Introduction to Artificial Neural Networks

Ali Alilooee

Ohio State University

Presentation Outline

Introduction

- Birds inspired us to fly, burdock plants inspired velcro, and countless more inventions were inspired by nature.

Introduction

- Birds inspired us to fly, burdock plants inspired velcro, and countless more inventions were inspired by nature.
- It seems only logical, then, to look at the brain's architecture for inspiration on how to build an intelligent machine.

Introduction

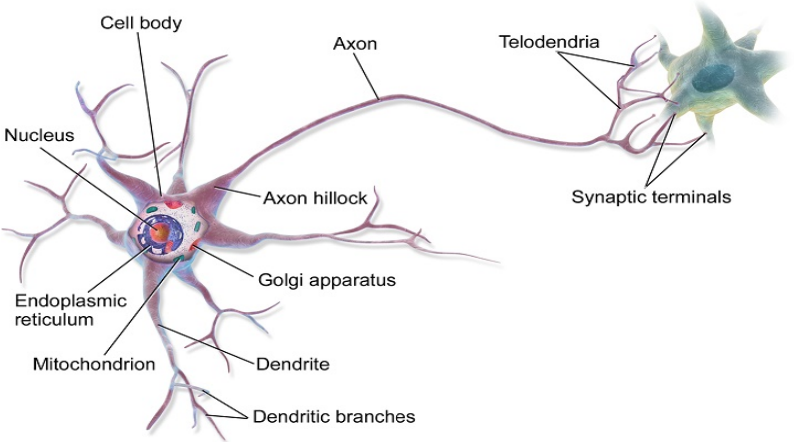
- Birds inspired us to fly, burdock plants inspired velcro, and countless more inventions were inspired by nature.
- It seems only logical, then, to look at the brain's architecture for inspiration on how to build an intelligent machine.
- This is the key idea that sparked **artificial neural networks** (ANNs).

Introduction

- Birds inspired us to fly, burdock plants inspired velcro, and countless more inventions were inspired by nature.
- It seems only logical, then, to look at the brain's architecture for inspiration on how to build an intelligent machine.
- This is the key idea that sparked **artificial neural networks** (ANNs).
- ANNs are at the very core of Deep Learning. They are powerful and scalable, making them ideal for tackling large and highly complex Machine Learning tasks.

Biological neurons

Before we discuss artificial neurons, let's take a quick look at a biological neuron.



The Perceptron

- The Perceptron is one of the simplest ANN architectures, invented in 1957 by Frank Rosenblatt.

The Perceptron

- The Perceptron is one of the simplest ANN architectures, invented in 1957 by Frank Rosenblatt.
- It is based on a slightly different artificial neuron called a threshold logic unit (TLU) or sometimes a linear threshold unit (LTU).

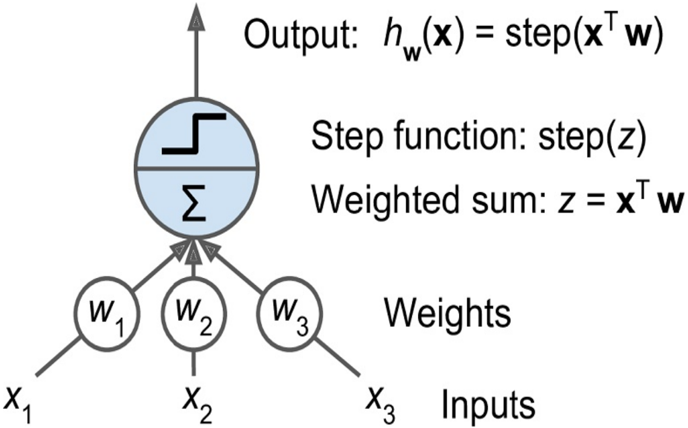
The Perceptron

- The Perceptron is one of the simplest ANN architectures, invented in 1957 by Frank Rosenblatt.
- It is based on a slightly different artificial neuron called a threshold logic unit (TLU) or sometimes a linear threshold unit (LTU).
- The inputs and output are now numbers (instead of binary on/off values), and each input connection is associated with a weight.

The Perceptron

- The Perceptron is one of the simplest ANN architectures, invented in 1957 by Frank Rosenblatt.
- It is based on a slightly different artificial neuron called a threshold logic unit (TLU) or sometimes a linear threshold unit (LTU).
- The inputs and output are now numbers (instead of binary on/off values), and each input connection is associated with a weight.
- The TLU computes a weighted sum of its inputs $z = \omega_1 x_1 + \omega_2 x_2 + \cdots + \omega_n x_n = \boldsymbol{x}^T \cdot \boldsymbol{\omega}$
- then applies a **step function** to that sum and outputs the result: $h_\omega(\boldsymbol{x}) = step(z)$, where $z = \boldsymbol{x}^T \cdot \boldsymbol{\omega}$.

The Perceptron



,

The Perceptron

- A single TLU can be used for simple linear binary classification.

The Perceptron

- A single TLU can be used for simple linear binary classification.
- It computes a linear combination of the inputs, and if the result exceeds a threshold, it outputs the positive class or else outputs the negative class (just like a Logistic Regression classifier or a linear SVM).

The Perceptron

- A single TLU can be used for simple linear binary classification.
- It computes a linear combination of the inputs, and if the result exceeds a threshold, it outputs the positive class or else outputs the negative class (just like a Logistic Regression classifier or a linear SVM).
- Training a TLU, in this case, means finding the right values for weights $\omega_0, \omega_1, \ldots, \omega_n$.

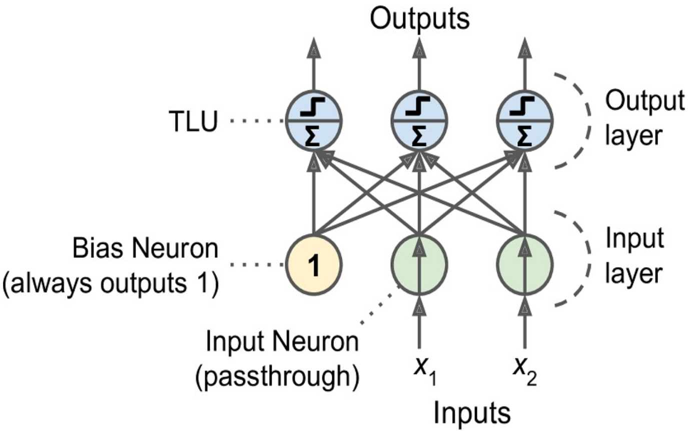
The Perceptron

- A single TLU can be used for simple linear binary classification.
- It computes a linear combination of the inputs, and if the result exceeds a threshold, it outputs the positive class or else outputs the negative class (just like a Logistic Regression classifier or a linear SVM).
- Training a TLU, in this case, means finding the right values for weights $\omega_0, \omega_1, \dots, \omega_n$.
- A Perceptron is simply composed of a single layer of TLUs, with each TLU connected to all the inputs.

The Perceptron

- A single TLU can be used for simple linear binary classification.
- It computes a linear combination of the inputs, and if the result exceeds a threshold, it outputs the positive class or else outputs the negative class (just like a Logistic Regression classifier or a linear SVM).
- Training a TLU, in this case, means finding the right values for weights $\omega_0, \omega_1, \ldots, \omega_n$.
- A Perceptron is simply composed of a single layer of TLUs, with each TLU connected to all the inputs.
- When all the neurons in a layer are connected to every neuron in the previous layer (i.e., its input neurons), it is called a **fully connected layer or a dense layer**.

Fully connected layer



To compute the outputs layer at once by using the following equation

$$h_{\omega,b}(\boldsymbol{X}) = \phi(\boldsymbol{X}\boldsymbol{W} + b)$$

Computing the outputs of a fully connected layer

$$h_{\omega,b}(\boldsymbol{X}) = \phi(\boldsymbol{X}\boldsymbol{W} + b)$$

- As always, $\boldsymbol{X}$ represents the matrix of input features. It has one row per instance and one column per feature.
- The weight matrix $\boldsymbol{W}$ contains all the connection weights except for the ones from the bias neuron. It has one row per input neuron and one column per artificial neuron in the layer.
- The bias vector $\boldsymbol{b}$ contains all the connection weights between the bias neuron and the artificial neurons. It has one bias term per artificial neuron.
- The function ϕ is called the **activation function**: when the artificial neurons are TLUs, it is a step function (but we will discuss other activation functions shortly).

Perceptron learning rule (weight update)

We can use the following formula to update weights

$$\omega_{i,j}^{\text{next}} = \omega_{i,j} + \alpha(y_j - \hat{y}_j)x_i$$

where

- $\omega_{i,j}$ is the connection weight between the ith input neuron and the jth output neuron.
- x_i is the ith input value of the current training instance.
- $\hat{y}_j$ is the output of the jth output neuron for the current training instance.
- y_j is the target output of the jth output neuron for the current training instance.
- α is the learning rate.

Perceptron convergence theorem

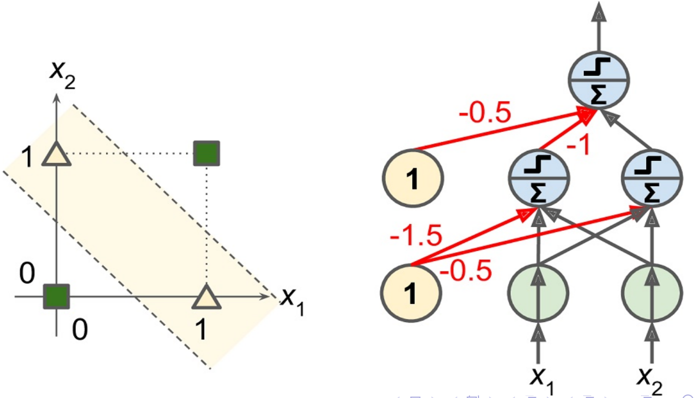
- The decision boundary of each output neuron is linear, so Perceptrons are incapable of learning complex patterns (just like Logistic Regression classifiers).

Perceptron convergence theorem

- The decision boundary of each output neuron is linear, so Perceptrons are incapable of learning complex patterns (just like Logistic Regression classifiers).
- However, if the training instances are linearly separable, Rosenblatt demonstrated that this algorithm would converge to a solution. This is called the **Perceptron convergence theorem**.

Presentation Outline

- In their 1969 monograph titled Perceptrons, Marvin Minsky and Seymour Papert highlighted a number of serious weaknesses of Perceptrons, in particular, the fact that they are incapable of solving some trivial problems (e.g., the Exclusive OR (XOR) classification problem.



MLP

- It turns out that some of the limitations of Perceptrons can be eliminated by stacking multiple Perceptrons.

MLP

- It turns out that some of the limitations of Perceptrons can be eliminated by stacking multiple Perceptrons.
- The resulting ANN is called a **Multi-Layer Perceptron (MLP)**.

MLP

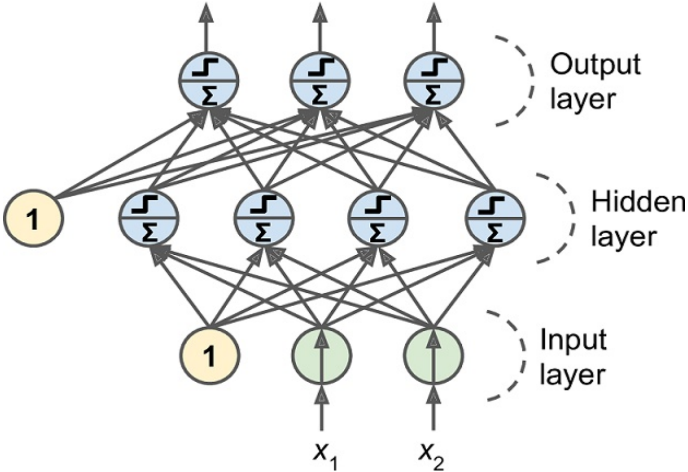
- It turns out that some of the limitations of Perceptrons can be eliminated by stacking multiple Perceptrons.
- The resulting ANN is called a **Multi-Layer Perceptron (MLP)**.
- In particular, an MLP can solve the XOR problem, as you can verify by computing the output of the MLP with inputs $(0, 0)$ or $(1, 1)$, the network outputs 0, and with inputs $(0, 1)$ or $(1, 0)$ it outputs 1.

MLP

- It turns out that some of the limitations of Perceptrons can be eliminated by stacking multiple Perceptrons.
- The resulting ANN is called a **Multi-Layer Perceptron (MLP)**.
- In particular, an MLP can solve the XOR problem, as you can verify by computing the output of the MLP with inputs $(0, 0)$ or $(1, 1)$, the network outputs 0, and with inputs $(0, 1)$ or $(1, 0)$ it outputs 1.
- All connections have a weight equal to 1, except the four connections where the weight is shown.

Layers of an MLP

An MLP is composed of one (passthrough) input layer, one or more layers of TLUs, called hidden layers, and one final layer of TLUs called the output layer.



Backpropagation

- When an ANN contains a deep stack of hidden layers, it is called a deep neural network (DNN). The field of Deep Learning studies DNNs, and, more generally, models containing deep stacks of computations.

Backpropagation

- When an ANN contains a deep stack of hidden layers, it is called a deep neural network (DNN). The field of Deep Learning studies DNNs, and, more generally, models containing deep stacks of computations.
- For many years, researchers struggled to find a way to train MLPs, without success. But in 1986, David Rumelhart, Geoffrey Hinton, and Ronald Williams published a groundbreaking paper introducing the backpropagation training algorithm, which is still used today.

Backpropagation

- When an ANN contains a deep stack of hidden layers, it is called a deep neural network (DNN). The field of Deep Learning studies DNNs, and, more generally, models containing deep stacks of computations.
- For many years, researchers struggled to find a way to train MLPs, without success. But in 1986, David Rumelhart, Geoffrey Hinton, and Ronald Williams published a groundbreaking paper introducing the backpropagation training algorithm, which is still used today.
- In short, it is simply Gradient Descent using an efficient technique for computing the gradients automatically in just two passes through the network (forward and backward).

Backpropagation

- When an ANN contains a deep stack of hidden layers, it is called a deep neural network (DNN). The field of Deep Learning studies DNNs, and, more generally, models containing deep stacks of computations.
- For many years, researchers struggled to find a way to train MLPs, without success. But in 1986, David Rumelhart, Geoffrey Hinton, and Ronald Williams published a groundbreaking paper introducing the backpropagation training algorithm, which is still used today.
- In short, it is simply Gradient Descent using an efficient technique for computing the gradients automatically in just two passes through the network (forward and backward).
- It can find out how each connection weight and each bias term should be tweaked in order to reduce the error.

Backpropagation in a bit more detail

- It handles one mini-batch at a time (for example containing 32 instances each), and it goes through the full training set multiple times. Each pass is called an **epoch**.

Backpropagation in a bit more detail

- It handles one mini-batch at a time (for example containing 32 instances each), and it goes through the full training set multiple times. Each pass is called an **epoch**.
- Each mini-batch is passed to the network's input layer, which just sends it to the first hidden layer. The algorithm then computes the output of all the neurons in this layer.

Backpropagation in a bit more detail

- It handles one mini-batch at a time (for example containing 32 instances each), and it goes through the full training set multiple times. Each pass is called an **epoch**.
- Each mini-batch is passed to the network's input layer, which just sends it to the first hidden layer. The algorithm then computes the output of all the neurons in this layer.
- Then the result is passed on to the next layer, its output is computed and passed to the next layer, and so on, until we get the output of the last layer, the output layer. This is the **forward pass**.

Backpropagation in a bit more detail

- It handles one mini-batch at a time (for example containing 32 instances each), and it goes through the full training set multiple times. Each pass is called an **epoch**.
- Each mini-batch is passed to the network's input layer, which just sends it to the first hidden layer. The algorithm then computes the output of all the neurons in this layer.
- Then the result is passed on to the next layer, its output is computed and passed to the next layer, and so on, until we get the output of the last layer, the output layer. This is the **forward pass**.
- Next, the algorithm measures the network's output error (i.e., it uses a loss function that compares the desired output and the actual output).

Backpropagation in a bit more detail

- It handles one mini-batch at a time (for example containing 32 instances each), and it goes through the full training set multiple times. Each pass is called an **epoch**.
- Each mini-batch is passed to the network's input layer, which just sends it to the first hidden layer. The algorithm then computes the output of all the neurons in this layer.
- Then the result is passed on to the next layer, its output is computed and passed to the next layer, and so on, until we get the output of the last layer, the output layer. This is the **forward pass**.
- Next, the algorithm measures the network's output error (i.e., it uses a loss function that compares the desired output and the actual output).
- Then it computes how much each output connection contributed to the error. This is done analytically by simply applying the chain rule from Calculus.

Backpropagation in a bit more detail cont.

- The algorithm then measures how much of these error contributions came from each connection in the layer below, again using the chain rule—and so on until the algorithm reaches the input layer.

Backpropagation in a bit more detail cont.

- The algorithm then measures how much of these error contributions came from each connection in the layer below, again using the chain rule—and so on until the algorithm reaches the input layer.
- As we explained earlier, this reverse pass efficiently measures the error gradient across all the connection weights in the network by propagating the error **gradient backward** through the network (hence the name of the algorithm).

Backpropagation in a bit more detail cont.

- The algorithm then measures how much of these error contributions came from each connection in the layer below, again using the chain rule—and so on until the algorithm reaches the input layer.
- As we explained earlier, this reverse pass efficiently measures the error gradient across all the connection weights in the network by propagating the error **gradient backward** through the network (hence the name of the algorithm).
- Finally, the algorithm performs a Gradient Descent step to tweak all the connection weights in the network, using the error gradients it just computed.

Summarizing backpropagation

1. For each training instance, the backpropagation algorithm first makes a prediction (**forward pass**).

Summarizing backpropagation

1. For each training instance, the backpropagation algorithm first makes a prediction (**forward pass**).
2. Measures the error, then goes through each layer in reverse to measure the error contribution from each connection (**reverse pass**),

Summarizing backpropagation

1. For each training instance, the backpropagation algorithm first makes a prediction (**forward pass**).
2. Measures the error, then goes through each layer in reverse to measure the error contribution from each connection (**reverse pass**),
3. Finally, slightly tweaks the connection weights to reduce the error (**Gradient Descent step**).

Initialize the weights and biases

- It is important to initialize all the hidden layers' connection weights randomly, or else training will fail.

Initialize the weights and biases

- It is important to initialize all the hidden layers' connection weights randomly, or else training will fail.
- For example, if you initialize all weights and biases to zero, then all neurons in a given layer will be perfectly identical, and thus backpropagation will affect them in exactly the same way, so they will remain identical.

Initialize the weights and biases

- It is important to initialize all the hidden layers' connection weights randomly, or else training will fail.
- For example, if you initialize all weights and biases to zero, then all neurons in a given layer will be perfectly identical, and thus backpropagation will affect them in exactly the same way, so they will remain identical.
- In other words, despite having hundreds of neurons per layer, your model will act as if it had only one neuron per layer: it won't be too smart.

Initialize the weights and biases

- It is important to initialize all the hidden layers' connection weights randomly, or else training will fail.
- For example, if you initialize all weights and biases to zero, then all neurons in a given layer will be perfectly identical, and thus backpropagation will affect them in exactly the same way, so they will remain identical.
- In other words, despite having hundreds of neurons per layer, your model will act as if it had only one neuron per layer: it won't be too smart.
- If instead you randomly initialize the weights, you **break the symmetry** and allow backpropagation to train a diverse team of neurons.

Some activation functions

In MLP's architecture: we replaced the step function with one of the following activation functions:

- **logistic function (Sigmoid)**

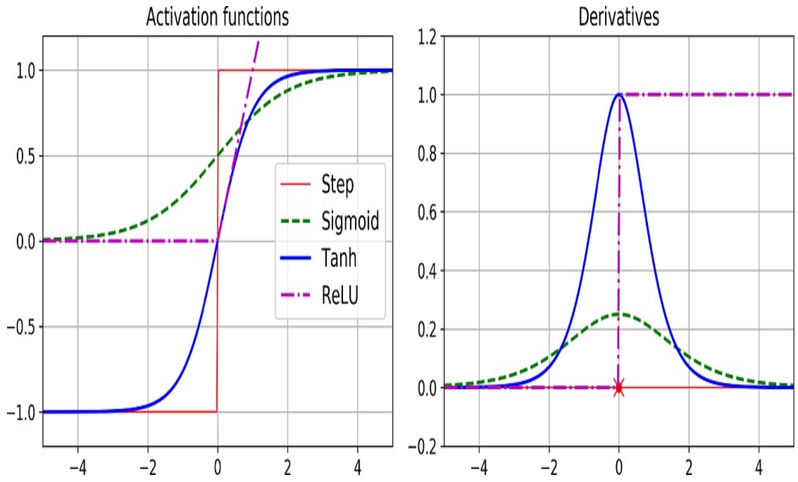
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

- **hyperbolic tangent function**

$$\tanh(z) = \frac{1 - e^{z}}{1 + e^{z}}$$

- **The Rectified Linear Unit function**:

$$\mathrm{ReLU}(z) = \max(0, z) = \begin{cases} z & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases}$$


Activation functions
Derivatives
Step
Sigmoid
Tanh
ReLU
1.0
0.5
0.0
−0.5
−1.0
1.2
0.8
0.6
0.4
0.2
−0.2
−4
−2
0
2
4

,

Presentation Outline

Regression MLPs

- MLPs can be used for regression tasks. If you want to predict a single value (e.g., price of a house, given many of its features), then you just need a single output neuron: its output is the predicted value.

Regression MLPs

- MLPs can be used for regression tasks. If you want to predict a single value (e.g., price of a house, given many of its features), then you just need a single output neuron: its output is the predicted value.
- For multivariate regression (i.e., to predict multiple values at once), you need one output neuron per output dimension.

Regression MLPs

- MLPs can be used for regression tasks. If you want to predict a single value (e.g., price of a house, given many of its features), then you just need a single output neuron: its output is the predicted value.
- For multivariate regression (i.e., to predict multiple values at once), you need one output neuron per output dimension.
- For example, to locate the center of an object on an image, you need to predict 2D coordinates, so you need two output neurons.

Regression MLPs

- MLPs can be used for regression tasks. If you want to predict a single value (e.g., price of a house, given many of its features), then you just need a single output neuron: its output is the predicted value.
- For multivariate regression (i.e., to predict multiple values at once), you need one output neuron per output dimension.
- For example, to locate the center of an object on an image, you need to predict 2D coordinates, so you need two output neurons.
- If you also want to place a bounding box around the object, then you need two more numbers: the width and the height of the object. So you end up with 4 output neurons.

More on regression MLPs

- In general, when building an MLP for regression, you do not want to use any activation function for the output neurons, so they are free to output any range of values.

More on regression MLPs

- In general, when building an MLP for regression, you do not want to use any activation function for the output neurons, so they are free to output any range of values.
- However, if you want to guarantee that the output will always be positive, then you can use the ReLU activation function or the soft plus activation function in the output layer.

More on regression MLPs

- In general, when building an MLP for regression, you do not want to use any activation function for the output neurons, so they are free to output any range of values.
- However, if you want to guarantee that the output will always be positive, then you can use the ReLU activation function or the soft plus activation function in the output layer.
- Finally, if you want to guarantee that the predictions will fall within a given range of values, then you can use the logistic function or the hyperbolic tangent, and scale the labels to the appropriate range: 0 to 1 for the logistic function or -1 to 1 for the hyperbolic tangent.

Loss function for regression MLPs

- The loss function to use during training is typically the mean squared error, but if you have a lot of outliers in the training set, you may prefer to use the mean absolute error instead. Alternatively, you can use the **Huber loss**, which is a combination of both.

Loss function for regression MLPs

- The loss function to use during training is typically the mean squared error, but if you have a lot of outliers in the training set, you may prefer to use the mean absolute error instead. Alternatively, you can use the **Huber loss**, which is a combination of both.
- The Huber loss is quadratic when the error is smaller than a threshold σ (typically 1), but linear when the error is larger than . This makes it less sensitive to outliers than the mean squared error, and it is often more precise and converges faster than the mean absolute error.

Typical regression MLP architecture

Hyperparameter	Typical Value
# Input neurons	One per input feature
# Hidden layers	Depend on the problem Typically 1 to 5.
# Neurons per hidden layer	Depends on problem. Typically 10 to 100.
# output neurons	1 per prediction dimension
Hidden activation	ReLU
Output activation	None or ReLU/Softplus (if positive outputs) or Logistic/Tanh (if bounded outputs)
Loss function	MSE or MAE/Huber

Typical classification MLP architecture

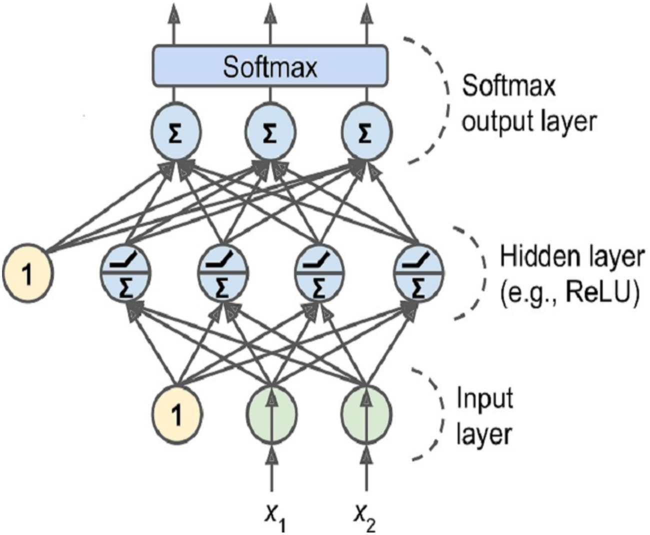
Hyperparameter	Binary classification
Input and hidden layers	Same as regression
# output neurons	1
Output layer activation	Logistic
Loss function	Cross-Entropy

Typical classification MLP architecture

Hyperparameter	Binary classification
Input and hidden layers	Same as regression
# output neurons	1
Output layer activation	Logistic
Loss function	Cross-Entropy

Hyperparameter	Multiclass classification
Input and hidden layers	Same as regression
# output neurons	1 per class
Output layer activation	Softmax
Loss function	Cross-Entropy

Typical classification MLP architecture (Multiclass classification)



,

Softmax

- Softmax is an activation function that scales numbers/logits into probabilities.

Softmax

- Softmax is an activation function that scales numbers/logits into probabilities.
- The output of a Softmax is a vector (say v) with probabilities of each possible outcome. The probabilities in vector v sum to one for all possible outcomes or classes.

Softmax

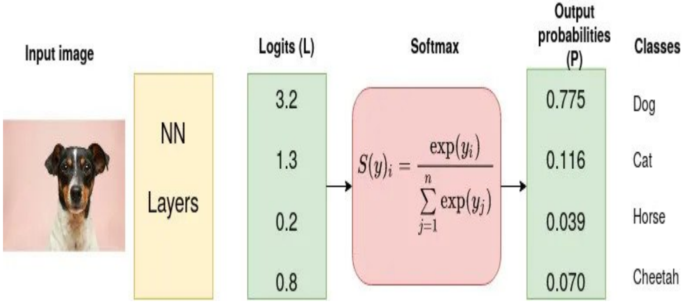
- Softmax is an activation function that scales numbers/logits into probabilities.
- The output of a Softmax is a vector (say v) with probabilities of each possible outcome. The probabilities in vector v sum to one for all possible outcomes or classes.
- Mathematically, Softmax is defined as,

$$\text{Softmax}(y)_i = \frac{\exp(y_i)}{\sum_{j=1}^{n} \exp(y_j)}$$

when $y = (y_1, y_2, \ldots, y_n)$ is the input function consisting of n elements for n classes.

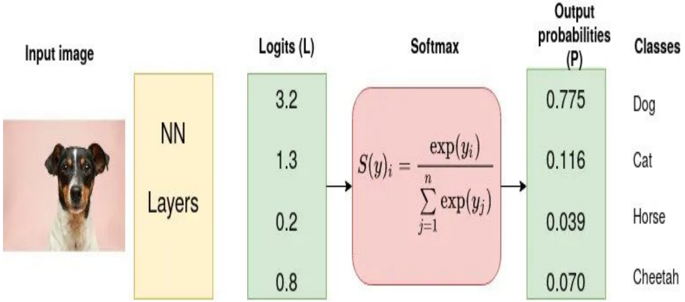
Cross-entropy loss function

Consider a 4-class classification task where an image is classified as either a dog, cat, horse, or cheetah.



Cross-entropy loss function

Consider a 4-class classification task where an image is classified as either a dog, cat, horse, or cheetah.



In the above Figure, Softmax converts logits into probabilities. The purpose of the Cross-Entropy is to take the output probabilities (P) and measure the distance from the truth values (as shown in Figure below).

Cross-entropy

- For $p(x)$ — probability distribution and a random variable X, entropy is defined as follows

$$H(x) = -\sum_{x \in X} p(x) \log (p(x))$$

Cross-entropy

- For $p(x)$ — probability distribution and a random variable X, entropy is defined as follows

$$H(x) = -\sum_{x \in X} p(x) \log\left(p(x)\right)$$

- Cross-entropy is defined as

$$L_{CE} = -\sum_{i=1}^{n} t_i \log\left(p_i\right)$$

where t_i is the truth label and p_i is the softmax probability for the i^{th} class.