Deep Neural Networks - Hyperparameters

Pattern Recognition

Dennis Madsen

Topic overview

- *Neural Networks and Deep Learning*
- **Improving DNN: Hyperparameter tuning, regularization, optimization**
- Convolutional Neural Networks (CNN)
- CNN popular architectures
- Sequence Models/Recurrent neural networks (RNN)
- Beyond the basics (object detection and segmentation)

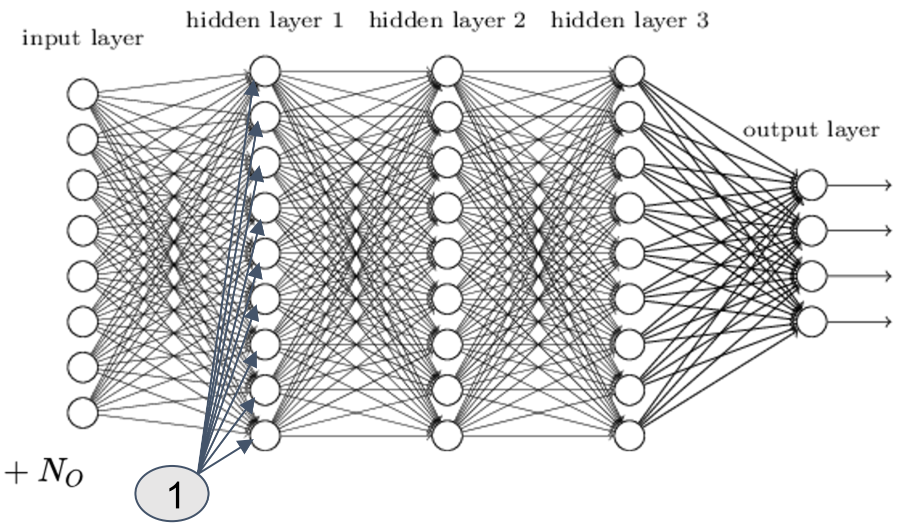
Architecture Design - Fully connected

Neural networks are organized into groups of units called layers. Most neural network architectures arrange these layers in a chain structure, with each layer being a function of the layer that preceded it.

Multi-layer networks are preferable over 3-layered networks because they often generalize better

$$\boldsymbol{h}^{(1)} = g^{(1)}\left(\boldsymbol{W}^{(1)\top}\boldsymbol{x} + \boldsymbol{b}^{(1)}\right),$$

$$\boldsymbol{h}^{(2)} = g^{(2)}\left(\boldsymbol{W}^{(2)\top}\boldsymbol{h}^{(1)} + \boldsymbol{b}^{(2)}\right),$$



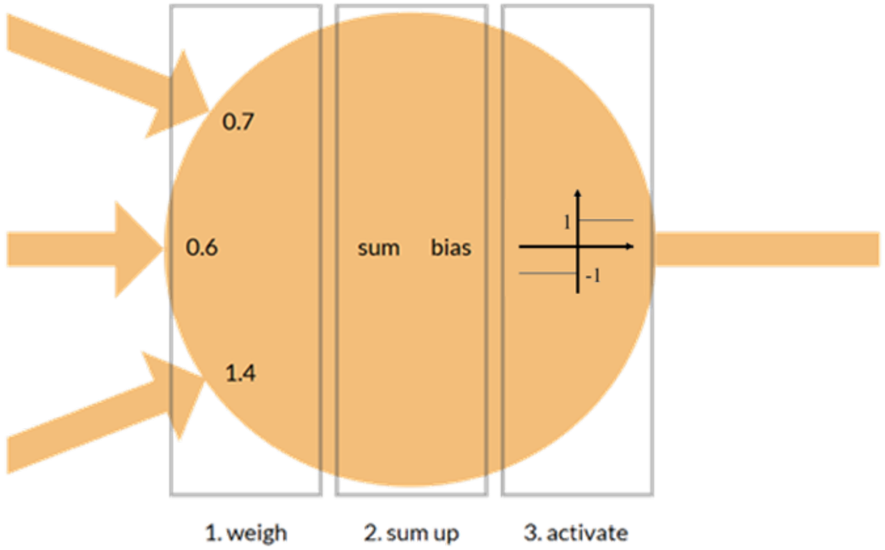
Number of parameters in fully connected network:

$$N_F = N_I H_1 + H_1 + H_1 H_2 + H_2 + H_2 H_3 + H_3 + H_3 N_O + N_O$$

- The bias term is always set to 1 and shared among all the neurons in a layer.
- The bias is often left out in architecture visualizations.

A Neuron modelled as a Perceptron

- The structural building block of Neural Networks.
- Perceptrons are also referred to as “artificial neurons”, highlighting the original **inspiration** from biological neurons.

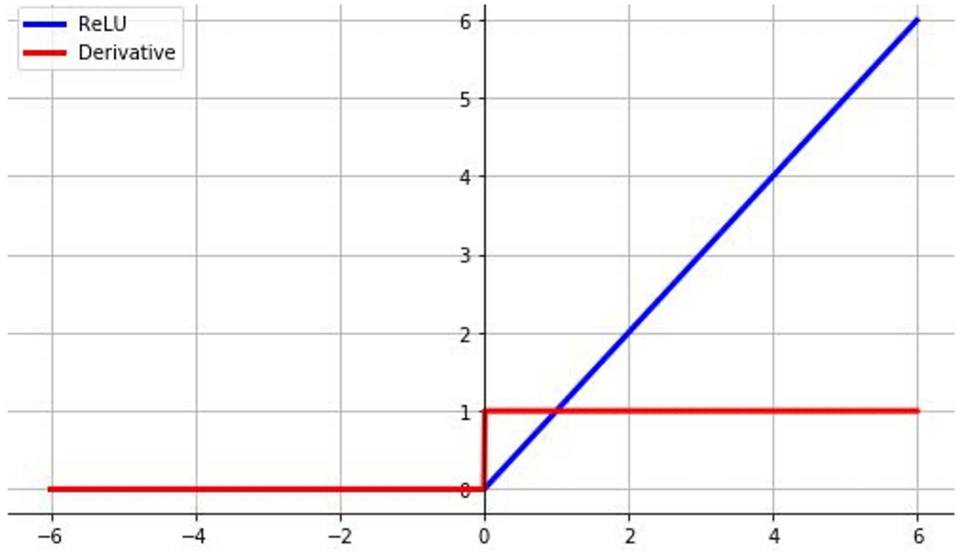


$$w^T x + b > 0 \text{ class } 1$$
$$w^T x + b < 0 \text{ class } 0$$

Estimate the weights **w** using gradient descent

*A Neural network (feedforward network) is a **function approximation machine** that is designed to achieve **statistical generalization**, which **occasionally** draws some insight from what we know about the brain. Neural networks are **NOT** models of brain functions.*

Activation functions: ReLU



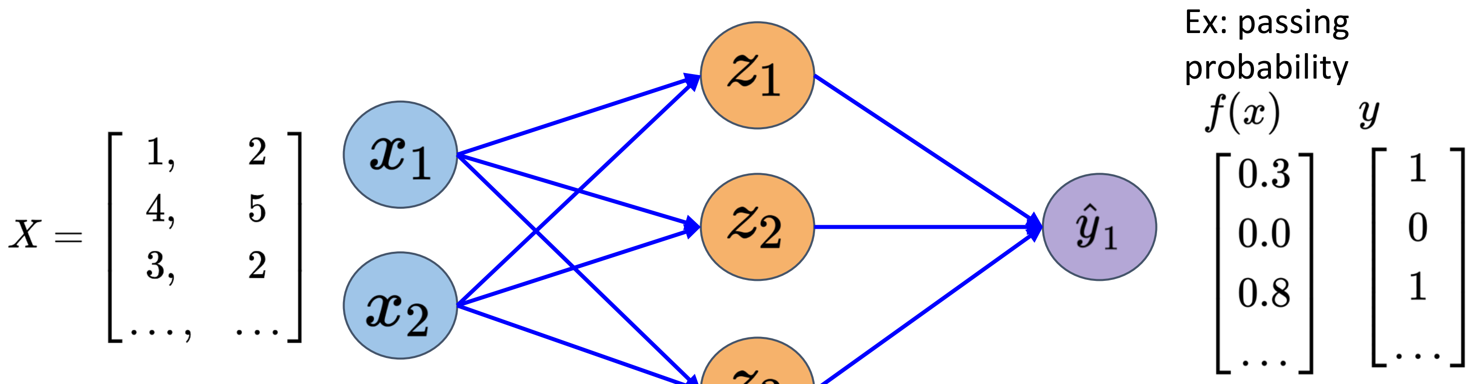
$$f(x) = \begin{cases} 0 \text{ for } x < 0 \\ x \text{ for } x \geq 0 \end{cases}$$

$$\frac{\partial f(x)}{\partial x} = \begin{cases} 0 \text{ for } x < 0 \\ 1 \text{ for } x > 0 \end{cases}$$

- “Rectified linear unit”
- Efficient to compute
- Smaller risk of vanishing gradients

Binary Cross Entropy Loss

- The cross entropy loss (as introduced with logistic regression) can be used with models that output a probability between 0 and 1.
- Improvement from the mean-squared error used in the 80s and 90s.
- Categorical cross entropy for multi-class classification


$X = \begin{bmatrix} 1, & 2 \\ 4, & 5 \\ 3, & 2 \\ \ldots, & \ldots \end{bmatrix}$
x_1
x_2
z_1
z_2
z_3
$\hat{y}_1$
Ex: passing probability
$f(x)$
y
$\begin{bmatrix} 0.3 \\ 0.0 \\ 0.8 \\ \ldots \end{bmatrix}$
$\begin{bmatrix} 1 \\ 0 \\ 1 \\ \ldots \end{bmatrix}$

$$J(W) = \frac{1}{n}\sum_{i=1}^{n} y^{(i)} log(f(x^{(i)})) + (1 - y^{(i)}) log(1 - f(x^{(i)}))$$

True label Predictio True label

Prediction

Source: http://introtodeeplearning.com/

Topic overview

- **Improving DNN: Hyperparameter tuning, regularization, optimization**
    - Hyperparameters vs parameters
    - Datasets
    - Hyperparameter tuning
    - Optimization algorithms

What are hyperparameters

- Parameters: W[1], b[1], W[2], b[2], W[3], b[3], …

- Hyperparameters (control how to find W and b):
    - Learning rate
    - Number of iterations
    - Regularization
    - Network architecture
        - Number of hidden layers
        - Number of hidden units
        - Choice of activation function
    - Optimization algorithm hyperparameters (e.g. momentum)
    - Minibatch size

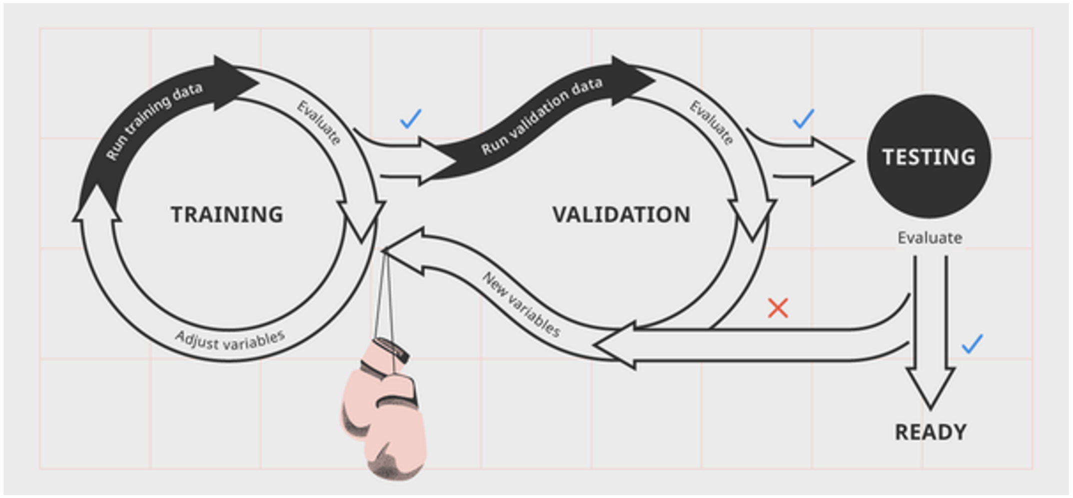
Source:

Dataset

The dataset should be split into 3 categories

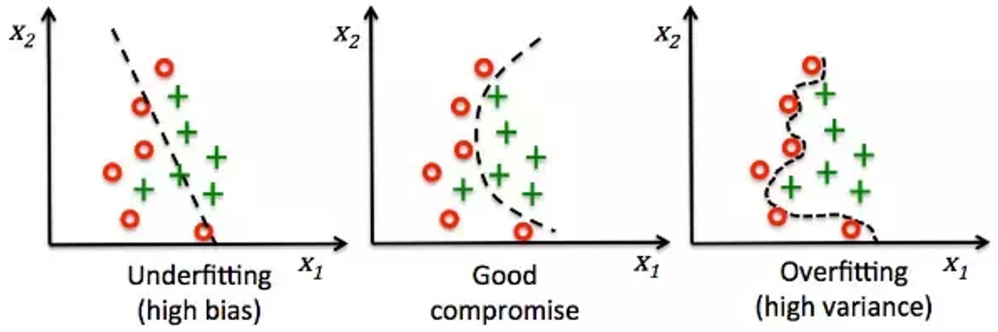
- Training
- Validation/hold-out/cross validation
- Test

Using a small dataset (100-10.000), the split will usually be 60/20/20 (or even 70/30).
Big data (>1M), around 10.000 examples for validation and test could be sufficient, i.e. split the data 98/1/1



Very important: Do **NOT** optimize your hyperparameters on the test set!

Bias and Variance



- High bias (training data):
  - Bigger network
  - Train longer
  - New architecture
- High variance (validation data):
  - More data
  - Regularization
  - New architecture

With enough data, no bias/variance trade-off:

Add more data, make the network deeper. Will lower both bias and variance.

In high dimensional cases (such as image classification) it is not possible to visualize the classification hyperplane.

Instead, we can look at classification accuracy.

Example	High variance	High bias	High bias High variance	Low bias Low variance
Training set error:	1%	15%	15%	0.5%
Validation set error:	11%	16%	30%	1%

Given that the classification is easily solvable by humans ~0% error

Source: https://www.coursera.org/learn/deep-neural-network/

Regularization

To prevent overfitting

Techniques that constrains our optimization problem to discourage complex models.

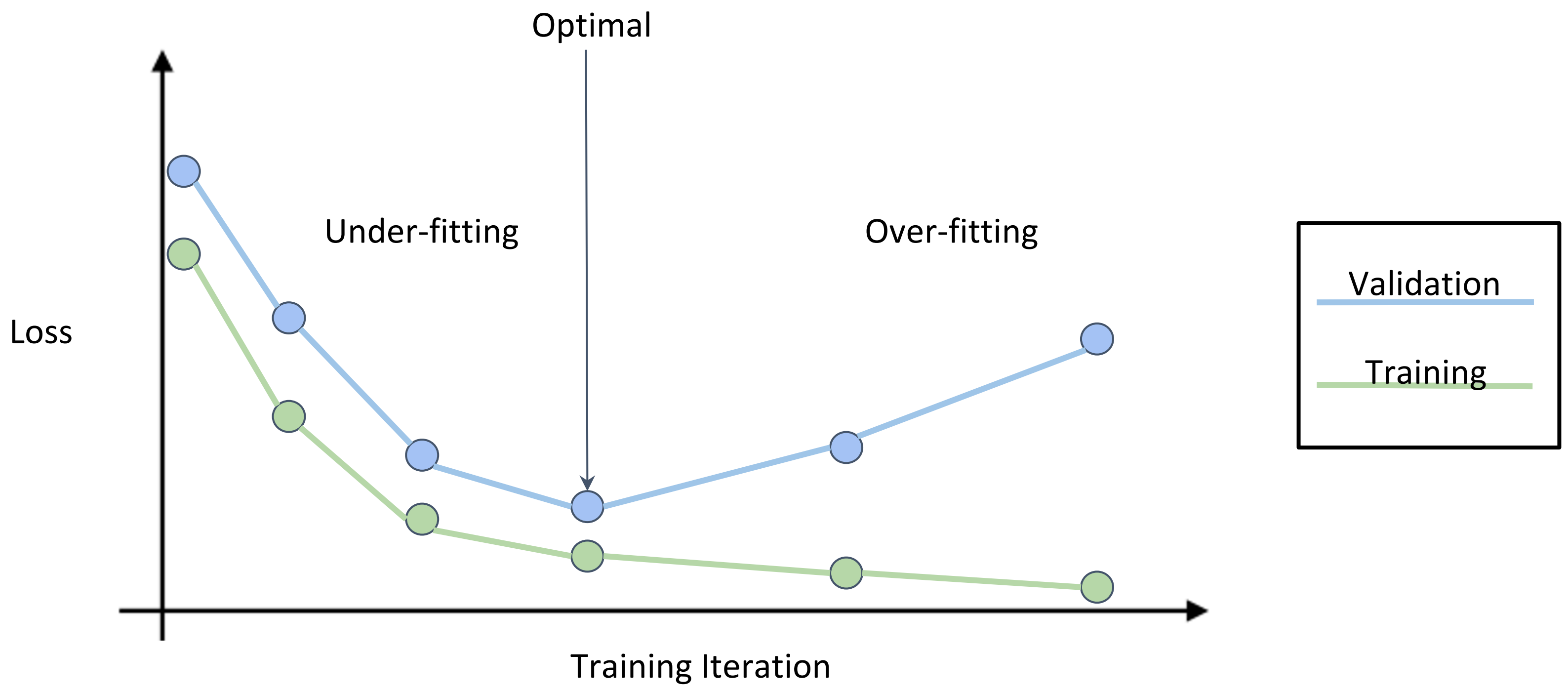
- Weight regularization (L2, L1)

$$L(y, \hat{y}) = (y_i - \hat{y}_i)^2 + \alpha ||W||_2$$

- Early stopping
- Dropout
- Data augmentation

Early Stopping

Idea: Stop training before we have a chance to overfit

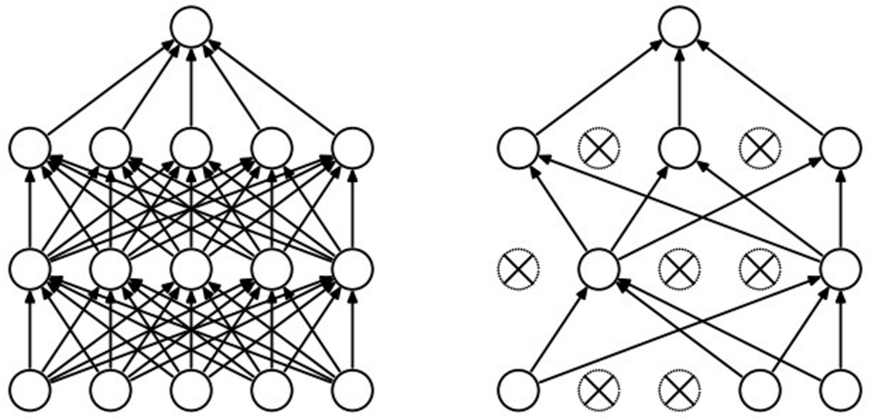


Source: http://introtodeeplearning.com/

Dropout

During **training**, randomly set some activations to 0

- Typical dropout value of 50% activation for all hidden units
- Avoids the network relies on single nodes



(a) Standard Neural Net (b) After applying dropout.

Downside: Cost function is no longer well defined (i.e. might not decrease in every step).

Srivastava, Nitish, et al. "Dropout: a simple way to prevent neural networks from overfitting." *The Journal of Machine Learning Research* 15.1 (2014): 1929-1958.

Data augmentation

Getting more training data can help to regularize, but it can be expensive.

Augmentation by:

- horizontal flip, random rotation, random crop
- random distortions
- Color shifting

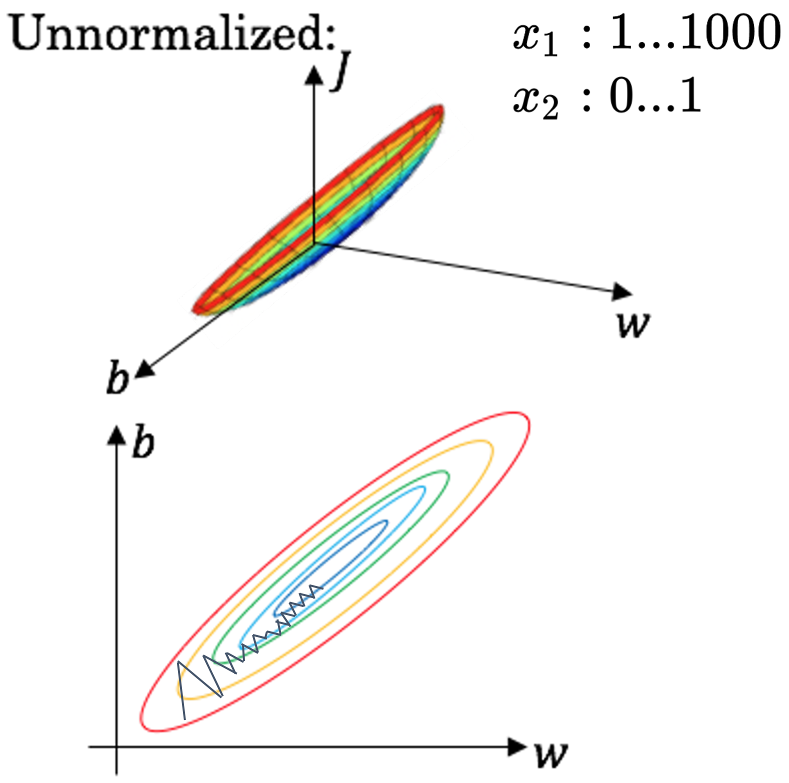
Note: This is not as good as adding independent examples to the dataset

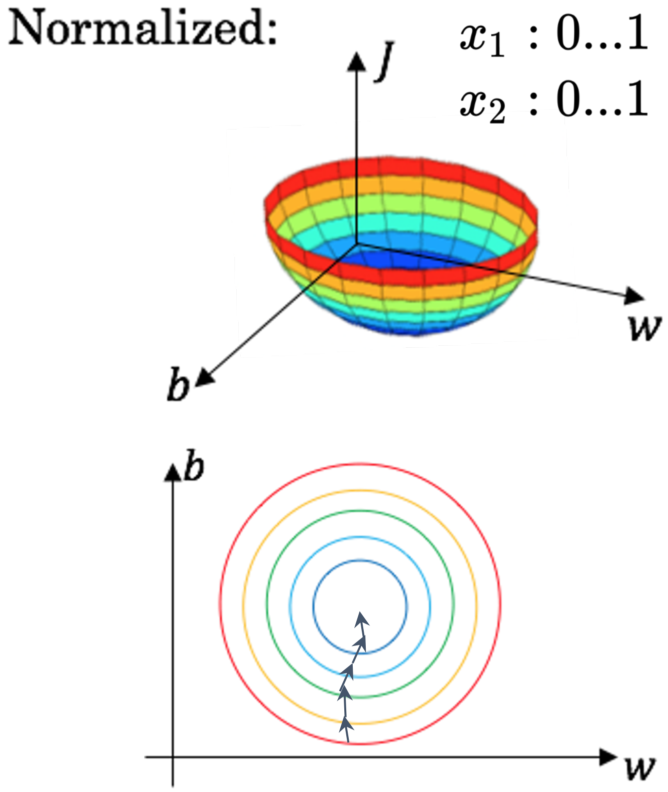
Source: https://www.coursera.org/learn/deep-neural-network/

Data normalization

Why is dataset normalization necessary?

Input features of very different sizes will make the optimization take a very long time as the weight will be in very different ranges as well.

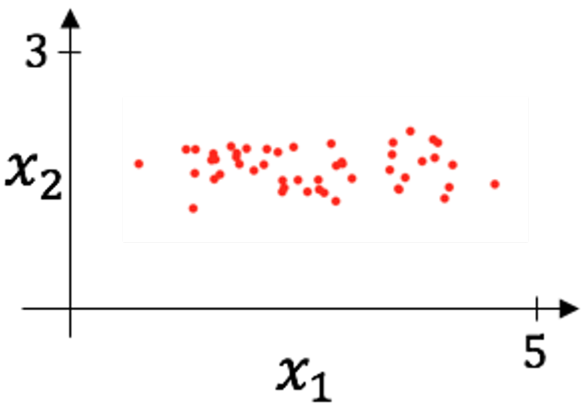


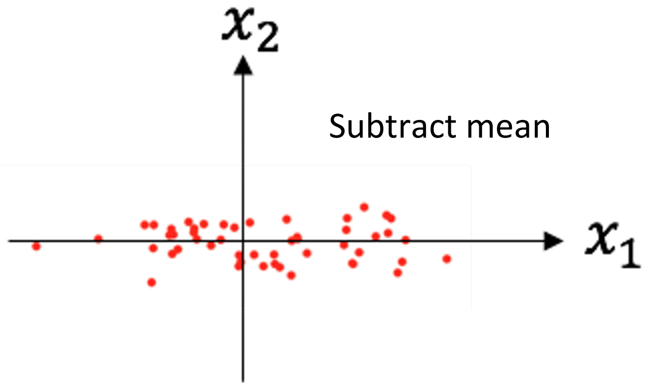


Source: https://www.coursera.org/learn/deep-neural-network/

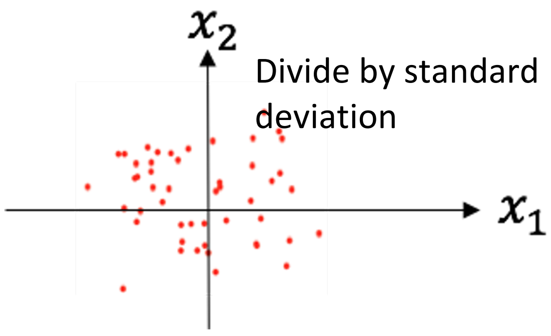
Data normalization

Dataset normalization is an important step in most machine learning methods.





Subtract mean



Divide by standard deviation

$$\mu = \frac{1}{n} \sum_{i-1}^{m} x_i$$

$$x := x - \mu$$

$$\sigma^2 = \frac{1}{n} \sum_{i-1}^{m} x_i^2$$

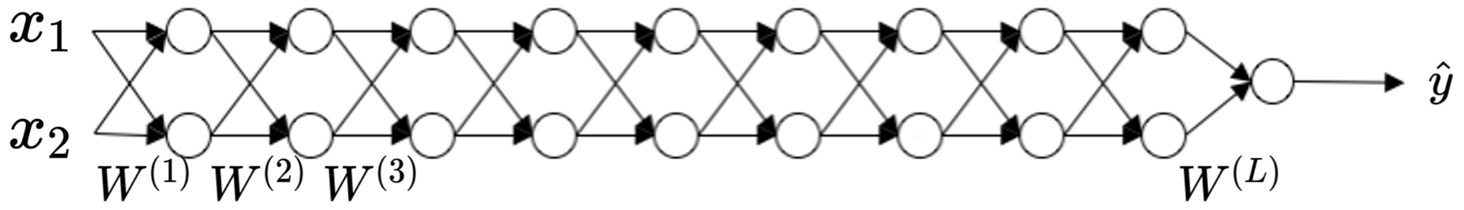
$$x := \frac{x}{\sigma}$$

Remember to normalize the test data as well: $x_{test} = \frac{x_{test} - \mu}{\sigma}$

Source: https://www.coursera.org/learn/deep-neural-network/

Vanishing / Exploding gradients

- Example network with L hidden layers
- For simplicity, we use a linear activation function for all neurons: $g(z) = z$



$$\hat{y} = W^{(L)} W^{(L-1)} \dots W^{(3)} W^{(2)} W^{(1)} x$$

Exploding gradients:

$$W^{(1)} = W^{(2)} = \dots = W^{(L-1)} = \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}$$

$$\hat{y} = W^{(L)} W^{L-1} x \sim W^{(L)} 1.5^{L-1} x \rightarrow \inf$$

Vanishing gradients:

$$W^{(1)} = W^{(2)} = \dots = W^{(L-1)} = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}$$

$$\hat{y} = W^{(L)} W^{L-1} x \sim W^{(L)} 0.5^{L-1} x \rightarrow 0$$

The activations g(z) are used in backward propagation which leads to the exploding/vanishing gradient problem. The gradients of the cost with respect to the parameters are too big/small.

Source: http://www.deeplearning.ai/ai-notes/initialization/

Weight Initialization problems

- Initial parameters needs to "break symmetry".
  - 2 hidden units connected to same input must have different initial parameters.
  - Initializing all neurons with 0 leads to all neurons learning the same features.
- Previously used: Random initializing weight from a standard normal distribution N(0,1).
- The bias value depends only on the linear activation of that layer, and not on the gradients of deeper layers. Bias weights can therefore be initialized to a constant value, typically 0.

- http://www.deeplearning.ai/ai-notes/initialization

Source: http://www.deeplearning.ai/ai-notes/initialization/

Weight Initialization

To prevent vanishing/exploding gradients, we need to follow 2 rules:

- Mean of activations = 0 $E[a^{(l-1)}] = E[a^{(l)}]$
- Variance of activations should be equal across every layer $Var[a^{(l-1)}] = Var[a^{(l)}]$

Xavier initialization

- Zero weight for all bias weights $b^{(l)} = 0$
- Weights from a normal distribution with variance depending on number of neurons in the previous layer $W^{(l)} \sim \mathcal{N}(\mu = 0, \sigma^2 = \frac{1}{n^{l-1}})$

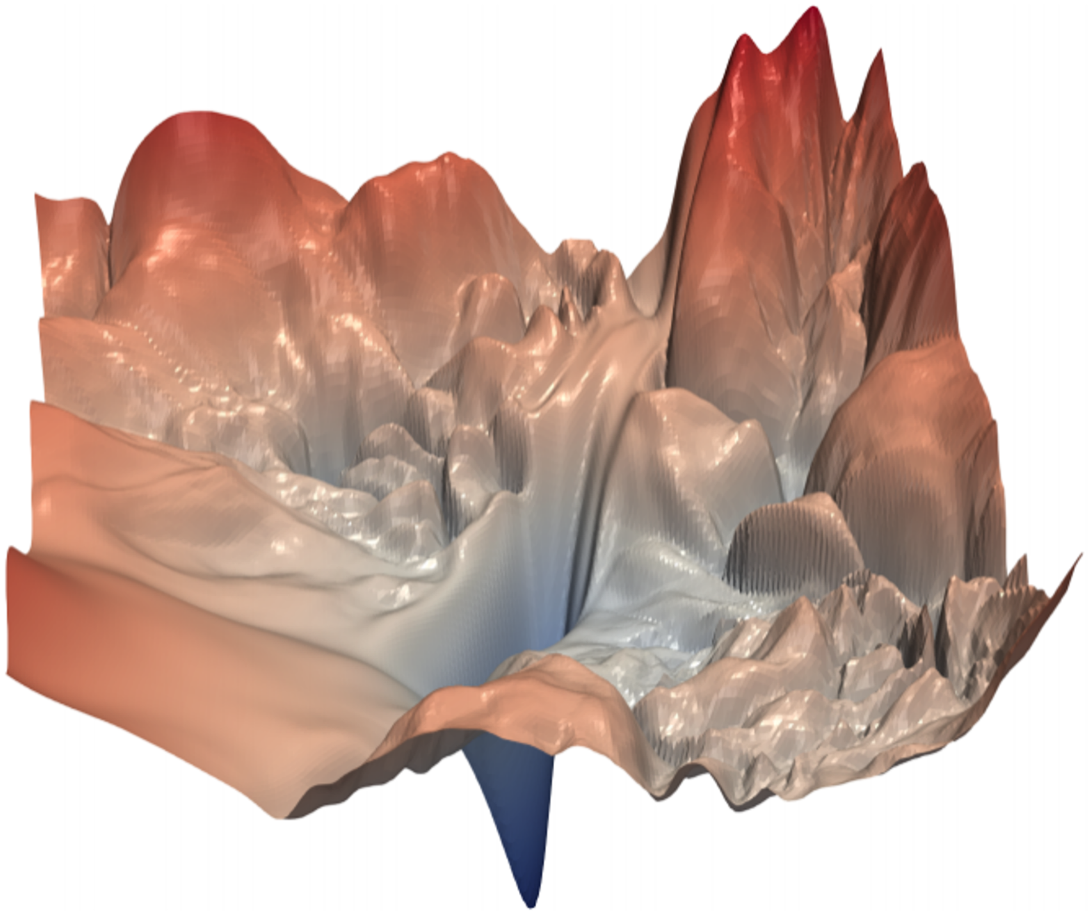
For ReLU it is better to use $\sigma^2 = \frac{2}{n^{l-1}}$

Another method to prevent vanishing/exploding gradients is Gradient Clipping

1. Gradients over a threshold are clipped to the threshold.
2. Specifying a vector norm for the gradients which the derivatives are scaled to.

Source: http://www.deeplearning.ai/ai-notes/initialization/

Loss landscape of Neural Nets



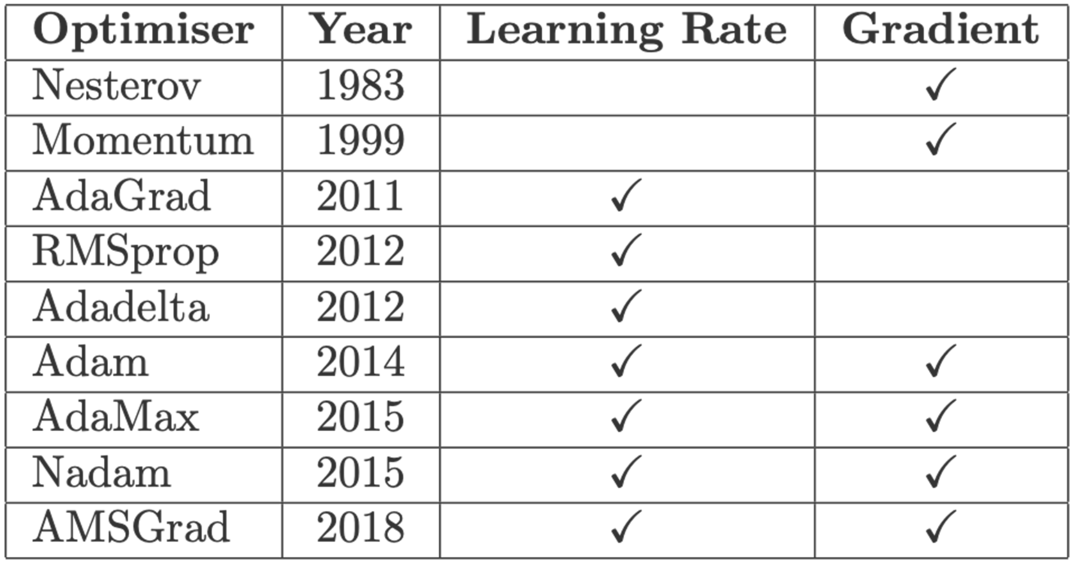
Visualizing the Loss Landscape of Neural Nets, NeurIPS 2018

Optimization Algorithms

Variants of gradient descent act either on the learning rate or the gradient itself

Typically search for the method which is best suited for your problem via trial and error

- *Gradient Descent*
- Stochastic Gradient Descent
- Mini Batch Gradient Descent
- Momentum
- RMSprop
- Adam optimization



Optimiser	Year	Learning Rate	Gradient
Nesterov	1983		✓
Momentum	1999		✓
AdaGrad	2011	✓	
RMSprop	2012	✓	
Adadelta	2012	✓	
Adam	2014	✓	✓
AdaMax	2015	✓	✓
Nadam	2015	✓	✓
AMSGrad	2018	✓	✓

Gradient Descent is not only Backprop

Backpropagation “Backprop” refers to the process of backpropagating the error through a computational graph to compute the gradients of the error function with respect to the weights of the graph/network.

Gradient Descent iterative optimization algorithm to find the minimum of a function. The algorithm takes steps proportional to the negative of the size of the gradient.

1. Loop until convergence, optimizing **w** (and bias weights)
   a. Compute gradients $\frac{\partial J}{\partial w}$
   b. Update weights, $w_{t+1} \leftarrow w_t - \lambda \frac{\partial J}{\partial w}$
2. Return weights **w**

Mini-batch Gradient Descent

- Stochastic gradient descent performs an update for **each** training example
- Mini-batch GD is a mixture where the cost is computed from *random* subsets of the training data (“mini-batches”)

Gradient descent:

$$w_{t+1} = w_t + \lambda \frac{\partial}{\partial w_t} \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

Mini-batch GD:

$$\approx w_t + \lambda \frac{\partial}{\partial w_t} \frac{1}{M} \sum_{i=1}^{M} (y_i - \hat{y}_i)^2, \qquad M \ll N$$

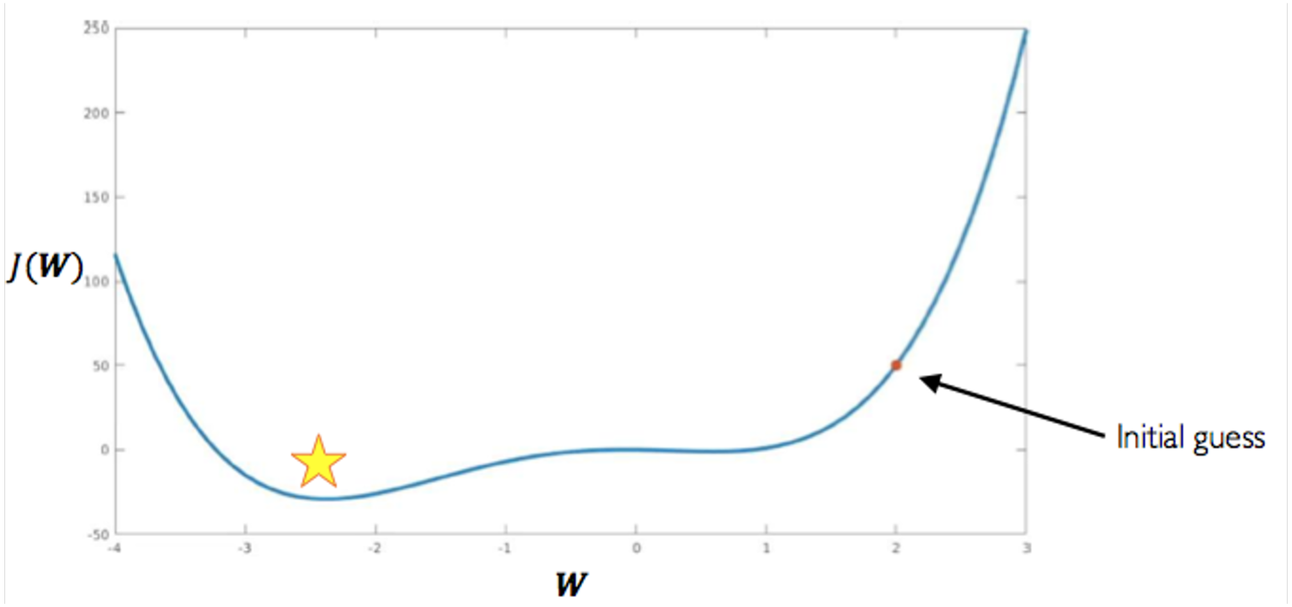
- More efficient in terms of memory consumption and computational cost
- Problem: fluctuating cost function which might increase in some mini-batches.
- Epoch: Entire dataset passed through the network one time.

Setting the learning rate of GD

With Gradient Descent: Try lots of different learning rates to see what works the best

- **Small learning rate:** Converges slowly and gets stuck in false local minima
- **Large learning rate:** Overshoots, becomes unstable and diverges
- **Stable learning rate:** Converges smoothly and avoids local minima

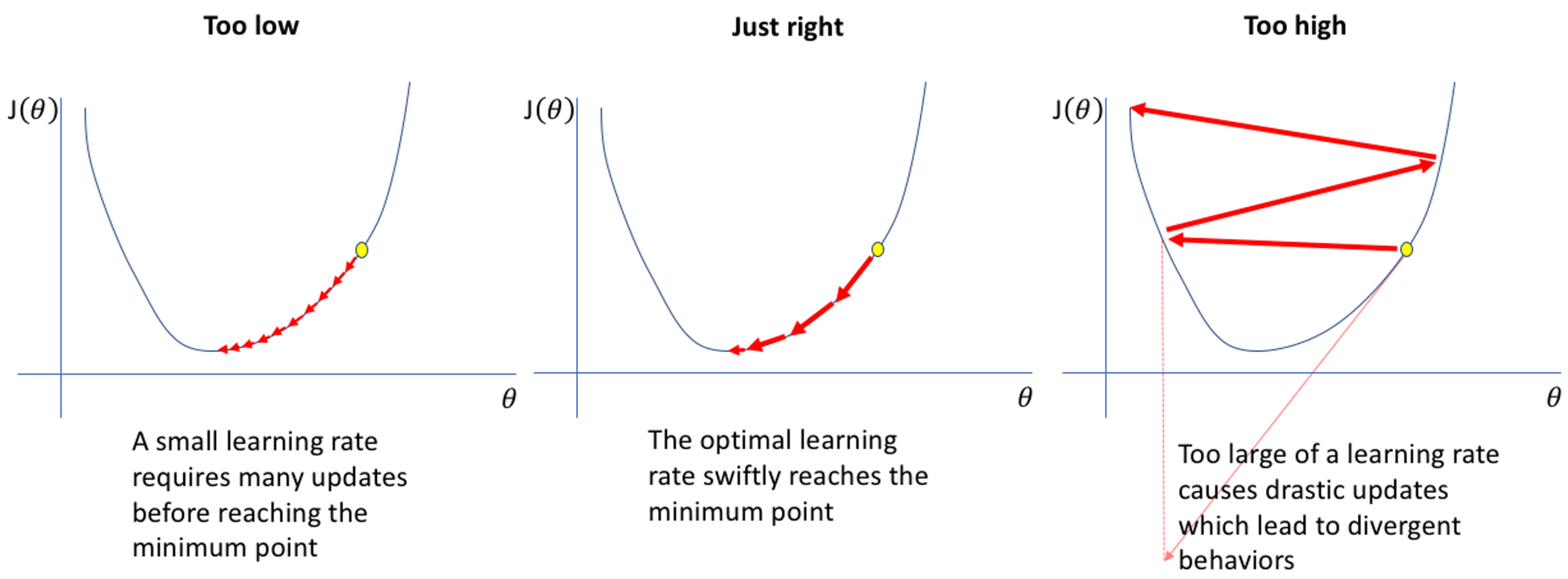
Visualization: https://www.deeplearning.ai/ai-notes/optimization/



Source: http://introtodeeplearning.com/ https://www.deeplearning.ai/ai-notes/optimization/

Learning rate annealing

With Gradient Descent: Try lots of different learning rates to see what works the best


Too low
Just right
Too high
$J(\theta)$
θ
A small learning rate requires many updates before reaching the minimum point
The optimal learning rate swiftly reaches the minimum point
Too large of a learning rate causes drastic updates which lead to divergent behaviors

- When Gradient Descent nears a minima in the cost surface, the parameter values can oscillate back and forth around the minima.
- Slow down the parameter updates by decreasing the learning rate
- This could be done manually, however automated techniques are preferable

Learning rate annealing: Adagrad

$$w_{t+1} = w_t + \frac{\lambda}{\sqrt{S_t + \epsilon}} \frac{\partial L}{\partial w_t}$$

$$S_t = S_{t-1} + \left[\frac{\partial L}{\partial w_t} \right]^2$$

- Adapt **learning rate** by dividing with the cumulative sum of current and past squared gradients *for each feature independently* (large update for infrequent features, smaller update for frequent features.
- This is beneficial for training since the scale of the gradients in each layer is often different by several orders of magnitude.
- Problem: Learning rate will eventually become close to 0 due to the accumulated sum in the denominator.

Adaptive Learning rate: RMSprop

Root mean square prop, solves the diminishing learning rate in Adagrad.

Adapt learning rate by dividing with the root of squared gradients.

Uses the moving average to smooth over multiple mini-batches.

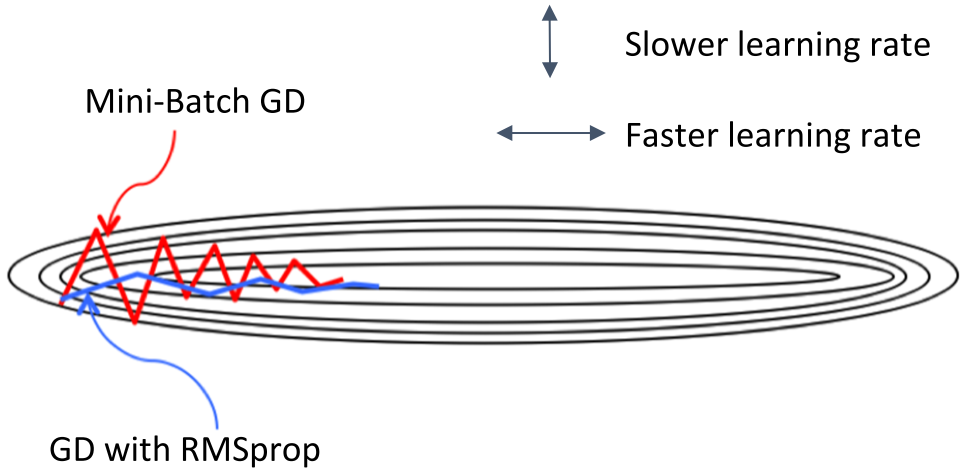
- Higher learning rate is possible as RMSprop smoothens the gradient direction

Default RMSprop values: $\beta = 0.999$

$\epsilon = 10^{-8}$

$$s_t = \beta s_{t-1} + (1-\beta)\left(\frac{\partial J}{\partial w}\right)^2$$

$$w_{t+1} \leftarrow w_t - \frac{\lambda}{\sqrt{s_t}+\epsilon}\frac{\partial J}{\partial w}$$



Works the same way as Adadelta (developed at the same time independently)

Gradient smoothing: Momentum

Smoothen out the fluctuation in gradient direction from using mini-batches.

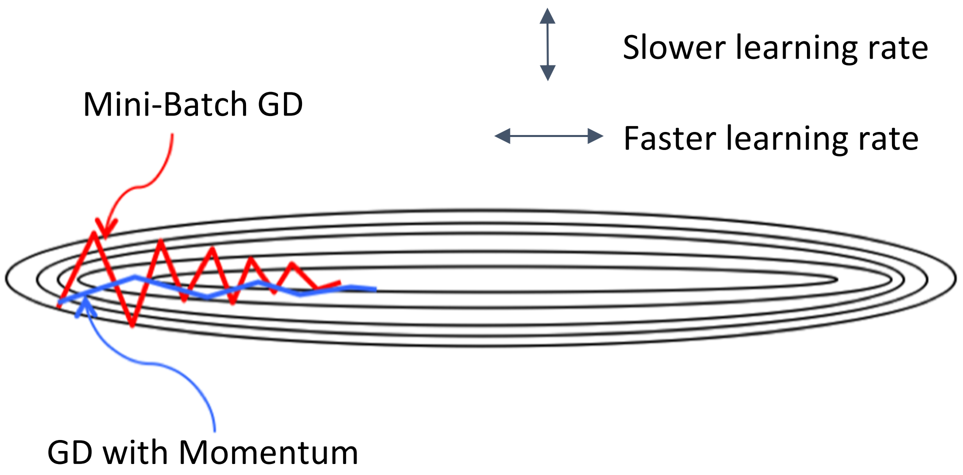
How?

- Adding a fraction of the update vector of the past time step to the current update vector.
- Momentum increases for dimensions whose gradients point in the same direction.
- Momentum decreases dimensions whose gradients change directions.
- Faster convergence and reduced oscillation.

Default momentum value: $\gamma = 0.9$

$$m_t = \gamma m_{t-1} + (1-\gamma)\frac{\partial J}{\partial w}$$

$$w_{t+1} \leftarrow w_t - \lambda m_t$$



Adam: Adaptive moment estimation

Putting together Momentum and RMSprop and adding bias-corrected first and second moment estimates:

$$\gamma = 0.9$$
$$\beta = 0.999$$
$$\epsilon = 10^{-8}$$

$$m_t = \gamma m_{t-1} + (1-\gamma)\frac{\partial J}{\partial w}$$ "Momentum"

$$s_t = \beta s_{t-1} + (1-\beta)(\frac{\partial J}{\partial w})^2$$ "RMSprop"

$$\hat{m}_t = \frac{m_t}{1-\gamma} \qquad \hat{s}_t = \frac{s_t}{1-\beta}$$ Bias corrected

$$w_{t+1} \leftarrow w_t - \frac{\lambda}{\sqrt{\hat{s}_t+\epsilon}}\hat{m}_t$$

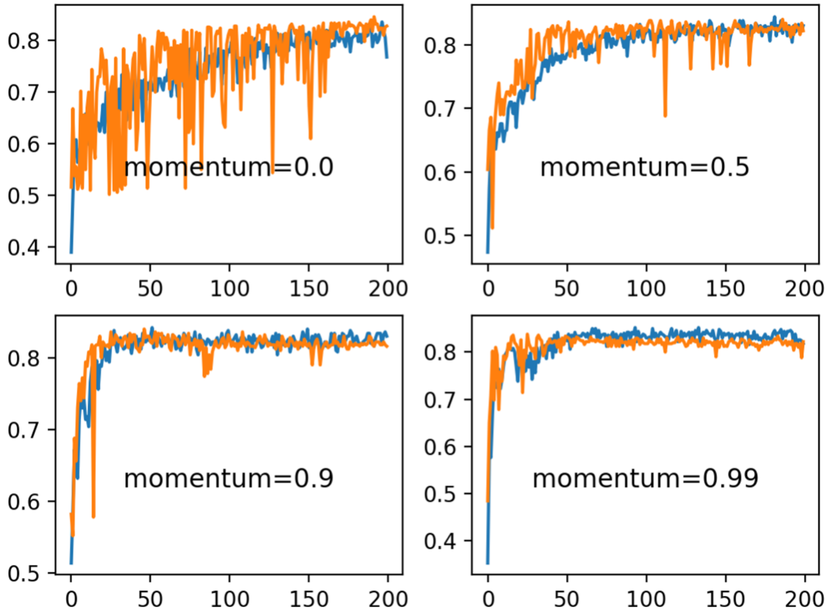
If the bias correction is not used, the momentum and RMSprop terms are biased towards zero, especially during the initial time steps, or using small decay rates (hyperparameters close to 1)

Optimization Algorithm comparison

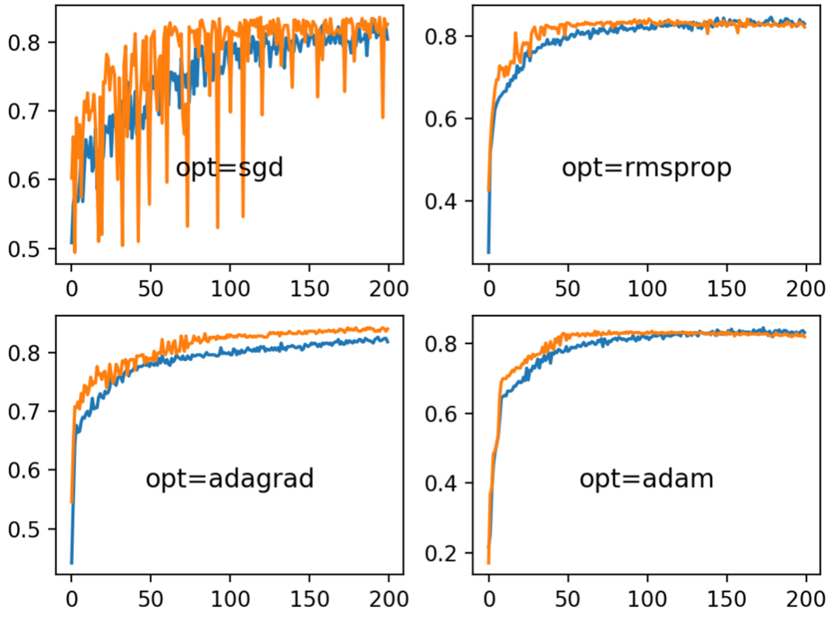
Line plots of train and test data accuracy

Visualization comparison https://www.deeplearning.ai/ai-notes/optimization/

Gradient smoothing



Adaptive learning rates



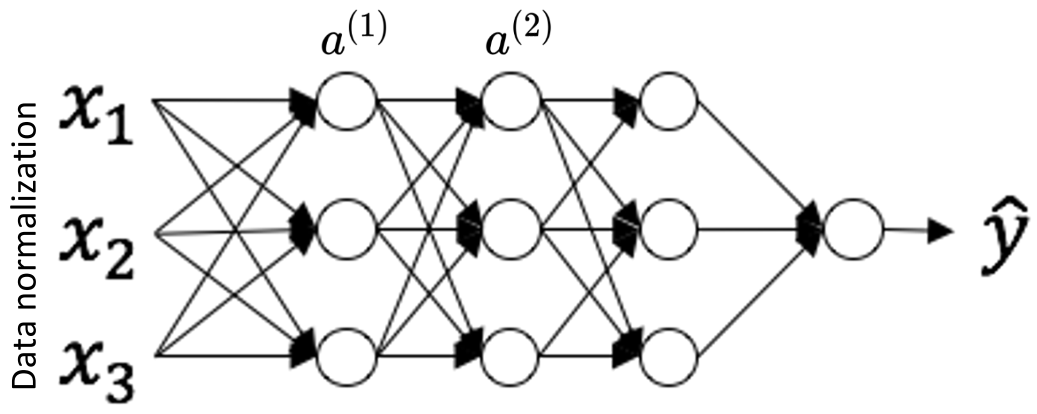
Source: https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks/

Batch Normalization

- Normalizing activations in a network.
- Normalize before or after the activation function is possible (mostly done before).

- The idea is to normalize the activation outputs for faster training.
- Make sure that the activation function input is in a "good" range.

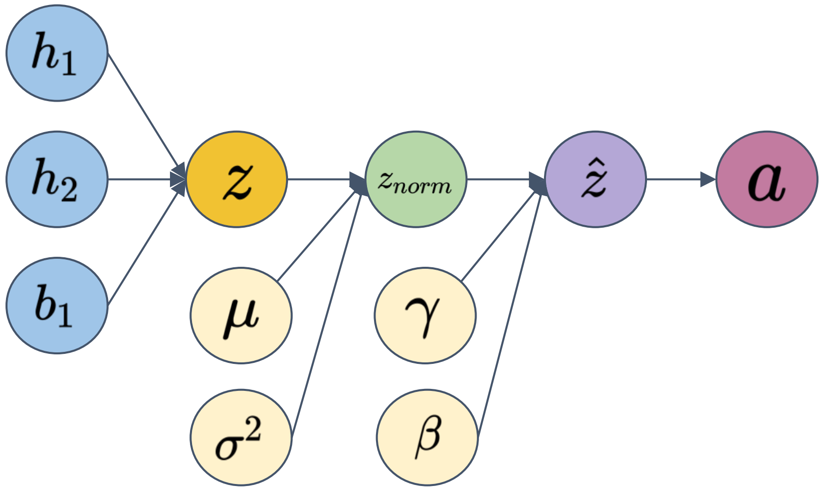
$$a^{(i)} = g(z^{(i)})$$



Batch Normalization

Normalizing the mean and standard deviation of a unit can reduce the expressive power of the neural network. To maintain expressive power, 2 learnable hyperparameters are introduced.

- Old parameterization (computing mean/variance): determined by interaction between parameters in the layers below.
- New parametrization: mean and variance is determined solely by the hyperparameters



$$z = w^T h + b$$

$$z_{norm} = \frac{z-\mu}{\sqrt{\sigma^2+\epsilon}}$$

$$\hat{z} = \gamma z_{norm} + \beta$$

$$a = g(\hat{z})$$

Mean and variance is computed based on the z values in a hidden layer for each mini-batch.

For test time: weighted average across mini-batches are used for the mean and variance values.

Summary

- Hyperparameters should be tuned on a validation set (not on the final test set)
- For regularization, mainly use:
  - Early stopping
  - Dropout
  - Data augmentation
- Choose an optimization algorithm that suits your needs (or use a general good optimizer like Adam)
- Start by tuning the learning rate
- Try out with different batch sizes (from 1 to dataset-size)

Credits

Books:

- https://www.deeplearningbook.org/
- http://neuralnetworksanddeeplearning.com/

Online Course from MIT:

- http://introtodeeplearning.com/

Online course from Stanford University:

- https://www.coursera.org/specializations/deep-learning?

Other

- cs231n.github.io
- appliedgo.net
- brohrer.github.io
- learnopencv.com