Lectures 9-10: Multiple Layer Perceptrons and Deep Network Learning

Prof. Krishna R. Pattipati
Dept. of Electrical and Computer Engineering
University of Connecticut
Contact: krishna@engr.uconn.edu **(860) 486-2890**

Fall 2018
November 12, 2018

Reading List

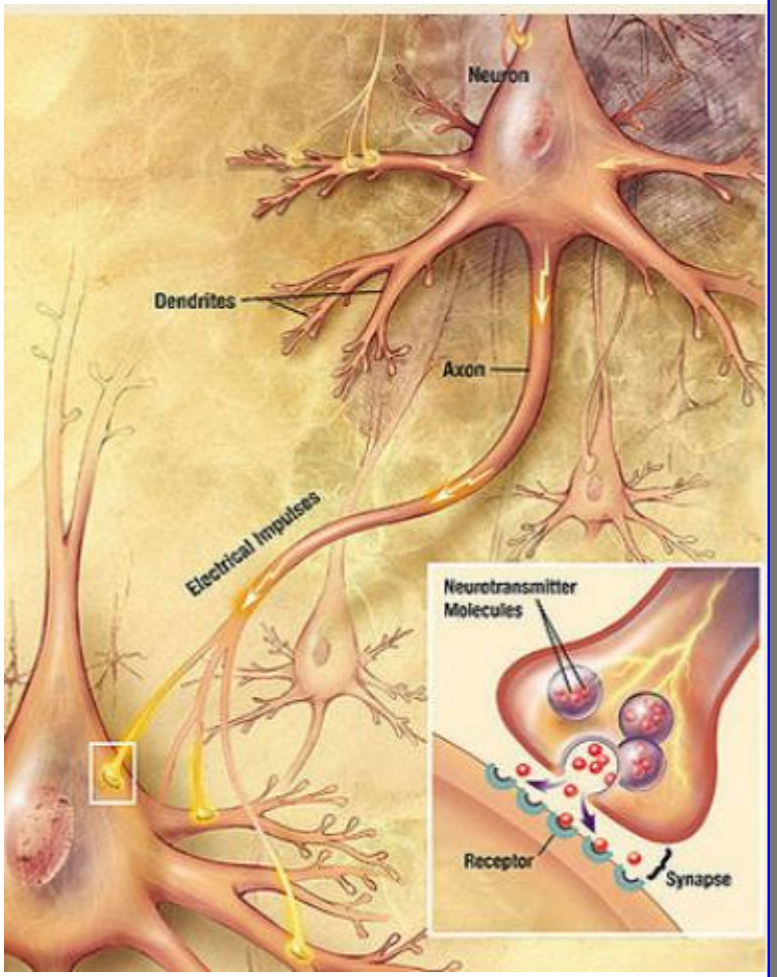
- Duda, Hart and Stork, Chapter 6
- Murphy, Chapters 13 and 16
- Bishop, Chapter 5
- Theodoridis, Chapter 18
- Recent Papers on Deep Neural Networks
- T.J. Sejnowski, Deep Learning Revolution, MIT Press, 2018.
- Cited Papers on Deep Neural Networks
- Lectures 5 and 9 of Fei-Fei Li, Justin Johnson and Serena Yeung http://cs231n.stanford.edu/syllabus.html
- 2016 lectures: CS 231: Convolutional Neural Networks for Visual Recognition by Fei-Fei Li, Andrej Karpathy and Justin Johnson

Lecture Outline

- Multiple Layer Perceptrons (MLPs)
- MLPs as Universal Approximators
- Back Propagation Algorithm
- Network Pruning
- Semi-supervised Learning of MLPs
- Introducing Deep Networks
  - ReLU, Batch normalization, Dropout, Convolution, Max pooling, GPUs, Stochastic Optimization
- Recurrent Neural Networks, LSTMs, GRUs, Transformer, 2D Convolution Networks
- Summary

History of NN - 1

- One can trace the history of Neural networks to the work of Santiago Ramon y Cajal, who discovered that the basic building element of the brain is the neuron. The brain comprises approximately 60-100 billion neurons!
- Each neuron is connected with other neurons via elementary structural and functional units/links, known as **synapses**. It is estimated that there are 50-100 trillion synapses. These links mediate information between connected neurons.
- The most common type of synapses are the chemical ones, which convert electrical pulses, produced by a neuron, to a chemical signal and then back to an electrical one. Depending on the input pulse(s), a synapse is either activated or inhibited. Via these links, each neuron is connected to other neurons and this happens in a hierarchical way, in a layer-wise fashion.



History of NN - 2

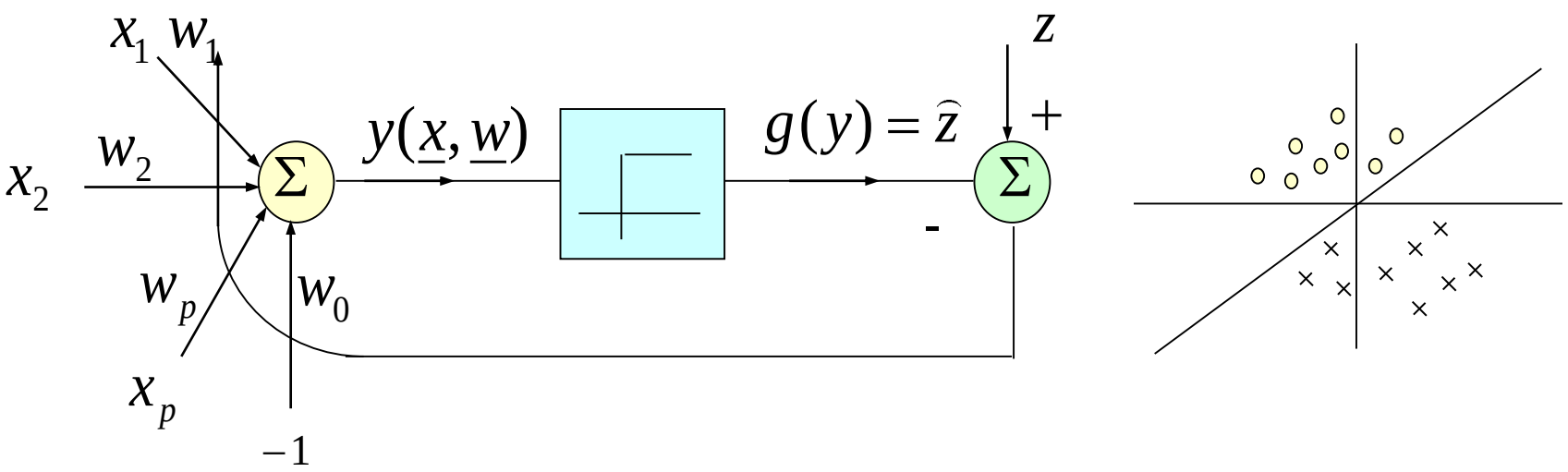
- In 1943, Warren McCulloch and Walter Pitts, developed a computational model for the basic neuron linking neurophysiology with mathematical logic. They showed that given a sufficient number of neurons and adjusting appropriately the synaptic links, each one represented by a weight, one can compute, in principle, any computable function. As a matter of fact, it is generally accepted that this is the paper that gave birth to the fields of neural networks and artificial intelligence.
- Frank Rosenblatt borrowed the idea of a neuron model, as suggested by McCulloch and Pitts, and proposed a true learning machine, which **learns** from a set of training data. In its most basic version, he used a single neuron and adopted a rule that can learn to separate data, which belong to two linearly separable classes. That is, he built a Pattern Recognition system. He called the basic neuron a perceptron and developed a rule/algorithm, the perceptron algorithm, which we discussed earlier and review it briefly.

Perceptron

- Let us review what we have learnt so far

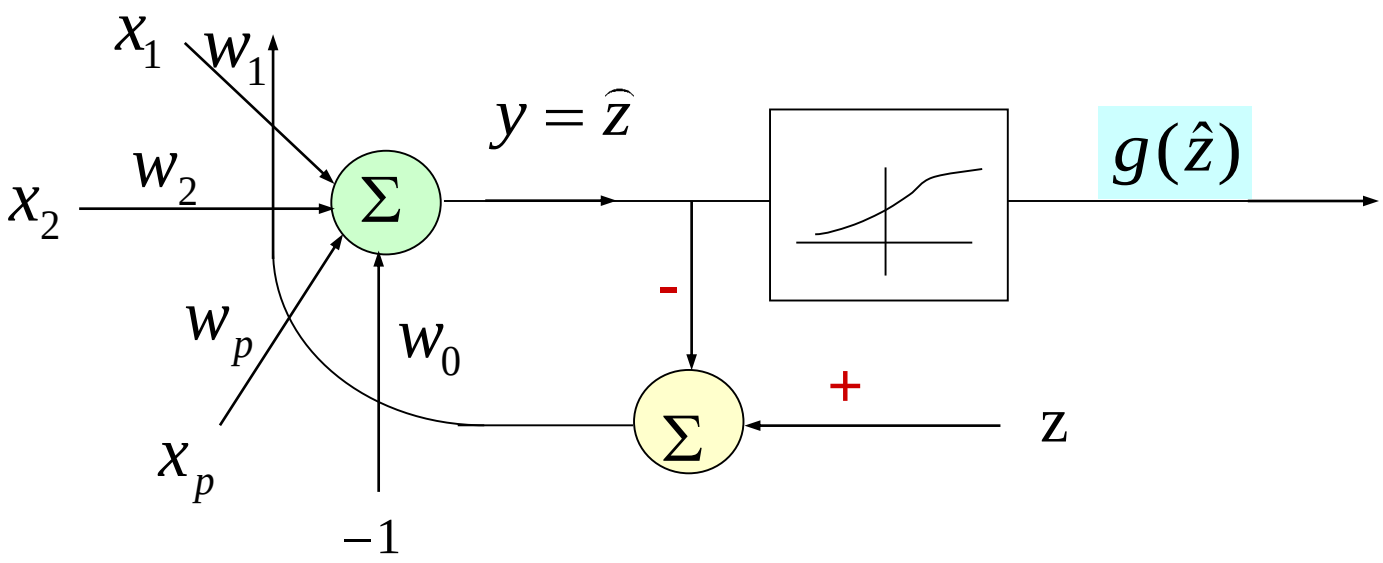
  -- ***Rosenblatt's Peceptron***

  - Can categorize linearly separable classes



Function Approximation

A Single neuron can approximate a function (LMS)



Function Approximation/Classification

- Function can be binary for classification, but need not be.
- Training can be accomplished by sequential steepest descent ("LMS"), Incremental Gauss-Newton, SVM, and optimization techniques (Conjugate Gradient, Quasi Newton, Newton, . . .)
- Logistic $g(y) = \frac{1}{1+e^{-y}}$ comes naturally from Gaussian binary classification problems. **Softmax approximation in C-class case**
- Also $g' = g(1-g)$

XOR Problem

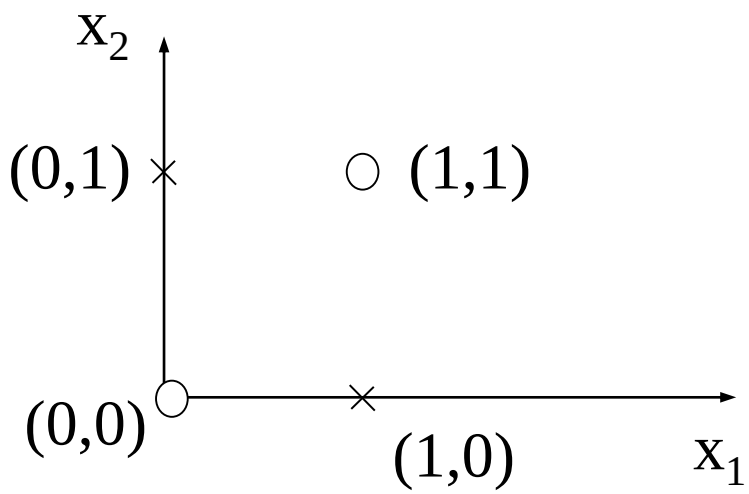
- Are we done?

  No. Consider the so-called "XOR problem" (parity problem, Exclusive OR), $t = x_1 \oplus x_2$

x_1	x_2	t
0	0	0
0	1	1
1	0	1
1	1	0

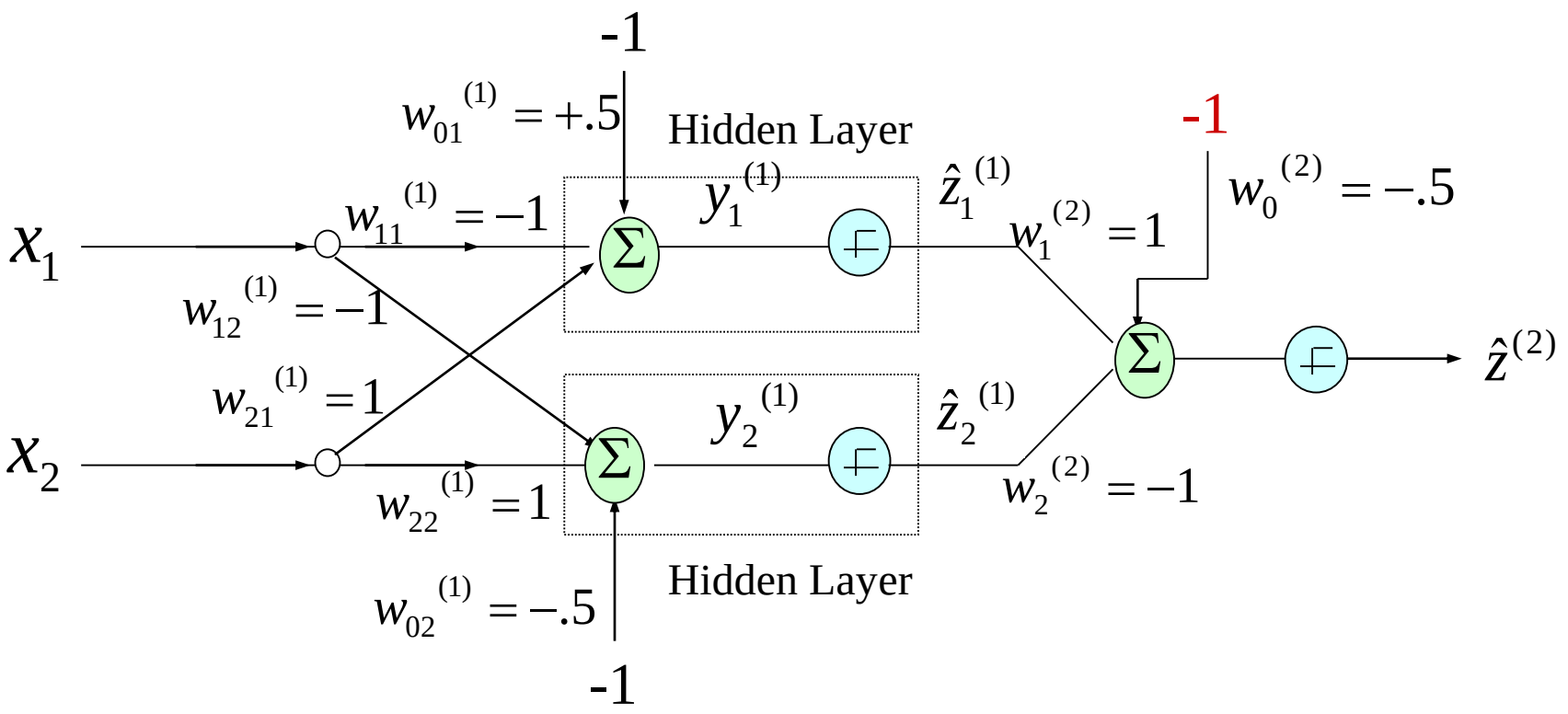
Odd parity

even parity



The two classes are **not** linearly separable. But, can separate by two hyperplanes or a nonlinear discriminant.

Multiple Layer Perceptron for XOR Problem



$$\hat{z}_1^{(1)} = U\left[x_2 - x_1 - \frac{1}{2}\right]$$

$$\hat{z}_2^{(1)} = U\left[x_2 - x_1 + \frac{1}{2}\right]$$

$$\hat{z}^{(2)} = U\left[\hat{z}_1^{(1)} - \hat{z}_2^{(1)} + \frac{1}{2}\right]$$

How Does it Classify it Correctly?

$$\hat{z}_1^{(1)} = U\left[x_2 - x_1 - \frac{1}{2}\right] \quad \hat{z}_2^{(1)} = U\left[x_2 - x_1 + \frac{1}{2}\right] \quad \hat{z}^{(2)} = U\left[z_1^{(1)} - z_2^{(1)} + \frac{1}{2}\right]$$

x_1	x_2	$\hat{z}_1^{(1)}$	$\hat{z}_2^{(1)}$	$\hat{z}^{(2)}$	
0	0	0	1	0	Class 1 (o)
0	1	1	1	1	Class 2 (x)
1	0	0	0	1	Class 2 (x)
1	1	0	1	0	Class 1 (o)

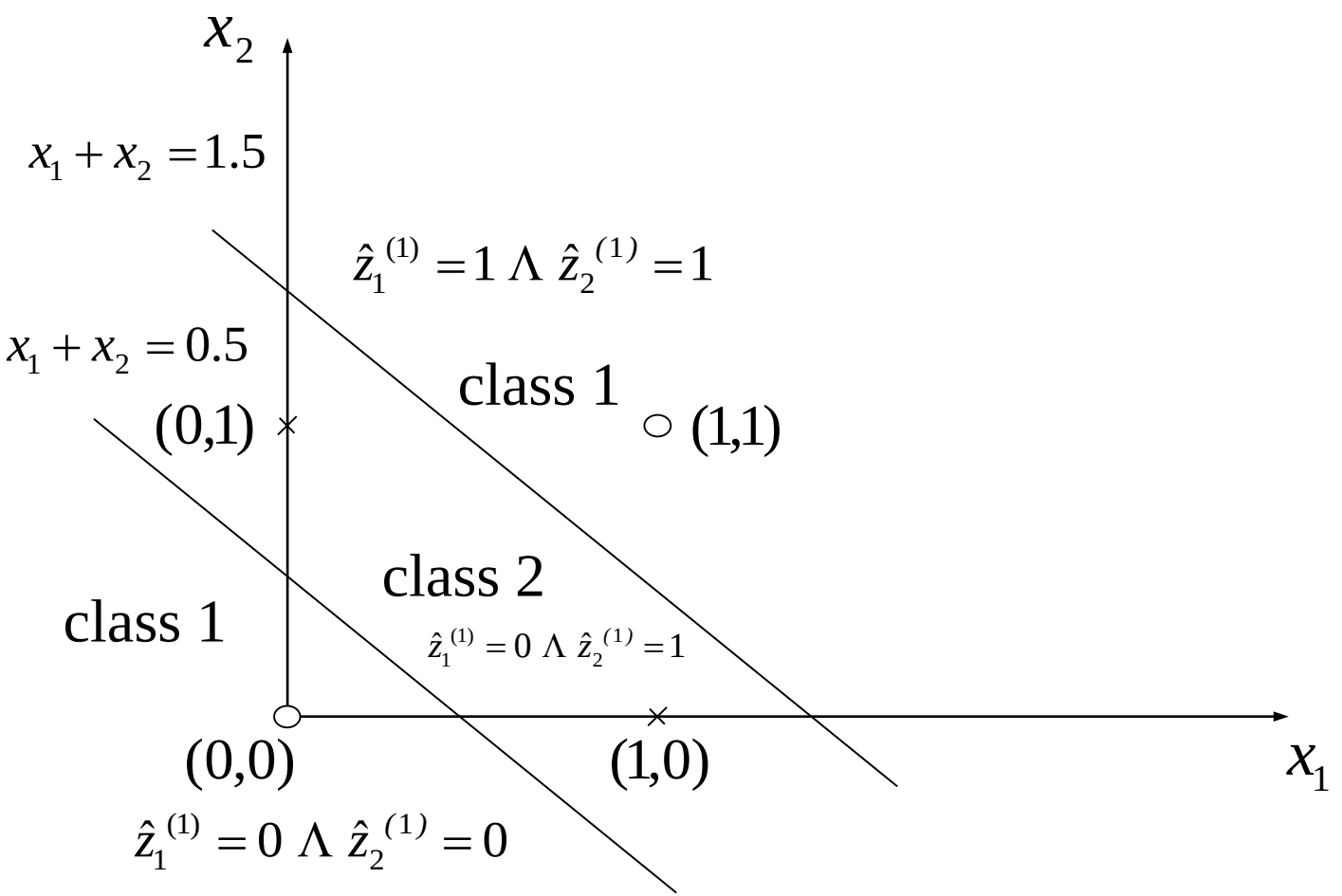
Alternate solution

$$\hat{z}_1^{(1)} = U\left[x_1 + x_2 - 1.5\right] \quad \hat{z}_2^{(1)} = U\left[x_1 + x_2 - \frac{1}{2}\right] \quad \hat{z}^{(2)} = U\left[\hat{z}_2^{(1)} - 2\hat{z}_1^{(1)} - \frac{1}{2}\right]$$

x_1	x_2	$\hat{z}_1^{(1)}$	$\hat{z}_2^{(1)}$	$\hat{z}^{(2)}$	
0	0	0	0	0	Class 1 (o)
1	0	0	1	1	Class 2 (x)
0	1	0	1	1	Class 2 (x)
1	1	1	1	0	Class 1 (o)

Geometric Interpretation

- Geometrically:



MLP as Universal Approximator

- Can extend it to *n*-bit parity problem
- In fact, any continuous function $f(x) = y$ (or a set of functions $\underline{f}(\underline{x}) = \underline{y}$) can be approximated by a three layer perceptron (or a NN with a single hidden layer). This is called the *universal approximation theorem*, e.g.,

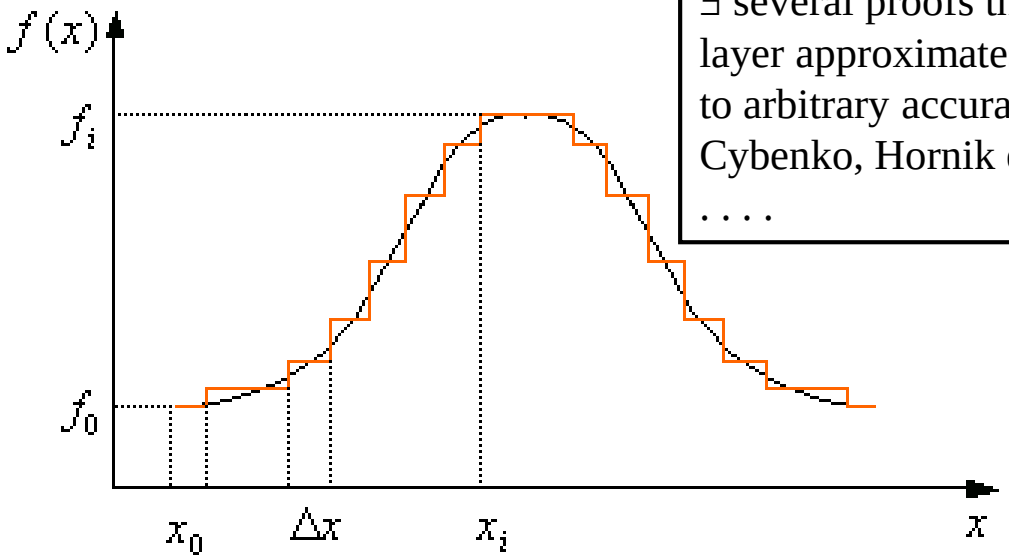
$$f = y\left(\underline{x}, \underline{w}^{(1)}, \underline{w}^{(2)}\right) = g\left[\sum_{i=0}^{M} w_i^{(2)}\, \underline{g}\left(\sum_{j=0}^{p} w_{ij}^{(1)} x_j\right)\right]$$

- An important corollary of this result is that, in the context of a classification problem, a method with sigmoidal nonlinearities and a hidden layer, can approximate any decision boundary to arbitrary accuracy e.g., 2–2–1 or 2–4–1 NN and 2–8–1 or 2–12–1 NN, for XOR problem.

Universal Approximation of MLP-1

Ideas of proof:

1. Any continuous function *f(x)* can be approximated by piecewise step functions.



∃ several proofs that a NN with a single hidden layer approximates a function or decision boundary to arbitrary accuracy. Funahasi, Hecht-Nielson, Cybenko, Hornik et al., Stinche Cormbe and White, ...

$$f(x) = f_0 + \sum_{i=0}^{N} (f_{i+1} - f_i) U(x - x_i)$$

can approximate it as closely as desired by controlling Δx (or equivalently $\Delta f_i = f_{i+1} - f_i$)

Universal Approximation of MLP - 2

2. Consider Fourier decomposition of $z(x_1, \cdots\cdots x_p)$ in the variable x_p

$$z(x_1, \cdots\cdots x_p) \cong \sum_{i_p} A_{i_p}(x_1, x_2 \cdots\cdots x_{p-1}) \cos(i_p x_p + \psi_{i_p})$$

similarly,

$$z(x_1, \cdots\cdots x_p) \approx \sum_{i_{p-1}} \sum_{i_p} A_{i_{p-1} i_p}(x_1, x_2 \cdots\cdots x_{p-2}) \cos(i_{p-1} x_{p-1} + \psi_{i_{p-1}}) \cos(i_p x_p + \psi_{i_p})$$

continuing,

$$z(x_1, \cdots\cdots x_p) \approx \sum_{i_1} \sum_{i_2} \cdots \sum_{i_p} A_{i_1 \ldots i_P} \cos(i_1 x_1 + \psi_{i_1}) \cdots\cdots \cos(i_p x_p + \psi_{i_p})$$

Universal Approximation of MLP - 3

using repeatedly the trigonometric identity

$$\cos\theta_1 \cos\theta_2 = \frac{1}{2}\left(\cos(\theta_1+\theta_2)+\cos(\theta_1-\theta_2)\right)$$

$$z(x_1,\cdots\cdots x_p) \cong \sum_{n=0}^{\infty} \sum_{i_1\cdots i_p : \sum_{k=1}^{p} i_k = n} C_{i_1\cdots i_p} \cos\left(\sum_{k=1}^{p} i_k x_k + \psi_{i_1\cdots i_p}\right)$$

Each of the cosine functions can be approximated by piece-wise step functions. Since sigmoidal, *tanh*, etc. approximate step functions, a two layer NN can approximate a function to arbitrary accuracy.

Training MLPs

How to train MLPs with (L+1) layers of Perceptrons?

Inputs at layer 0 ~ we generally ignore this layer

Outputs at layer L

Most widely touted algorithm is the LMS algorithm, which is called "*Back propagation*".

$\hat{z}_i(l) = i^{th}$ output of l^{th} *layer or state at layer l*

$$= g(\hat{y}_i(l)) == g\left(\sum_{j=0}^{M(l-1)} w_{ij}(l)\hat{z}_j(l-1)\right); \underline{\hat{z}}(l) = \underline{g}(W(l)\underline{z}^{(l-1)})$$

$$If\ \ g(z_i) = \frac{1}{1+e^{-\alpha z_i}} \quad \Rightarrow g' = \alpha g(1-g)$$

$$If\ \ g(z_i) = \tanh(\alpha z_i) \Rightarrow g' = \alpha(1-g^2)$$

Training Problem

$$\Rightarrow \hat{z}_i(L) = g\left[\sum_{j_{L-1}=0}^{M(L-1)} w_{ij_{(L-1)}}(L) \cdots\cdots g\left[\sum_{j_1=0}^{M(1)} w_{j_2 j_1}(2) \cdots\cdots g\left[\sum_{j_0=0}^{M(0)} w_{j_1 j_0}(1)\ \underbrace{z_{j_0}(0)}_{\text{input features}}\right]\right]\right]$$

- Training: want to match $\hat{z}_i(L)$ with $\left\{ z_i^{(n)} \right\}_{n=1}^{N}$ for $1 \le i \le M(L) = C$
- Formulate an LMS cost $E(W) = \frac{1}{2}\sum_{i=1}^{C}\sum_{n=1}^{N}\left[z_i^{(n)} - \hat{z}_i^{(n)}(L)\right]^2$
  - Same ideas for cross-entropy
- Want to use **incremental** gradient algorithm

$$w_{ij}^{(n+1)}(l) = w_{ij}^{(n)}(l) - \eta \frac{\partial E}{\partial w_{ij}^{(n)}(l)} = w_{ij}^{(n)}(l) + \Delta w_{ij}^{(n)}(l)$$

- For simplicity, we ignore superscript n from now on.

$$\hat{z}_i^{(n)} \rightarrow \hat{z}_i$$

Why Back Propagation?

Consider scalar equation $\dot{x} = a(\underline{\theta})x; x(0)$ *given* $\Rightarrow x(t) = e^{a(\underline{\theta})t}x(0)$

$$J = cx(t_f) = ce^{a(\underline{\theta})t_f}x(0) = ce^{a(\underline{\theta})(t_f - t)}e^{a(\underline{\theta})t}x(0) = \lambda(t)x(t)\forall t$$

$$\dot{\lambda}(t) = -a(\underline{\theta})\lambda(t); \lambda(t_f) = c$$

$$\frac{\partial J}{\partial \theta_i} = t_f ce^{a(\underline{\theta})t_f}\frac{\partial a(\underline{\theta})}{\partial \theta_i}x(0) = t_f\lambda(0)\frac{\partial a(\underline{\theta})}{\partial \theta_i}x(0) = t_f\lambda(t)\frac{\partial a(\underline{\theta})}{\partial \theta_i}x(t)$$

$$Note: \frac{\partial J}{\partial x(t)} = \lambda(t); \frac{\partial J}{\partial \lambda(t)} = x(t)$$

$$\Rightarrow \frac{\partial J}{\partial \theta_i} = \frac{\partial J}{\partial x(t)}\frac{\partial x(t)}{\partial \theta} + \frac{\partial J}{\partial \lambda(t)}\frac{\partial \lambda(t)}{\partial \theta}$$

$$= t\lambda(t)\frac{\partial a(\underline{\theta})}{\partial \theta_i}x(t) + (t_f - t)\lambda(t)\frac{\partial a(\underline{\theta})}{\partial \theta_i}x(t) = t_f\lambda(t)\frac{\partial a(\underline{\theta})}{\partial \theta_i}x(t)$$

Back propagation is essentially a chain rule of differentiation!

$$L(x) = L(f(h(g(x))$$

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial f}\frac{\partial f}{\partial h}\frac{\partial h}{\partial g}\frac{\partial g}{\partial x} = \lambda_g\frac{\partial g}{\partial x}$$

$$\lambda_f = \frac{\partial L}{\partial f}; \lambda_h = \frac{\partial L}{\partial h} = \lambda_f\frac{\partial f}{\partial h};$$

$$\lambda_g = \frac{\partial L}{\partial g} = \lambda_h\frac{\partial h}{\partial g}$$

$$x \to g \to h \to f \to L$$

$$\lambda_g\frac{\partial g}{\partial x} \leftarrow \lambda_g \leftarrow \lambda_h \leftarrow \lambda_f$$

So, to evaluate gradient, solve the forward equation for x and backward equation for λ at any time t, and evaluate the gradient.

Computing Gradient -1

$$w_{ij} \rightarrow \hat{y}_i \rightarrow \hat{z}_i$$

- The sequential gradient is given by

$$\frac{\partial E}{\partial w_{ij}(l)} = \frac{\partial E}{\partial \hat{z}_i(l)} \frac{\partial \hat{z}_i(l)}{\partial \hat{y}_i(l)} \frac{\partial \hat{y}_i(l)}{\partial w_{ij}(l)} == \lambda_i(l) g'(\hat{y}_i(l)) \hat{z}_j(l-1); \lambda_i(l) = \frac{\partial E}{\partial \hat{z}_i(l)}$$

- The key to back propagation is that $\lambda_i(l)$ can be computed from $\{\lambda_j(l+1)\}_{j=0}^{M(l+1)}$. Indeed, $\{\lambda_i(l)\}$ are **Lagrange multipliers or co-states in optimal control theory**. This is also called the **error signal**. Let us see how it works.

- At layer L, $\lambda_i(L) = \frac{\partial E}{\partial \hat{z}_i(L)} = \hat{z}_i(L) - z_i$

Now consider $\lambda_i(l)$ at layer $l = L-1, L-2, \cdots\cdots, 1$

$$\lambda_i(l) = \frac{\partial E}{\partial \hat{z}_i(l)} = \sum_{j=0}^{M(l+1)} \frac{\partial E}{\partial \hat{z}_j(l+1)} \frac{\partial \hat{z}_j(l+1)}{\partial \hat{y}_j(l+1)} \frac{\partial \hat{y}_j(l+1)}{\partial \hat{z}_i(l)}$$

Note that E can be any function of $\underline{z}$ and $\underline{\hat{z}}(L)$

e.g., $z_i \ln[z_i / \hat{z}_i]$

$|z_i - \hat{z}_i|^R, 1 \le R \le 2$

$$\hat{z}_i(l) \rightarrow \{\hat{y}_j(l+1)\} \rightarrow \{\hat{z}_j(l+1)\}$$

Computing Gradient -2

So, $\lambda_i(l) = \sum_{j=0}^{M(l+1)} \lambda_j(l+1) g'(\hat{y}_j(l+1)) w_{ji}(l+1)$

Or, $\underline{\lambda}(l) = W^T(l+1)\ Diag\left(g'(\underline{\hat{y}}(l+1))\right)\underline{\lambda}(l+1)$

where $W = [w_{ij}]$ is an $\left(M(l+1)+1\right)$ by $\left(M(l)+1\right)$ matrix

$\underline{\lambda}(l)$ is an $\left(M(l)+1\right)$ vector

$\underline{\lambda}(l+1)$ is an $\left(M(l+1)+1\right)$ vector

$$Diag\left(g'(\underline{\hat{y}}(l+1))\right) = \begin{bmatrix} g'(\hat{y}_0(l+1)) & & \\ & \ddots & \\ & & g'(\hat{y}_{M(l+1)}(l+1)) \end{bmatrix}$$

an $M(l+1)+1$ by $M(l+1)+1$ diagonal matrix

Computing Gradient - 3

- Note: For $l=L-1$, the summation goes from 1 to $M(L) = C$

This recursive formula is the key to **back propagation**. It allows the error signal (co-state) of a lower layer $\lambda_i(l)$ to be computed as a linear combination of the error signals at the layer $(l+1)$, viz., $\{\lambda_i(l+1)\}$. That is, the error signals can be ***back propagated*** from L to 1.

Sometimes (most NN books including Bishop's), one uses

$$\delta_i(l) = \frac{\partial E}{\partial \hat{y}_i(l)} = \frac{\partial E}{\partial \hat{z}_i(l)} \frac{\partial \hat{z}_i(l)}{\partial \hat{y}_i(l)} = \lambda_i(l) g'(\hat{y}_i(l))$$

Nonlinearity has local effect

the recursion then is:

$$\delta_i(l) = \left[\sum_{j=0}^{M(l+1)} \delta_j(l+1) w_{ji}(l+1) \right] g'(\hat{y}_i(l))$$

with the terminal condition $\delta_i(L) = \left(\hat{z}_i(L) - z_i\right) g'(\hat{y}_i(L))$

Weight Matrix Update

- In matrix form:

$$\underline{\delta}(l) = Diag\begin{bmatrix} g'(\hat{y}_0(l)) & & \\ & \ddots & \\ & & g'(\hat{y}_{M(l)}(l)) \end{bmatrix} W^T(l+1)\underline{\delta}(l+1)$$

- The weight update in matrix form is:

$$W^{n+1}(l) = W^n(l) - \eta^{(n)}\ Diag\begin{bmatrix} g'(\hat{y}_0(l)) & & \\ & \ddots & \\ & & g'(\hat{y}_{M(l)}(l)) \end{bmatrix} .\underline{\lambda}(l)\hat{\underline{z}}^T(l-1)$$

$$= W^n(l) - \eta^{(n)}\ \underline{\delta}(l)\hat{\underline{z}}^T(l-1)$$

Outer product

Training Algorithm in Brief

✓ ***Algorithmically***

1. Shuffle data $\left\{\underline{x}^n = \underline{\hat{z}}^n(0), \underline{z}^n\right\}_{n=1}^{N}$

2. For $n = 1,2,\ldots\ldots N$ Do

(i) Take data $\left\{\underline{x}^n, \underline{z}^n\right\}$

(ii) Execute forward pass. Determine $\hat{z}_i(l)$ and $g'(\hat{y}_i(l))$ and remember them.

(iii) Execute backward pass to evaluate $\{\lambda_i(l)\}$ and $\dfrac{\partial E}{\partial w_{ij}(l)}$

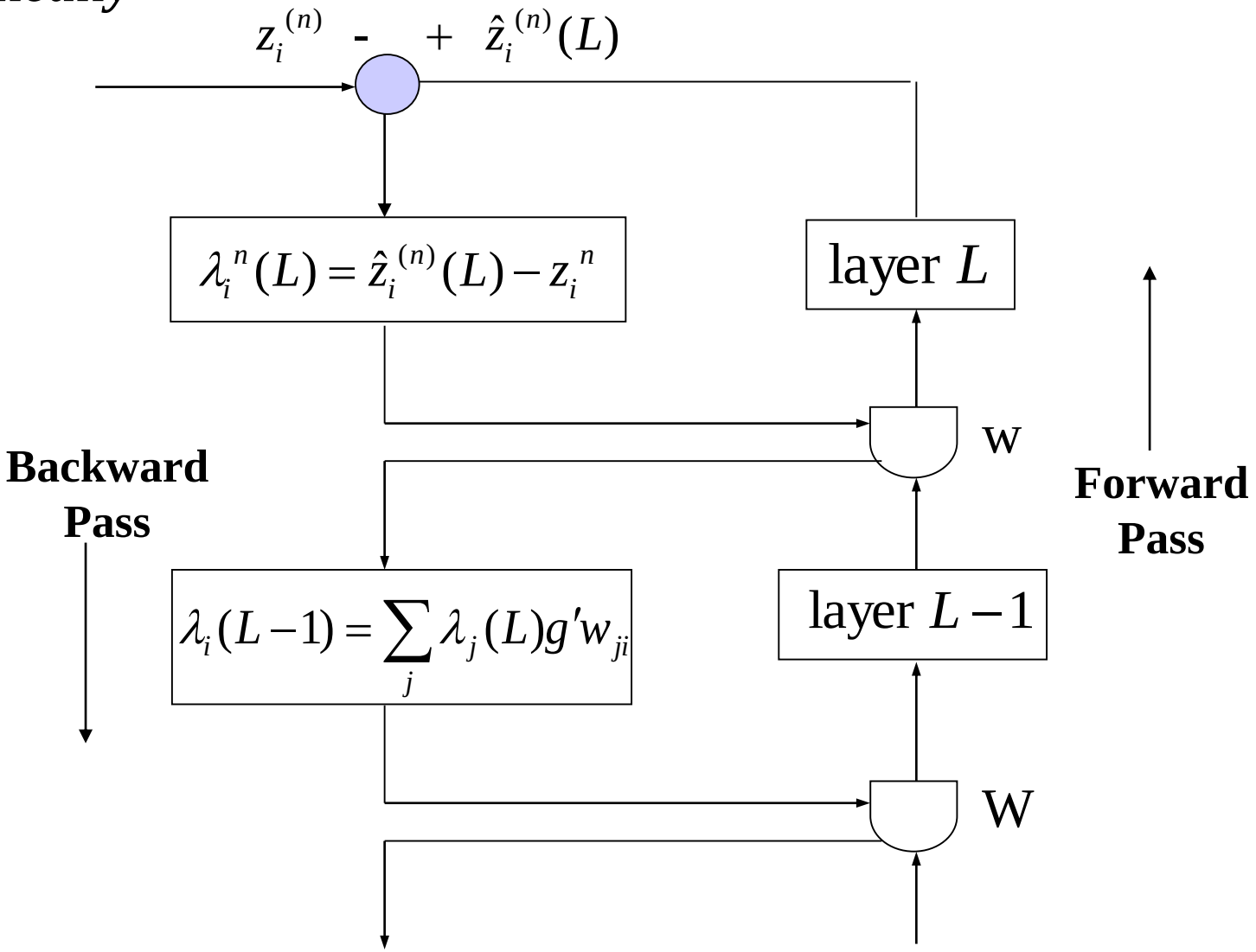
Update weights

End

3. Return to step 1

Graphical Illustration

✓ ***Graphically***



Backprop Practicalities

Some hints to make back propagation work better

- Use of "tanh" activation functions

$$g(z_i) = \alpha \tanh(\beta z_i) \qquad z_i = \underline{w}_i^T \underline{x}$$

$$\textit{Suggested} \quad \alpha, \beta: \quad \alpha = 1.716, \quad \beta = 2/3$$

- ReLU, Leaky ReLU, Exponential Linear Unit,… (see discussion on deep neural networks later)
- Weight initialization in the range

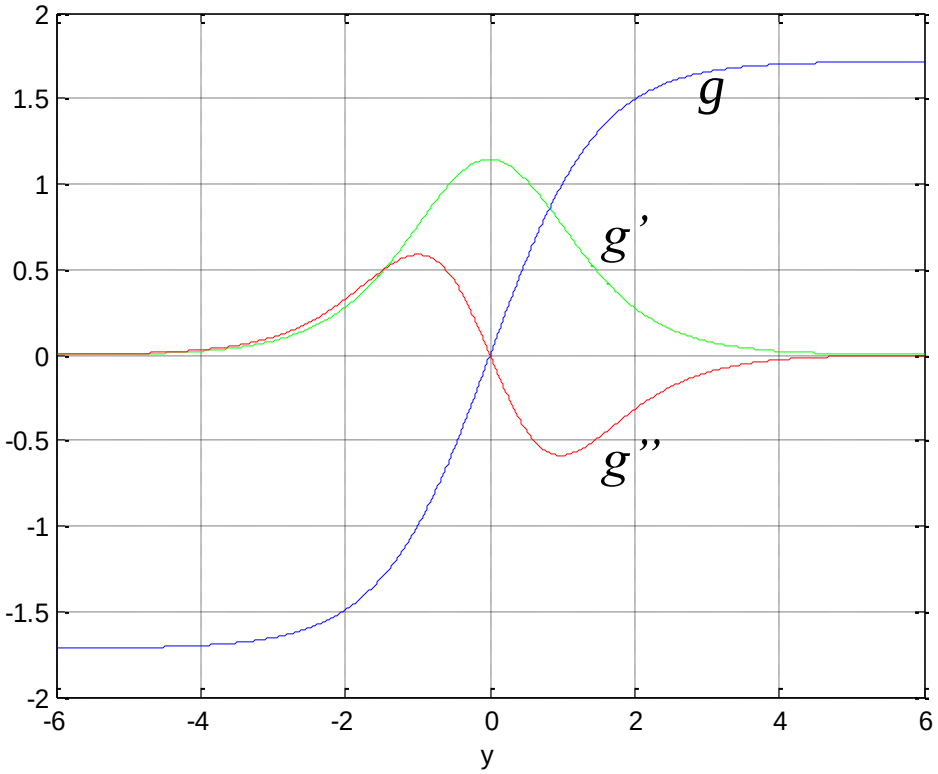
$$\left(\frac{-2.4}{|In_i|}, \frac{2.4}{|In_i|} \right); \text{ Alternate: } w_{ij} = N(0, 1/\frac{In_i}{2})$$

A variation of Xavier Initialization

In_i = number of incoming arcs to node i.

- Smaller η for higher layers because λ^s are smaller for lower layers. With batch normalization, don't need to do this anymore. Also, see http://ruder.io/optimizing-gradient-descent/index.html#adagrad

Why [–1.716, 1.716] Range?



- $g'(0) = \alpha\beta = 1.144$
- linear range (-1, 1)
- extrema of $g''(.)$ near ± 1

Training Practicalities - 1

- Some hints to make training work better
  - Target values for Classes: $class\ i\ traget = [-1 -1 -1 .... +1 -1 -1 .. -1]$
  - Minibatch averaged gradient. Typical batch sizes from 32 to 256.
  - Training with Noise: generate surrogate data with small noise added (e.g., 0.1 variance for scaled data). Good for unbalanced case.
  - Training with hints: e.g., in fault diagnosis, add another output for the fault severity during training only. Remove it for testing.
  - Randomized (Shuffled) training. Scale data to have zero mean and unit variance or between (0.15 0.85) for sigmoid or (-1.716 1.716) for tanh function. Use ReLU. Restart if training error does not decrease fast enough. Multi-start
  - Minimum training set size $N \approx 10^*$ number of weights
  - Perform cross validation or bootstrap. At worst, split N as follows: 65% Training; 10% Validation (use these to evaluate errors in training); 25% Testing (should not be seen and used only once)
  - Network size from prediction error

$$PE = \frac{2E}{N} + \frac{2N_W}{N}\sigma_e^2; \quad N_W = \text{number of weights}, \ \sigma_e^2 = \text{variance of input noise}$$

Training Practicalities - 2

Some hints to make training work better

- Adaptive η …. delta-bar-delta learning rule, quickprop, adagrad,…
- NLP techniques: Conjugate gradient, Memoryless Quasi-Newton method, MEKA. 2nd order methods need big batch sizes, like 10,000.
- Regularization (Network pruning techniques) to improve generalization. Build the simplest possible model or try to drive small weights to zero

$$E(w) = \frac{1}{2}\sum_{i=1}^{C}\sum_{n=1}^{N}\left(z_i^{\,n} - \hat{z}_i^{\,n}(L)\right)^2 + \sum_{l=1}^{L}\gamma_l E_l\left(w(l)\right);\ \gamma_l = \gamma \approx 0.001 - 0.1\sigma_e^2$$

$$E_l\left(w(l)\right) = \sum_{i=1}^{M(l)}\sum_{j=1}^{M(l-1)} w_{ij}^{\,2} \ldots \text{ weight decay procedure or } L_2 \text{ regularization}$$

$$w_{ij}^{\,(n+1)}(l) = w_{ij}^{\,(n)}(l) - \eta\left[\lambda_i(l)g'(\hat{y}_i(l))\hat{z}_j(l-1) + \gamma_l w_{ij}^{\,(n)}(l)\right]$$

$$= = [1-\eta\gamma_l]w_{ij}^{\,(n)}(l) - \eta\delta_i(l)\hat{z}_j(l-1)$$

- Weight eliminator

$$E_l\left(w(l)\right) = \sum_{i=1}^{M(l)}\sum_{j=1}^{M(l-1)}\left(\frac{w_{ij}}{w_0}\right)^2\left(1+\left(\frac{w_{ij}}{w_0}\right)^2\right)^{-1}$$

Training Practicalities - 3

- Some hints to make training work better
  - Curvature-driven smoothing

$$\sum_{n=1}^{N}\sum_{i=1}^{C}\sum_{j=1}^{p}\left(\frac{\partial^2 \hat{z}_i^{\,n}(L)}{\partial x_j^{\,n2}}\right)$$

- Network pruning … which weight should be set to zero

$$\nabla_{\underline{w}} E = -\sum_{n=1}^{N}\underbrace{\left(z^n - \hat{z}^n\right)}_{E^n} g(1-g)\underline{x}^n$$

$$Q = \nabla_{\underline{w}}{}^2 E = \sum_{n=1}^{N} g(1-g)\left[g(1-g) + E^n(2g-1)\right]\underline{x}^n\,\underline{x}^{nT} \approx \sum_{n=1}^{N}\tilde{x}^n\tilde{x}^{nT}; \tilde{x}^n = g(1-g)\underline{x}^n$$

Add μI if Q is not positive definite. The goal of optimal brain surgeon (OBS) procedure is to set one of the weights to zero so as to minimize the incremental increase in E.

$$\min J_i = \frac{1}{2}\Delta\underline{w}^T Q \Delta\underline{w}\ s.t.\ \underline{e}_i^{\,T}\Delta w + w_i = 0$$

Zero out small weights with large uncertainty

$$Solution: \Delta\underline{w} = -\frac{w_i}{[Q^{-1}]_{ii}} Q^{-1}\underline{e}_i \Rightarrow J_i = \frac{w_i^{\,2}}{2[Q^{-1}]_{ii}}$$

Training Practicalities - 4

- Estimation Error is a function of (Barron, 1994)
  - Smoothness of the approximation function, C_f
  - Number of neurons, m
  - Number of Training Examples, N
  - Dimension of the input data, p

$$error \approx Training\ Error + Generalization\ Error$$
$$\approx O(\frac{C_f}{m}) + O(\frac{mp}{N} \ln N)$$

- Dropout; Srivastava, Nitish, et al. *"Dropout: a simple way to prevent neural networks from overfitting."* *Journal of machine learning research* (2014)
- Normalization: Normalize activations of each layer (over a min-batch for each node (feature) or over nodes (features) for each sample of a mini-batch) to have same mean, β and variance, γ^2 (β and γ are hyper parameters). Alternately, normalize the weights of each layer:

$$W = g\frac{V}{\|V\|};\ \text{optimize over } g \text{ and } V \text{ via SGD}$$

- Early stopping: stop when validation error has not improved over n iterations (n is called "patience").
- Stochastic Gradient Descent is good for generalization

Predictive Posterior for Regression

- Nice to know uncertainty in predictions
- Parameter Posterior for Regression Case

$$J(\underline{w}) = \frac{1}{2}\sum_{i=1}^{C}\sum_{n=1}^{N}\beta_i\left(z_i^n - \hat{z}_i^n(L)\right)^2 + \frac{1}{2}\underline{w}^T Diag(\alpha_j)\underline{w}; \beta_i = \frac{1}{\sigma_i^2}\ i^{th}\text{output noise variance}$$

At optimum

$$p(\underline{w}\,|\,\underline{\alpha},\underline{\beta},D) = N(\underline{w};\underline{w}_{MAP},H^{-1}); H = \nabla^2_{\underline{w}} J(\underline{w}_{MAP}) \approx \sum_{n=1}^{N} G(\underline{x}^n)Diag(\beta_i)G^T(\underline{x}^n) + Diag(\alpha_j)I$$

$$G(\underline{x}^n) = \nabla_{\underline{w}}\,\underline{\hat{z}}^n(L) = \nabla_{\underline{w}}\,\underline{f}(\underline{x}^n,\underline{w}_{MAP}) = Jacobian$$

- Predictive Posterior for Regression

$$\underline{\hat{z}}(\underline{x}) = \underline{f}(\underline{x},\underline{w}) \approx \underline{f}(\underline{x},\underline{w}_{MAP}) + \nabla_{\underline{w}}\underline{f}^T(\underline{x},\underline{w}_{MAP})\underbrace{(\underline{w}-\underline{w}_{MAP})}_{\Delta\underline{w}}$$

$$p(\underline{z}\,|\,\underline{x},\underline{\alpha},\underline{\beta},D) \approx N(\underline{z};\underline{f}(\underline{x},\underline{w}_{MAP}),\Sigma_z(\underline{x}))$$

$$\Sigma_z(\underline{x}) = Diag(1/\beta_i) + \nabla_{\underline{w}} f^T(\underline{x},\underline{w}_{MAP})H^{-1}\nabla_{\underline{w}}\underline{f}(\underline{x},\underline{w}_{MAP})$$

- You can use EM-like algorithm to update $\{\underline{\alpha},\underline{\beta}\}$…. This is called automatic relevance determination (ARD) …. A form of feature selection

Automatic Relevance Determination

- ARD via Regression

$$\hat{\underline{z}}(\underline{x}) = \underline{f}(\underline{x}, \underline{w}) \approx \underline{f}(\underline{x}, \underline{w}_{MAP}) + \nabla_{\underline{w}} \underline{f}^T(\underline{x}, \underline{w}_{MAP})(\underline{w} - \underline{w}_{MAP})$$

$$p(\underline{z} \mid \underline{x}, \underline{\alpha}, \underline{\beta}, D) \approx N(\underline{z}; \underline{f}(\underline{x}, \underline{w}_{MAP}), \Sigma_z(\underline{x}))$$

$$\Sigma_z(\underline{x}) = Diag(1/\beta_i) + \nabla_{\underline{w}} f^T(\underline{x}, \underline{w}_{MAP}) H^{-1} \nabla_{\underline{w}} \underline{f}(\underline{x}, \underline{w}_{MAP})$$

$$H = \nabla^2_{\underline{w}} J(\underline{w}_{MAP}) \approx \sum_{n=1}^{N} G(\underline{x}^n) Diag(\beta_i) G^T(\underline{x}^n) + Diag(\alpha_j) I$$

Assume $\alpha_j \sim Ga(a_j, b_j)$ and $\beta_i \sim Ga(c_i, d_i)$

Expected Negative log of posterior:

$$l(\underline{\alpha}, \underline{\beta}) = \frac{1}{2} \sum_{n=1}^{N} \{\ln |\Sigma_z(\underline{x}^n)| + [\underline{z}^n - \underline{f}(\underline{x}^n, \underline{w}_{MAP})]^T \Sigma_z^{-1}(\underline{x}^n)[\underline{z}^n - \underline{f}(\underline{x}^n, \underline{w}_{MAP})]\} -$$

$$\left[\sum_{j=1}^{M} (a_j - 1) \ln \alpha_j - b_j \alpha_j\right] - \left[\sum_{i=1}^{C} (c_i - 1) \ln \beta_i - d_i \beta_i\right]$$

- Exploit the special structure of H in deriving gradients. We will revisit this issue in the context of relevance vector machine (RVM).

Predictive Posterior for Classification

- Classification

$$J(\underline{w}) = \sum_{n=1}^{N} \left[z^n \ln g(\underline{x}^n, \underline{w}) + (1 - z^n) \ln(1 - g(\underline{x}^n, \underline{w})) \right]$$

Cross entropy or Deviance

$g(\underline{x}^n, \underline{w}) = \sigma(y(\underline{x}^n, \underline{w}))$... sigmoid function of $y(\underline{x}^n, \underline{w})$

At optimum

$$p(\underline{w} \mid D) = N(\underline{w}; \underline{w}_{MAP}, H^{-1}); H = \nabla^2_{\underline{w}} J(\underline{w}_{MAP})$$

$$p(g(\underline{x}, \underline{w}) \mid \underline{x}, D) \approx N(g(\underline{x}, \underline{w}); g(\underline{x}, \underline{w}_{MAP}), \nabla_{\underline{w}} g^T(\underline{x}, \underline{w}) H^{-1} \nabla_{\underline{w}} g^T(\underline{x}, \underline{w}))$$

$$P(z = 1 \mid \underline{x}, D) \approx \sigma\left(\frac{g(\underline{x}, \underline{w}_{MAP})}{\sqrt{1 + \frac{\pi \nabla_{\underline{w}} g^T(\underline{x}, \underline{w}) H^{-1} \nabla_{\underline{w}} g^T(\underline{x}, \underline{w})}{8}}} \right)$$

Hedges against uncertainty

Semi-supervised Learning of MLPs

- L labeled and U unlabeled set of training examples
- Have side information (e.g., objects n and m are similar)

$S_{nm} = 1$ *if* n *and* m *are similar*; 0 *otherwise*

Let $\hat{z}(\underline{x}^n, W)$ *be output estimate for an input* $\underline{x}^n$

Define

$$L(\hat{z}(\underline{x}^n, W), \hat{z}(\underline{x}^m, W), S_{nm}) = \begin{cases} \| \hat{z}(\underline{x}^n, W) - \hat{z}(\underline{x}^m, W) \|^2 \text{ if } S_{nm} = 1 \\ \max(0, M - \| \hat{z}(\underline{x}^n, W) - \hat{z}(\underline{x}^m, W) \|^2) \text{if } S_{nm} = 0 \end{cases}$$

M = minimal margin between dissimilar data

$$J(W) = \frac{1}{2}\left(\sum_{n \in L} \left(z^n - \hat{z}^n(L) \right)^2 + \lambda \sum_{n,m \in U} L(\hat{z}(\underline{x}^n, W), \hat{z}(\underline{x}^m, W), S_{nm}) \right)$$

Optimize W *via stochastic gradient by successively sampling from labeled data, unlabeled data with* $S_{nm} = 1$ *and unlabeled data with* $S_{nm} = 0$.

Variations

- Extreme Learning Machines
  - Randomized projection of inputs to hidden layers

    $h_i(\underline{x}) = g(\underline{w}_i^T \underline{x} + w_0); i = 1, 2, .., K; K$ reasonably large;

    $g \sim sigmoid, \tanh, rectifier, ...; \underline{w}_i, w_0$ are selected randomly
  - Linearly combine the outputs of hidden layers

    $\hat{z}(\underline{x}) = \sum_{i=1}^{K} v_i g(\underline{w}_i^T \underline{x} + w_0); \{v_i\}_{i=1}^{K}$ selected to minimize

    $$\frac{1}{2}\sum_{n=1}^{N}\left(z^n - \hat{z}^n(\underline{x}^n)\right)^2 + \frac{\lambda}{2}\underline{v}^T\underline{v} \Rightarrow \text{Re}\, gularized\ LS; LASSO, Elastic\ Net, ...$$

- Convolutional Neural Networks (image processing/character recognition; used in deep learning)
  - Series of convolutional and subsampling layers
    - **Local receptive fields and weight sharing**: Inputs from a small region (say 3x3 pixel patch) are mapped into next layer via sigmoid/ReLU (same weights and bias) for all units
    - **Pooling:** Subsampling takes small regions of convolution layers and pools them (e.g., average, max value), scales it, adds a bias and transforms via a nonlinearity, such as ReLU

Why Deep Networks?

- Training MLP is difficult if more than two hidden layers are used. The more layers one uses, the **more difficult the training becomes (unstable gradients)** and **probability of getting stuck at local minima increases**.
  - Is there any need for networks with more than two or three layers?
    - The cortex of human brain can be seen as a multilayer architecture with 5-10 layers dedicated only to our visual system (Hubel and Wiesel, 1962; also DL Rev. book)
    - Using networks with **more layers** can lead to more compact representations of the input-output mapping.
- Hastad switching lemma (1987) – Boolean circuits representable with a polynomial number of nodes with *k* layers may require an exponential number of nodes with (*k*-1) layers (e.g., parity) … more depth is better
  - Deep networks may require less number of weights than a shallow representation
- Feature hierarchies (e.g., pixel → edge → texton → motif → part → object in an image recognition task) identified in deep networks can potentially be shared among multiple tasks
  - Transfer/Multi-task learning
- Currently DNN training is a painful "trial-and-error" process or uses genetic algorithms or Bayesian optimization

What Made Deep Learning Feasible?

- Deluge of data and ability to crowd source the labeling process for supervisory learning …Availability of *large scale (and often pre-trained) labeled images* (e.g., Alexnet, ImageNet, VGG16) and advances in *cross-mod*al (image, text/data, video and audio) *representations*…. "*Big Data*"
- Availability of *Graphics Processing Units* (GPUs) that enable training of very large image/text/speech datasets from several weeks to a few days
- *New nonlinearities* (e.g., rectified linear units) and *signal normalization* that avoid numerical problems associated with gradient computations
- *Convolution and max-pooling* operations that exploit local connectivity to reduce the dimension of the weight space, control overfitting and make the network robust
- Concept of *dropout* to realize an exponentially large ensemble of networks from a single network
- Advances in *stochastic optimization, adversarial training and regularization* for robust network training
- Features are learned rather than hand-crafted
- More layers capture more invariances (Information-theoretic insights)

ReLU and Cross-Entropy Avoid Unstable Gradient Problem

- Deep networks with sigmoid and tanh nonlinearities experience unstable gradient problem
  - Vanishing Gradient – recall saturation of sigmoids and tanh ⇒ slow training

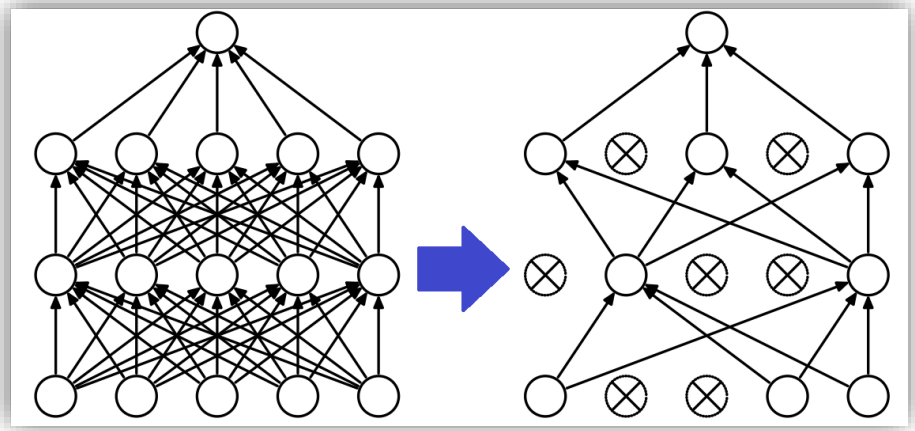
$$\delta_i(l) = \left[\sum_{j=0}^{M(l+1)} \delta_j(l+1) w_{ji}(l+1) \right] g'(\hat{y}_i(l))$$

  - Exploding gradient - products of many terms over layers ⇒ unstable training
- One solution: Rectified Linear Units (ReLU) … easy to compute gradient
  - $g(y) = \max(0, y)$. Note non-differentiability at $y = 0$
  - Leaky ReLU: set $g(y) = ay$ for $y \leq 0$ where a is small positive number (≈ 0.1)
  - Exponential linear unit (ELU): set $g(y) = \alpha(\exp(y)-1)$ for $y \leq 0$; $\alpha \approx 1$
- Second Solution: Cross-entropy (Deviance) cost function avoids gradient saturation
  - Recall gradient does not have g' and convexity of cross-entropy… Recall

$$\nabla_w J = -\sum_{n=1}^{N} \{z_n - g_n) \underline{x}_n \;\&\; \nabla_w^2 J = \sum_{n=1}^{N} g_n' \underline{x}_n \underline{x}_n^T$$

Regularization using Dropout

- Each node should interact with a small random set of other nodes
  - At each iteration during training, each node is retained with a probability p
  - During testing, the whole network is used, but the weights are scaled-down by p (Google's Tensorflow scales weights by $1/p$ and does not scale them during testing!)
  - Essentially, dropout may be viewed as an ensemble of 2^n networks (n = number of nodes in the network) …. Better than weight decay regularization



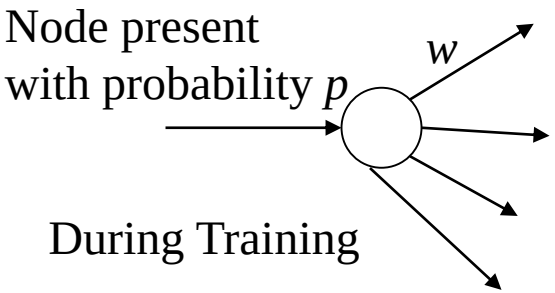
p = 0.5 works!

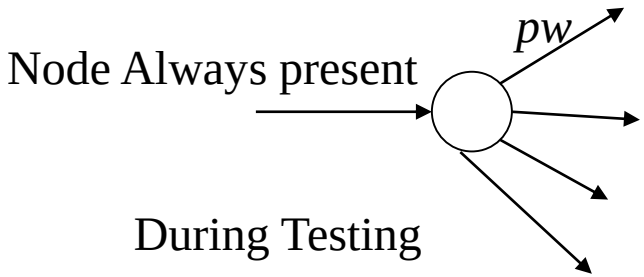
Approximation:

R = node masking matrix

$$E_R(g(R(l+1)W(l+1)\underline{z}^{(l)})) \approx g(pW(l+1)\underline{z}^{(l)})$$

Dropout does not help if you have big data





Batch Normalization

- Powers of 2 Minibatch stochastic methods ideal for multi-core and GPUs
- Renormalize activations at each layer over a minibatch … Whitening
  - Improves backpropagation-based gradient computation through the network
  - Allows higher learning rates
  - Reduces the strong dependence on initial weights
  - Store the running average of sample mean and sample variance for testing

$B =$ Minibatch; $z_i^{(0)} = x_i; i = 1, 2, .., p+1$

For $l = 1, 2, .., L$

$$u_j^{(l,n)} = \sum_{i=1}^{M(l-1)} w_{ij}^{(l)} z_i^{(l-1,n)} = \text{Activation of node } j \text{ at layer } l \text{ for traing sample } n \in B$$

$$m_j^{(l)} = \frac{1}{|B|}\sum_{n\in B} u_j^{(l,n)}; s_j^{(l)} = \frac{1}{|B|-1}\sum_{n\in B}(u_j^{(l,n)} - m_j^{(l)})^2; \; j = 1, 2, ., M(l)$$

$$\hat{u}_j^{(l,n)} = \frac{u_j^{(l,n)} - m_j^{(l)}}{\sqrt{s_j^{(l)} + \varepsilon}}; \; j = 1, 2, ., M(l); n \in B$$

$$y_j^{(l,n)} = \gamma^{(l)} \hat{u}_j^{(l,n)} + \beta^{(l)}; \; j = 1, 2, ., M(l); n \in B$$

$$z_j^{(l,n)} = g^{(l)}(y_j^{(l,n)}); \; j = 1, 2, ., M(l); n \in B; \text{ nonlinearity can be layer dependent}$$

End

Alternatives: Normalize over features for each sample of mini-batch or normalize weights of each layer

γ & β are learned from data. Backpropagation can be modified easily https://kratzert.github.io/2016/02/12/understanding-the-gradient-flow-through-the-batch-normalization-layer.html

Convolutional Networks

- Inspired by human's visual system structure (local receptive fields, weight sharing and pooling/aggregation)
- Much Faster to train (fewer weights)
- Recall convolution for one-dimensional discrete-time signals

$$s(k) = x(k) * w(k) = \sum_{n=-\infty}^{\infty} x(n)w(n-k) = \sum_{n=-\infty}^{\infty} x(n-k)w(n)$$

- For two-dimensional signals (e.g., images), convolution operator is

$$s(i,j) = x(i,j) * w(i,j) = \sum_m \sum_n x(m,n)w(i-m, j-n) = \sum_m \sum_n x(i-m, j-n)w(m,n)$$

- Most NN libraries use cross-correlation instead

$$s(i,j) = x(i,j) * w(i,j) = \sum_m \sum_n x(i+m, j+n)w(m,n)$$

- **Key**: W (receptive field) will have much smaller dimension than image dimension X and W is shared $\Rightarrow$ smaller number of weights to learn. You can have multiple W's … called convolution kernels/filters/maps
  - X is 32x32x3 and each W is 5 x 5x3 and 6 weight maps $\Rightarrow$ need to learn only (5x5x3+1)x6=456 weight parameters to be learned
  - You get 6 maps of 28x28x1 as outputs or 28x28x6 stacked maps….. (28=32-5+1)
  - Connections: 76x6x28x28 =456x28x28=357,504 edges

How Convolution Works with Strides and Zero Padding ?

- What are the output volume sizes? *F*: filter size; *N*: input size; *L*: # Filters
  - 32x32x3 image → 6 5x5x3 convolution filters + ReLU → 10 5x5x6 convolution filters + ReLU
  - Output of first set of convolution filters: (*N*-*F*+1)x(*N*-*F*+1) x *L*=28x28x6
  - Output of second set of convolution filters: 24x24x10
  - Number of weight parameters: 456+1510=1966
- Typically, we move the filter one pixel at a time. What if we skip *S* pixels called stride, *S*
  - 32x32x3 image → 6 5x5x3 convolution filters, stride 3+ ReLU → 10 5x5x6 convolution filters, stride 1 + ReLU
  - Output of first set of convolution filters: ((*N*-*F*)/*S*+1)x((*N*-*F*)/*S*+1) x *L*=10x10x6
  - Output of second set of convolution filters: 6x6x10
  - Stride 2 will not work in the first set of convolution filters because (32-5)/2+1=14.5
  - Mostly, use stride, *S*=1; *L* is typically a power of 2
- In practice, common to zero pad the border with zeros. If pad with *P*= (*F*-1)/2 zeros with stride, *S*=1 will not change the dimension spatially
  - 32x32x3 image → 6 5x5x3 convolution filters, stride 1, zero pad 2+ ReLU → 10 5x5x6 convolution filters, stride 1, zero pad 2 + ReLU
  - Output: 32x32x3 → 32x32x6 → 32x32x10

Summary

- Convolution layer takes a volume I_1 x J_1 x K_1 as input
- Convolution layer is parametrized by
  - F: filter size
  - L: # Filters
  - S: Stride
  - P: Amount of zero padding
- Produces a volume of I_2 x J_2 x K_2 as output

$$I_2 = \frac{(I_1 - F + 2P)}{S} + 1$$

$$J_2 = \frac{(J_1 - F + 2P)}{S} + 1$$

$$K_2 = L$$

$$Note: \frac{(I_1 - F + 2P)}{S} \text{ and } \frac{(J_1 - F + 2P)}{S} \text{ must be divisible}$$

- Number of weights per filter: F x F x K_1 + 1 (for bias)
- Total Number of weights: (F x F x K_1 +1)x L
- 1x1 convolution is perfectly OK to use

Example: LeNet 5

- Y. LeCun, L. Bottou, Y. Bengio and P. Haffner,"Gradient-based Learning Applied to Document Recognition, *Proceedings of the IEEE*, Vol. 86, No. 11, pp. 2278-2324, November 1998.

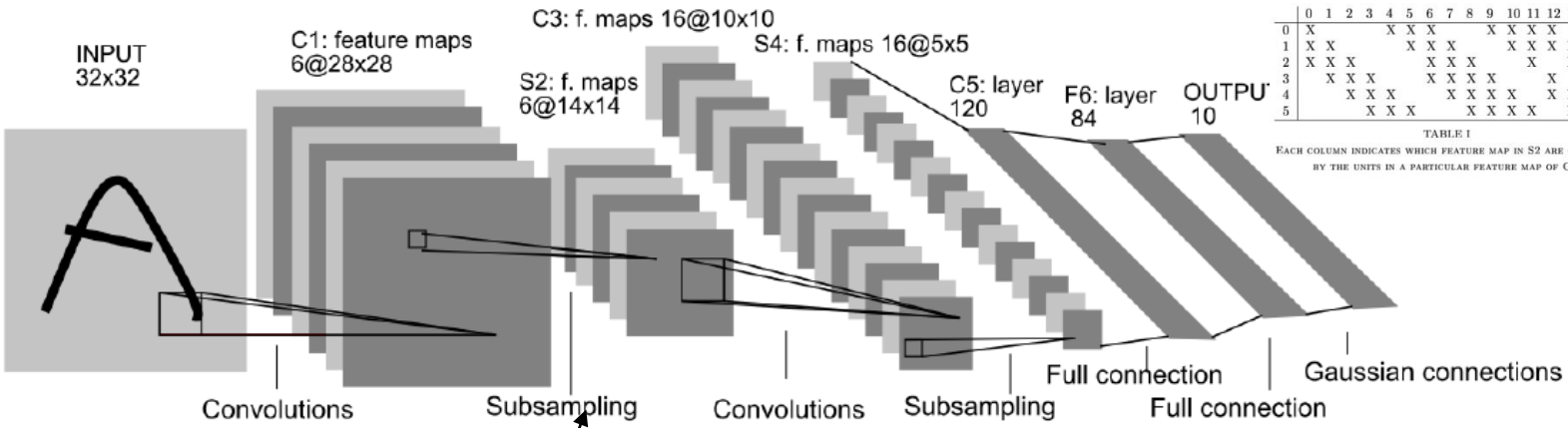


TABLE I

EACH COLUMN INDICATES WHICH FEATURE MAP IN S2 ARE COMBINED BY THE UNITS IN A PARTICULAR FEATURE MAP OF C3.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	X				X	X	X			X	X	X	X		X	X
1	X	X				X	X	X			X	X	X	X		X
2	X	X	X				X	X	X			X		X	X	X
3		X	X	X			X	X	X	X			X		X	X
4			X	X	X			X	X	X	X		X	X		X
5				X	X	X			X	X	X	X		X	X	X

http://www.cs.cmu.edu/~aarti/Class/10701_Spring14/slides/DeepLearning.pdf

Max pooling is common. Other options: averaging, norm,….

C1,3,5: Convolutional layers
S2,4: Subsampling (pooling) layers
F6: Fully connected layer

Layer	Trainable Weights	Connections (Edges)
C1	(25+1)x6 = 156	(25+1)x6x28x28 = 122,304
S2	(1+1)x6 = 12	(4+1)x6x14x14 = 5880 (2x2 links and bias = 5)
C3	6x(25x3+1) + 9x(25x4+1) + 1x(25x6+1) = 1516	1516x10x10 = 151,600
S4	16x2 = 32	16x5x5x5 = 2000 (2x2 links and bias = 5)
C5	120x(5x5x16+1) = 48,120	Same since fully connected MLP at this point
F6	84x(120+1) = 10,164	Same
Output	10x(84+1) = 850 (RBF)	Same

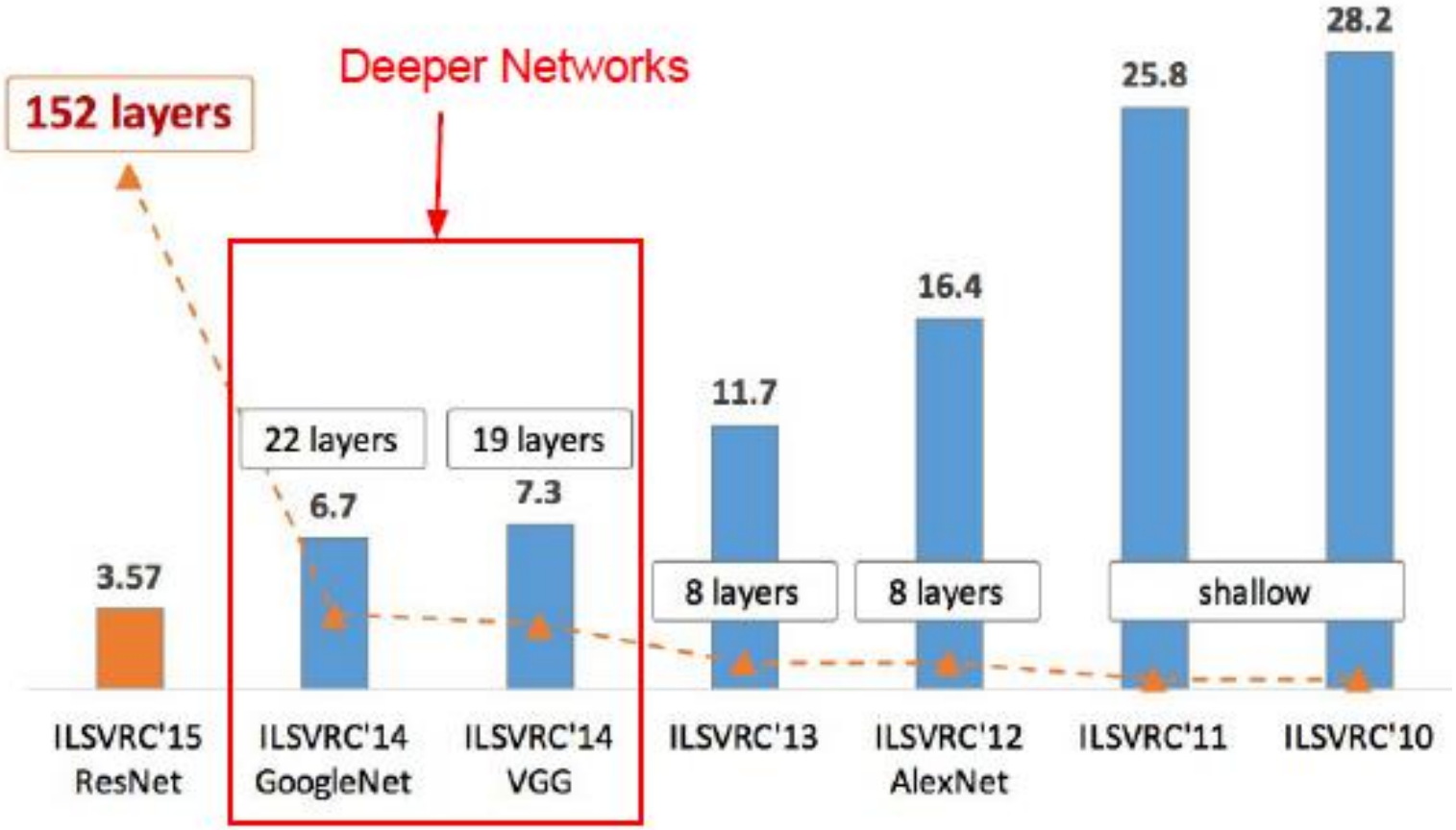
60,850 weights
(instead of millions)

Training Convolutional Networks

- Trained with BP, but with weight sharing
  - Just average the weight updates over the shared weights in feature map layers
- Convolution layer
  - A *k*x*k* receptive field would have a total of (k^2 + 1) weights
  - If a convolution layer had *m* feature maps, then only a total of (k^2 + 1)*m* unique weights to be trained in that layer (much less than a fully connected layer!)
- Sub-Sampling (Pooling) Layer
  - Take maximum, average, norm, or weighted average of all elements of a receptive filed. Result multiplied by one trainable weight and a bias added, then passed through a non-linear activation function (e.g. ReLU) for each pooling node
  - If a layer has *m* pooling feature maps, then 2*m* unique weights are to be trained
- The structure of the CNN is usually hand-crafted using trial and error
  - Number of layers, number of receptive fields, sizes of receptive fields, sizes of sub-sampling (pooling) fields, which fields of the previous layer to connect to, etc.
  - Typically, decrease the size of feature maps and increase the number of feature maps for later layers … typically by a factor of 2 every few layers

CNN's Success in ImageNet Large Scale Visual Recognition Challenge

- ImageNet Classification Top-5 Error in Percentage

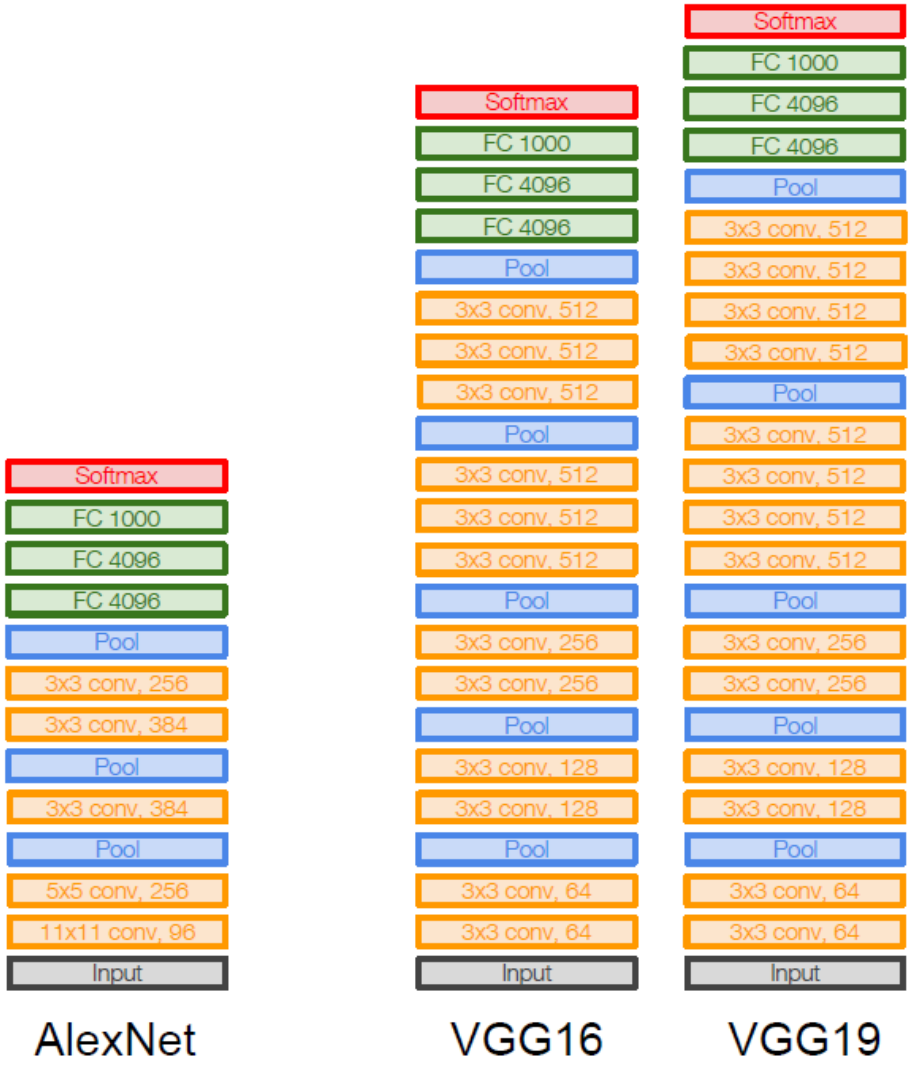


http://icml.cc/2016/tutorials/icml2016_tutorial_deep_residual_networks_kaiminghe.pdf

CNN Structures in Image Recognition Competitions - 1

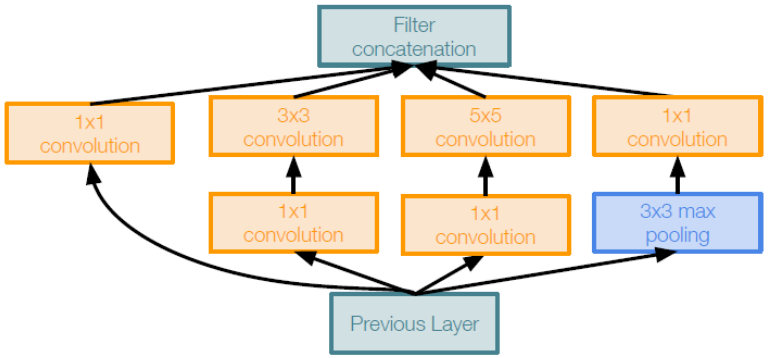
- AlexNet, 8 Layers, top 5 error: 16.4% (2012), 1st used ReLU ~60M weights
  - Input image 227x227x3
  - 5 combined convolution and pooling layers: 11x11 convolution, 96 maps (conv 11-96)→ → max pool → batch norm → conv 5-256 → max pool → batch norm → conv 3-384 → conv3-384 → conv 3-256 → max pool → batch norm
    - Stride of 4 for first convolution kernel (⇒,output (227-11)/4+1=55x55x96 and 11x11x96 parameters) and 1 for the rest; Pooling layers with 3x3 receptive fields and stride of 2 throughout
  - 2 fully connected (FC) layer with 4096 nodes
  - Output layer with 1000 output nodes for classes
  - https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf
- ZFNet: 8 layers, top 5 error: 11.7% … hyper parameter tuning of AlexNet
  - AlexNet with conv 11-96 stride 4 → conv 7-96 stride 2; conv 3-384 → conv 3-512; conv 3-384 → conv 3-1024 and conv 3-256 → conv 3-512
  - https://arxiv.org/pdf/1311.2901.pdf
- VGG 19 layers, top 5 error: 7.3% (2014)
  - 2 conv 3-64 → max pool → 2 conv 3-128 → max pool → 4 conv 3-256 → max pool → 4 conv 3-512 → max pool → 4 conv 3-512 → max pool → 2 FC 4096 → FC 1000 → softmax ….. 3x3 conv with 1 stride has less parameters and same effective receptive field as 7x7 conv; Most memory in early layers and most weights in later FC layers
  - https://arxiv.org/pdf/1409.1556.pdf

CNN Structures in Image Recognition Competitions - 2

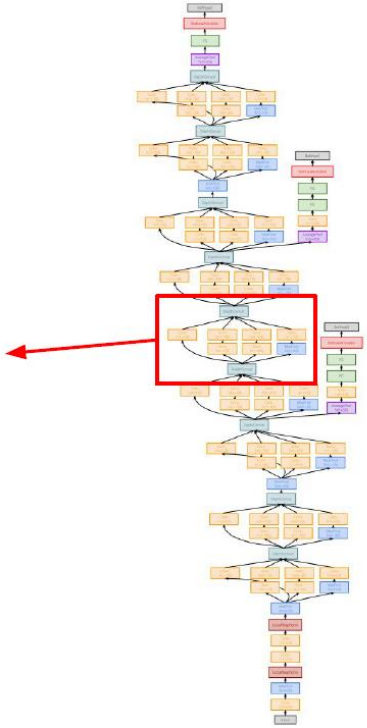


CNN Structures in Image Recognition Competitions - 2

- GoogleNet, 22 Layers, top 5 error: 6.7% (2014),… called Inception-v1
  - Conv 7-64 with stride 2 → max pool 3/3 stride 2 → Conv 3-192 → max pool 3x3 stride 2
  - Inception module: a good local network module with multiple parallel conv filters (1x1, 3x3 and 5x5) and a 3x3 pooling operation. These are then concatenated.
  - No fully connected layers at the end
  - Only 5M weights
  - https://arxiv.org/pdf/1409.4842.pdf

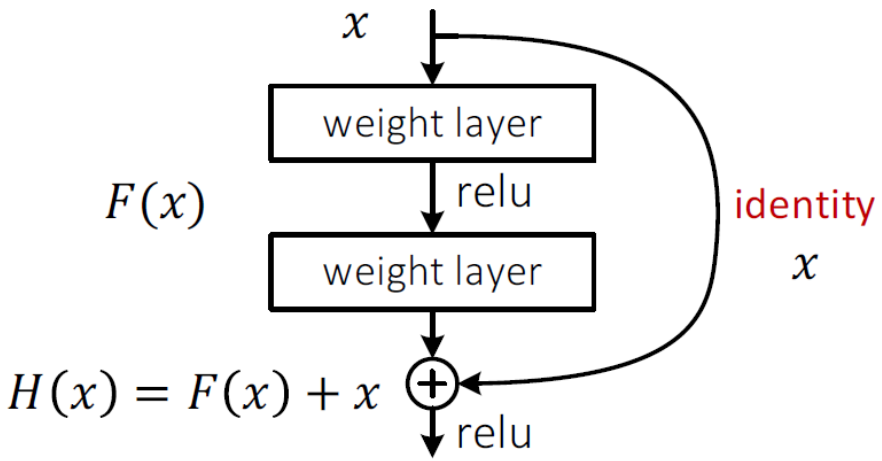


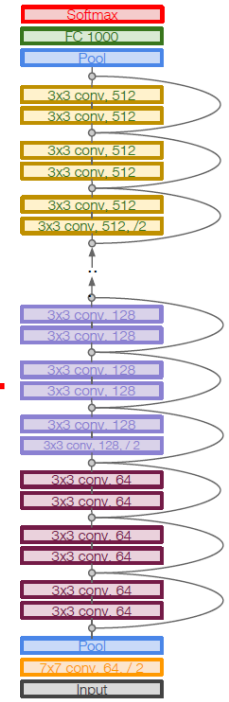
Inception module with dimension reduction



CNN Structures in Image Recognition Competitions - 5

- ResNet, 152 Layers, top 5 error: 3.57% (2015)
  - Idea: Instead of training $H(\underline{x})$ directly, train the residual $F(\underline{x}) = H(\underline{x}) - \underline{x}$
  - Stack residual blocks, 18 or 34 or 50 or 101 or 152 layers
  - Every residual block has two 3x3 conv layers
  - Periodically, double the number of maps and down sample using stride 2.
  - At the beginning Conv 7-64 with stride 2 and 3x3 max pooling with stride 2
  - No fully connected layers at the end
  - https://arxiv.org/pdf/1512.03385.pdf





Current State of DNNs for Image Recognition

- Inception-ResNet-v2 by Google combines GoogleNet and ResNet ideas. Inception-4 is the latest incarnation of GoogleNet
  - https://arxiv.org/pdf/1602.07261.pdf
  - https://arxiv.org/pdf/1605.07678.pdf

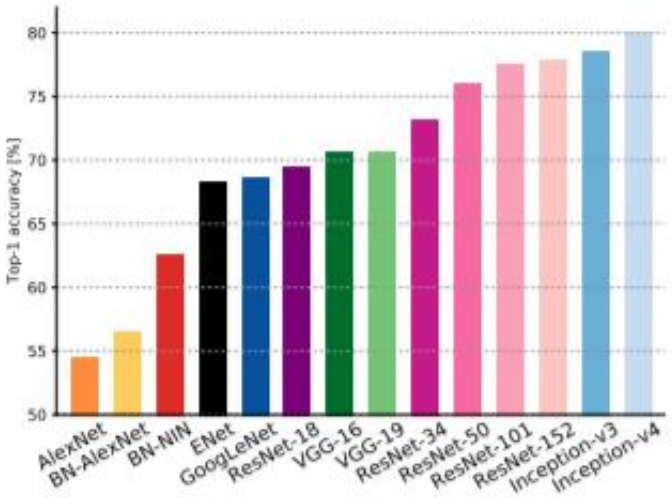


Figure 1: **Top1 *vs.* network.** Single-crop top-1 validation accuracies for top scoring single-model architectures. We introduce with this chart our choice of colour scheme, which will be used throughout this publication to distinguish effectively different architectures and their correspondent authors. Notice that networks of the same group share the same hue, for example ResNet are all variations of pink.

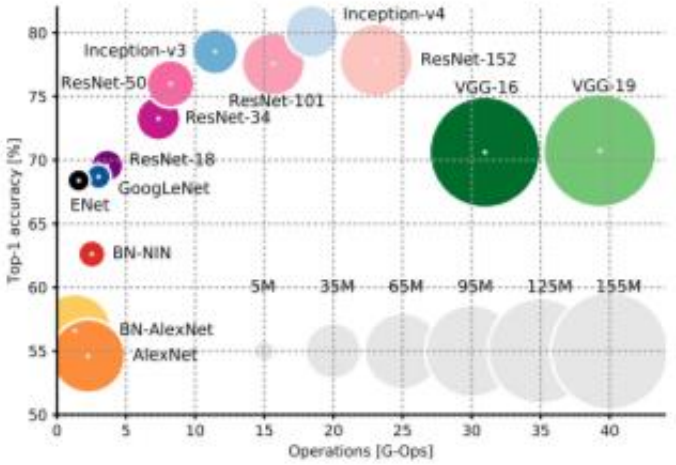


Figure 2: **Top1 *vs.* operations, size $\propto$ parameters.** Top-1 one-crop accuracy versus amount of operations required for a single forward pass. The size of the blobs is proportional to the number of network parameters; a legend is reported in the bottom right corner, spanning from 5×10^6 to 155×10^6 params. Both these figures share the same y-axis, and the grey dots highlight the centre of the blobs.

Early Methods of Training Deep Networks

- Use of unsupervised learning to transform inputs into features that are easier to learn by a final BP-based supervised model
- Often not a lot of labeled data available, while there may be lots of unlabeled data. Unsupervised Pre-Training can take advantage of unlabeled data. Can be a huge issue for some tasks.
- Q: Is there a training scheme that can get "good" local minima, which can then be fine-tuned by BP?
  - Approach 1: Pre-train each layer, via an unsupervised learning algorithm, one layer at a time, in a greedy manner. The most popular approach is the **Restricted Boltzmann Machines** (RBM).
  - Approach 2: Replace layered RBMs with stacked **auto-encoders**. The latter have been proposed as methods for dimensionality reduction. An auto-encoder consists of two parts, the **encoder** and the **decoder**.

$$Encoder: h_i = f_e(\underline{w}_i^T \underline{x} + w_0); i = 1,2,..,r << p; W = \left[\underline{w}_1 \quad \underline{w}_2 \quad . \quad \underline{w}_r\right]$$

$$Decoder: \hat{x}_j = g_d(\underline{v}_j^T \underline{\hat{h}} + v_0); j = 1,2,..,, p; V = \left[\underline{v}_1 \quad \underline{v}_2 \quad . \quad \underline{v}_p\right] = W^T$$

  - The new CNN and BP-based deep network training has made these obsolete.

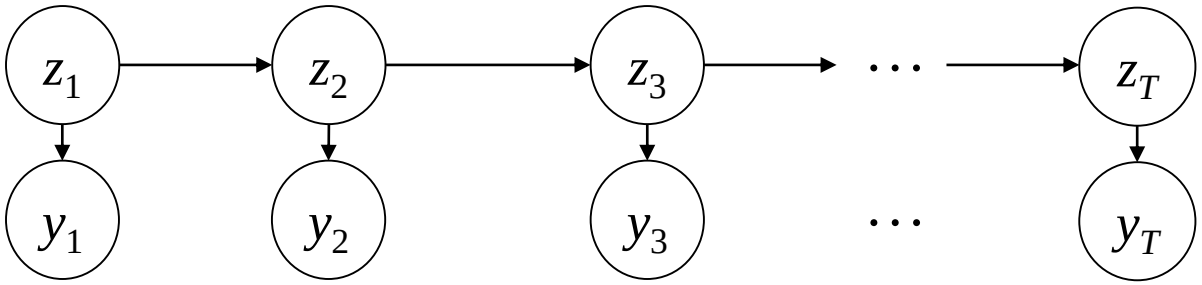
Dealing with Sequences

- **Autoregressive (AR) models**: Predict the next term (word, phrase, expression, label) in a sequence from a fixed number of previous ones using delay taps.
- **Nonlinear Autoregressive Moving Average Models (NARMA) via MLPs**: MLPs with delayed inputs and outputs generalize autoregressive models with nonlinear hidden units
- **Nonlinear State space models**

  Hidden state evolution: $\underline{z}_{t+1} = \underline{f}(\underline{z}_t, \underline{x}_t); \underline{x}_t \sim$ input features at time step t

  Observations (output): $\underline{y}_t = \underline{h}(\underline{z}_t, \underline{x}_t)$

- **Hidden Markov Models** (Lecture 12).. need large # of states to model language



Viterbi algorithm for inference and EM (Baum-Welch) algorithm for learning

$$p(z_{1:T}, y_{1:T}) = p(z_{1:T})\, p(y_{1:T} \mid z_{1:T}) = p(z_1).\left(\prod_{t=2}^{T} p(z_t \mid z_{t-1})\right).\left(\prod_{t=1}^{T} p(y_t \mid z_t)\right)$$

Dealing with Sequences: RNNs

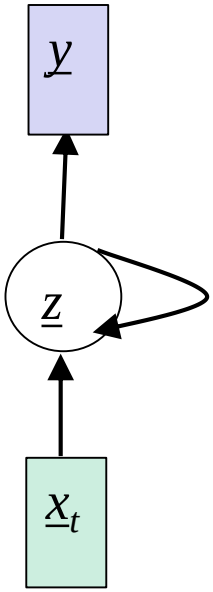
- **Recurrent Neural Networks (RNNs)**: Introduce cycles into NN graph and the notion of discrete time steps (epochs)

  Hidden state evolution: $\underline{z}_t = f_W(\underline{z}_{t-1}, \underline{x}_t)$; $\underline{x}_t \sim$ input vector at time t

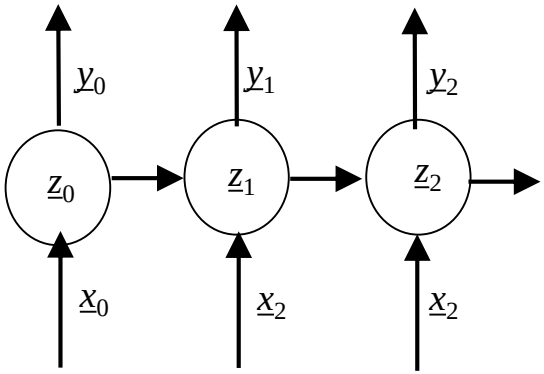
  $f_W \sim$ parameterized function..... same function at each time step t!

  $Example: \underline{z}_t = \tanh(W_{zz}\underline{z}_{t-1} + W_{zx}\underline{x}_t)$

  Observations (output): $\underline{y}_t = W_{yz}\underline{z}_t$



- RNNs can be unrolled in time as a DAG and apply Back Propagation



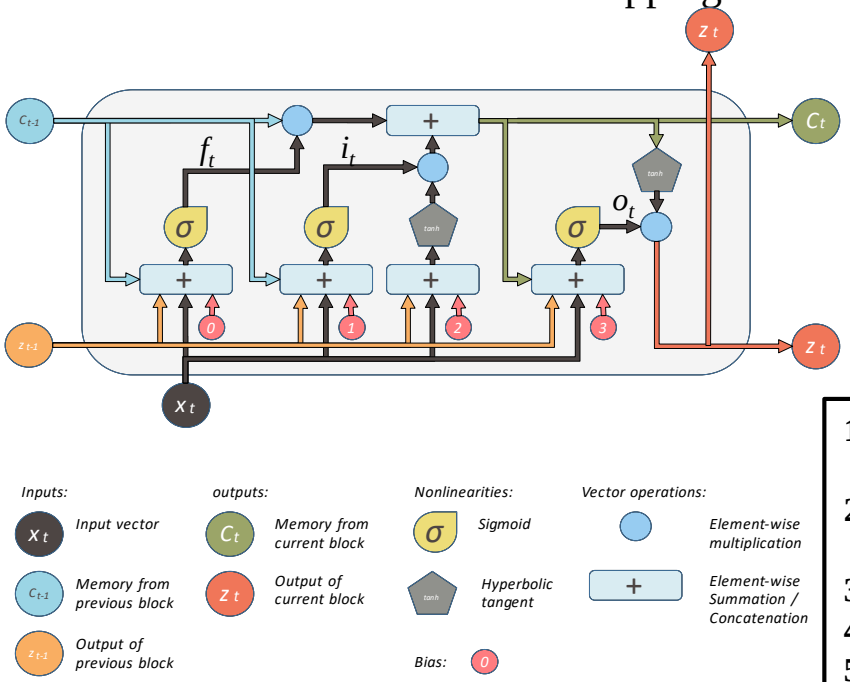
- Since W parameters are the same at each epoch, the gradients need to be added over time epochs. Recall how BP was introduced. Trivial concept if you know optimal control!
- Has exploding or vanishing gradient problem!

- How far to unroll? Typically less than 25 steps… initialize hidden variables from last unroll
- Often layers are stacked vertically (spatially) to obtain deep RNNs with different W parameters at each layer. Back propagation still works.
- Recurrent networks are flexible: image captioning (CNNs feeding into RNNs), sentiment classification, machine translation, video classification frame by frame, code generation,

Good source on RNNs: A. Graves, " Generating sequences with RNNs," https://arxiv.org/pdf/1308.0850v5.pdf

Long Short-term Memory Networks

- Trick for vanishing and exploding gradient problem: Long Short-term Memory Networks (Hochreiter and Schmidhuber, 1997).
- **Long Short-term Memory Networks (LSTMs)**: use purpose-built memory cells to store information and are better at finding and exploiting long range dependencies in data. Each LSTM block contains one or more self-connected memory cells (c) and three multiplicative units (input gate (i), output gate (o), forget/remember gate (f)). The cell remembers values over arbitrary time intervals and the three gates regulate the flow of information into and out of the cell.. It is like **ResNet** in skipping units. Back propagation still works



$$\underline{f}_t = \sigma\left(W_{fx}\underline{x}_t + W_{fz}\underline{z}_{t-1} + W_{fc}\underline{c}_{t-1} + \underline{b}_f\right)$$

$$\underline{i}_t = \sigma\left(W_{ix}\underline{x}_t + W_{iz}\underline{z}_{t-1} + W_{ic}\underline{c}_{t-1} + \underline{b}_i\right)$$

$$\underline{c}_t = \underline{f}_t \odot \underline{c}_{t-1} + \underline{i}_t \odot \tanh(W_{cx}\underline{x}_t + W_{cz}\underline{z}_{t-1})$$

$$\underline{o}_t = \sigma\left(W_{ox}\underline{x}_t + W_{oz}\underline{z}_{t-1} + W_{0c}\underline{c}_t + \underline{b}_o\right)$$

$$\underline{z}_t = \underline{o}_t \odot \tanh(\underline{c}_t); \odot = \text{Hadamard product}$$

1. **Forget gate** decides whether previous cell state should be kept or thrown out
2. **Input gate** decides what new information we will store in the cell state
3. **Cell state** equation implements 1 and 2.
4. **Output gate** decides what should be in hidden state
5. **Strength of cell state** is encoded in tanh

http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Original source on LSTMs: S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, 9(8):1735-1780, 1997.

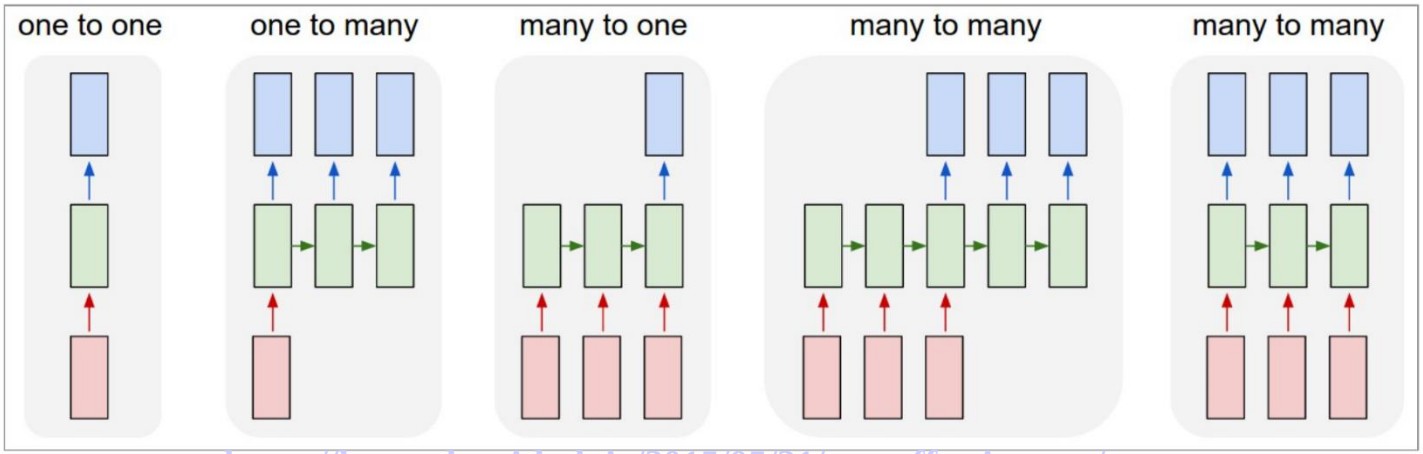
Some Practical Observations and Recent Variants

- LSTM without peepholes (i.e., where (f,i,o) gates are driven only by ($\underline{x}_t$, $\underline{z}_{t-1}$)) perform reasonably well
- Forget and output gates are salient
- Start small, calibrate and build complex network models
- Gated Recurrent Unit (GRU)
  - Merges forget and input gates into a **reset gate**
  - Merges cell state and hidden state
  - Fewer parameters and similar or better performance on some tasks

$$\underline{r}_t = \sigma\left(W_{rx}\underline{x}_t + W_{rz}\underline{z}_{t-1} + \underline{b}_f\right)$$
$$\underline{o}_t = \sigma\left(W_{ox}\underline{x}_t + W_{oz}\underline{z}_{t-1} + \underline{b}_o\right)$$
$$\underline{z}_t = (\underline{e} - \underline{o}_t) \odot \underline{z}_{t-1} + \underline{o}_t \odot \tanh(W_{zx}\underline{x}_t + W_{zz}(\underline{r}_t \odot \underline{z}_{t-1}))$$

- LSTMs and GRUs can remember sequences of 100's. For longer sequences, recent variants are attention-based encoder-decoder (RNN, LSTM) models and Causal 2D convolution networks (simultaneous source and target modeling)
- Attention-based models: e.g., Transformer : https://arxiv.org/pdf/1706.03762.pdf
  - An attention model enables representation of context (e.g., focus on relevant text or image)
- 2D Convolutional Networks: https://arxiv.org/pdf/1808.03867.pdf
  - Jointly encodes the source and target sequence in a 2D CNN
  - It models context because each layer re-encodes the input in the context of the target sequence

Applications of LSTM and Attention-based Models



https://karpathy.github.io/2015/05/21/rnn-effectiveness/

- Image classification (one-to-one): AlexNet, VGG19, VGG16, GoogleNet, ResNet,….
- Image captioning (one-to-many): https://arxiv.org/pdf/1411.4555v...
- Sentiment analysis (many-to-one): Sentence (a sequence) classified as expressing positive or negative sentiment
- Machine Translation also known as sequence-to-sequence learning (e.g., English to French Translation or many to many): https://arxiv.org/pdf/1409.3215.pdf
- Video to text (synced and sequenced input and output; many-to-many): https://arxiv.org/pdf/1505.00487...
- Hand writing generation: http://arxiv.org/pdf/1308.0850v5...
- Image generation using attention models: https://arxiv.org/pdf/1502.04623...
- Question-answering: http://www.aclweb.org/anthology/...

Summary

- Multiple Layer Perceptrons (MLPs)
- MLPs as Universal Approximators
- Back Propagation Algorithm
- Network Pruning
- Introducing Deep Networks
  - ReLU, Batch normalization, Dropout, Convolution, Max pooling, GPUs, Stochastic Optimization
- Semi-supervised Learning of MLPs
- RNNs, LSTMs, GRUs