
(3) CLDNN

[T.N. Sainath et al, ICASSP 2015]

- We combine convolutional neural networks, LSTMs and deep neural networks in a unified framework (CLDNN)
- Architecture uses 1 CNN, 3 LSTM and 1 DNN layer
- CLDNNs give an 8% relative improvement over LSTMs

Method	WER – Seq
LSTM	8.9
CLDNN	**8.2**

- Optimization Challenges:
    - Increasing number of layers saturates performance
    - Can only unroll LSTM for 20 time steps



Conclusions

- DNN performance
  - Pre-training strategies and GPUs
  - This encouraged deeper networks with more output targets
- Training improvements
  - Parallel GPU training via 1-bit GPU
  - Asynchronous SGD via CPUs
  - 2nd order Hessian-free via Blue Gene
- Architecture challenges
  - LSTMs and their variants are popular but still have optimization issues
  - Can only unroll for limited time steps
  - Can only make the architectures so deep

References – Neural Network Architectures

- G. Hinton, L. Deng, D. Yu, G. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury, “Deep Neural Networks for Acoustic Modeling in Speech Recognition,” *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82–97, 2012.
- T. N. Sainath, B. Kingsbury, G. Saon, H. Soltau, A. Mohamed, G. Dahl and B. Ramabhadran, "Deep Convolutional Neural Networks for Large-Scale Speech Tasks,” in Elsevier, Special Issue in Deep Learning, November 2014.
- H. Soltau, G. Saon and T.N. Sainath, “Joint Training of Convolutional and Non-Convolutional Neural Networks,” in Proc. ICASSP, 2014.
- H. Sak, A. Senior, and F. Beaufays, “Long Short-Term Memory Recurrent Neural Network Architectures for Large Scale Acoustic Modeling,” in *Proc. Interspeech*, 2014.
- T. N. Sainath, O. Vinyals, A. Senior and H. Sak, “Convolutional, Long Short-Term Memory, Fully Connected Deep Neural Networks ” in *Proc. ICASSP*, 2015.

References – Training Improvements

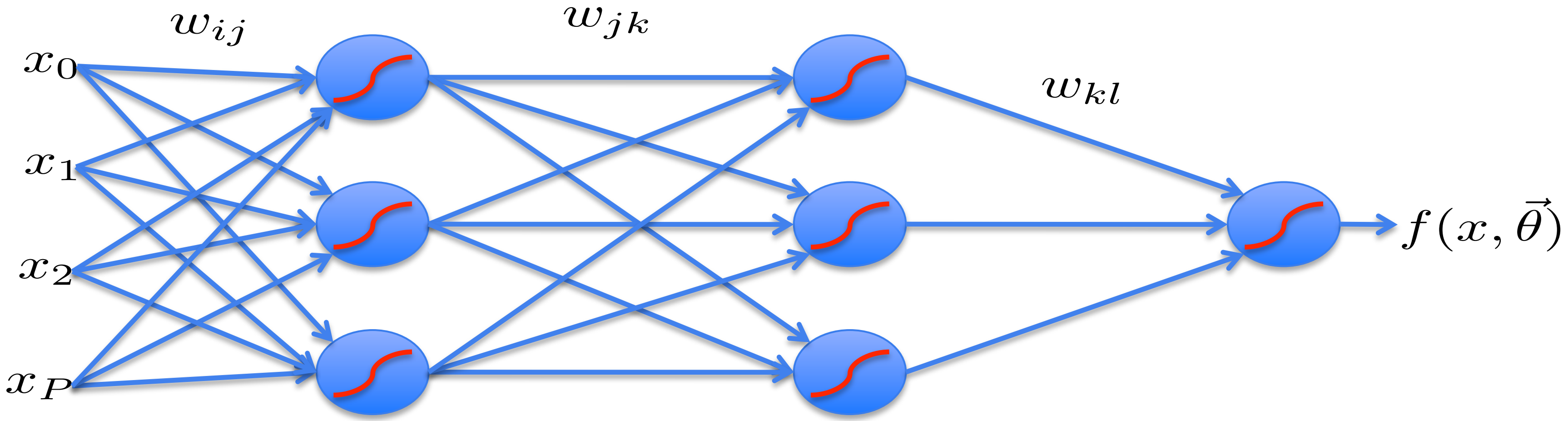
- Jeff Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc Le, Mark Mao, Marc'Aurelio Ranzato, Antrew Senior, Paul Tucker, Ke Yang, Andrew Ng Large Scale Distributed Deep Networks. NIPS 2012.
- B. Kingsbury, T. N. Sainath, and H. Soltau, "Scalable Minimum Bayes Risk Training of Deep Neural Network Acoustic Models Using Distributed Hessian-free Optimization," in Proc. Interspeech, 2012.
- G. Saon and H. Soltau, "A comparison of Two Optimization Techniques for Sequence Training of Deep Neural Networks," in Proc. ICASSP 2014.
- T. N. Sainath, I. Chung, B. Ramabhadran, M. Picheny, J. Gunnels, B. Kingsbury, G. Saon, V. Austel, U. Chaudhri, "Parllel Deep Neural Network Training for LVCSR using Blue Gene/Q," in Proc. Interspeech, September 2014.
- F. Seide, H. Fu, J. Droppo, G. Li and D. Yu, "1-Bit Stochastic Gradient Descent and Application to Data-Parallel Distributed Training of Speech DNNs," in Interspeech 2014.

Backup Slides

(1) Error Backpropagation

Introduce variables over the neural network

$$\vec{\theta} = \{w_{ij}, w_{jk}, w_{kl}\}$$



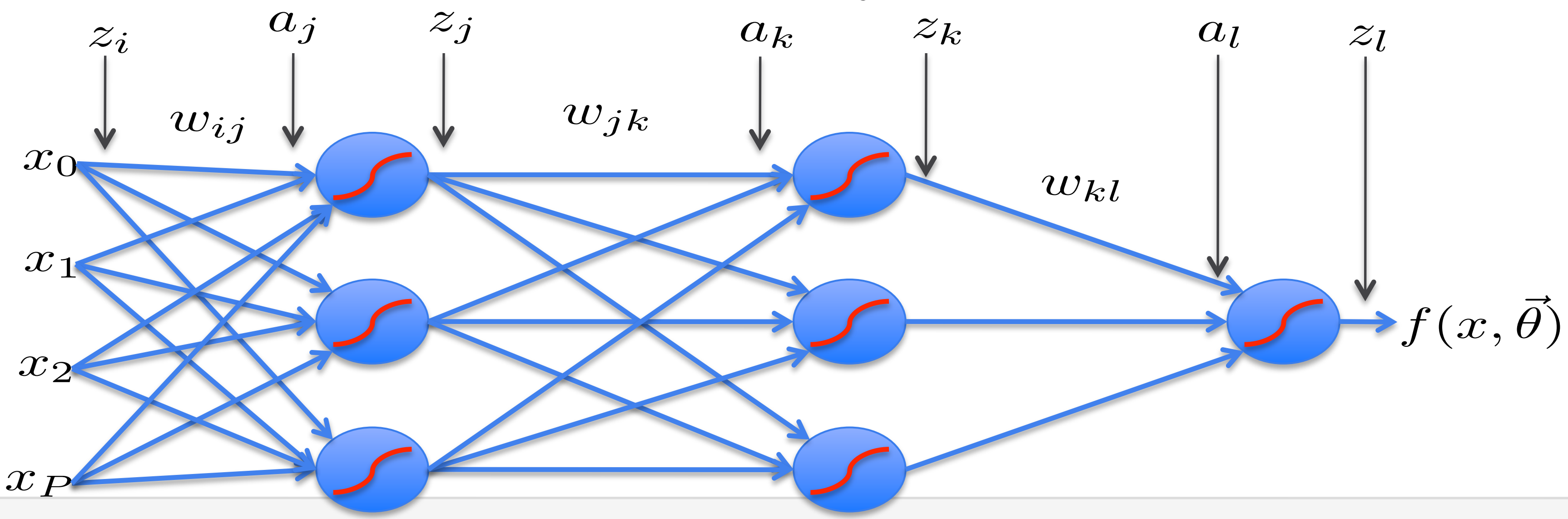
Error Backpropagation

$$\vec{\theta} = \{w_{ij}, w_{jk}, w_{kl}\}$$

Introduce variables over the neural network

Define *a* to be the input of each non-linearity

Define *z* to be the output of each non-linearity

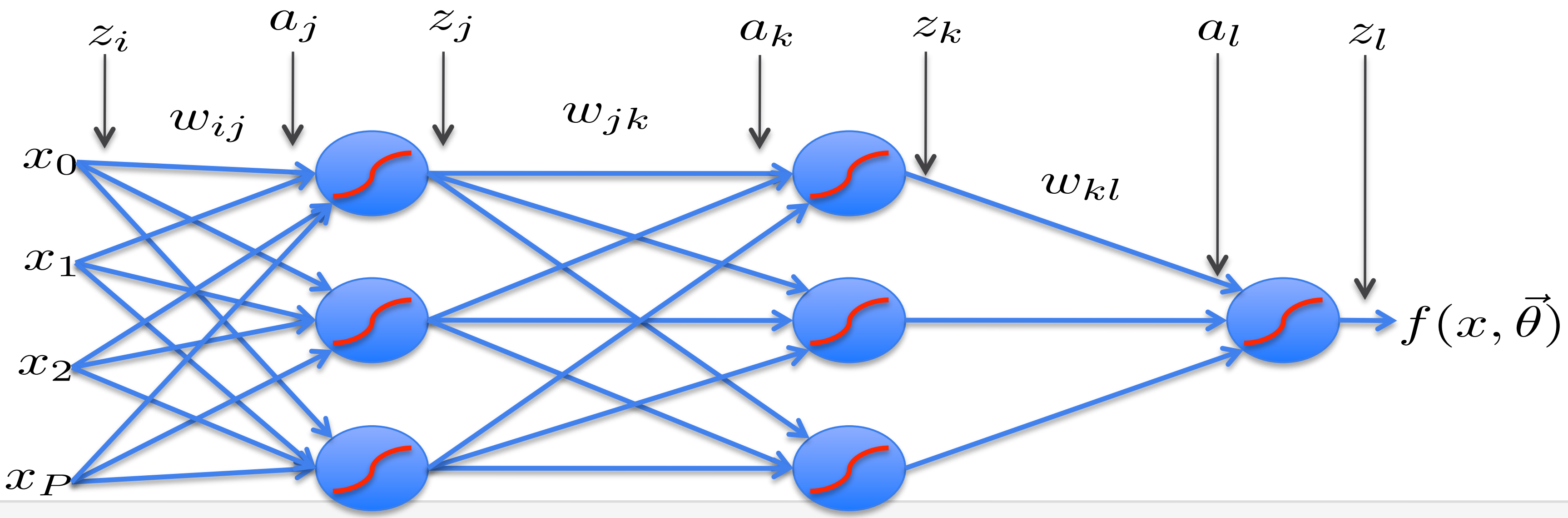


Error Backpropagation

$$\vec{\theta} = \{w_{ij}, w_{jk}, w_{kl}\}$$

$$a_j = \sum_i w_{ij} z_i \qquad a_k = \sum_j w_{jk} z_j \qquad a_l = \sum_k w_{kl} z_k$$

$$z_j = g(a_j) \qquad z_k = g(a_k) \qquad z_l = g(a_l)$$



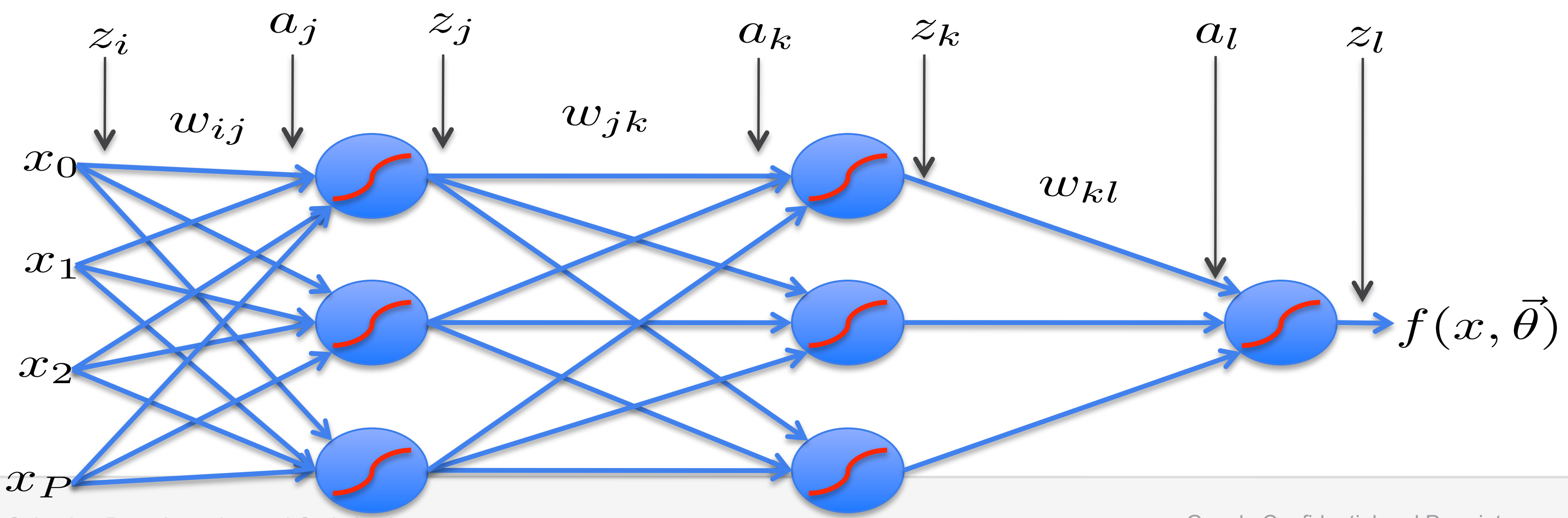
Error Backpropagation

$$\vec{\theta} = \{w_{ij}, w_{jk}, w_{kl}\}$$

Training: Take the gradient of the last component and iterate backwards

$$a_j = \sum_i w_{ij} z_i \qquad a_k = \sum_j w_{jk} z_j \qquad a_l = \sum_k w_{kl} z_k$$

$$z_j = g(a_j) \qquad z_k = g(a_k) \qquad z_l = g(a_l)$$



Error Backpropagation

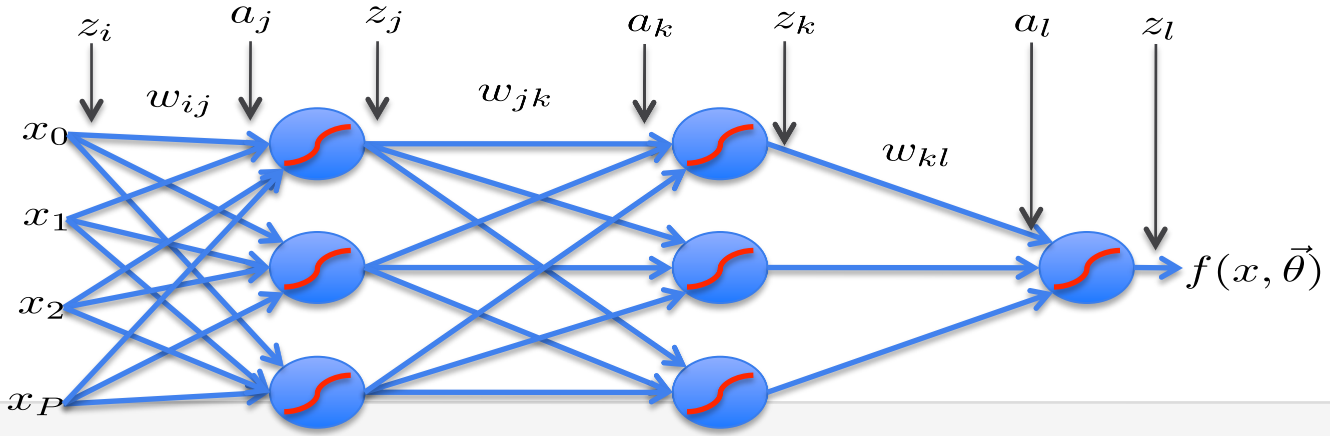
$$R(\theta) = \frac{1}{N}\sum_{n=1}^{N} L_n(y_n, f(x_n, \theta))$$

$$= \frac{1}{N}\sum_{n=1}^{N} \frac{1}{2}(y_n - f(x_n, \theta))^2$$

$$= \frac{1}{N}\sum_{n=1}^{N} \frac{1}{2}\left(y_n - g\left(\sum_k w_{k1}\, g\left(\sum_j w_{jk}\, g\left(\sum_i w_{ij} x_{n,i}\right)\right)\right)\right)^2$$

Empirical Risk Function

n is number of training points



Error Backpropagation

Optimize last layer weights w_{kl}

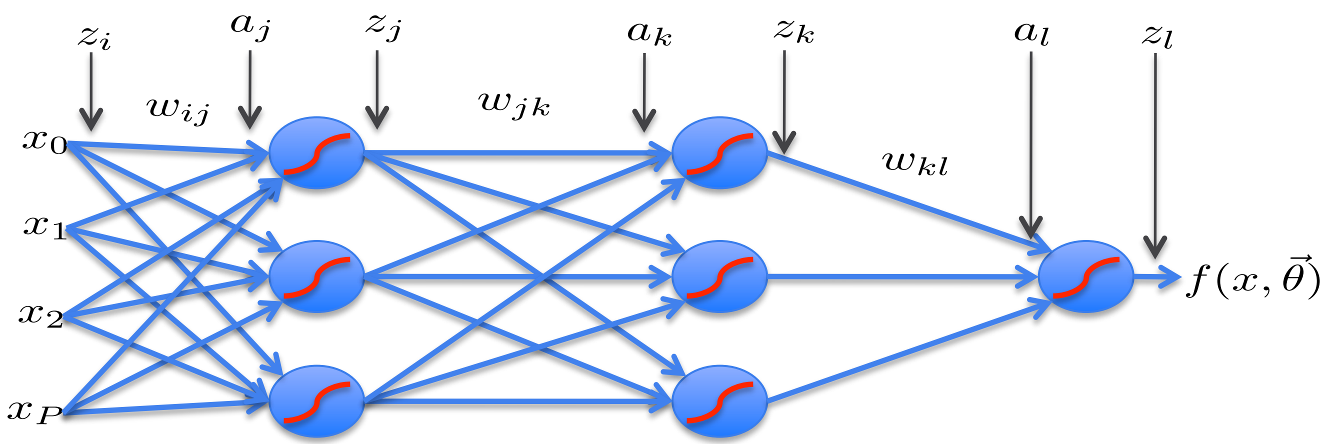
$$L_n = \frac{1}{2}\left(y_n - f(x_n)\right)^2$$

$$a_{l,n} = \sum_k w_{kl} z_k$$

$$f(x_n) = z_{l,n} = g(a_{l,n})$$

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N} \sum_n \left[\frac{\partial L_n}{\partial a_{l,n}}\right] \left[\frac{\partial a_{l,n}}{\partial w_{kl}}\right]$$

Calculus chain rule



Error Backpropagation

Optimize last layer weights w_{kl}

$$L_n = \frac{1}{2}\left(y_n - f(x_n)\right)^2$$

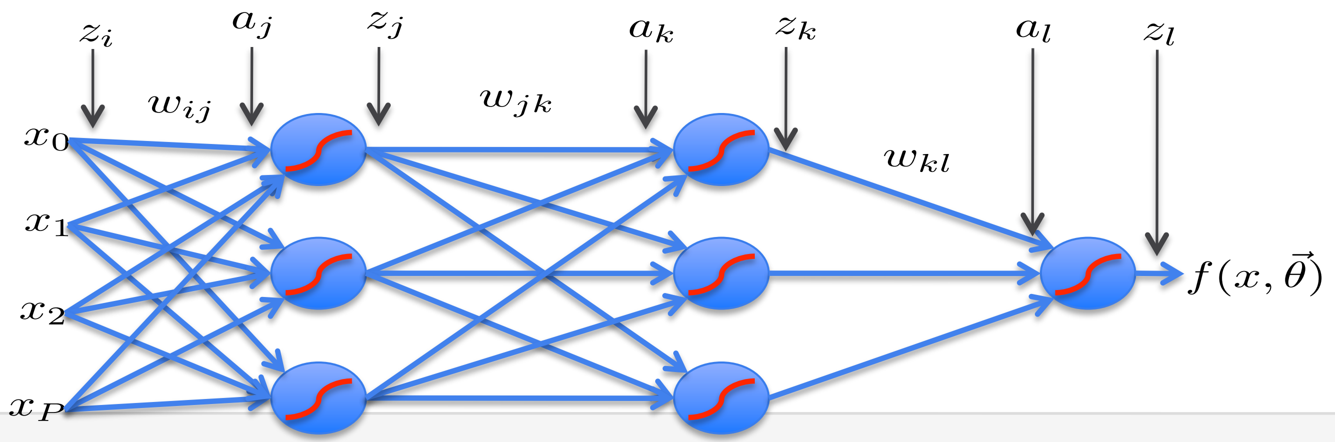
$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{l,n}}\right]\left[\frac{\partial a_{l,n}}{\partial w_{kl}}\right]$$

Calculus chain rule

$$a_{l,n} = \sum_k w_{kl} z_k$$

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial \frac{1}{2}(y_n - g(a_{l,n}))^2}{\partial a_{l,n}}\right]\left[\frac{\partial a_{l,n}}{\partial w_{kl}}\right]$$

$$f(x_n) = z_{l,n} = g(a_{1,n})$$



Error Backpropagation

Optimize last layer weights w_{kl}

$$L_n = \frac{1}{2}\left(y_n - f(x_n)\right)^2$$

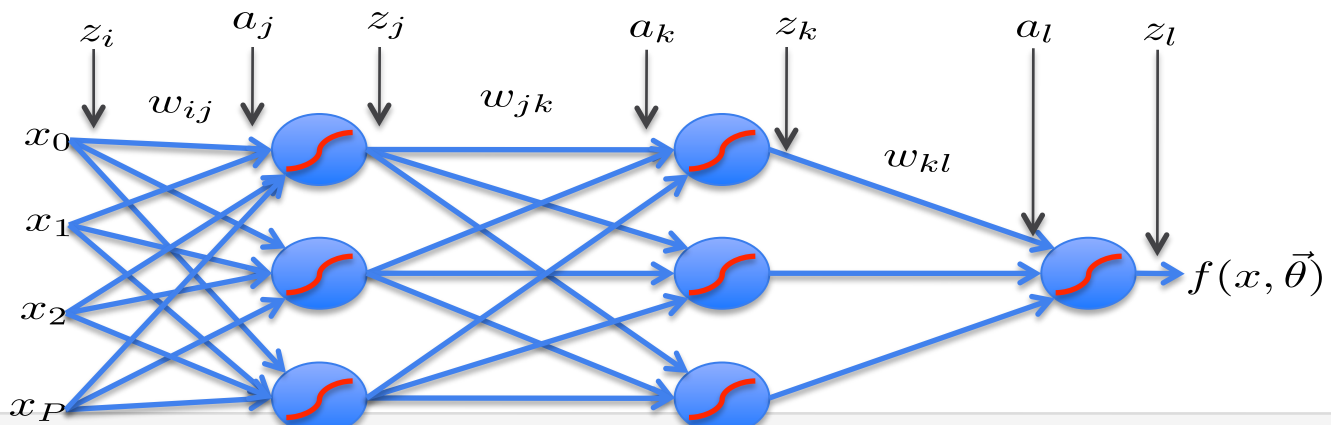
$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{l,n}}\right]\left[\frac{\partial a_{l,n}}{\partial w_{kl}}\right]$$

Calculus chain rule

$$a_{l,n} = \sum_k w_{kl} z_k$$

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial \frac{1}{2}(y_n - g(a_{l,n}))^2}{\partial a_{l,n}}\right]\left[\frac{\partial z_{k,n} w_{kl}}{\partial w_{kl}}\right]$$

$$f(x_n) = z_{l,n} = g(a_{1,n})$$



Error Backpropagation

Optimize last layer weights w_{kl}

$$L_n = \frac{1}{2}\left(y_n - f(x_n)\right)^2$$

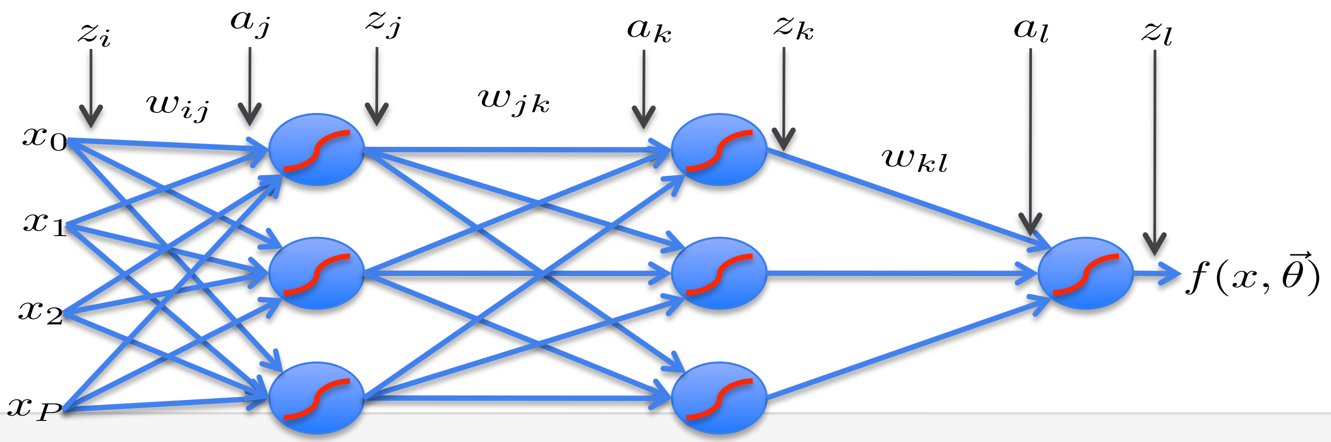
$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{l,n}}\right]\left[\frac{\partial a_{l,n}}{\partial w_{kl}}\right]$$

Calculus chain rule

$$a_{l,n} = \sum_k w_{kl} z_k$$

$$f(x_n) = z_{l,n} = g(a_{1,n})$$

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial \frac{1}{2}(y_n - g(a_{l,n}))^2}{\partial a_{l,n}}\right]\left[\frac{\partial z_{k,n} w_{kl}}{\partial w_{kl}}\right] = \frac{1}{N}\sum_n \left[-(y_n - z_{l,n})g'(a_{l,n})\right] z_{k,n}$$



Error Backpropagation

Optimize last layer weights w_{kl}

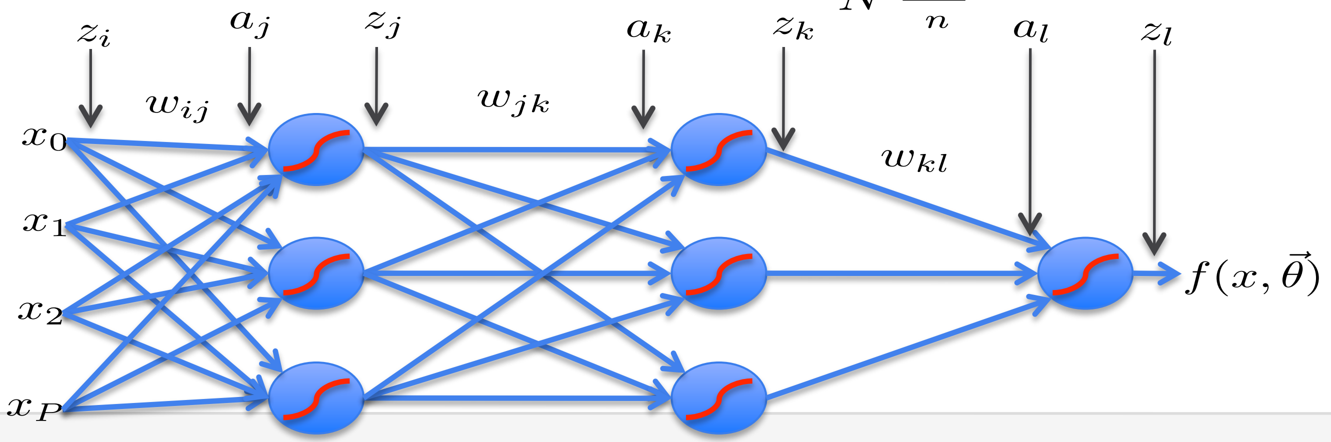
$$L_n = \frac{1}{2}\left(y_n - f(x_n)\right)^2$$

Calculus chain rule

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{l,n}}\right]\left[\frac{\partial a_{l,n}}{\partial w_{kl}}\right]$$

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial \frac{1}{2}(y_n - g(a_{l,n}))^2}{\partial a_{l,n}}\right]\left[\frac{\partial z_{k,n} w_{kl}}{\partial w_{kl}}\right] = \frac{1}{N}\sum_n \boxed{\left[-(y_n - z_{l,n})g'(a_{l,n})\right]} z_{k,n}$$

$$= \frac{1}{N}\sum_n \delta_{l,n} \quad z_{k,n}$$



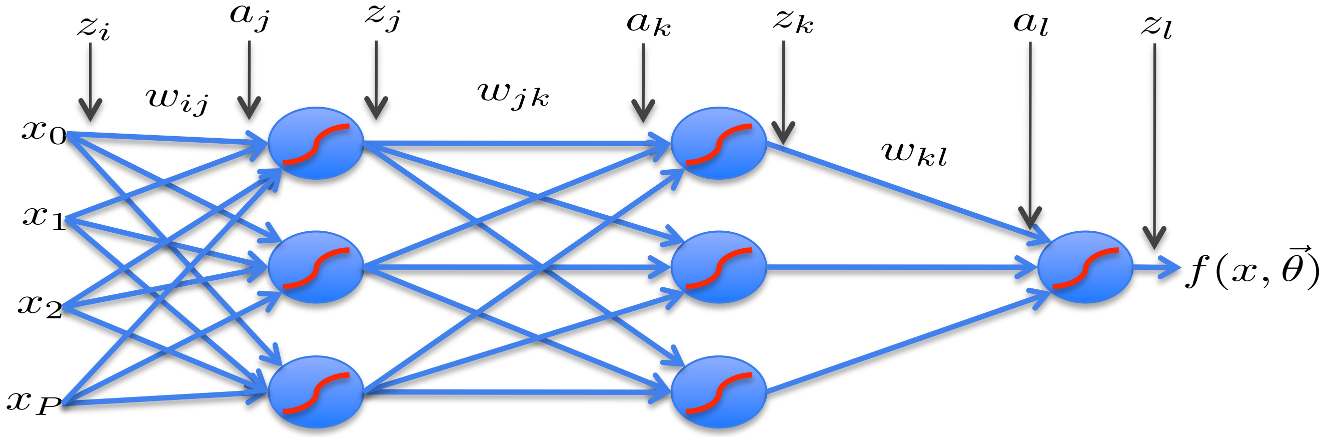
Error Backpropagation

Optimize last hidden weights w$_{jk}$

$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{k,n}}\right]\left[\frac{\partial a_{k,n}}{\partial w_{jk}}\right]$$

$$L_n = \frac{1}{2}\left(y_n - f(x_n)\right)^2$$

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \delta_{l,n} z_{k,n}$$



Error Backpropagation

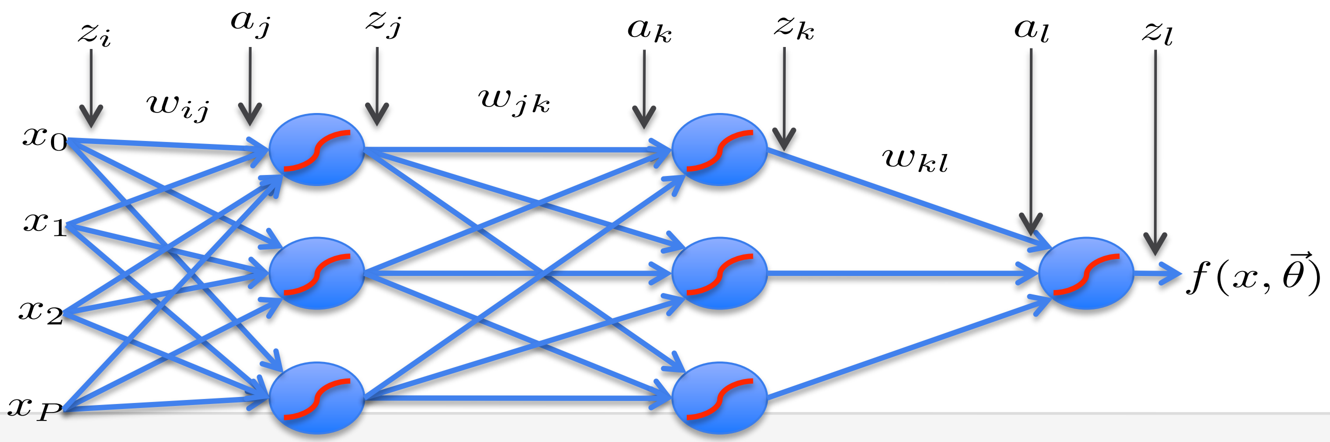
Optimize last hidden weights w$_{jk}$

$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n \left[\sum_l \frac{\partial L_n}{\partial a_{l,n}} \frac{\partial a_{l,n}}{\partial a_{k,n}}\right] \left[\frac{\partial a_{k,n}}{\partial w_{jk}}\right]$$

$$L_n = \frac{1}{2}\left(y_n - f(x_n)\right)^2$$

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \delta_{l,n} z_{k,n}$$

Multivariate chain rule



Error Backpropagation

Optimize last hidden weights w_{jk}

$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n \left[\sum_l \frac{\partial L_n}{\partial a_{l,n}} \frac{\partial a_{l,n}}{\partial a_{k,n}}\right] \left[\frac{\partial a_{k,n}}{\partial w_{jk}}\right]$$

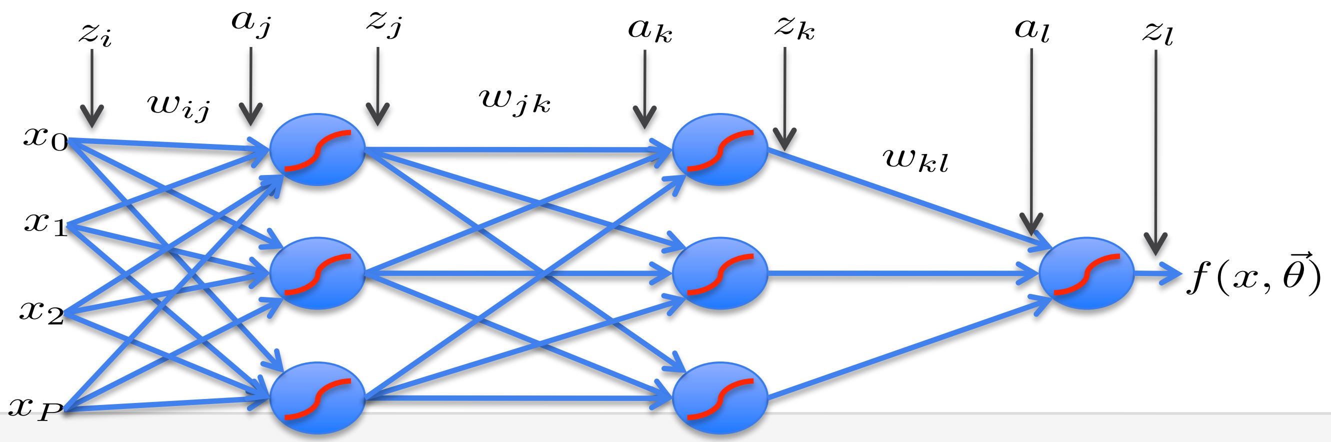
$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n \left[\sum_l \delta_{l,n} \frac{\partial a_{l,n}}{\partial a_{k,n}}\right] \left[z_{j,n}\right]$$

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \delta_{l,n} z_{k,n}$$

Multivariate chain rule

$$\frac{\partial L_n}{\partial a_{l,n}} = \delta_{l,n}$$

$$a_{k,n} = \sum_j w_{jk} z_{j,n}$$



Google

Error Backpropagation

Optimize last hidden weights w_{jk}

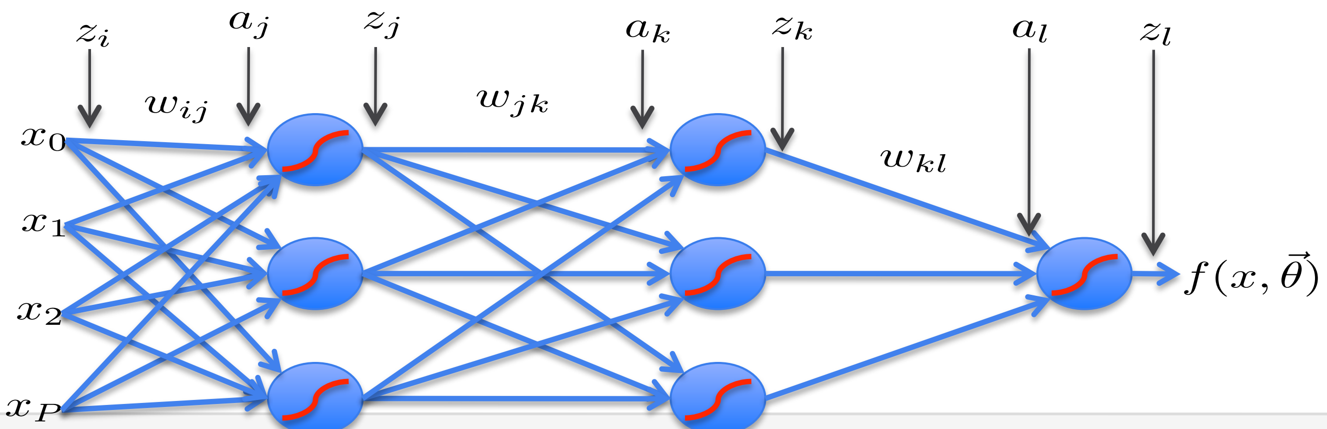
$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \delta_{l,n} z_{k,n}$$

$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n \left[\sum_l \frac{\partial L_n}{\partial a_{l,n}} \frac{\partial a_{l,n}}{\partial a_{k,n}}\right] \left[\frac{\partial a_{k,n}}{\partial w_{jk}}\right]$$

Multivariate chain rule

$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n \left[\sum_l \delta_{l,n} \frac{\partial a_{l,n}}{\partial a_{k,n}}\right]\left[z_{j,n}\right]$$

$$a_{l,n} = \sum_k w_{kl} g(a_{k,n})$$



Error Backpropagation

Optimize last hidden weights w_{jk}

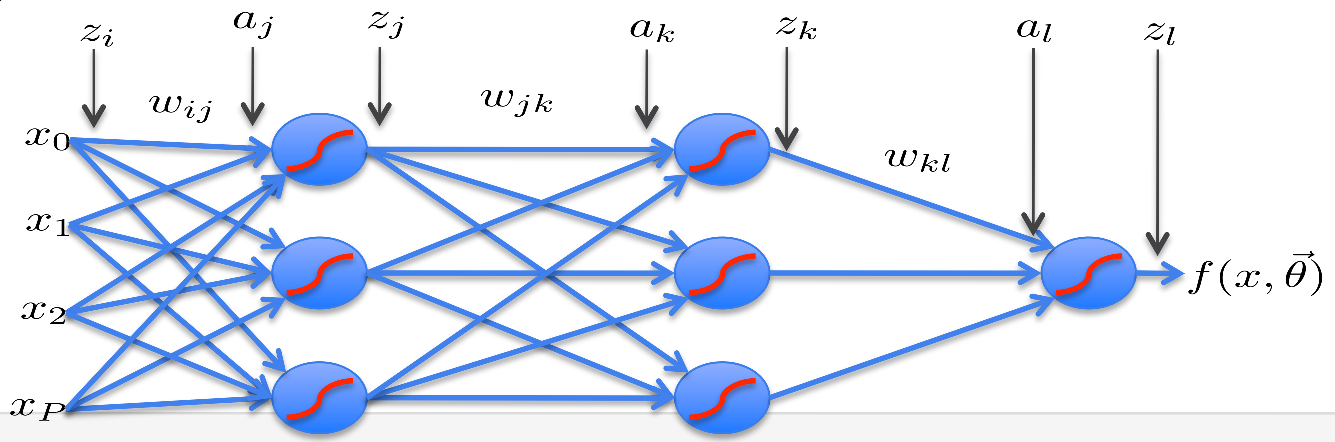
$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \delta_{l,n} z_{k,n}$$

$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n \left[\sum_l \frac{\partial L_n}{\partial a_{l,n}} \frac{\partial a_{l,n}}{\partial a_{k,n}}\right]\left[\frac{\partial a_{k,n}}{\partial w_{jk}}\right]$$

Multivariate chain rule

$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n \left[\boxed{\sum_l \delta_l w_{kl} g'(a_{k,n})}\right][z_{j,n}] = \frac{1}{N}\sum_n [\delta_{k,n}][z_{j,n}]$$

$$a_{l,n} = \sum_k w_{kl} g(a_{k,n})$$



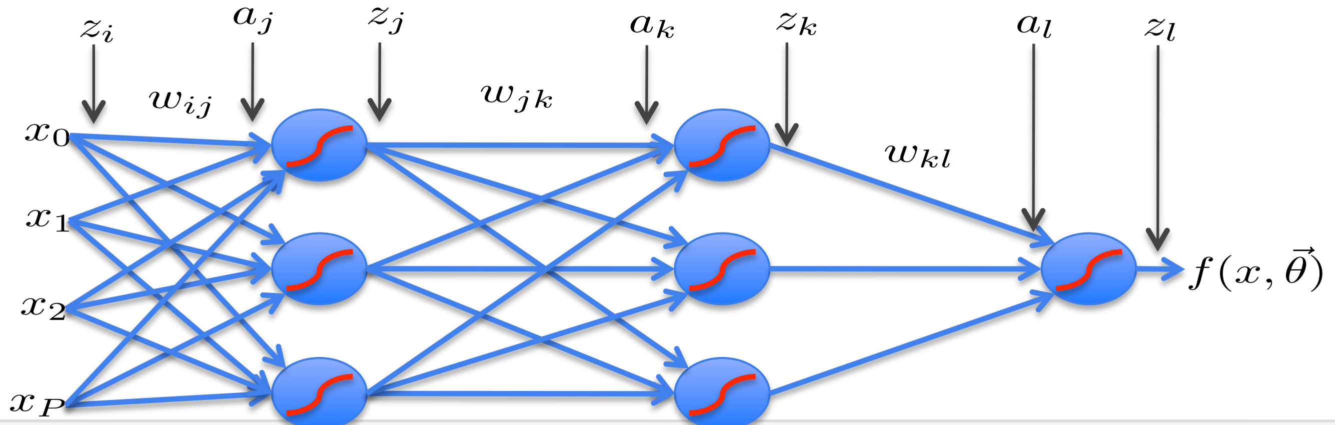
Error Backpropagation

Repeat for all previous layers

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{l,n}}\right]\left[\frac{\partial a_{l,n}}{\partial w_{kl}}\right] = \frac{1}{N}\sum_n \left[-(y_n - z_{l,n})g'(a_{l,n})\right] z_{k,n} = \frac{1}{N}\sum_n \delta_{l,n} z_{k,n}$$

$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{k,n}}\right]\left[\frac{\partial a_{k,n}}{\partial w_{jk}}\right] = \frac{1}{N}\sum_n \left[\sum_l \delta_{l,n} w_{kl} g'(a_{k,n})\right] z_{j,n} = \frac{1}{N}\sum_n \delta_{k,n} z_{j,n}$$

$$\frac{\partial R}{\partial w_{ij}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{j,n}}\right]\left[\frac{\partial a_{j,n}}{\partial w_{ij}}\right] = \frac{1}{N}\sum_n \left[\sum_k \delta_{k,n} w_{jk} g'(a_{j,n})\right] z_{i,n} = \frac{1}{N}\sum_n \delta_{j,n} z_{i,n}$$



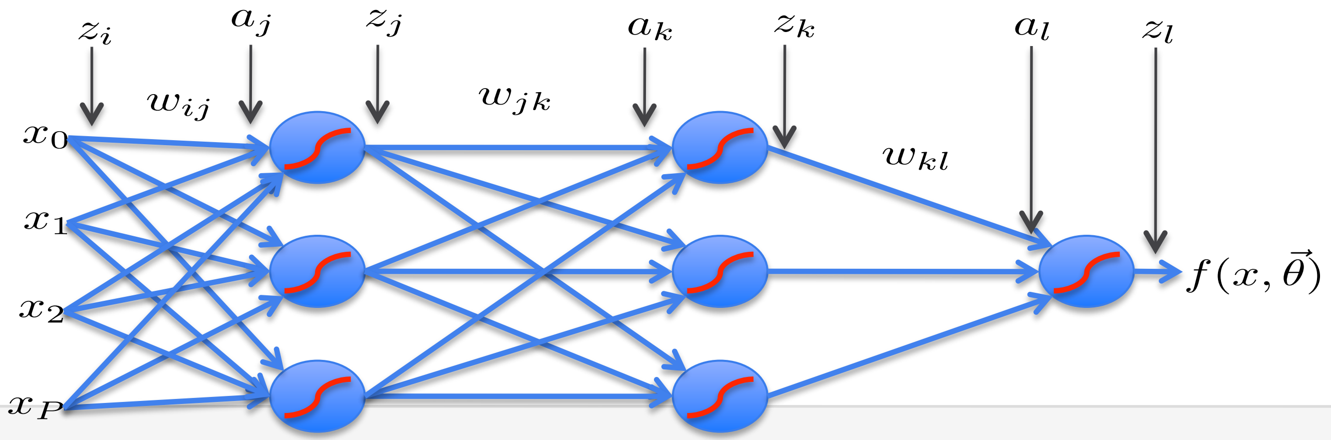
Error Backpropagation

Now that we have well defined gradients for each parameter, update using Gradient Descent

$$w_{ij}^{t+1} = w_{ij}^{t} - \eta \frac{\partial R}{w_{ij}}$$

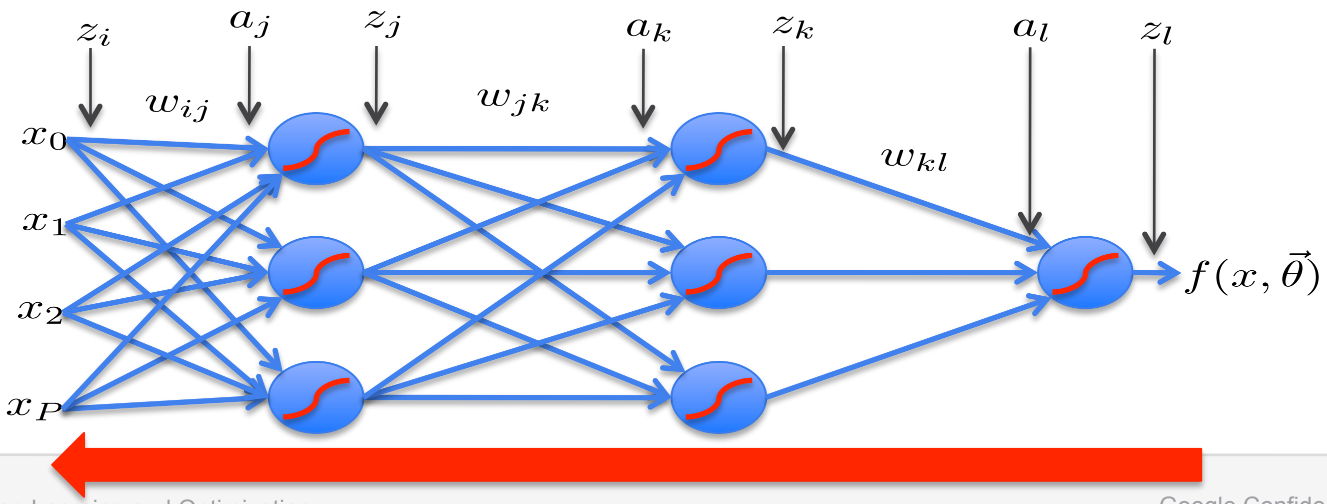
$$w_{jk}^{t+1} = w_{jk}^{t} - \eta \frac{\partial R}{w_{kl}}$$

$$w_{kl}^{t+1} = w_{kl}^{t} - \eta \frac{\partial R}{w_{kl}}$$



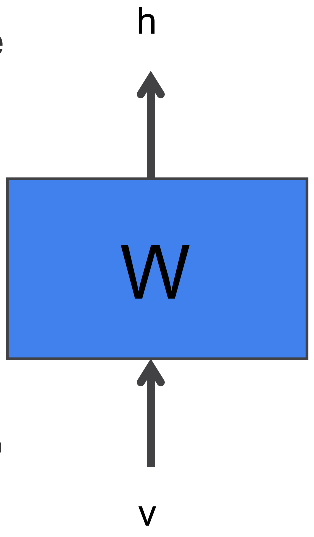
Error Back-propagation

Error backprop unravels the multivariate chain rule and solves the gradient for each layer separately.
The error δ is backpropagated along the network from one layer to the next



(2) Pre-Training via Unsupervised Learning

- The goal of unsupervised learning is to put the weights in a good initial space to encourage deeper and larger networks during fine-tuning
- Unsupervised learning systems can be designed using the encoder-decoder paradigm
    - Encoder: transform input *v* into code representation *h*
    - Decoder: reconstructs input from the code by minimizing reconstruction error
- Encoder-decoder paradigm learns weights such that the code captures higher-order relevant information from input signal, these weights are used to initialize network for fine-tuning
- Unsupervised learning
    - Restricted Boltzmann Machine (RBM) [Hinton – Toronto]
    - Sparse Encoding Symmetric Machine (SESM) [Lecun – NYU]
    - De-noising Auto-Encoder [Bengio – Montreal]



Restricted Boltzmann Machines

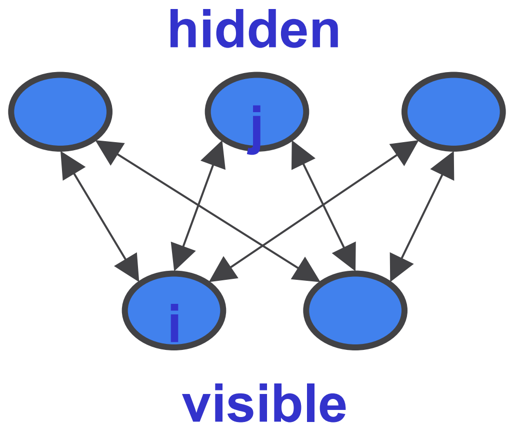
Consider a one layer RBM

Weights are fully connected between hidden and visible units

No connections between hidden units

Hidden units are conditionally independent given the visible states.

Relationship between *v* and *h* for Bernoulli-Bernoulli RBMs given as:



$$p(h_j = 1 | \mathbf{v}) = 1/1 + \exp(-(\sum_i v_i w_{ij} + b_j)$$

$$p(v_i = 1 | \mathbf{h}) = 1/1 + \exp(-\sum_j h_j w_{ij} + b_i))$$

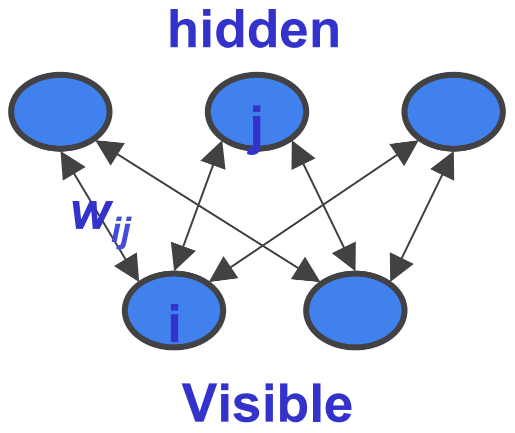
The Energy of Joint Configuration

- Each possible joint configuration of the visible and hidden units (ignoring biases) has an energy

$$E(v,h) = -\sum_{i,j} v_i h_j w_{ij}$$

- The energy of a joint configuration of the visible and hidden units determines its probability:

$$p(v,h) \propto e^{-E(v,h)}$$



Using energies to define probabilities

The probability of a joint configuration over both *v* and *h* depends on the energy *E* of that joint configuration compared with the energy of all other joint configurations.

$$p(v,h) = \frac{e^{-E(v,h)}}{\sum_{u,g} e^{-E(u,g)}}$$

partition function

The probability of a configuration of the visible units is the sum of the probabilities of all the joint configurations that contain it.

$$p(v) = \sum_h p(v,h) = \frac{\sum_h e^{-E(v,h)}}{\sum_{u,g} e^{-E(u,g)}}$$

How to maximize *p(v)*

Goal of supervised fine-tuning is to maximize log *p(class|v)*
It follows that the goal of unsupervised learning is to maximize *log p(v)*

$$w := w - \varepsilon \frac{\partial \log p(v)}{\partial w}$$

Define free-energy as

$$F(v) = -\log \sum_h e^{-E(v,h)}$$

$$E(v,h) = -\sum_{i,j} v_i h_j w_{ij}$$

$$\boxed{p(v) = \frac{\sum_h e^{-E(v,h)}}{\sum_{u,g} e^{-E(u,g)}}}$$

Gradient given as

$$\boxed{-\frac{\partial \log p(v)}{\partial w} = \frac{\partial F(v)}{\partial w} - \sum_{\tilde{v}} p(\tilde{v}) \frac{\partial F(\tilde{v})}{\partial w}}$$

Positive phase term easy to compute directly

Negative phase term difficult to analytically compute

Computing Derivative of Negative Phase

$$\boxed{-\frac{\partial \log p(v)}{\partial w} = \frac{\partial F(v)}{\partial w} - \sum_{\tilde{v}} p(\tilde{v}) \frac{\partial F(\tilde{v})}{\partial w}}$$

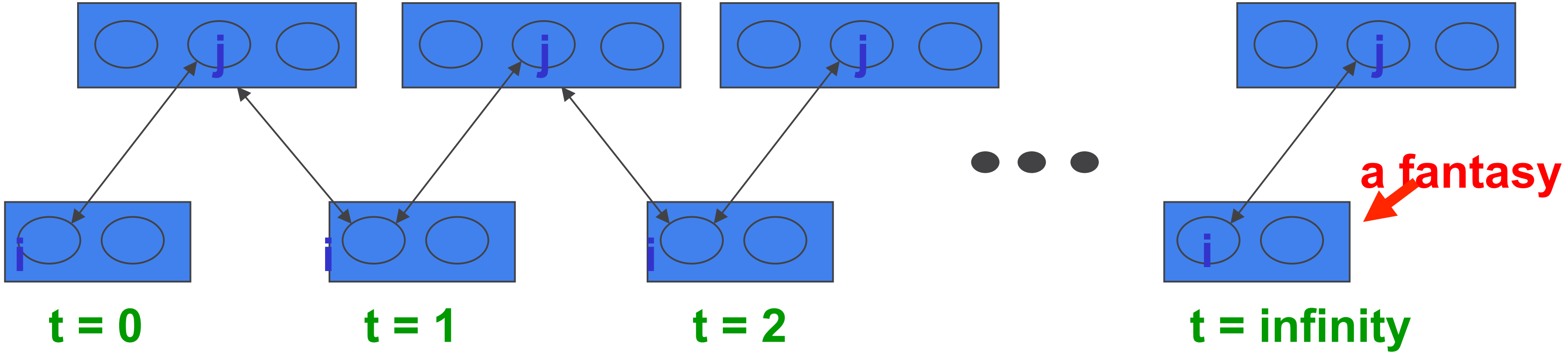
Negative phase term can be represented as $E_p\left[\frac{\partial F(v)}{\partial w}\right]$, the expectation over all possible configurations of the input (under the distribution formed by the model), which is difficult to estimate analytically

Estimate the expectation using a fixed number of model samples

$$-\frac{\partial \log p(v)}{\partial w} \approx \frac{\partial F(v)}{\partial w} - \frac{1}{|N|} \sum_{\tilde{v} \in N} \frac{\partial F(\tilde{v})}{\partial w}$$

Obtain samples of *p(v)* using Gibbs sampling

Sampling in an RBM



- In the RBM structure, *v* and *h* are conditionally independent
- To obtain samples of p(v,h):

$$p(h_j = 1 \mid \mathbf{v}) = 1/1 + \exp(-(\sum_i v_i w_{ij} + b_j)$$

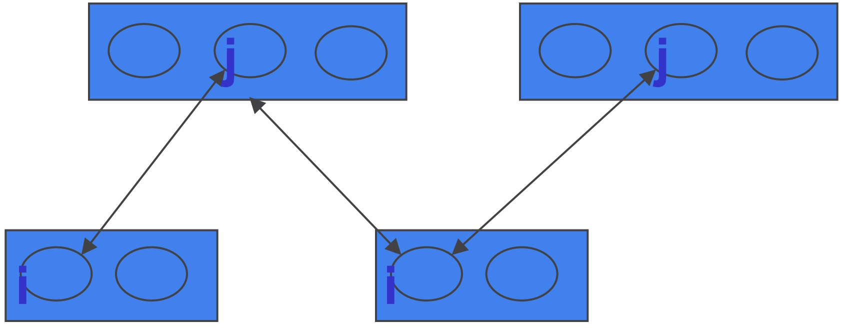
$$p(v_i = 1 \mid \mathbf{h}) = 1/1 + \exp(-\sum_j h_j w_{ij} + b_i))$$

Start with a training vector on the visible units.

Sample hidden units given fixed visible units

Then sample visible units given fixed hidden units

Contrastive Divergence



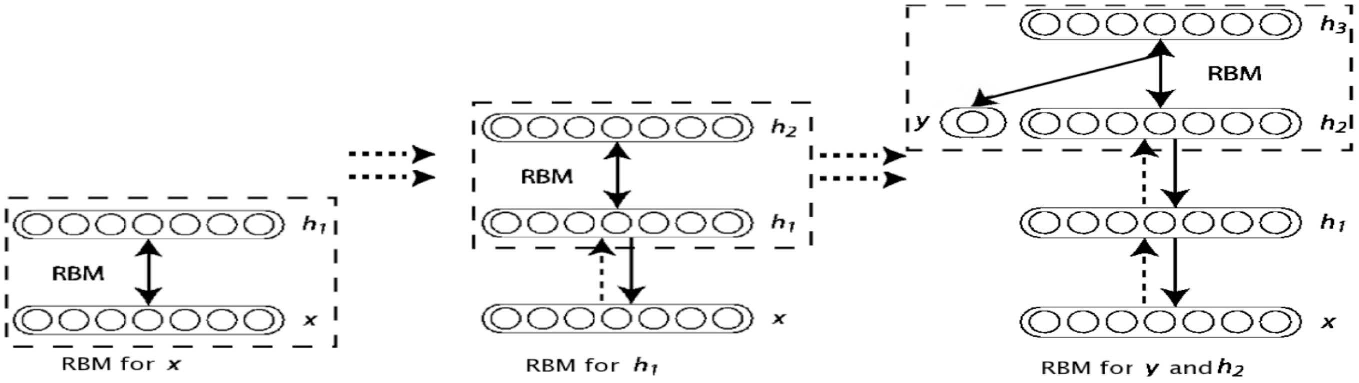
t = 0
data

t = 1
reconstruction

1. Start with a training vector on the visible units.
2. Update all the hidden units in parallel
3. Update the all the visible units in parallel to get a "reconstruction".
4. Update the hidden units again.

With contrastive divergence, just one step of Gibbs sampling is run
While this approximates -log p(v), it seems to work well in practice (Carreira-Perpinan & Hinton, 2005).

Constructing Deep Belief Networks



First train a layer of features that receive input directly from the speech features
Then treat the activations of the trained features as if they were speech features and learn features of features in a second hidden layer.
Why greedy?

It can be proved that each time we add another layer of features we improve a variational lower bound on the log probability of the training data.

Simplicity of training