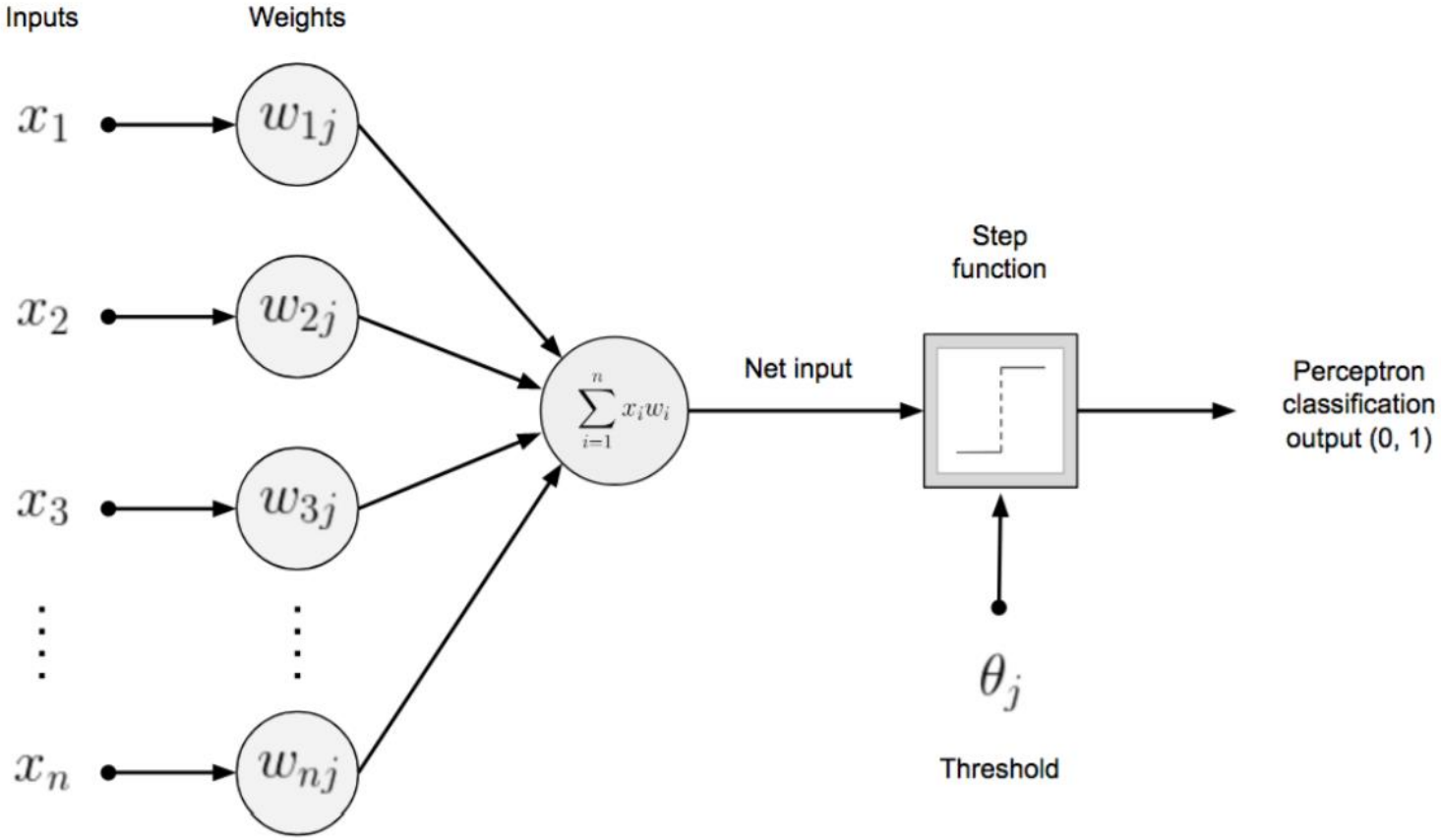
DL - Module 1

Introduction of Neural Networks

Perceptron

- In the field of neural networks, the **perceptron** is considered an artificial neuron using the Heaviside step function for the activation function.
- The perceptron is a **linear-model binary classifier** with a simple input–output relationship as depicted:



- Here, we're summing n number of inputs times their associated weights and then sending this "net input" to a step function with a defined threshold.
- Typically, with perceptrons, this is a **Heaviside step function** with a **threshold value** of 0.5.
- This function will **output** a real-valued **single binary value (0 or a 1)**, depending on the input.
- The Heaviside step function equation is given as:

$$f(x) = \begin{cases} 0 & x < 0 \\ 1 & x >= 0 \end{cases}$$

- To produce the net input to the activation function we take the dot product of the input and the connection weights.
- The output of the step function (i.e., the activation function) is the output for the perceptron and gives us a classification of the input values.

Perceptron Learning Algorithm

- The perceptron learning algorithm **changes the weights** in the perceptron model **until all input records** are all **correctly classified**.
- The algorithm **will not terminate if** the learning input is **not linearly separable**.
- A linearly separable dataset is one for which we can find the values of a hyperplane that will cleanly divide the two classes of the dataset.
- The perceptron learning algorithm **initializes** the **weight vector** with **small random values** or **0s** at the beginning of training.
- The perceptron learning algorithm **takes each input record**, as we can see the previous figure, and **computes** the **output classification** to **check against** the **actual classification label**.
- The **first input value** is the **bias input**, which is **always 1** because we don't affect the bias input. The **first weight** is our **bias term** in this diagram.
- The dot product of the input vector and the weight vector gives us the input to our activation function.
- **If** the **classification** is **correct**, **no weight changes** are made. If the classification is **incorrect**, the **weights** are **adjusted accordingly**.
- Weights are updated between individual training examples. This loop continues until all of the input examples are correctly classified.
- If the dataset is not linearly separable, the training algorithm will not terminate. Example: XOR and XNOR cannot be solved by a single layer perceptron

---

Limitations of the Early Perceptron

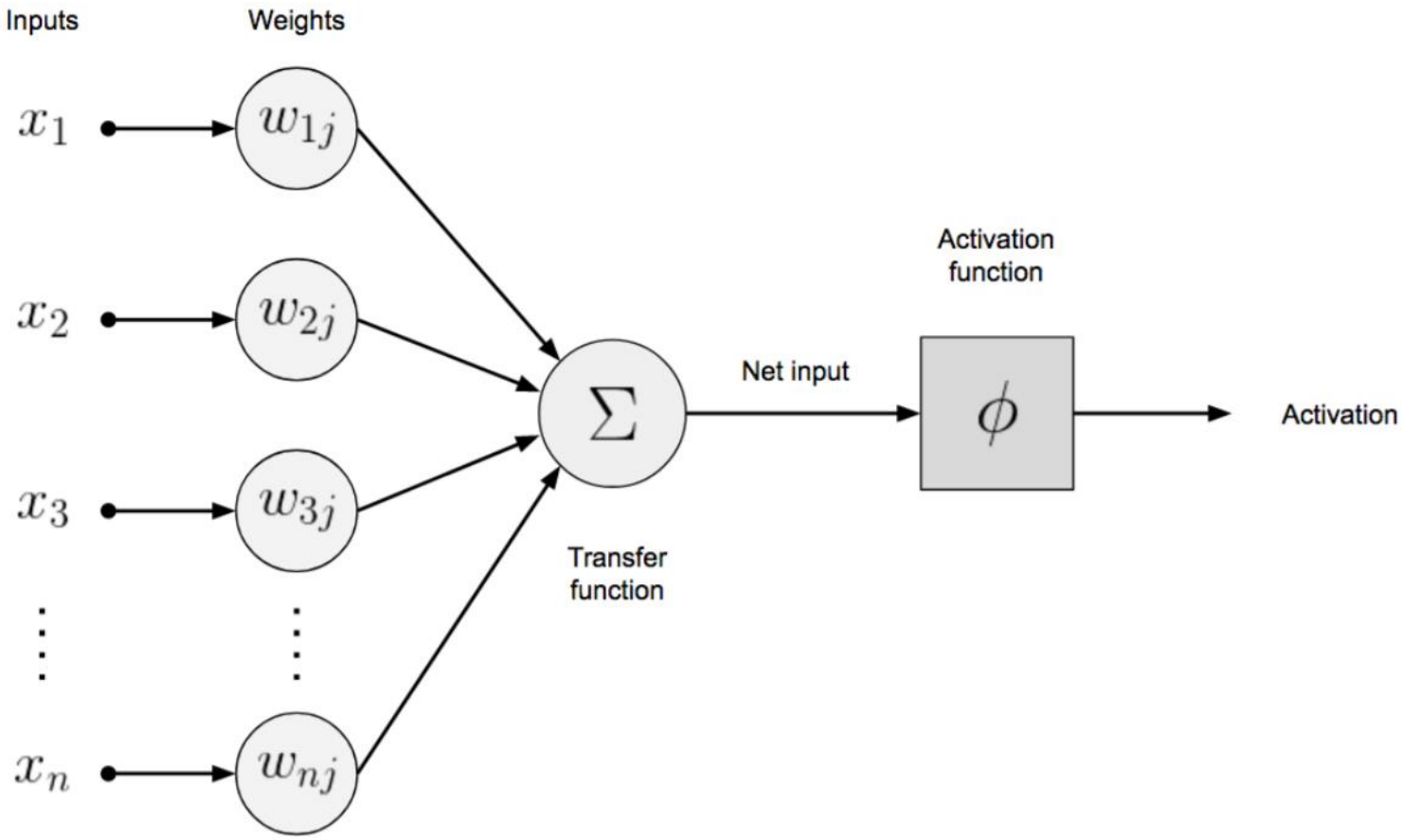
- After initial promise, the perceptron was found to be **limited in the types of patterns it could recognize**.
- The initial **inability to solve nonlinear** (e.g., datasets that are not linearly separable) problems was seen as a failure for the field of neural networks.

Multilayer Feed-Forward Networks

- The multilayer feed-forward network is a neural network with **an input layer**, **one or more hidden layers**, and **an output layer**.
- Each layer has one or more artificial neurons. These artificial neurons are similar to their perceptron precursor yet have a different activation function depending on the layer's specific purpose in the network.

Evolution of the Artificial Neuron

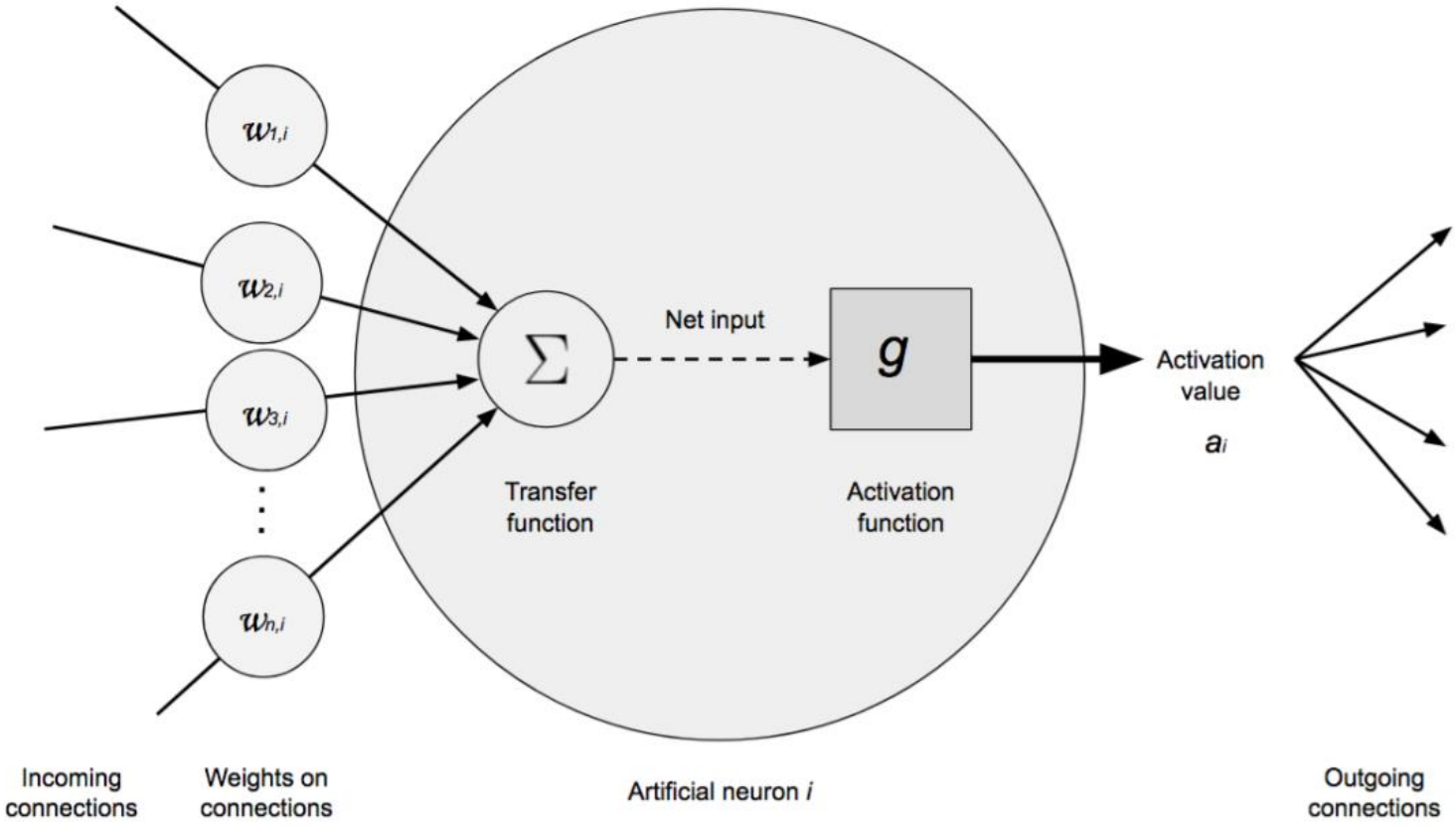
- The artificial neuron of the multilayer perceptron is similar to its predecessor, the perceptron, but it adds flexibility in the type of activation function we can use.
- The below figure shows an updated diagram for the artificial neuron that is based on the perceptron.



- The net input to the activation function is still the dot product of the weights and input features, yet the **flexible activation function allows** us **to create different types out of output values**.
- This is a major contrast to the earlier perceptron design that used a piecewise linear Heaviside step function.
- This improvement now allowed the artificial neuron to express a more complex activated output.

The Artificial Neuron

- We express the net input to an artificial neuron in a multilayer perceptron neural network, as the weights on connections multiplied by activation incoming on connection, as shown:



- In the neurons in the input layer, the activation function is linear, i.e., we pass on the input values as it is, as input to the next layer.
- For hidden layers, the input is the activation from the previous layer neurons. Mathematically, we can express the net input (total weighted input) of the artificial neuron as:

  $$\boldsymbol{input\_sum} = \mathbf{W}_{\boldsymbol{i}} \cdot \mathbf{A}_{\boldsymbol{i}} + \boldsymbol{b}$$

  Where, $\mathbf{W}_{\boldsymbol{i}}$ is the **vector of all weights** leading into neuron i and $\mathbf{A}_{\boldsymbol{i}}$ is the **vector of activation values** for the inputs to neuron i. b is the bias added at every layer to all the neurons in that layer.
- To produce output from the neuron, we'd then wrap this net input with an activation function g , as demonstrated in the following equation:

  $$\boldsymbol{a_i} = \boldsymbol{g}(\boldsymbol{input\_sum_i}) = \boldsymbol{g}(\mathbf{W}_{\boldsymbol{i}} \cdot \mathbf{A}_{\boldsymbol{i}} + \boldsymbol{b})$$
- This activation value for neuron i is the output value passed on to the next layer through connections to other artificial neurons (multiplied by weights on connections) as an input value.

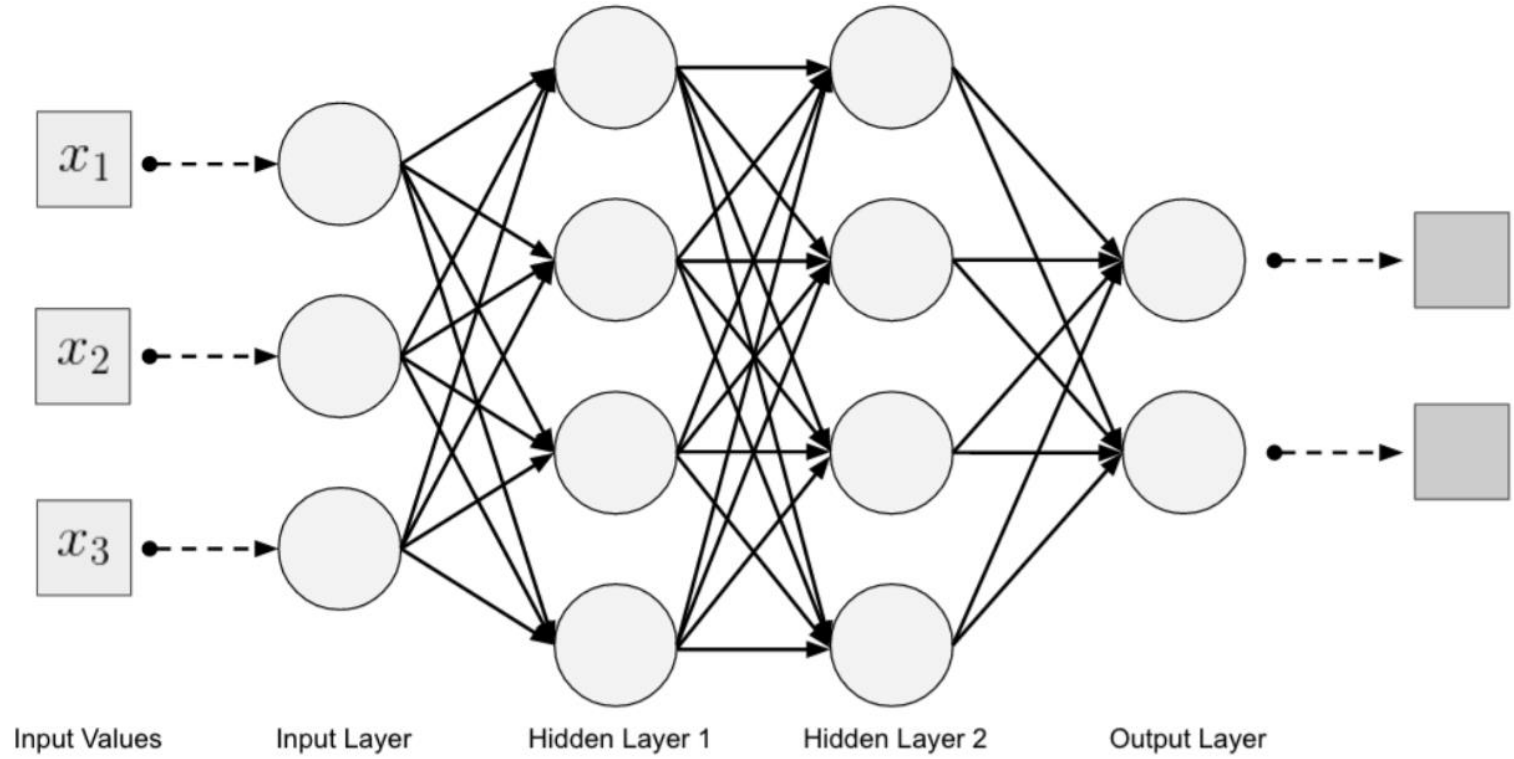
Basic Terminologies

- **Connection weights**:
  - Weights on connections in a neural network are **coefficients that scale** (amplify or minimize) **the input signal** to a given neuron in the network.
  - In common representations of neural networks, these are **the lines/arrows going from one neuron to another**.
  - Often, connections are **notated as w** in mathematical representations of neural networks
- **Biases**
  - Biases are **scalar values added to the input** to **ensure** that **at least a few nodes per layer** are **activated regardless of signal strength**.
  - Biases **allow learning to happen** by giving the network action, **in the event of low signal**. They allow the network to try new interpretations or behaviours.
  - Biases are generally **notated b**, and, like weights, biases are modified throughout the learning process.
- **Activation functions**
  - The functions that **govern the artificial neuron's behavior** are called activation functions.
  - The transmission of that input is known as forward propagation.
  - Activation functions transform the combination of inputs, weights, and biases.
  - Products of these transforms are input for the next neuron layer.
  - These transformations are **usually non-linear** but **can also be linear**.
  - It **transforms** the **data** into a **convenient range**, such as **0 to 1** or **–1 to 1**.

Feed-Forward Neural Network Architecture

- With multilayer feed-forward neural networks, we have artificial neurons arranged into groups called layers. It has the following:
  - A single input layer
  - One or many hidden layers, fully connected
  - A single output layer
- The **neurons in each layer** (represented by the circles) are all **fully connected to** all **neurons in the following layer**.

- The **neurons in one layer** all **use** the **same** type of **activation function**.
- For the input layer, the input is the raw vector input. The input to neurons of the other layers is the output (activation) of the previous layer's neurons.
- As data moves through the network in a feed-forward fashion, it is influenced by the connection weights and the activation function type.



Input Layer

- This layer is how we get input data (vectors) fed into our network.
- The number of neurons in an input layer is typically the same number as the input feature to the network.
- Input layers are followed by one or more hidden layers.
- Input layers in classical feed-forward neural networks are fully connected to the next hidden layer, yet in other network architectures, the input layer might not be fully connected.

Hidden Layer

- There are one or more hidden layers in a feed-forward neural network.
- The weight values on the connections between the layers are how neural networks encode the learned information extracted from the raw training data.
- Hidden layers are the key to allowing neural networks to model nonlinear functions, as we saw from the limitations of the single-layer perceptron networks.

Output Layer

- We get the answer or prediction from our model from the output layer.
- Depending on the setup of the neural network, the final output may be a real valued output (regression) or a set of probabilities (classification).
- This is controlled by the type of activation function we use on the neurons in the output layer.

Connections between Layers

- In a fully connected feed-forward network, the connections between layers are the outgoing connections from all neurons in the previous layer to all of the neurons in the following layer.
- We change these weights progressively as our algorithm finds the best solution (one with minimum error) it can with the backpropagation learning algorithm.

Training Neural Networks

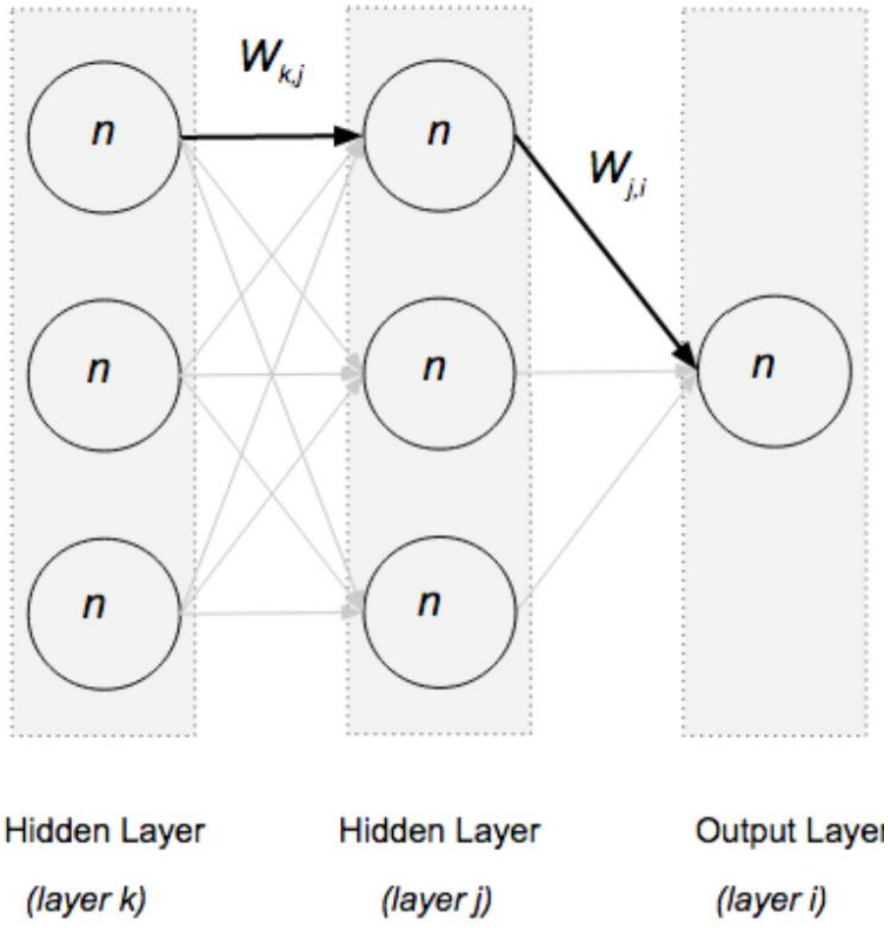
- A **well-trained artificial neural network** has **weights that amplify the signal** and **dampen the noise**.
- Inputs paired with **large weights** will **affect** the **network's interpretation** of the **data more** than inputs paired with smaller weights.
- The **process of learning** for any learning algorithm using weights is the process of **readjusting the weights and biases**, making some smaller and others larger, thus, **allocating significance** to **certain bits** of information and **minimizing other bits**.
- In most datasets, **certain features** are **strongly correlated with certain labels** (e.g., square footage relates to the sale price of a house).
- Neural networks learn these relationships blindly by making a guess based on the inputs and weights and then measuring how accurate the results are.
- The **loss functions** in optimization algorithms, such as stochastic gradient descent (SGD), **reward the network** for **good guesses** and **penalize it for bad ones**.
- SGD moves the parameters of the network toward making good predictions and away from bad ones.

Backpropagation Learning

- Backpropagation is an important part of reducing error in a neural network model.
- With backpropagation, we're trying to minimize the error between the label (or "actual") output associated with the training input and the value generated from the network output.
- **Generic Neural Network Learning Algorithm pseudocode**

```
function neuralNetworkLearning (trainingDataset) returns network
  network ⟵ initializeWeights (randomly)
  start loop
    for each record in trainigDataset do
      networkOutput = neuralNetworkOutput (network, record)
      actualOutput = observed outcome as per training record
      update weights in network based on
        (networkOutput, actualOutput)
    end for
  end loop when all examples correctly predicted or hits
    stopping conditions
  return network
```

- **Backpropagation Algorithm Pseudocode**



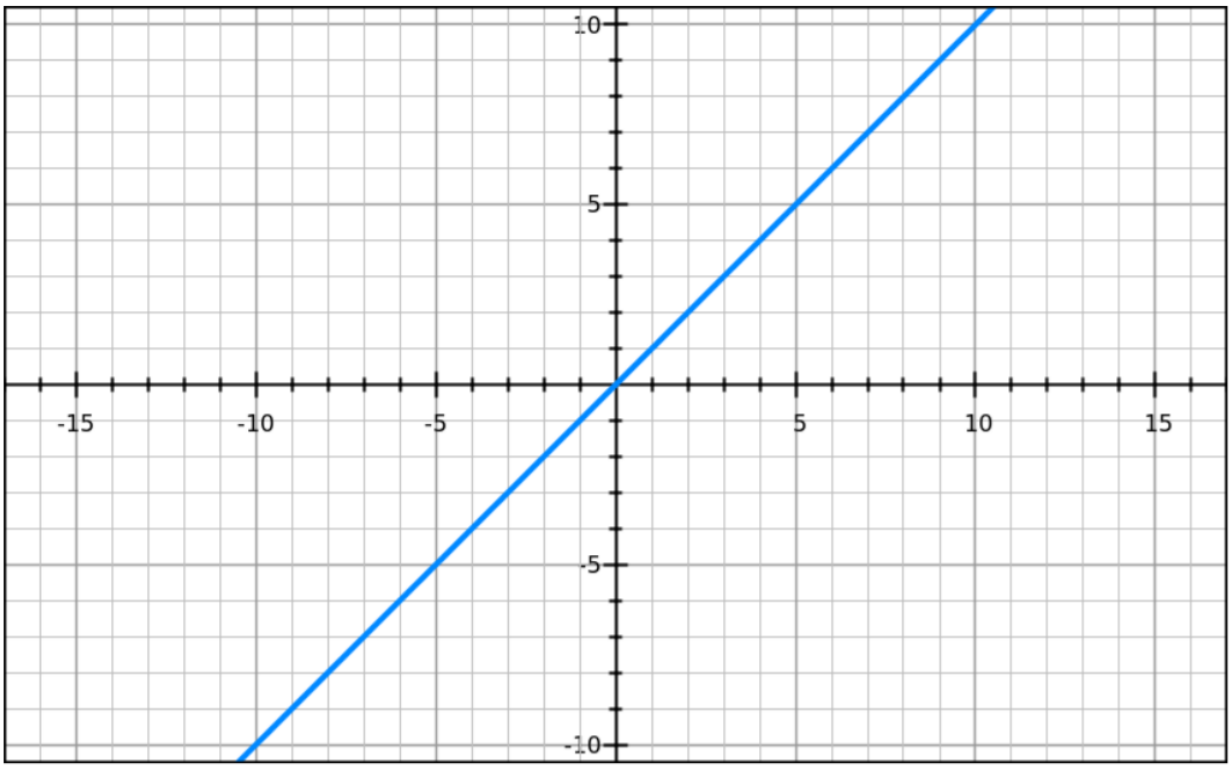
```
function BP (network, trainigDataset, LRate) returns network
  network ⟵ initializeWeights (randomly)
  start loop
    for each record in trainigDataset do
      // compute the output for this input example
      networkOutput = neuralNetworkOutput (network, record)

      // compute the error
      Err_i = targetOutput − networkOutput

      // update the weights leading to the output layer
      δ_i = Err_i · g'(input_sum_i)
      Δw_ji = α · a_j · δ_i
      new(W_ji) = old(W_ji) + Δw_ji

      for each subsequent-layer in network do
        // compute the error at each node and update weights
        δ_j = g'(input_sum_j) · ∑ W_ij δ_i
        Δw_kj = α · a_k · δ_j
        new(W_kj) = old(W_kj) + Δw_kj
      end for
    end for
  end loop when network has converged
  return network
```

Activation Functions

- We use activation functions to propagate the output of one layer's nodes forward to the next layer (up to and including the output layer).
- Activation functions are a scalar-to-scalar function, yielding the neuron's activation.
- We use activation functions for hidden neurons in a neural network to introduce nonlinearity into the network's modelling capabilities.
- Many activation functions belong to a logistic class of trans- forms that (when graphed) resemble an S. This class of function is called sigmoidal.

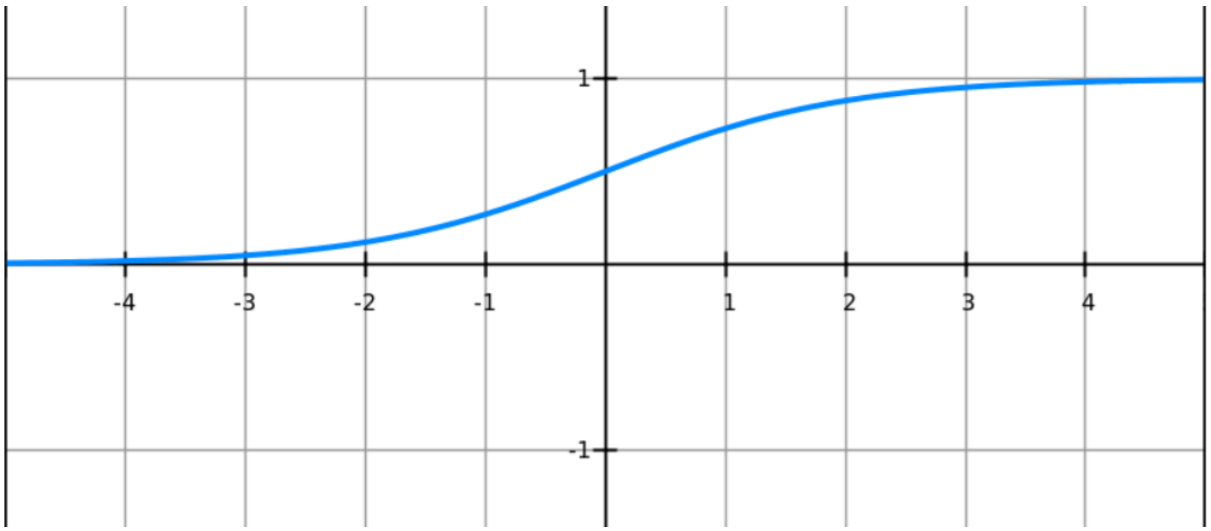
Linear Activation Functions

- A linear transform is **basically the identity function**, and $\boldsymbol{f(x) = Wx}$, where the **dependent variable** has a **direct / proportional relationship with** the **independent variable**.
- In practical terms, it means the function passes the signal through unchanged.
- We see this activation function **used in the input layer** of neural networks.



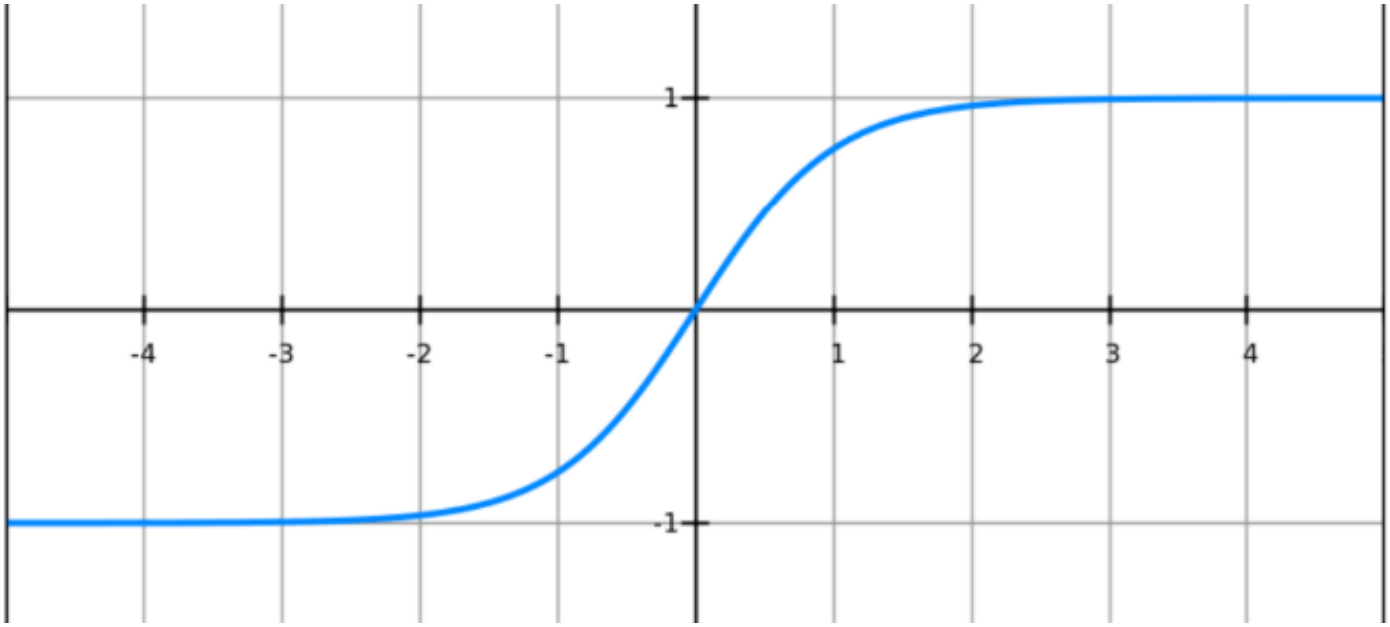
Sigmoidal Activation Functions

- Like all logistic transforms, sigmoids can **reduce extreme values** or **outliers** in data **without removing them**.
- A sigmoid function is a machine that **converts independent variables** of **near infinite range** into **simple probabilities between 0 and 1**, and most of its output will be very close to 0 or 1.



Tanh Activation Function

- It is a **hyperbolic trigonometric function**.
- Just as the tangent represents a ratio between the opposite and adjacent sides of a right triangle, tanh represents the ratio of the hyperbolic sine to the hyperbolic cosine: $tanh(x) = sinh(x)/cosh(x)$.
- Unlike the Sigmoid function, the **normalized range of tanh is –1 to 1**.
- The advantage of tanh is that it can **deal** more **easily with negative numbers**.



Hard Tanh

- Similar to tanh, hard tanh simply applies hard caps to the normalized range. **Anything more than 1 is made into 1**, & **anything less than –1 is made into –1**.
- This allows for a more robust activation function that allows for a limited decision boundary.

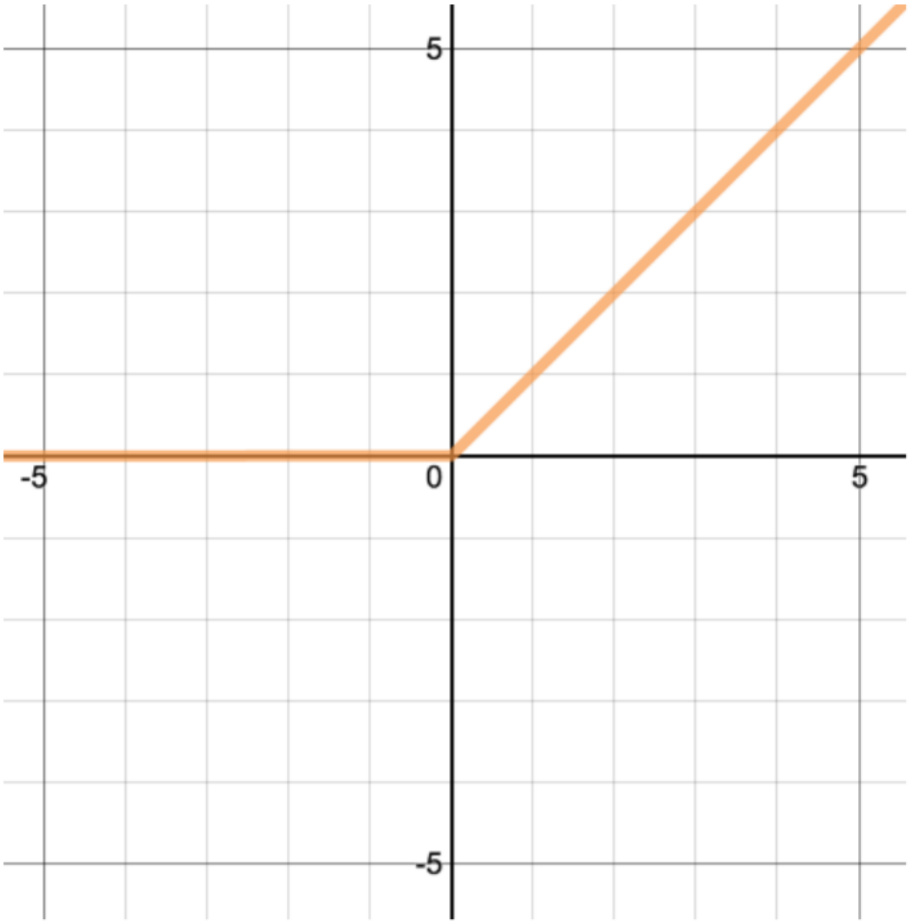
Softmax

- Softmax is a **generalization of logistic regression** since it can be applied to **continuous data** (rather than classifying binary) and can contain multiple decision boundaries.
- It handles **multinomial labelling systems**.
- Softmax is the function you will often **find** at the **output layer of a classifier.**
- The softmax activation function **returns the probability distribution over mutually exclusive output classes**. Given by:

$$f(x_j) = \frac{e^{x_j}}{\sum_{i=1}^{k} e^{x_i}} \quad (where, 1 \leq j \leq k)$$

Rectified Linear Units (ReLU)

- It **activates** a node **only if** the **input** is **above** a **certain quantity**.
- While the input is below zero, the output is zero, but when the input rises above a certain threshold, it has a linear relationship with the dependent variable $\boldsymbol{f(x) = max(0,\ x)}$



- Because the gradient of a ReLU is either zero or a constant, it is possible to **reign in the vanishing exploding gradient issue.**
- ReLU activation functions have **shown to train better** in practice than sigmoid activation functions.

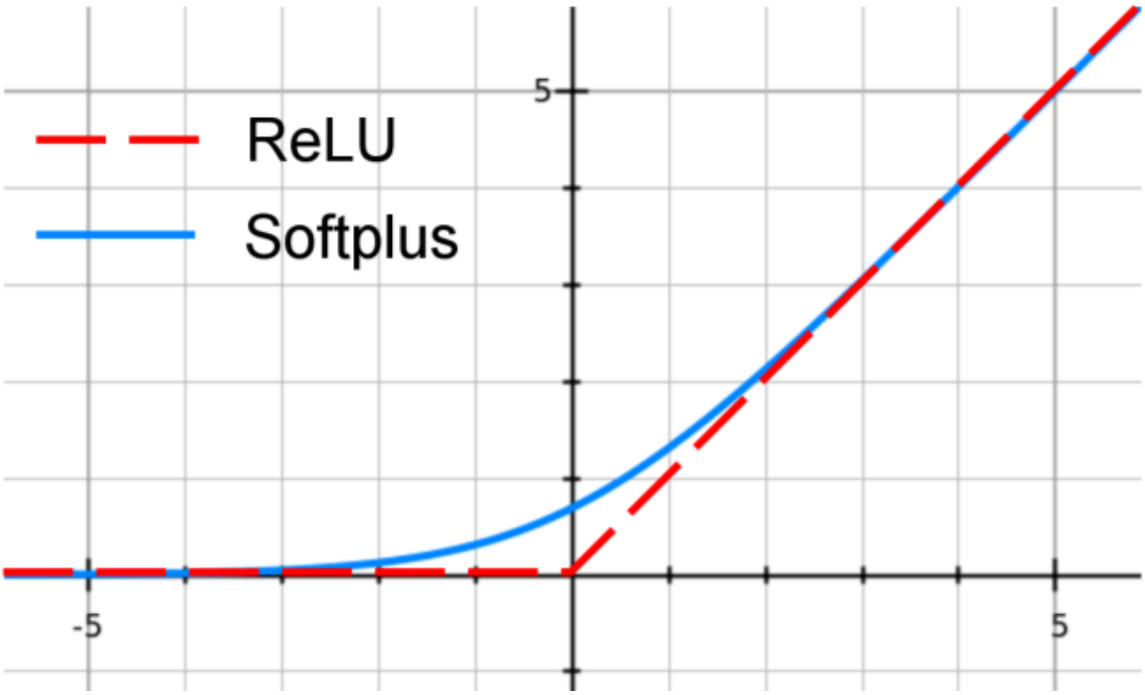
Leaky ReLU

- Leaky ReLUs are a strategy to **mitigate the "dying ReLU" issue**.
- As opposed to having the function being zero **when x < 0**, the leaky ReLU will instead have a **small negative slope (**e.g., "around 0.01").
- The equation is given here:

$$f(x) = \begin{cases} x & if x > 0 \\ 0.01x & otherwise \end{cases}$$

Softplus

- This activation function is considered to be the "smoother version of the ReLU," as is illustrated



- It is given by the following equation:
  $f(x) = \log(1 + e^x)$

Loss Functions

- Loss functions **quantify how close a given neural network is to the ideal** toward which it is training.
- We calculate a metric based on the error we observe in the network's predictions.
- We then aggregate these errors over the entire dataset and average them and now we have a single number representative of how close the neural network is to its ideal.
- Looking for this ideal state is equivalent to finding the parameters (weights and biases) that will minimize the "loss" incurred from the errors.

Loss Functions for Regression

1. Mean Squared Error Loss (MSE)

- When working on a regression model that requires a real valued output, we use the squared loss function.
- Consider the case in which we have to predict only one output feature (M = 1). The error in a prediction is squared and is averaged over the number of data points, as given below:

$$L(w,b) = \frac{1}{N}\sum_{i=1}^{N}(\widehat{Y_i} - Y_i)^2$$

Note:

$N \longrightarrow$ number of training records/samples

$P \longrightarrow$ number of input features

$M \longrightarrow$ number of output features

$(X, Y) \longrightarrow$ (Input data, Output data) pair. There are N such pairs, where input X is a collection of P values and output Y is a collection of M values.

$\hat{Y} \longrightarrow$ output of the neural network having M features

$Y_{i,j} \longrightarrow$ refers to the jth feature observed in the ith sample collected.

- Another variation of the MSE loss function:

$$L(w,b) = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{M}\sum_{j=1}^{M}(\hat{\mathrm{Y}}_{i,j} - Y_{i,j})^2$$

2. Mean Absolute Error Loss (MAE)

- In a similar vein, an alternative to the MSE loss is the mean absolute error (MAE) loss, given by:

$$L(w,b) = \frac{1}{\mathbf{2}N}\sum_{i=1}^{N}\sum_{j=1}^{M}\left|\hat{\mathrm{Y}}_{i,j} - Y_{i,j}\right|$$

3. Mean Squared Log Error Loss (MSLE)

- Another loss function used for regression is the mean squared log error (MSLE):

$$L(w,b) = \frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M}(\log\hat{\mathrm{Y}}_{i,j} - \log Y_{i,j})^2$$

4. Mean Absolute Percentage Error Loss (MAPE)

- Finally, we have mean absolute percentage error (MAPE) loss:

$$L(w,b) = \frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M}\frac{100\ \times\left|\hat{\mathrm{Y}}_{i,j} - Y_{i,j}\right|}{Y_{i,j}}$$

Loss Functions for Classification

1. Hinge Loss

- Hinge loss is the most commonly used loss function when the network must be optimized for a hard classification.
- For example, 0 = no fraud and 1 = fraud, which by convention is called a 0-1 classifier.
- The 0,1 choice is somewhat arbitrary and –1, 1 is also seen in lieu of 0–1.
- Following is the equation for hinge loss when data points must be categorized as –1 or 1:

$$L(w, b) = \frac{1}{N}\sum_{i=1}^{N} \max\left(0, 1 - Y_{i,j} \times \hat{Y}_{i,j}\right)$$

- The hinge loss is mostly used for binary classifications.

2. Logistic Loss

- Logistic loss functions are used when probabilities are of greater interest than hard classifications.
- Predicting valid probabilities means generating numbers between 0 and 1 and making sure the probability of mutually exclusive outcomes should sum to one.
- For this reason, it is essential that the very last layer of a neural network used in classification uses softmax as activation function.
- Based on the notation described earlier, we can express our output for a given input X_i as $h(X_i)$ and $1 - h(X_i)$, given a set of weights and biases, W and b, expressing the probability of 1 and 0, respectively, as shown here:

  $P(y_i = 1 \mid X_i; \mathrm{W}, \mathrm{b}) = h_{\mathrm{W,b}}(X_i)$

  $P(y_i = 0 \mid X_i; \mathrm{W}, \mathrm{b}) = 1 - h_{\mathrm{W,b}}(X_i)$

  We can combine these equations and express them as follows:

  $$P(y_i \mid X_i; \mathrm{W}, \mathrm{b}) = \left(h_{\mathrm{W,b}}(X_i)\right)^{y_i} \times \left(1 - h_{\mathrm{W,b}}(X_i)\right)^{1-y_i}$$

3. Negative Log Likelihood

- For the sake of mathematical convenience, when dealing with the product of probabilities, it is customary to convert them to the log of the probabilities.
- Hence, the product of the probabilities transforms to the sum of the log of the probabilities.

- We also negate the expression so that the equation now corresponds to a "loss." So, the loss function in question becomes the following, commonly referred to as the negative log likelihood:

$$L(W, b) = -\sum_{i=1}^{N} \sum_{j=1}^{M} Y_{i,j} \times \log \hat{Y}_{i,j}$$

- This is mathematically equivalent to what is called the cross-entropy between two probability distributions.

Loss Functions for Reconstruction

- This set of loss functions relates to what is called reconstruction.
- A neural network is trained to recreate its input as closely as possible. So, why is this any different from memorizing the entire dataset?
- The key here is to tweak the scenario so that the network is forced to learn commonalities and features across the dataset.
- In one approach, the number of parameters in the network is constrained such that the network is forced to compress the data and then re-create it.
- Another often-used approach is to corrupt the input with meaningless "noise" and train the network to ignore the noise and learn the data.
- Examples of these kinds of neural nets are Restricted Boltzmann Machines (RBMs), autoencoders, and so on.
- Following is the equation for KL divergence:

$$D_{KL}(Y \,||\, \hat{Y}) = -\sum_{i=1}^{N} Y_i \times \log\left(\frac{Y_i}{\hat{Y}_i}\right)$$

Hyperparameters

- In machine learning, we have both model parameters (weights and biases) and parameters we tune to make networks train better and faster.
- These tuning parameters are called hyperparameters, and they deal with controlling optimization functions and model selection during training with our learning algorithm.
- Hyperparameter selection focuses on ensuring that the model neither underfits nor overfits the training dataset, while learning the structure of the data as quickly as possible.

Learning Rate

- The learning rate affects the amount by which you adjust parameters during optimization in order to minimize the error of neural network's guesses.
- The learning rate determines how much of the error gradient we want to use for the algorithm's next step.
- A large error and steep gradient combine with the learning rate to produce a large step. As we approach minimal error and the gradient flattens out, the step size tends to shorten.
- A large learning rate coefficient (e.g., 1) will make your parameters take leaps, and small ones (e.g., 0.00001) will make it inch along slowly.
- Large leaps will save time initially, but they can be disastrous if they lead us to overshoot our minimum.
- A learning rate too large oversteps the nadir (minimum), making the algorithm bounce back and forth on either side of the minimum without ever coming to rest.
- In contrast, small learning rates should lead you eventually to an error minimum but they can take a very long time and add to the burden of an already computationally intensive process.

Regularization

- Regularization's main purpose is to control overfitting in machine learning.
- It controls the trade-off between finding a good fit and keeping the value of certain feature weights low as the exponents on features/parameters increase.
- Regularization coefficients L1 and L2 help fight overfitting by making certain weights smaller and reducing the model's complexity.
- Smaller-valued weights lead to simpler hypotheses (models), and simpler hypotheses are the most generalizable.
- Unregularized weights with several higher-order polynomials in the feature set tend to overfit the training set.

Momentum

- Momentum helps the learning algorithm get out of spots in the search space where it would otherwise become stuck
- In the errorscape, it helps the updater find the gulleys that lead toward the minima.

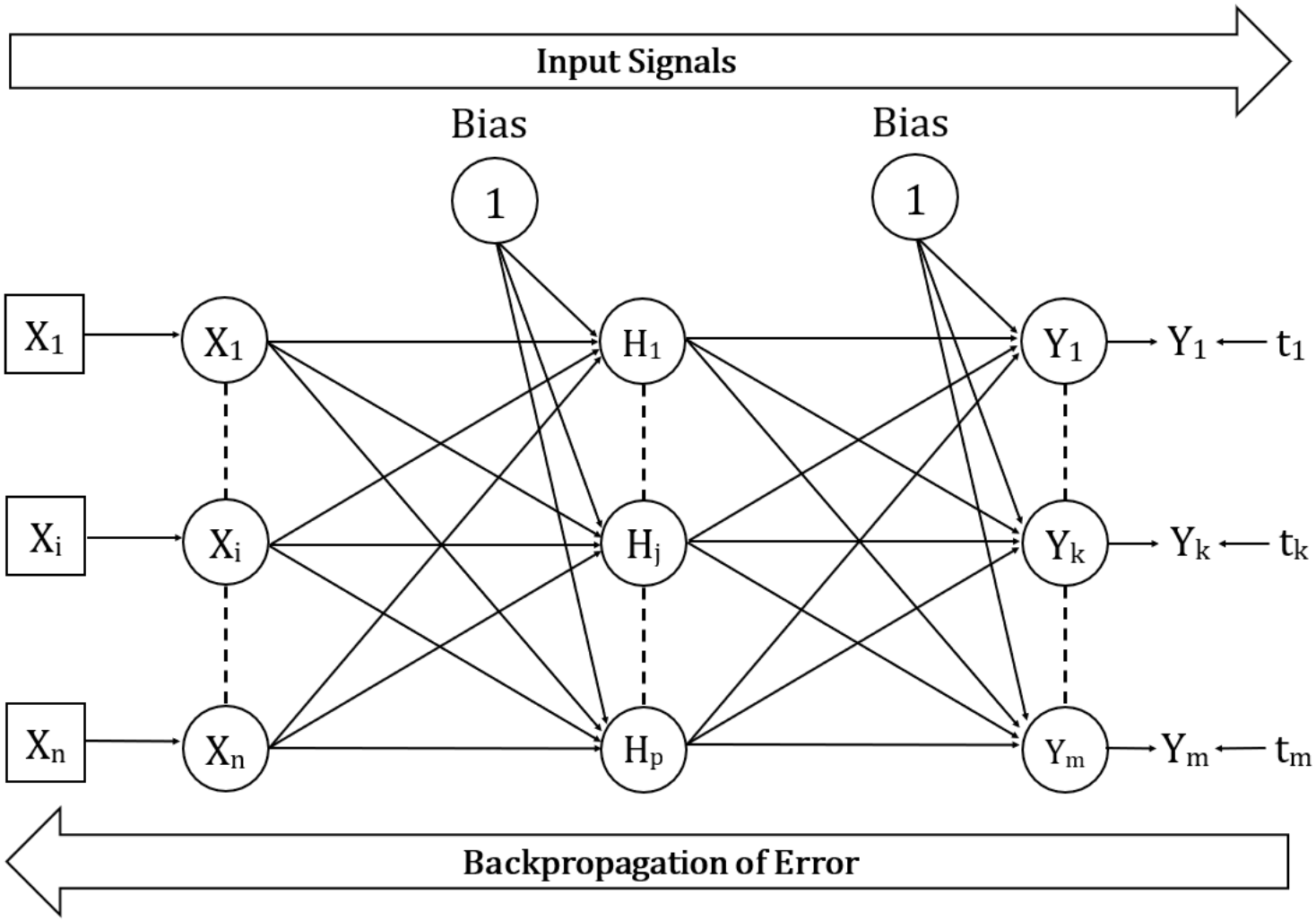
Sparsity

- The sparsity hyperparameter recognizes that for some inputs only a few features are relevant.
- For example, let's assume that a network can classify a million images. Any one of those images will be indicated by a limited number of features.
- But to effectively classify millions of images a network must be able to recognize considerably more features, many of which don't appear most of the time.
- The features that indicate, for example, sea urchins will be few and far between, in the vastness of the neural network's layers.
- That's a problem, because sparse features can limit the number of nodes that activate and impede a network's ability to learn.
- In response to sparsity, biases force neurons to activate and the activations stay around a mean that keeps the network from becoming stuck.

Backpropagation Algorithm

Initialization

- Initialize the following to start the training: Weights and Learning rate (α)
- For easy calculation and simplicity, take some small random values.



Phase 1: Feed Forward Phase

1. Each input unit receives input signal $\mathbf{x_i}$ and sends it to the hidden unit for all $1 \leq i \leq n$
2. Calculate the net input $\boldsymbol{H_{inj}}$ at each hidden unit $\boldsymbol{H_j}$ using the following relation for all $\mathbf{1} \leq \boldsymbol{j} \leq \boldsymbol{p}$

$$H_{inj} = b_j + \sum_{i=1}^{n} x_i \, w_{ij}$$

where, $\boldsymbol{b_j}$ is the bias on hidden unit $\boldsymbol{H_j}$ and $\boldsymbol{w_{ij}}$ is the weight on the link coming from input unit $\boldsymbol{X_i}$ to $\boldsymbol{H_j}$

Now calculate the net output by applying the following activation function:

$$H_j = f(H_{inj})$$

Send these output signals of the hidden layer units to the output layer units.

3. Calculate the net input $\boldsymbol{Y_{ink}}$ at each output unit $\boldsymbol{Y_k}$ using the following relation for all $\boldsymbol{1 \leq k \leq m}$

$$Y_{ink} = b_k + \sum_{j=1}^{p} H_j \, w_{jk}$$

where, $\boldsymbol{b_k}$ is the bias on output unit $\boldsymbol{Y_k}$ and $\boldsymbol{w_{jk}}$ is the weight on the link coming from hidden unit $\boldsymbol{H_j}$ to $\boldsymbol{Y_k}$

Now calculate the net output by applying the following activation function:

$$Y_k = f(Y_{ink})$$

Phase 2: Back Propagation of error

1. Compute the error correcting term, in correspondence with the target pattern received at each output unit, as follows

$$\delta_k = (t_k - Y_k) \, f'(Y_{ink})$$

On this basis, update the deltas of weight and bias of output layer as follows:

$$\Delta w_{jk} = \alpha \, \delta_k \, H_j$$

$$\Delta b_k = \alpha \, \delta_k$$

Then, send $\boldsymbol{\delta_k}$ back to the hidden layer.

2. Now each hidden unit will be the sum of its delta inputs from the output units.

$$\delta_{inj} = \sum_{k=1}^{m} \delta_k \, w_{kj}$$

Error term can be calculated as follows:

$$\delta_j = \delta_{inj} \, f'(H_{inj})$$

On this basis, update the deltas of weight and bias of hidden layer as follows:

$$\Delta w_{ij} = \alpha \, \delta_j \, X_i$$

$$\Delta b_j = \alpha \, \delta_j$$

Phase 3: Updating of weights

1. Each output unit $\boldsymbol{Y_k}$ updates the weight and bias as follows:

$$w_{jk}(new) = w_{jk}(old) + \Delta w_{jk}$$

$$b_k(new) = b_k(old) + \Delta b_k$$

2. Each hidden unit $\boldsymbol{H_j}$ updates the weight and bias as follows:

$$w_{ij}(new) = w_{ij}(old) + \Delta w_{ij}$$

$$b_j(new) = b_j(old) + \Delta b_j$$

Continue the above 3 phases for every training pair if the stopping condition is not true. The stopping condition, may either be the number of epochs reached or if the target output matches the actual output.