ML for Finance: Intro to Deep Learning - Multilayer Perceptrons[1]

S. Yanki Kalfa

JHU-SAIS

October 17, 2022

[1] These slides use Dr. Vural's Data Science for Finance class (MGTF-495) as basis

Outline

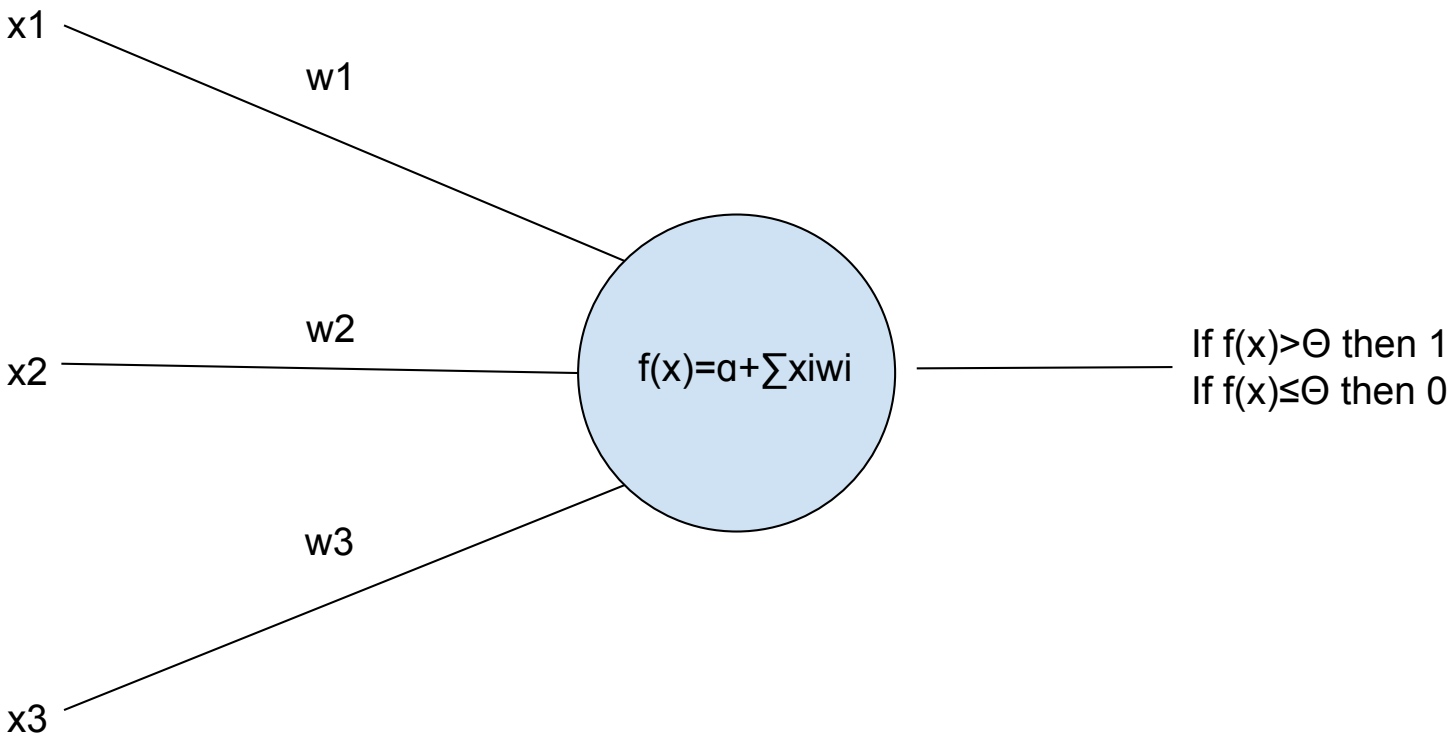
Recap of Machine Learning Methods

- Standard time series methods allow us to use history of a variable to forecast the future (e.g. ARMA)
- ARMAx models introduce exogenous regressors to the table and use other potential correlated features to forecast
- Penalized regressions introduce a penalty term into the objective function and serve as model selection tools
- Tree based methods allow us to introduce non-linearities by dividing the feature space into high-dimesional rectangles (e.g. Random Forests).

Perceptron

- Used for binary (0 or 1) classification
- Calculates sum of weighted inputs
- Classifies output as 1 if sum is greater than certain threshold θ

Perceptron- Diagram

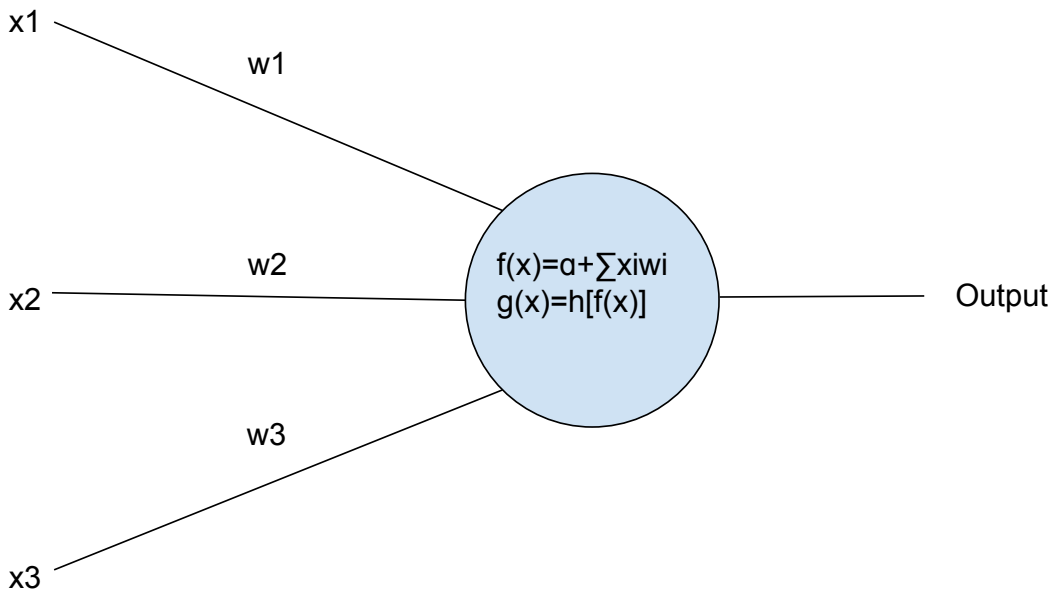


Perceptron- Drawback

- The perceptron only uses linear combinations of inputs
- How to include non-linearities?

Activation Functions

- Activation functions introduce non-linearities
- Transform the weighted sum of inputs



Common Activation Functions

- Sigmoid:

$$S(x) = \frac{1}{1 + e^{-x}}$$

$$S'(x) = S(x)(1 - S(x))$$

- Tanh:

$$T(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

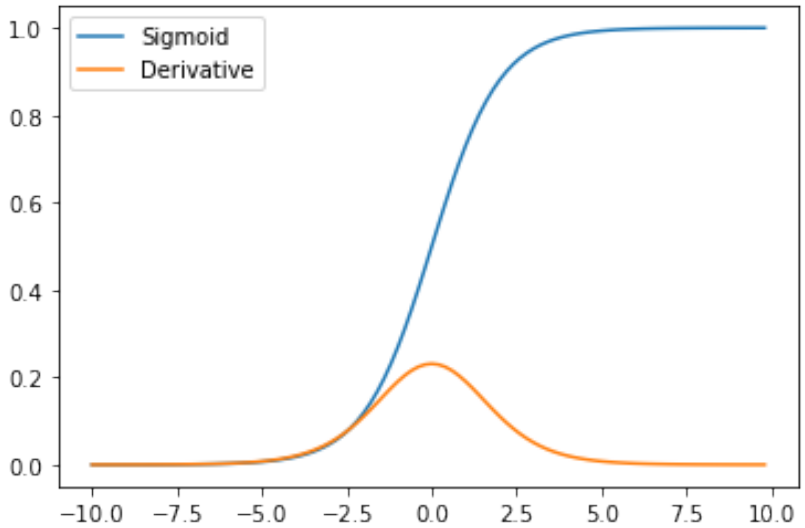
$$T'(x) = 1 - \left[\frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}\right]^2$$

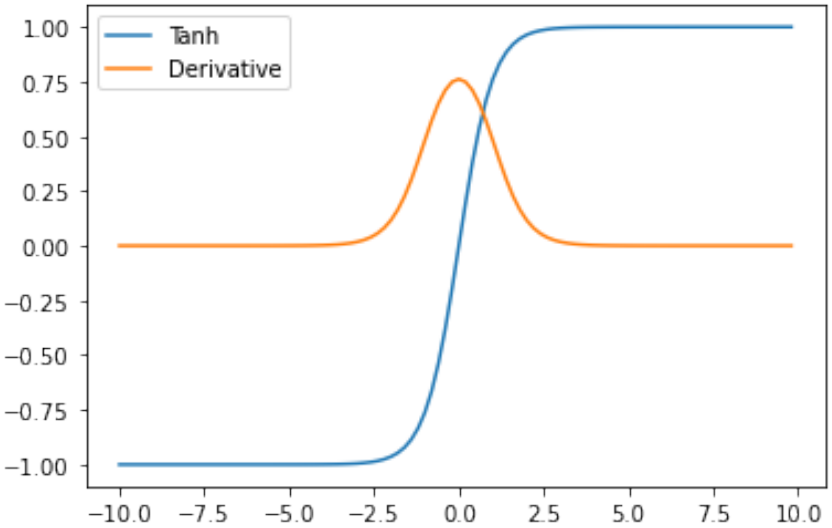
- ReLU:

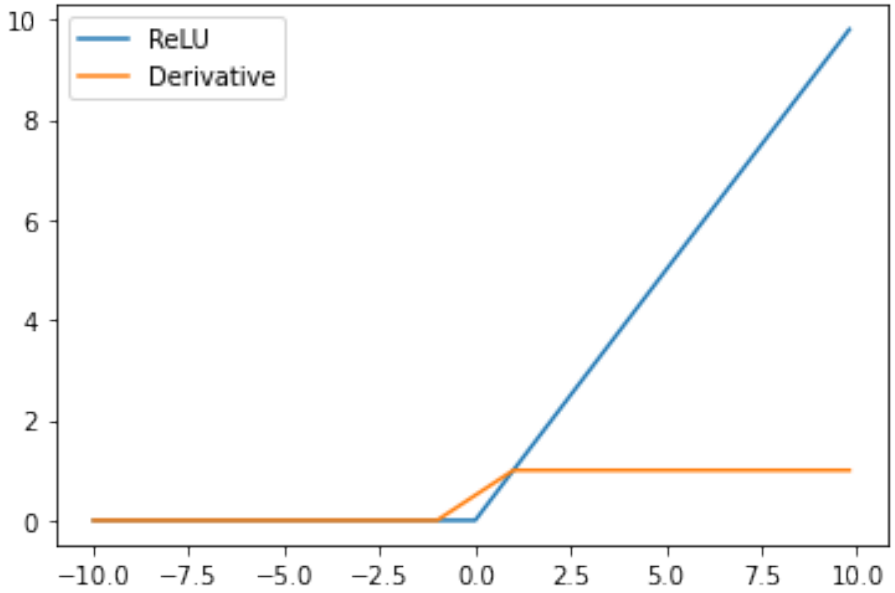
$$R(x) = \max(0, x)$$

$$R'(x) = \begin{cases} 0 \text{ if } x \leq 0 \\ 1 \text{ if } x > 0 \end{cases}$$

Common Activation Functions



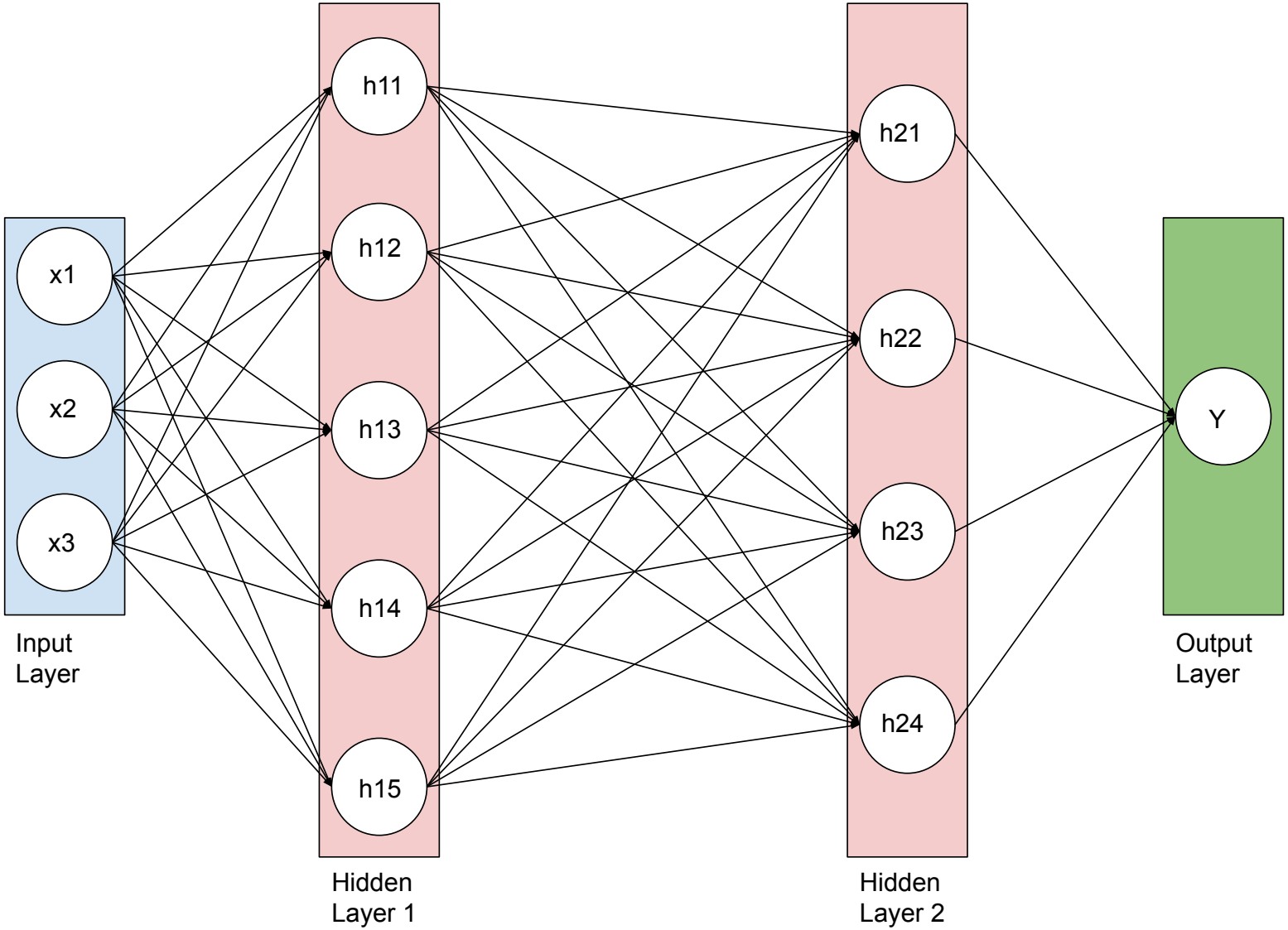




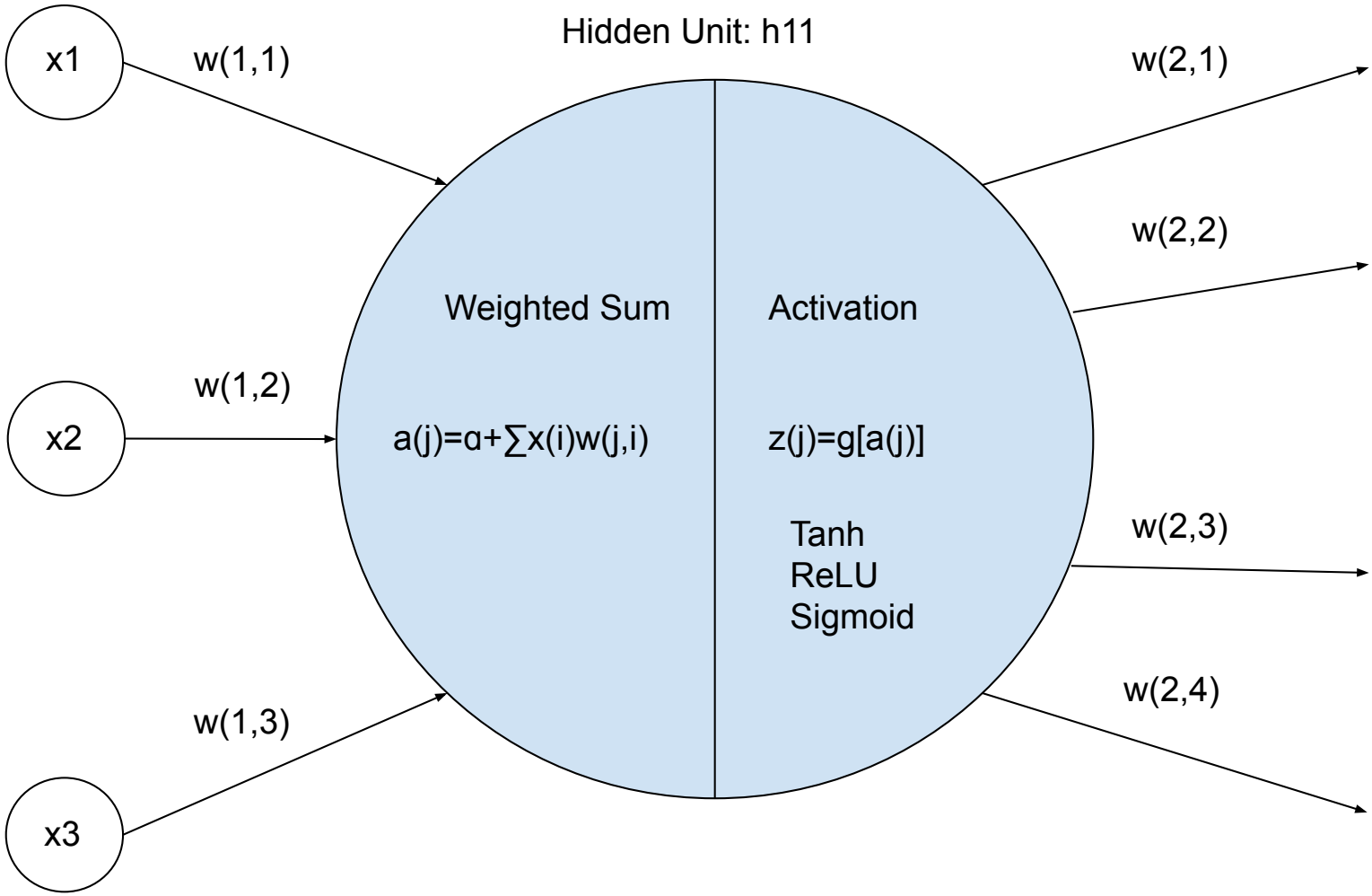
Multilayer Perceptron-Introduction

- Multilayer Perceptron (MLP) is another term for Fully Connected Feed-Forward Neural Networks
- "Fully Connected" because all neurons are conncected to each other
- We call them feed forward because the information "feeds forward" through the network
- There are three main components of MLP:
  1. Input Layer
  2. Hidden Layer(s)
  3. Output Layer

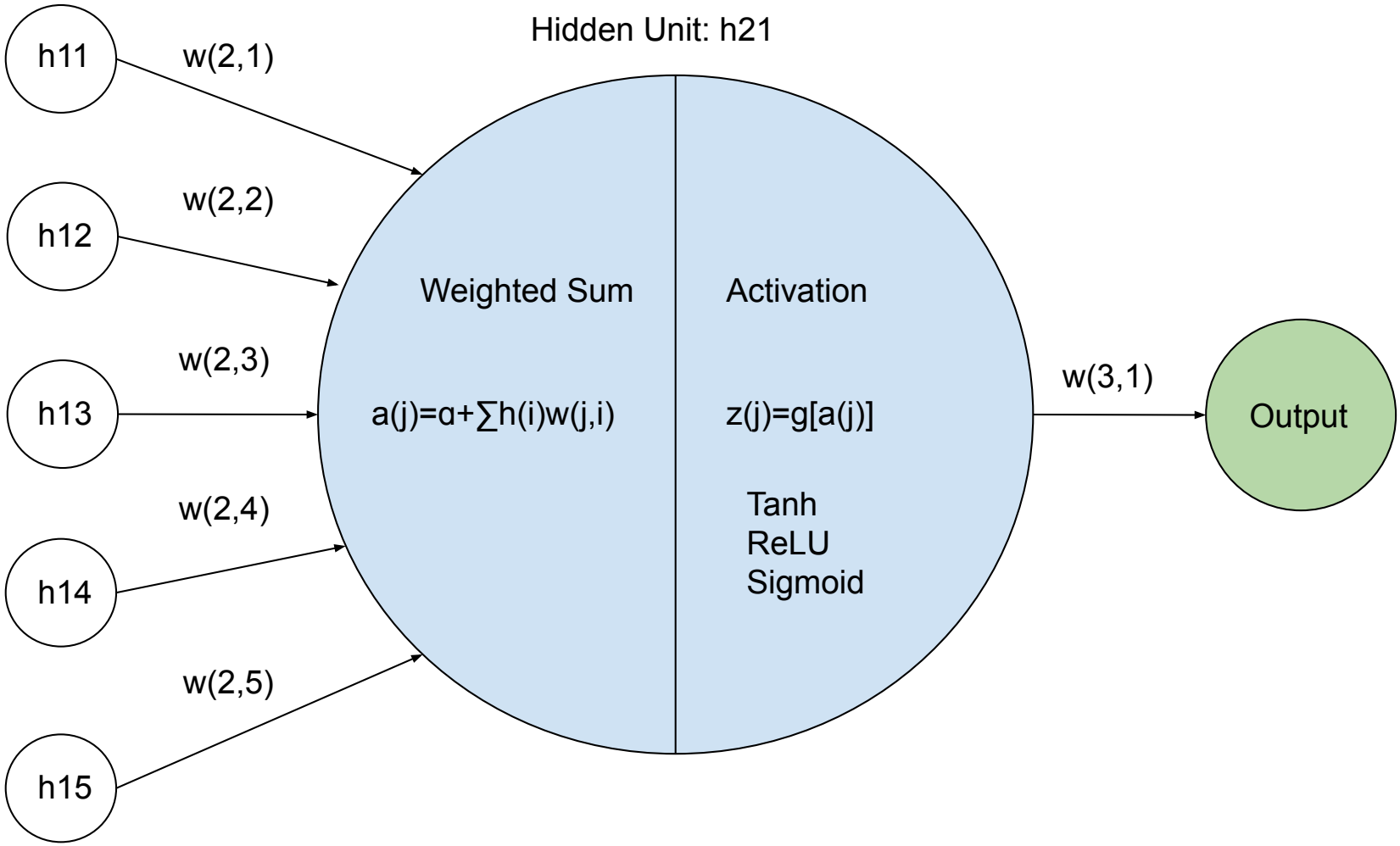
MLP - Architecture



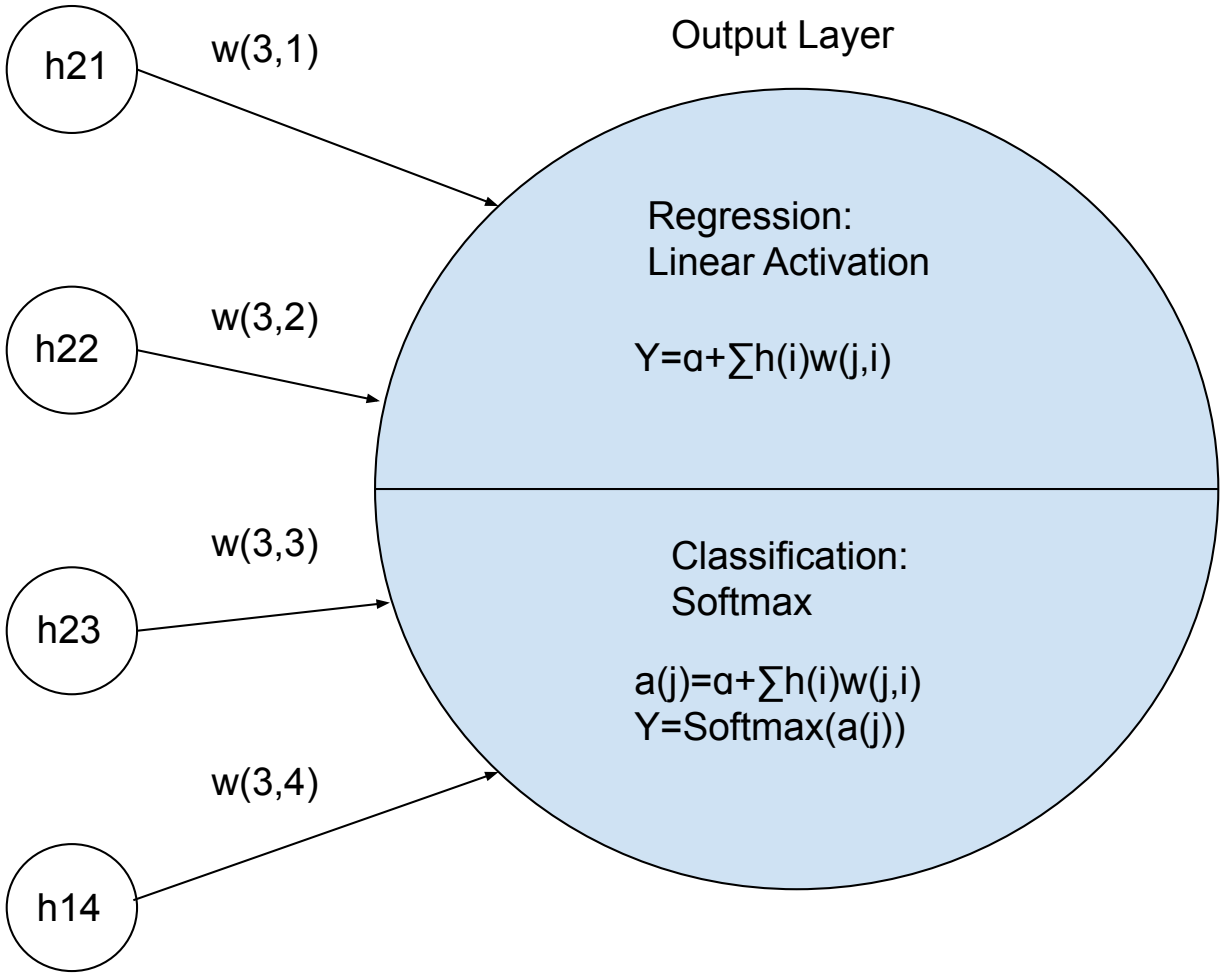
MLP - Hidden Neuron Layer 1



MLP - Hidden Neuron Layer 2



MLP - Output Layer



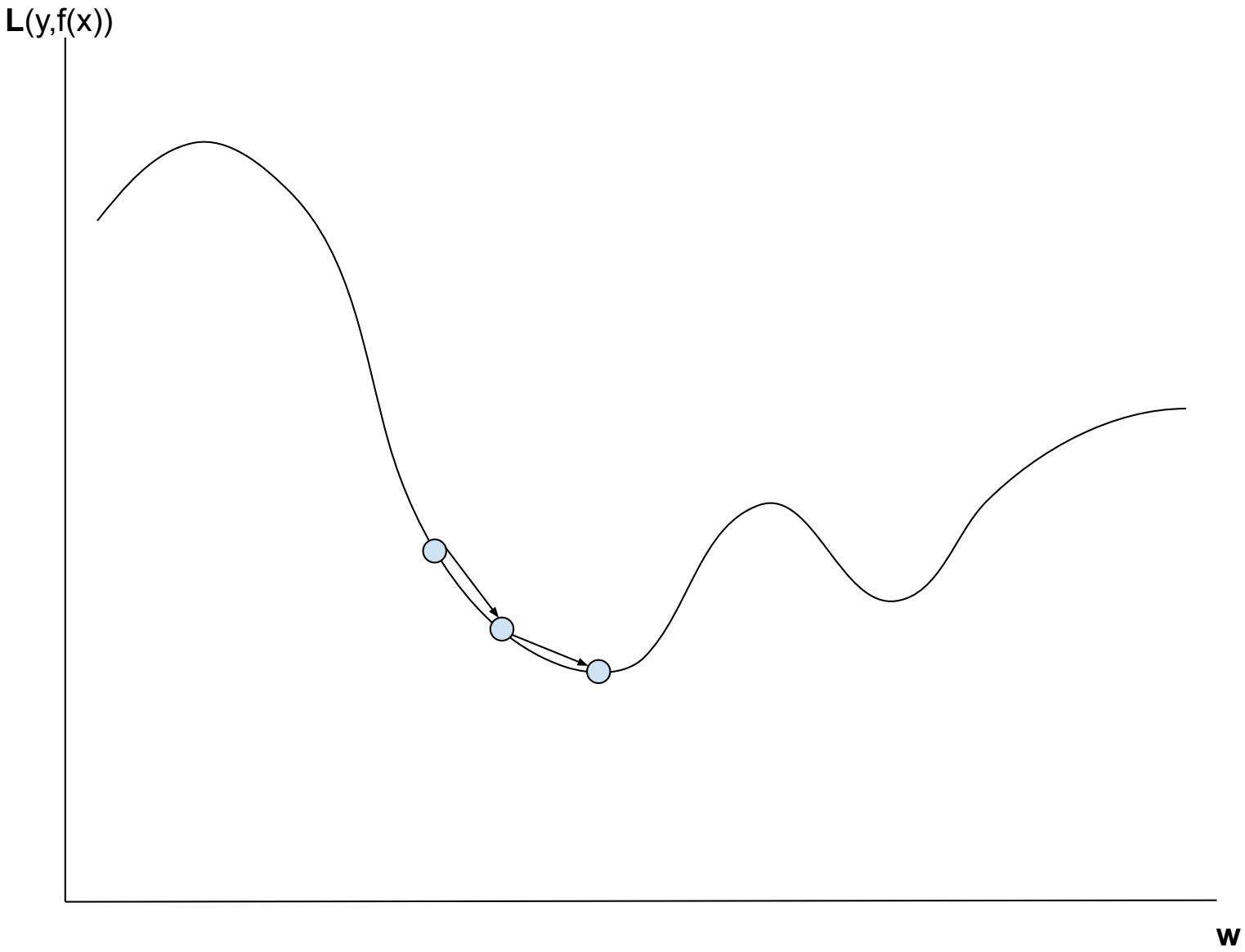
Gradient Descent

- We use gradient descent to find the weights that minimize the loss function
- For regression the commonly used loss function is the squared error loss (MSE)
- For classification we use Cross Entropy
- The loss function almost surely is non-convex, hence taking derivatives does not guarantee finding the minimum

How does Gradient Descent work?

1. Make prediction
2. Calculate loss
3. Update weights

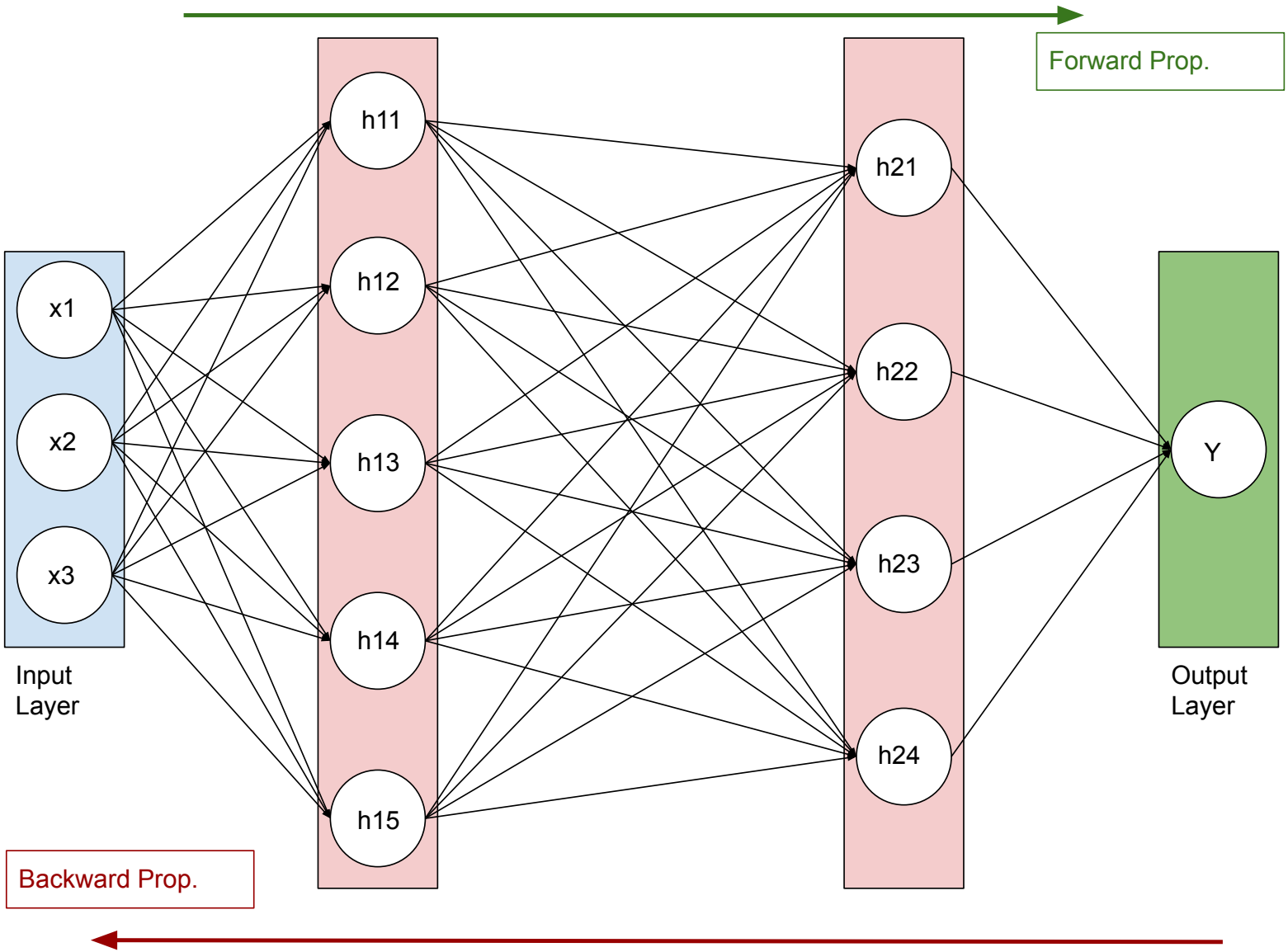
Loss Function Diagram



Optimal Weights

- How do we find optimal weights?
- The process in which we update weights to minimize the loss function is called Backpropagation

Backpropagation: Diagram



Backpropagation

- For any weight in the architecture we use the following rule to update the weights

$$\omega_{i,j}^{(t+1)} = \omega_{i,j}^{(t)} - \eta \frac{\partial L}{\partial \omega_{i,j}^{(t)}}$$

- Where η is the learning rate. The smaller the learning the longer the convergence. However, this does not mean that you should pick a large η because you can jump over the optimum.
- η is a hyperparameter- you can tune it or just choose one.

Technical Details

Some notation first:

- z_j can be:
  1. the output of a hidden unit
  2. the input to a hidden unit

We want to calculate:

$$\frac{\partial L}{\partial w_{i,j}^{(t)}}$$

We will use the chain rule to calculate the partial derivative defined above.

Technical Details

$$\frac{\partial L}{\partial w_{i,j}^{(t)}} = \frac{\partial L}{\partial a_j}\frac{\partial a_j}{\partial w_{i,j}}$$

$$a_j = \alpha + \sum h_{i,j} w_{i,j}$$

$$\frac{\partial a_j}{\partial w_{i,j}} = z_j$$

$$\frac{\partial L}{\partial w_{i,j}^{(t)}} = \frac{\partial L}{\partial a_j} z_i$$

$$\delta_j \equiv -\frac{\partial L}{\partial a_j}$$

$$\omega_{i,j}^{(t+1)} = \omega_{i,j}^{(t)} + \delta_j z_i$$

We now have a general definition for the updating rule.

Optimizers

- Deep learning algorithms are not one shot estimators.
- This is evident from the forward and back propagation.
- We should not confuse how the gradients are calculated at each layer with how the Loss function is minimized.
- Optimizers minimize the Loss function given the paremeters of the model.
- Neural Networks are trained using optimizers and backpropagation.

Gradient Descent

- 3 Variants:
  - Batch Gradient Descent: Takes in the full training sample

$$\theta = \theta - \eta \nabla_\theta L(\theta)$$

  - Stochastic Gradient Descent: Takes in one sample at a time

$$\theta = \theta - \eta \nabla_\theta L(\theta | x_t, y_t)$$

  - Mini-batch Gradient Descent: Takes a sample smaller than the full sample but larger than one sample

$$\theta = \theta - \eta \nabla_\theta L(\theta | x_{t:t+n}, y_{t:t+n})$$

Gradient Descent Optimizers- Momentum

1. Momentum: Accelerate SGD. Compute gradient at current location then jump in the direction of the gradient.

$$\nu_t = \gamma \nu_{t-1} + \eta \nabla_\theta L(\theta)$$
$$\theta = \theta - \nu_t$$

2. Nesterov Accelerated Momentum: Accelerate but know where you are going: Jump in the previously computed gradient, measure the gradient at the current location then adjust.

$$\nu_t = \gamma \nu_{t-1} + \eta \nabla_\theta L(\theta - \gamma \nu_{t-1})$$
$$\theta = \theta - \nu_t$$

Adaptive Learning Rate Optimizers

1. Adagrad: Adapts learning rate to the parameters. Large update for infrequent parameters, small updates for frequent parameter.

$$g_{t,i} = \nabla_{\theta_t} L(\theta_{i,t})$$
$$\theta_{t+1,i} = \theta_{i,t} - \eta g_{i,t}$$
$$\theta_{t+1,i} = \theta_{i,t} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} g_{i,t}$$

$G_{t,ii}$ diagonal matrix where element are sum of squares of gradient. ϵ is a smoothing term to avoind division by 0. No need to tune the learning rate manually. Because Adagrad accumulates square graidents in the denominator, the learning rate shrinks to zero fast, and so we stop learning.

Adaptive Learning Rate Optimizers

2. AdaDelta: Extension to Adagrad, where it restricts the the window of accumulated square gradients. The sum of gradients recursively defined as a decayinh average of all past square gradients.

$$\mathbb{E}[g_t^2] = \gamma \mathbb{E}[g_{t-1}^2] + (1-\gamma) g_t^2$$
$$\Delta\theta_t = -\theta_{i,t}$$
$$\theta_{t+1} = \theta_t + \Delta\theta_t$$
$$\Delta\theta_t = -\frac{\eta}{\sqrt{\mathbb{E}[g_t^2] + \epsilon}} g_t$$

Adaptive Learning Rate Optimizers

3. Adam: Adaptive Moment Estimation. Stores exponentially decaying average past square and level gradients.

$$
\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\
v_t &= \beta_2 v_{t-1} + (1 - beta_2) g_t^2 \\
\hat{m}_t &= \frac{m_t}{1-\beta_{1,t}} \\
\hat{v}_t &= \frac{v_t}{1-\beta_{2,t}} \\
\theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t
\end{aligned}
$$

m_t and v_t are the mean and variance of the gradients. They are initialized at 0.

Vanishing Gradient Problem

- Gradients are crucial to train neural networks
- We calculate the gradients with backpropagtion
- We use the gradients to minimize the Loss function

Vanishing Gradient

- We start calculating the gradients from the end of the MLP structure and go backwards
- To calculate gradients at the shallower hidden layers we use the chain rule
- What is the potential pitfall here?

Vanishing Gradient- Sigmoid Activation

Let's take the case of an MLP with 3 hidden layers with a single unit. The sigmoid activation function takes the following form:

$$S(x) = \frac{1}{1 + e^{-x}}$$

$$\nabla = \frac{dS(x)}{dx} = S(x)(1 - S(x))$$

$$\nabla_{max} = \frac{1}{4}$$

Vanishing Gradient- Sigmoid Activation

The best possible outcome is $1/4$. We know the updating rule is:

$$\omega = \omega - \eta \frac{\partial L}{\partial \omega}$$

If we go the third hidden layer the gradient at best can be $1/4$, in the second layer the gradient can be at best $(1/4)^2$, in the third hidden layer the gradient can be at best $(1/4)^3$.
At the shallower layers, we have almost no information update. To fix this problem, we need to pick a different activation function such as the ReLU function.

Regularization-Dropout

- Neural Nets are very flexible tools that fit the data with almost perfect fit.
- This means that we need to protect ourselves against overfitting.
- How do we avoid overfitting?

Regularization-Dropout

Regularization-Dropout

Regularization-Dropout

- Benefits:
  - We drop some neurons
  - The neurons left learn generic representation
  - Prevent Overfitting
- Adds noise to Gradients:
  - Generalize with noise (mimic unseen data)
- Lower Model Complexity:
  - Smaller number of neurons
- During test all neurons are included

Feature Scaling

- MinMax Scaling:Scale feature between 0 and 1

$$x_{new} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

- Standard Scaler: Zero mean and Unit variance

$$x_{new} = \frac{x - \mu}{\sigma}$$

Feature Scaling

- Why do we care about the scaling
- When features have very large and very small scales optimization takes a long time.
- Okay, but why?

Feature Scaling

- When features are disproportionate then the ellipses are elongated.
- This causes the steepest gradient to be almost perpendicular to the direction of the minimum point.

Feature Scaling

Batch Normalization

- We can conduct feature scaling before the neural net.
- Yet, it appears that we can also scale the outputs of each hidden cell.
- This method is called batch normalization
- it helps speed up optimization

Early Stopping

- We monitor valdation loss with each run
- Early stopping happens when we stop training the model when validation loss does not get smaller within a given amount of runs.
- We call this the patience parameter