Week 2

Components of Neural Networks

Activation Functions

Activation Functions

1. Artificial neuron works in three steps
   1. First it multiplies the input signals with corresponding weights
   2. Second, adds the weighted signals
   3. Third, converts the result to another value using a mathematical transformation function

2. For the third step, there are multiple mathematical functions available but all together are called the activation function

3. The purpose of the activation function is to act like a switch for the neuron. Should the neuron fire or not.

4. The activation function is critical to the overall functioning of the neural network. Without it, the whole neural network will mathematically become equivalent to one single neuron!

5. Activation function is one of the critical component that gives the neural networks ability to deal with complex problems

Activation Functions -why?

1. Let us take a fully connected neural network. Every neuron in every layer takes multiple inputs
2. The inputs are weighted and summed up at each neuron
3. The nodes in the second layer are simply scaling up the output of neurons in previous layer
4. For e.g the neuron G takes as input weight sums from D,E and F, G's output is scaled version of output of D, E and F
5. G_Out = 3D -2E + 1F
   = 3(1A + 2B + 3C) – 2(-3A + 2B -1C) + 1(2A+4B-2C)
   = 11A + 6B + 9C

6. Thus this part of the network is like a single neuron with weights of 8, 6, 3
7. Same argument holds for other neurons
8. Thus the entire neural network collapses to on neuron!
9. A single neuron is not capable of doing much
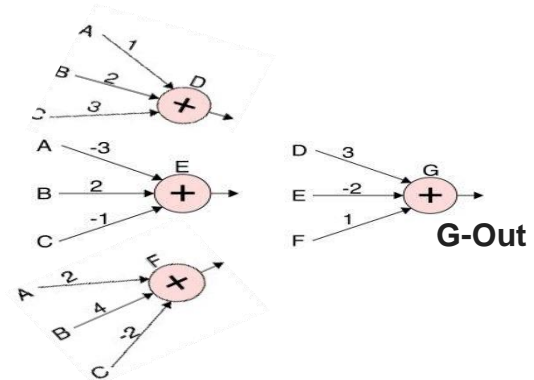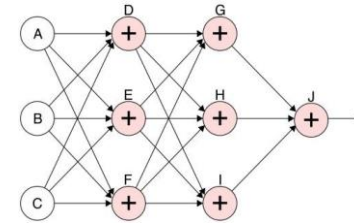
Activation Functions - What are they?

1. The activation functions are a mathematical transformations that prevent the network from collapsing to a single neuron

2. The collapse can happen when the neurons do simple addition and multiplications of the inputs. These are called linear operations. Thus linear operations collapse the network

3. All activations functions are nonlinear transformers for exactly the same reason. This non-linear transformation not only prevents collapse, it also empowers the network to do complex tasks because each neuron does something in the network totality

Activation Functions - Types

4. Types of non-linear activation functions include –
   a. **Piecewise linear functions**
      I. Step function
      II. ReLU – Rectified Linear Units
      III. Leaky ReLU
      IV. Parametric ReLU
      V. Shifted ReLU

   b. **Smooth functions**
      I. Smooth ReLU / Exponential ReLU
      II. Sigmoid / Logisticfunctions
      III. Hyperbolic Tangent (tanh)
      IV. Swish (combination of Sigmoid and ReLU)

Activation Functions -

Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for} \quad x < 0 \\ 1 & \text{for} \quad x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for} \quad x \neq 0 \\ ? & \text{for} \quad x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \dfrac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
TanH		$f(x) = \tanh(x) = \dfrac{2}{1 + e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \dfrac{1}{x^2 + 1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for} \quad x < 0 \\ x & \text{for} \quad x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for} \quad x < 0 \\ 1 & \text{for} \quad x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU)		$f(x) = \begin{cases} \alpha x & \text{for} \quad x < 0 \\ x & \text{for} \quad x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for} \quad x < 0 \\ 1 & \text{for} \quad x \geq 0 \end{cases}$
Exponential Linear Unit (ELU)		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for} \quad x < 0 \\ x & \text{for} \quad x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for} \quad x < 0 \\ 1 & \text{for} \quad x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \dfrac{1}{1 + e^{-x}}$

Cover's theorem for non-linear functions

Neuron1

$ACC = m1X + C1$

N1Output = Sigmoid(ACC)

X

Input distribution

Neuron2

$ACC = m2X + C2$

N2Output = Sigmoid(ACC)

Transformed thru sigmoid

X

N1

N2

N3

Data points get linearly separable

Input distribution

Transformed through sigmoid
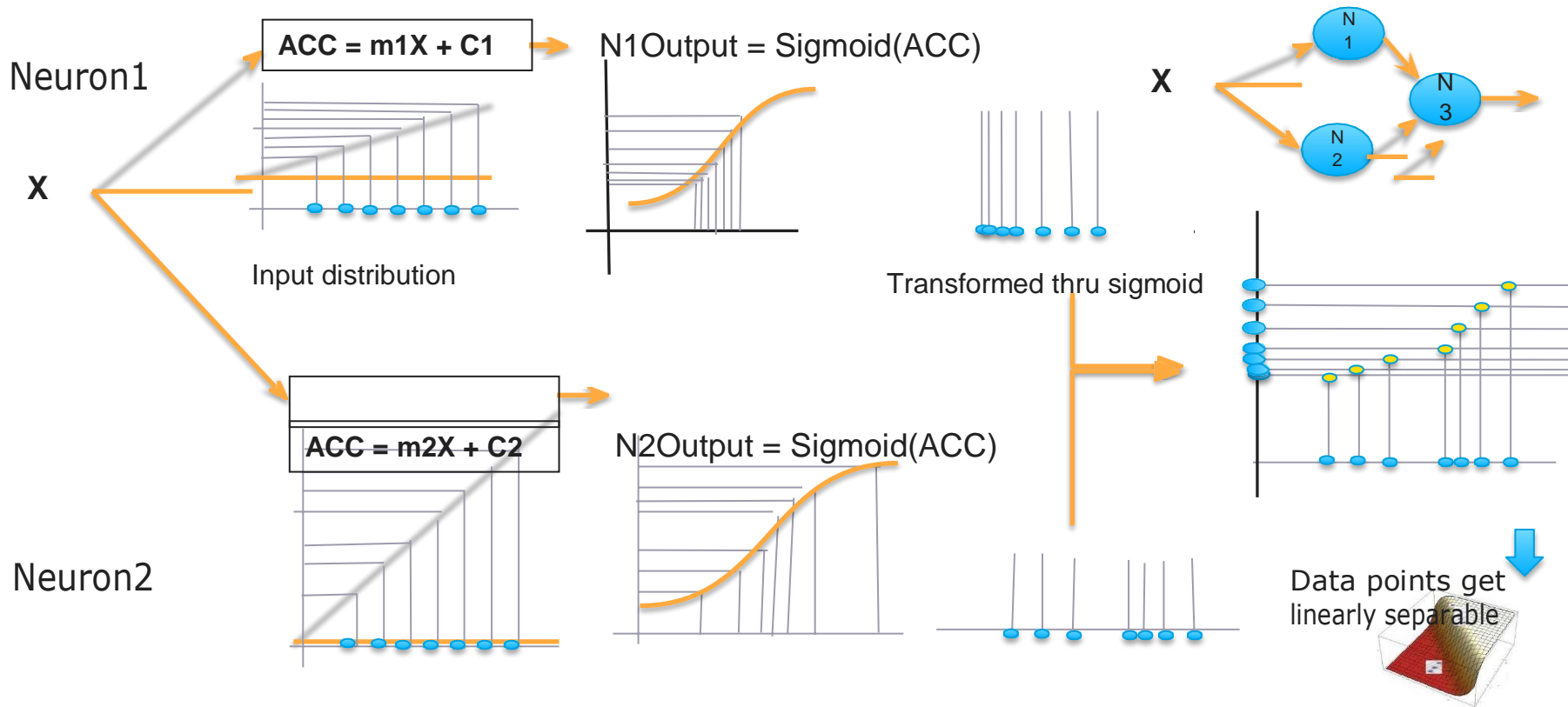
Cover's theorem (Continued)
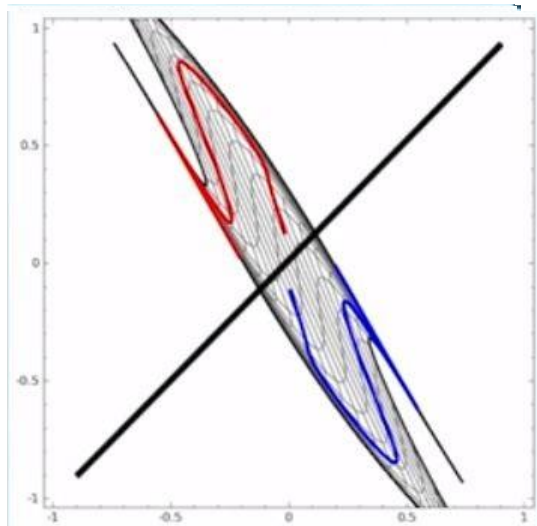
Neurons stretch the features pace through non-linear functions and achieve Cover's theorem
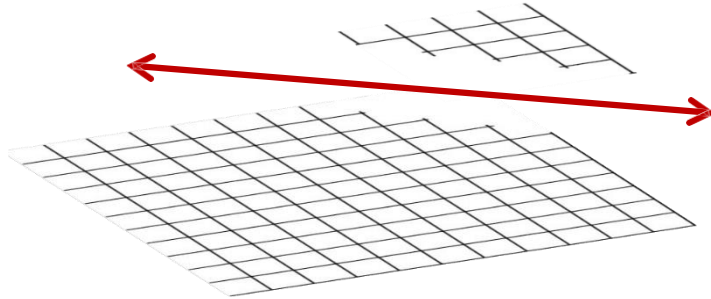
Image Source: https://colah.github.io/posts/2014-03-NN-Manifolds-Topology/

Ref: https://cs.stanford.edu/people/karpathy/convnetjs//demo/classify2d.html

Step Function

Step function divides the input space into two halves ⬜ 0 and 1
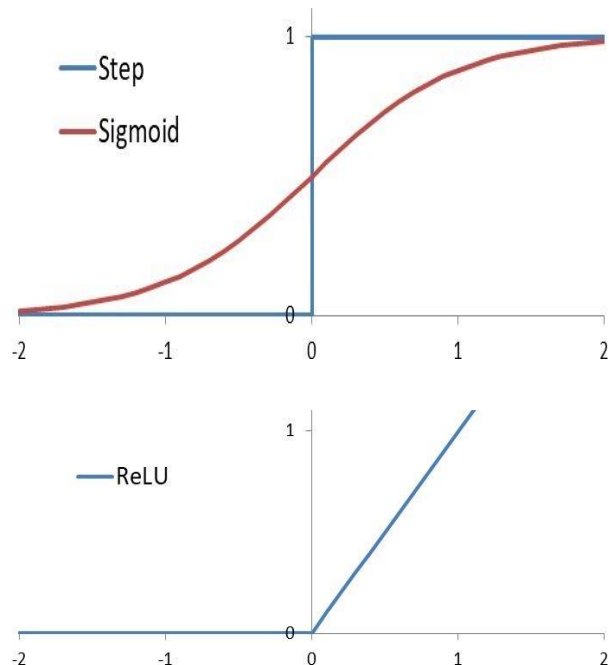
- In a single neuron, step function is a linear binary classifier
- The weights and biases determine where the step will be in n-dimensions
- But, as we shall see later, it gives little information about how to change the weights if we make a mistake
- So, we need a smoother version of a step function
- Enter: the Sigmoid function

Problems with Activation Functions

- For both large positive and negative input values, sigmoid doesn't change much with change of input. The problem with sigmoid is (near) zero gradient on both extremes

-

- ReLU has a constant gradient for almost half of the inputs

- But, ReLU cannot give a meaningful final output

Output activation functions
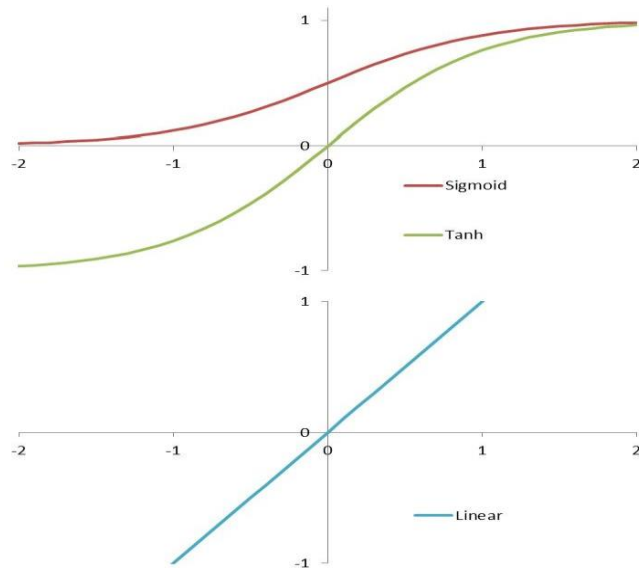
- **Sigmoid gives binary classification output**
- **Tanh can also do that provided the desired output is in {-1, +1}**
- **Softmax generalizes sigmoid to n-ary classification**
- **Linear is used for regression**
- **ReLU is only used in internal nodes (non- output)**

RELU VS Simple linear activation function

1. Half range of RELU (default RELU) is a simple linear transformation of the form y = mx (as long as mx + c > 0), for mx + c <=0 y = 0

2. If half of the input is transformed using linear model, then why does not RELU network collapse to Perceptron? To understand this let us look at the dummy network below which has no nonlinear transformation –
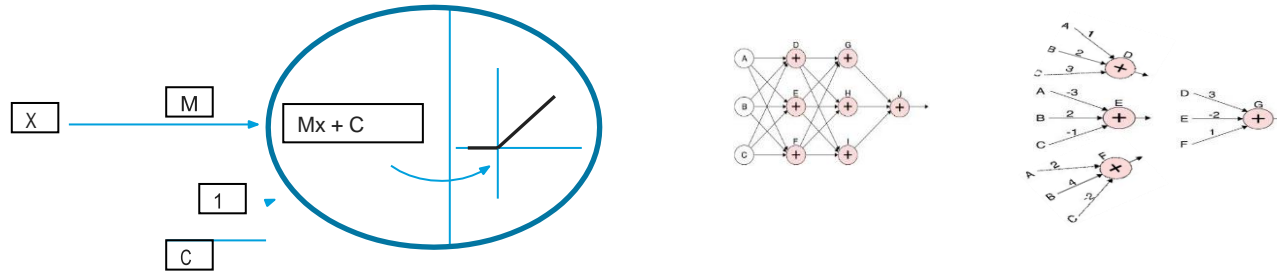
3. When we replace all the neurons in the network with the neuron shown above with RELU, Gout will be 0 if 3D -2E + 1F < = 0 and 3D - 2E + 1F will be same as Gout only when 3D -2E + 1F is >0.

4. Thus, Gout cannot be generalized as 3D-2E+1F. Same goes for D0, E0 and F0. Hence the neuron does not become an equivalent of a perceptron

Softmax Activation Function

SoftMax Function -

1. A kind of operation applied at the output neurons of a classifier network
2. Used only when we have two or more output neurons and is applied simultaneously to all the output neurons
3. Turns raw numbers coming out of the pen-ultimate layer into probability values in the output layer
4. Suppose output layer neurons emit (Op1, Op2, Opn). The raw numbers may not make much sense. We convert that into probabilities using Softmax which becomes more meaningful. For e.g. input belongs to cycle is 30 times more likely than sailboat, 13 times more probable than car.
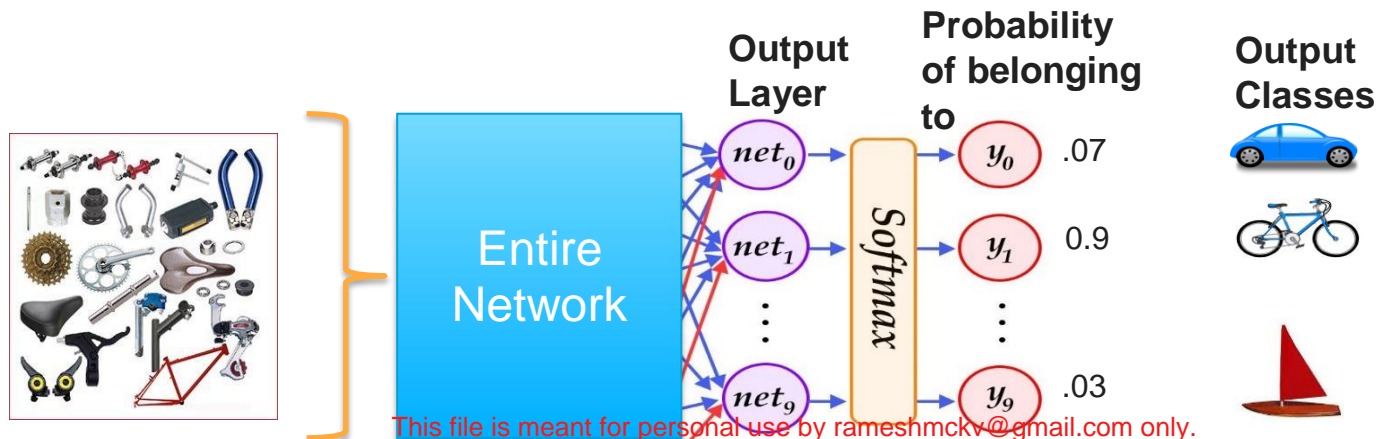
Output Layer

Probability of belonging to

Output Classes

Entire Network

Softmax

net_0 → y_0 .07

net_1 → y_1 0.9

net_9 → y_9 .03

Forward Propagation and Bias

Forward Propagation

1. The directed acyclic path taken by input data from input layer to get transformed using non-linear functions into final network level outputs

2. Input data is propagated forward from the input layer to hidden layer till it reaches final layer where predictions are emitted

3. At every layer, data gets transformed non-linearly in every neuron

4. There may be multiple hidden layers with multiple neurons in each layer

5. The last layer is the output layer which may have a softmax function (if the network is a multi class classifier)

Forward Propagation and Matrix operations
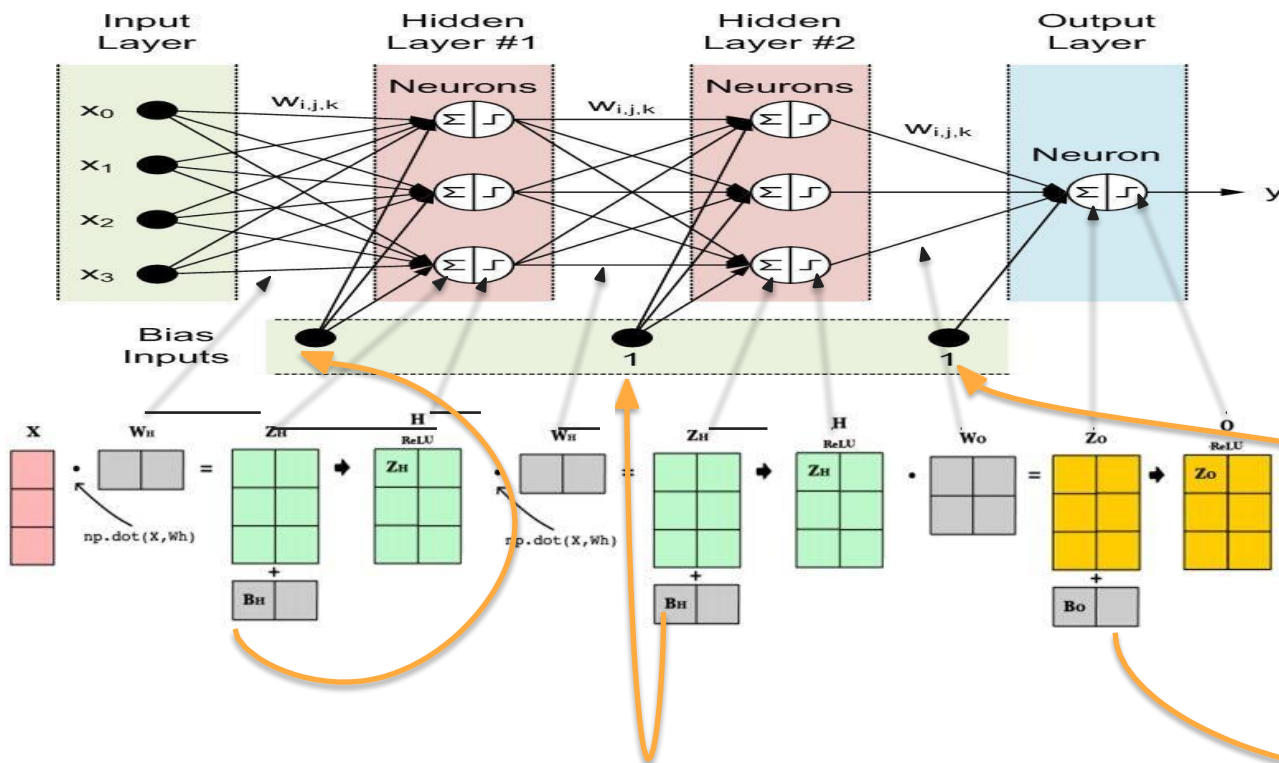
Bias Term

1. **Every neuron in the hidden layers are associated with a bias term. The bias term help us to control the firing threshold in each neuron**
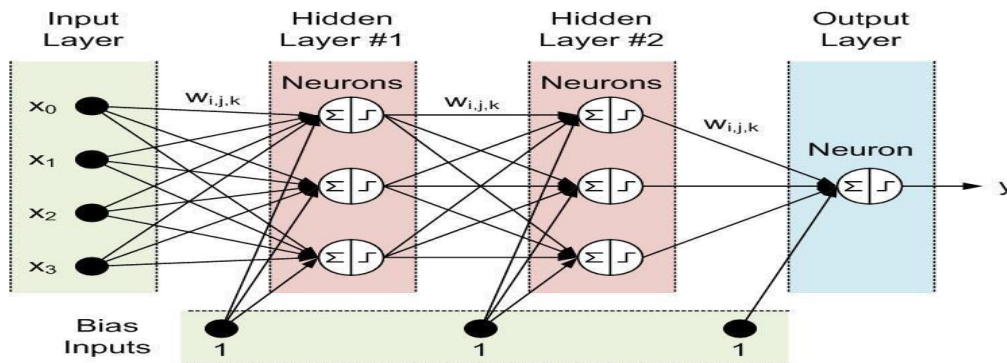
2. **It acts like the intercept in a linear equation (y = sum(mx) + c). If sum(mx) is not crossing the threshold but the neuron needs to fire**

3. **bias will be adjusted to lower that neuron's threshold to make it fire! Network learns richer set of patterns using bias**

4. **The bias term is also considered as input though it does not come from data**

Forward Propagation

6. **Forward prop steps–**

    a.   **Calculate the weighted input to the hidden layer by multiplying ⬚ by the hidden weight ⬚h**

    b.   **Apply the activation function and pass the result to the final layer**

    c.   **At output layer, repeat step b replacing ⬚ by the hidden layer's output**

Loss Function

Interpreting Bad Model using Loss Function

- **Loss trends opposite of accuracy**
    - **Loss is low when accuracy is high**
    - **Loss is zero for perfect accuracy (by convention)**
    - **Loss is high when accuracy is low**

- **Loss is a function of actual and desired output**

- **Minimizing the loss function with respect to parameters leads to good parameters**

Properties of a good loss function

- **Minimum value for perfect accuracy**
  - **Usually zero**
  - *Note:* **low loss on training does not guarantee low loss on validation or testing**
- **Varies smoothly with input**
- **Varies smoothly with parameters**
- **Good to be convex in parameters (but is usually not)**
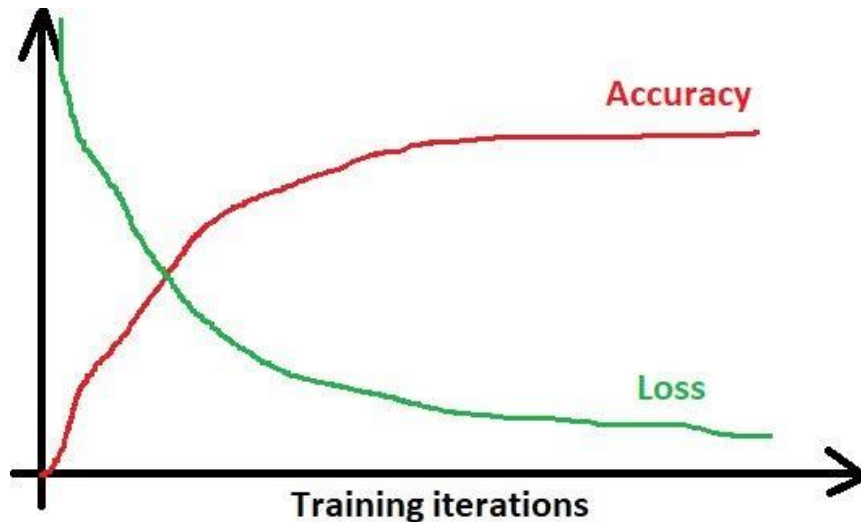 - **Like a paraboloid**

Loss and accuracy

- **Training accuracy saturates to a maximum**
- **Training loss saturates to a minimum**
- **Loss is a measure of error**
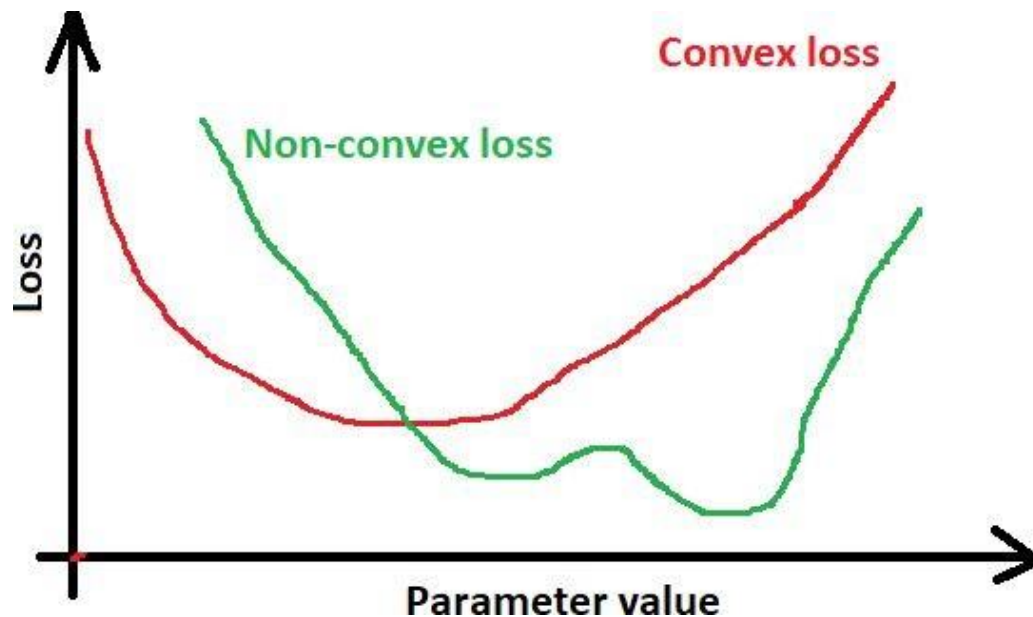
Convex vs. non-convex loss

Non-convex loss can have multiple minima
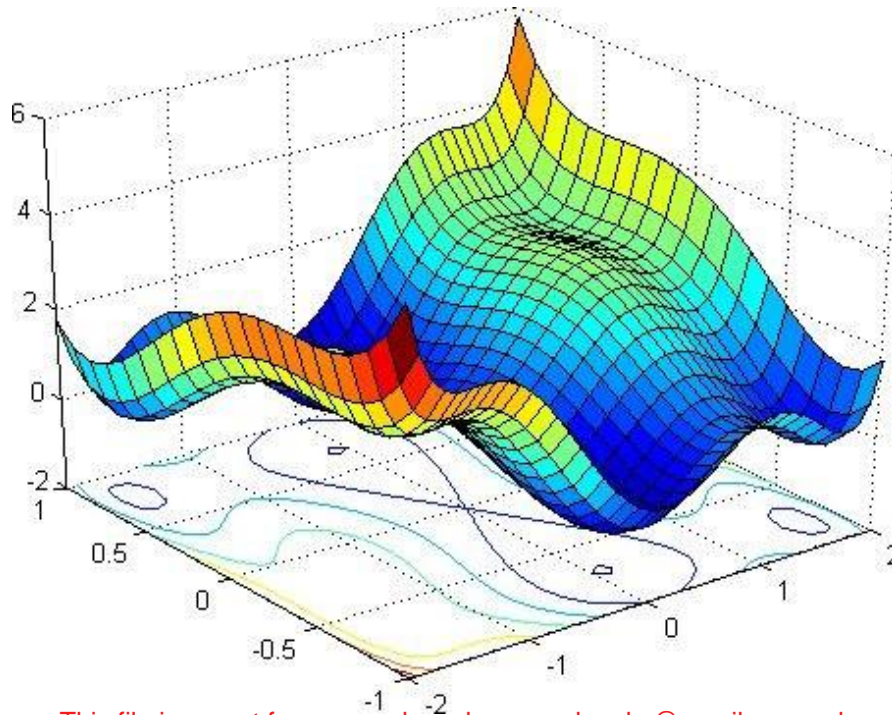
More about loss function

- **Choice of loss function depends on:**
  - ○ **Desired output type: continuous or categorical?**
  - ○ **Predicted output type: continuous or categorical?**
  - ○ **Goal: supervised or unsupervised?**

- **Loss function over a set is the average of loss over each sample in the set.**

- **Loss function over the validation set is the most important thing to monitor during training.**

- **High training loss means under-fitting.**

- **Large gap between training and validation losses means over-fitting**

Examples of loss functions

- **Regression with continuous output**
  - Mean square error (MSE), log MSE, mean absolute error

- **Classification with probabilistic output**
  - Cross entropy (negative log likelihood), hinge loss

- **Similarity between vectors or clustering**
  - Euclidean distance, cosine

Choice of Loss function

MSE
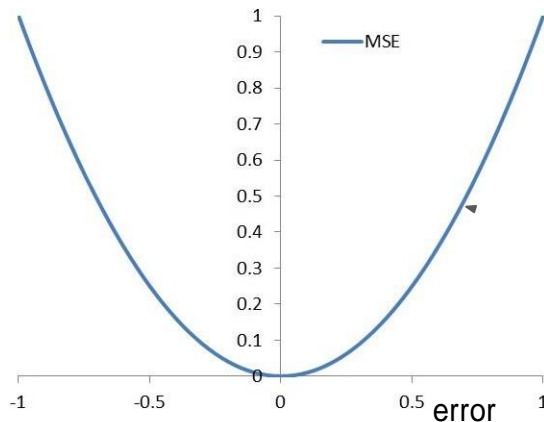
error →

- There are positive and negative errors in classification and MSE is the most common loss function

    MSE
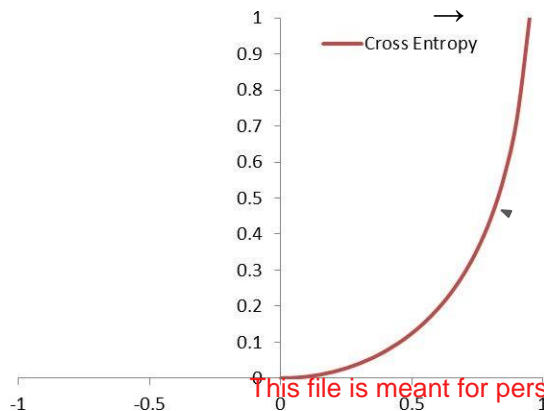
Cross Entropy

- There is a probability of correct class in classification, for which cross entropy is the most common loss function

    Cross Entropy

MSE loss for regression
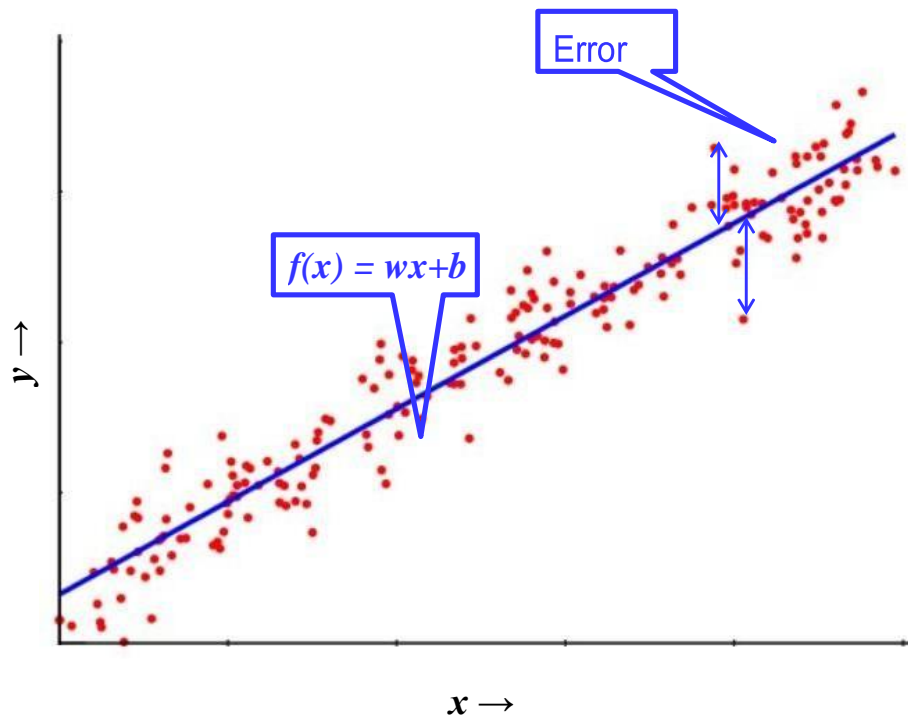
$$\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} (Y_i - \hat{Y}_i)^2$$

MSE = mean squared error

n = number of data points

Y_i = observed values

$\hat{Y}_i$ = predicted values

Advantage of MAE
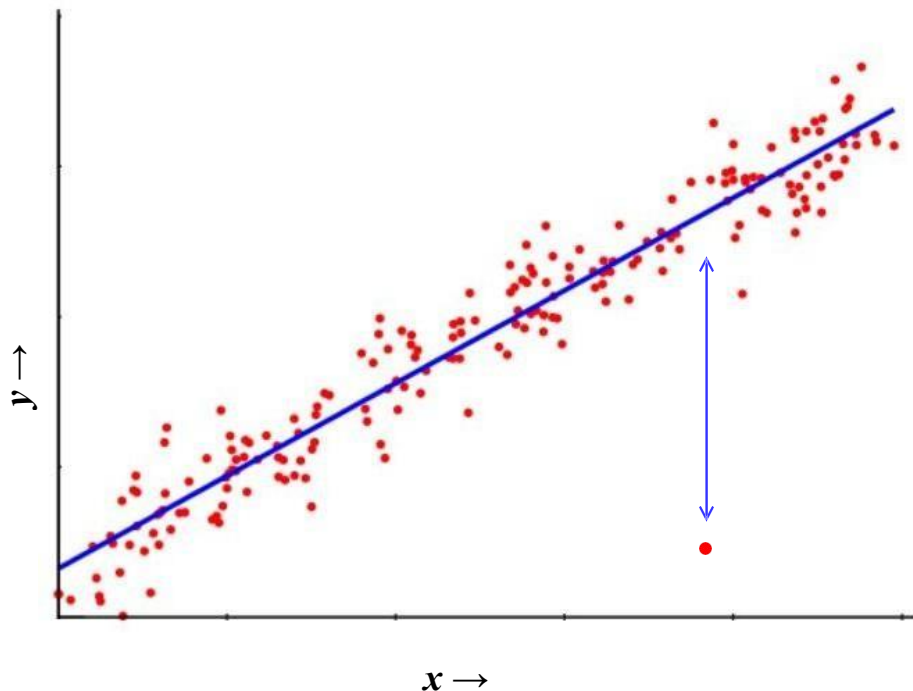
$$\text{MAE} = \frac{\sum_{i=1}^{n} |y_i - x_i|}{n}$$

MAE = mean absolute error
y_i = prediction
x_i = true value
n = total number of data points

MAE loss is less affected by outliers than MSE

Is MSE appropriate for classification?
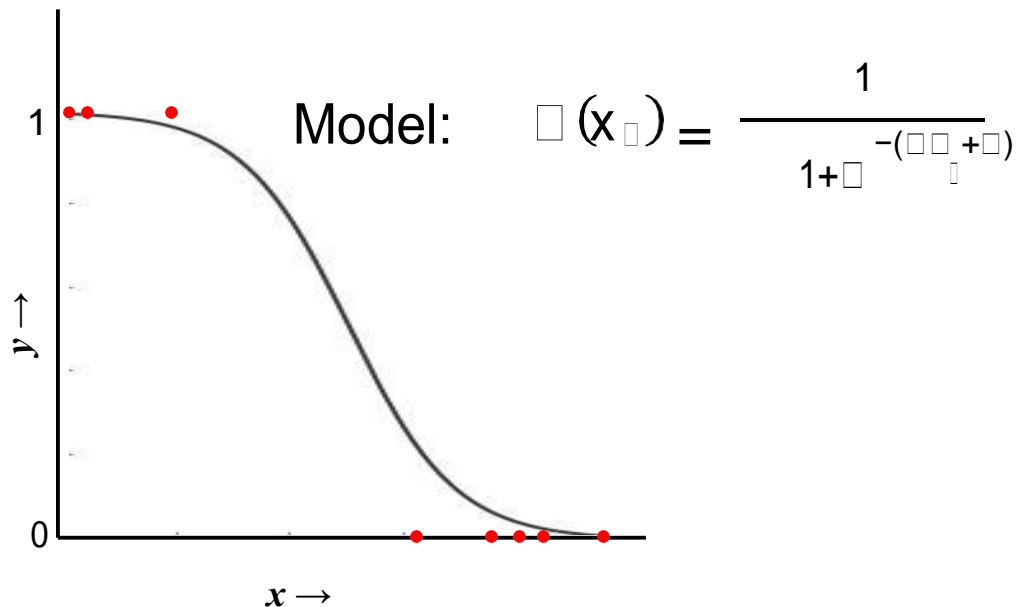
Model: $\sigma(x_i) = \dfrac{1}{1+e^{-(\beta x_i + \alpha)}}$

Some loss functions
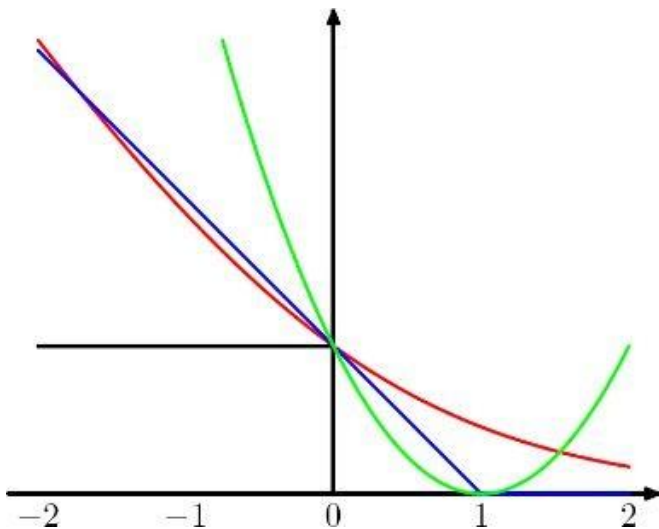
- **Problem: binary classification**

- **Assumption: desired output is 1**

- **Notice rate of convergence at different points**

Types of Loss Function

(Classification)

Loss Function for Classification

- **Categorical Cross Entropy**

- **Sparse categorical Cross Entropy**

- **Binary Cross Entropy**

Loss Function for Classification (Continued)

- **Categorical Cross Entropy**
  - It is a loss function used in multi-class classification and the target labels are always one-hot encoded.
  - It is well suited for classification tasks so that one example can belong to one class with probability 1 and to the other with probability 0.
  - For example, In MNIST digit recognition, the model gives higher probability prediction for the correct digit and lower probabilities for the other digits.

- **Sparse Categorical cross entropy**
  - It is a loss function similar to categorical cross entropy but here the labels are provided as integers rather than being one-hot encoded.
  - An advantage of using sparse categorical cross entropy is that it saves the memory time as it simply uses a single integer rather than a whole vector.

- **Binary Cross Entropy**
  - It is mainly used in problems related to binary classification.
  - It compares each of the probability predicted to the output class which can be either 0 or 1.

Types of Loss Function

(Regression)

Loss Function for Regression

- **Mean Squared Error (MSE)**

- **Root Mean Squared Error (RMSE)**

- **Mean Absolute Error (MAE)**

- **Huber Loss**

Loss Function for Regression (Continued)

- **Mean Squared Error (MSE)**
  - It is calculated as the average of the squared differences between actual and predicted values.
  - The result is always positive and the case with no error will always have MSE=0
  - As it is sensitive to outliers so it is used so MSE will be great for ensuring our trained models has no outlier predictions with huge errors as it implies larger weights on outliers due to the square function

- **Root Mean Squared Error (RMSE)**
  - It is similar to MSE with a root function over it.
  - Here as the errors are squared before the average is taken so RMSE always gives a high weight to large errors.

Both MSE and RMSE have a range from 0 to ∞. The model is said to be good if the value of these errors are low.

- **Mean Absolute Error (MAE)**
  - It is the sum of the absolute difference between the actual and the predicted values.
  - It is robust to outliers.
  - Its range even lies between 0 and ∞.

- **Huber Loss**
  - It is a loss function used in robust regression as it is less sensitive to outliers in data than the squared error loss.
  - It is differentiable at 0. It is basically absolute error which becomes quadratic when error is small.
  - It combines the good properties of MSE and MAE to yield the best results.

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}|$$

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y})^2$$

$$RMSE = \sqrt{MSE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y})^2}$$

Back Propagation and Learning rate

Back Propagation

1. Back propagation is the process of learning that the neural network employs to re-calibrate the weights and bias at every layer and every node to minimize the error in the output layer

2. During the first pass of forward propagation, the weights and bias are random number. The random numbers are generated within a small range say 0 – 1

3. Needless to say, the output of the first iteration is almost always incorrect. The difference between actual value / class and predicted value / class is the error

4. All the nodes in all the preceding layers have contributed to the error and hence need to get their share of the error and correct their weights

5. This process of allocating proportion of the error to all the nodes in the previous layer is back propagation

6. The goal of back propagation is to adjust weights and bias in proportion to the error contribution and in iterative process identify the optimal combination of weights

7. At each layer, at each node, gradient descent algorithm is applied to adjust the weights

Back Propagation
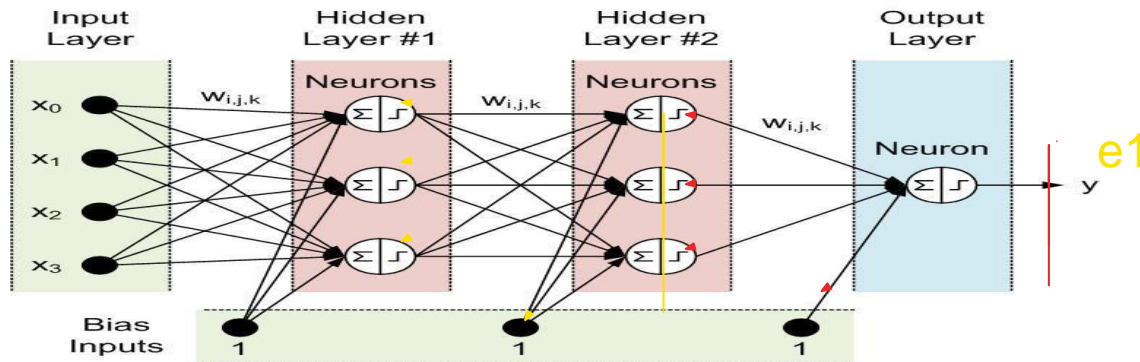
1. Error in output node shown as e1, is contributed by node 1, 2 and 3 of layer 2 through weights w(3,1), w(3,2), w(3,3)

2. Proportionate error is assigned back to node 1 of hidden layer 2 is (w(3,1) / w(3,1) + w(3,2) + w(3,3)) * e1

3. The error assigned to node 1 of hidden layer 2 is proportionately sent back to hidden layer 1 neurons

4. All the nodes in all the layers re-adjust the input weights and bias to address the assigned error (for this they use gradient descent)

5. The input layer is not neurons, they are like input parameters to a function and hence have no errors
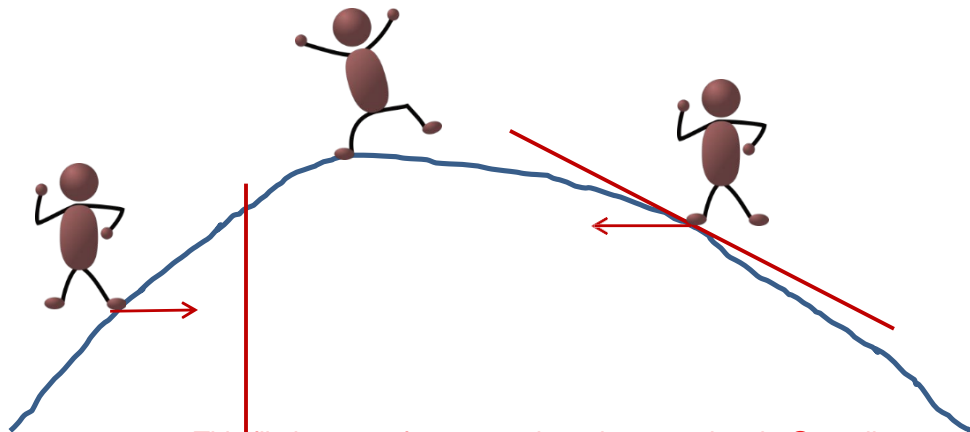
Optimization

Gradient descent

- If you didn't know the shape of a mountain

- But at every step you knew the slope

- Can you reach the top of the mountain?

Gradient descent minimizes the loss function

- **At every point, compute**
  - **Loss (scalar):** $L_w(\square)$
  - **Gradient of loss with respect to weights (vector):**
    $$\nabla_w L_w(\square)$$
  - **Take a step towards negative gradient:**
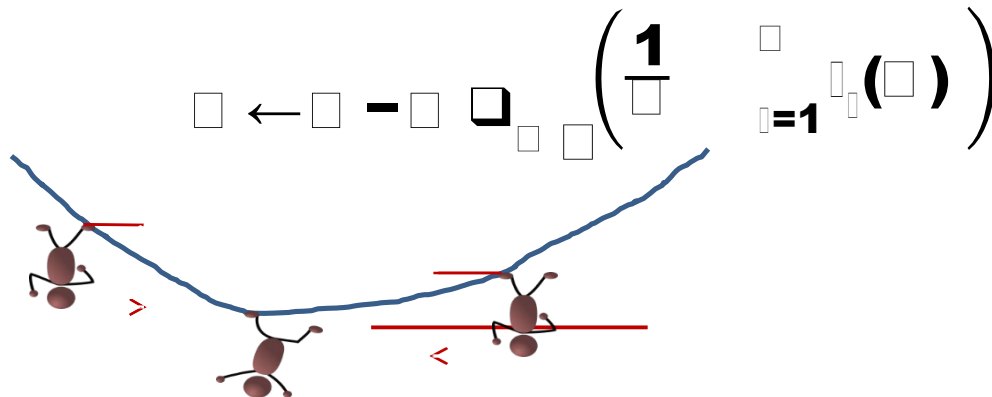
$$\square \leftarrow \square - \square \; \nabla_\square \left(\frac{1}{\square} \sum_{\square=1}^{\square} \square_\square(\square) \right)$$

Role of step size and learning rate

- **Tale of two loss functions**
  - **Same value,**
  - **Same gradient (first derivative), but**
  - **Different Hessian (second derivative)**
  - **Different step sizes needed**
- **Success not guaranteed**
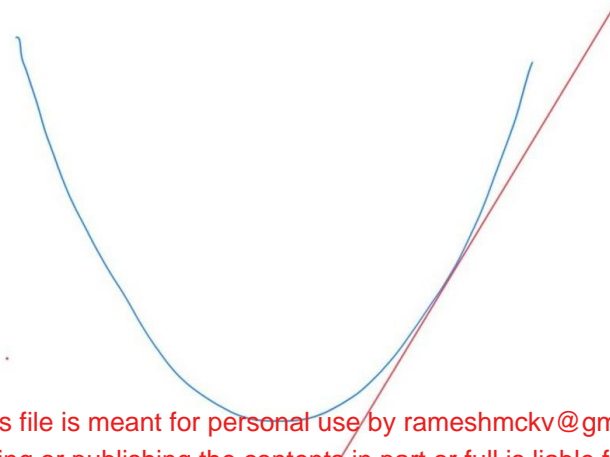
Step size (Continued)

- **The perfect step size is impossible to guess. Goldilocks finds the perfect balance only in a fairy tale**

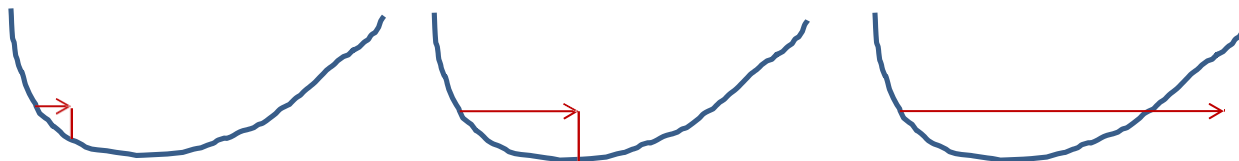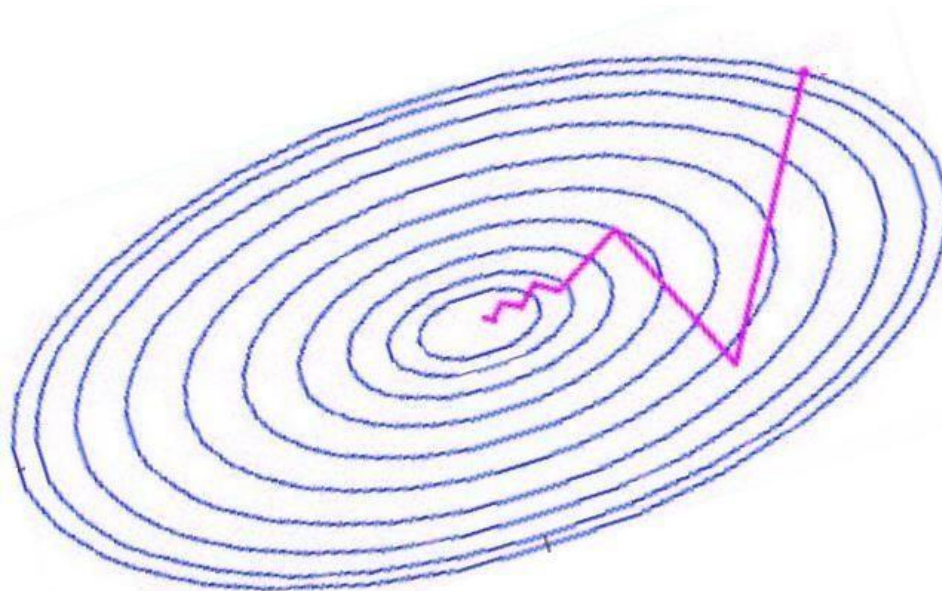- **The step size is decided by learning rate** η **and the gradient**

Unfolding step-size in multiple dimensions

Relation between error and change in weights

Since part of neuron function is linear equation (before applying the non-linear transformation), the error at each neuron can be expressed in terms of the linear equation.

$$Y = (w1 + d(w))x +$$

Actual y1 given X1

Error e1 in prediction

d(w)

w1

Predicted y1 given X1

$$Y = w1x + c$$

e1 = yellow – red
 = y_actual - y_pred
= (w1+dm)x + c - (w1x + c)
= w1x + c + dwx – w1x -c
= dwx
Hence dw = e1 / x
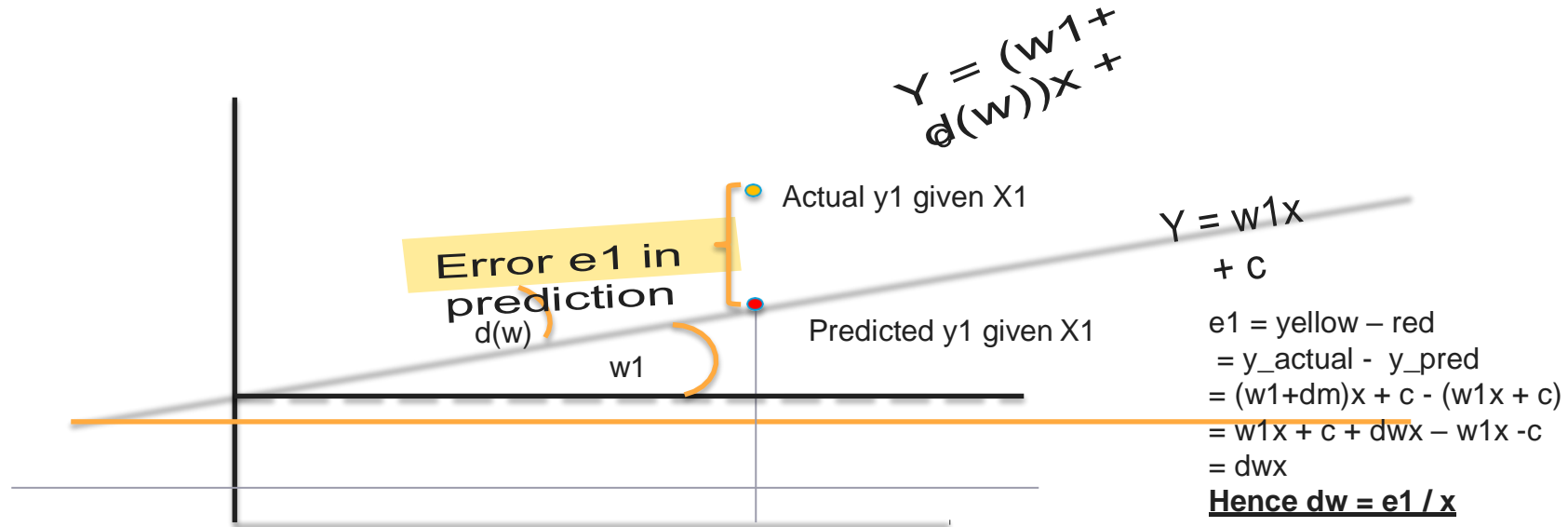
The change required in m (dw) is e1/x. However, change required w.r.t another data point may be different. To prevent jumping around with dw, we moderate the change in W by introducing a learning rate l . Hence dw = l(e1/x)

Number of Samples for each update

$$w \leftarrow w - \eta \; \nabla_w \left(\frac{1}{N} \sum_{n=1}^{N} L_n(w) \right)$$

- **Vanilla gradient descent: use samples in a fixed sequence:**

  $$w \leftarrow w - \eta \; \nabla_w L_n(w)$$

- **Stochastic gradient descent: Choose a random sample for each update. In practice, we make random mini-batches based on computational resources**
  - Divide N samples into equal sized batches
  - Update weights once per batch
  - One epoch completes when all samples used once

- **Batch gradient descent: use all samples, and update once per epoch for all samples**

Loss of different sets of samples

- Different mini-batches (or samples) have their own loss surfaces
- The loss surface of the entire training sample (dotted) may be different
- Local minima of one loss surface may not be local minima of another one
- This helps us escape local minima using stochastic or batch gradient descent
- Mini-batch size depends on computational resource utilization
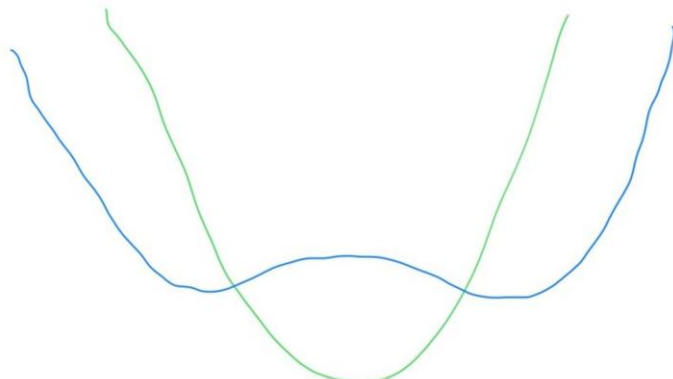
Learning rate decay

- **Initially use large learning rate: further from the global minima, we need to rapidly converge**
- **Later, reduce the learning rate: Closer to the solution we need to start fine-tuning with smaller steps**
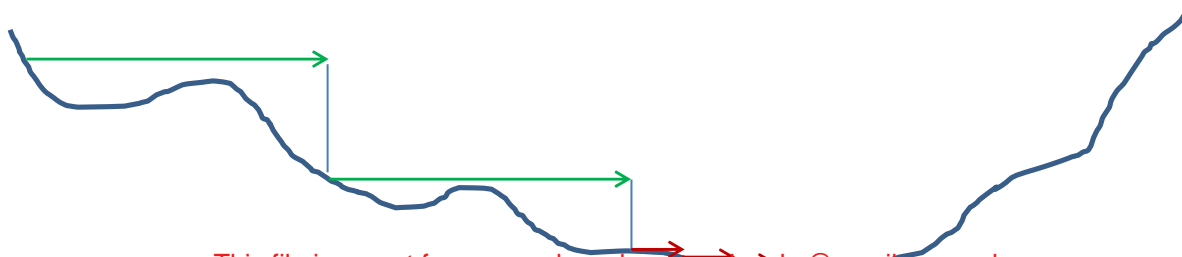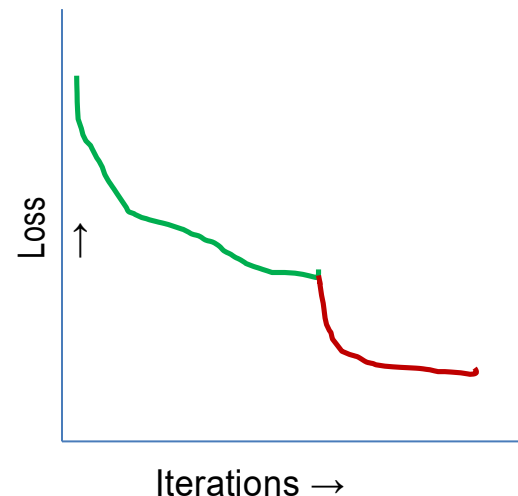
Momentum

- Momentum means using the memory of previous step to build up speed or to slow down with forgetting factor ▯; 0 ≤ ▯ < 1
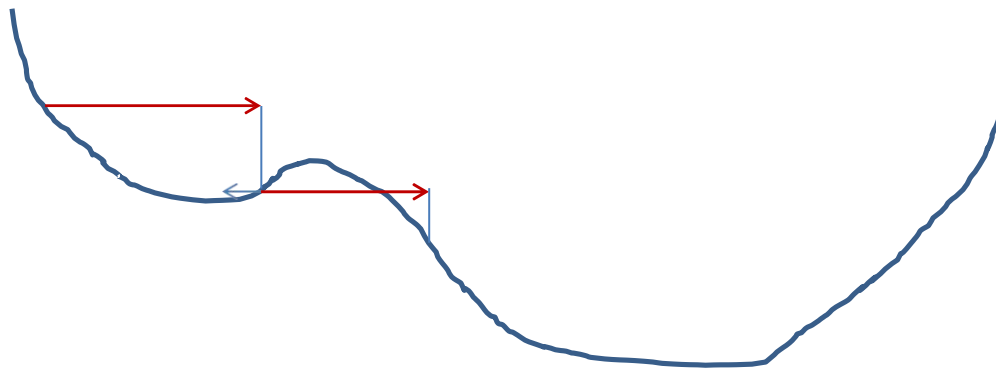
$$\Delta \square^{(\square)} = \square \; \Delta \square^{(\square-1)} - \square \; \square \quad \square(\square \; (\square),$$

Momentum

))

Gradient Descent

The challenge is, all the weights in all the inputs of all the neurons need to be adjusted. It is not manually possible to find the right combination of weights using brute force. Instead, the neural network algorithm uses a learning function called gradient descent.

Convex error function

Note: in 3D error surface can be visualized as shown but not in more than 3 dimensions.

1. A random combination of bias B1 and input weights W1 (showing only one as more than one is not possible to visualize)

2. Each combination of W1 and B1 is one particular linear model in a neuron. That model is associated with proportionate error e1 (red dashed line).

3. Objective is to drive e1 towards 0. For which we need to find the optimal weight (Woptimal) and bias (Boptimal)

4. The algorithm uses gradient descent algorithm to change bias and weight form starting values of B1 and W1 towards the Boptimal, Woptimal.

Gradient Descent (Continued)

Input Layer · Hidden Layer #1 · Hidden Layer #2 · Output Layer

e1

Happens at every node

Actual y1 given X1

Predicted y1 given X1

w1

Error

e1
e2

Bi
aBopti
s mal

B
1

Wopti
mal
Wei
ght

W
1

Least error E2 is at the global minima of the convex function which only one unique combination of weight (woptimal) and bias (boptimal) will fetch us.

Gradient Descent (Continued)

- Let target value for a training example X be y i.e. The data frame used for training has value X,y

- Let the model (represented by random m and c) predict the value for the training example X to be yhat

- Error in prediction is E = yhat − y. If we sum all the errors across all data points, some will be positive some negative and thus cancel out

- To prevent the sum of errors becoming 0, we square the error i.e. E = (y − yhat)^2. Note: in squared expression, y − yhat or yhat − y mean the same

Gradient Descent (Continued)

- Sum of (y – yhat)^2 across all the X values is called SSE (Sum of Squared Errors)

- Using gradient descent (descend towards the global minima). Gradient descent uses partial derivatives i.e how the SSE changes on slightly modifying the model parameters m and c one at a time

$$d(E) / d(m) = d(sum(yhat - y)^2) / d(m)$$
$$d(E) / d(c) = d(sum(yhat - y)^2) / d(c)$$

Gradient Descent (Continued)

- Gradient descent is a way to minimize an objective function / cost function such as Sum of Squared Errors (SSE) that is dependent on model parameters of weight / slope and bias.

- The parameters are updated in the direction opposite to the direction of the gradient (direction of maximum increase) of the objective function.

- In other words we change the values of weight and bias following the direction of the slope of the surface of the error function down the hill until we reach minima.

- This movement from starting weight and bias to optimal weight and bias may not happen in one shot. It is likely to happen in multiple iterations. The values change in steps.
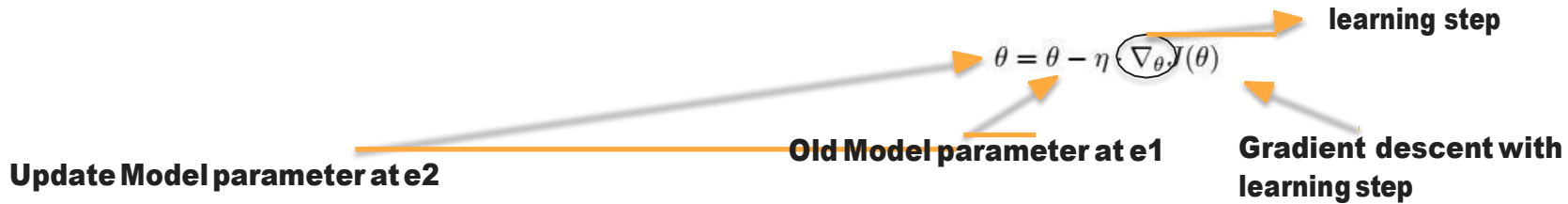
Gradient Descent (Continued)

- The step size can be influenced using a parameter called Learning Rate. It decides the size of the steps i.e. the amount by which the parameters are updated. Too small learning step will slow down the entire process while too large may lead to an infinite loop.

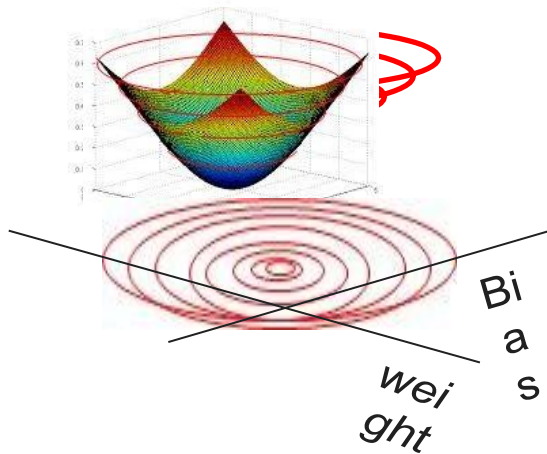- The mathematical expression of gradient descent.

learning step

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta)$$

Old Model parameter at e1

Gradient descent with learning step

Update Model parameter at e2

Transform our error function (which is a quadratic / convex function) into a contour graph. Gradient is always found on the input model parameters only
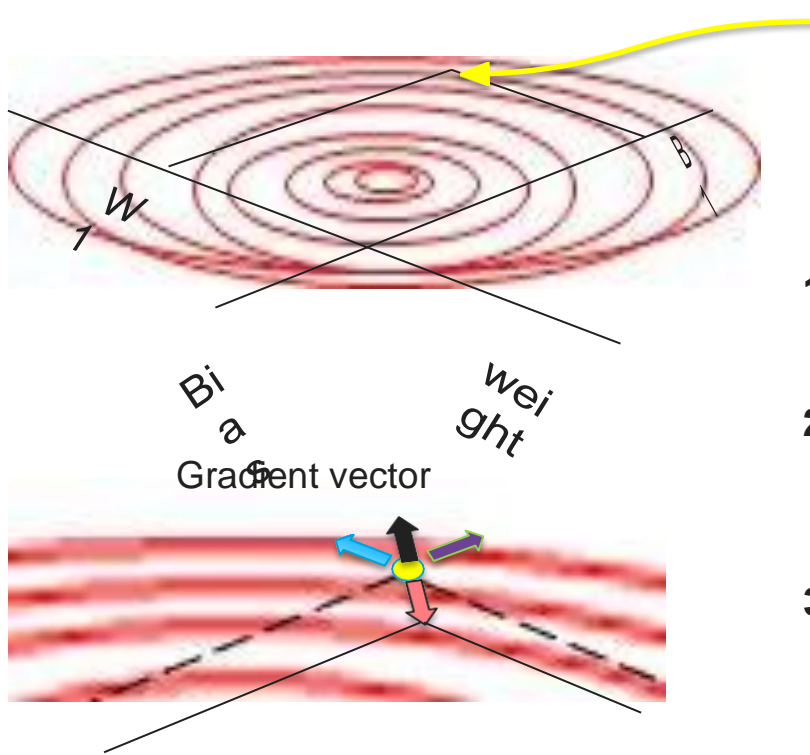

1. Every ring on the error function represents a combination of coefficients (m1 and m2 in the image) which result in same quantum of error i.e. SSE

2. Let us convert that to a 2d contour plot. In the contour plot, every ring represents one quantum of error.

3. The innermost ring / bull's eye is the combination of the coefficients that gives the lease SSE

Contour graph

Randomly selected starting point

W

B

Bias

weight

Gradient vector

About the contour graph –

1. **Outermost circle is highest error while innermost is the least error circle**

2. **A circle represents combination of parameters which result in same error. Moving on a circle will not reduce error.**

3. **Objective is to start from anywhere but reach the innermost circle**

Gradient Descent Steps –

1. First evaluate dy(error)/d(weight) to find the direction of highest increase in error given a unit change in weight (Blue arrow). Partial derivative w.r.t. to weight

2. Next find dy(error) /d(bias) to find the direction of highest increase in error given a unit change in bias (green arrow). Partial derivative w.r.t. to bias

3. Partial derivatives give the gradient in the given axis and gradient is a vector

4. Add the two vectors to get the direction of gradient (black arrow) i.e. direction of max increase in error

5. We want to decrease error, so find negative of the gradient i.e. opposite to black arrow (Orange arrow). The arrow tip is new value of bias and weight.

6. Recalculate the error at this combination an iterate to step 1 till movement in any direction only increases the error

Drawbacks of ANN

Drawbacks of ANN

- ANN models are not much interpretable so it does not give any clue on how it has come up with a solution. Thus it does not make ANN models reliable or trustworthy.

- There is no specific rule for determining the architecture of the network so we mostly use hit and trial method.

- In-order to work efficiently, ANN requires parallel computing power.

- ANN can only work with numerical data so problems or data need to be converted into numerical format before feeding it to ANN.

- Duration of the model to train on the data takes longer.