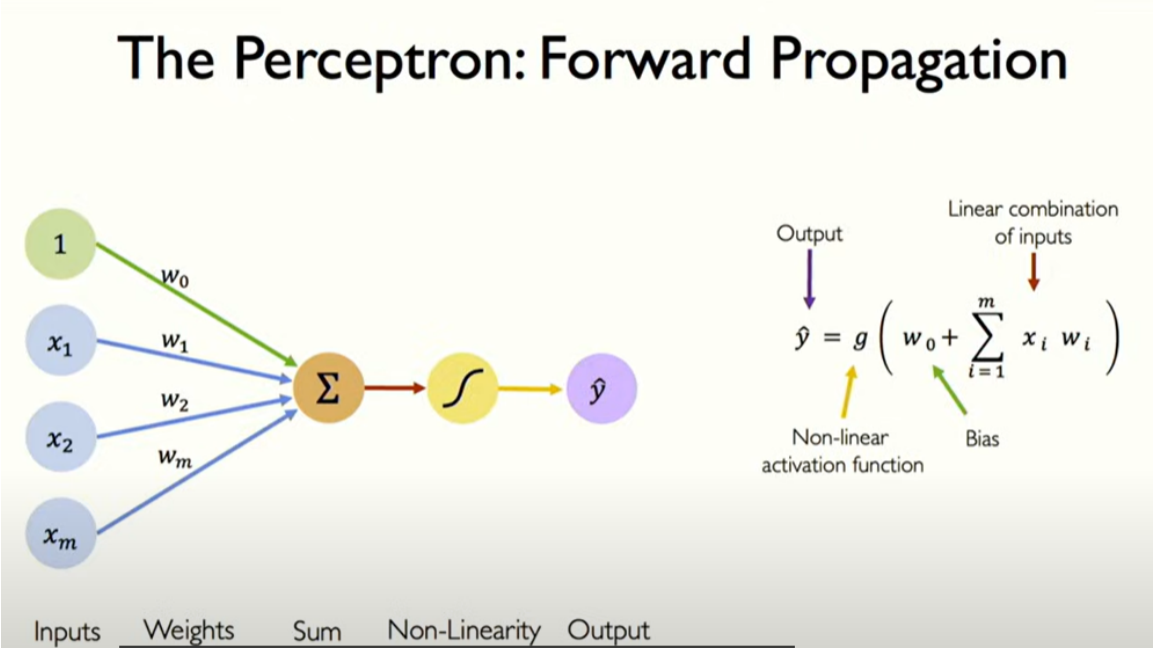
Lec-1(Introduction to DL)

Perceptron

- A single node/neuron inside the neural network - structural building block of deep learning
- Mathematically - dot product of the inputs with respective weights, then add bias weight. this is given to a non-linear activation function which gives output of our neuron.



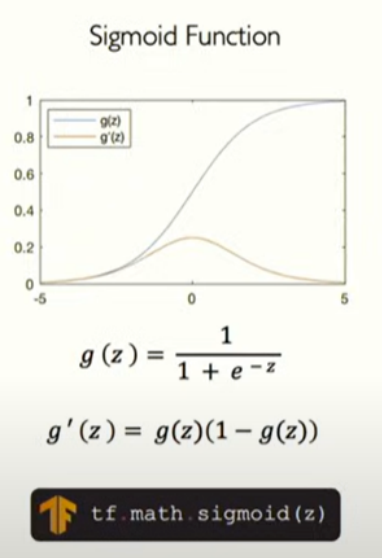
$$y = g(w_0 + X^T W)$$

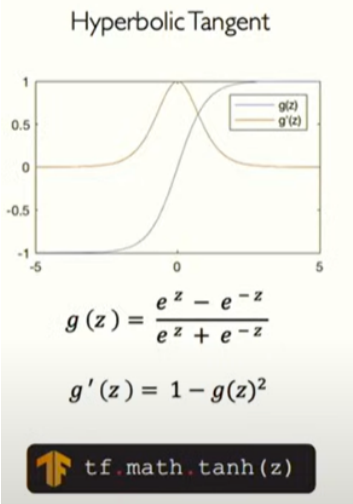
where X^T is transpose of matrix X viz. the inputs and W is matrix of weights.

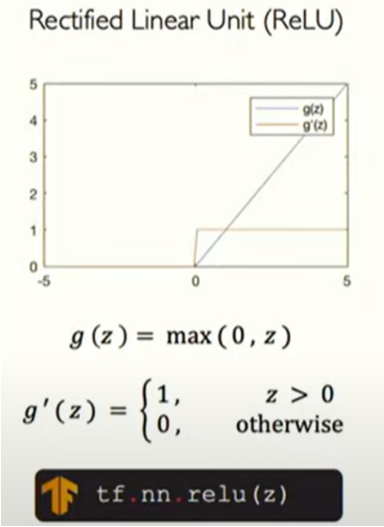
Activation functions

The purpose of activation functions is to introduce non-linearities into network. Some commonly used functions are:

1. Sigmoid function
2. Hyperbolic Tangent
3. Rectified Linear Unit (ReLU)





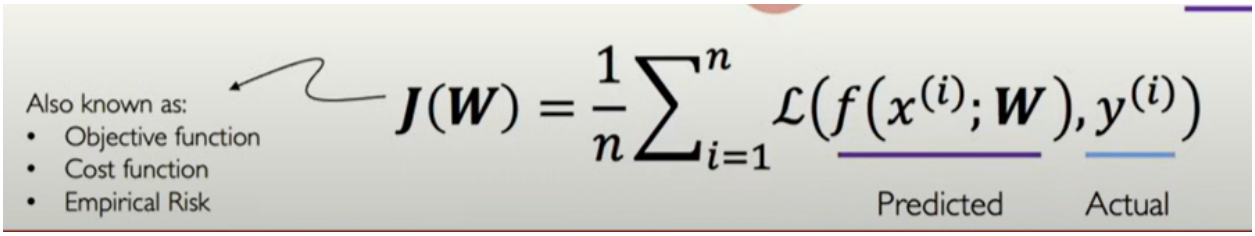


Loss Functions

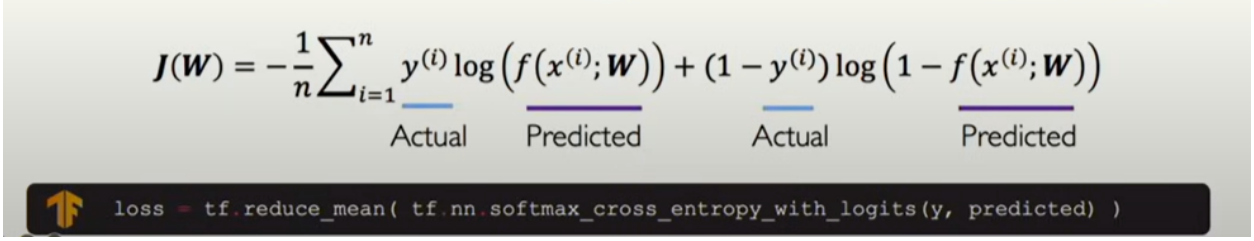
To teach the model on incorrect predictions, we need loss functions which shows the gap between actual and predicted values.

The smaller the difference between actual and predicted values - smaller the loss and vice versa

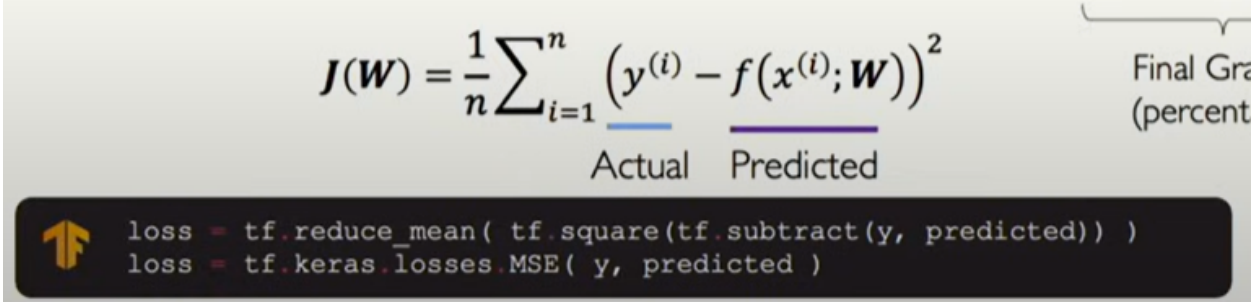
Empirical loss - total loss over the entire dataset



Binary Cross Entropy loss - Softmax function - for binary classification



Mean squared error loss - regression models that output continuous real numbers.



Gradient Descent

We want loss optimization - find network weights that achieves lowest loss.

`Loss is a function of network weights`

Loss function is a gradient which has multi-dimensional weight measures and we need to find the lowest point in that gradient whose coordinated give the least loss.

💡 **GD Algorithm**

1. Initialize weights randomly $\sim N = (0, \sigma^2)$
2. Loop until convergence
    a. Compute gradient, $\frac{\delta J(W)}{\delta W}$
    b. Update weights, $W \leftarrow W - \eta \frac{\delta J(W)}{\delta W}$
3. Return weights

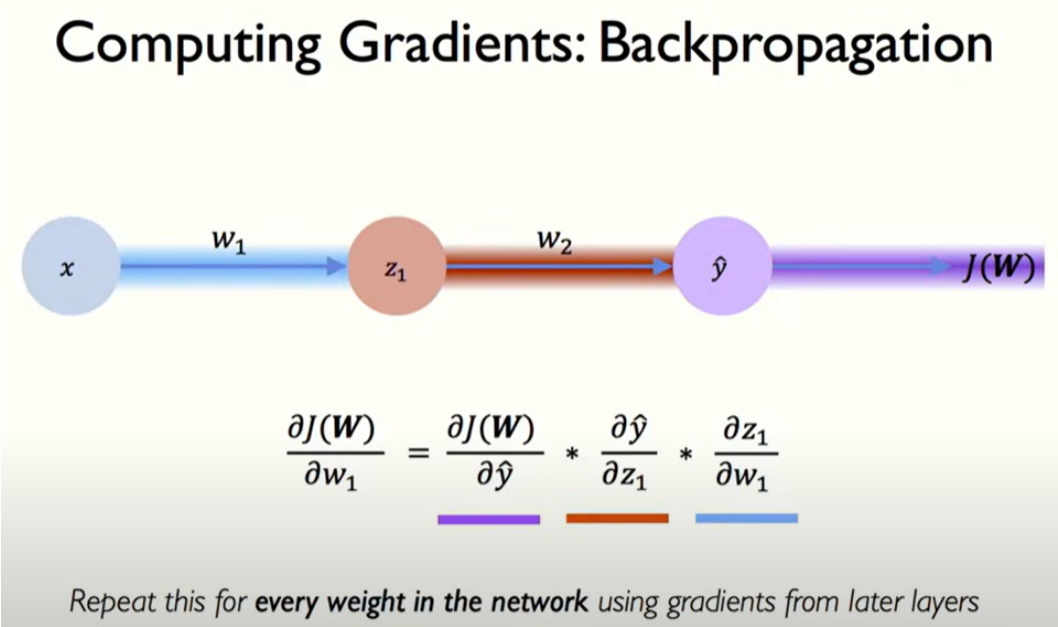
```
import tensorflow as tf

weights = tf.Variable([tf.random.normal()])

while True: #loop forever
    with tf.GradientTape() as g:
        loss = compute_loss(weights)
        gradient = g.gradient(loss, weights)
    weights = weights - lr * gradient
```

Backpropagation

The differential in 2.a above is actually how our loss changes as a function of our weights.



Learning Rate (η)

- Too small learning rate can result in getting stuck at local minima.
- Too large learning rate can result in overshooting the minima and sometimes even get deflected entirely

Instead of keeping fixed learning rate, we can use adaptive learning rate - can be made larger or smaller depending on

- how large the gradient is
- how fast learning is happening
- size of particular weights
- etc...

Different gradient algo —

1. SGD - tf.keras.optimizers.SGD - Keifer & Wolfowitz. “Stochastic Estimation of the Maximum of a Regression Function.” 1952
2. Adam - tf.keras.optimizers.Adam - Kingma et. al. ”Adam: A Method for Stochastic Optimization.” 2014

3. Adadelta - tf.keras.optimizers.Adadelta - Zeiler et. al. “ADADELTA: An adaptive learning rate method.” 2012
4. Adagrad - tf.keras.optimizers.Adagrad - Duchi et. al. “Adaptive Subgradient methods for Online Learning and Stochastic Optimization.” 2011
5. RMSProp - tf.keras.optimizers.RMSProp

Stochastic Gradient Descent (SGD)

💡 **Algorithm**

1. Initialize weights randomly $\sim N = (0, \sigma^2)$
2. Loop until convergence
    a. Pick a single data point i
    b. Compute gradient, $\frac{\delta J_i(W)}{\delta W}$
    c. Update weights, $W \leftarrow W - \eta \frac{\delta J(W)}{\delta W}$
3. Return weights

→ Adv - easier to compute from only 1 point → Disadv - very noisy (stochastic)

So the solution is to take a batch of data point instead of 1 data point.

💡 **Algorithm**

1. Initialize weights randomly $\sim N = (0, \sigma^2)$
2. Loop until convergence
    a. Pick batch of B single data points
    b. Compute gradient, $\frac{\delta J(W)}{\delta W} = \frac{1}{B} \sum_{k=1}^{B} \frac{\delta J_k(W)}{\delta W}$
    c. Update weights, $W \leftarrow W - \eta \frac{\delta J(W)}{\delta W}$
3. Return weights

Mini batches while training :

→ more accurate estimation of gradient

→ smoother convergence

→ allows for larger learning rates

→ fast training

→ parallelize computation

→ achieves significant speed increases on GPUs

Overfitting & Regularization

Overfitting can create highly complex model which cannot generalize, hence resulting in inaccurate predictions on new/unknown data.

Regularization constrains our optimization problem to discourage complex models.

1. Dropout - during training, randomly select some activations/neurons to 0
    a. Typically ‘drop’ 50% of activations in layer
    b. Forces network to not rely on any 1 node
2. Early stopping - (model agnostic, can be applied to any type of model)

Regularization 2: Early Stopping

- Stop training before we have a chance to overfit