Deep Learning

ENEE 4583/5583 Deep Learning

Dr. Alsamman

Slide Credits: Deep Learning Book, B. Raj, M. Nielsen, A. Radford, adventuresinmachinelearning.com

Improvement to Deep Learning

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

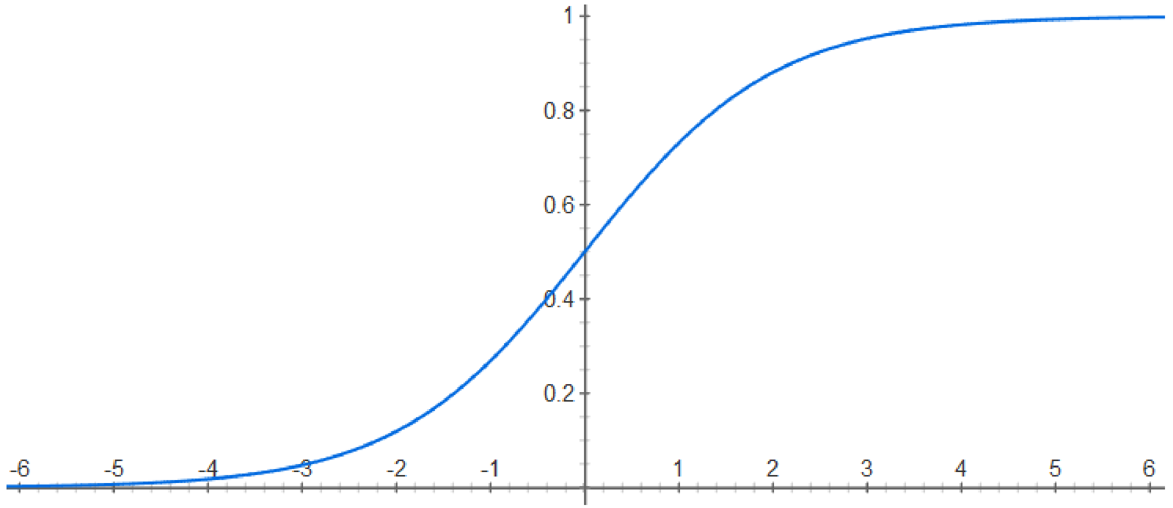
Better Activation

Deep Learning Problem: Gradient

- Learning slowdown
  - In early layers (close to input)
- Gradient instability
  - Vanishing gradient
  - Exploding Z

Backprop Problems: Activations

- No zero centered!
- Negative direction switches off output
- Convergence issues
- Gradient has a max of 0.25



Tanh(z)

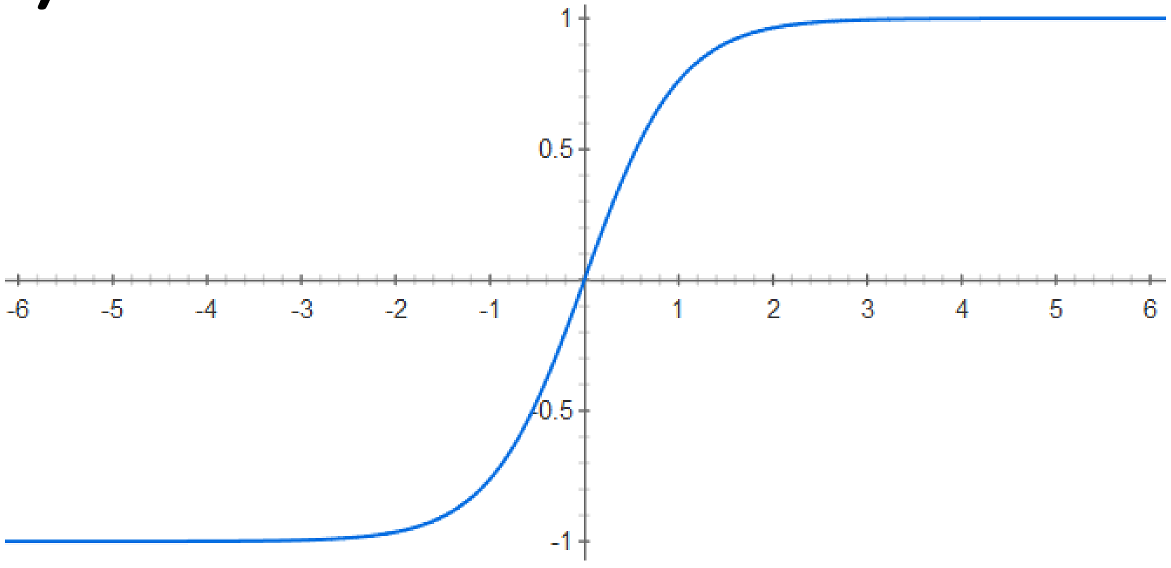
- Relationship to step function:

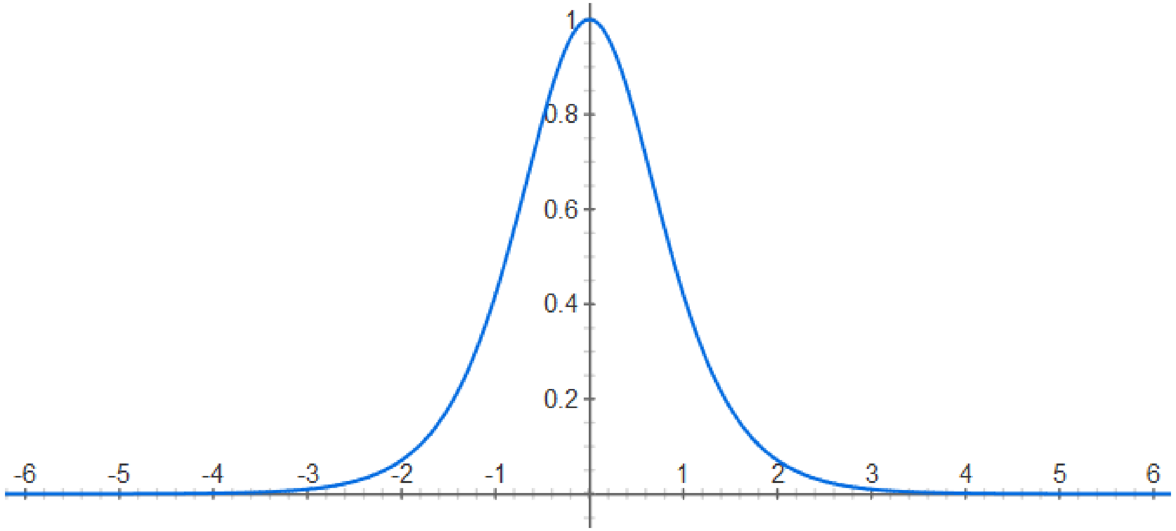
$$u(x) = \lim_{k \to \infty} \left(\frac{1 + \tanh kx}{2} \right)$$

- Relationship to sigmoid:

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

- Derivative: $g'(z) \ = \ 1 - g^2(z)$

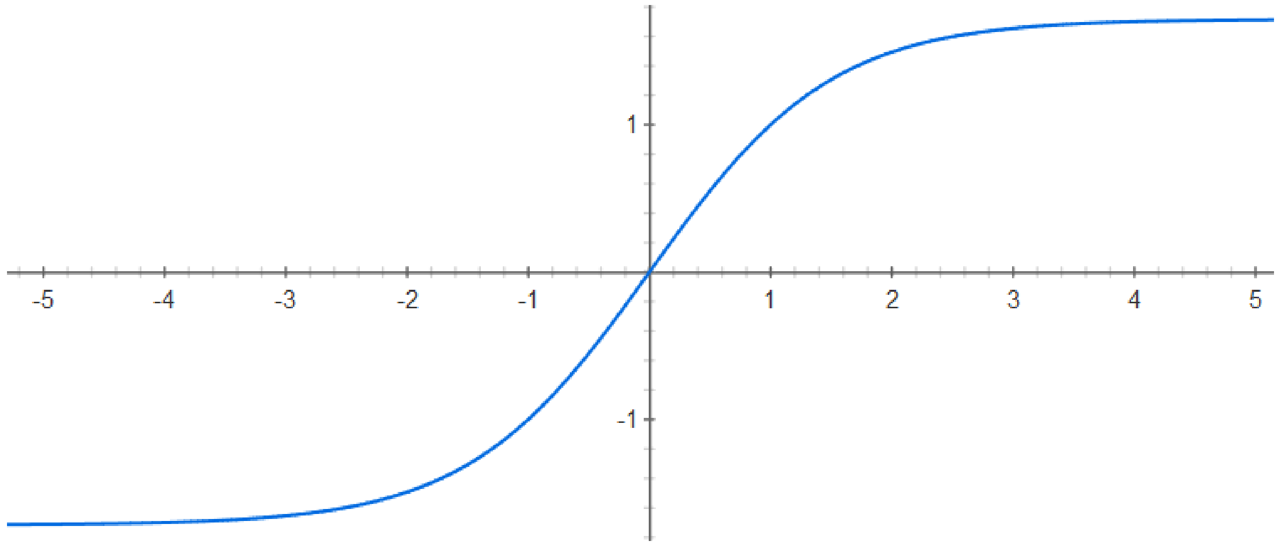




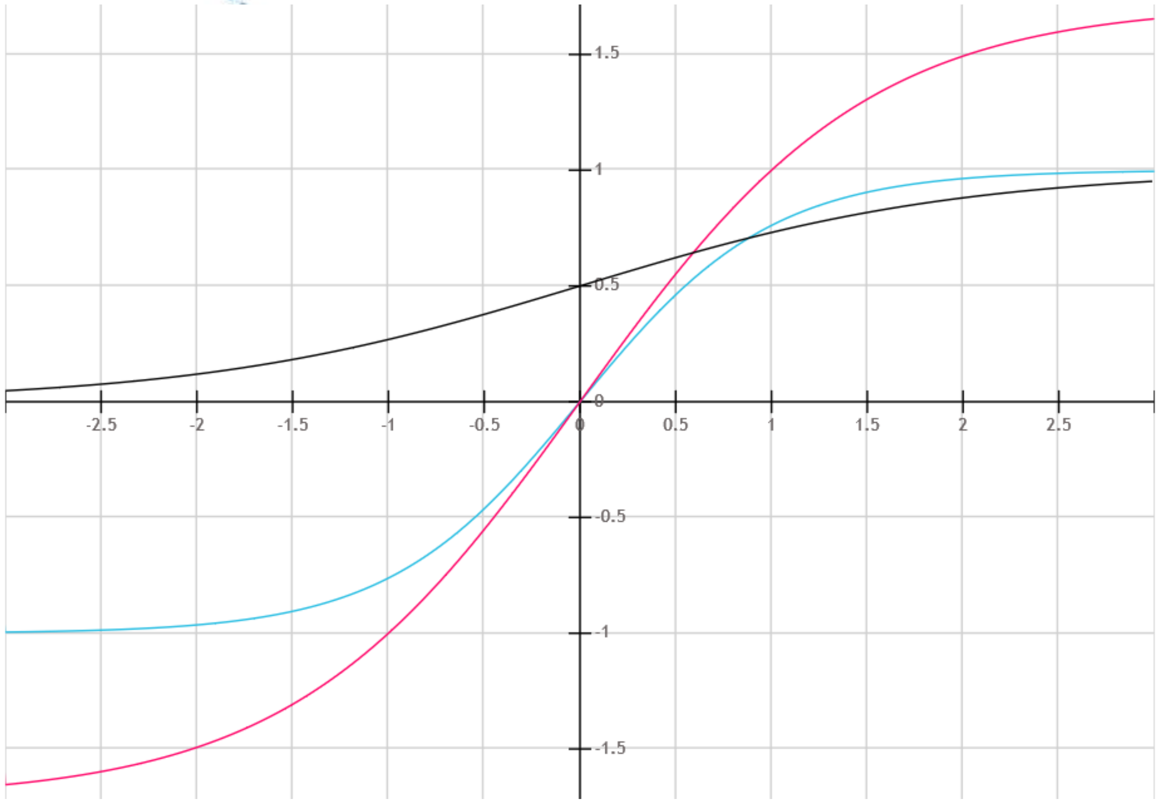
Backprop Problems: Activations

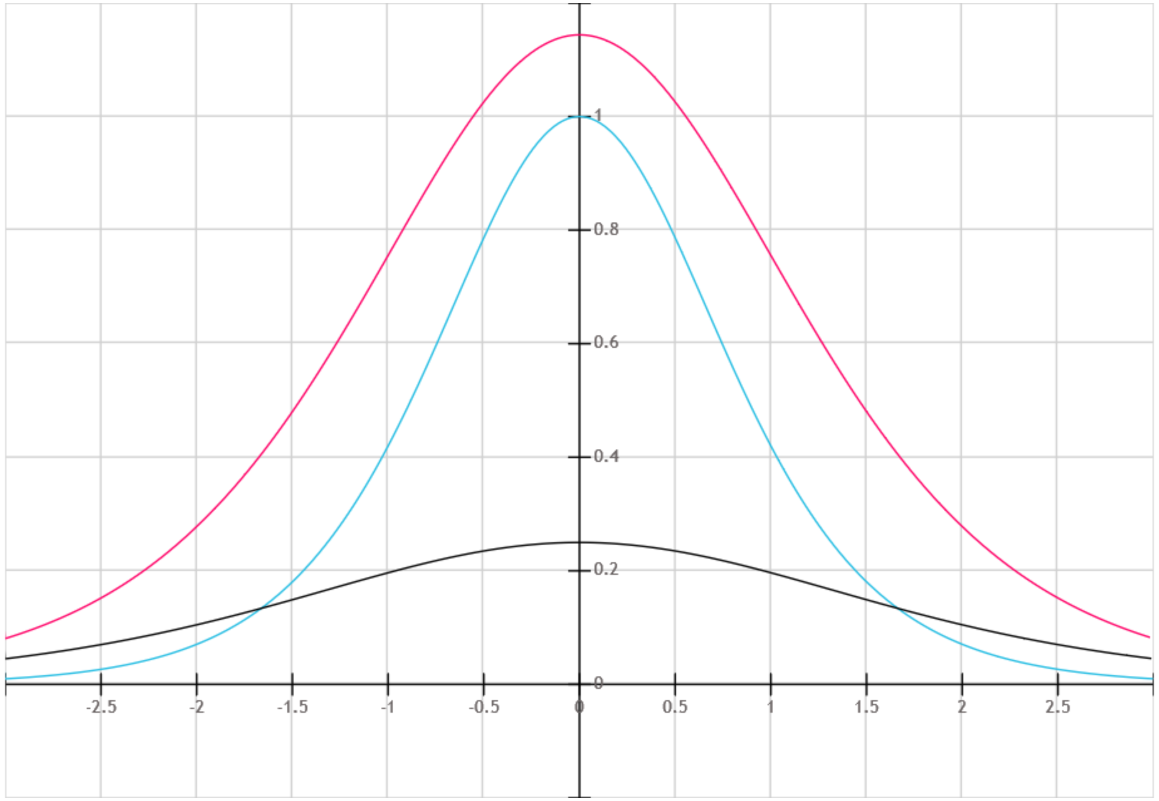
- Tanh is balanced
- Slope is twice steeper than sigmoid
- Pushes output to -1, 1 faster
- Solution: Lecun's optimized

$$1.7159 \tanh\left(\frac{2}{3}z\right)$$



Comparison





Functions

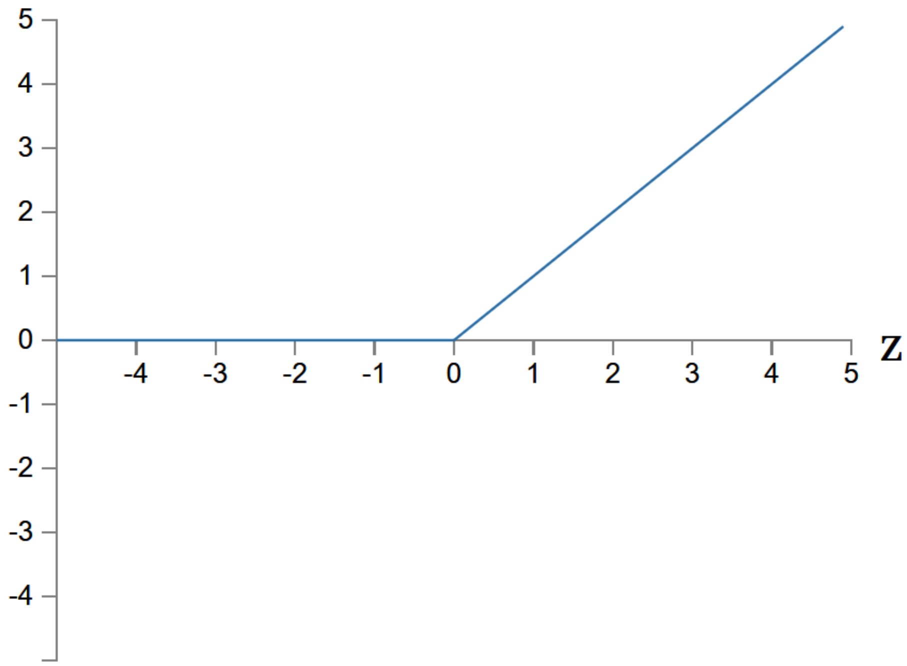
Sigmoid Tanh Lecun's Tanh

Derivative

Sigmoid Tanh Lecun's Tanh

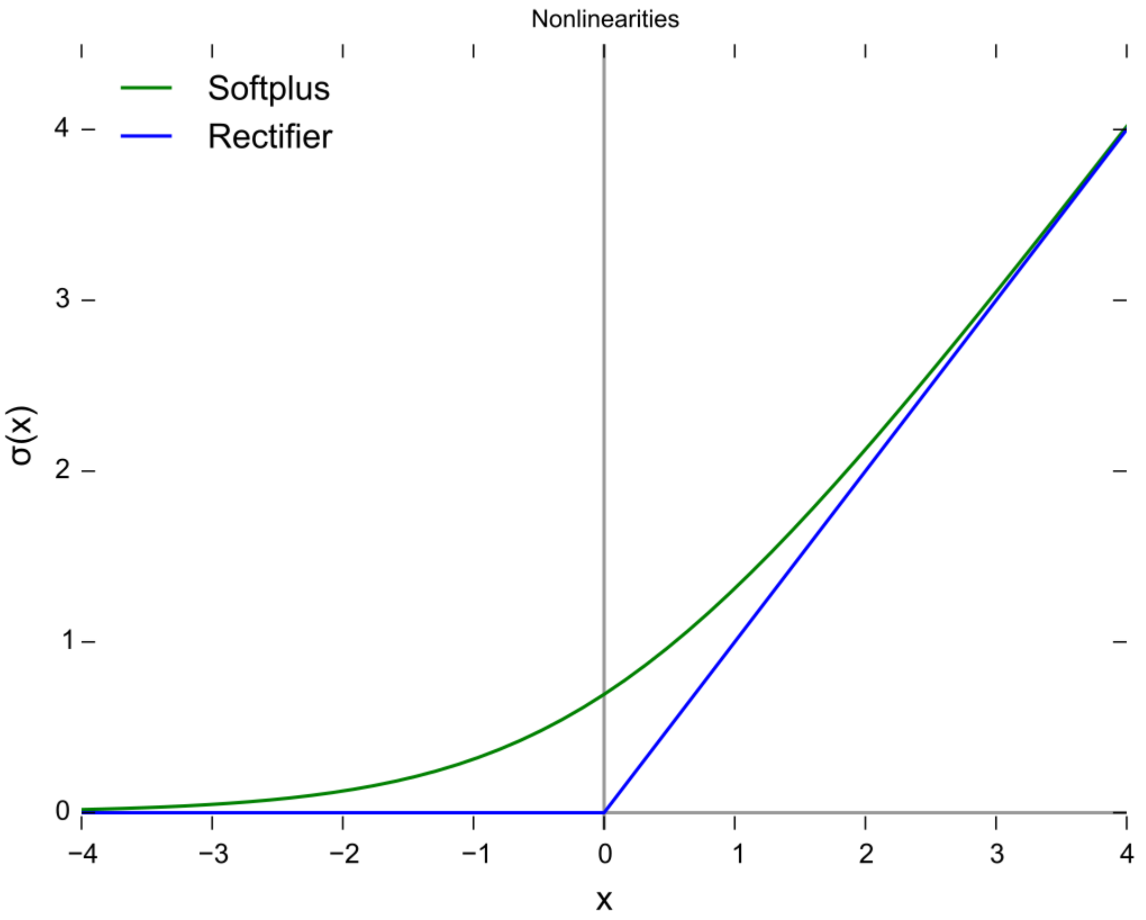
ReLU

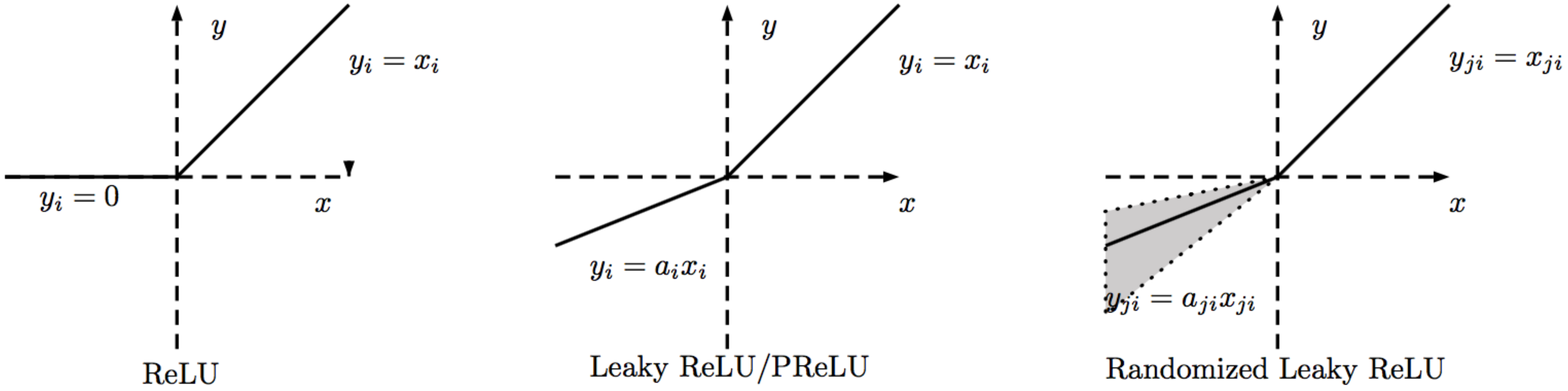
- Rectified linear unit activation function
  - AKA max function
  - Max{0,Z}
- No learning slowdown
- Negative Z causes output to 0
- Simple to calculate the gradient
- Preforms better than tanh and sigmoid
- No real understanding of when/why RELU are preferable
- Eliminates the need of unsupervised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

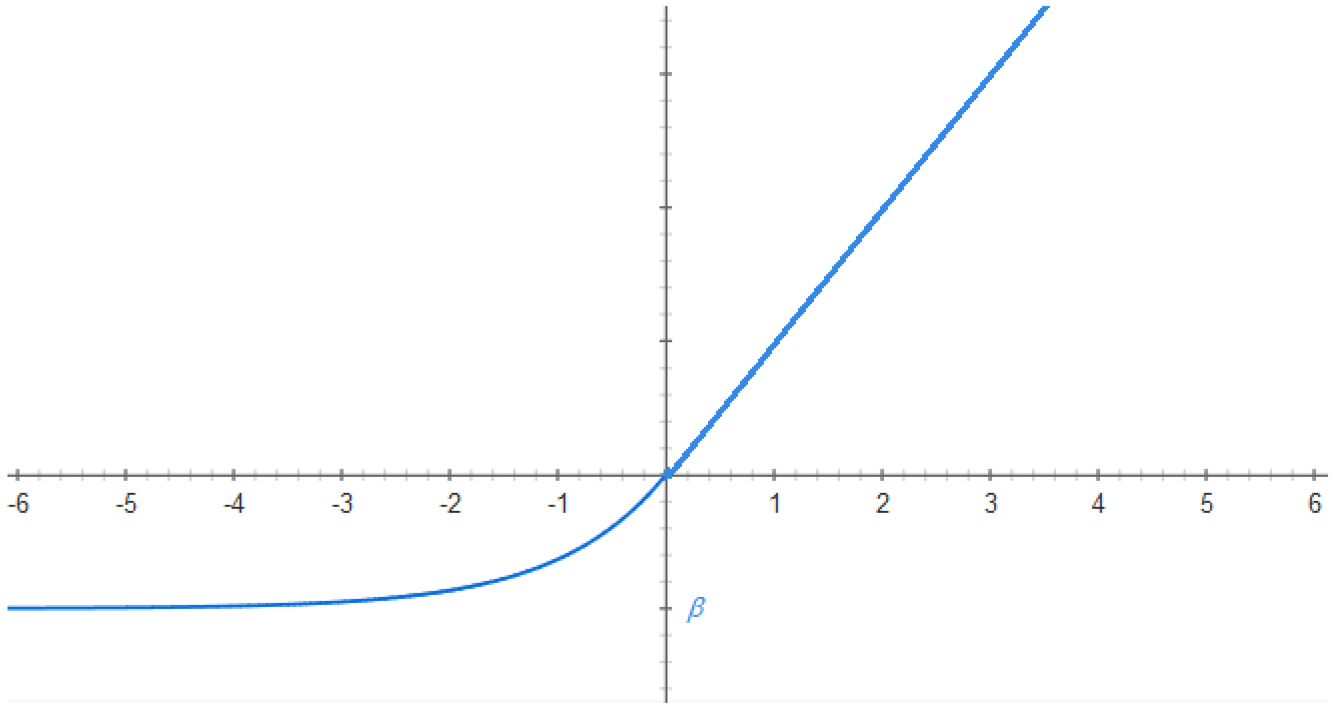


ReLU Variants

- 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: $$a(z) = \max\left(0, z + N\big(0, \sigma(z)\big)\right)$$
- 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


Nonlinearities
Softplus
Rectifier
σ(x)
x
0
1
2
3
4
−4
−3
−2
−1
0
1
2
3
4


y
$y_i = x_i$
$y_i = 0$
x
ReLU
y
$y_i = x_i$
x
$y_i = a_i x_i$
Leaky ReLU/PReLU
y
$y_{ji} = x_{ji}$
x
$y_{ji} = a_{ji} x_{ji}$
Randomized Leaky ReLU


-6
-5
-4
-3
-2
-1
1
2
3
4
5
6
β

Initialization

Deep Learning Problems: Initialization

- Initialization affects:
  - Convergence to local or global minima
  - Speed of convergence
  - Generalization error
- Initialization similar to imposing a Gaussian prior, p(w)
- Modern schemes focus on heuristics and simplicity
- Spending time/computation power on initialization is sometime less costly than validation step
- Major Goal: “Break symmetry” between different units
  - Symmetry leads to duplication of neurons

Unsupervised Pre-training

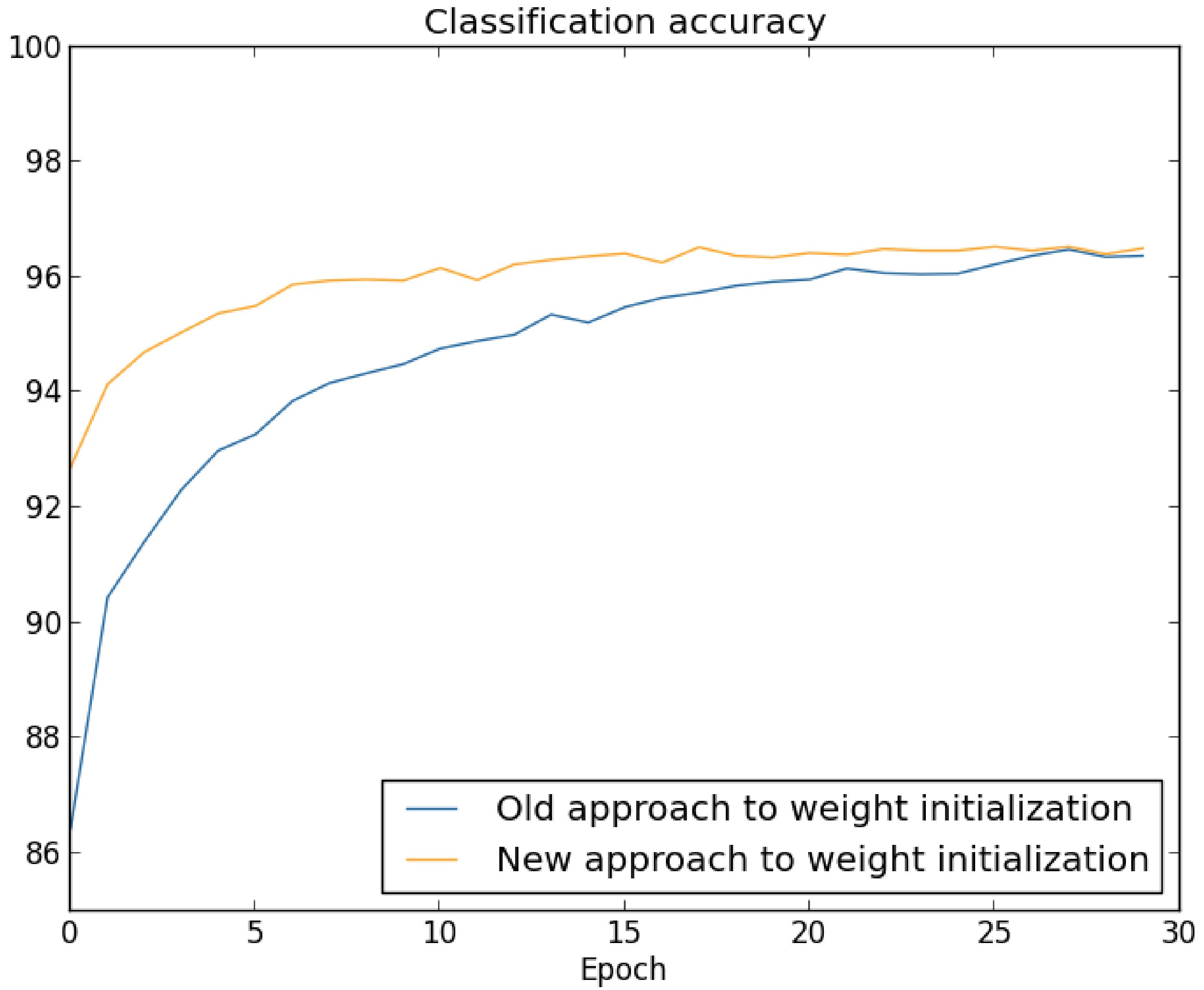
- Goals:
  - Independent neurons
  - Avoid initialization that can be close to a local minima
- Scheme
  - Process all data without labels (unsupervised)
  - Auto-encoder
- Problem: very lengthy

Conservation of Variance

- Goal 1: independent neurons
  - Heuristic approach
  - Uses random initialization
  - Normalized weights: $w \sim N(\mu = 0; \sigma = 1)$
- Goal 2: Conserve variance
  - if a neuron has n_{in} weights: $Z = W^T X$: $Z \sim N\left(0, \sqrt{n_{in}}\right)$
- Initialization Scheme:
  - Weight: $W \sim N(\mu = 0; \sigma = 1/\sqrt{n_{in}})$
  - Biases: $B \sim N(\mu = 0; \sigma = 1)$
- Problem: Doesn’t conserve variance between layers

Xavier Initialization

- Goals:
  - Desire operation in the linear range, where gradients are the larges: $|w| < 1$
  - Conserve variance of back-propagated gradients from layer to layer: $\sigma^2 = 1/n_{in} + 1/n_{out}$
    - Reduces discrepancies between layers.
- Normalized Initialization Scheme:
  - For sigmoid: $W \sim U\left[-\sqrt{\frac{6}{n_{in}+n_{out}}}, \sqrt{\frac{6}{n_{in}+n_{out}}}\right]$
  - For tanh: $W \sim U\left[-4\sqrt{\frac{6}{n_{in}+n_{out}}}, 4\sqrt{\frac{6}{n_{in}+n_{out}}}\right]$
- Problem: No recommendation for other AF
  - Can’t be used for ReLU
  - Assumes network is a chain of matrix multiplications with no non-linearities
    - Real nets: non-linearity is applied after each layer


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

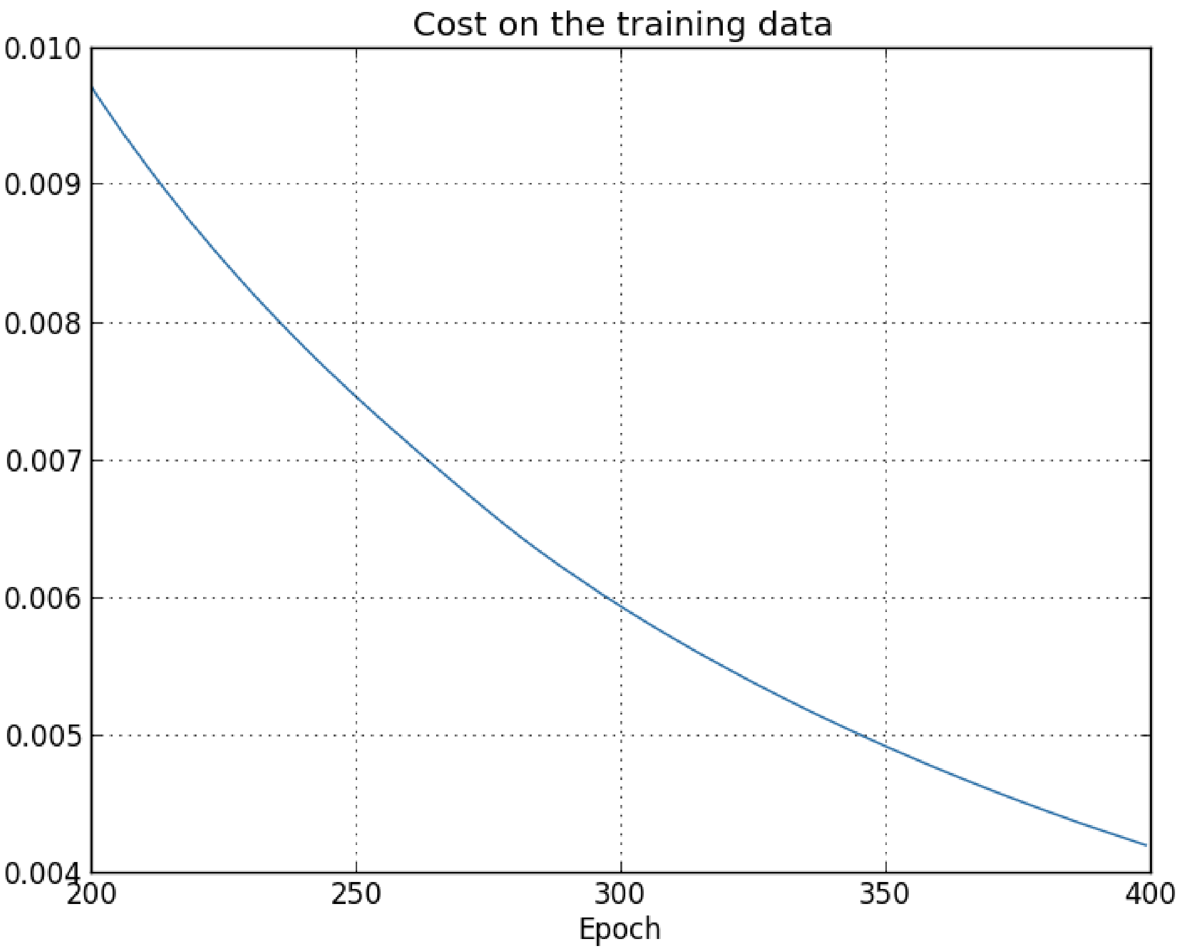
ReLU Initialization

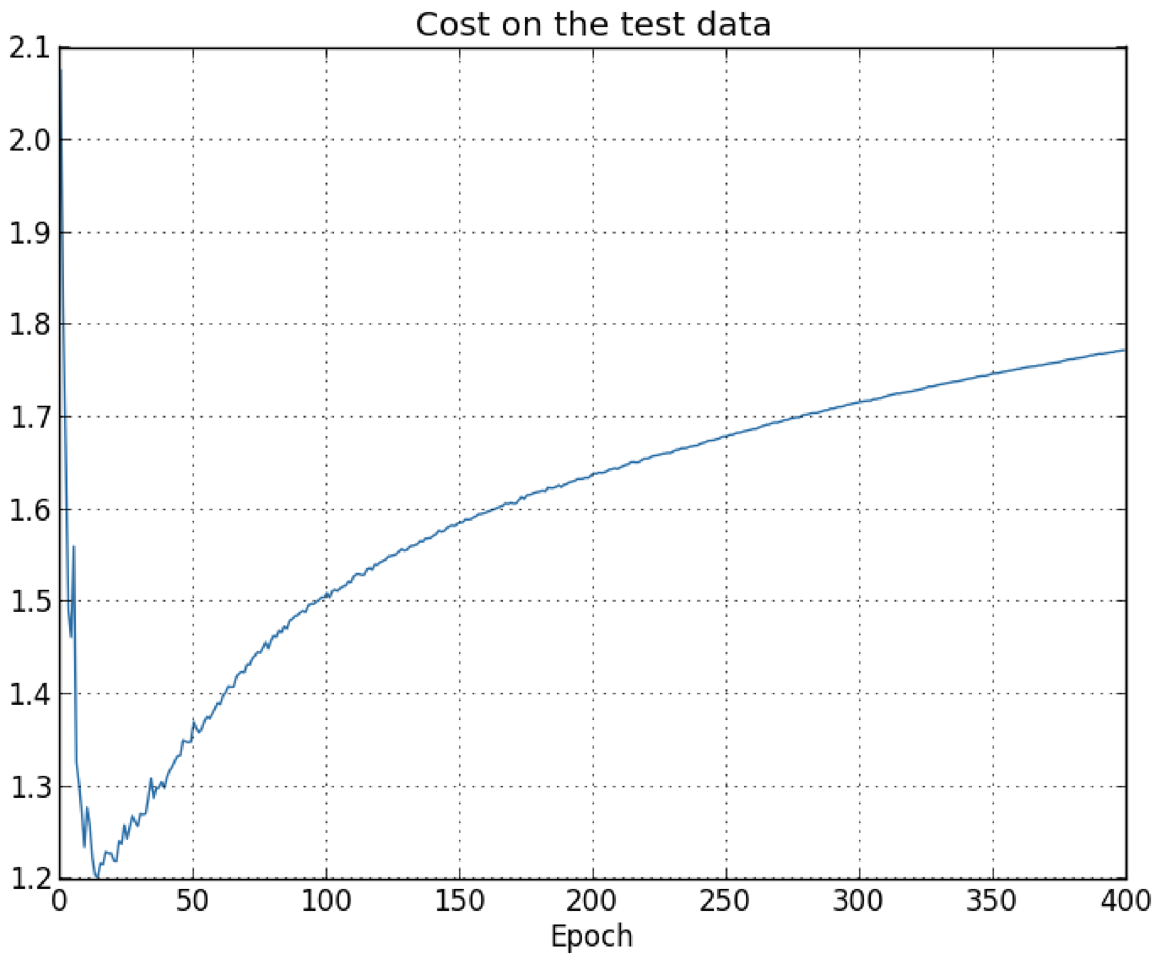
- ReLU output is not symmetric with zero-mean
- To make output symmetric: $\frac{1}{2} n_{in} \sigma_w^2 = 1$
- Scheme:
  - $W \sim U\left[-\sqrt{\frac{6}{n_{in}}}, \sqrt{\frac{6}{n_{in}}}\right]$

Generalization

Overfitting

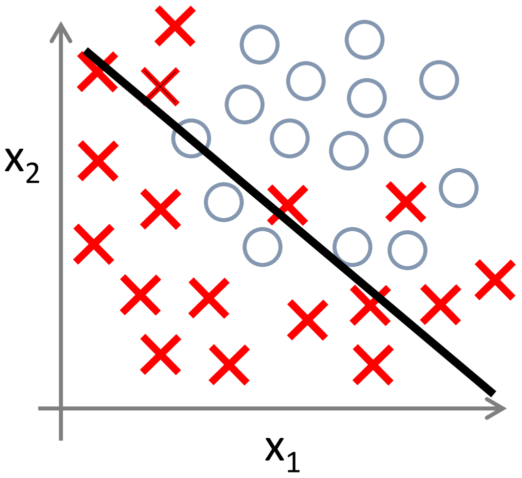
- Overfitting: solution that works well during training but bad in real-scenarios
- Neural nets have MANY parameters
    - Modern deep nets have M’s – B’s
    - High order
- Can’t report accuracy on training data
- Must split data into training and testing
    - Random split to avoid bias
    - Training typically 80%
    - Testing 20%
    - Use testing dataset to report accuracy/error


Cost on the training data
0.010
0.009
0.008
0.007
0.006
0.005
0.004
200
250
300
350
400
Epoch

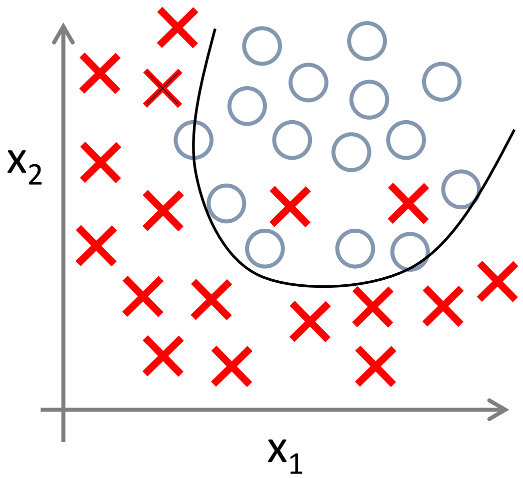

Cost on the test data
2.1
2.0
1.9
1.8
1.7
1.6
1.5
1.4
1.3
1.2
0
50
100
150
200
250
300
350
400
Epoch

Regularization

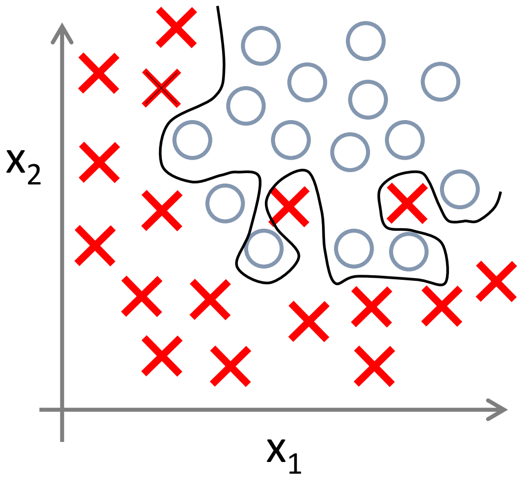
- Effort to prevent over fitting
- Underfitting is a problem of high bias
- Bias:
  - Making generalized assumptions about the data/model
  - E.g. oversimplification of the model
  - Mathematically: average(model output) – actual output
- Over-fitting is a problem of high variance
- Variance:
  - Sensitivity of the results to the choice of data
  - Computed from the output of the model
  - Mathematically: average of squared differences from the Mean.



$$h = a(w_0 + w_1x_1 + w_2x_2)$$



$$h = a\begin{pmatrix} w_0 + w_1x_1 + w_2x_2 \\ +w_3x_1^2 + w_4x_2^2 \\ +w_5x_1x_2 \end{pmatrix}$$



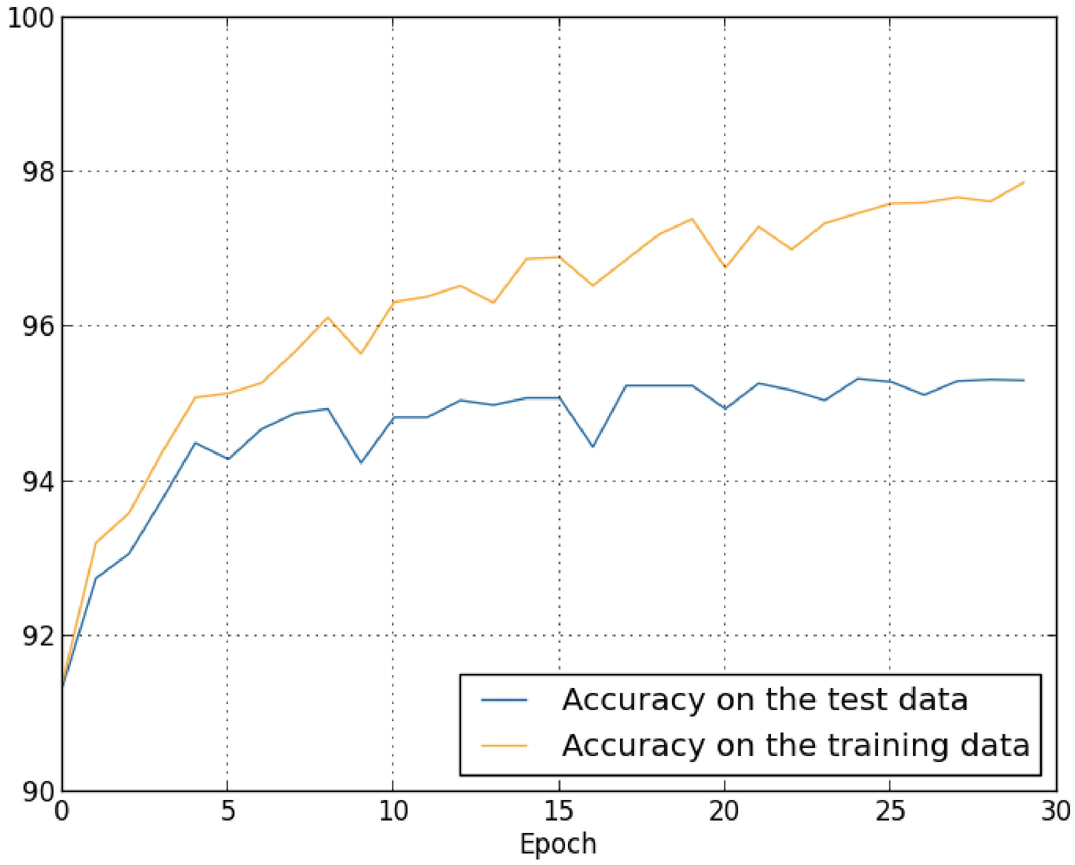
$$h = a\begin{pmatrix} w_0 + w_1x_1 + w_2x_1^2 \\ +w_3x_1^2x_2 + w_4x_1^2x_2^2 \\ +w_5x_1^2x_2^3 + w_6x_1^3x_2 + \cdots \end{pmatrix}$$

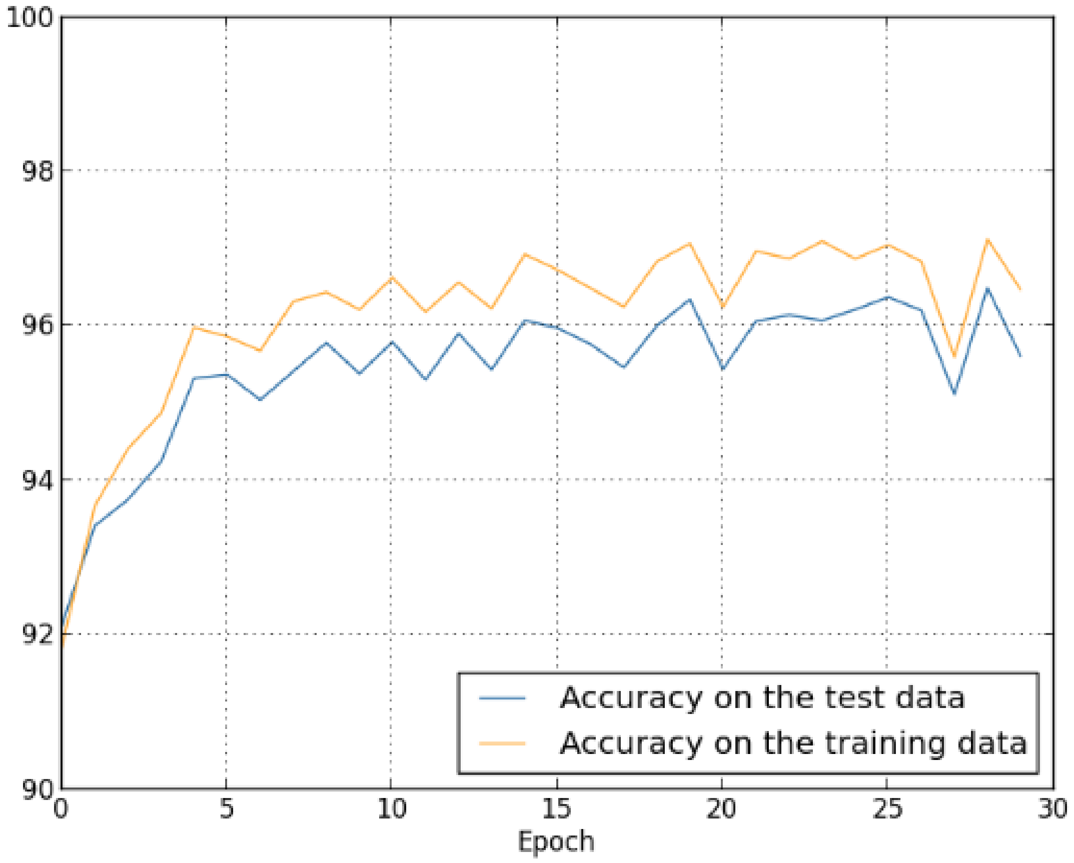
Generalization Techniques

- Larger training database
  - Synthetically augmented training
- Validation training set
- Regularization parameters
  - L1 and L2 parameters
- Gradient clipping
- Dropout

Increasing Training Data

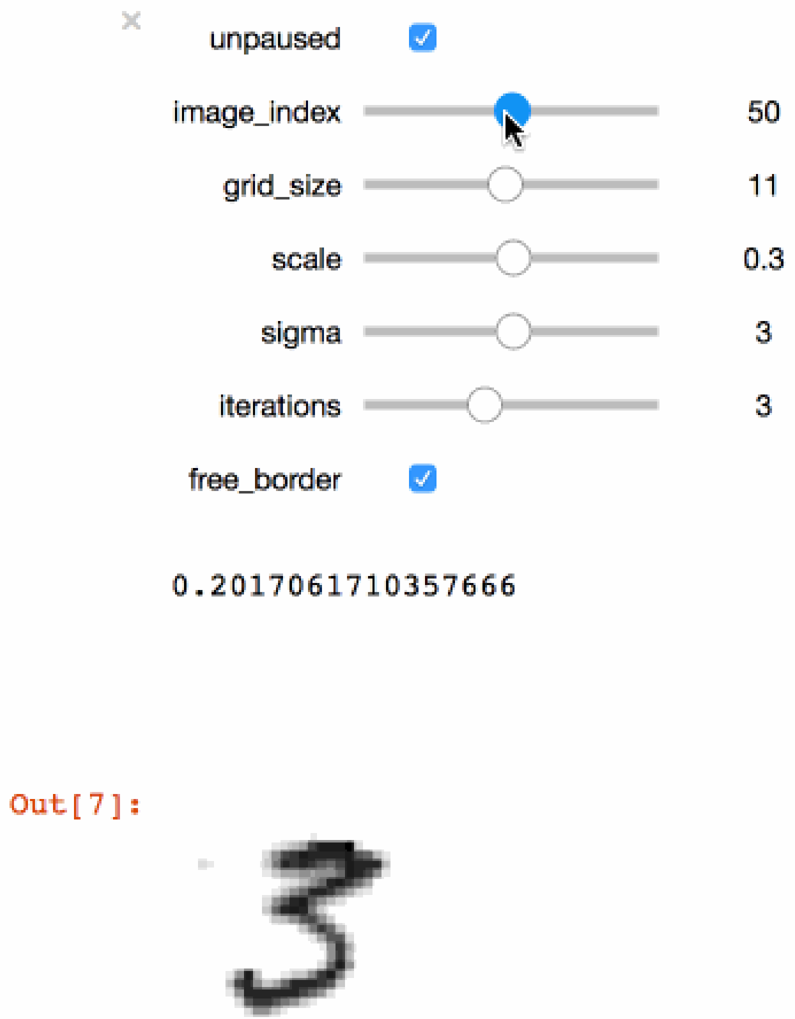
- MNIST 10K vs 50K

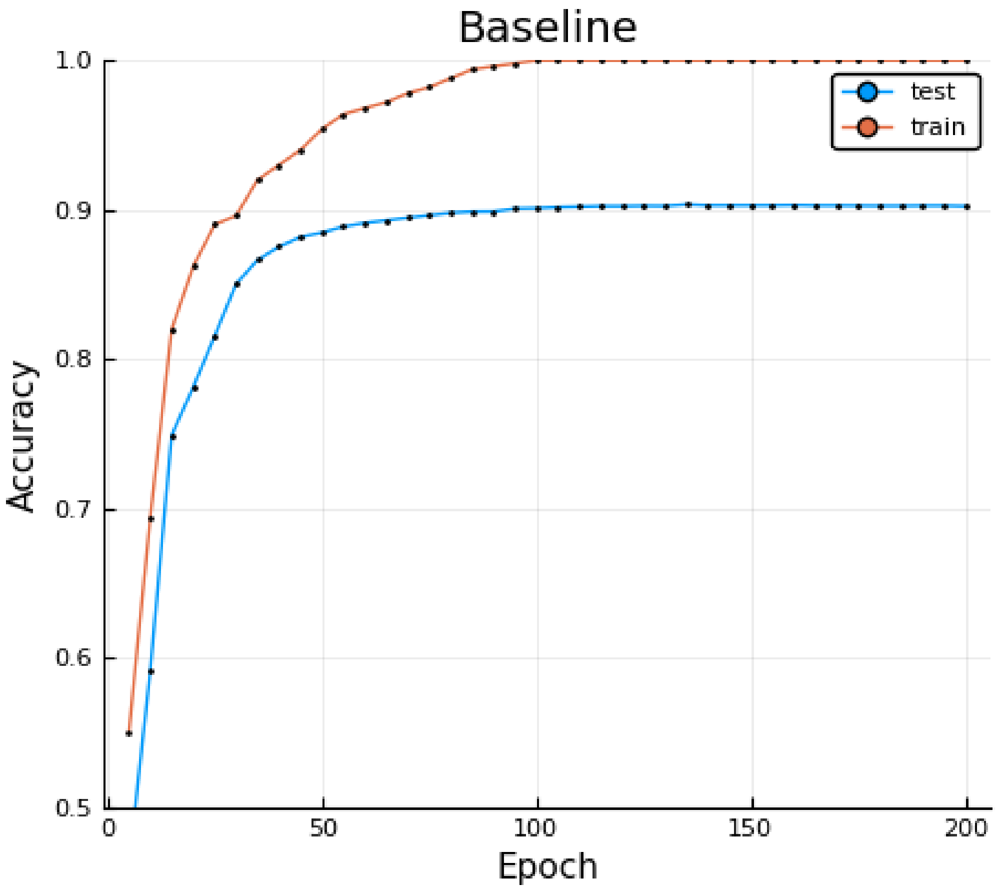


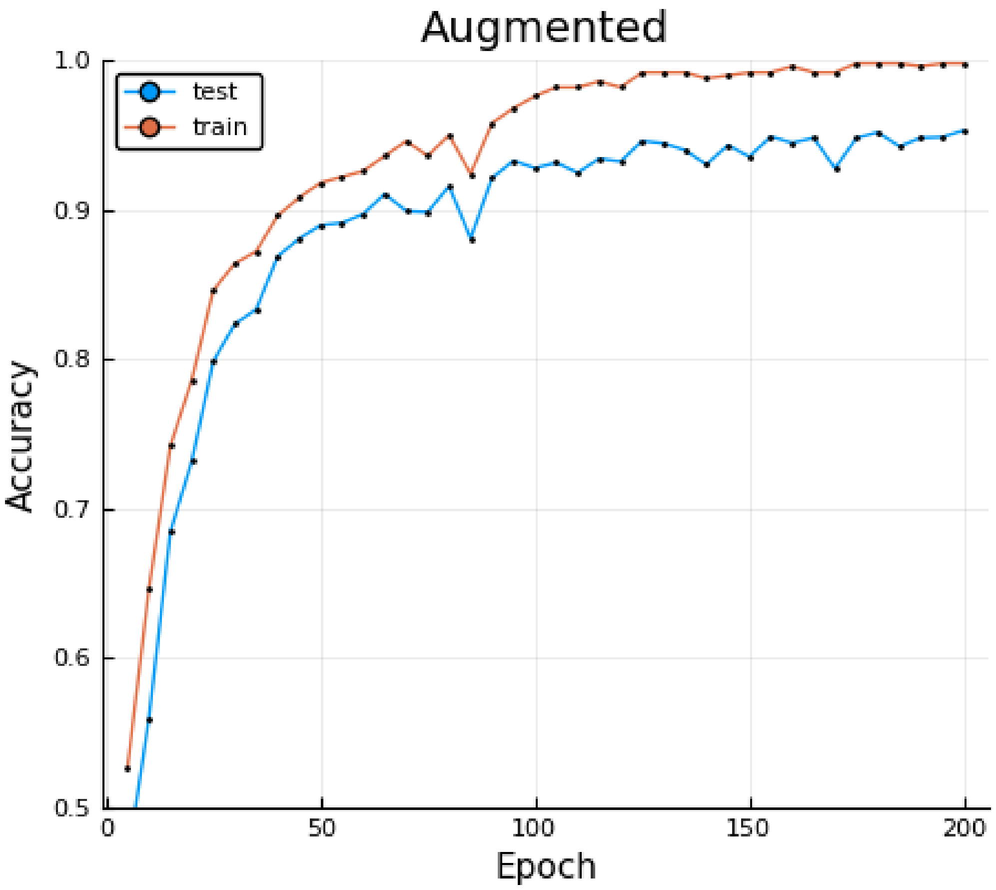


Augmenting

- 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




Baseline
test
train
Accuracy
Epoch
1.0
0.9
0.8
0.7
0.6
0.5
0
50
100
150
200


Augmented
test
train
Accuracy
Epoch
1.0
0.9
0.8
0.7
0.6
0.5
0
50
100
150
200

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:

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

  - C_i is the cost function: cross-entropy/log-likelihood/loss or mean square error
  - λ: regularization parameter
  - Doesn’t affect bias (aka $w_{0\,j}$)
- Learning objective always to reduce cost (error)
  - w^2 punishes large weights

Overfitting and Local Minima

- Another view of overfitting: system stuck in local minima
- Large weights
  - Make neuron sensitive to input
  - cause diminishing gradient effect
  - makes it difficult to explore weight space and find true minima
- Random initialization of weights
  - Can cause system to get stuck in local minima
- Large bias doesn’t make the neuron sensitive to input
  - Allowing large biases gives our networks more flexibility in behavior.
- λ is small we prefer to minimize the original cost function
  - λ is large we prefer small 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

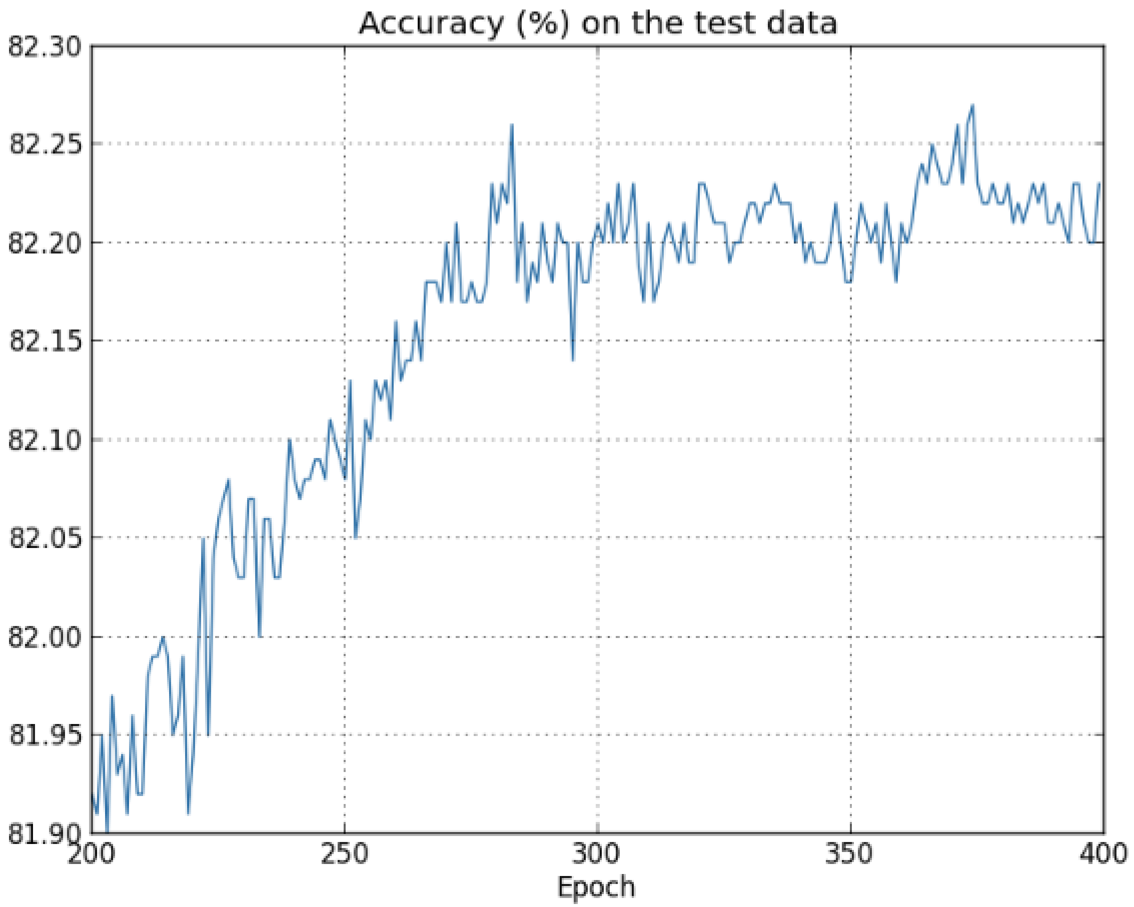
L1 Regularized Backprop

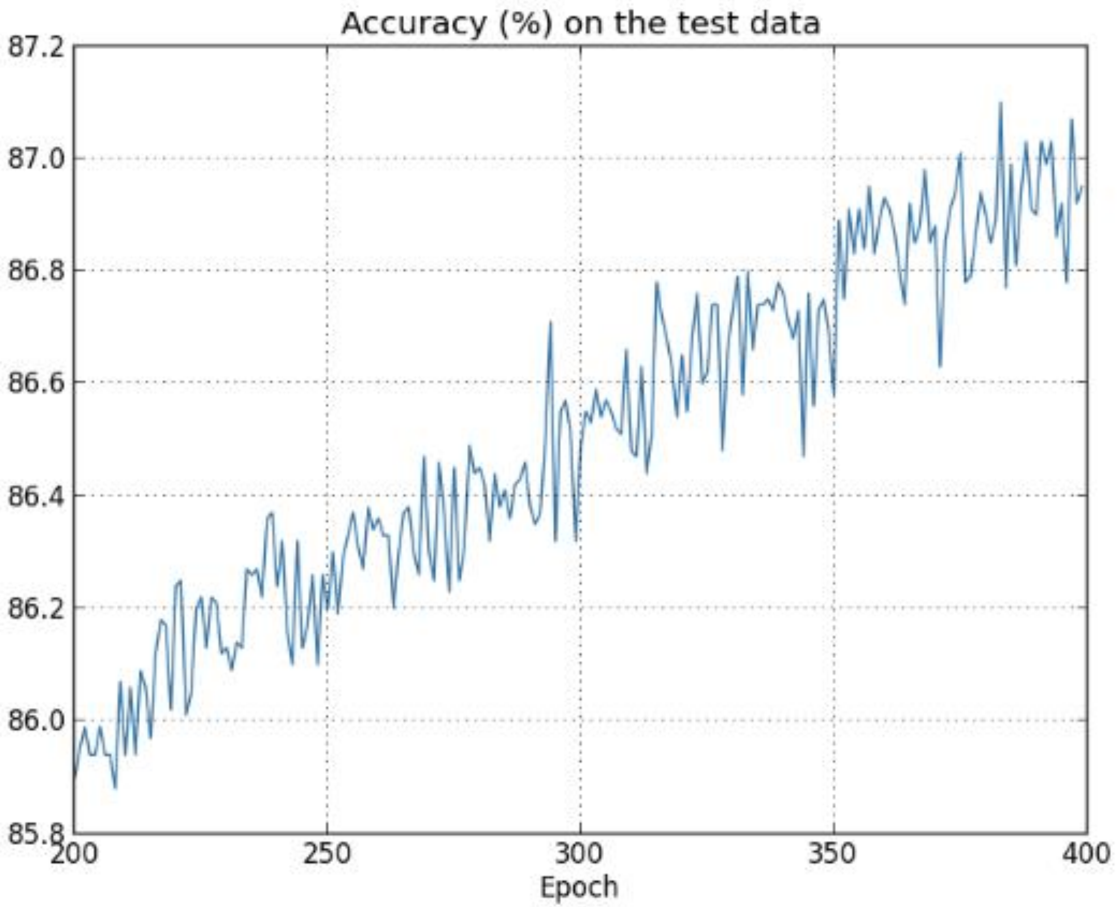
- Learning:

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

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

Regularized vs Non-regularized Cost Function



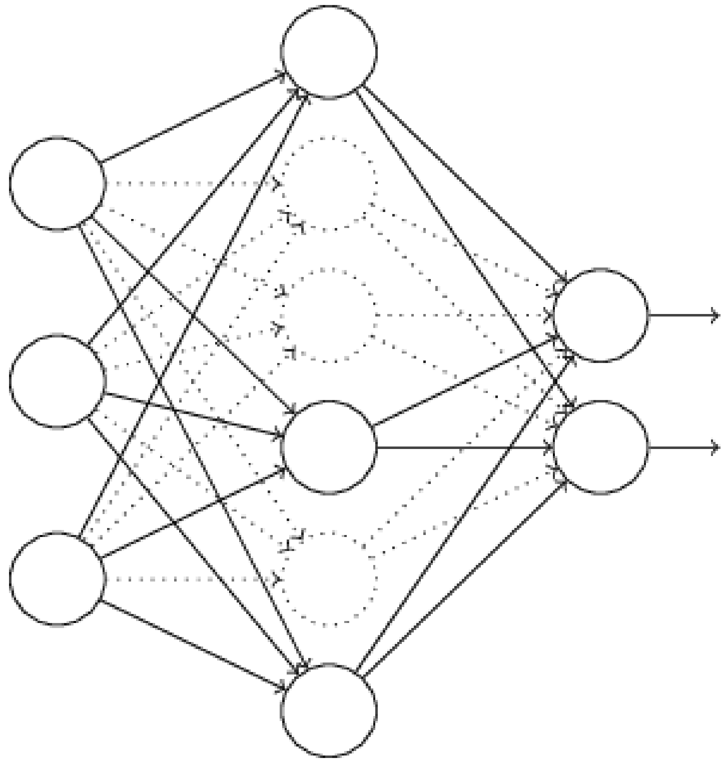


Gradient Clipping

- Large gradient causes instability
- Set a max value (ceiling) for gradient

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

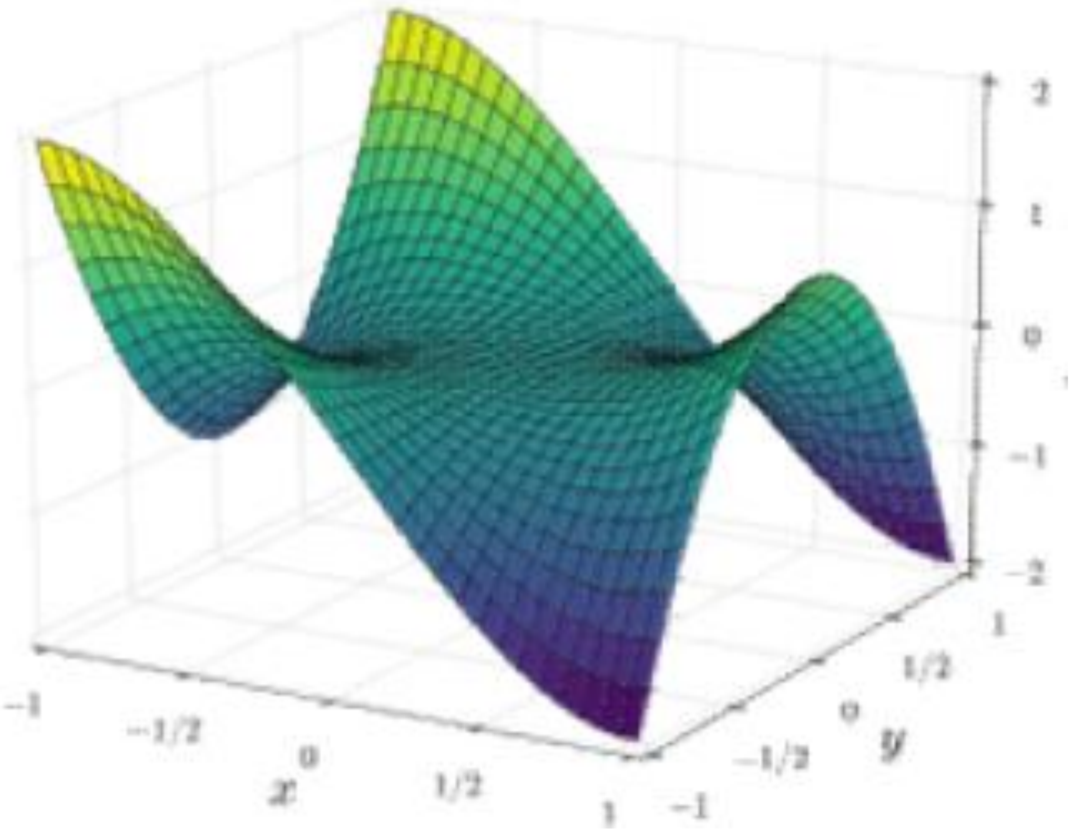
Validation

- Aka cross-validation
- Randomly split data:
  - Training (60%), Testing (20%), Validation (20%)
  - Validation size ~ test size
- Validate you models
  - Use validation results to validate modeling choices
  - Check results of validation as part of your training
- Early stop:
  - Determine when learning is no longer beneficial

Learning

Deep Learning Problem: Non-Convex Cost

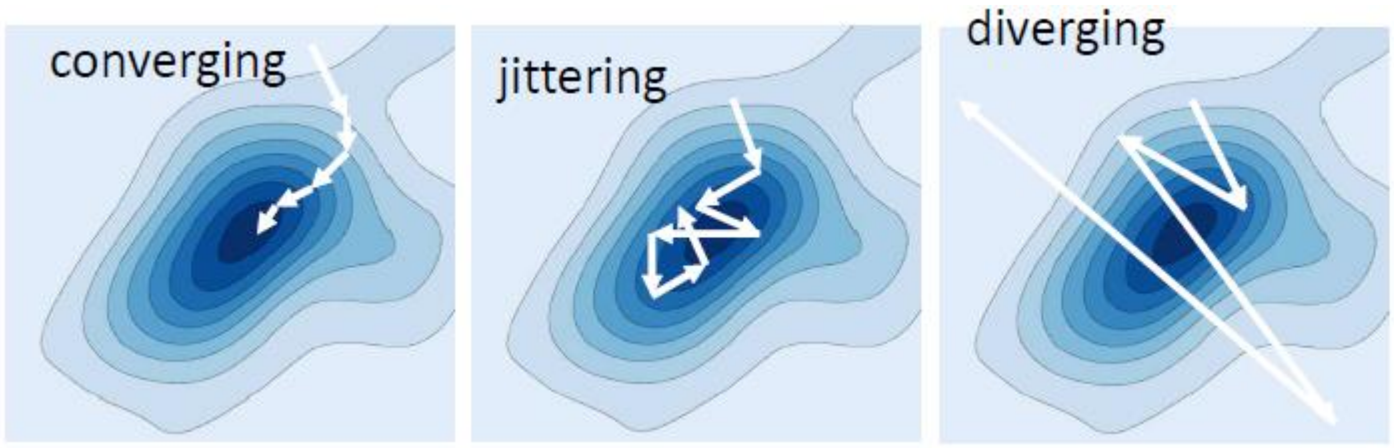
- In large networks, saddle points are far more common than local minima
- Gradient descent algorithms often get “stuck” in saddle points



Deep Learning Problem: Learning rate

$$w_{ij}^{(l)} = w_{ij}^{(l)} + \alpha \frac{\partial \boldsymbol{E}}{\partial w_{ij}^{(l)}}$$

- Rate assumes slope is identical in all directions



Momentum Based GD

- Introduce a velocity parameter, v, for each, w, such that:

$$w = w + v$$
$$v = \beta v - \alpha \nabla C$$

- β is friction, β=1 no friction (no slowing down)
  - AKA momentum cooefficient
  - >>1 can causes overshoot
  - <<1 dampens
- If the direction of ∇C doesn't change velocity builds up
  - Faster converging

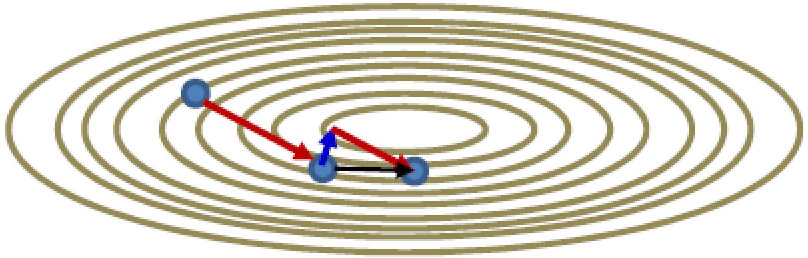
Momentum Learning

- Scheme:
  - Initialize w, α, β
  - While not (stopping criterion):
    - Given $\{X, Y\}$ feedforward
    - Compute gradient
    - Compute velocity: $v = \beta v - \alpha \nabla C$
    - Compute weights: $w = w + v$
- Twice as many parameters!

Nesterov Momentum

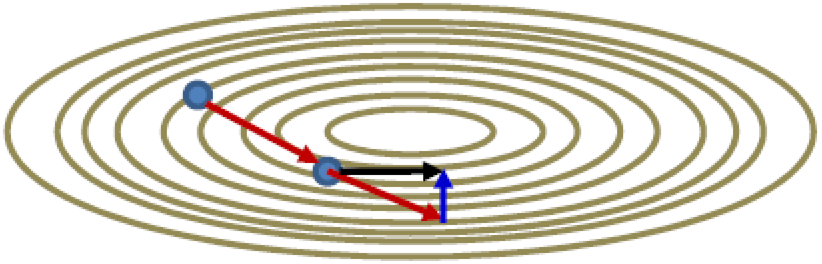
- Similar to mementum SDG
- Evaluate gradient AFTER applying velocity
  - Correction to standard momentum
- Scheme:
  - Initialize $w, \mathrm{v}, \alpha, \beta$
  - While not (stopping criterion):
    - Given $\{X, Y\}$ feedforward
    - Compute weights: $w = w + v$
    - Compute gradient
    - Compute velocity: $v = \beta v - \alpha \nabla C$

Momentum



$$\Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_W Err\left(W^{(k-1)}\right)$$

Nestorov



$$W_{extend}^{(k)} = W^{(k-1)} + \beta \Delta W^{(k-1)}$$

$$\Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_W Err\left(W_{extend}^{(k)}\right)$$

$$W^{(k)} = W^{(k-1)} + \Delta W^{(k)}$$

Adaptive Learning Rates

- Learning rate is most difficult to set
- Momentum helps but introduces new parameters
- Delta-bar-Delta (1988)
  - Idea: aggressively explore gradient
  - if gradient has the same sign (ie direction), then the learning rate should increase.
  - If gradient changes sign, learning rate should decrease
  - Can result in excessive increase in learning rate

AdaGrad (2011)

- Idea: explore gently sloped directions
- Weights with largest gradient: decrease their learning rate
  - Weights with the smallest gradient: increase their learning rate.
- Performs well in convex settings
- Can result in a premature and excessive decrease in the effective learning
- Performs poorly in non-convex settings

AdaGrad Scheme

- Initialize w, α
- Initialize $r = 0$
- While not (stopping criterion):
  - Feedforward $\{X, Y\}$, compute C
  - Compute gradient, $g = \frac{\partial C}{\partial w}$
  - Compute accumulated squared gradient, $r = r + g \odot g$
  - Update weights: $w = w - \frac{\alpha}{\gamma + \sqrt{r}} \odot g$
    - γ is a stabilization constant

RMSProp (2012)

- Optimized learning
  - Improve AdaGrad
- Gradient accumulation into an exponentially(rms) weighted moving average.
  - Uses entire history to compute average
  - Apply a rate of decay to history so that it doesn’t overwhelm

RMSProp Scheme

- Initialize w, α, ρ
  - ρ is a rate of decay
- Initialize $r = 0$
- While not (stopping criterion):
  - Feedforward $\{X, Y\}$, computer C
  - Compute gradient, $g = \frac{\partial C}{\partial w}$
  - Compute accumulated squared gradient, $r = \rho r + (1 - \rho) g \odot g$
  - Update weights: $w = w - \frac{\alpha}{\sqrt{\gamma + r}} \odot g$
    - γ is a stabilization constant

Adam (2014)

- Adaptive moments
- Variant that combines RMSProp and momentum
- RMS applied to first order momentum and second order momentum

Adam Scheme

- Set
  - α (default = 0.001)
  - decay for 1st and 2nd momentum, ρ_1, ρ_2 (default 0.9, 0.999, suggested (0,1])
  - stabilization γ (default 10^{-8})
- Initialize w
- Initialize 1st and 2nd momentum: $s, r = 0$
- While not (stop criterion):
  - Feedforward $\{X, Y\}$, compute C
  - Compute gradient: $g = \frac{\partial C}{\partial w}$
  - $s = \frac{\rho_1 s + (1-\rho_1) g}{1-\rho_1}$
  - $r = \frac{\rho_2 r + (1-\rho_2) g \odot g}{1-\rho_2}$
  - $w = w - \alpha \frac{s}{\gamma + \sqrt{r}}$

Hessian

- Alternative to gradient descent
  - Second order
- Cost function redefined:

$$C(w+\Delta w)=C(w)+\sum_i \frac{\partial C}{\partial w_i}\Delta w_i+\frac{1}{2}\sum_{ij}\Delta w_i \frac{\partial^2 C}{\partial w_i \partial w_j}\Delta w_j+\cdots$$

$$C(w+\Delta w)=C(w)+\nabla C\ \cdot\ \Delta W+\frac{1}{2}\Delta W\cdot H\cdot \Delta W+\cdots$$

$$\Delta W=-\alpha H^{-1}\nabla C$$

- Converges faster that gradient
- Computationally more challenging
- Initialization problems

Conjugate Gradients

- Second order
- Avoids calculating the Hessian
- Searches for a direction of descent that is conjugate to the previous direction

$$d_t = \nabla_w C(w_t) + \beta_t d_{t-1}$$

$$(d_t)^T H \, d_{t-1} = 0$$

  - $\nabla_w C(w_t)$ is gradient at iteration t
  - d_t is the direction of descent at iteration t
  - β_t is the correction coefficient

- Can calculate without the Hessian using 2 methods:
  - Fletcher-Reeves: $\beta_t = \dfrac{\left(\nabla_w C(w_t)\right)^T \nabla_w C(w_t)}{\left(\nabla_w C(w_{t-1})\right)^T \nabla_w C(w_{t-1})}$
  - Polak-Rebiere: $\beta_t = \dfrac{\left(\nabla_w C(w_t) - \nabla_w C(w_{t-1})\right)^T \nabla_w C(w_t)}{\left(\nabla_w C(w_{t-1})\right)^T \nabla_w C(w_{t-1})}$

Conjugate Gradient Scheme

- Initialize w
- Set $d_0, g_0 = 0; t = 1$
- While not (stop criterion)
  - Feedforward, compute C_t
  - Compute Gradient: $g_t = \frac{\partial C_t}{\partial w_t}$
  - Compute β_t
  - Compute d_t
  - Search of α that minimizes $C(w_t + \alpha\, d_t)$
  - $t = t + 1$

Nonlinear Conjugate Gradient

- Conjugate gradient works best for quadratic costs
  - Line search w along conjugate directions
- For non-quadratic conjugate directions don’t guarantee minimization of C
- Nonlinear conjugate gradient: occasional reset line search to pure gradient directions
  - i.e. $\beta = 0$

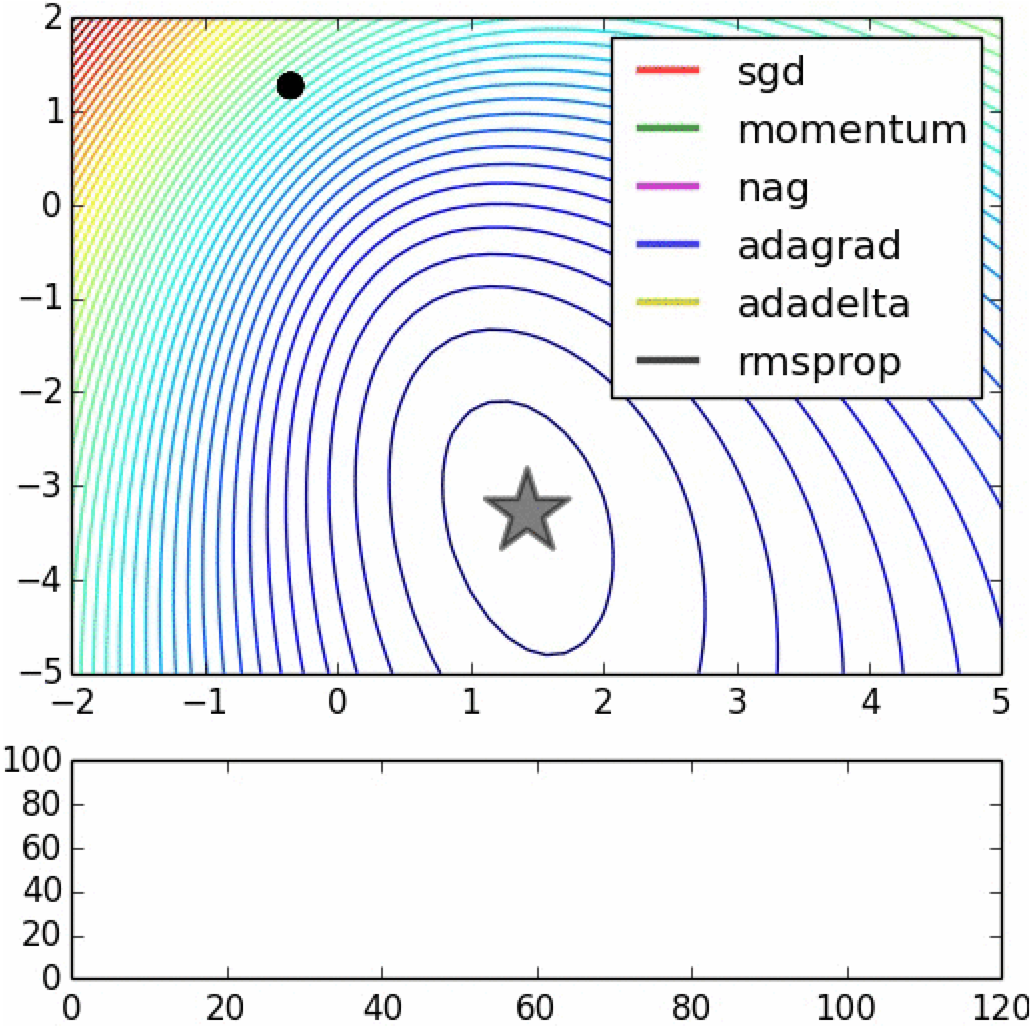
BFGS

- Newton’s second order method
- Hessian inverse is difficult to compute
  - Eigenvalues must be non-zero
  - Negative eigenvalues must be close to 0
- Broyden–Fletcher–Goldfarb–Shannon
  - Iterative algorithm
  - Approximates H using M

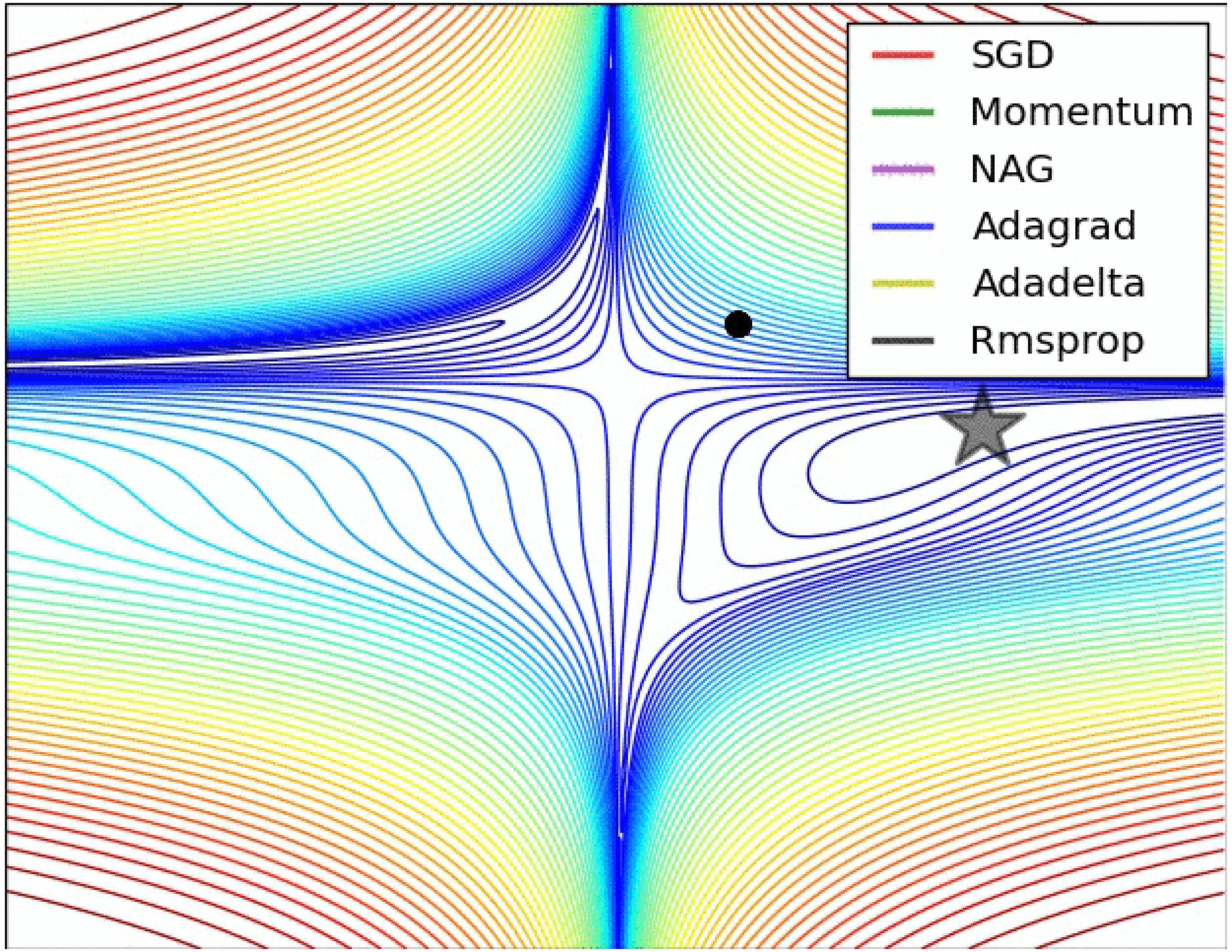
BFGS Scheme

- Initialize w
- While not (stop criterion)
  - Feedforward, compute C
  - Compute gradient: g
  - Approximate Hessian using BFGS: M
  - $w = w - M^{-1}g$

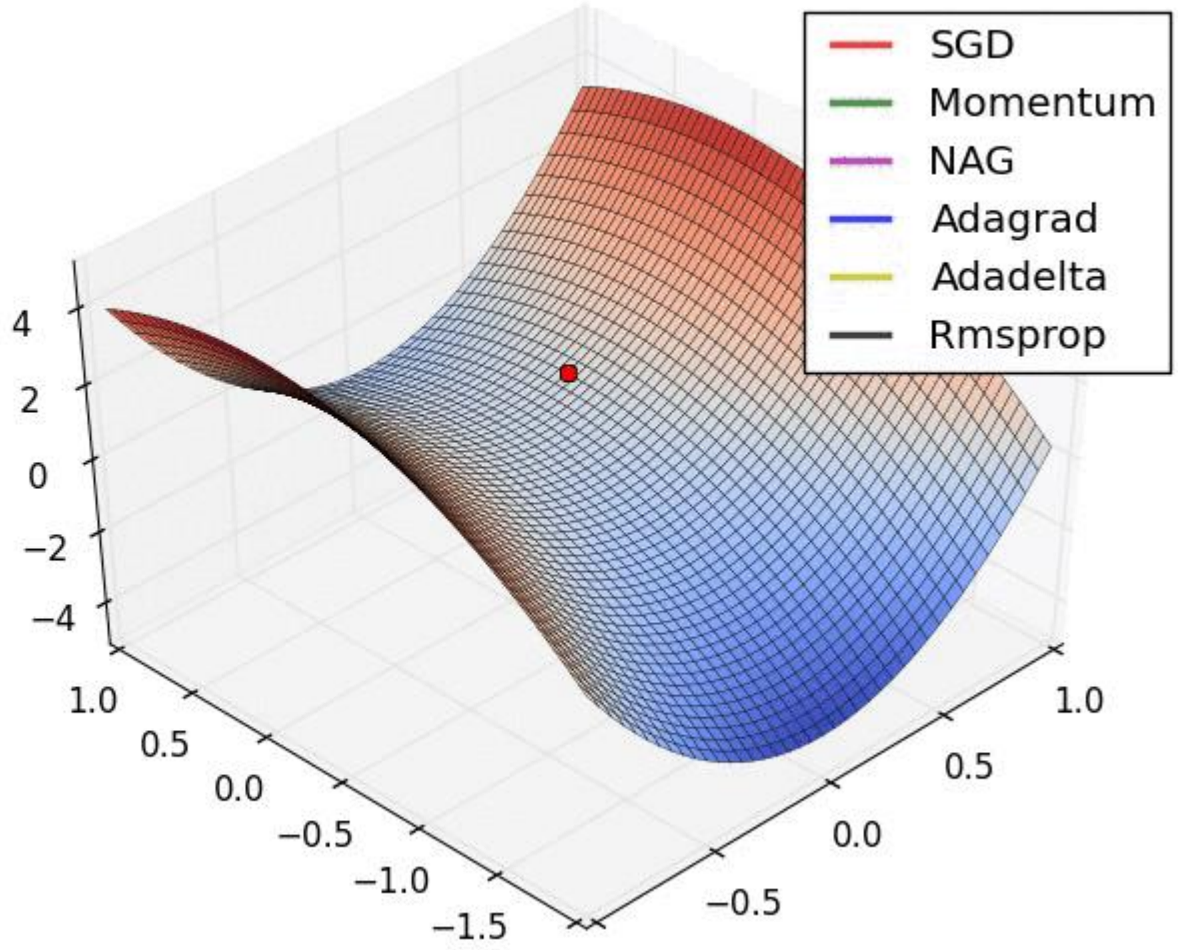
Convex

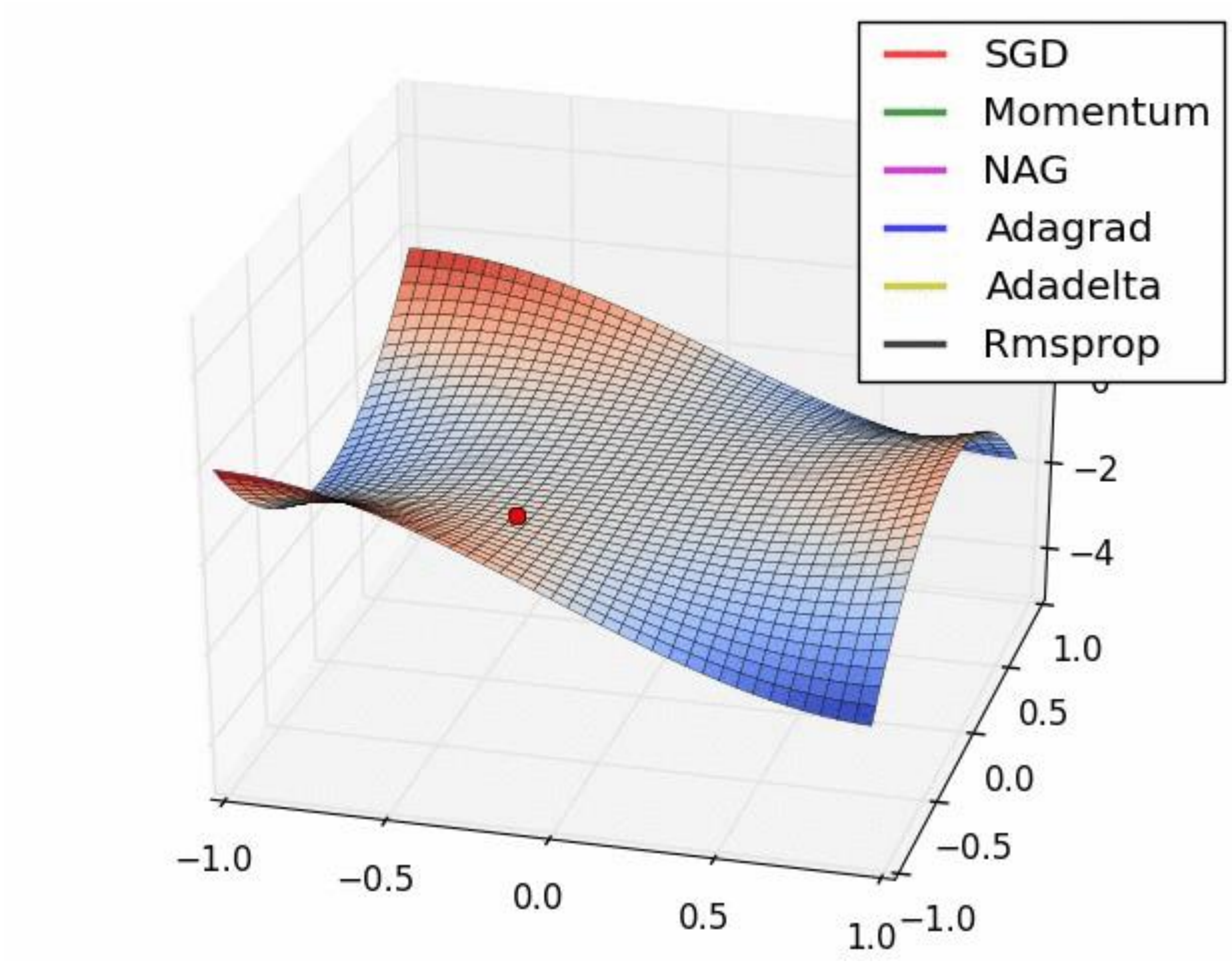


Non-Convex



Saddle Points




SGD
Momentum
NAG
Adagrad
Adadelta
Rmsprop
−2
−4
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0