Techniques for Improving Deep Neural Networks

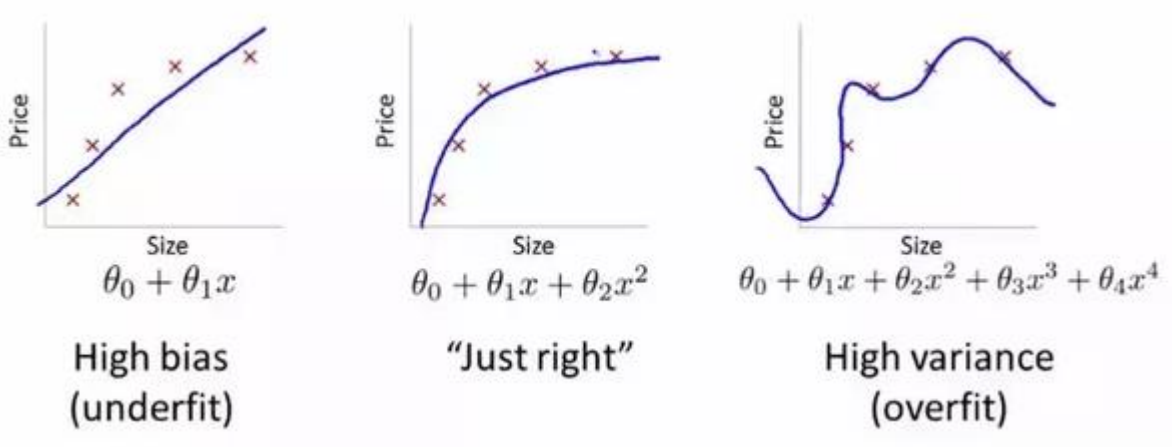
SUTD EPD pillar Big data meeting

Lahiru Jayasinghe
aruna_Jayasinghe@sutd.edu.sg

Content

- Bias/Variance concept
- Regularization
- Initialization
- Normalization
- Optimization
- Hyperparameter Tuning

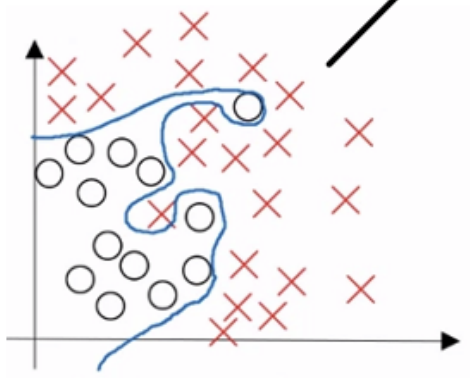
Bias/Variance concept



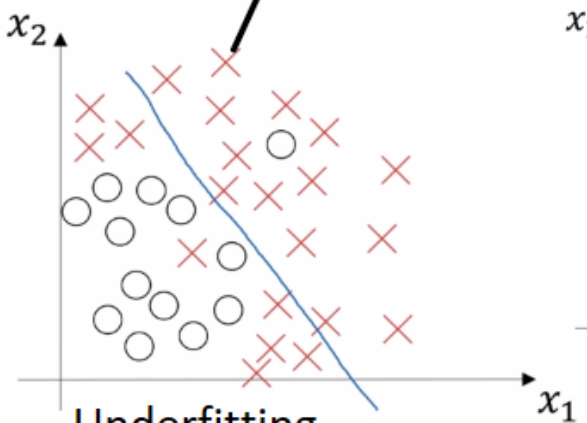
- Bias are the simplifying assumptions made by a model to make the target function easier to learn
- Bias models generalize well, but doesn't fit the data perfectly (under-fitting)
  - High-Bias models → linear regression
- Variance is the amount that the estimate of the target function will change if different training data was used.
- High variance models fits the training data too much (overfitting)
  - High-Variance → Decision Tree, k-NN
- Therefore, balancing the Bias and the Variance of a model is mandatory.

How to check the bias-variance problem

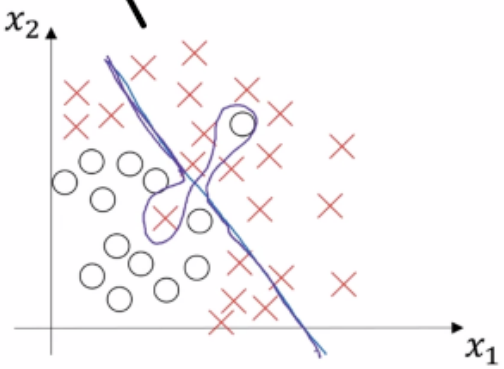
Train set error :	1%	15%	15%	0.5%
Test set error :	11%	16%	30%	1%
Bias/Variance :	High variance	High bias	High bias and High variance	Low bias and Low variance



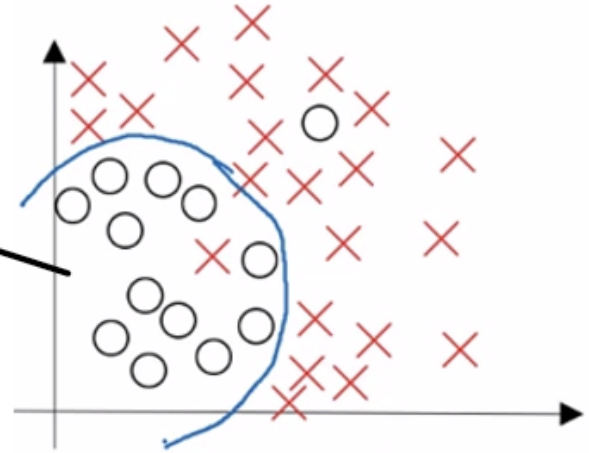
overfitting



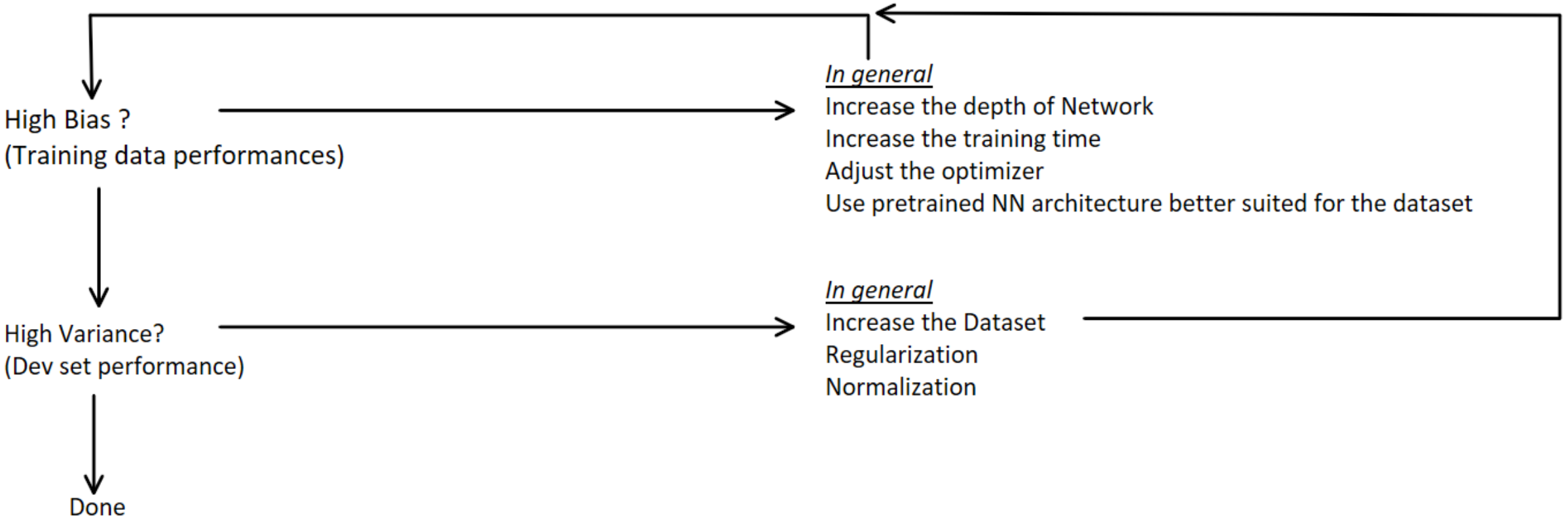
Underfitting



Some part is overfitted and some part is underfitted



Bias-variance in Deep Learning



Not like other Machine Learning algorithms Deep Learning algorithms has the ability to change the bias and variance independently

Regularization

Regularization

If the cost = $J(w, b)$ Then objective function of training = $\min_{w,b} J(w, b)$

<u>For a Neural network</u>

$$J\left(w^{[1]}, b^{[1]}, \dots, w^{[L]}, b^{[L]}\right) = \frac{1}{m}\sum_{i=1}^{m} L(y^{(i)}, x^{(i)})$$

With regularization term, weights will be penalizes

$$J\left(w^{[1]}, b^{[1]}, \dots, w^{[L]}, b^{[L]}\right) = \frac{1}{m}\sum_{i=1}^{m} L(y^{(i)}, x^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L}\left\|w^{[l]}\right\|_F^2 \qquad \text{Where } \left\|w^{[l]}\right\|_F^2 = \sum_{i=1}^{n^{[l-1]}}\sum_{j=1}^{n^{[l]}}(w_{i,j}^{[l]})^2$$

Sometimes people use L1 regularization

$$\frac{\lambda}{2m}\sum_{j=1}^{n}\left|w_j\right| = \frac{\lambda}{2m}\|w\|_1$$

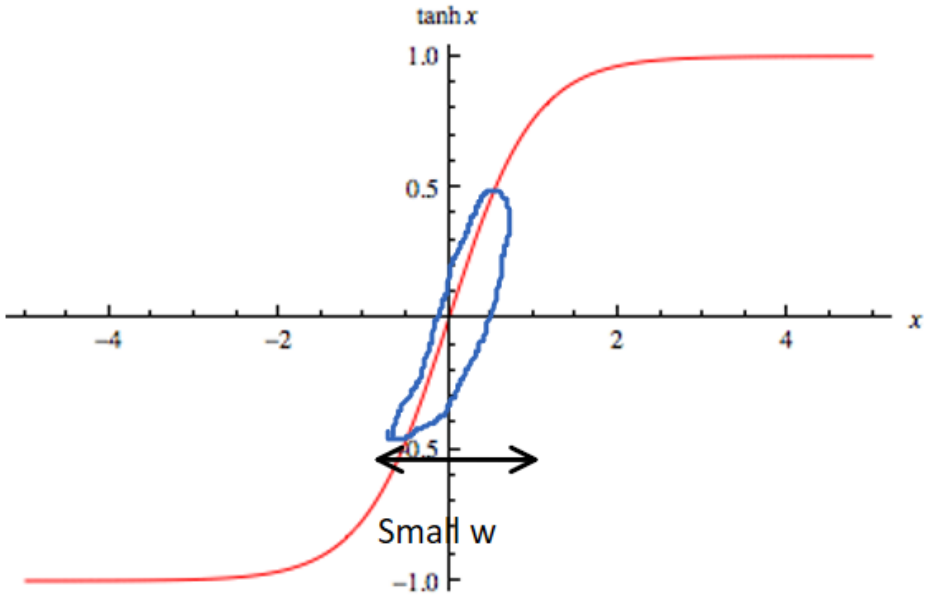
This kind of regularization induce sparsity in neurons. With this, weights tend to become zero or very close to zero in the training process

Regularization

$$J(w^{[L]}, b^{[L]}) = \frac{1}{m}\sum_{i=1}^{m} L(y^{(i)}, x^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L}\|w^{[l]}\|_F^2$$

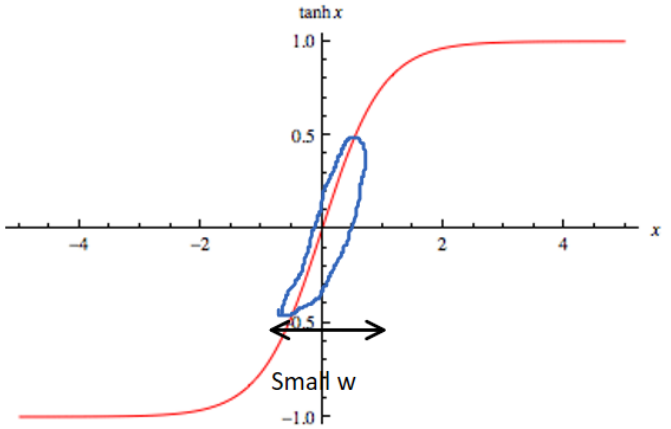
If $\boldsymbol{\lambda}$ is too large then even if the loss is very small, the cost or objective function value become very huge value. Therefore, only remedy for the optimizer is to minimize the $\boldsymbol{w}$

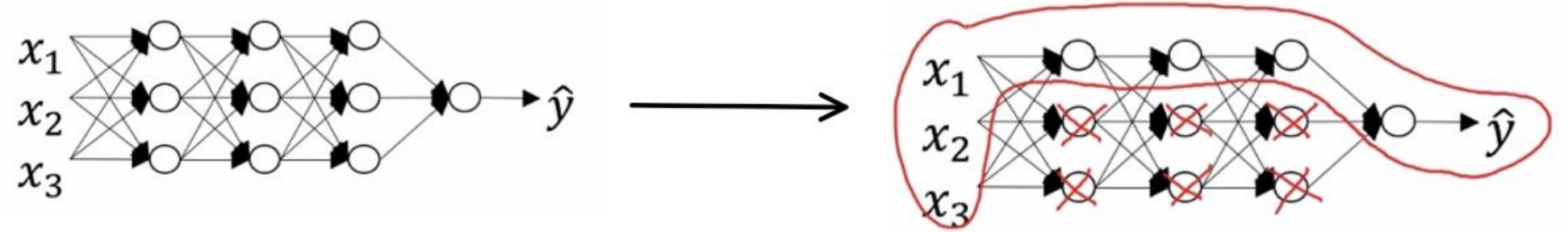
According to $z^{[l]} = \tanh(w^{[l]}a^{[l-1]} + b^{[l]})$ z[l] operates in a linear region in the tanh activation function

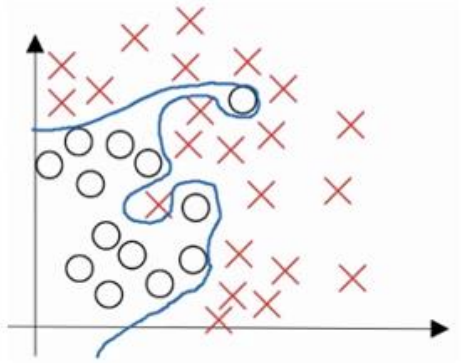


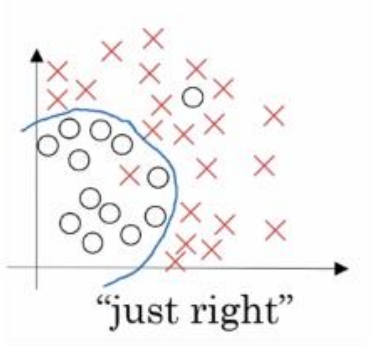
Regularization

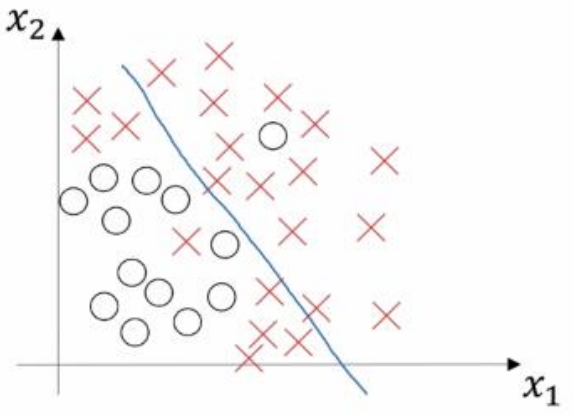
When λ --> ∞ then w --> 0







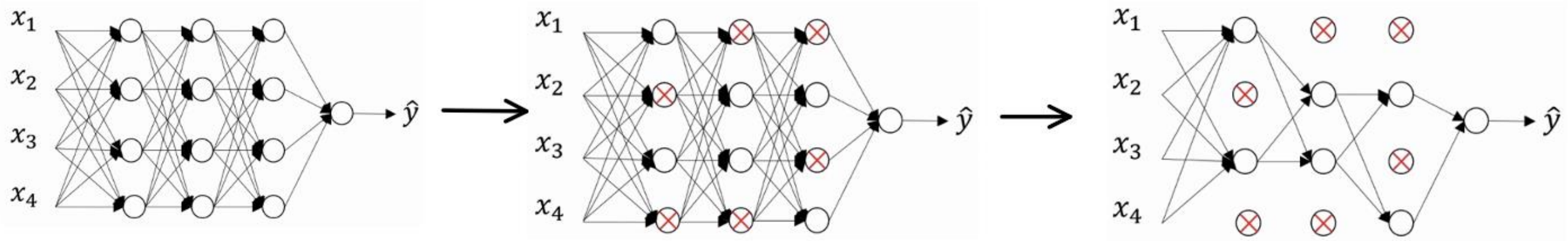




Overfitting,
contain too complex decision boundary

Underfitting
Contain simple decision boundary

Dropout Regularization

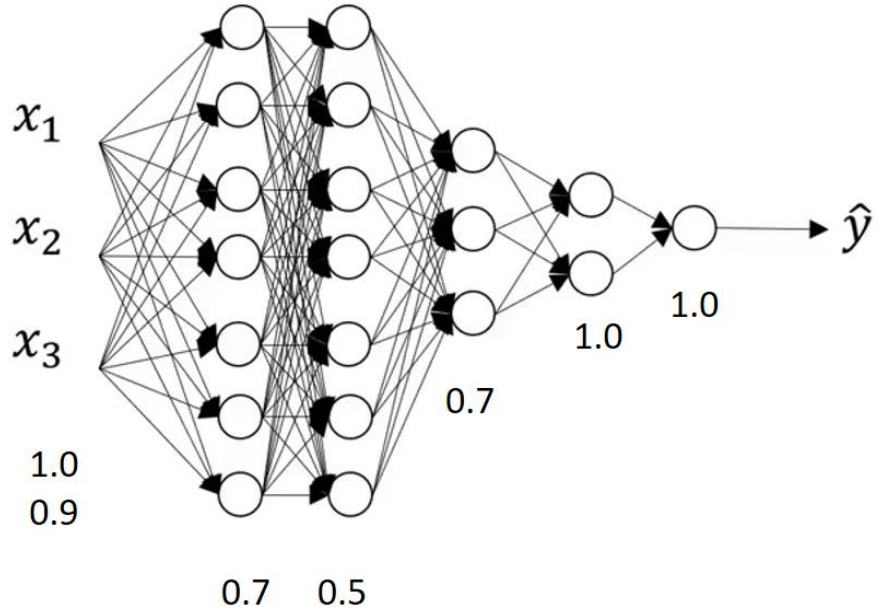


Each of these layers have a p probability of keeping each node and $1 - p$ probability of removing each node
After you decided the nodes, you remove the ingoing and outgoing links from those selected nodes

$$a = D \odot \sigma(Z)$$

Where 'a' is the output activation, D is a vector of Bernoulli variables and $\sigma(Z)$ is the intermediate activation of the neuron before dropout. $Bernulli\ random\ variable\ f(k,h) = \begin{cases} p\ \ if\ k = 1 \\ 1 - p\ if\ \ k = 0 \end{cases}$

Apply Dropout



- Now the activation is reduced by $(1 - p)\%$ at every hidden node. Therefore, the outputs of the training will be scaled and in the testing phase, you might have to do the necessary scaling.

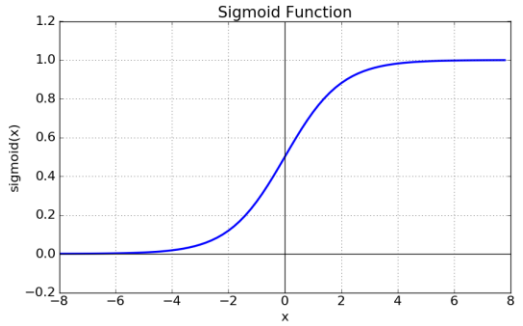
- This is an extra overhead. In order to keep the testing phase simple, every hidden layer activation need to be rescale to match the expected value

- $a = \frac{a}{p}$
- This is know as inverted dropout

- Now in the testing phase, simply make $p = 1$. Otherwise it will introduce a noise to the output in the testing phase

Weight Initialization

Weights Initialization



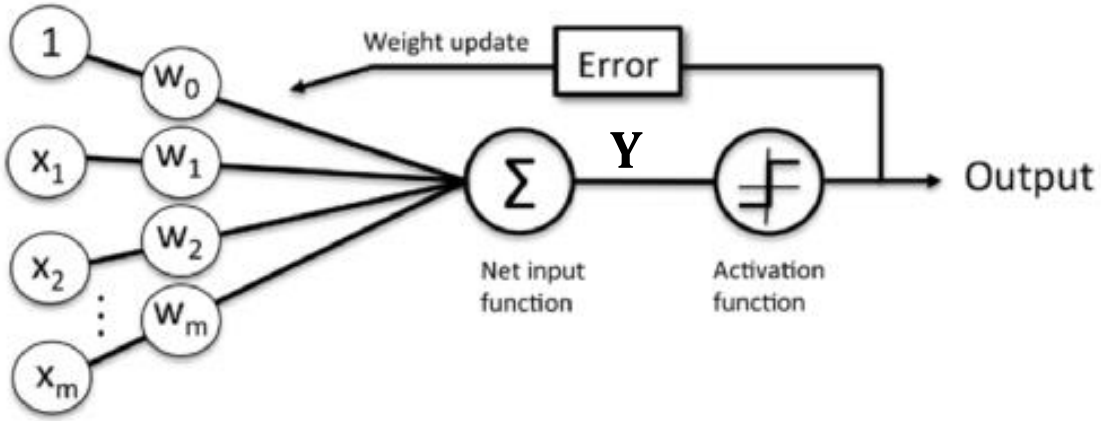
<u>If the weights are too small</u>

- Then the variance of the input signal starts diminishing as it passes through each layer in the network. The input eventually drops to a really low value and can no longer be useful.
- If we consider sigmoid as the activation function, then we know that it is approximately linear when we go close to zero.
- This basically means that there won't be any non-linearity. If that's the case, then we lose the advantages of having multiple layers.

<u>If the weights are too large</u>

- Then the variance of input data tends to rapidly increase with each passing layer.
- Since the sigmoid function has almost a constant value or saturated region (1 or 0) for large values, the gradients will start approaching zeros for large weights

Weights Initialization



Assume that we are sampling weights from gaussian distribution, then

$$Y = (w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_m x_m)$$

With each passing layer, we want the variance to remain the same. This helps us keep the signal from exploding to a high value or vanishing to zero.

Weights Initialization

Then ideally

$$var(Y) = var(w_0 + w_1x_1 + w_2x_2 + \cdots + w_mx_m)$$

If you consider a general term, we have

$$var(w_ix_i) = E(x_i)^2 var(w_i) + E(w_i)^2 var(x_i) + var(w_i)var(x_i)$$

Since weights are from a gaussian distribution of zero mean

$$var(w_ix_i) = var(w_i)var(x_i)$$

Note that 'w_0' is a constant and has zero variance, we can simplify the variance as.

$$var(Y) = var(w_1)var(x_1) + \cdots + var(w_m)var(x_m)$$

Weights Initialization

Since they are identically distributed

$$var(Y) = m * var(w_i)var(x_i)$$

So if we want the $var(Y) = var(x)$, then the

$$var(w_i) = \frac{1}{m}$$

This is known as the Xavier initialization formula

The authors take the average of the number of input neurons and the output neurons so

then it becomes, $var(w_i) = \frac{2}{m_{in}+m_{out}}$

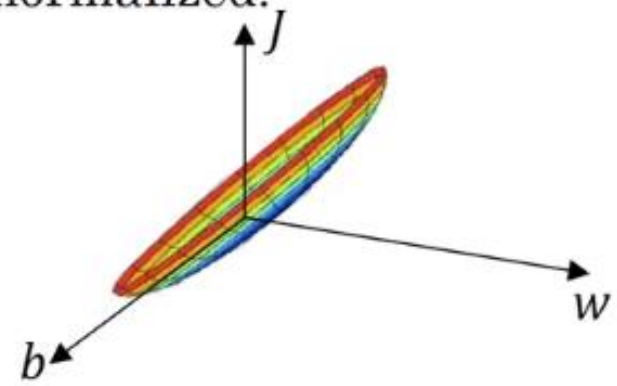
When you use ReLu activation $var(w_i) = \frac{2}{m}$

Normalization

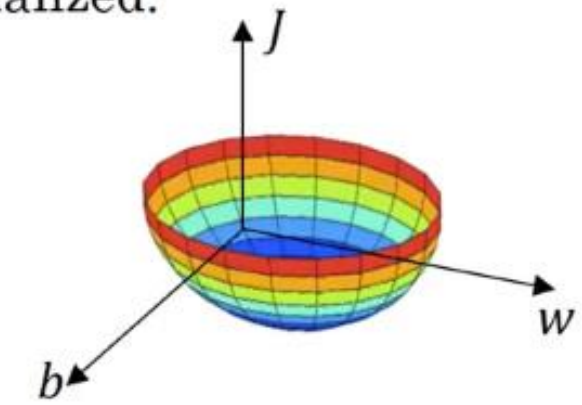
Training set - Normalization

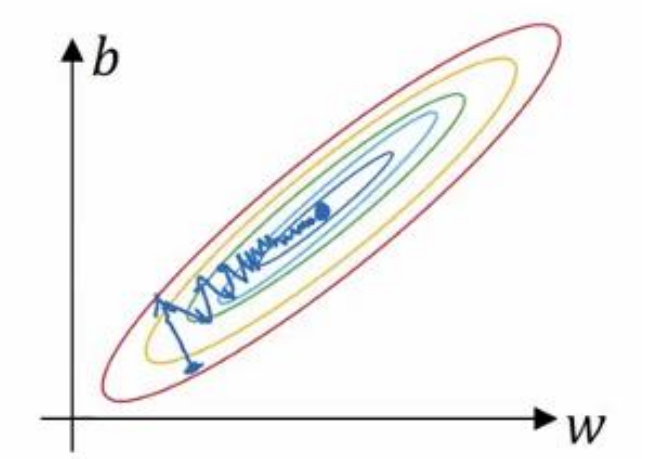
$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right)$$

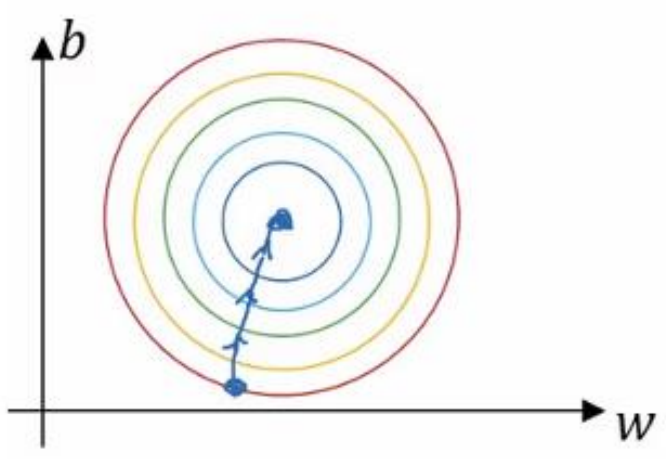
Unnormalized:



Normalized:

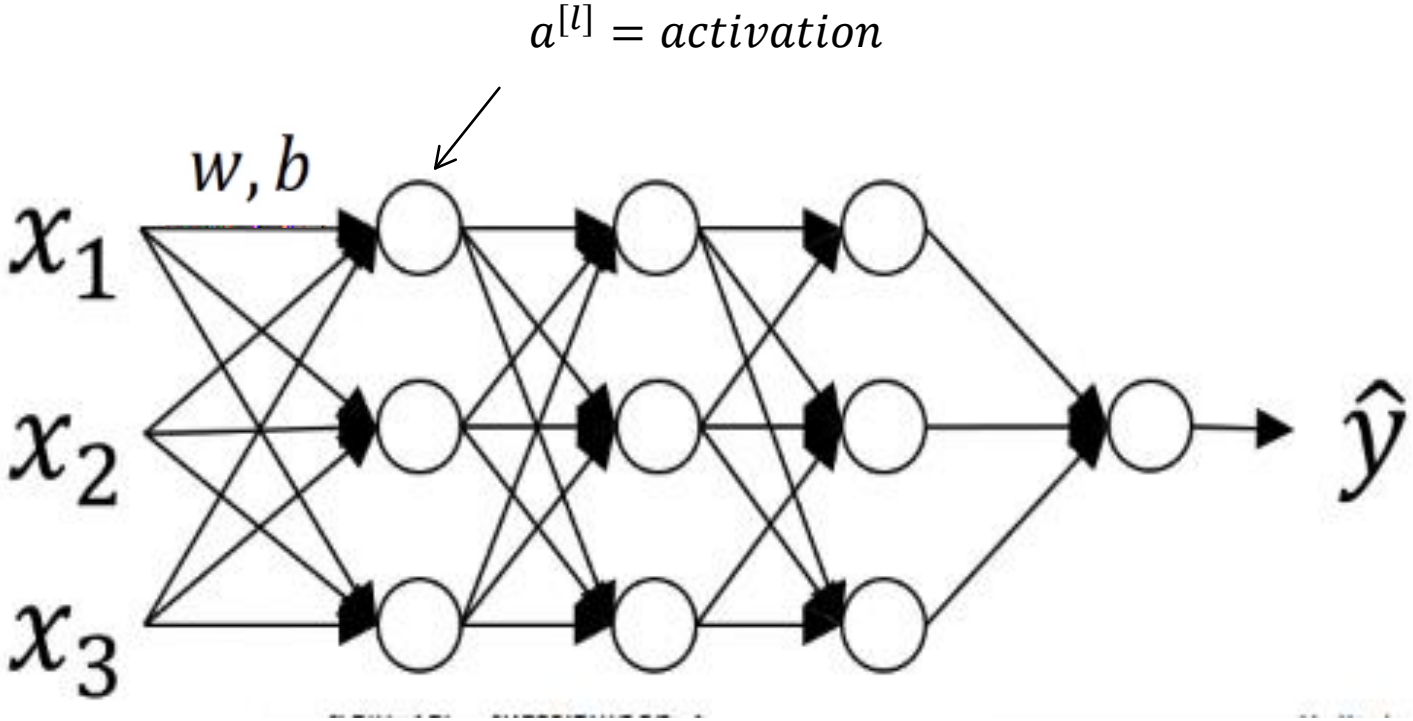






Use the training mean and variance to normalize the test set also

Batch Normalization



$Z = wX + b$ activation : $a^{[1]} = f(Z)$

Batch Normalization over a mini-batch size of : m

$$\mu = \frac{1}{m}\sum Z^{[i]}$$

$$\sigma^2 = \frac{1}{m}\sum\left(Z^i - \mu\right)^2$$

$$Z^{[i]}_{norm} = \frac{Z^{[i]} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

Batch Normalization

But now the hidden units have same value distribution. Therefore, it could restrict the neural network from creating complex decision boundaries at the output layer.

Solution:

$$\hat{Z}^{[i]} = \gamma Z_{norm}^{[i]} + \beta \qquad \gamma \; and \; \beta \; are \; learnable \; parameters$$

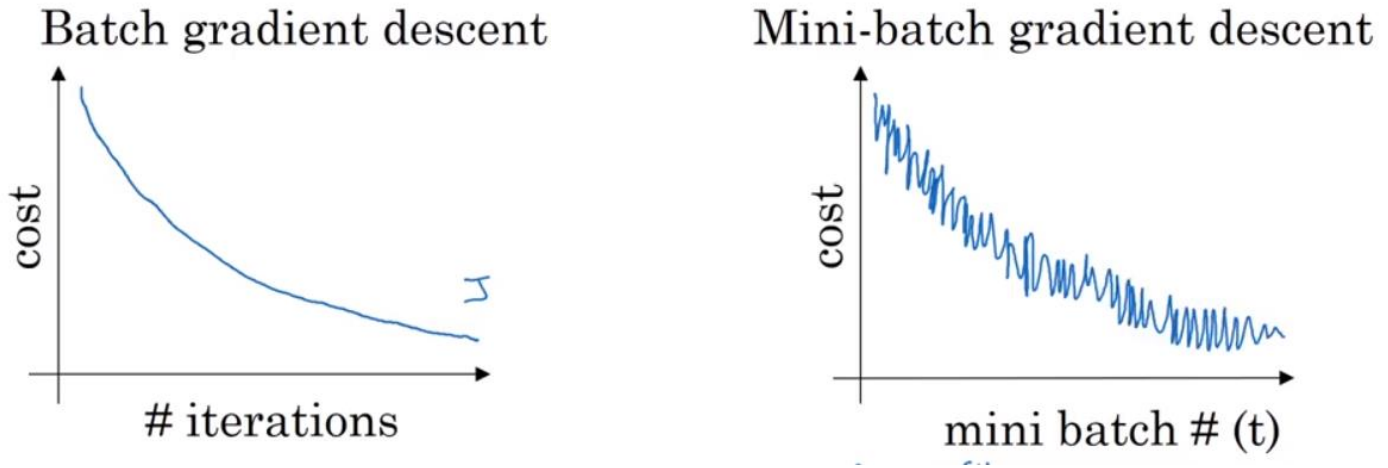
If $\gamma = \sqrt{\sigma^2 + \varepsilon} \; and \; \beta = \mu \;\; then \;\; \hat{Z}^{[i]} = Z^{[i]}$

At the test time we do not have a mini-batch. Then it's impossible to calculate the $Z_{norm}^{[i]}$

The solution is during the training, a separate $\mu_{test} \; and \; \sigma_{test}^2$ need to be calculated using exponentially weighted average. Those calculated variable will be used during the testing phase.

Optimization

Mini-batch Gradient Descent



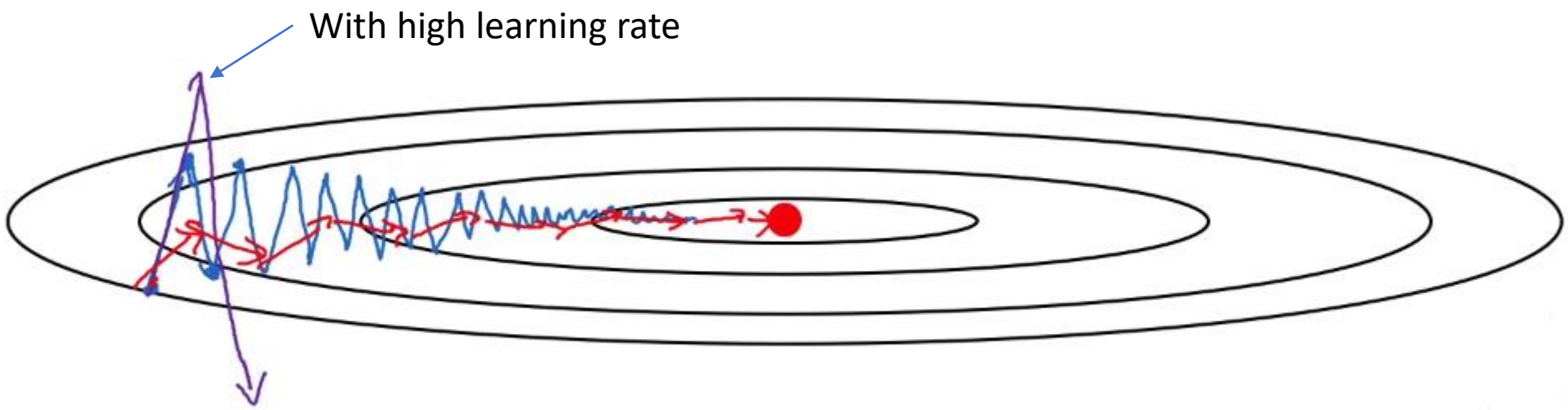
Steps for Mini-batch Gradient Descent

Shuffle the training sets synchronously between X and Y to ensure that examples will be split randomly into mini-batches. (maybe not applicable for time series data)

Partition the shuffled batch into different mini batches of size as the power of 2 (64, 128, 256, 512, 1024…etc.). It is because sometimes algorithm work faster if it matches the design of computer memory.

Make sure your batch size fit to the CPU/GPU memory
If small training set: use batch gradient descent

Improvements to Gradient Descent



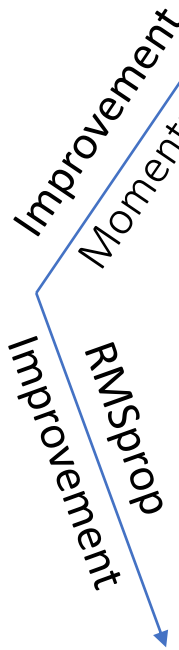
Gradient Descent does not have a control over learning speed and its fluctuations

Implementing exponentially weighted average, we can control the gradient descent

Improvements to Gradient Descent

gradiant decend update

$$w^{[l]} = w^{[l]} - \alpha\, dw^{[l]}$$
$$b^{[l]} = b^{[l]} - \alpha\, db^{[l]}$$



On iteration t:

Compute dw, db on current mini − batch

$$V_{dw} = \beta V_{dw} + (1-\beta)dw$$
$$V_{db} = \beta V_{db} + (1-\beta)db$$
$$w = w - \alpha V_{dw} \text{ and } b = b - \alpha V_{dw}$$

On iteration t:

Compute dw, db on mini − batch

$$s_{dw} = \beta s_{dw} + (1-\beta)dw^2 \text{ (element wise squre)}$$
$$s_{db} = \beta s_{db} + (1-\beta)db^2$$
$$w = w - \alpha \frac{dw}{\sqrt{s_{dw} + \varepsilon}} \text{ and } b = b - \alpha \frac{db}{\sqrt{s_{db} + \varepsilon}} \qquad \varepsilon = 10^{-8}$$

Adam - Adaptive moment estimation

$$V_{dw} = 0, s_{dw} = 0, V_{db} = 0, s_{db} = 0$$

$$on\ iteration\ t:$$

$$compute\ dw, db\ using\ mini - batch$$

$$V_{dw} = \beta_1 V_{dw} + (1 - \beta_1) dw, V_{db} = \beta_1 V_{db} + (1 - \beta_1) db < -'moment'$$

$$s_{dw} = \beta_2 s_{dw} + (1 - \beta_2) dw^2, s_{db} = \beta_2 s_{db} + (1 - \beta_2) db < -\text{RMSprop}$$

$$V_{dw}^{corrected} = \frac{V_{dw}}{(1 - \beta_1^t)}, V_{db}^{corrected} = \frac{V_{db}}{1 - \beta_1^t}$$

$$s_{dw}^{corrected} = \frac{s_{dw}}{1 - \beta_2^t}, s_{db}^{corrected} = \frac{s_{db}}{1 - \beta_2^t}$$

$$w = w - \alpha \frac{V_{dw}^{corrected}}{\sqrt{s_{dw}^{corrected} + \varepsilon}}, b = b - \alpha \frac{V_{db}^{corrected}}{\sqrt{s_{db}^{corrected} + \varepsilon}}$$

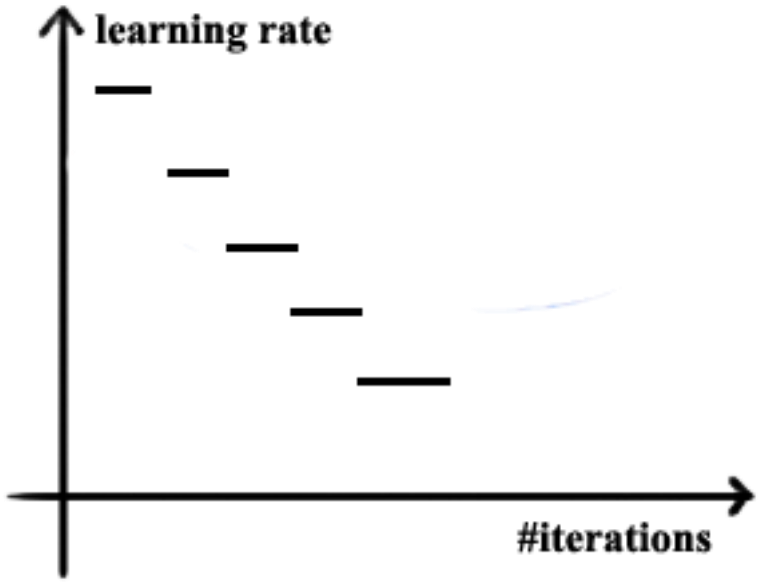
- TensorFlow: learning_rate=0.001, beta1=0.9, beta2=0.999, epsilon=1e-08.
- Keras: lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-08, decay=0.0.
- Caffe: learning_rate=0.001, beta1=0.9, beta2=0.999, epsilon=1e-08
- Torch: learning_rate=0.001, beta1=0.9, beta2=0.999, epsilon=1e-8

Learning rate decay

$$\alpha = \frac{1}{1 + decayrate * epoch} \alpha_0$$

$$\alpha = decayrate^{epoch} * \alpha_0$$

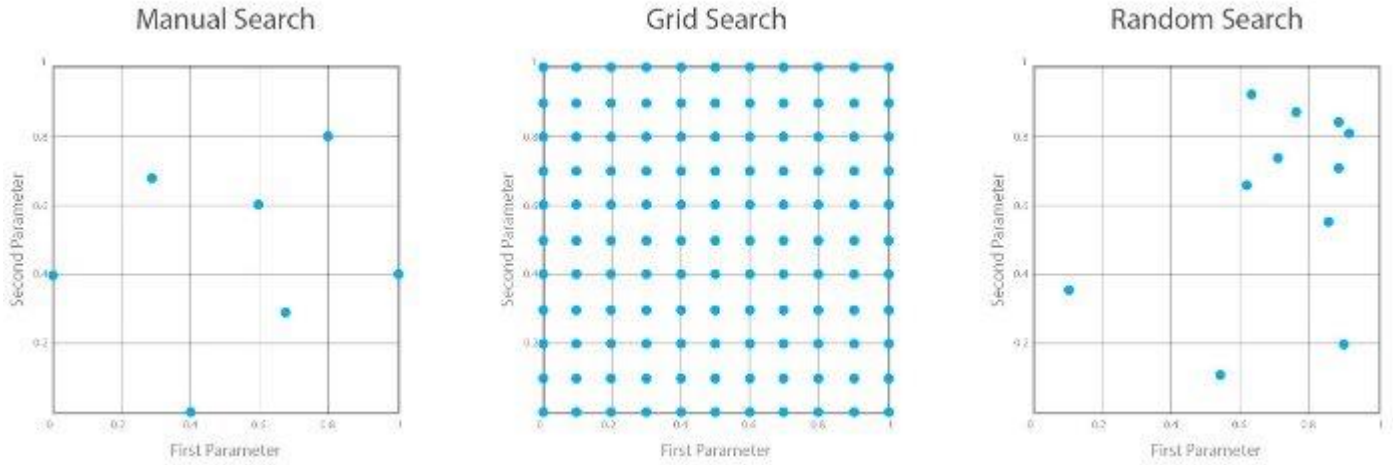
$$\alpha = \frac{k}{\sqrt{epoch}} * \alpha_0 \; or \; \frac{k}{\sqrt{batchsize}} * \alpha_0$$



Manual decay: decrease learning rate manually day by day or hour by hour

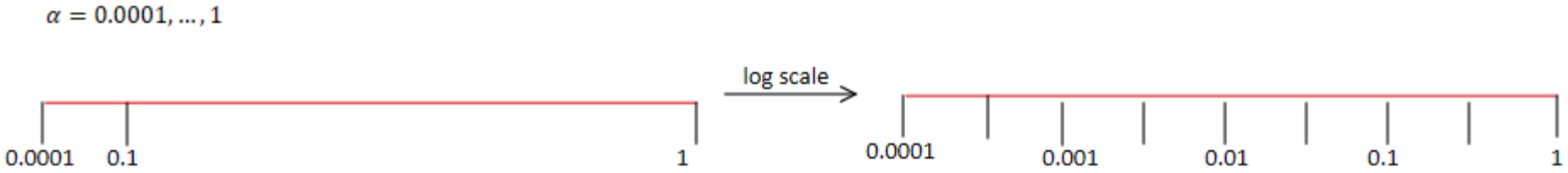
Hyperparameters Tuning

1. Prioritize parameters
   - α
   - Mini-batch size
   - #hidden units
   - #layers
   - Decay rate
   - Etc



2. Use random sampling not a grid search

3. Use an appropriate scale to pick hyper parameters



Reference

- Deeplearning.ai
- Dropout: A Simple Way to Prevent Neural Networks from Overfitting : http://jmlr.org/papers/volume15/srivastava14a.old/srivastava14a.pdf
- Understanding the difficulty of training deep feedforward neural networks : http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf
- Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification : https://arxiv.org/pdf/1502.01852.pdf
- Understanding Xavier Initialization In Deep Neural Networks: https://prateekvjoshi.com/2016/03/29/understanding-xavier-initialization-in-deep-neural-networks/

Thank You