Deep Learning: Introduction

Sanjiv R. Das

Professor of Finance and Data Science

Santa Clara University

London, June 2018

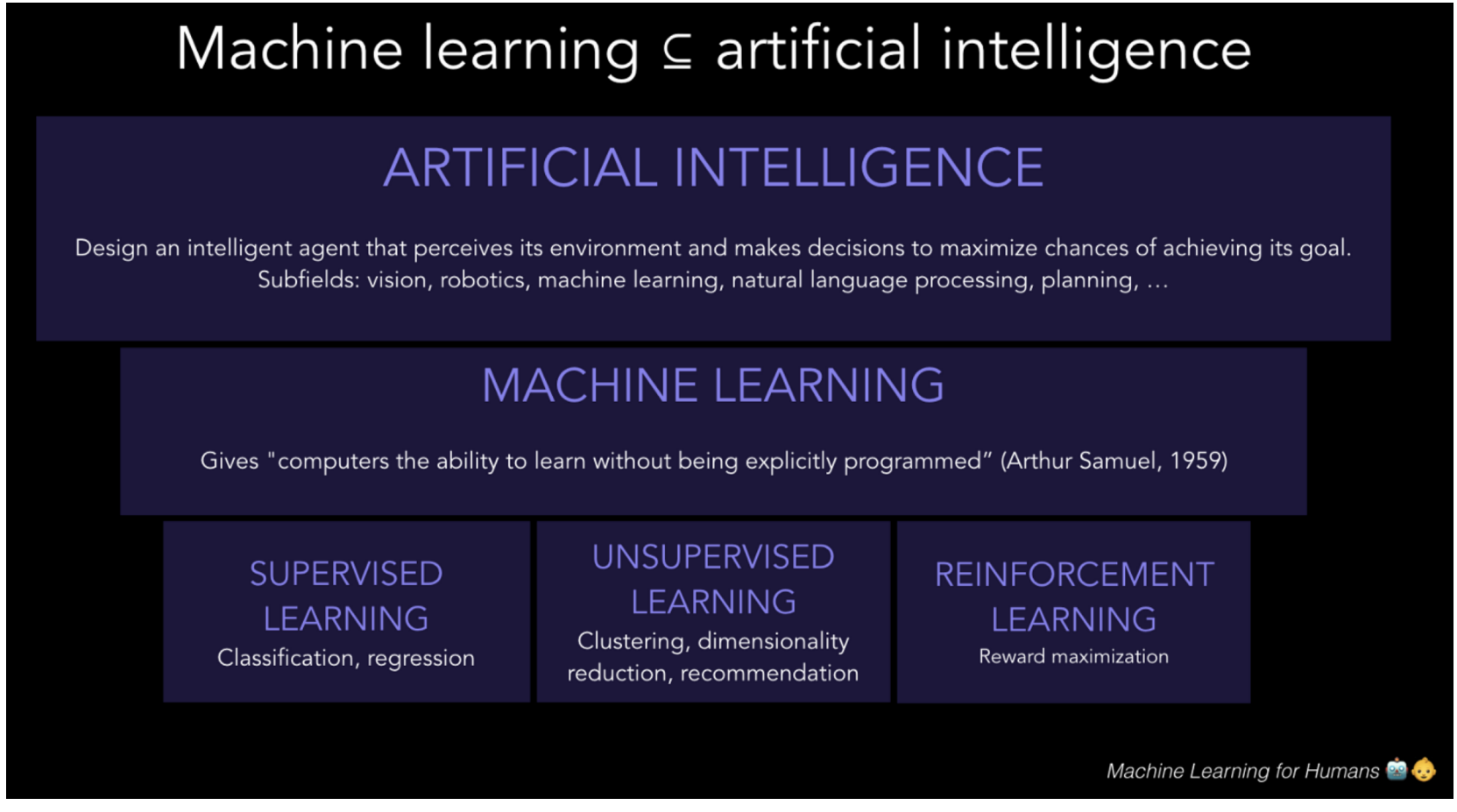
REFERENCE BOOK: http://srdas.github.io/DLBook (http://srdas.github.io/DLBook)

ABSTRACT: This workshop will cover the essentials of deep learning from a mathematical and programming perspective.

- What is deep learning?
- Types of deep learning
- Mathematics of Deep Learning: universal approximation using neural nets, Backpropagation, and techniques for fitting deep learning nets.
- Programming deep learning: tensorflow, mxnet, h2o in R and Python with applications.

In [4]:
```
%pylab inline
import pandas as pd
from IPython.external import mathjax
```

```
Populating the interactive namespace from numpy and matplotlib

/Users/srdas/anaconda3/lib/python3.6/site-packages/IPython/core/magics/pylab.p
y:160: UserWarning: pylab import has clobbered these variables: ['norm']
`%matplotlib` prevents importing * from pylab and numpy
  "\n`%matplotlib` prevents importing * from pylab and numpy"
```



https://medium.com/machine-learning-for-humans/why-machine-learning-matters-6164faf1df12 (https://medium.com/machine-learning-for-humans/why-machine-learning-matters-6164faf1df12)

Deep Learning is Pattern Recognition

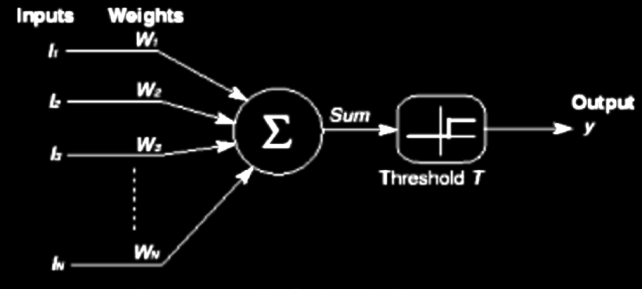


Figure 1.1: McCullough Pitts neuron

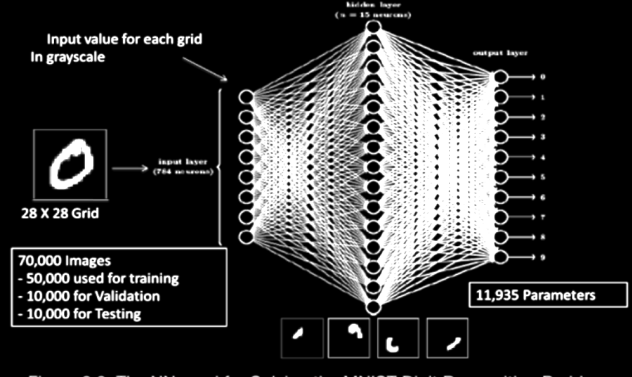


Figure 2.2: The NN used for Solving the MNIST Digit Recognition Problem

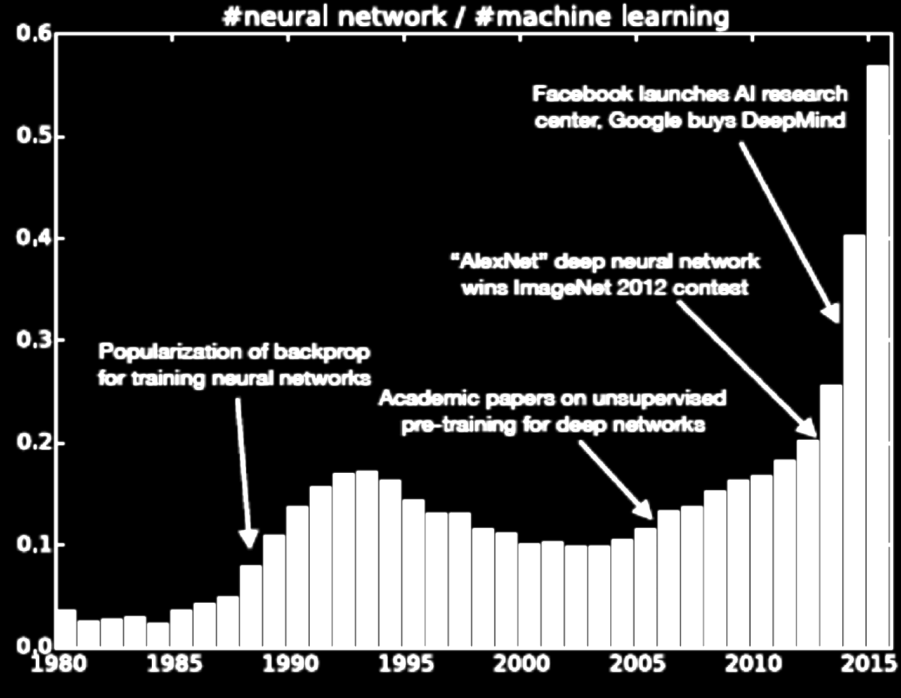
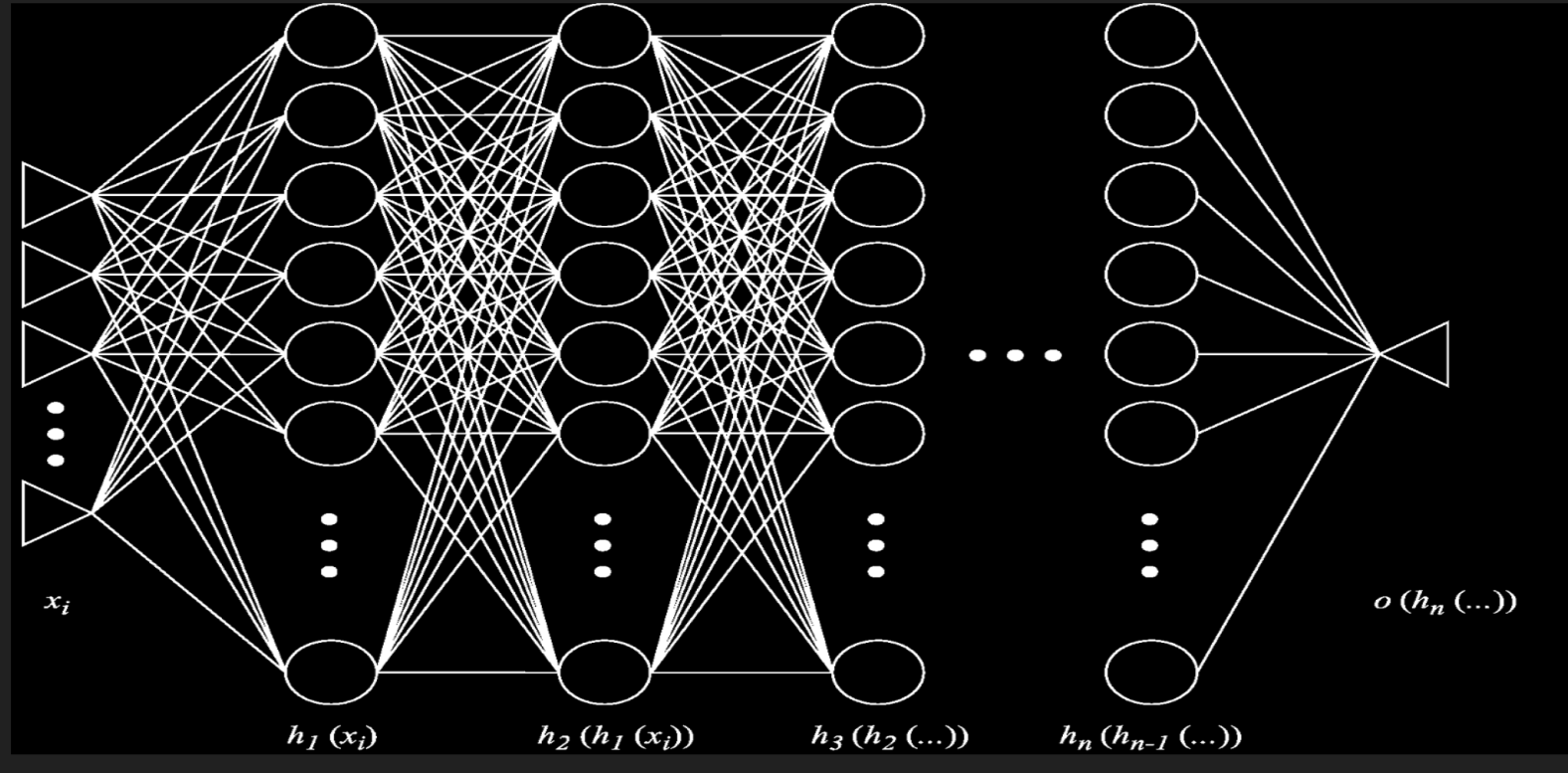
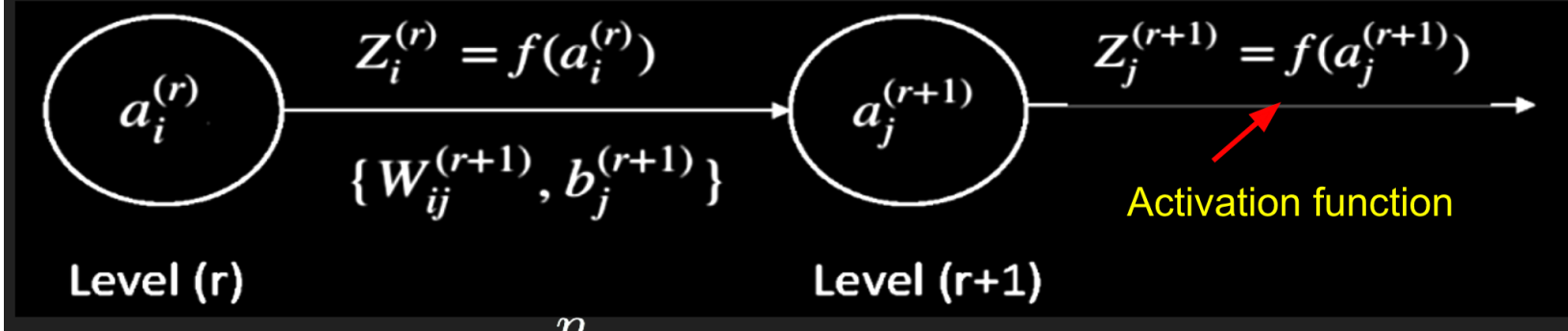


Figure 1.2: Number of NN research reports

The Mathematics of Deep Learning (http://srdas.github.io/DLBook/)



Subset of the Net

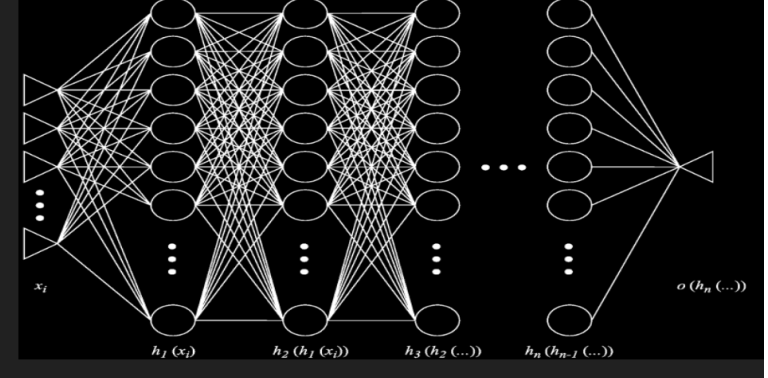


$$a_j = b_j + \sum_{I=1}^{n} w_{ij} z_i$$

Net input

$$\sigma(a_j) = \frac{1}{1 + \exp(-a_j)}$$

Sigmoid function



Activation Functions

https://en.wikipedia.org/wiki/Activation_function

Activation function	Equation	Example	1D Graph
Unit step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Linear	$\phi(z) = z$	Adaline, linear regression	
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multi-layer NN	
Hyperbolic tangent	$\phi(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$	Multi-layer NN	

Net input

$$a_j^{(r)} = \sum_{i=1}^{n^{(r-1)}} W_{ij} Z_i^{(r-1)} + b_j^{(r)}$$

Examples of Different Types of Neurons

Sigmoid

$$Z_j^{(r)} = \frac{1}{1 + \exp(a_j^{(r)})} \in (0, 1)$$

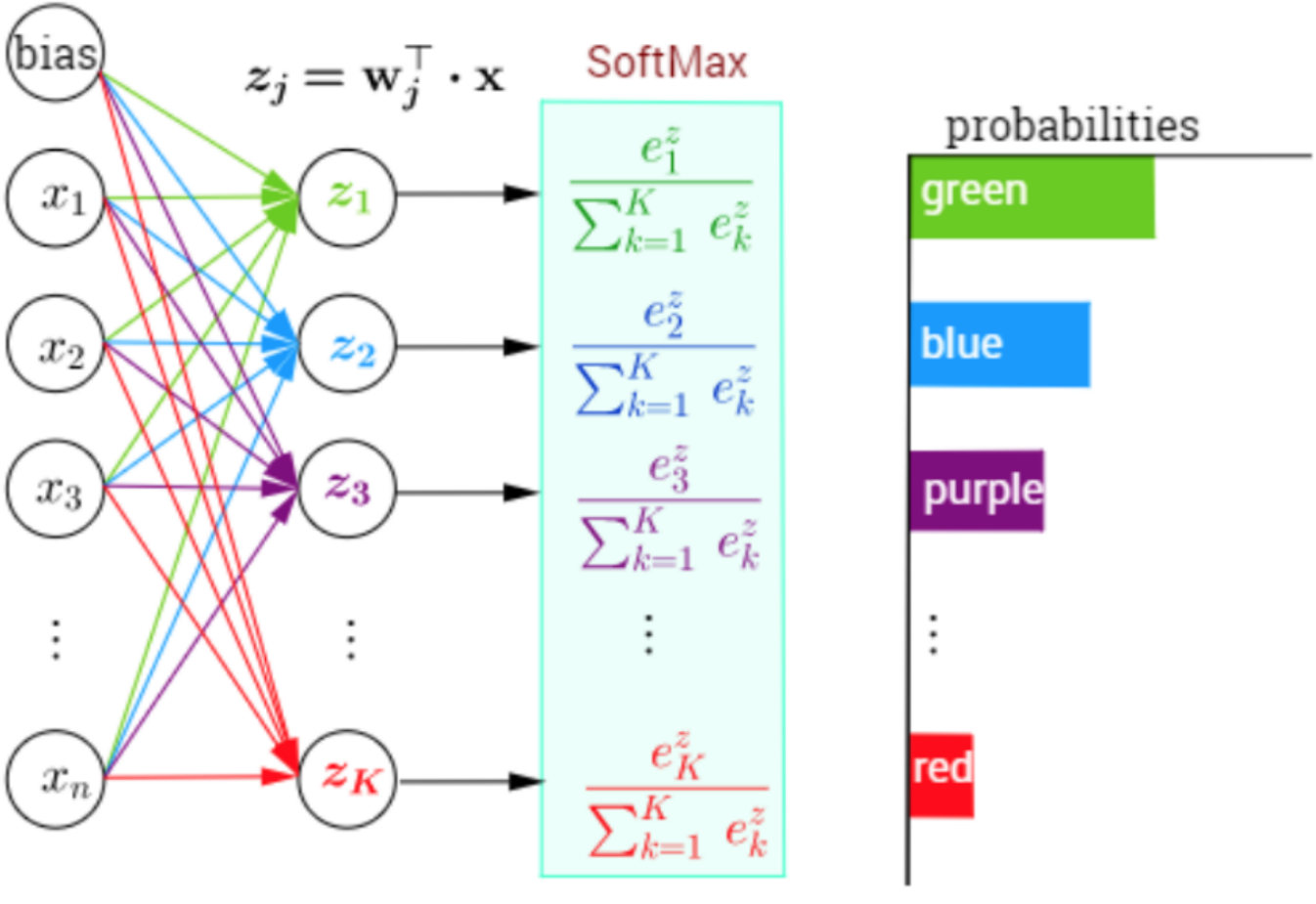
ReLU (restricted linear unit)

$$Z_j^{(r)} = \max[0, a_j^{(r)}] \in (0, 1)$$

TanH (hyperbolic tangent)

$$Z_j^{(r)} = \tanh[a_j^{(r)}] \in (-1, +1)$$

Output Layer



In [2]:
```
#The Softmax function
#Assume 10 output nodes with randomly generated values
z = randn(32)  #inputs from last hidden layer of 32 nodes to the output layer
w = rand(32*10).reshape((10,32))  #weights for the output layer
b = rand(10)   #bias terms at output later
a = w.dot(z) + b  #Net input at output layer
e = exp(a)
softmax_output = e/sum(e)
print(softmax_output.round(3))
print('final tag =',where(softmax_output==softmax_output.max())[0][0])
```

```
[0.034 0.057 0.072 0.18  0.205 0.012 0.276 0.004 0.096 0.063]
final tag = 6
```

Loss Function

Fitting the DLN is an exercise where the best weights $\{W, b\} = \{W_{ij}^{(r+1)}, b_j^{(r+1)}\}, \forall r$ for all layers are determined to minimize a loss function generally denoted as

$$\min_{W,b} \sum_{m=1}^{M} L_m[h(X^{(m)}), T^{(m)}]$$

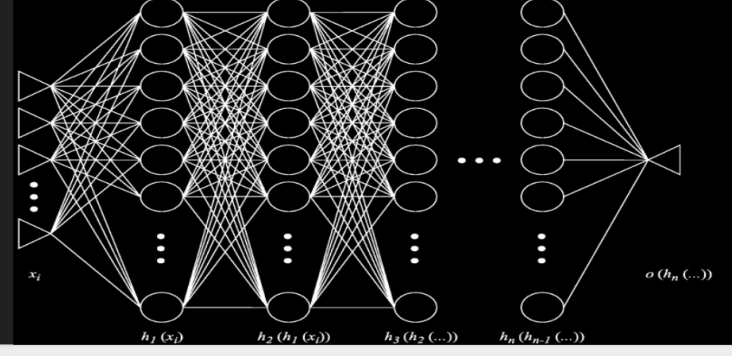
where M is the number of training observations (rows in the data set), $T^{(m)}$ is the true value of the output, and $h(X^{(m)})$ is the model output from the DLN. The loss function L_m quantifies the difference between the model output and the true output.

Cross Entropy

$$C = -\frac{1}{n} \sum_x [y \ln a + (1 - y) \ln(1 - a)]$$

Quadratic Loss

$$C = \frac{(y - a)^2}{2}$$



More on Cross Entropy

https://rdipietro.github.io/friendly-intro-to-cross-entropy-loss/ (https://rdipietro.github.io/friendly-intro-to-cross-entropy-loss/)

Notation from the previous slides:

- y_i: actual probability of the correct class i, i.e., 1 or 0.
- a_i or $\hat{y}_i$: predicted probability of the correct class.
- n: number of observations in the data.

Entropy

$$E = -\frac{1}{n} \sum_{i} [y_i \ln y_i]$$

In [5]:
```
y = [0.33, 0.33, 0.34]
bits = log2(y)
entropy = -sum(y*bits)
print(entropy)
```

```
1.58481870497303
```

In [6]:
```
y = [0.2, 0.3, 0.5]
entropy = -sum(y*log2(y))
print(entropy)
```

```
1.4854752972273344
```

In [7]:
```
y = [0.1, 0.1, 0.8]
entropy = -sum(y*log2(y))
print(entropy)
```

```
0.9219280948873623
```

Cross-entropy:

$$C = -\frac{1}{n} \sum_i [y_i \ln a_i]$$

where $a_i = \hat{y}_i$.

Note that $C > E$, always.

In [9]:
```
#Correct prediction
y = [0, 0, 1]
yhat = [0.1, 0.1, 0.8]
crossentropy = -sum(y*log2(yhat))
print(crossentropy)
```

```
0.3219280948873623
```

In [10]:
```
#Wrong prediction
yhat = [0.1, 0.6, 0.3]
crossentropy = -sum(y*log2(yhat))
print(crossentropy)
```

```
1.7369655941662063
```

Kullback-Leibler Divergence

$$KL = \sum_i y_i \cdot \log\left(\frac{y_i}{\hat{y}_i}\right)$$

Measures the extra bits required if the wrong selection is made.

In [6]:
```
#Correct prediction
y = [0, 0, 1.0]
yhat = [0.1, 0.1, 0.8]
KL = -sum(y[2]*log2(y[2]/yhat[2]))
print(KL)

#Wrong prediction
yhat = [0.1, 0.6, 0.3]
KL = -sum(y[2]*log2(y[2]/yhat[2]))
print(KL)
```

```
-0.32192809488736235
-1.7369655941662063
```

Gradient Descent

Fitting the DLN requires getting the weights $\{W, b\}$ that minimize L_m. These are done using gradient descent, i.e.,

$$W_{ij}^{(r+1)} \leftarrow W_{ij}^{(r+1)} - \eta \cdot \frac{\partial L_m}{\partial W_{ij}^{(r+1)}}$$

$$b_j^{(r+1)} \leftarrow b_j^{(r+1)} - \eta \cdot \frac{\partial L_m}{\partial b_j^{(r+1)}}$$

Here η is the learning rate parameter. We iterate on these functions until the gradients become zero, and the weights discontinue changing with each update, also known as an **epoch**.

Total Gradient

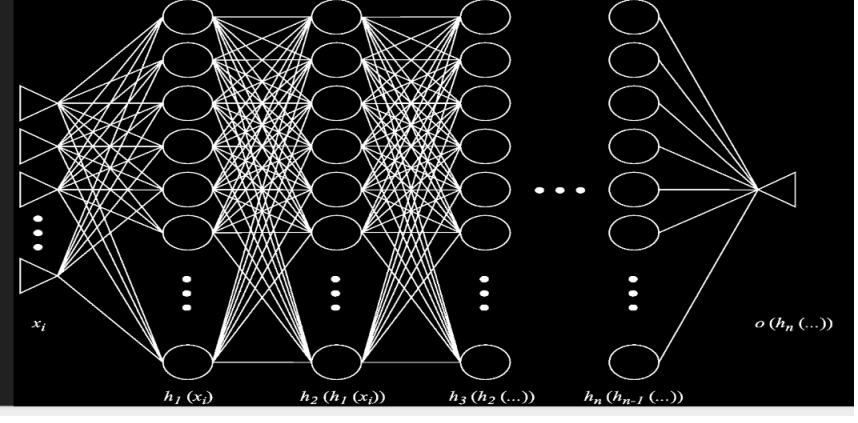
$$w_{ij}^{(r)} \leftarrow w_{ij}^{(r)} - \frac{\eta}{M} \sum_{m=1}^{M} \frac{\partial L_m(w, b; X(m), T(m))}{\partial w_{ij}^{(r)}}$$

$$b_i^{(r)} \leftarrow b_i^{(r)} - \frac{\eta}{M} \sum_{m=1}^{M} \frac{\partial L_m(w, b; X(m), T(m))}{\partial b_i^{(r)}}$$

Stochastic Batch Gradient

$$w_{ij}^{(r)} \leftarrow w_{ij}^{(r)} - \frac{\eta}{s} \sum_{m=1}^{s} \frac{\partial L_m(w, b; X(m), T(m))}{\partial w_{ij}^{(r)}}$$

$$b_i^{(r)} \leftarrow b_i^{(r)} - \frac{\eta}{s} \sum_{m=1}^{s} \frac{\partial L_m(w, b; X(m), T(m))}{\partial b_i^{(r)}}$$



Gradients and the Chain Rule

To solve this minimization problem, we need gradients for all W, b. These are denoted :

$$a_j^{(r+1)} = \sum_{i=1}^{n_r} W_{ij}^{(r+1)} Z_i^{(r)} + b_j^{(r+1)}$$

$$\frac{\partial L_m}{\partial W_{ij}^{(r+1)}}, \quad \frac{\partial L_m}{\partial b_j^{(r+1)}}, \quad \forall r+1, j$$

We write out these gradients using the chain rule:

$$\frac{\partial L_m}{\partial W_{ij}^{(r+1)}} = \frac{\partial L_m}{\partial a_j^{(r+1)}} \cdot \frac{\partial a_j^{(r+1)}}{\partial W_{ij}^{(r+1)}} = \delta_j^{(r+1)} \cdot Z_i^{(r)}$$

where we have written

$$\delta_j^{(r+1)} = \frac{\partial L_m}{\partial a_j^{(r+1)}}$$

Likewise, we have

$$\frac{\partial L_m}{\partial b_j^{(r+1)}} = \frac{\partial L_m}{\partial a_j^{(r+1)}} \cdot \frac{\partial a_j^{(r+1)}}{\partial b_j^{(r+1)}} = \delta_j^{(r+1)} \cdot 1 = \delta_j^{(r+1)}$$

Delta Values

we need the following intermediate calculation

$$a_j^{(r+1)} = \sum_{i=1}^{n_r} W_{ij}^{(r+1)} Z_i^{(r)} + b_j^{(r+1)}$$

$$= \sum_{i=1}^{n_r} W_{ij}^{(r+1)} f(a_i^{(r)}) + b_j^{(r+1)}$$

$$\frac{\partial a_j^{(r+1)}}{\partial a_i^{(r)}} = W_{ij}^{(r+1)} \cdot f'(a_i^{(r)})$$

$$\delta_i^{(r)} = \frac{\partial L_m}{\partial a_i^{(r)}}$$

$$= \sum_{j=1}^{n_{r+1}} \frac{\partial L_m}{\partial a_j^{(r+1)}} \cdot \frac{\partial a_j^{(r+1)}}{\partial a_i^{(r)}}$$

$$= \sum_{j=1}^{n_{r+1}} \delta_j^{(r+1)} \cdot W_{ij}^{(r+1)} \cdot f'(a_i^{(r)})$$

$$= f'(a_i^{(r)}) \cdot \sum_{j=1}^{n_{r+1}} \delta_j^{(r+1)} \cdot W_{ij}^{(r+1)}$$

Output Layer

$$a_j^{(R+1)} = \sum_{i=1}^{n_R} W_{ij}^{(R+1)} Z_j^{(R)} + b_j^{(R+1)}$$

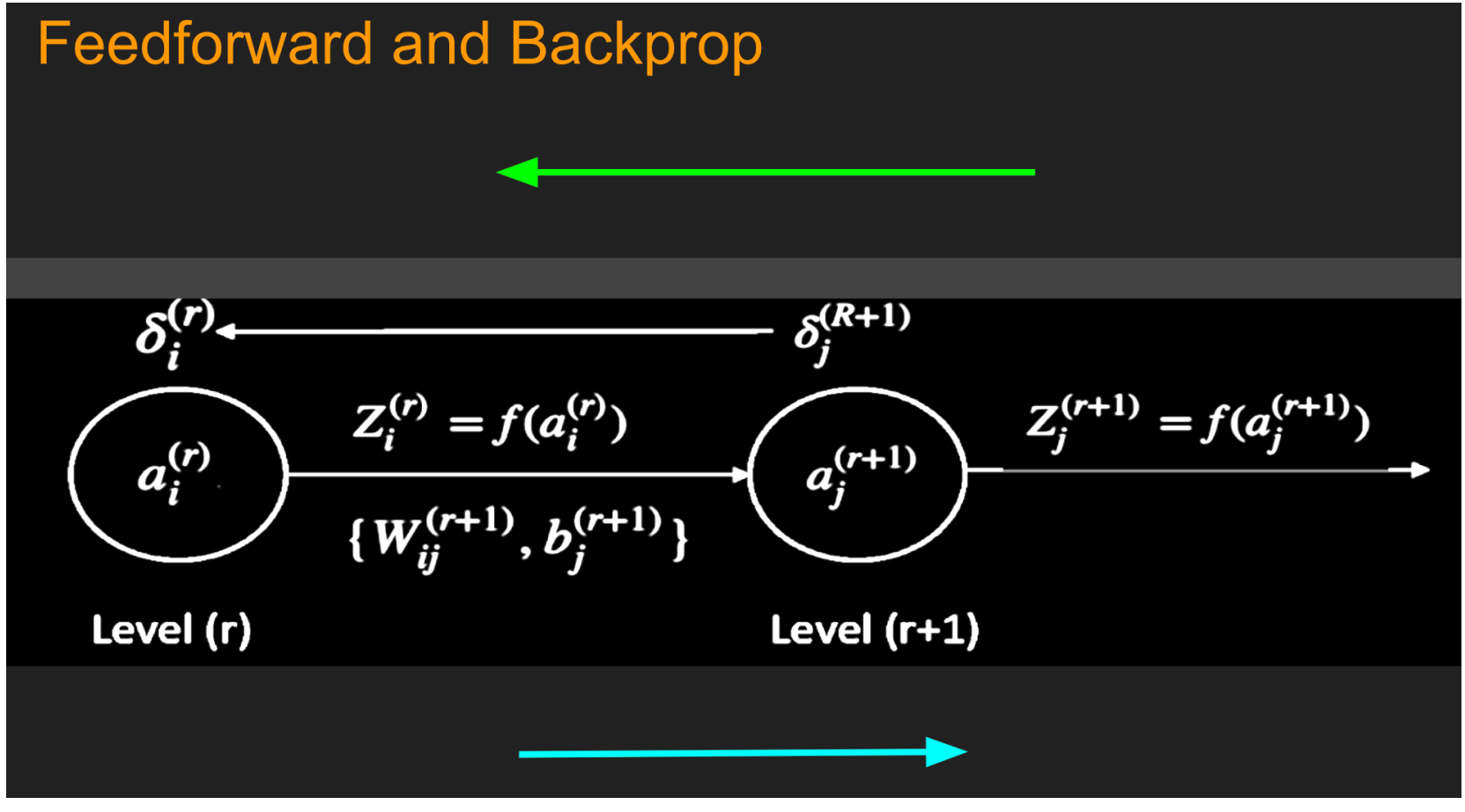
$$\text{Final output} = h(a_j^{(R+1)})$$

$$\delta_j^{(R+1)} = \frac{\partial L_m}{\partial a_j^{(R+1)}} = h'(a_j^{(R+1)})$$

Iterative Process

1. Start with an initial set of weights $\{w, b\}$.
2. Feedforward the initial data and weights into the DLN, and find the $\{a_i^{(r)}, Z_i^{(r)}\}, \forall r, i$.
3. Then, using backpropagation, compute all $\delta_i^{(r)}, \forall r, i$.
4. Use these $\delta_i^{(r)}$ values to get all the new gradients.
5. Apply gradient descent to get new weights.
6. Keep iterating steps 2-5, until the chosen number of epochs is completed.

Feedforward and Backprop



Recap

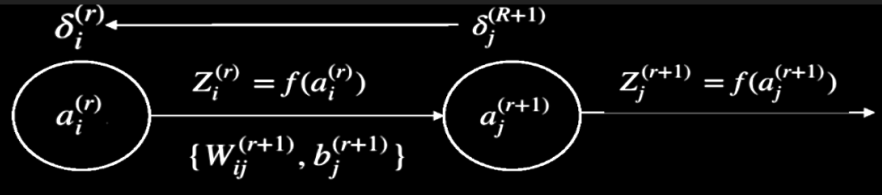
$$\frac{\partial L_m}{\partial W_{ij}^{(r+1)}} = \frac{\partial L_m}{\partial a_j^{(r+1)}} \cdot \frac{\partial a_j^{(r+1)}}{\partial W_{ij}^{(r+1)}} = \delta_j^{(r+1)} \cdot Z_i^{(r)}$$

$$\delta_j^{(r+1)} = \frac{\partial L_m}{\partial a_j^{(r+1)}}$$

$$= f'(a_i^{(r)}) \cdot \sum_{j=1}^{n_{r+1}} \delta_j^{(r+1)} \cdot W_{ij}^{(r+1)}$$

$$\frac{\partial L_m}{\partial b_j^{(r+1)}} = \frac{\partial L_m}{\partial a_j^{(r+1)}} \cdot \frac{\partial a_j^{(r+1)}}{\partial b_j^{(r+1)}} = \delta_j^{(r+1)} \cdot 1 = \delta_j^{(r+1)}$$

The Magic of Backpropagation



$$\min_{W,b} \sum_{m=1}^{M} L_m[h(X^{(m)}), T^{(m)}]$$

Loss Function

$$a_j^{(r+1)} = \sum_{i=1}^{n_r} W_{ij}^{(r+1)} Z_i^r + b_j^{(r+1)} = \sum_{i=1}^{n_r} W_{ij}^{(r+1)} f(a_i^{(r)}) + b_j^{(r+1)} \Rightarrow \frac{\partial a_j^{(r+1)}}{\partial a_i^{(r)}} = W_{ij}^{(r+1)} \cdot f'(a_i^{(r)})$$

Gradients

$$\frac{\partial L_m}{\partial W_{ij}^{(r+1)}} = \frac{\partial L_m}{\partial a_j^{(r+1)}} \cdot \frac{\partial a_j^{(r+1)}}{\partial W_{ij}^{(r+1)}} = \delta_j^{(r+1)} \cdot Z_i^{(r)}$$

$$\frac{\partial L_m}{\partial b_j^{(r+1)}} = \frac{\partial L_m}{\partial a_j^{(r+1)}} \cdot \frac{\partial a_j^{(r+1)}}{\partial b_j^{(r+1)}} = \delta_j^{(r+1)} \cdot 1 = \delta_j^{(r+1)}$$

Gradient Descent

$$W_{ij}^{(r+1)} \leftarrow W_{ij}^{(r+1)} - \eta \cdot \frac{\partial L_m}{\partial W_{ij}^{(r+1)}}$$

$$b_j^{(r+1)} \leftarrow b_j^{(r+1)} - \eta \cdot \frac{\partial L_m}{\partial b_j^{(r+1)}}$$

$$\begin{aligned}
\delta_i^{(r)} &= \frac{\partial L_m}{\partial a_i^{(r)}} \\
&= \sum_{j=1}^{n_{r+1}} \frac{\partial L_m}{\partial a_j^{(r+1)}} \cdot \frac{\partial a_j^{(r+1)}}{\partial a_i^{(r)}} \\
&= \sum_{j=1}^{n_{r+1}} \delta_j^{(r+1)} \cdot W_{ij}^{(r+1)} \cdot f'(a_i^{(r)}) \\
&= f'(a_i^{(r)}) \cdot \sum_{j=1}^{n_{r+1}} \delta_j^{(r+1)} \cdot W_{ij}^{(r+1)}
\end{aligned}$$

Deltas

$$\delta_j^{(R+1)} = \frac{\partial L_m}{\partial a_j^{(R+1)}} = h'(a_j^{(R+1)})$$

SoftMax Function

$$L = -\sum_j T_j \log(h_j)$$

$$h_j = \frac{e^{a_j}}{\sum_i e^{a_i}} = \frac{e^{a_j}}{D}$$

$$a_j = \sum_i w_{ij} z_i + b_j$$

Delta of Softmax

$$\frac{\partial L}{\partial a_j} = -\sum_i T_i \frac{\partial \log h_i}{\partial a_j}$$

$$\log h_j = a_j - \log D, \qquad \frac{\partial D}{\partial a_j} = \sum_i e^{z_i} \delta_{ij} = e^{z_j}$$

$$\frac{\partial \log h_j}{\partial a_j} = \delta_{ji} - \frac{1}{D} \frac{\partial D}{\partial a_j} = \delta_{ji} - h_j$$

Noting that

$$\frac{\partial \log h_j}{\partial a_j} = \frac{1}{h_j} \frac{\partial h_j}{\partial a_j}$$

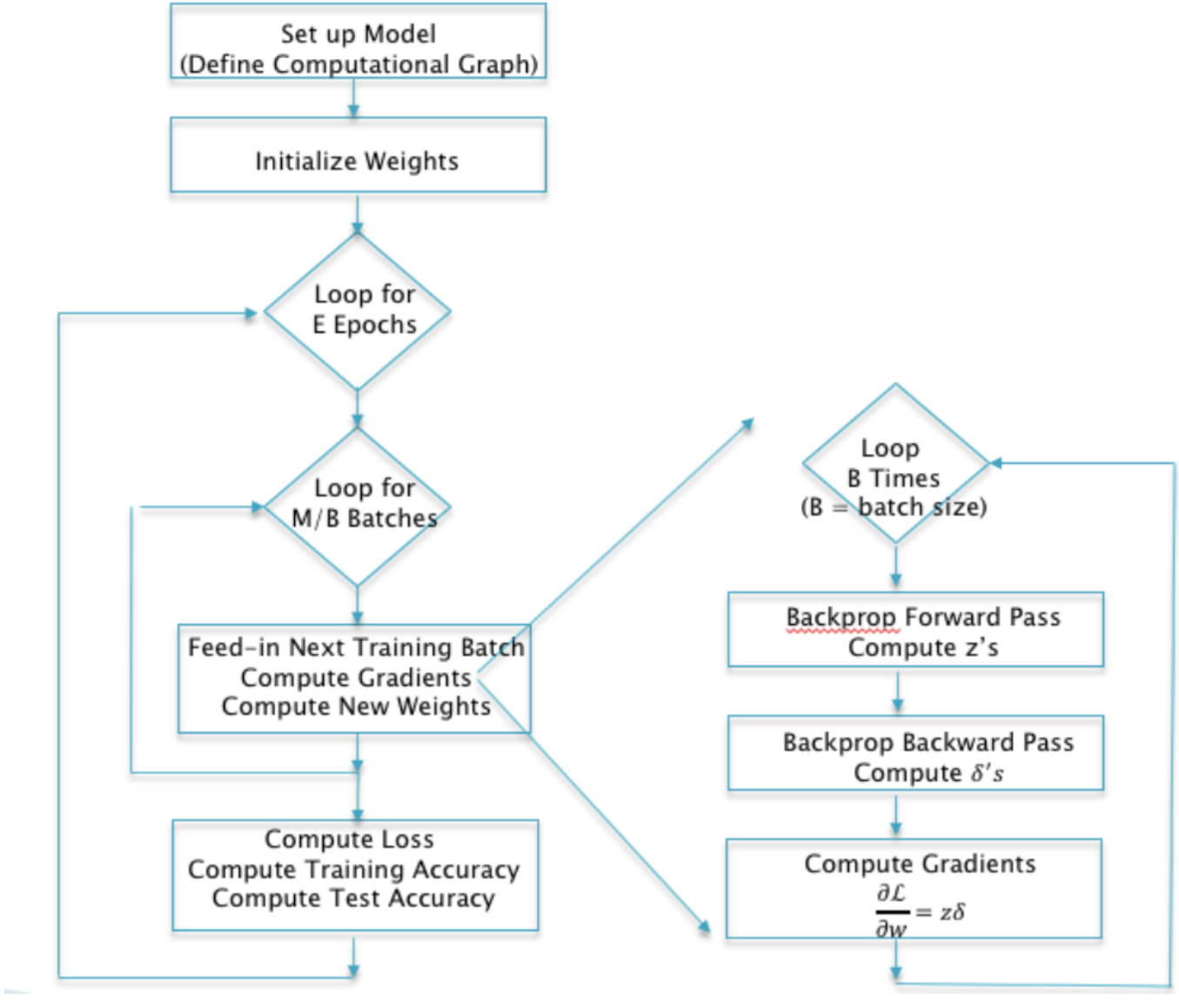
we get

$$\frac{\partial h_j}{\partial a_j} = h_j(\delta_{ji} - h_i)$$

Finally,

$$\frac{\partial L}{\partial a_j} = \sum_i T_i(h_j - \delta_{ij}) = h_j(\sum_j T_j) - T_j = h_j - T_j$$

Batch Stochastic Gradient



Fitting the NN

1. Initialize all the weight and bias parameters $(w_{ij}^{(r)}, b_i^{(r)})$ (this is a critical step).

2. For $q = 0, \ldots, \frac{M}{B} - 1$ repeat the following steps (2a) - (2f):

a. For the training inputs $X_i(m)$, $qB \leq m \leq (q+1)B$, compute the model predictions $y(m)$ given by

$$a_i^{(r)}(m) = \sum_{j=1}^{P^{r-1}} w_{ij}^{(r)} z_j^{(r-1)}(m) + b_i^{(r)} \quad \text{and} \quad z_i^{(r)}(m) = f(a_i^{(r)}(m)), \quad 2 \leq r \leq R, 1 \leq i \leq$$

and for $r = 1$,

$$a_i^{(1)}(m) = \sum_{j=1}^{P^1} w_{ij}^{(1)} x_j(m) + b_i^{(1)} \quad \text{and} \quad z_i^{(1)}(m) = f(a_i^{(1)}(m)), \quad 1 \leq i \leq N$$

The logits and classification probabilities are computed using

$$a_i^{(R+1)}(m) = \sum_{j=1}^{K} w_{ij}^{(R+1)} z_j^{(R)}(m) + b_i^{(R+1)}$$

and

$$y_i(m) = \frac{\exp(a_i^{(R+1)}(m))}{\sum_{k=1}^{K} \exp(a_k^{(R+1)}(m))}, \qquad 1 \leq i \leq K$$

This step constitutes the *forward pass* of the algorithm.

b. Evaluate the gradients $\delta_k^{(R+1)}(m)$ for the logit layer nodes using

$$\delta_k^{(R+1)}(m) = y_k(m) - t_k(m), \quad 1 \leq k \leq K$$

This step and the following one constitute the start of the backward pass of the algorithm, in which we compute the gradients $\delta_k^{(r)}, 1 \leq k \leq K, 1 \leq r \leq R$ for all the hidden nodes.

c. Back-propagate the δs using the following equation to obtain the $\delta_j^{(r)}(m), 1 \leq r \leq R, 1 \leq j \leq P^r$ for each hidden node in the network.

$$\delta_j^{(r)}(m) = f'(a_j^{(r)}(m)) \sum_k w_{kj}^{(r+1)} \delta_k^{(r+1)}(m), \quad 1 \leq r \leq R$$

d. Compute the gradients of the Cross Entropy Function $\mathcal{L}(m)$ for the m-th training vector $(X(m), T(m))$ with respect to all the weight and bias parameters using the following equation.

$$\frac{\partial \mathcal{L}(m)}{\partial w_{ij}^{(r+1)}} = \delta_i^{(r+1)}(m) z_j^{(r)}(m)$$

and

$$\frac{\partial \mathcal{L}(m)}{\partial b_i^{(r+1)}} = \delta_i^{(r+1)}(m), \quad 0 \leq r \leq R$$

e. Change the model weights according to the formula

$$w_{ij}^{(r)} \leftarrow w_{ij}^{(r)} - \frac{\eta}{B} \sum_{m=qB}^{(q+1)B} \frac{\partial \mathcal{L}(m)}{\partial w_{ij}^{(r)}},$$

$$b_{i}^{(r)} \leftarrow b_{i}^{(r)} - \frac{\eta}{B} \sum_{m=qB}^{(q+1)B} \frac{\partial \mathcal{L}(m)}{\partial b_{i}^{(r)}},$$

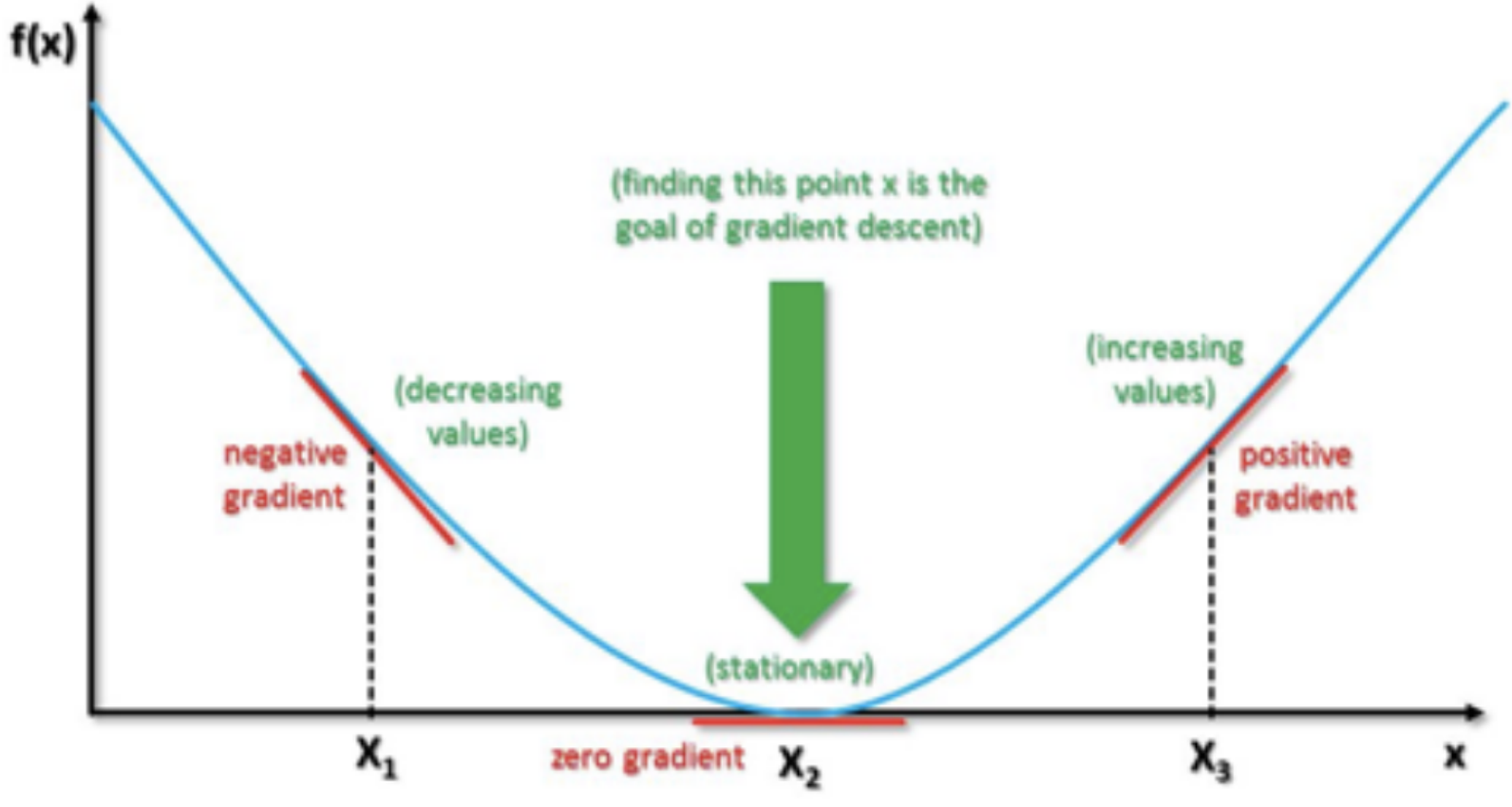
f. Increment $q \leftarrow ((q+1) \mod B)$ and go back to step (a).

Compute the Loss Function L over the Validation Dataset given by

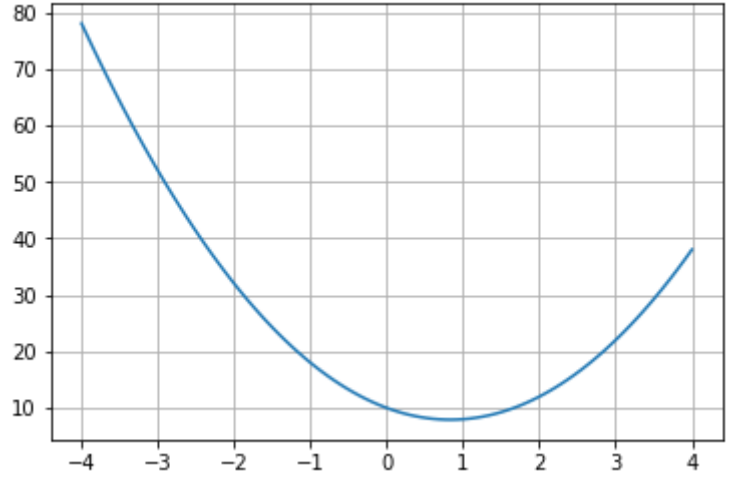
$$L = -\frac{1}{V} \sum_{m=1}^{V} \sum_{k=1}^{K} t_k(m) \log y_k(m)$$

If L has dropped below some threshold, then stop. Otherwise go back to Step 2.

Gradient Descent Example



In [30]:
```
def f(x):
    return 3*x**2 -5*x + 10

x = linspace(-4,4,100)
plot(x,f(x))
grid()
```

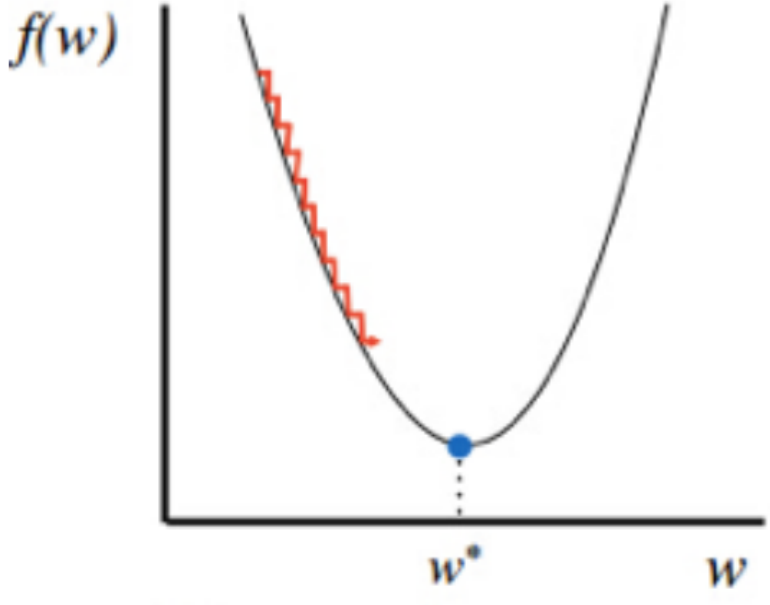


In [35]:
```
dx = 0.001
eta = 0.05  #learning rate
x = -3
for j in range(20):
    df_dx = (f(x+dx)-f(x))/dx
    x = x - eta*df_dx
    print(x,f(x))
```

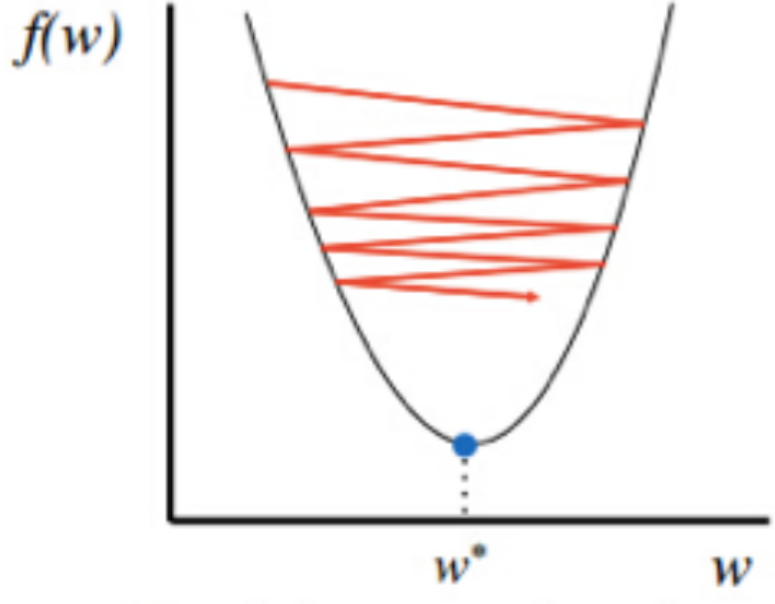
```
-1.8501500000001698 29.519915067502733
-1.0452550000002532 18.503949045077853
-0.4818285000003115 13.105618610239208
-0.08742995000019249 10.460081738472072
0.18864903499989083 9.163520200219716
0.3819043244999847 8.528031116715445
0.5171830271499616 8.216519714966186
0.6118781190049631 8.06379390252634
0.6781646833034642 7.988898596522943
0.7245652783124417 7.95215813604575
0.7570456948186948 7.934126078037087
0.7797819863730542 7.925269906950447
0.7956973904611502 7.920916059254301
0.8068381733227685 7.918772647178622
0.8146367213259147 7.917715356568335
0.8200957049281836 7.917192371084045
0.8239169934497053 7.916932669037079
0.8265918954147882 7.9168030076222955
0.8284643267903373 7.916737788340814
0.8297750287532573 7.916704651261121
```

Vanishing Gradients

- In large problems, gradients simply vanish too soon before training has reached an acceptable level of accuracy.
- There are several issues with gradient descent that need handling, and to solve this, there are several fixes that may be applied.
- Learning rate η may be too large or too small.

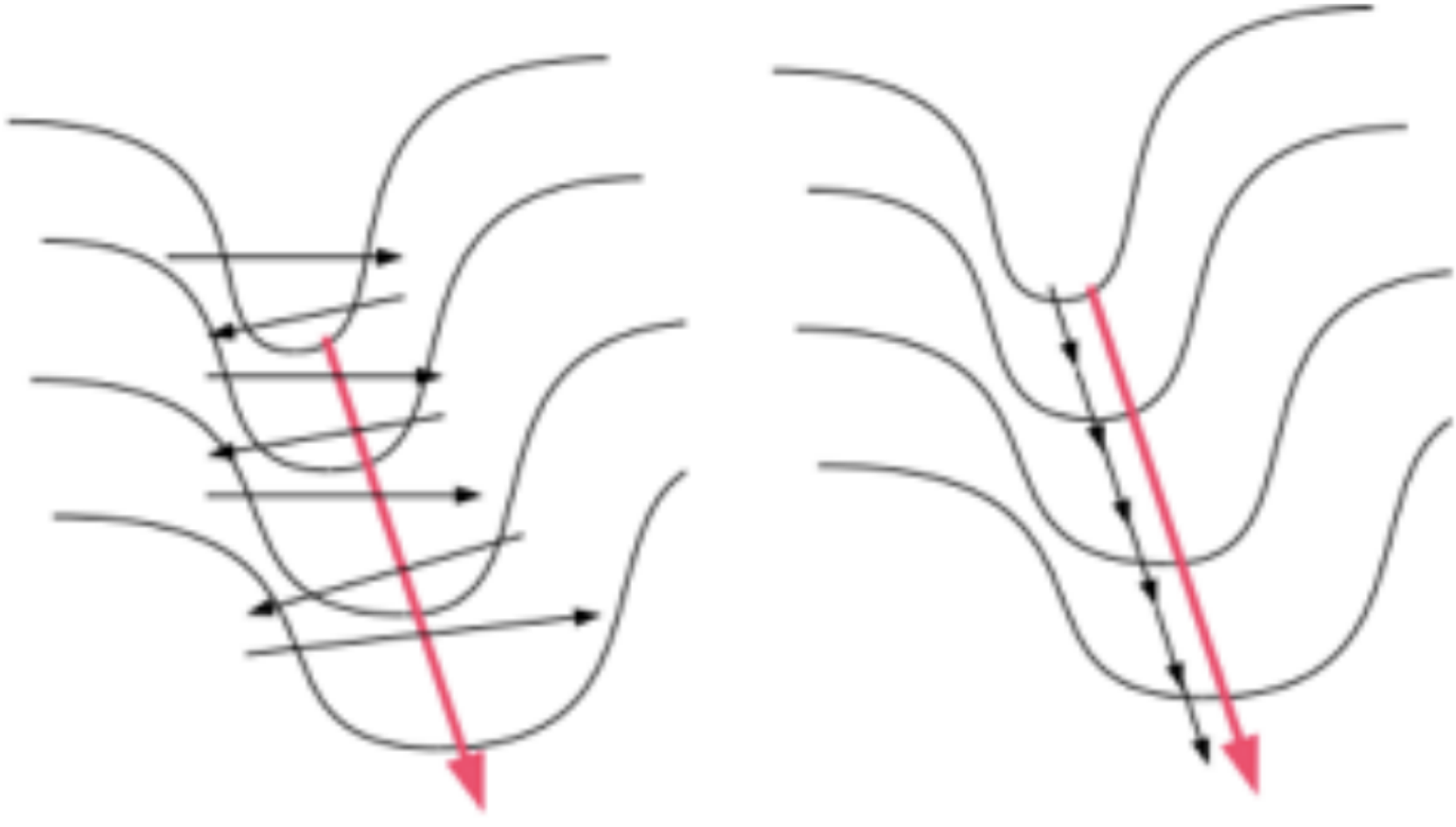


Too small: converge very slowly



Too big: overshoot and even diverge

Gradients in Multiple Dimensions



Saddle Points

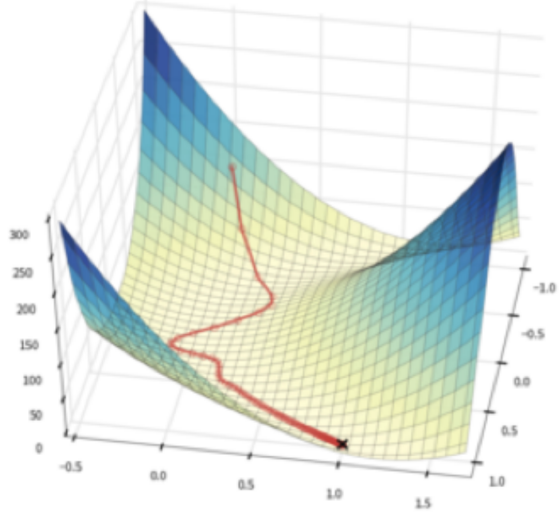
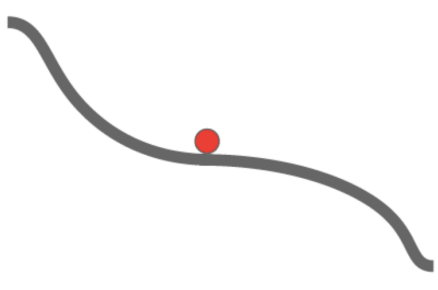
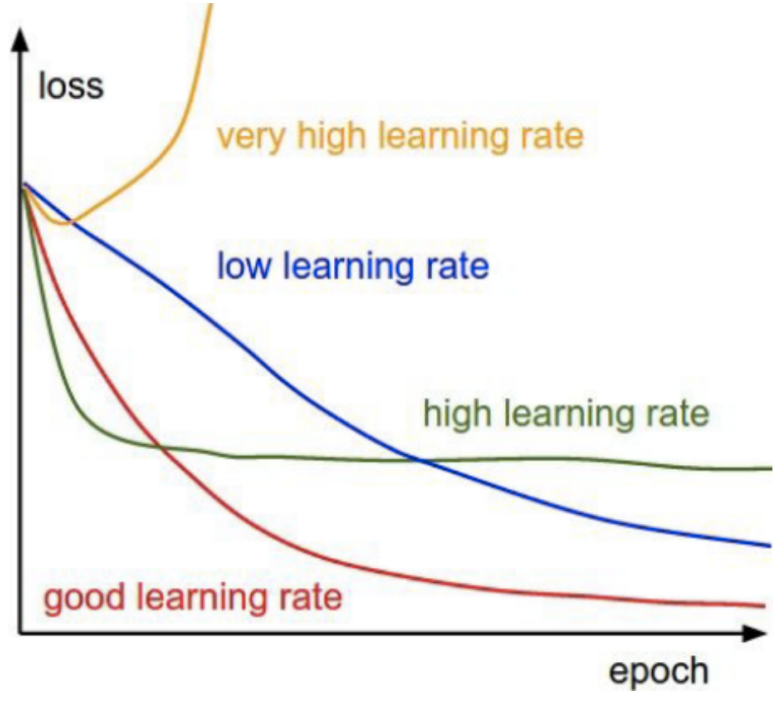


Figure 7.4: Saddle Points



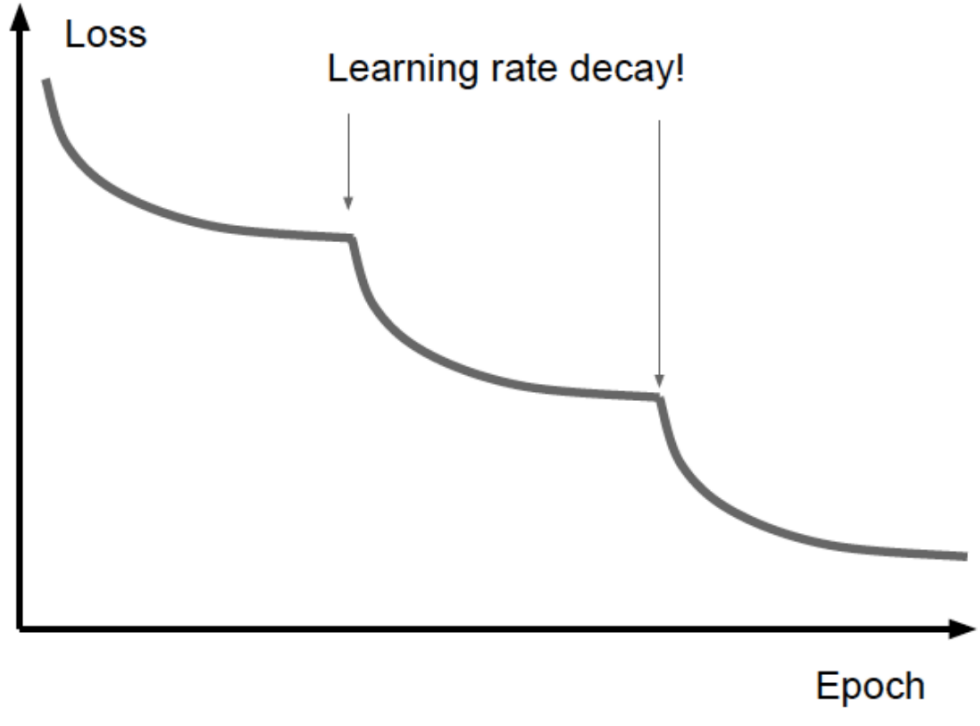
Effect of the Learning Rate

The idea is to start with a high learning rate and then adaptively reduce it as we get closer to the minimum of the loss function.



Annealing

Reduce the learning rate as a step function when the reduction in the loss function begins to plateau.



Learning Rate Algorithms

- Improve the speed of convergence (for example the Momentum, Nesterov Momentum, and Adam algorithms).
- Adapt the effective Learning Rate as the training progresses (for example the ADAGRAD, RMSPROP and Adam algorithms).

Momentum

Fixes the problem where the gradient is multidimensional and has a fast gradient on some axes and a slow one on the others.

At the end of the n^{th} iteration of the Backprop algorithm, define a sequence $v(n)$ by

$$v(n) = \rho\ v(n-1) - \eta\ g(n), \qquad v(0) = -\eta g(0)$$

where ρ is new hyper-parameter called the "momentum" parameter, and $g(n)$ is the gradient evaluated at parameters value $w(n)$.

$g(n)$ is defined by

$$g(n) = \frac{\partial \mathcal{L}(n)}{\partial w}$$

for Stochastic Gradient Descent and

$$g(n) = \frac{1}{B} \sum_{m=nB}^{(n+1)B} \frac{\partial \mathcal{L}(m)}{\partial w}$$

for Batch Stochastic Gradient Descent (note that in this case n is an index into the batch number).

The change in parameter values on each iteration is now defined as

$$w(n+1) = w(n) + v(n)$$

It can be shown from these equations that $v(n)$ can be written as

$$v(n) = -\eta \sum_{i=0}^{n-1} \rho^{n-i} g(i)$$

so that

$$w(n+1) = w(n) - \eta \sum_{i=0}^{n-1} \rho^{n-i} g(i)$$

Behavior of Momentum Parameter

When the momentum parameter $\rho = 0$, then this equation reduces to the usual Stochastic Gradient Descent iteration. On the other hand, when $\rho > 0$, then we get some interesting behaviors:

- If the gradients $g(i)$ are such that they change sign frequently (as in the steep side of the loss surface), then the stepsize $\sum_{i=0}^{n} \rho^{n-i} g(i)$ will be small. Thus the change in these parameters with the number of iterations will limited.

- If the gradients $g(i)$ are such that they maintain their sign (as in the shallow portion of the loss surface), then the stepsize $\sum_{i=0}^{n} \rho^{n-i} g(i)$ will be large. This means that if the gradients maintain their sign then the corresponding parameters will take bigger and bigger steps as the algorithm progresses, even though the individual gradients may be small.

Properties of the Momentum algorithm

- The Momentum algorithm thus accelerates parameter convergence for parameters whose gradients consistently point in the same direction, and slows parameter change for parameters whose gradient changes sign frequently, thus resulting in faster convergence.
- The variable $v(n)$ is analogous to velocity in a dynamical system, while the parameter $(1 - \rho)$ plays the role of the co-efficient of friction.
- The value of ρ determines the degree of momentum, with the momentum becoming stronger as ρ approaches 1.

Note that

$$\sum_{i=0}^{n} \rho^{n-i} g(i) \leq \frac{g_{max}}{1-\rho}$$

ρ is usually set to the neighborhood of 0.9 and from the above equation it follows that $\sum_{i=0}^{n} \rho^{n-i} g(i) \approx 10g$ assuming all the $g(i)$ are approximately equal to g. Hence the effective gradient is ten times the value of the actual gradient. This results in an "overshoot" where the value of the parameter shoots past the minimum point to the other side of the bowl, and then reverses itself. This is a desirable behavior since it prevents the algorithm from getting stuck at a saddle point or a local minima, since the momentum carries it out from these areas.

Nesterov Momentum

1. Recall that the Momentum parameter update equations can be written as:

$$v(n) = \rho\ v(n-1) - \eta\ g(w(n))$$
$$w(n+1) = w(n) + v(n)$$

2. These equations can be improved by evaluation of the gradient at parameter value $w(n+1)$ instead.

Circular? in order to compute $w(n+1)$ we first need to compute $g(w(n))$.

$$w(n+1) \approx w(n) + \rho v(n-1)$$

1. Gives the velocity update equation for Nesterov Momentum
   $$v(n) = \rho\ v(n-1) - \eta\ g(w(n) + \rho v(n-1))$$
   where $g(w(n) + \rho v(n-1))$ denotes the gradient computed at parameter values $w(n) + \rho v(n-1)$.
2. Gradient Descent process speeds up considerably when compared to the plain Momentum method.

The ADAGRAD Algorithm

1. Parameter update rule:

$$w(n+1) = w(n) - \frac{\eta}{\sqrt{\sum_{i=1}^{n} g(n)^2 + \epsilon}}\, g(n)$$

2. Constant ϵ has been added to better condition the denominator and is usually set to a small number such 10^{-7}.

1. Each parameter gets its own adaptive Learning Rate, such that large gradients have smaller learning rates and small gradients have larger learning rates (η is usually defaulted to 0.01). As a result the progress along each dimension evens out over time, which helps the training process.

2. The change in rates happens automatically as part of the parameter update equation.

3. Downside: accumulation of gradients in the denominator leads to the continuous decrease in Learning Rates which can lead to a halt of training in large networks that require a greater number of iterations.

The RMSPROP Algorithm

1. Accumulates the sum of gradients using a sliding window:
$$E[g^2]_n = \rho E[g^2]_{n-1} + (1-\rho)g(n)^2$$
where ρ is a decay constant (usually set to 0.9). This operation (called a Low Pass Filter) has a windowing effect, since it forgets gradients that are far back in time.
2. The quantity $RMS[g]_n$ defined by
$$RMS[g]_n = \sqrt{E[g^2]_n + \epsilon}$$
$$w(n+1) = w(n) - \frac{\eta}{RMS[g]_n}\, g(n)$$

Note that

$$E[g^2]_n = (1-\rho)\sum_{i=0}^{n}\rho^{n-i}g(i)^2 \leq \frac{g_{max}}{1-\rho}$$

which shows that the parameter ρ prevents the sum from blowing up, and a large value of ρ is equivalent to using a larger window of previous gradients in computing the sum.

The ADAM Algorithm

1. The Adaptive Moment Estimation (Adam) algorithm combines the best of algorithms such as Momentum that speed up the training process, with algorithms such as RMSPROP that adaptively vary the effective Learning Rate.

$$\Lambda(n) = \beta\Lambda(n-1) + (1-\beta)g(n), \quad \hat{\Gamma}(n) = \frac{\Gamma(n)}{1-\beta^n}$$

$$\Delta(n) = \alpha\Delta(n-1) + (1-\alpha)g(n)^2, \quad \hat{\Delta}(n) = \frac{\Delta(n)}{1-\alpha^n}$$

$$w(n+1) = w(n) - \eta \frac{\hat{\Lambda}(n)}{\sqrt{\hat{\Delta}(n) + \epsilon}}$$

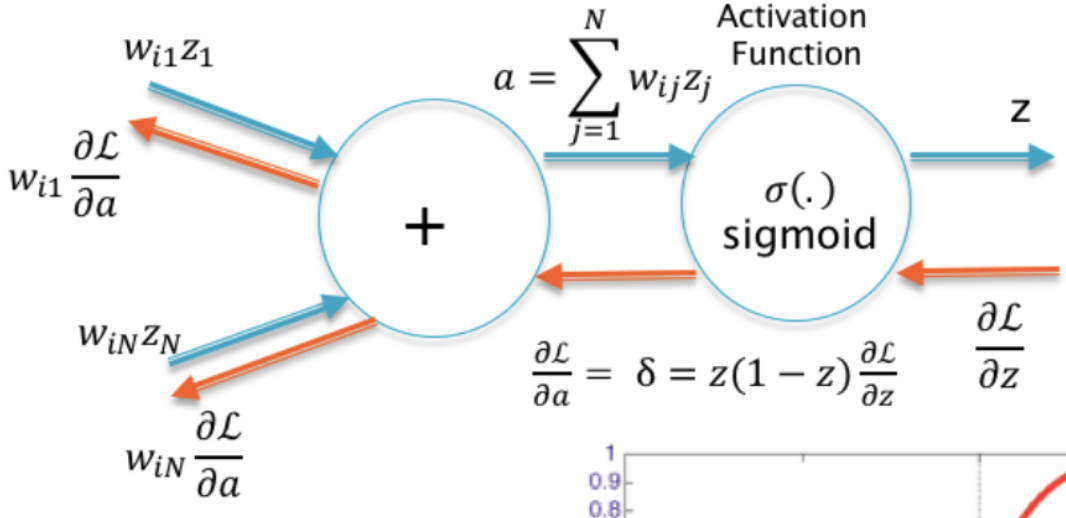
1. $\Delta(n)$ is identical to that of $E[g^2]_n$ in the RMSPROP, and it serves an identical purpose, i.e., rates for parameters with larger gradients are equalized with those for parameters with smaller gradients.

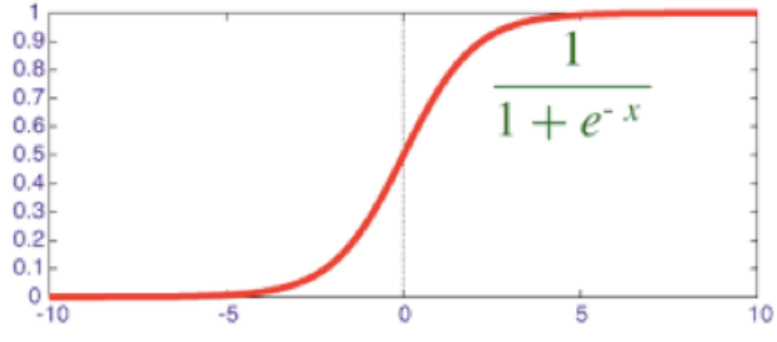
1. The sequence $\Lambda(n)$ is used to provide "Momentum" to the updates, and works in a fashion similar to the velocity sequence $v(n)$ in the Momentum algorithm.

$$\Lambda(n) = (1 - \rho) \sum_{i=0}^{n} \rho^{n-i} g(i) \leq \frac{g_{max}}{1 - \rho}$$

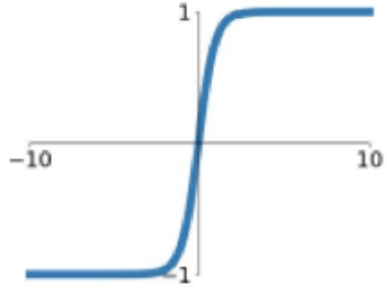
1. The parameters α and β are usually defaulted to 10^{-8} and 0.999 respectively.

Back to Activation Functions (Vanishing Gradients)





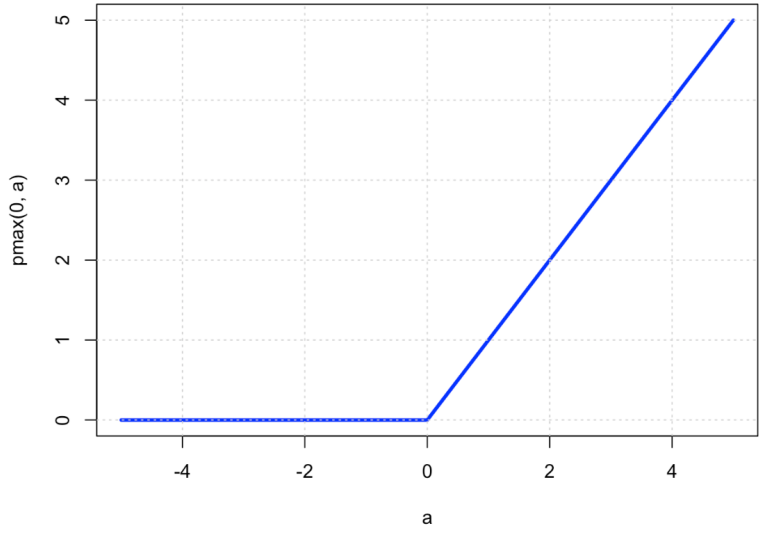
tanh Activation



$$z = \tanh a = \frac{e^{a} - e^{-a}}{e^{a} + e^{-a}}$$

1. Unless the input is in the neghborhood of zero, the function enters its saturated regime.
2. It is superior to the sigmoid in one respect, i.e., its output is zero centered. This speeds up the training process.
3. The tanh function is rarely used in modern DLNs, the exception being a type DLN called LSTM.

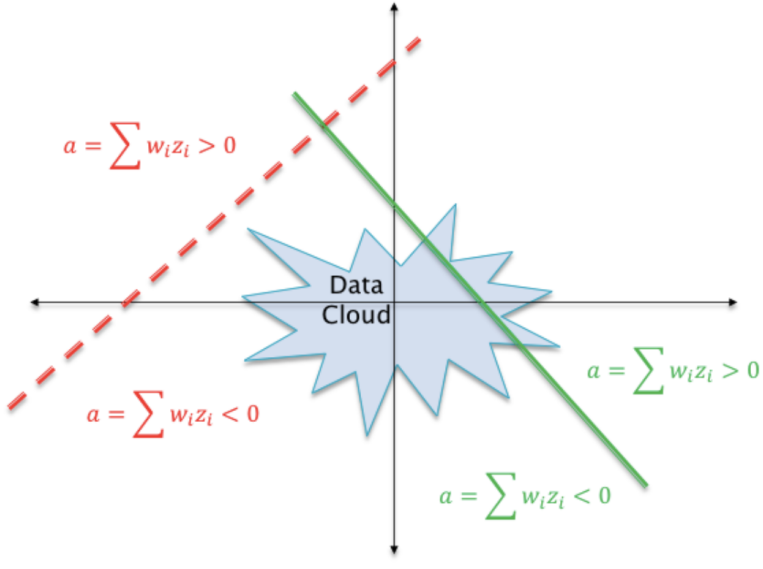
ReLU Activation



$$z = \max(0, a)$$

1. No saturation problem.
2. Gradients $\frac{\partial L}{\partial w}$ propagate undiminished through the network, provided all the nodes are active.

Dead ReLU Problem

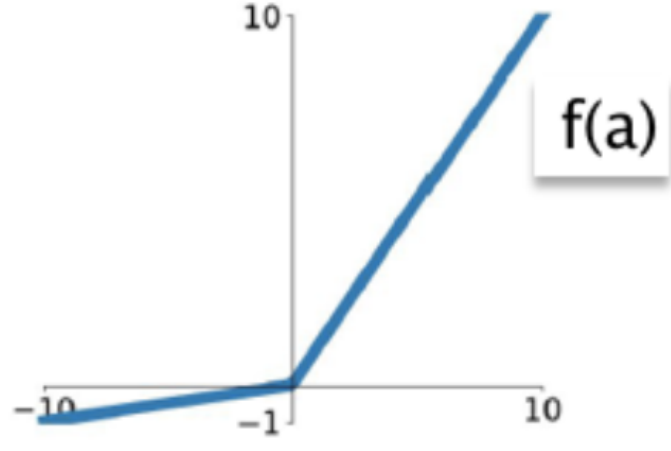


1. The dotted line in this figure shows a case in which the weight parameters w_i are such that the hyperplane $\sum w_i z_i$ does not intersect the "data cloud" of possible input activations. There does not exist any possible input values that can lead to $\sum w_i z_i > 0$. The neuron's output activation will always be zero, and it will kill all gradients backpropagating down from higher layers.

2. Vary initialization to correct this.

Leaky ReLU

Leaky ReLU
z = max(ca,a)

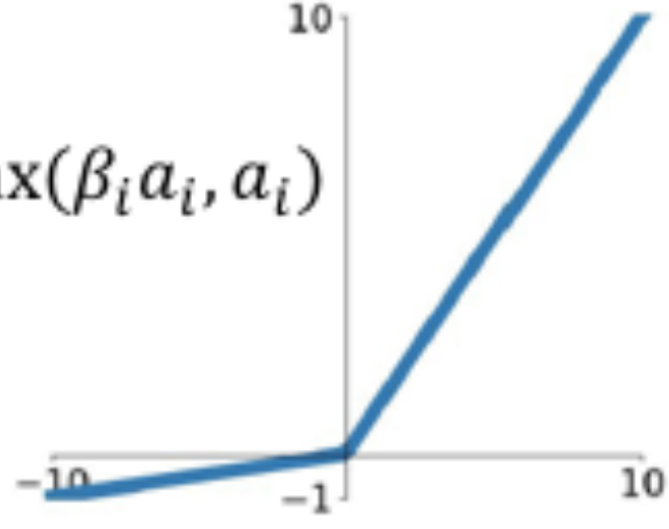


$$z = \max(ca, a), \quad 0 \leq c < 1$$

- Fixes the dead ReLU problem.

PreLU

$$z_i = \max(\beta_i a_i, a_i)$$



$$z_i = \max(\beta_i a_i, a_i), \quad 1 \leq i \leq S$$

1. Instead of deciding on the value of c through experimentation, why not determine it using Backpropagation as well. This is the idea behind the Pre-ReLU or PReLU function.

Note that each neuron i now has its own parameter β_i, $1 \leq i \leq S$, where S is the number of nodes in the network. These parameters are iteratively estimated using Backprop.

$$\frac{\partial \mathcal{L}}{\partial \beta_i} = \frac{\partial \mathcal{L}}{\partial z_i} \frac{\partial z_i}{\partial \beta_i}, \quad 1 \leq i \leq S$$

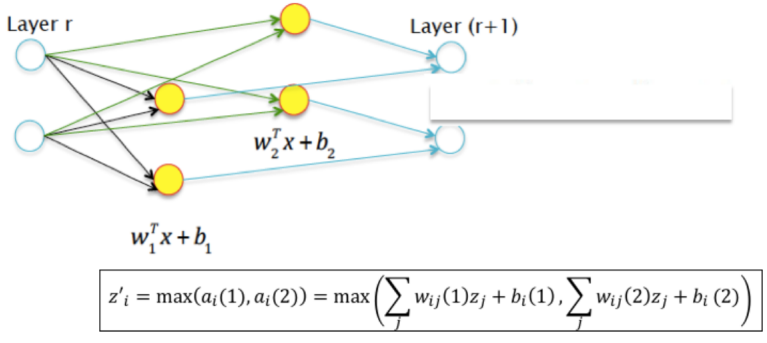
Substituting the value for $\frac{\partial z_i}{\partial \beta_i}$ we obtain

$$\frac{\partial \mathcal{L}}{\partial \beta_i} = a_i \frac{\partial \mathcal{L}}{\partial z_i} \quad if \ a_i \leq 0 \ \text{ and } \ 0 \ \text{ otherwise}$$

which is then used to update β_i using $\beta_i \rightarrow \beta_i - \eta \frac{\partial \mathcal{L}}{\partial \beta_i}$.

Once training is complete, the PreLU based DLN network ends up with a different value of β_i at each neuron, which increases the flexibility of the network at the cost of an increase in the number of parameters.

Maxout



Generalizes Leaky ReLU.

$$z'_i = \max(c\Big[\sum_j w_{ij} z_j + b_i\Big], \sum_j w_{ij} z_j + b_i),$$

We may allow the two hyperplanes to be independent with their own set of parameters, as shown in the Figure above.

Initializing Weights

In practice, the DLN weight parameters are initialized with random values drawn from Gaussian or Uniform distributions and the following rules are used:

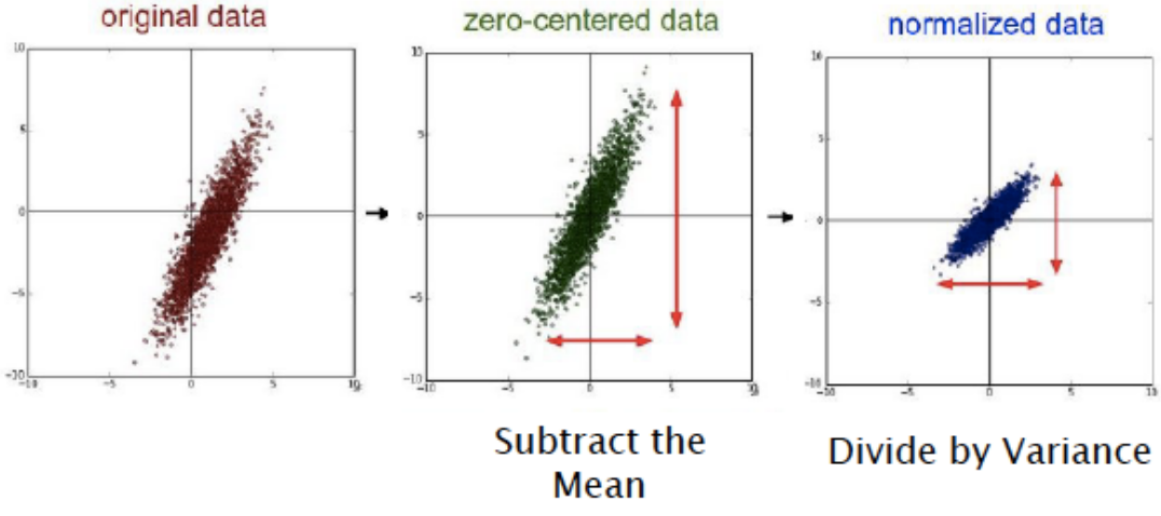
- Guassian Initialization: If the weight is between layers with n_{in} input neurons and n_{out} output neurons, then they are initialized using a Gaussian random distribution with mean zero and standard deviation $\sqrt{\frac{2}{n_{in}+n_{out}}}$.
- Uniform Initialization: In the same configuration as above, the weights should be initialized using an Uniform distribution between $-r$ and r, where $r = \sqrt{\frac{6}{n_{in}+n_{out}}}$.

When using the ReLU or its variants, these rules have to be modified slightly:

- Guassian Initialization: If the weight is between layers with n_{in} input neurons and n_{out} output neurons, then they are initialized using a Gaussian random distribution with mean zero and standard deviation $\sqrt{\frac{4}{n_{in}+n_{out}}}$.
- Uniform Initialization: In the same configuration as above, the weights should be initialized using an Uniform distribution between $-r$ and r, where $r = \sqrt{\frac{12}{n_{in}+n_{out}}}$.

The reasoning behind scaling down the initialization values as the number of incident weights increases is to prevent saturation of the node activations during the forward pass of the Backprop algorithm, as well as large values of the gradients during backward pass.

Data Preprocessing



Centering: This is also sometimes called Mean Subtraction, and is the most common form of preprocessing. Given an input dataset consisting of M vectors $X(m) = (x_1(m), \ldots, x_N(m)), m = 1, \ldots, M$, it consists of subtracting the mean across each individual input component $x_i, 1 \leq i \leq N$ such that

$$x_i(m) \leftarrow x_i(m) - \frac{\sum_{s=1}^{M} x_i(s)}{M}, \quad 1 \leq i \leq N, 1 \leq m \leq M$$

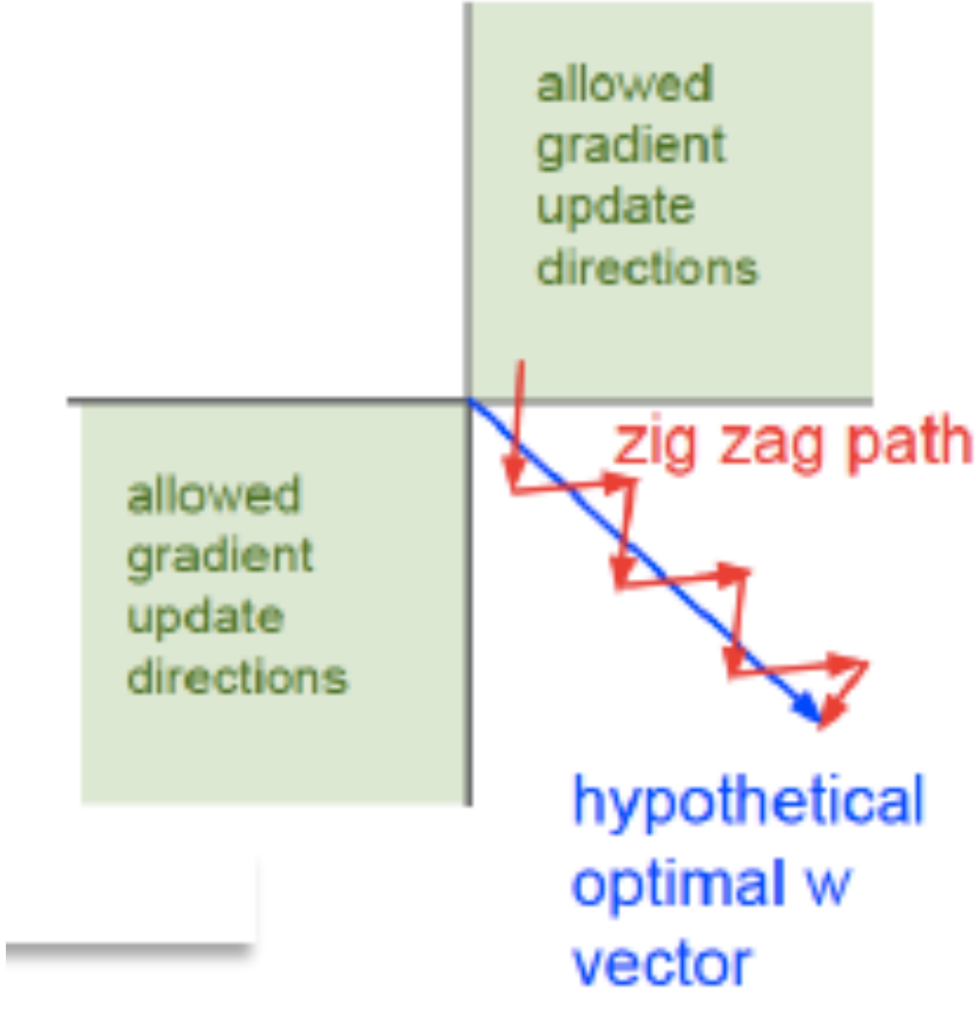
Scaling: After the data has been centered, it can be scaled in one of two ways:

- By dividing by the standard deviation, once again along each dimension, so that the overall transform is

$$x_i(m) \leftarrow \frac{x_i(m) - -\frac{\sum_{s=1}^{M} x_i(s)}{M}}{\sigma_i}, \quad 1 \leq i \leq N, 1 \leq m \leq M$$

- By Normalizing each dimension so that the min and max along each axis are -1 and +1 repectively.
- In general Scaling helps optimization because it balances out the rate at which the weights connected to the input nodes learn.

Zero Centering Helps



Recall that for a K-ary Linear Classifier, the parameter update equation is given by:

$$w_{kj} \leftarrow w_{kj} - \eta x_j(y_k - t_k), \quad 1 \leq k \leq K, \quad 1 \leq j \leq N$$

If the training sample is such that $t_q = 1$ and $t_k = 0, j \neq q$, then the update becomes:

$$w_{qj} \leftarrow w_{qj} - \eta x_j(y_q - 1)$$

and

$$w_{kj} \leftarrow w_{kj} - \eta x_j(y_k), \quad k \neq q$$

Lets assume that the input data is not centered so that $x_j \geq 0, j = 1, \ldots, N$. Since $0 \leq y_k \leq 1$ it follows that

$$\Delta w_{kj} = -\eta x_j y_k < 0, k \neq q$$

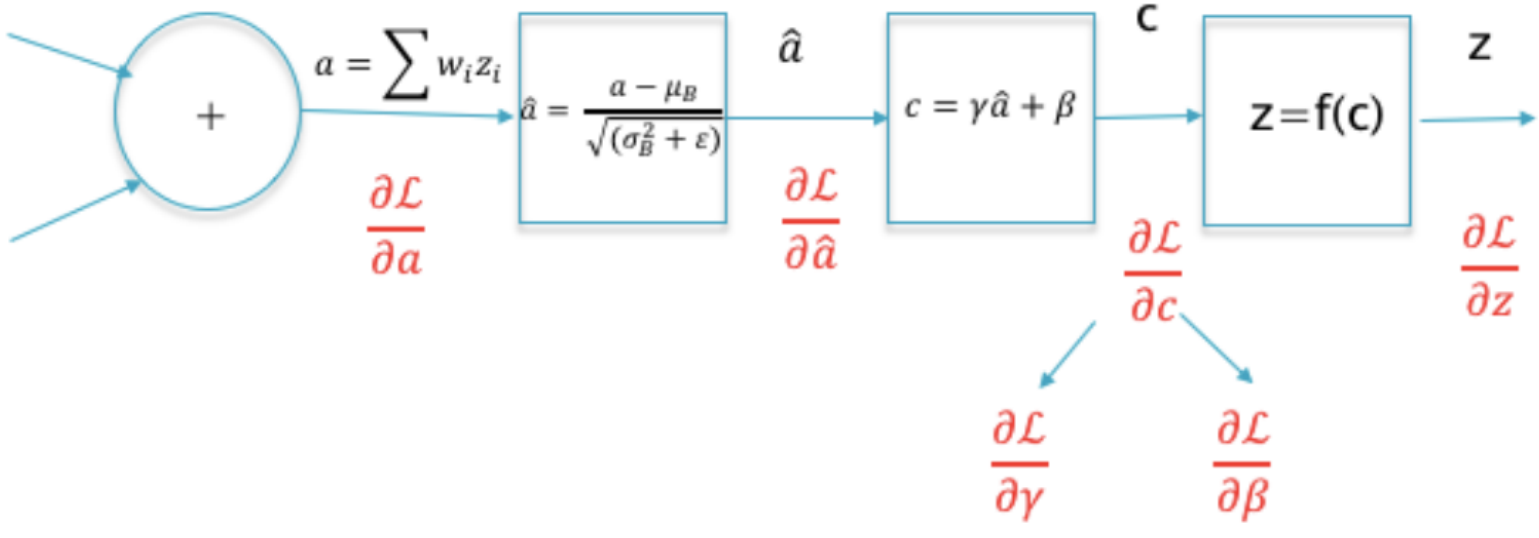
and

$$\Delta w_{qj} = -\eta x_j (y_q - 1) > 0$$

i.e. the update results in all the weights moving in the same direction, except for one. This is shown graphically in the Figure above, in which the system is trying move in the direction of the blue arrow which is the quickest path to the minimum. However if the input data is not centered, then it is forced to move in a zig-zag fashion as shown in the red-curve. The zig-zag motion is caused due to the fact that all the parameters move in the same direction at each step due to the lact of zero-centering in the input data.

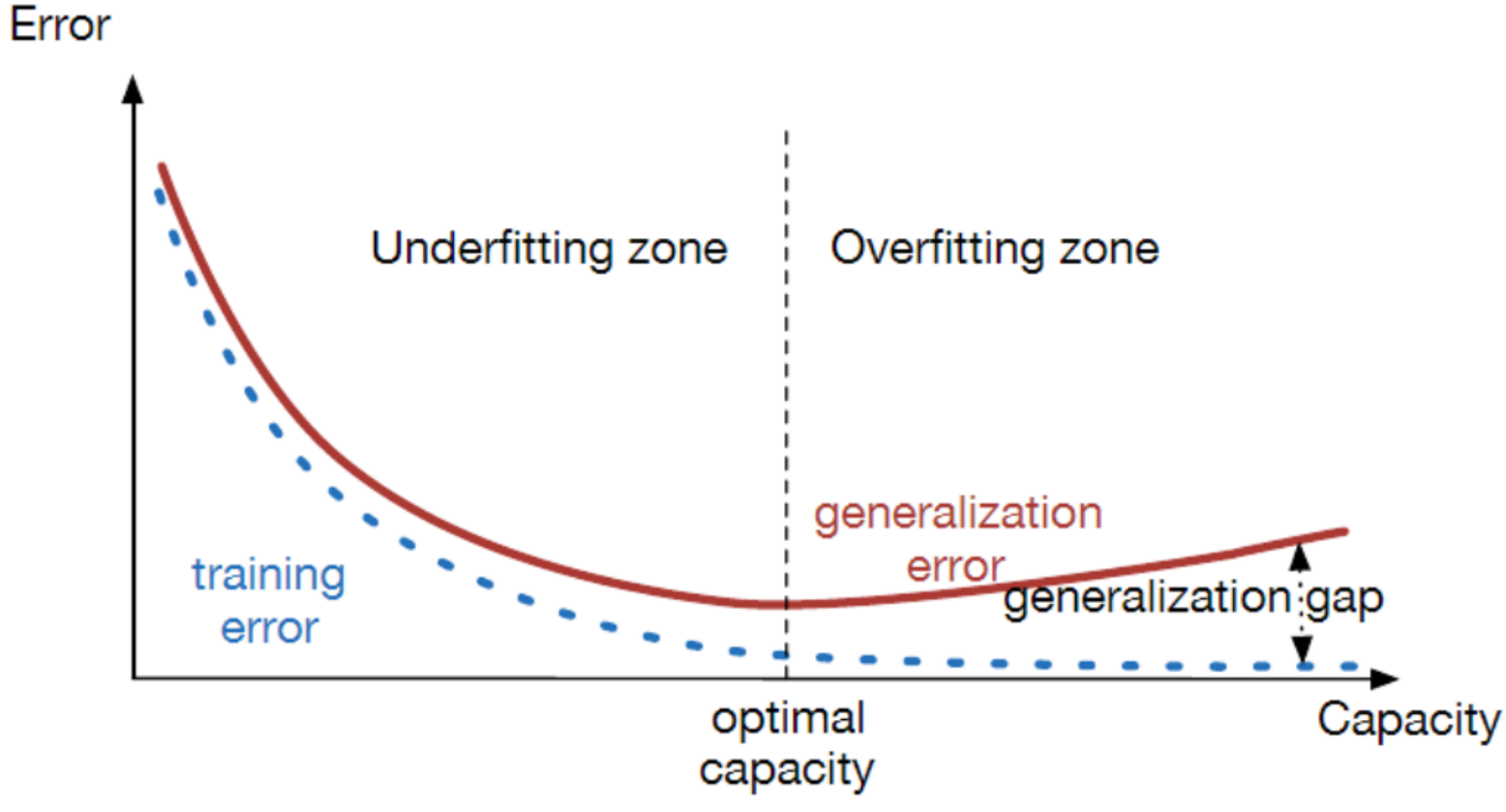
Batch Normalization

Normalization applied to the hidden layers.



- Higher learning rates: In a non-normalized network, a large learning rate can lead to oscillations and cause the loss function increase rather than decrease.
- Better Gradient Propagation through the network, enabling DLNs with more hidden layers.
- Reduces strong dependencies on the parameter initialization values.
- Helps to regularize the model.

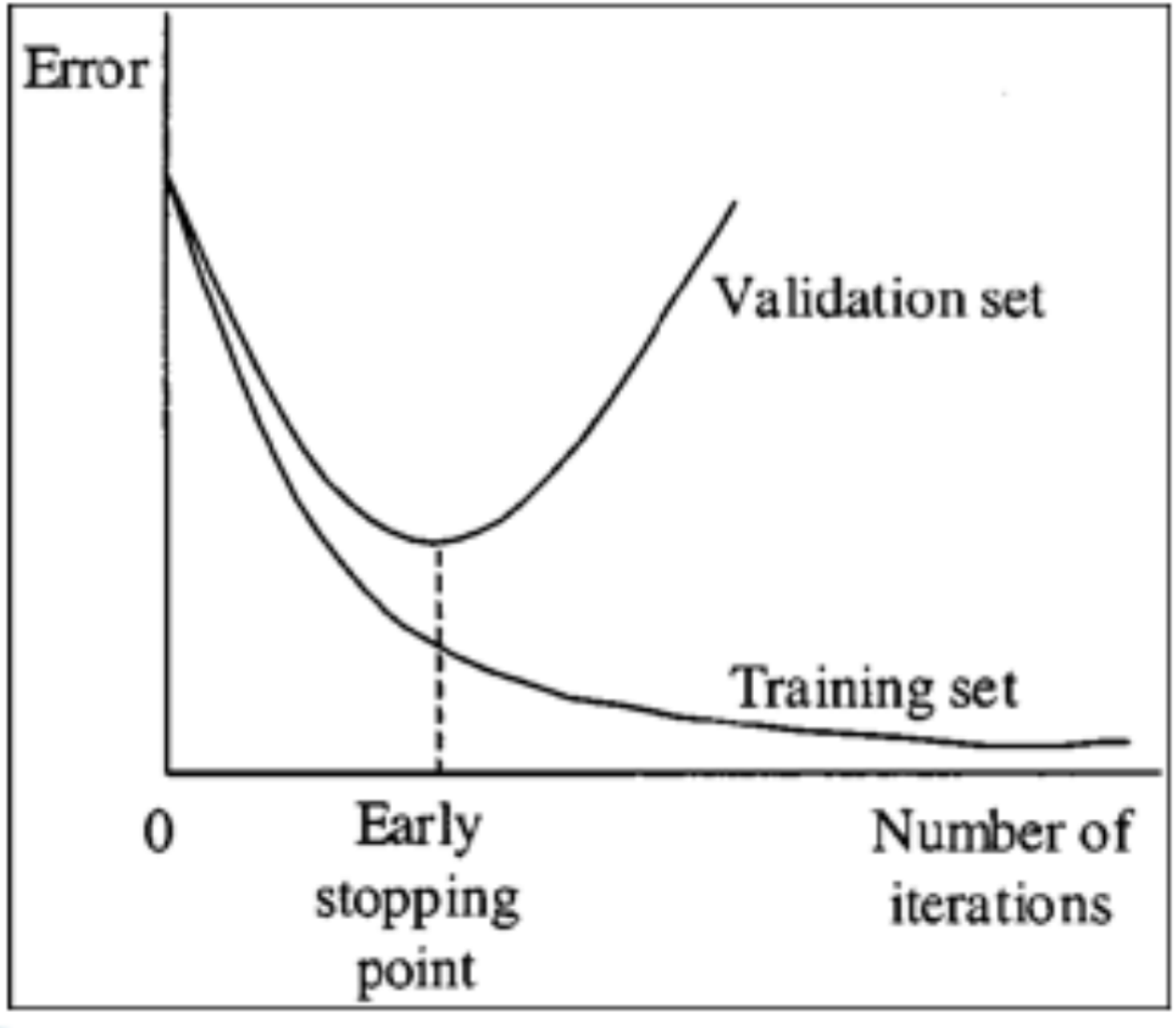
Under and Over-fitting



Regularization

- Early Stopping
- L1 Regularization
- L2 Regularization
- Dropout Regularization
- Training Data Augmentation
- Batch Normalization

Early Stopping



L2 Regularization

L2 Regularization is a commonly used technique in ML systems is also sometimes referred to as “Weight Decay”. It works by adding a quadratic term to the Cross Entropy Loss Function $\mathcal{L}$, called the Regularization Term, which results in a new Loss Function $\mathcal{L}_R$ given by:

$$\mathcal{L}_R = \mathcal{L} + \frac{\lambda}{2} \sum_{r=1}^{R+1} \sum_{j=1}^{P^{r-1}} \sum_{i=1}^{P^r} (w_{ij}^{(r)})^2$$

L2 Regularization also leads to more "diffuse" weight parameters, in other words, it encourages the network to use all its inputs a little rather than some of its inputs a lot.

L1 Regularization

L1 Regularization uses a Regularization Function which is the sum of the absolute value of all the weights in DLN, resulting in the following loss function ($\mathcal{L}$ is the usual Cross Entropy loss):

$$\mathcal{L}_R = \mathcal{L} + \lambda \sum_{r=1}^{R+1} \sum_{j=1}^{P^{r-1}} \sum_{i=1}^{P^r} |w_{ij}^{(r)}|$$

At a high level L1 Regularization is similar to L2 Regularization since it leads to smaller weights.

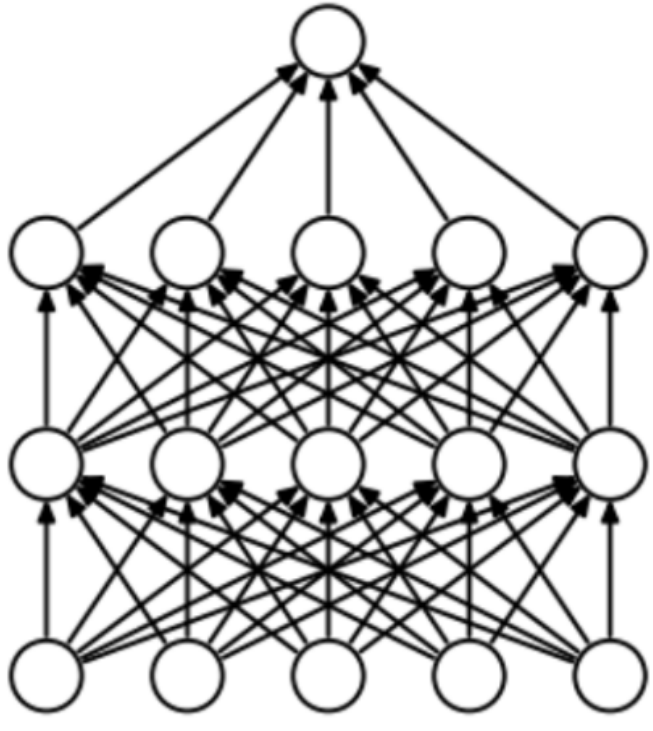
1. Both L1 and L2 Regularizations lead to a reduction in the weights with each iteration. However the way the weights drop is different:

2. In L2 Regularization the weight reduction is multiplicative and proportional to the value of the weight, so it is faster for large weights and de-accelerates as the weights get smaller.

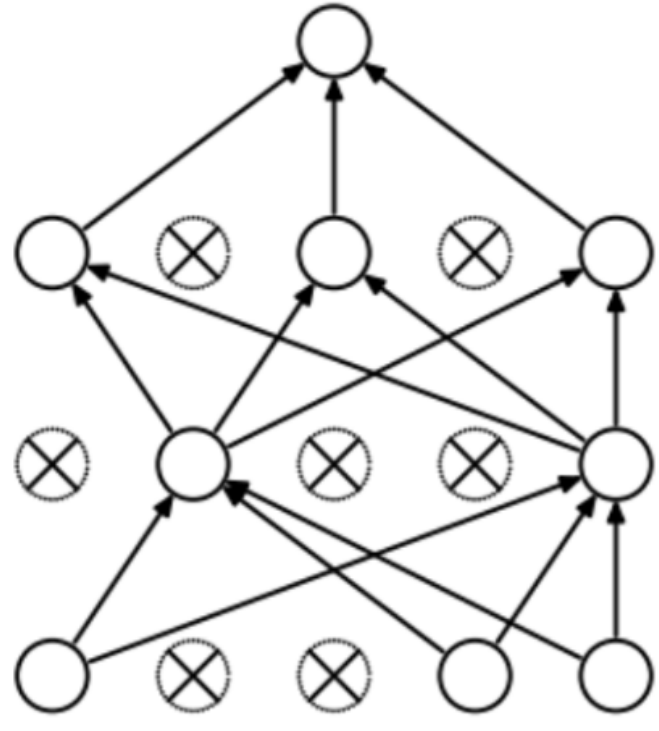
3. In L1 Regularization on the other hand, the weights are reduced by a fixed amount in every iteration, irrespective of the value of the weight. Hence for larger weights L2 Regularization is faster than L1, while for smaller weights the reverse is true.

4. As a result L1 Regularization leads to DLNs in which the weight of most of the connections tends towards zero, with a few connections with larger weights left over. This type of DLN that results after the application of L1 Regularization is said to be “sparse”.

Dropout regularization



(a) Standard Neural Net



(b) After applying dropout.

The basic idea behind Dropout is to run each iteration of the Backprop algorithm on randomly modified versions of the original DLN. The random modifications are carried out to the topology of the DLN using the following rules:

- Assign probability values $p^{(r)}$, $0 \leq r \leq R$, which is defined as the probability that a node is present in the model, and use these to generate $\{0, 1\}$-valued Bernoulli random variables $e_j^{(r)}$:

$$e_j^{(r)} \sim Bernoulli(p^{(r)}), \quad 0 \leq r \leq R, \;\; 1 \leq j \leq P^r$$

- Modify the input vector as follows:

$$\hat{x}_j = e_j^{(0)} x_j, \quad 1 \leq j \leq N$$

- Modify the activations $z_j^{(r)}$ of the hidden layer r as follows:

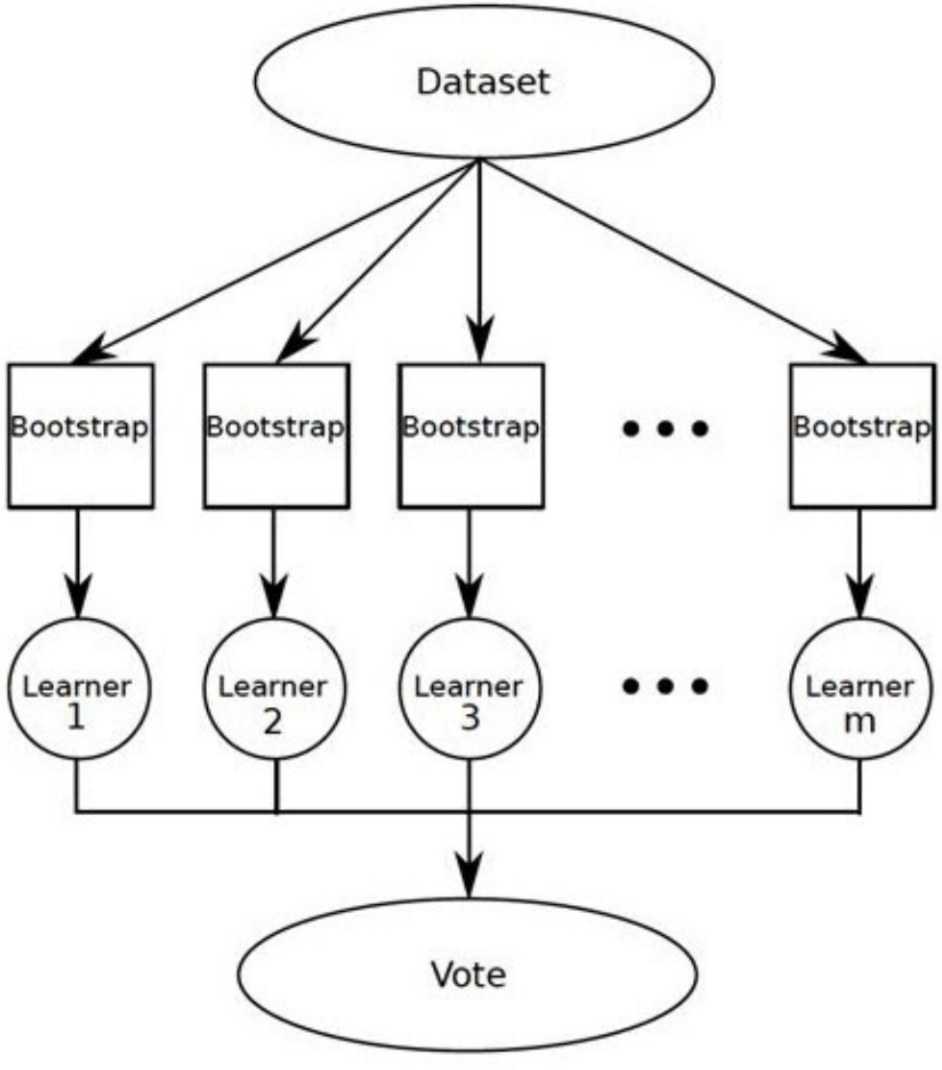
$$\hat{z}_j^{(r)} = e_j^{(r)} z_j^{(r)}, \quad 1 \leq r \leq R, \;\; 1 \leq j \leq P^r$$

1. After the Backprop is complete, we have effectively trained a collection of up to 2^s thinned DLNs all of which share the same weights, where s is the total number of hidden nodes in the DLN.

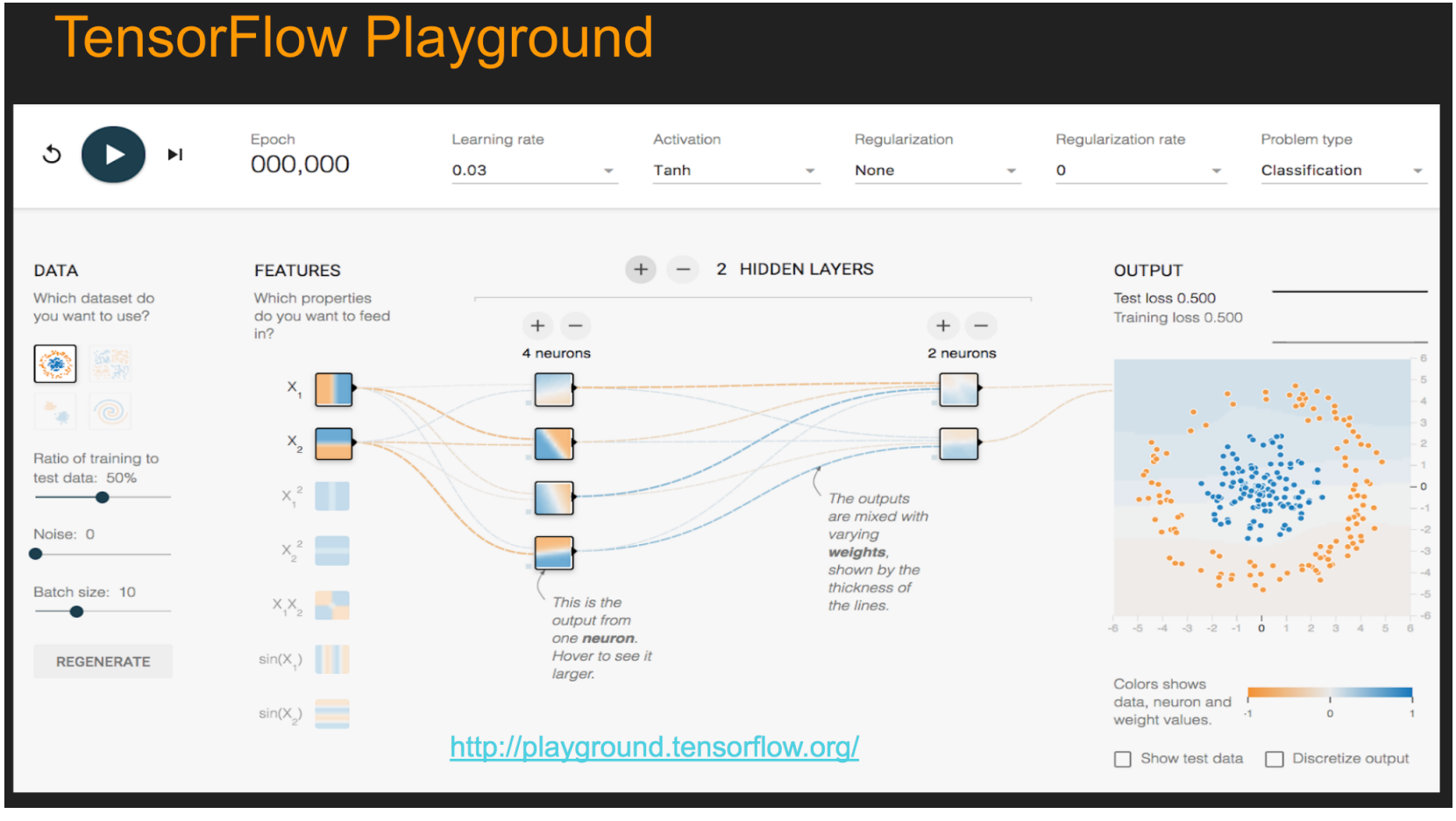
2. In order to test the network, strictly speaking we should be averaging the results from all these thinned models, however a simple approximate averaging method works quite well.

3. The main idea is to use the complete DLN as the test network.

Bagging (Ensemble Learning)



TensorFlow Playground



http://playground.tensorflow.org/ (http://playground.tensorflow.org/)

Pattern Recognition: Cancer

In [2]:
```
## Read in the data set
data = pd.read_csv("data/BreastCancer.csv")
data.head()
```

Out[2]:

	Id	Cl.thickness	Cell.size	Cell.shape	Marg.adhesion	Epith.c.size	
0	1000025	5	1	1	1	2	
1	1002945	5	4	4	5	7	
2	1015425	3	1	1	1	2	
3	1016277	6	8	8	1	3	
4	1017023	4	1	1	3	2	

In [4]:
```
## Convert the class variable into binary numeric
ynum = zeros((len(x),1))
for j in arange(len(y)):
    if y[j]=="malignant":
        ynum[j]=1
ynum[:10]
```

Out[4]:
```
array([[0.],
       [0.],
       [0.],
       [0.],
       [0.],
       [1.],
       [0.],
       [0.],
       [0.],
       [0.]])
```

In [5]:
```
## Make label data have 1-shape, 1=malignant
from keras import utils
y.labels = utils.to_categorical(ynum, num_classes=2)
#x = x.as_matrix()
print(y.labels[:10])
print(shape(x))
print(shape(y.labels))
print(shape(ynum))
```

```
/Users/srdas/anaconda3/lib/python3.6/site-packages/h5py/__init__.py:36: Future
Warning: Conversion of the second argument of issubdtype from `float` to `np.f
loating` is deprecated. In future, it will be treated as `np.float64 == np.dty
pe(float).type`.
  from ._conv import register_converters as _register_converters
Using TensorFlow backend.

[[1. 0.]
 [1. 0.]
 [1. 0.]
 [1. 0.]
 [1. 0.]
 [0. 1.]
 [1. 0.]
 [1. 0.]
 [1. 0.]
 [1. 0.]]
(683, 9)
(683, 2)
(683, 1)
```

In [6]:

```
## Define the neural net and compile it
from keras.models import Sequential
from keras.layers import Dense, Activation

model = Sequential()
model.add(Dense(32, activation='relu', input_dim=9))
model.add(Dense(32, activation='relu'))
model.add(Dense(32, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

In [7]:
```
## Fit/train the model (x,y need to be matrices)
model.fit(x, ynum, epochs=25, batch_size=32,verbose=2)
```

```
Epoch 1/25
 - 0s - loss: 0.6186 - acc: 0.5476
Epoch 2/25
 - 0s - loss: 0.4578 - acc: 0.8946
Epoch 3/25
 - 0s - loss: 0.3477 - acc: 0.9151
Epoch 4/25
 - 0s - loss: 0.2756 - acc: 0.9458
Epoch 5/25
 - 0s - loss: 0.2359 - acc: 0.9444
Epoch 6/25
 - 0s - loss: 0.2116 - acc: 0.9414
Epoch 7/25
 - 0s - loss: 0.1978 - acc: 0.9502
Epoch 8/25
 - 0s - loss: 0.1680 - acc: 0.9561
Epoch 9/25
 - 0s - loss: 0.1535 - acc: 0.9649
Epoch 10/25
 - 0s - loss: 0.1389 - acc: 0.9649
Epoch 11/25
 - 0s - loss: 0.1416 - acc: 0.9561
Epoch 12/25
 - 0s - loss: 0.1184 - acc: 0.9649
Epoch 13/25
 - 0s - loss: 0.1204 - acc: 0.9649
Epoch 14/25
 - 0s - loss: 0.1076 - acc: 0.9678
Epoch 15/25
 - 0s - loss: 0.0929 - acc: 0.9678
Epoch 16/25
 - 0s - loss: 0.0966 - acc: 0.9707
Epoch 17/25
```

```
 - 0s - loss: 0.0902 - acc: 0.9736
Epoch 18/25
 - 0s - loss: 0.0809 - acc: 0.9722
Epoch 19/25
 - 0s - loss: 0.0824 - acc: 0.9766
Epoch 20/25
 - 0s - loss: 0.0741 - acc: 0.9766
Epoch 21/25
 - 0s - loss: 0.0773 - acc: 0.9751
Epoch 22/25
 - 0s - loss: 0.0657 - acc: 0.9839
Epoch 23/25
 - 0s - loss: 0.0671 - acc: 0.9810
Epoch 24/25
 - 0s - loss: 0.0647 - acc: 0.9780
Epoch 25/25
 - 0s - loss: 0.0635 - acc: 0.9795
```

Out[7]: <keras.callbacks.History at 0x1134a5400>

In [8]:
```
## Accuracy
yhat = model.predict_classes(x, batch_size=32)
acc = sum(yhat==ynum)
print("Accuracy = ",acc/len(ynum))

## Confusion matrix
from sklearn.metrics import confusion_matrix
confusion_matrix(yhat,ynum)
```

```
Accuracy =  0.9853587115666179
```

Out[8]:
```
array([[434,   0],
       [ 10, 239]])
```

Another Canonical Example: Digit Recognition (MNIST)

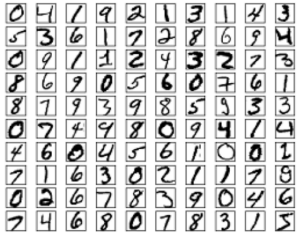
- Extensible to many finance prediction problems.
- Information set: 784 (28 x 28) pixels for category prediction.
- Would you run a multinomial regression on these 784 columns.

THE MNIST DATABASE

of handwritten digits

Yann LeCun, Courant Institute, NYU
Corinna Cortes, Google Labs, New York
Christopher J.C. Burges, Microsoft Research, Redmond

The MNIST database of handwritten digits, available from this page, has a training set of 60,000 examples, and a test set of 10,000 examples. It is a subset of a larger set available from NIST. The digits have been size-normalized and centered in a fixed-size image.



	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15	C16	C17	C18	C19	C20	C21	C22	C23	C24	C25	C26	C27	C28
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
21	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
22	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
23	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
24	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

60,000 x 784

Showing 1 to 25 of 60,000 entries

https://www.cs.toronto.edu/~kriz/cifar.html (https://www.cs.toronto.edu/~kriz/cifar.html)

In [9]:
```
## Read in the data set
train = pd.read_csv("data/train.csv", header=None)
test = pd.read_csv("data/test.csv", header=None)
print(shape(train))
print(shape(test))
```

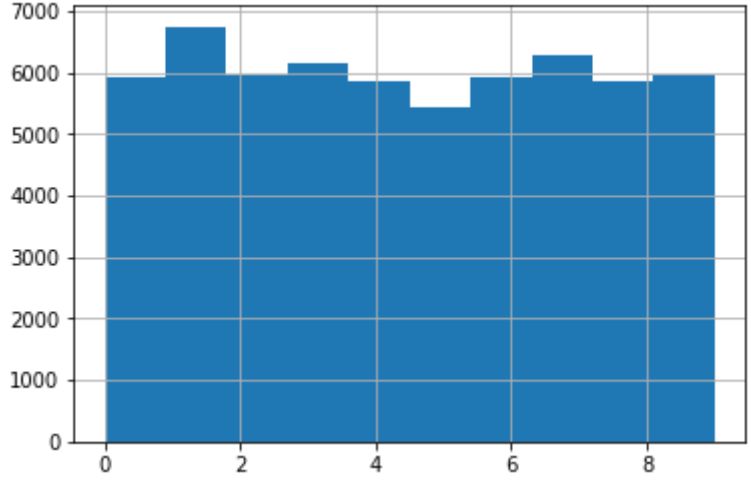
```
(60000, 785)
(10000, 785)
```

In [10]:
```
## Reformat the data
X_train = train.as_matrix()[:,0:784]
Y_train = train.as_matrix()[:,784:785]
print(shape(X_train))
print(shape(Y_train))
X_test = test.as_matrix()[:,0:784]
Y_test = test.as_matrix()[:,784:785]
print(shape(X_test))
print(shape(Y_test))
y.labels = utils.to_categorical(Y_train, num_classes=10)
print(shape(y.labels))
print(y.labels[1:5,:])
print(Y_train[1:5])
```

```
(60000, 784)
(60000, 1)
(10000, 784)
(10000, 1)
(60000, 10)
[[0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]]
[[3]
 [0]
 [0]
 [2]]

/Users/srdas/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:2: Fu
tureWarning: Method .as_matrix will be removed in a future version. Use .value
s instead.

/Users/srdas/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:3: Fu
tureWarning: Method .as_matrix will be removed in a future version. Use .value
s instead.
  This is separate from the ipykernel package so we can avoid doing imports un
```

```
til
/Users/srdas/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:6: Fu
tureWarning: Method .as_matrix will be removed in a future version. Use .value
s instead.

/Users/srdas/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:7: Fu
tureWarning: Method .as_matrix will be removed in a future version. Use .value
s instead.
  import sys
```

In [11]:
```
hist(Y_train); grid()
```



In [12]:

```
## Define the neural net and compile it
from keras.models import Sequential
from keras.layers import Dense, Activation, Dropout
from keras.optimizers import SGD

data_dim = shape(X_train)[1]

model = Sequential([
    Dense(100, input_shape=(784,)),
    Activation('sigmoid'),
    Dense(100),
    Activation('sigmoid'),
    Dense(100),
    Activation('sigmoid'),
    Dense(100),
    Activation('sigmoid'),
    Dense(10),
    Activation('softmax'),
])

#model = Sequential()
#model.add(Dense(100, activation='sigmoid', input_dim=data_dim))
#model.add(Dropout(0.25))
#model.add(Dense(100, activation='sigmoid'))
#model.add(Dropout(0.25))
#model.add(Dense(100, activation='sigmoid'))
#model.add(Dropout(0.25))
#model.add(Dense(100, activation='sigmoid'))
#model.add(Dropout(0.25))
#model.add(Dense(10, activation='softmax'))
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

In [13]:
```
## Fit/train the model (x,y need to be matrices)
model.fit(X_train, y.labels, epochs=10, batch_size=32,verbose=2)
```

```
Epoch 1/10
 - 7s - loss: 0.7339 - acc: 0.7693
Epoch 2/10
 - 7s - loss: 0.3392 - acc: 0.9016
Epoch 3/10
 - 8s - loss: 0.2837 - acc: 0.9159
Epoch 4/10
 - 9s - loss: 0.2514 - acc: 0.9253
Epoch 5/10
 - 10s - loss: 0.2290 - acc: 0.9307
Epoch 6/10
 - 9s - loss: 0.2204 - acc: 0.9336
Epoch 7/10
 - 10s - loss: 0.2124 - acc: 0.9357
Epoch 8/10
 - 9s - loss: 0.2053 - acc: 0.9384
Epoch 9/10
 - 9s - loss: 0.1949 - acc: 0.9414
Epoch 10/10
 - 9s - loss: 0.1930 - acc: 0.9415
```

Out[13]:
```
<keras.callbacks.History at 0x1151adcf8>
```

In [14]:
```
## In Sample
yhat = model.predict_classes(X_train, batch_size=32)

## Confusion matrix
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(yhat,Y_train)
print(" ")
print(cm)

##
acc = sum(diag(cm))/len(Y_train)
print("Accuracy = ",acc)
```

```
[[5804    0   64   27    9   67   57   13   30   21]
 [   4 6577   21   19   17    8    8   36   77   26]
 [  11   42 5593   97   24   11   13   58   39   11]
 [   9   18   37 5539    0   82    0   32   79   43]
 [  11    6   54    3 5520   12   34   23   19  140]
 [  18   30   12  234    9 5063   56   14   96   49]
 [  26    6   72   15   44   68 5722    1   34    1]
 [   0   12   33   44   10   12    0 5995    7   99]
 [  34   48   62  119   22   74   28   14 5389   53]
 [   6    3   10   34  187   24    0   79   81 5506]]
Accuracy =  0.9451333333333334
```

In [15]:

```
## Out of Sample
yhat = model.predict_classes(X_test, batch_size=32)

## Confusion matrix
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(yhat,Y_test)
print(" ")
print(cm)

##
acc = sum(diag(cm))/len(Y_test)
print("Accuracy = ",acc)
```

```
[[ 963    0   16    2    1   12   13    2    9    6]
 [   0 1114    2    0    3    3    1   14    8    5]
 [   1    5  959   15    5    2    5   14    4    0]
 [   1    2    9  922    0   15    0    9    9    8]
 [   0    0    8    0  929    2    4    2    6   27]
 [   3    3    2   37    1  830   14    1   16    8]
 [   7    4   15    2   13    8  914    0    9    0]
 [   1    0    6    9    2    0    0  962    3   11]
 [   4    7   13   21    2   16    7    3  897   14]
 [   0    0    2    2   26    4    0   21   13  930]]
Accuracy =  0.942
```

Learning the Black-Scholes Equation

See : Hutchinson, Lo, Poggio (1994)

In [5]:
```
from scipy.stats import norm
def BSM(S,K,T,sig,rf,dv,cp):  #cp = {+1.0 (calls), -1.0 (puts)}
    d1 = (math.log(S/K)+(rf-dv+0.5*sig**2)*T)/(sig*math.sqrt(T))
    d2 = d1 - sig*math.sqrt(T)
    return cp*S*math.exp(-dv*T)*norm.cdf(d1*cp) - cp*K*math.exp(-rf*T)*norm.cdf(d2
*cp)

df = pd.read_csv('/Users/srdas/GoogleDrive/Papers/DeepLearning/DLinFinance/BlackSc
holesNN/training.csv')
```

Normalizing spot and call prices

C is homogeneous degree one, so

$$aC(S, K) = C(aS, aK)$$

This means we can normalize spot and call prices and remove a variable by dividing by K.

$$\frac{C(S, K)}{K} = C(S/K, 1)$$

In [6]:
```
df['Stock Price'] = df['Stock Price']/df['Strike Price']
df['Call Price']  = df['Call Price'] /df['Strike Price']
```

Data, libraries, activation functions

In [7]:
```
n = 300000
n_train =  (int)(0.8 * n)
train = df[0:n_train]
X_train = train[['Stock Price', 'Maturity', 'Dividends', 'Volatility', 'Risk-free'
]].values
y_train = train['Call Price'].values
test = df[n_train+1:n]
X_test = test[['Stock Price', 'Maturity', 'Dividends', 'Volatility', 'Risk-free']]
.values
y_test = test['Call Price'].values
```

In [8]:
```
#Import libraries
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, LeakyReLU
from keras import backend

def custom_activation(x):
    return backend.exp(x)
```

```
/Users/srdas/anaconda3/lib/python3.6/site-packages/h5py/__init__.py:36: Future
Warning: Conversion of the second argument of issubdtype from `float` to `np.f
loating` is deprecated. In future, it will be treated as `np.float64 == np.dty
pe(float).type`.
  from ._conv import register_converters as _register_converters
Using TensorFlow backend.
```

Set up, compile and fit the model

In [9]:
```
nodes = 120
model = Sequential()

model.add(Dense(nodes, input_dim=X_train.shape[1]))
#model.add("relu")
model.add(Dropout(0.25))

model.add(Dense(nodes, activation='elu'))
model.add(Dropout(0.25))

model.add(Dense(nodes, activation='relu'))
model.add(Dropout(0.25))

model.add(Dense(nodes, activation='elu'))
model.add(Dropout(0.25))

model.add(Dense(1))
model.add(Activation(custom_activation))

model.compile(loss='mse',optimizer='rmsprop')

model.fit(X_train, y_train, batch_size=64, epochs=10, validation_split=0.1, verbos
e=2)
```

```
Train on 216000 samples, validate on 24000 samples
Epoch 1/10
 - 11s - loss: 0.0056 - val_loss: 2.9596e-04
Epoch 2/10
 - 11s - loss: 0.0015 - val_loss: 4.5161e-04
Epoch 3/10
 - 10s - loss: 0.0011 - val_loss: 1.7058e-04
Epoch 4/10
 - 10s - loss: 9.3346e-04 - val_loss: 5.5905e-04
Epoch 5/10
 - 10s - loss: 8.1423e-04 - val_loss: 4.7955e-04
Epoch 6/10
 - 10s - loss: 7.3808e-04 - val_loss: 2.6507e-04
```

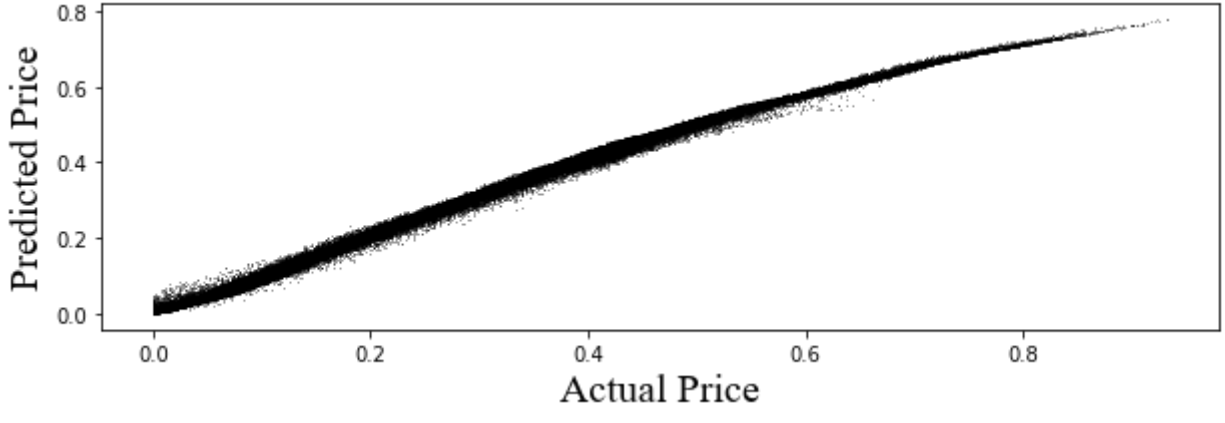
In [14]:

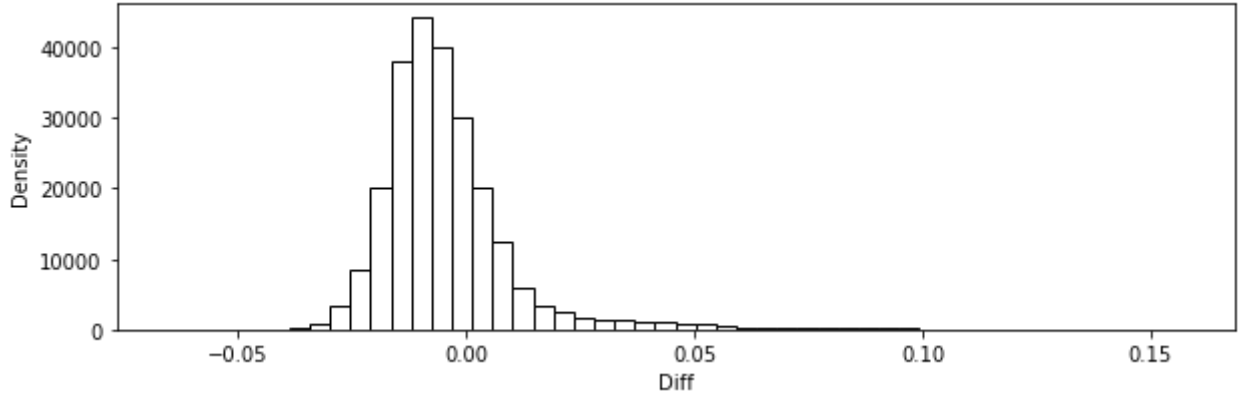
```
def CheckAccuracy(y,y_hat):
    stats = dict()

    stats['diff'] = y - y_hat

    stats['mse'] = mean(stats['diff']**2)
    print("Mean Squared Error:      ", stats['mse'])

    stats['rmse'] = sqrt(stats['mse'])
    print("Root Mean Squared Error: ", stats['rmse'])

    stats['mae'] = mean(abs(stats['diff']))
    print("Mean Absolute Error:     ", stats['mae'])

    stats['mpe'] = sqrt(stats['mse'])/mean(y)
    print("Mean Percent Error:      ", stats['mpe'])

    #plots
    mpl.rcParams['agg.path.chunksize'] = 100000
    figure(figsize=(10,3))
    plt.scatter(y, y_hat,color='black',linewidth=0.3,alpha=0.4, s=0.5)
    plt.xlabel('Actual Price',fontsize=20,fontname='Times New Roman')
    plt.ylabel('Predicted Price',fontsize=20,fontname='Times New Roman')
    plt.show()

    figure(figsize=(10,3))
    plt.hist(stats['diff'], bins=50,edgecolor='black',color='white')
    plt.xlabel('Diff')
    plt.ylabel('Density')
    plt.show()

    return stats
```

Predict and check accuracy (in-sample)

In [15]:
```
y_train_hat = model.predict(X_train)
#reduce dim (240000,1) -> (240000,) to match y_train's dim
y_train_hat = squeeze(y_train_hat)
CheckAccuracy(y_train, y_train_hat)
```

```
Mean Squared Error:       0.0002501892926358584
Root Mean Squared Error:  0.015817373126908854
Mean Absolute Error:      0.011584855780651939
Mean Percent Error:       0.05912880061083275
```

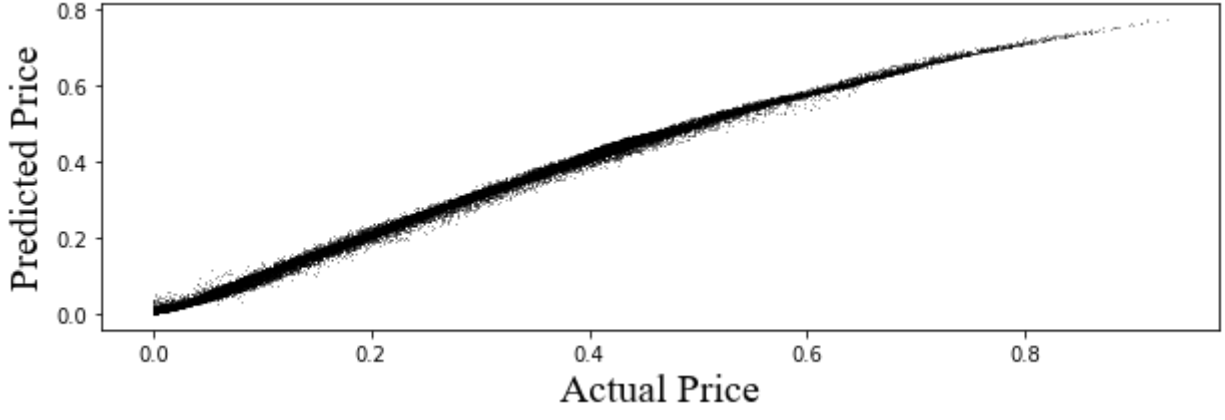


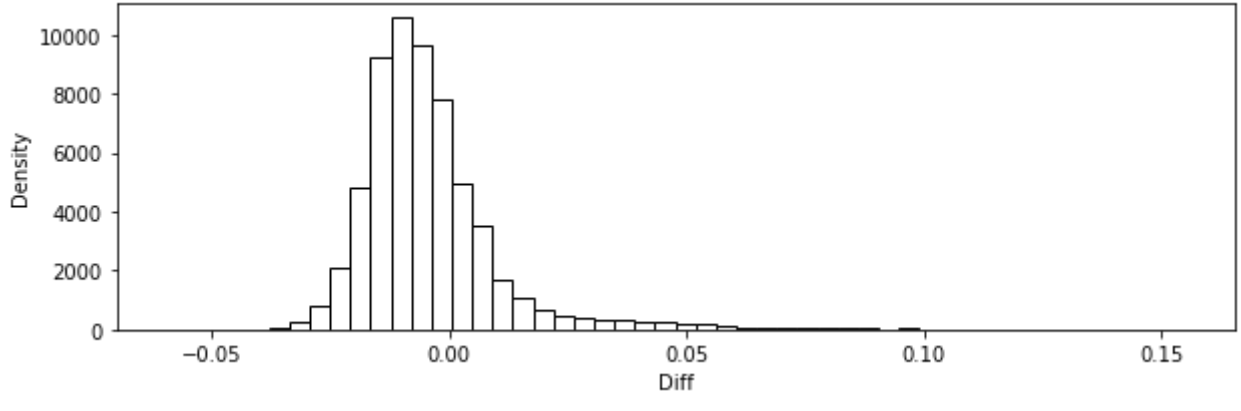


Predict and check accuracy (validation-sample)

In [16]:
```
y_test_hat = model.predict(X_test)
y_test_hat = squeeze(y_test_hat)
test_stats = CheckAccuracy(y_test, y_test_hat)
```

```
Mean Squared Error:        0.00024950104739288 44
Root Mean Squared Error:   0.01579560215353895
Mean Absolute Error:       0.011570688743185188
Mean Percent Error:        0.05913218242518501
```





Random Forest of decision trees

A Random Forest uses several decision trees to make hypotheses about regions within subsamples of the data, then makes predictions based on the majority vote of these trees. This safeguards against overfitting/memorization of the training data.

Prepare Data

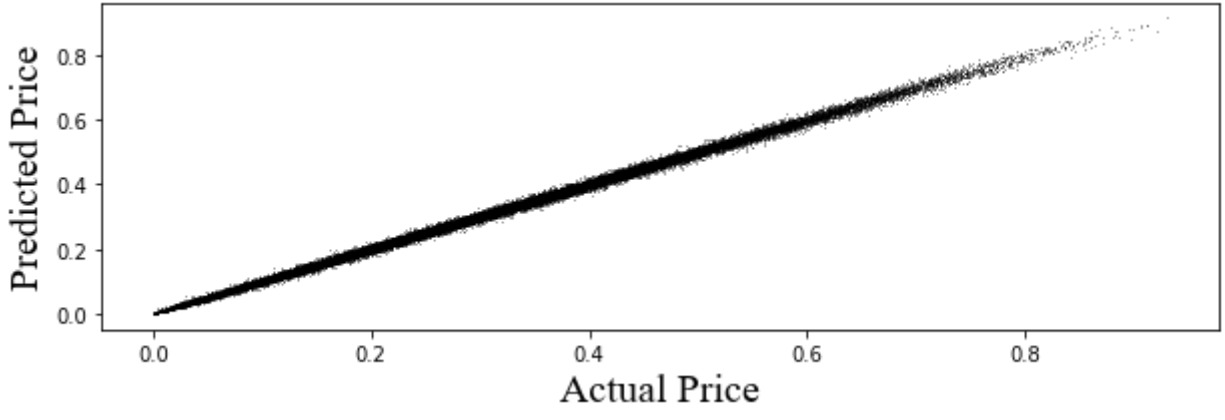
In [17]:
```
n = 300000
n_train =  (int)(0.8 * n)
train = df[0:n_train]
X_train = train[['Stock Price', 'Maturity', 'Dividends', 'Volatility', 'Risk-free'
]].values
y_train = train['Call Price'].values
test = df[n_train+1:n]
X_test = test[['Stock Price', 'Maturity', 'Dividends', 'Volatility', 'Risk-free']]
.values
y_test = test['Call Price'].values
```

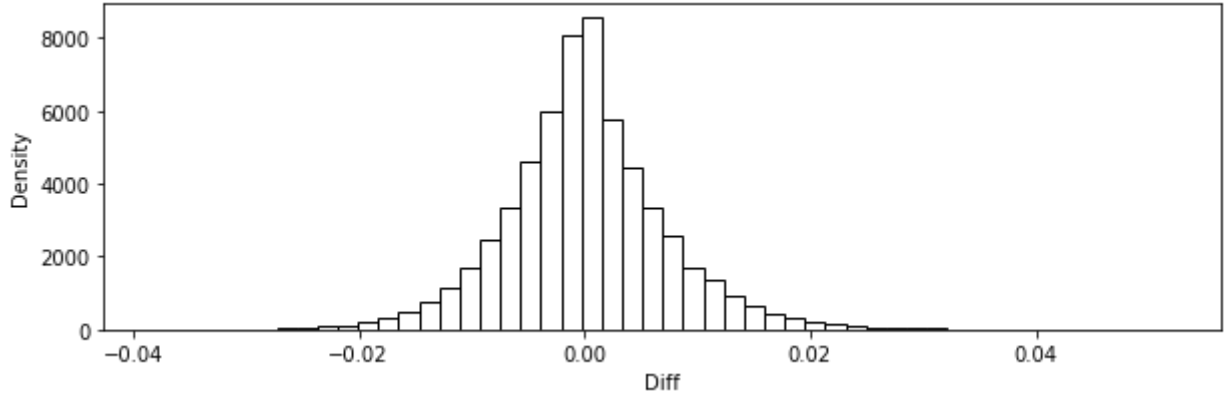
Fit Random Forest

In [18]:
```
from sklearn.ensemble import RandomForestRegressor

forest = RandomForestRegressor()
forest = forest.fit(X_train, y_train)
y_test_hat = forest.predict(X_test)
```

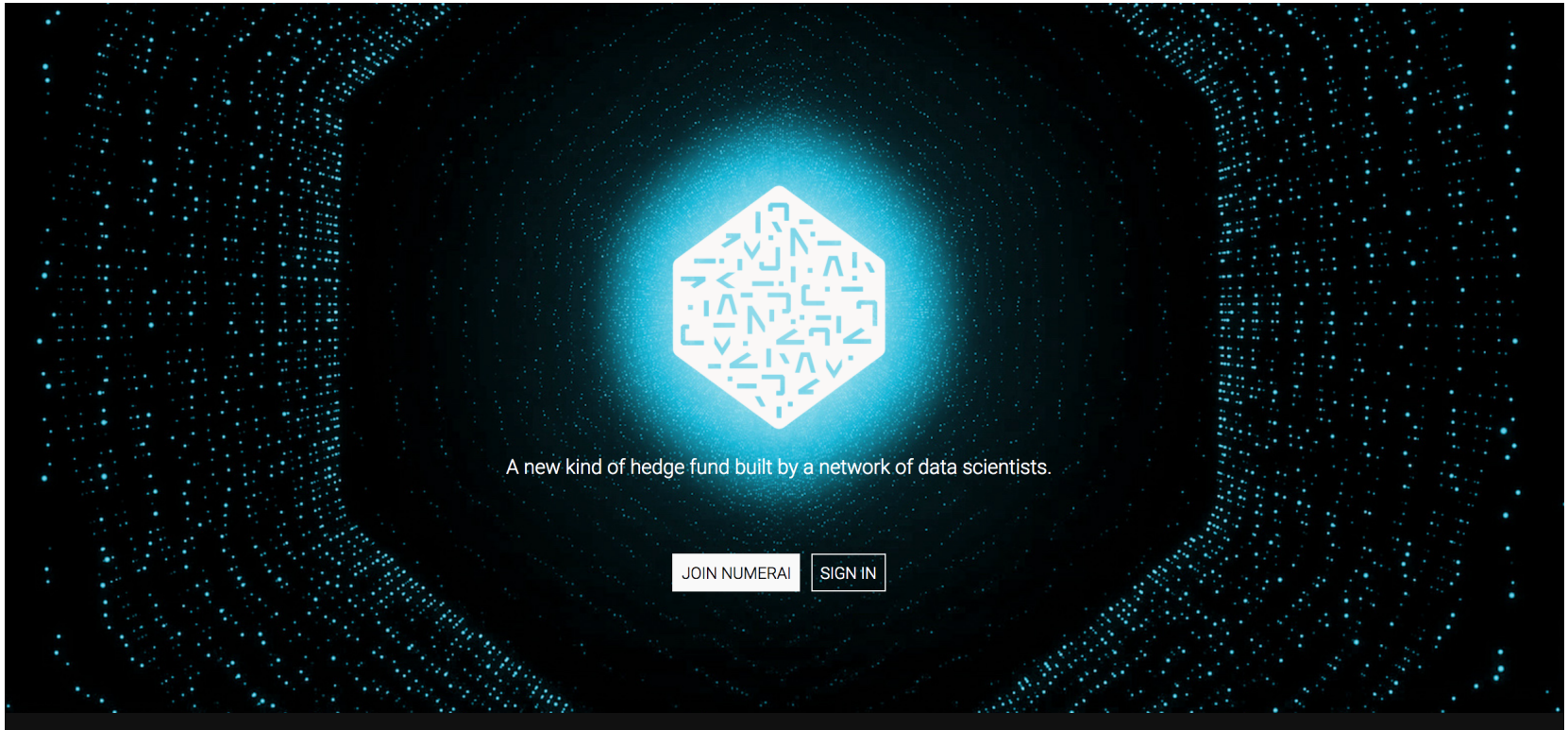
In [19]:
```
stats = CheckAccuracy(y_test, y_test_hat)
```

```
Mean Squared Error:        5.1224296628746105e-05
Root Mean Squared Error:   0.007157115105176534
Mean Absolute Error:       0.0052494561647393806
Mean Percent Error:        0.026793270172516082
```





Stock Market Efficiency



The stock market is inefficient with respect to new developments in machine learning — only a fraction of the world's data scientists have access to its data. Numerai is changing this.

Predicting the Direction of the S&P500 Index

Using daily data from 1963

In [20]:
```
# Predicting the Direction of the S&P500 Index

#Using daily data from 1963

## Read in the data into a dataframe
df = pd.read_csv("/Users/srdas/GoogleDrive/Papers/DeepLearning/DLinFinance/SP_Data
_shared/mer_df_percentile.csv")
df = df.drop(df.columns[[0]],axis=1)
print(shape(df))
#print(df.columns)
#df.head()

## Add a column for the sign of the SPX return: 1 is positive, 0 is negative
df["Sign"] = maximum(0,sign(df["SPX"]))
#df.head(20)
```

```
(13532, 21)
```

S&P 500 Returns

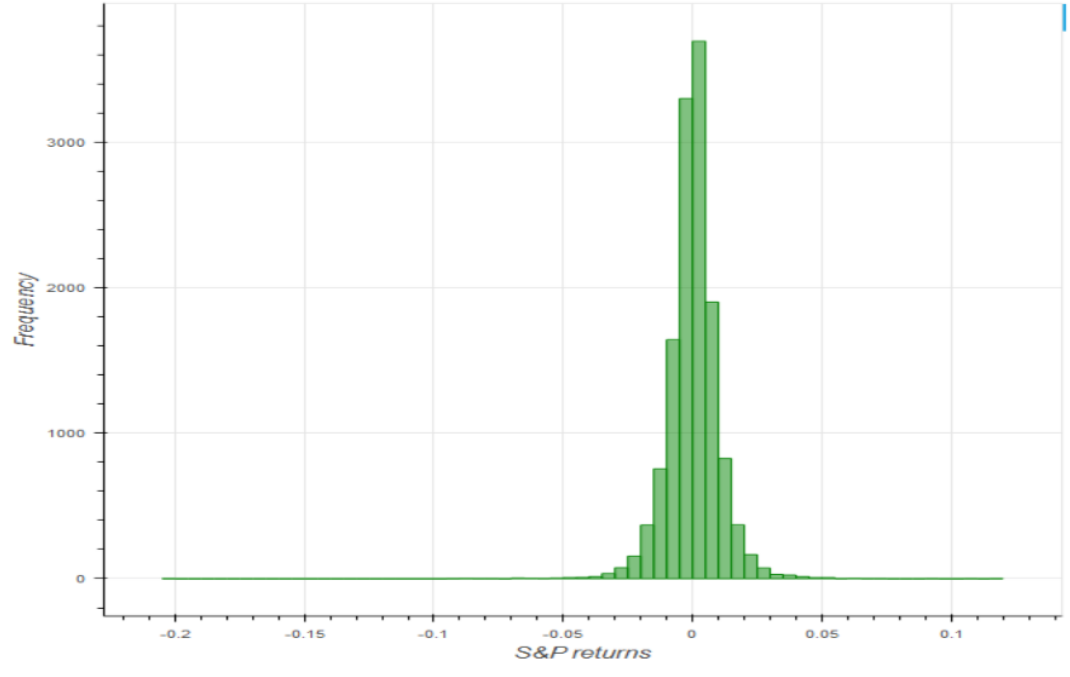


Figure 2: The distribution of daily S&P 500 index returns from 1963-2016. The mean return is 0.00031 and the standard deviation is 0.01. The skewness and kurtosis are -0.62 and 20.68, respectively.

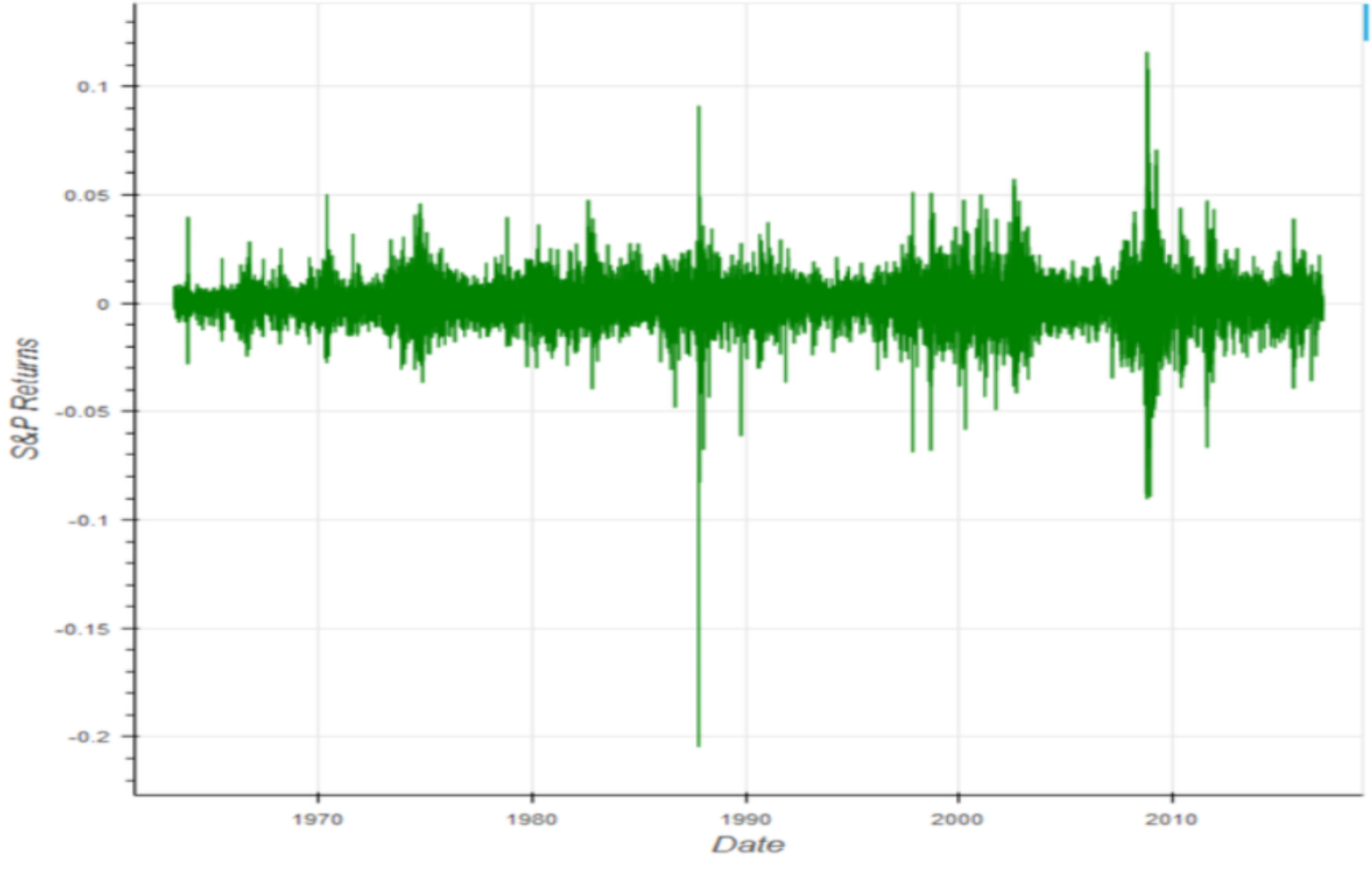


Figure 3: Daily S&P 500 index returns from 1963-2016.

Data, lookback, lookforward

In [29]:
```
n = 30                 #Lookback period
nfwd = 30              #Look ahead forecast period for validation
train_size = 10000     #Training size

m = shape(df)[0]
print(m) #Number of rows

df2 = df[n:m][df.columns[[0,1,21]]]
print(shape(df2))
df2.head()
```

```
13532
(13502, 3)
```

Out[29]:

	date	SPX	Sign
30	1963-05-14	-0.003831	0.0
31	1963-05-15	0.003133	1.0
32	1963-05-16	-0.002556	0.0
33	1963-05-17	0.000569	1.0
34	1963-05-20	-0.004695	0.0

In [30]:
```
#Special processing
## Reset the index so than pandas does not merge on index and just cbinds
df2 = df2.reset_index(drop=True)
#df2.head()

## Add n lagged columns of data, ncols = 3 + 19*n
for j in (arange(n)+1):
    tempdf = df[(n-j):(m-j)][df.columns[arange(2,21)]]
    tempdf = tempdf.reset_index(drop=True)
    df2 = pd.concat([df2,tempdf],axis=1)
print(shape(df2))
#df2.head()

df2.describe()
```

```
(13502, 573)
```

Out[30]:

	SPX	Sign	percentile_1	percentile_2	perce
count	13502.000000	13502.000000	13502.000000	13502.000000	13502.0
mean	0.000308	0.527033	0.002375	0.003322	0.0009
std	0.010157	0.499287	0.034434	0.037810	0.0396
min	-0.204669	0.000000	-0.193878	-0.126913	-0.2339
25%	-0.004395	0.000000	-0.017857	-0.022085	-0.0250
50%	0.000416	1.000000	0.000000	0.000000	0.0000
75%	0.005108	1.000000	0.019655	0.024072	0.0235
max	0.115800	1.000000	0.259489	0.166667	0.3194

8 rows × 572 columns

Select a subsample period at random and prepare data

We select a date, and then train the model on the date for a consecutive period of days given by *train_size*. Testing is done on the period of days immediately following the end of the training period for *nfwd* days.

In [31]:
```
import random
x = random.choice(range(n,m-train_size-nfwd))
print(x)   #Print the start observation
```

```
2995
```

Organize Data and Fit Model

In [32]:
```
## Fit the model to the first 10,000 rows of data
X_train = df2[x:x+train_size].as_matrix()
X_train = X_train[:,3:len(df2.columns)]
print(shape(X_train))
Y_train = df2[x:x+train_size].as_matrix()
Y_train = Y_train[:,2:3].astype(int32)
print(shape(Y_train))

data_dim = shape(X_train)[1]
print(data_dim)

X_test = df2[x+train_size:(x+train_size+nfwd)].as_matrix()
X_test = X_test[:,3:len(df2.columns)]
print(shape(X_test))
Y_test = df2[x+train_size:(x+train_size+nfwd)].as_matrix()
Y_test = Y_test[:,2:3].astype(int32)
print(shape(Y_test))
```

```
/Users/srdas/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:2: Fu
tureWarning: Method .as_matrix will be removed in a future version. Use .value
s instead.


(10000, 570)

/Users/srdas/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:5: Fu
tureWarning: Method .as_matrix will be removed in a future version. Use .value
s instead.
  """

(10000, 1)
570
(30, 570)
(30, 1)

/Users/srdas/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:12:
FutureWarning: Method .as_matrix will be removed in a future version. Use .va
```

In [33]:
```
from keras.models import Sequential
from keras.layers import Dense, Activation, Dropout
from keras.layers.advanced_activations import LeakyReLU
from keras.utils import to_categorical

Y_train2 = to_categorical(Y_train,2)
```

In [34]:
```
model = Sequential()
n_units = 64 #200

model.add(Dense(n_units, input_dim=data_dim))
model.add(Activation('relu'))
model.add(Dropout(0.25))

model.add(Dense(n_units))
model.add(Activation('relu'))
model.add(Dropout(0.25))

model.add(Dense(n_units))
model.add(Activation('relu'))
model.add(Dropout(0.25))

model.add(Dense(2))
model.add(Activation('sigmoid'))

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])
bsize = 32
model.fit(X_train, Y_train2, batch_size=bsize, epochs=25, validation_split=0.1, ve
rbose=2)
```

```
Train on 9000 samples, validate on 1000 samples
Epoch 1/25
 - 4s - loss: 0.6930 - acc: 0.5180 - val_loss: 0.6877 - val_acc: 0.5580
Epoch 2/25
 - 3s - loss: 0.6926 - acc: 0.5244 - val_loss: 0.6885 - val_acc: 0.5580
Epoch 3/25
 - 3s - loss: 0.6919 - acc: 0.5217 - val_loss: 0.6889 - val_acc: 0.5585
Epoch 4/25
 - 3s - loss: 0.6924 - acc: 0.5257 - val_loss: 0.6896 - val_acc: 0.5590
Epoch 5/25
 - 3s - loss: 0.6923 - acc: 0.5257 - val_loss: 0.6883 - val_acc: 0.5580
```

```
Epoch 6/25
 - 3s - loss: 0.6920 - acc: 0.5259 - val_loss: 0.6884 - val_acc: 0.5580
Epoch 7/25
 - 3s - loss: 0.6920 - acc: 0.5285 - val_loss: 0.6884 - val_acc: 0.5455
Epoch 8/25
 - 3s - loss: 0.6914 - acc: 0.5249 - val_loss: 0.6880 - val_acc: 0.5580
Epoch 9/25
 - 3s - loss: 0.6917 - acc: 0.5265 - val_loss: 0.6887 - val_acc: 0.5430
Epoch 10/25
 - 3s - loss: 0.6912 - acc: 0.5275 - val_loss: 0.6905 - val_acc: 0.5425
Epoch 11/25
 - 3s - loss: 0.6920 - acc: 0.5287 - val_loss: 0.6904 - val_acc: 0.5460
Epoch 12/25
 - 3s - loss: 0.6914 - acc: 0.5276 - val_loss: 0.6913 - val_acc: 0.5360
Epoch 13/25
 - 4s - loss: 0.6913 - acc: 0.5277 - val_loss: 0.6900 - val_acc: 0.5580
Epoch 14/25
 - 3s - loss: 0.6911 - acc: 0.5264 - val_loss: 0.6893 - val_acc: 0.5485
Epoch 15/25
 - 3s - loss: 0.6907 - acc: 0.5305 - val_loss: 0.6926 - val_acc: 0.5380
Epoch 16/25
 - 3s - loss: 0.6905 - acc: 0.5328 - val_loss: 0.6883 - val_acc: 0.5485
Epoch 17/25
 - 3s - loss: 0.6910 - acc: 0.5309 - val_loss: 0.6936 - val_acc: 0.5010
Epoch 18/25
 - 3s - loss: 0.6904 - acc: 0.5320 - val_loss: 0.6880 - val_acc: 0.5585
Epoch 19/25
 - 3s - loss: 0.6897 - acc: 0.5328 - val_loss: 0.6924 - val_acc: 0.5265
Epoch 20/25
 - 3s - loss: 0.6895 - acc: 0.5343 - val_loss: 0.6914 - val_acc: 0.5360
Epoch 21/25
 - 3s - loss: 0.6888 - acc: 0.5419 - val_loss: 0.6941 - val_acc: 0.5140
Epoch 22/25
 - 3s - loss: 0.6880 - acc: 0.5413 - val_loss: 0.6914 - val_acc: 0.5495
Epoch 23/25
 - 3s - loss: 0.6888 - acc: 0.5403 - val_loss: 0.6891 - val_acc: 0.5420
Epoch 24/25
 - 3s - loss: 0.6873 - acc: 0.5465 - val_loss: 0.6926 - val_acc: 0.5455
```

```
Epoch 25/25
 - 3s - loss: 0.6872 - acc: 0.5421 - val_loss: 0.6891 - val_acc: 0.5500
```

Out[34]: <keras.callbacks.History at 0x1a7d03aeb8>

Model Accuracy

$$parameters = 570 \times 65 + 64 \times 65 + 64 \times 65 + 65 = 45435$$

In [35]:
```
model.evaluate(X_train, Y_train2, verbose=0)
```

Out[35]:
```
[0.6846257612228394, 0.5495]
```

THE END