Deep Learning Basics

ENEE4584/5584 – CV Apps in DL

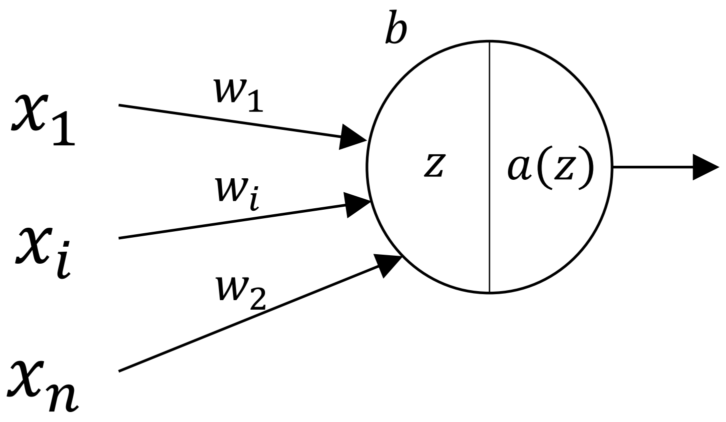
Dr. Alsamman

Artificial Neuron

- Weights: $\boldsymbol{w} = w_1, w_2, \dots, w_n$
- Bias: b
- Pre-activation: z

$$z = b + \sum_{i}^{n} w_i x_i = b + \boldsymbol{W}^{\boldsymbol{T}}\boldsymbol{X}$$

- Activation: $a(z)$
  - ~~Linear~~
  - ~~Threshold~~
  - Sigmoid
  - Tanh
  - ReLU

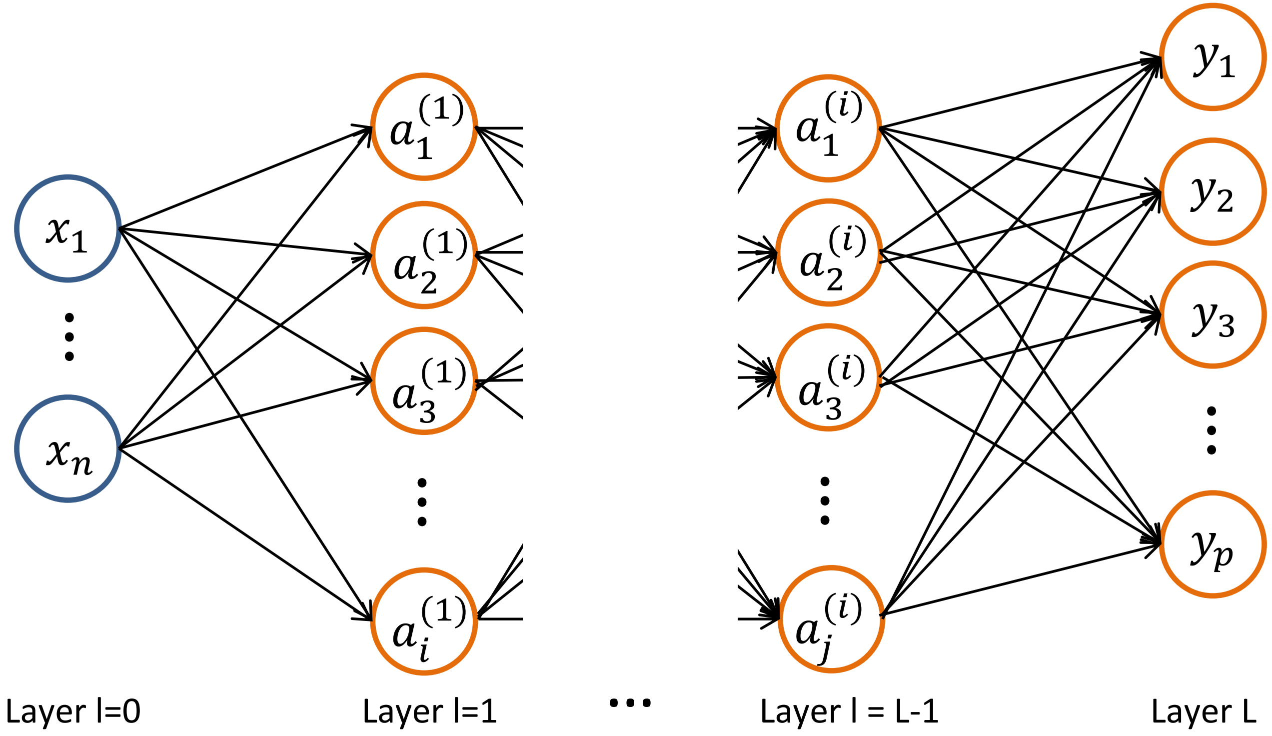


Multi-Layer Feedforward

- AKA directed acyclic graph
- AKA feedforward network
- Depth is L

$$f(\boldsymbol{X}) = f^{(L)}\left(f^{(i)}\left(\ldots\left(f^{(2)}\left(f^{(1)}(\boldsymbol{X})\right)\right)\right)\right)$$

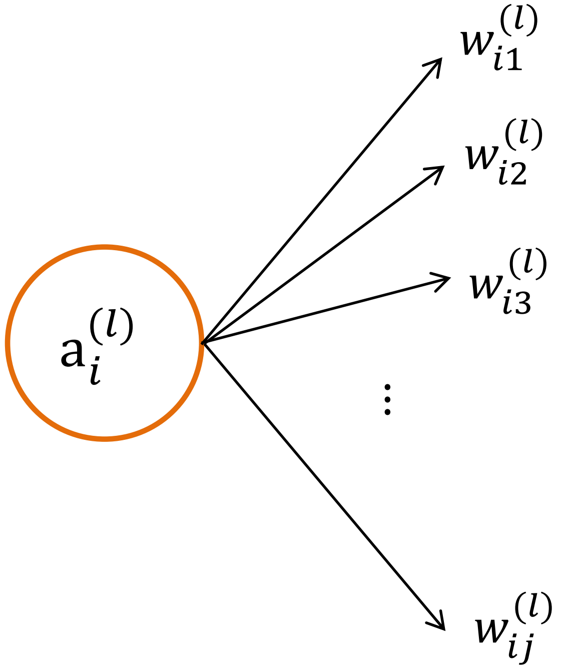
Input: $\boldsymbol{X} = \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_n \end{bmatrix}$; $a_i^{(l)}$ is the activation function



Feed-forward

- Layer l : column in network
- neuron i : row number
- j : row number of neuron in next layer
- $w_{ij}^{(l)}$: weight of connection

  between neuron i in layer l and neuron j in layer $l+1$



Feedforward Algorithm

1. $\forall\, i = 1 \to I, j = 1 \to J, l = 0 \to L$: Randomly assign weights $w_{ij}^{(l)}$
2. Initialize: $a_i^{(0)} = x_i$
3. $\forall\, l = 0 \to L$:

$$\text{compute } \boldsymbol{Z}^{(l+1)} = \left(\boldsymbol{W}^{(l)}\right)^T \boldsymbol{a}^{(l)}, \quad \boldsymbol{a}^{(l+1)} = a\left(\boldsymbol{Z}^{(l+1)}\right)$$

Objective Function

- A measurement of learning success
  - Maximization or minimization
- Depends on the learning type and learning problem
  - 2 Major types: Supervised, Unsupervised
  - 2 major applications: Regression and classification
- Supervised vs unsupervised:
  - “labeled” training data
  - Training data contains inputs matched to outputs
- Regression:
  - Real-valued output defined by the problem
  - value estimation
- Classification: aka Logistic Regression
  - Real valued output between 0 and 1 (percentage)
  - Group estimation

Objective Function: MSE

$$J(W,b) = \frac{1}{2m}\sum_{i=1}^{m}\left(\hat{y}^{(i)} - y^{(i)}\right)^2 = \frac{1}{2m}\sum_{i=1}^{m}\left(\hat{y}\left(x^{(i)};W,b\right)^{(i)} - y^{(i)}\right)^2$$

- Mean square error
- Supervised regression
- Minimization
- $\left(x^{(i)},y^{(i)}\right)$: input, output training sample i
- m: total number of samples
- $\hat{y}$: estimate, system output based on input and learning parameters W,b
- Variants: mean absolute error (MAE)

MSE Example

y	5.1	1.6	2.3	3.7
yhat	4.8	0.9	1.1	5.1
d = yhat - y	-0.3	-0.7	-1.2	1.4
d^2	0.09	0.49	1.44	1.96
mse (sum/4)	0.995			

Objective Function: CE

- Binary Cross-Entropy (BCE), log-likelihood

$$\mathcal{L}(W,b) = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)}\log\left(\hat{y}^{(i)}\right) + \left(1 - y^{(i)}\right)\log\left(1 - \hat{y}^{(i)}\right)$$

  - Classification/logistic regression
  - 2 classes (binary)
  - Minimization

- Cross-Entropy, multinomial (classes>2)

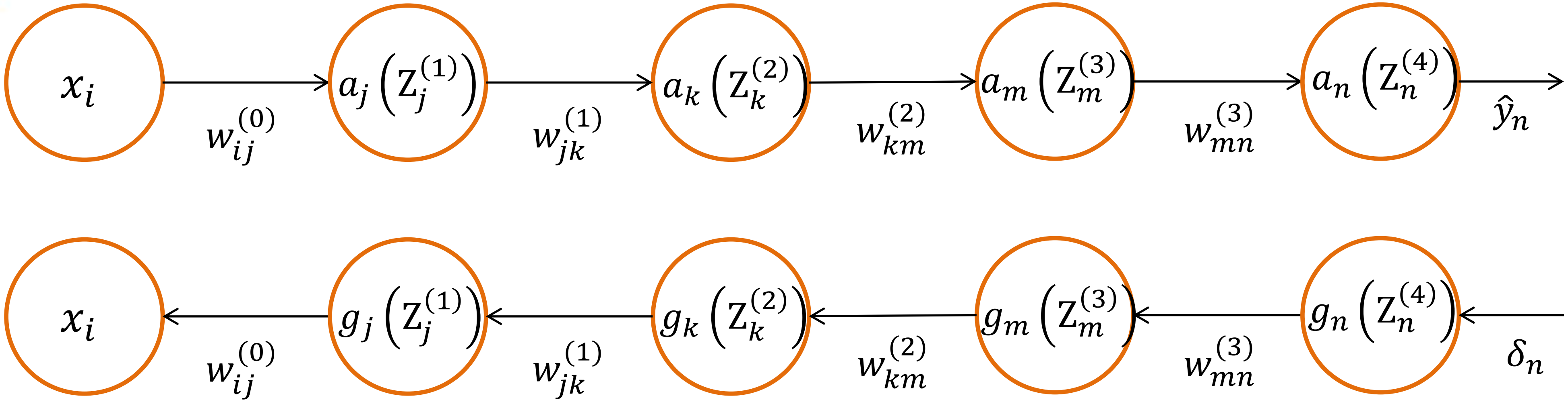
$$\mathcal{L}(W,b) = -\sum_{i=1}^{m} Y^{(i)}\log\left(\hat{Y}^{(i)}\right)$$

  - Y: multiple outputs

BCE Example

y	1	0	0	1
yhat	0.9	0.5	0.4	0.2
-log(yhat)	0.046			0.699
-log(1-yhat)		0.301	0.222	
BCE	0.317			

Backpropagation



$$\frac{\partial \mathcal{L}}{\partial w_{ij}^{(l)}} = \frac{\partial \mathcal{L}}{\partial a_i^{(l+1)}} \frac{\partial a_i^{(l+1)}}{\partial Z_i^{(l+1)}} \frac{\partial Z_i^{(l+1)}}{\partial w_{ij}^{(l)}} = \frac{\partial \mathcal{L}}{\partial a_j^{(l+1)}} g_j^{(l+1)} a_i^{(l)}$$

$$\frac{\partial \mathcal{L}}{\partial w_{ij}^{(0)}} = \delta_n \, g\left(Z_n^{(4)}\right) \; w_{mn}^{(3)} \, g\left(Z_m^{(3)}\right) \; w_{km}^{(2)} \; g\left(Z_k^{(2)}\right) \; w_{jk}^{(1)} \; g\left(Z_j^{(1)}\right) \; x_i$$

Backprop Algorithm

- Given a weight matrix, $\boldsymbol{W}^{(l)}$ and $\boldsymbol{Z}^{(l)}$ computed for each layer l in feedforward; and a cost function $\boldsymbol{\mathcal{L}}$
  1. Initialize $W^{(l)}$
  2. $\forall$ samples $\left(\boldsymbol{X}^{(m)}, \boldsymbol{Y}^{(m)}\right)$: Feedforward to generate $\boldsymbol{a}^{(l)}$
  3. Compute the gradient of each layer $\boldsymbol{g}^{(l)}$
  4. $\frac{\partial \mathcal{L}}{\partial \boldsymbol{a}^{(L)}} = \boldsymbol{\delta} = \left(\widehat{\boldsymbol{Y}} - \boldsymbol{Y}\right)$
  5. $\forall\, l = (L-1) \rightarrow 1$:

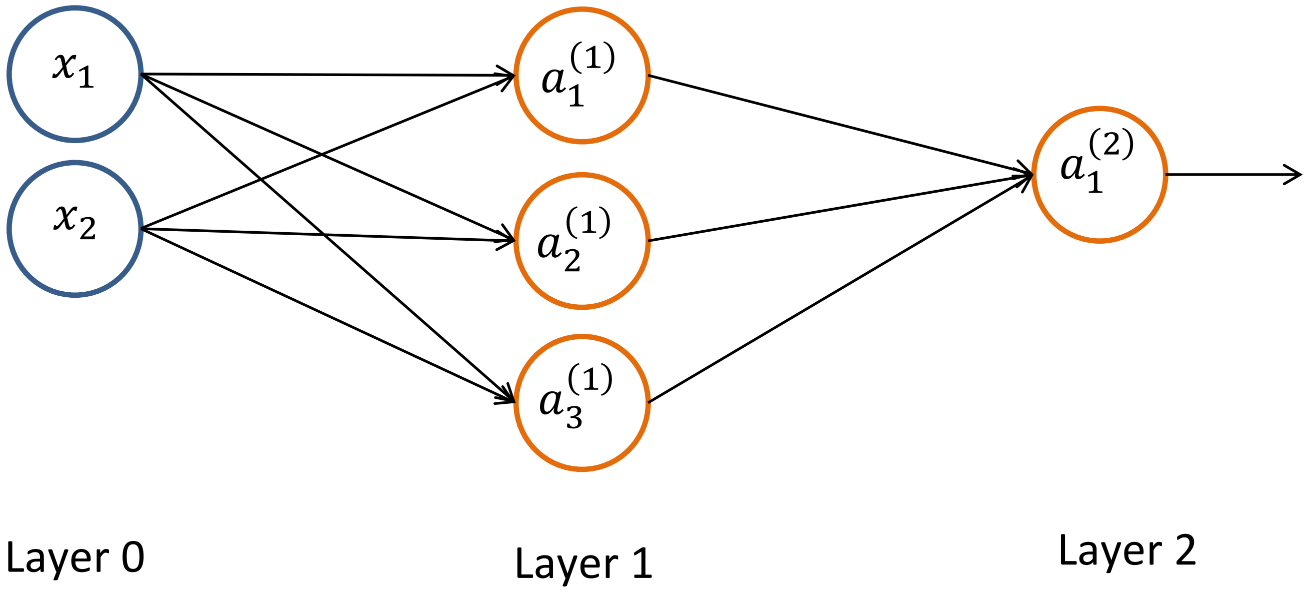
$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{a}^{(l)}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{a}^{(l+1)}} \boldsymbol{g}^{(l+1)} \boldsymbol{W}^{(l)}$$

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}^{(l)}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{a}^{(l+1)}} \boldsymbol{a}^{(l)}$$

$$\boldsymbol{W}^{(l)} = \boldsymbol{W}^{(l)} + \alpha \frac{1}{m} \frac{\partial \mathcal{L}}{\partial \boldsymbol{W}^{(l)}}$$

Backpropogation Example

- Assume
  - all activations are linear
  - $a_j^{(l+1)} = Z_{ij}^{(l+1)} = W_{ij}^{(l)} a_i^{(l)} + b_j^{(l)}$
  - $W_{ij}^{(l)}, b_j^{(l)} = 1$
  - $\{x_1, x_2; y\} = \{1,1; 0\}$
- Forward pass:
  - $a_1^{(1)}, a_2^{(1)}, a_3^{(1)} = 1 * 1 + 1 * 1 + 1 = 3$
  - $a_1^{(2)} = 3 * 1 + 3 * 1 + 3 * 1 + 1 = 10$
  - $\delta = \hat{y} - y = 10 - 0 = 10$
- Backprop:
  - $\frac{\partial L}{w_{11}^{(1)}} = \delta\, g_1^{(2)}\, a_1^{(1)} = 10 * 1 * 3 = 30$
  - $\frac{\partial L}{w_{11}^{(0)}} = \delta\, g_1^{(2)}\, w_{11}^{(1)}\, g_1^{(1)}\, x_1 = 10 * 1 * 1 * 1 * 1 = 10$



Forward Pass as Matrix

- $\boldsymbol{W}^{(0)} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix}, \boldsymbol{W}^{(1)} = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}, \boldsymbol{b}^{(0)} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \boldsymbol{b}^{(1)} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \boldsymbol{b}^{(2)} = [1]$
- $\boldsymbol{a}^{(0)} = X = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$
- $\boldsymbol{a}^{(1)} = W^{(0)}\boldsymbol{a}^{(0)} + \boldsymbol{b}^{(0)} = \begin{bmatrix} 3 \\ 3 \\ 3 \end{bmatrix}$
- $\boldsymbol{a}^{(2)} = W^{(1)}\boldsymbol{a}^{(1)} + \boldsymbol{b}^{(1)} = [10]$

Backprop as Vectors

- $\boldsymbol{g}^{(1)} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \boldsymbol{g}^{(2)} = [1]$
- $\frac{\partial L}{\boldsymbol{W}^{(1)}} = \delta \boldsymbol{g}^{(2)} \boldsymbol{a}^{(\mathbf{1})} = 10 * 1 * \begin{bmatrix} 3 \\ 3 \\ 3 \end{bmatrix} = \begin{bmatrix} 30 \\ 30 \\ 30 \end{bmatrix} = \begin{bmatrix} \partial L / w_{11}^{(1)} \\ \partial L / w_{21}^{(1)} \\ \partial L / w_{31}^{(1)} \end{bmatrix}$

Gradient Descent

$$w = w - \frac{\alpha}{m}\frac{\partial}{\partial w}J(w, b)$$

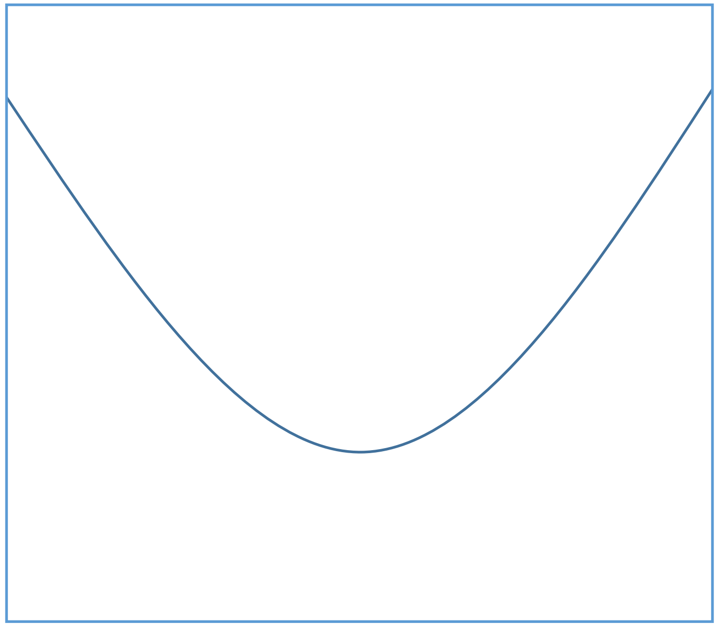
- Gradient: use derivate/slope
- Descent: make sure that with each iteration cost is decreasing
- Gradient descent algorithm:
  1. Start with initial values for W, b
  2. Compute J
  3. Update all variables **simultaneously**:

     $$w = w - \frac{\alpha}{m}\frac{\partial}{\partial w}J(w, b) \qquad\qquad b = b - \frac{\alpha}{m}\frac{\partial}{\partial b}J(w, b)$$

  4. Goto 2 until w, b converge.

Gradient Descent

- Understanding the effect of gradient descent
  - W,b: learning parameters
  - Learning rate α: hyperparameter



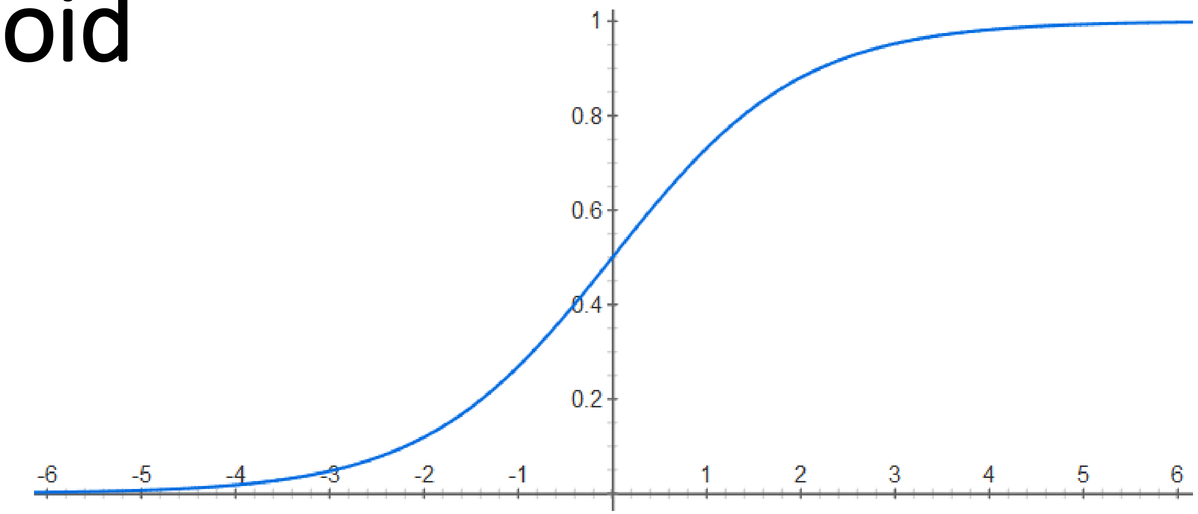
Batch, Stochastic, Minibatch

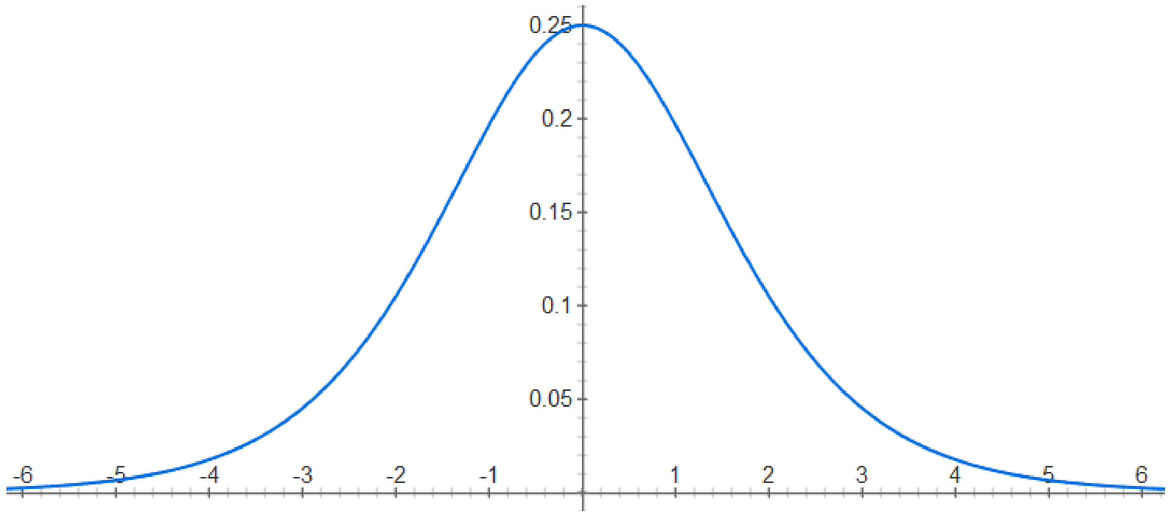
- Batch: m = entire data set
  - can be too large, slows down learning
- Stochastic: m = 1
  - Randomly selected
  - fast but susceptible to outliers
- Minibatch
  - 1 < m < total sample size
  - Randomly selected samples

To Enable Deep Learning

- Overcome overfitting
- Better activation functions
- Better weight initialization
- Better learning algorithms
- Better generalization algorithms

Sigmoid





- Relationship to step function:

$$a(z) = \lim_{k \to \infty} \left(\frac{1}{1 + e^{-kz}} \right)$$

- Probability driven
- Non-linear
- Derivative: $g(z) = a(z)\big(1 - a(z)\big)$
- Problems:
  - Gradient is small
  - Gradient slows down learning in deeper networks
  - Learning stops when |z|> 6

Tanh(z)

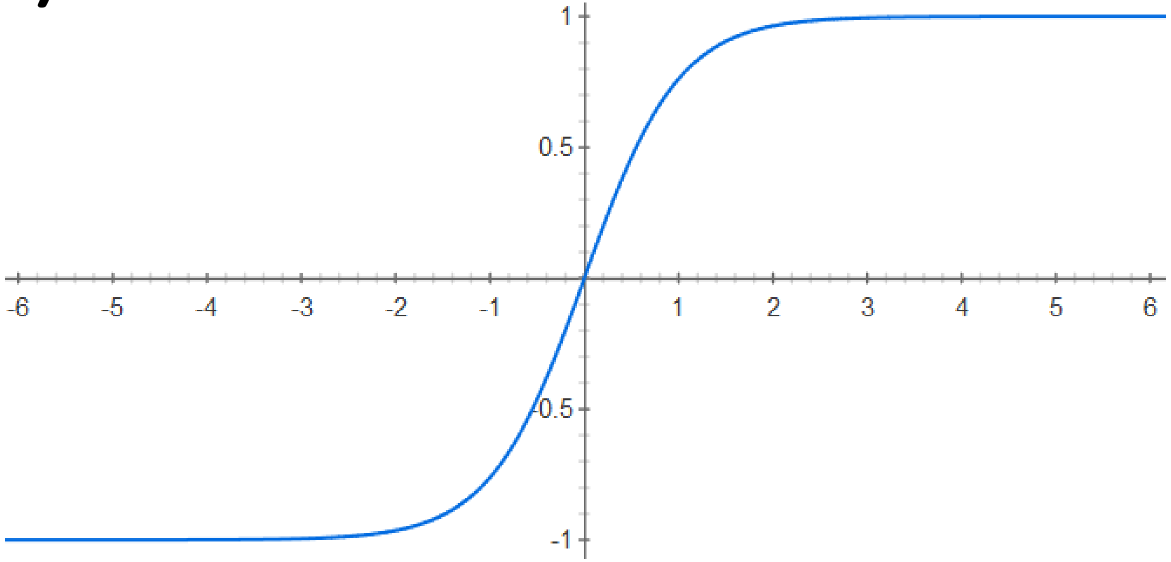
- Relationship to step function:

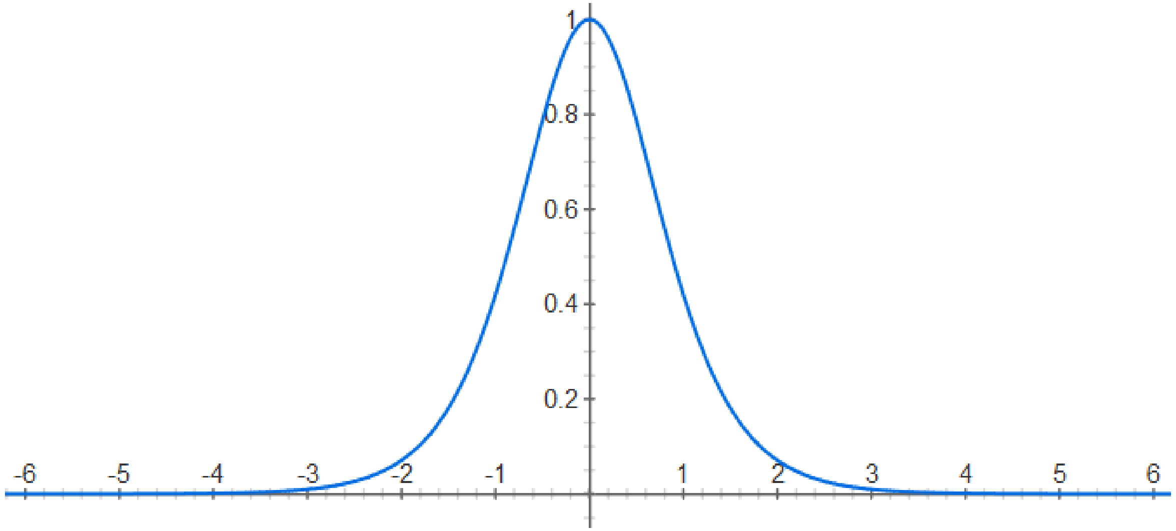
$$u(x) = \lim_{k\to\infty}\left(\frac{1+\tanh kx}{2}\right)$$

- Relationship to sigmoid:

$$\tanh(x) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}} = \frac{2}{1+e^{-2x}} - 1$$

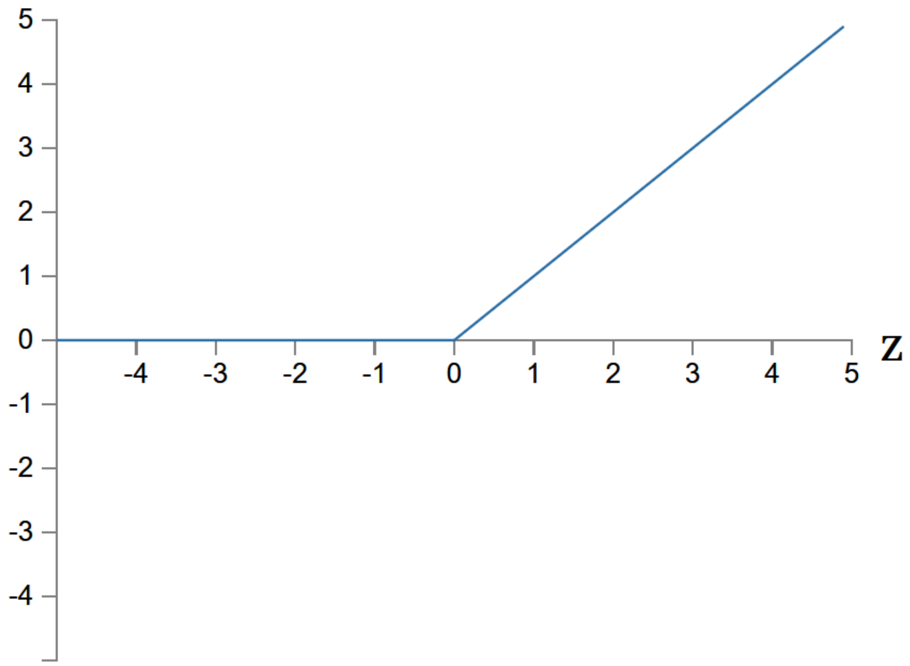
- Derivative: $g(z) \;=\; 1 - a^{2}(z)$
- “Faster” gradient
- Tristate output
- Problem: Learning stops when |z| > 3





ReLU

- Rectified linear unit activation function
  - AKA max function
  - Max{0,Z}
- No learning slowdown
- Negative Z causes output to 0
- Simple to calculate the gradient
- Preforms better than tanh and sigmoid
- No real understanding of when/why RELU are preferable
- Eliminates the need of unsupervised “pre-training” phase
- Problem: exploding Z.



ReLU Variants

- **Softplus:** $$a(z) = \ln(1 + e^z)$$
  - Derivative is sigmoid
- **Leaky ReLU:** $$a(z) = \begin{cases} z & \text{if } z > 0 \\ \beta z & \text{otherwise} \end{cases}$$
  - β a fraction < 1.
- **Noisy ReLU:** $$a(z) = \max\left(0, z + N\left(0, \sigma(z)\right)\right)$$
- **Exponential ReLU:** $$a(z) = \begin{cases} z & \text{if } z > 0 \\ \beta(e^z - 1) & \text{otherwise} \end{cases}$$
  - mean activations closer to zero which speeds up learning
  - $\beta \geq 0$ is a tuning parameter

Softmax

- Used in classification, multinomial cases
- The output is 1-hot encoded:
  - Each class gets an output
  - E.g. 3 classes = {0,1,2} => output: Y = [1,0,0] for class0; [0,1,0] for class1; [0,0,1] for class2.
- Problem with sigmoid: output for each class is independent from other outputs
- Softmax fixes the problem

$$y_i = \frac{\exp(Z_i)}{\sum_j \exp(Z_j)}$$

  - y_i: output for class i
  - Z_i: weight sum of inputs for class i output
  - Z_j: weight sum of input for all class outputs
  - E.g. For a 3 classes, to get y=[1,0,0] => $y_0 = \frac{\exp(Z_0)}{\exp(Z_0)+\exp(Z_1)+\exp(Z_2)}$, for y0>0.5 => y0 >> y1,y2

Softmax Gradient

$$\frac{\partial \mathcal{L}}{\partial w_k} = \sum_j (\hat{y}_k - \delta_{jk})\, x_k$$

https://madalinabuzau.github.io/2016/11/29/gradient-descent-on-a-softmax-cross-entropy-cost-function.html

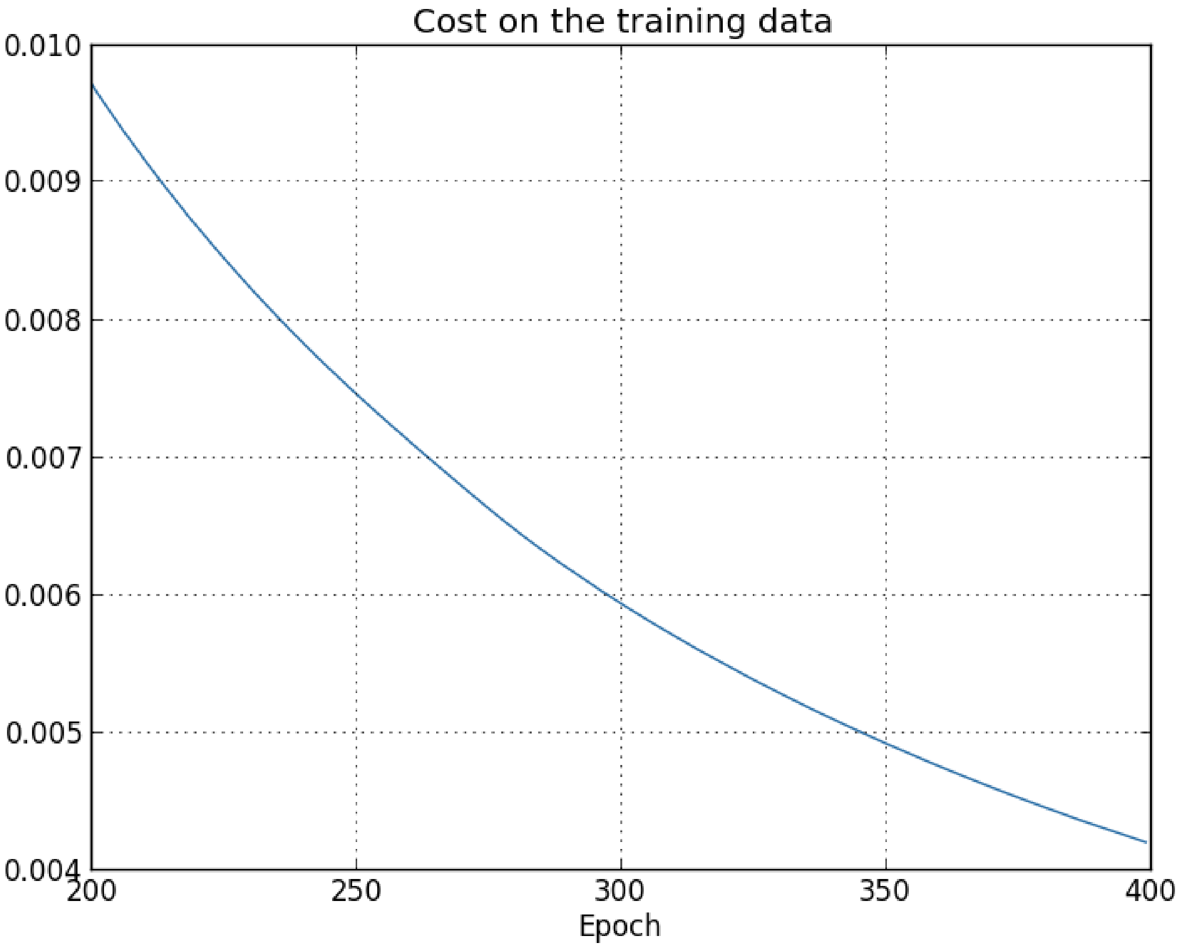
Generalization Techniques

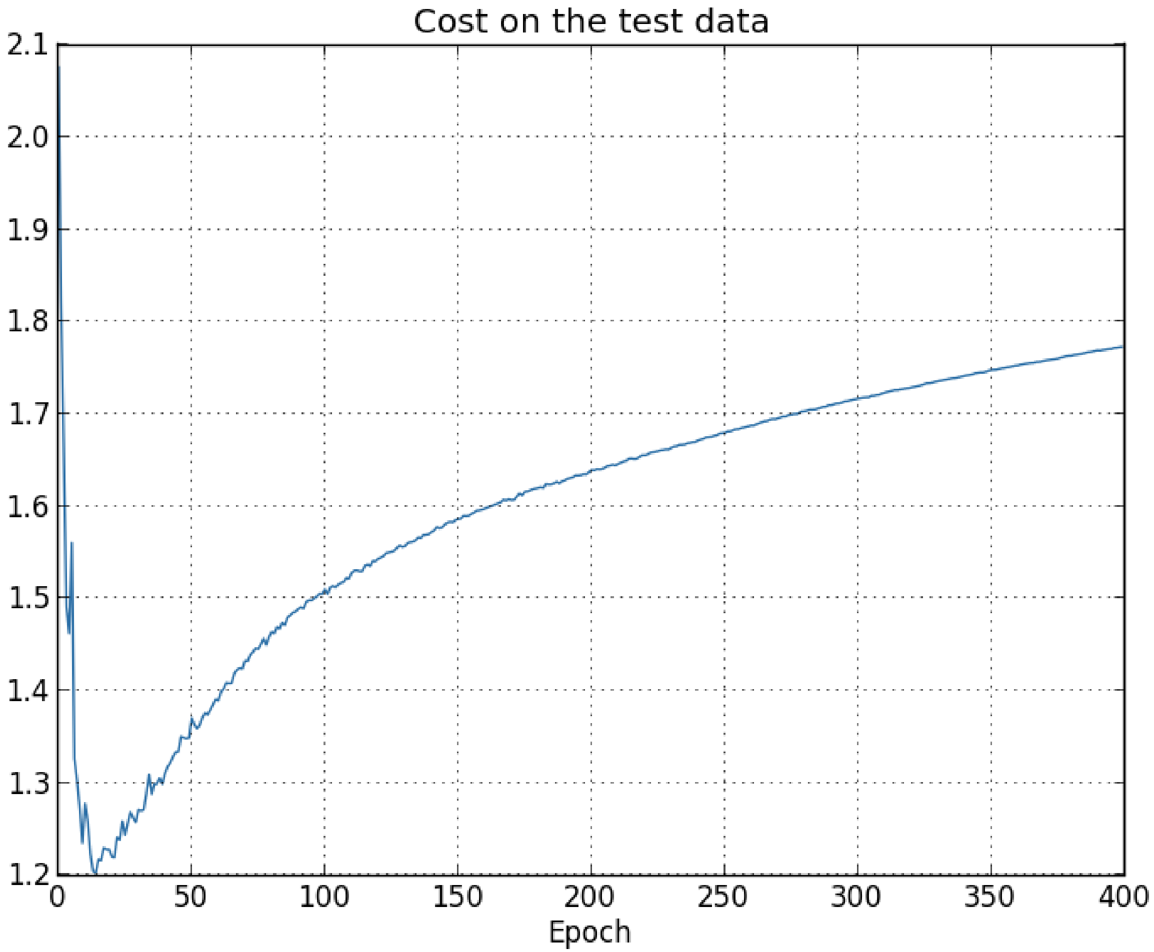
- Dataset Related:
  - Validation set for detection of overfitting
  - Larger training database
  - Data Normalization
- Parameter related:
  - Learning parameter Initialization
  - Regularization of learning parameters
    - L1 and L2
  - Learning rate: search, annealing
- Gradient related:
  - Gradient clipping
- Network related:
  - Dropout

Training Validation Split

- Training data split:
  - Training set (~80%): used for learning only
  - Testing set (~20%): used for reporting performance only
  - Randomly selected
  - Capture in the testing set the variety of input cases and outputs
- Training iterations vs epochs:
  - 1 epoch = training once for entire training set
  - Number of iterations per epoch = size of training / batch size
- Training-Validation split:
  - <20% of training data, randomly selected
  - Not used in training
  - Applied at the end of the epoch
  - Used to track performance, detecting overfitting

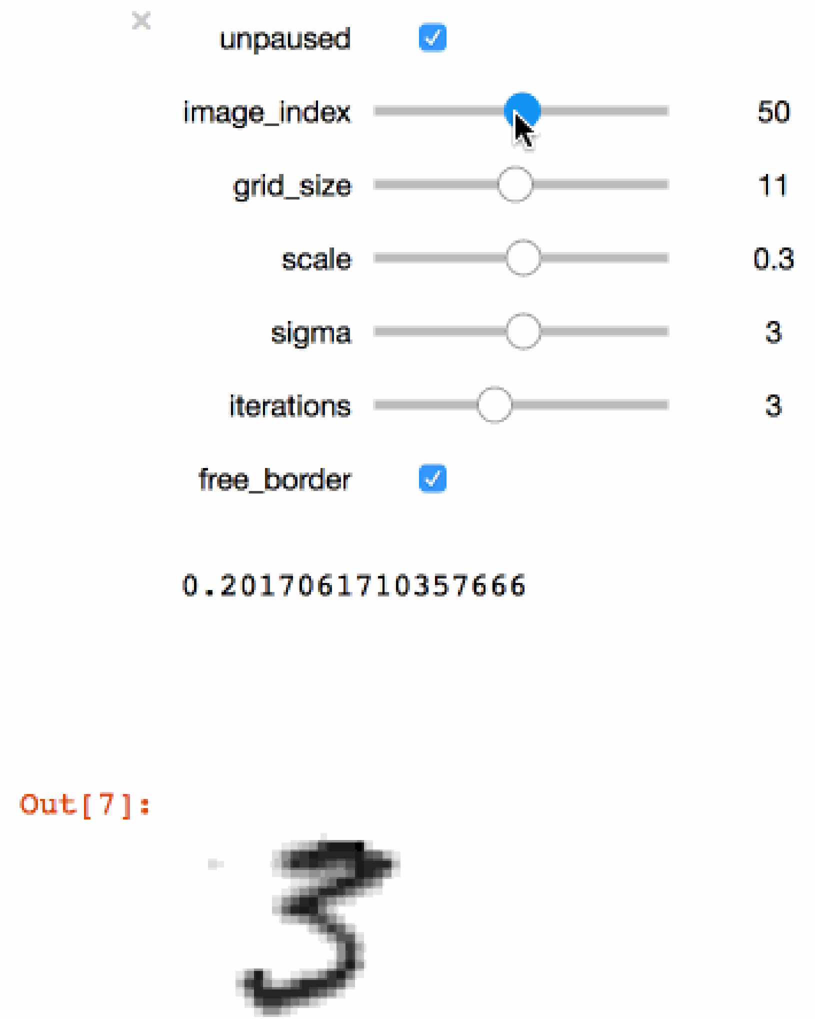
Overfitting During Training





Augmenting

- Problem with increasing training data: expensive or unavailable
- Solution: Artificial data expansion
  - Aka augmenting existing data
  - Corrupt the existing data with noise or other effects and add it to training
  - Noise must mimic RW noise
  - For images: rotate, scale, drop pixels



Learning Parameter Initialization

- Initialization affects:
  - Convergence to local or global minima
  - Speed of convergence
  - Generalization error
- Modern schemes focus on heuristics and simplicity
- Goal: independence & variance
  - Maintain independence between neurons
  - “Break symmetry” between different units
    - Symmetry leads to duplication of neurons
  - Conserve variance between layers

Initialization Schemes

- Xavier Glorot & Bengio initialization (aka Glorot, Xavier) :
  - sigmoid:

$$W \sim U\left[-\sqrt{\frac{6}{n_{in}+n_{out}}}\,,\sqrt{\frac{6}{n_{in}+n_{out}}}\right]$$

  - tanh:

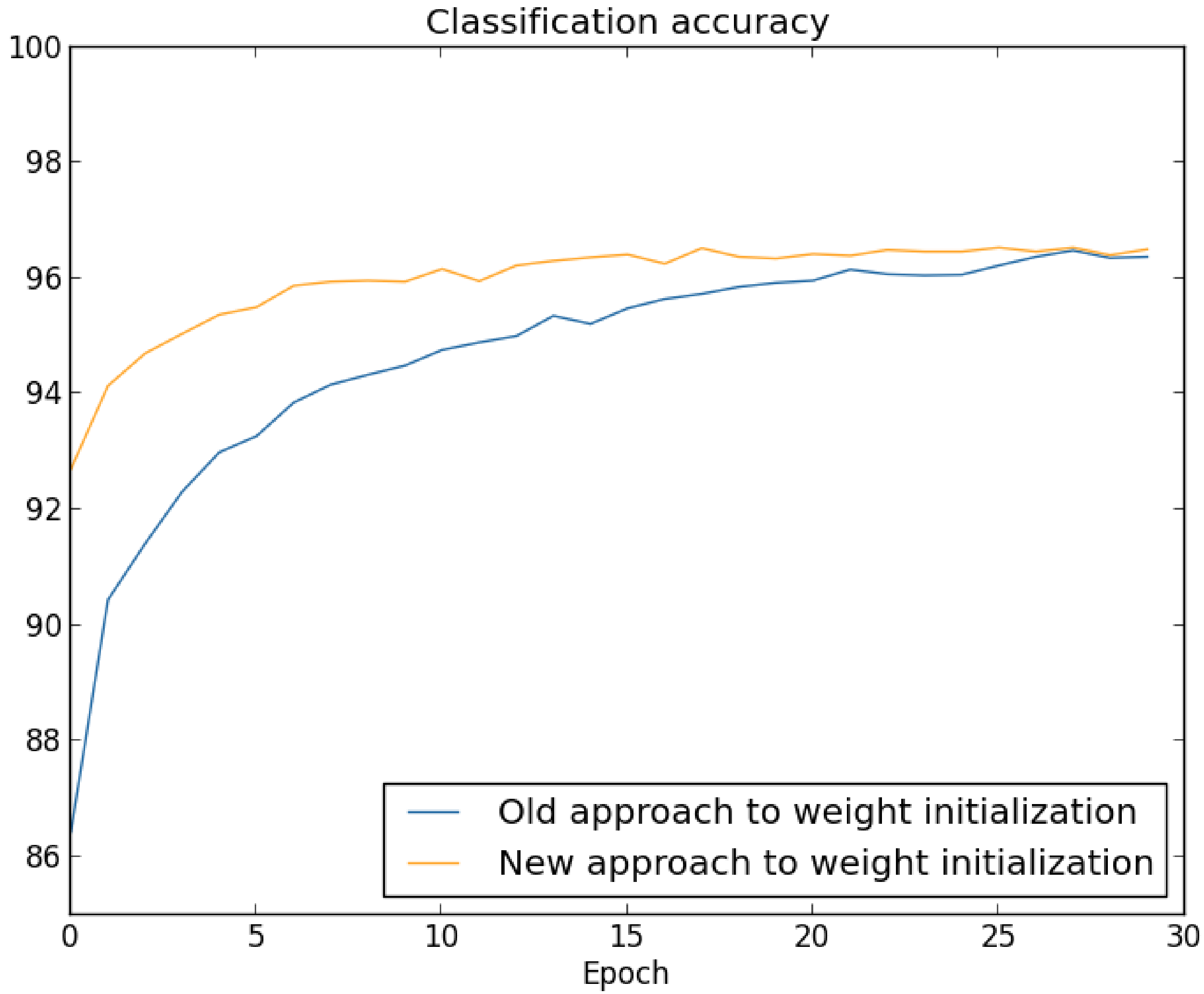
$$W \sim U\left[-4\sqrt{\frac{6}{n_{in}+n_{out}}},4\sqrt{\frac{6}{n_{in}+n_{out}}}\right]$$

  - ReLU:

$$W \sim U\left[-\sqrt{\frac{6}{n_{in}}},\sqrt{\frac{6}{n_{in}}}\right]$$

- Kiaming He initialization (aka He):
  - Xavier's derived for activation function is linear
  - $W \sim N\left[\mu = 0\,,\sigma = \sqrt{\frac{2}{n_{in}}}\right]$

- $U[\quad]$: uniform distribution
- n_{in}, n_{out}: number of neurons in input, output
- $N[\quad]$: normal distribution


Classification accuracy
100
98
96
94
92
90
88
86
0
5
10
15
20
25
30
Epoch
Old approach to weight initialization
New approach to weight initialization

Data Normalization

- Protect against outlier data
- Typical normalization: controls the mean, std dev of the data
- Can be applied to
  - Entire dataset
  - Mini batch
- Mini-batch normalization can be applied to any dimension
- The mean and std dev calculated and applied, or learnt

L2 Regularization Parameter

- Most popular in ML
- Force a weight decay
  - Keep weights from increasing uncontrollably
  - Sigmoid-like functions: Large weights => zero gradients
- Reformulate cost function:

$$C = \sum_i C_i + \frac{\lambda}{2m} \sum_{ijl} \left(\boldsymbol{w}_{\boldsymbol{ij}}^{(\boldsymbol{l})} \right)^{\mathbf{2}}$$

  - C_i is the cost function: cross-entropy/log-likelihood/loss or mean square error
  - λ: regularization parameter
  - Doesn’t affect bias
- Learning objective always to reduce cost (error)
  - w^2 punishes large weights

L2 Regularized Backprop

- L2 regularized cost function

$$C = \sum_{i} C_i + \frac{\lambda}{2m} \sum_{\boldsymbol{ijl}} \left(\boldsymbol{w_{ij}^{(l)}}\right)^{\mathbf{2}}$$

- Gradient descent

$$w_{ij}^{(l)} = w_{ij}^{(l)} - \alpha \frac{\partial C_i}{\partial w_{ij}^{(l)}} - \alpha \frac{\lambda}{m} w_{ij}^{(l)} = \left(\mathbf{1} - \boldsymbol{\alpha} \frac{\boldsymbol{\lambda}}{\boldsymbol{m}}\right) w_{ij}^{(l)} - \frac{\alpha}{m} \frac{\partial C_i}{\partial w_{ij}^{(l)}}$$

L1 Regularization

- Penalizes large weights

$$C = \sum_i C_i + \frac{\lambda}{m} \sum_{\boldsymbol{ijl}} \left| \boldsymbol{w}_{\boldsymbol{ij}}^{(\boldsymbol{l})} \right|$$

  - L2 rewards fractional weights, L1 doesn't
    - L2 reduces weights towards 1
    - L1 reduces weights towards 0
  - L2 small decrements in weights lead to great reduction in C.
    - L1 large decrements cause large reductions in C
- L1 few weights survive. Most weights are close to 0.
  - concentrate the weights in a relatively small number of connection

L1 Regularized Backprop

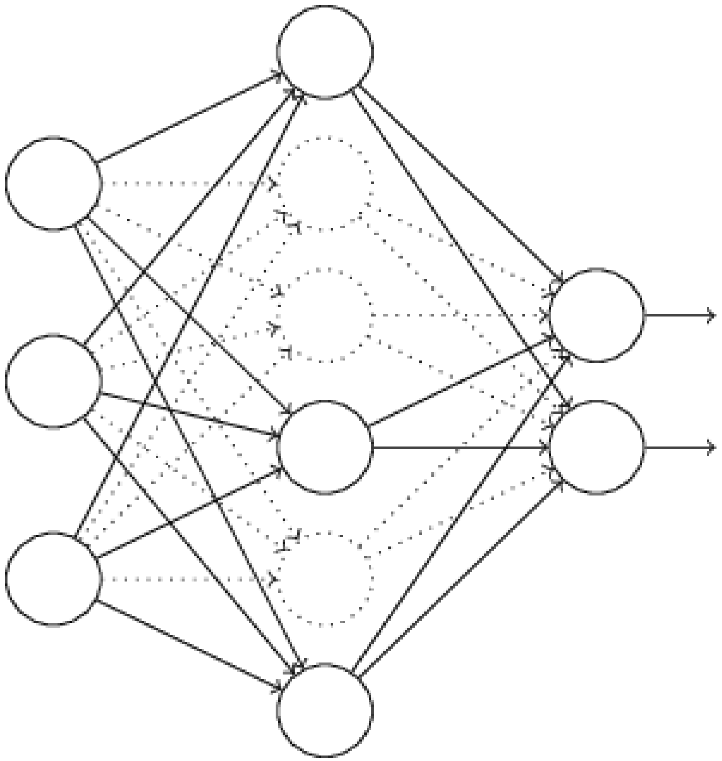
- Learning:

$$w_{ij}^{(l)} = w_{ij}^{(l)} - \alpha \frac{\partial C_i}{\partial w_{ij}^{(l)}} + \boldsymbol{\alpha} \frac{\boldsymbol{\lambda}}{\boldsymbol{m}} \mathbf{sgn}\left(\boldsymbol{w}_{\boldsymbol{ij}}^{(\boldsymbol{l})}\right) = w_{ij}^{(l)} - \frac{\alpha}{m} \frac{\partial C_i}{\partial w_{ij}^{(l)}} \pm \boldsymbol{\alpha} \frac{\boldsymbol{\lambda}}{\boldsymbol{m}}$$

- sgn() is the sign function
- Not defined at $w = 0$

Dropout

- Applied to the network not backprop
  - Easier on computation
- Strategy:
  - Randomly delete a percentage of the hidden neurons in each epoch
    - Not input or output neurons
  - Learn weights and biases
  - Repeat
  - When done, decrease weights and biases by percentage
- Why does it work?
  - Averaging
  - Reduction of co-dependence of neurons
  - Similar to bagging: multiple classifiers, averaged output



Dropout Alternatives

- Zoneout
  - RNN
  - Randomly chosen units remain unchanged across a time transition
- Dropconnect
  - Drop individual connections, instead of nodes
- Shakeout
  - Scale up the weights of randomly selected weights
    - $w = \alpha w + (1 - \alpha)\ c$
  - Fix remaining weights to a negative constant
    - $w = -c$
- Whiteout
  - Add or multiply weight-dependent Gaussian noise to the signal on each connection

Better Learning Functions

- SGD assumes convex cost function which is overly simplistic
- Alternative learners:
  - Momentum
  - Adaptive gradient
  - 2nd order gradients

Momentum Learning

- Introduce a velocity parameter, V, for each, W, such that:

$$W = W + V$$
$$V = \beta V - \frac{\alpha}{m} g$$

- β is friction, β=1 no friction (no slowing down)
- Scheme:
  - Initialize $w, v = 0, \alpha, \beta$
  - While not (stopping criterion):
    - Given $\{X, Y\}$ feedforward
    - Compute gradient: $g = \frac{\partial \mathcal{L}}{\partial W}$
    - Compute velocity: $V = \beta V - \frac{\alpha}{m} g$
    - Update weights: $W = W + V$
- Nestrov momentum:
  - Initialize $W, V = 0, \alpha, \beta$
  - While not (stopping criterion):
    - Given $\{X, Y\}$ feedforward
    - Update weights: $W = W + V$
    - Compute gradient: $g = \frac{\partial \mathcal{L}}{\partial W}$
    - Compute velocity: $V = \beta V - \frac{\alpha}{m} g$

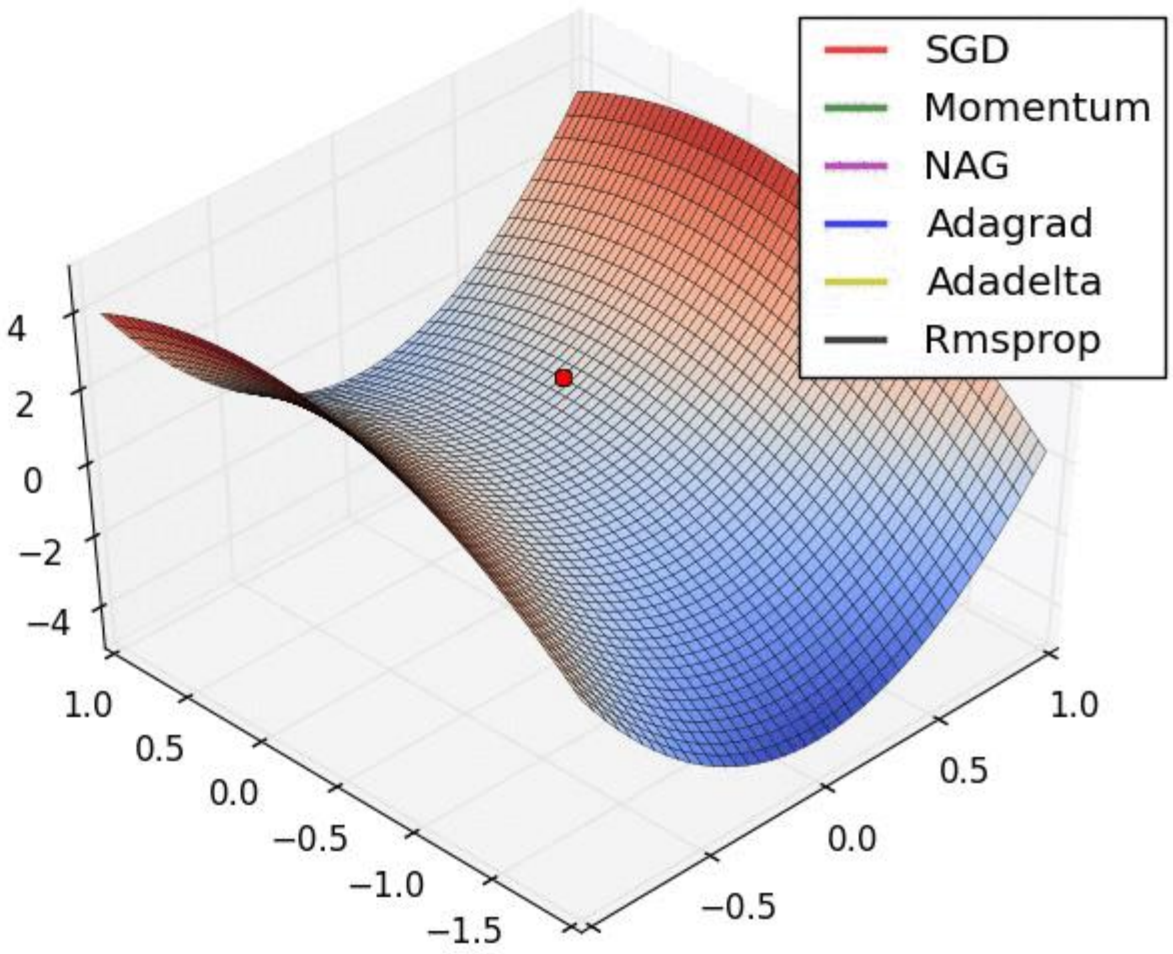
Adaptive Gradient

- Adagrad: $r = r + g^2;$ $W = W - \frac{\alpha}{m}\frac{g}{\sqrt{r+\gamma}}$
- RMSprop: $r = \rho r + (1-\rho)g^2;$ $W = W - \frac{\alpha}{m}\frac{g}{\sqrt{r+\gamma}}$
- Adam: $s = \frac{\rho_1 s+(1-\rho_1)g}{\rho_1};$

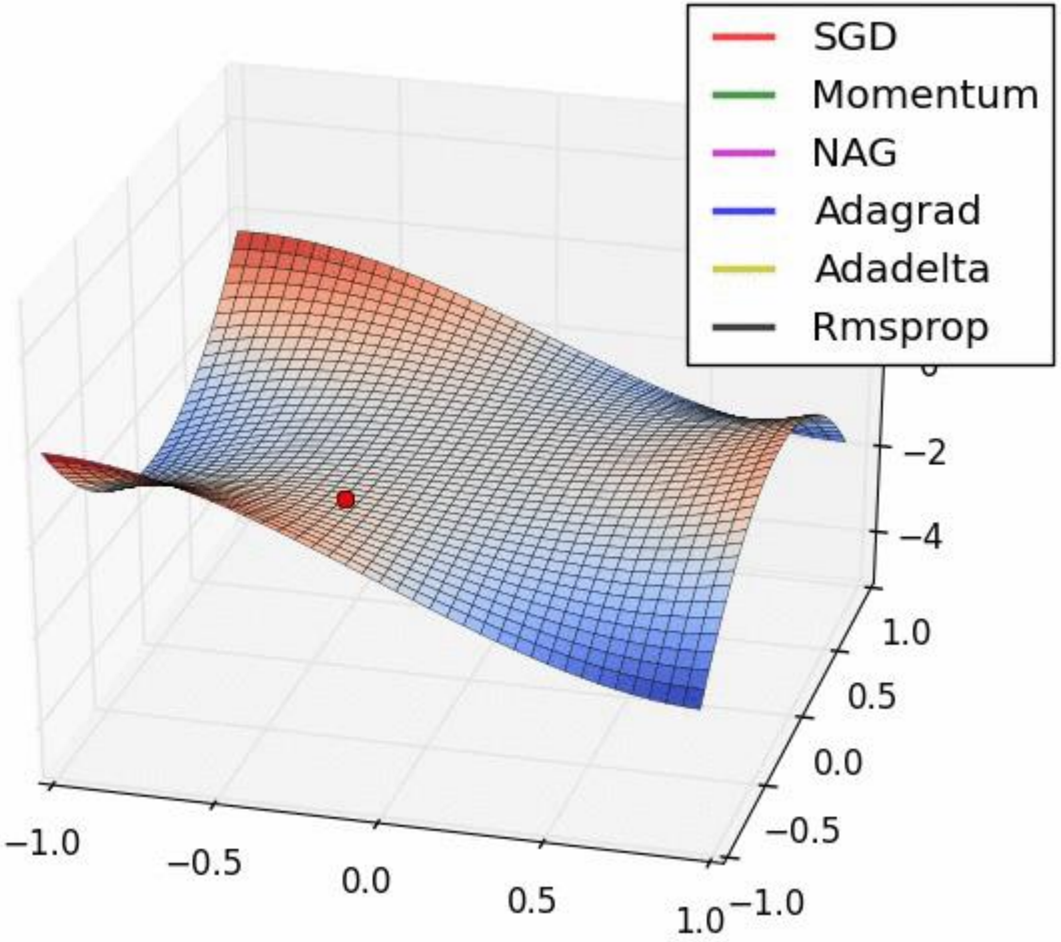
$$r = \frac{\rho_2 r+(1-\rho_2)g^2}{\rho_2}; \qquad W = W - \frac{\alpha}{m}\frac{s}{\sqrt{r+\gamma}}$$

$$0 < \rho, \rho_1 \rho_1 < 1$$

Saddle Points



adventuresinmachinelearning.com



adventuresinmachinelearning.com