

## Neural networks

- Layers and neurons

- Activation functions

  - Softmax function

- Multi-layer neural networks

  - Training a multi-layer neural network

    - Error functions

      - Real-valued output  $y^{(i)} \in \mathbb{R}$

      - Binary output  $y^{(i)} \in \{0, 1\}$

      - Multinomial output  $y^{(i)} \in \{1, \dots, K\}$

    - Using the error functions in backpropagation

- Feedforward neural networks

## Resources

# Neural networks

**Artificial neural networks** are powerful and versatile machine learning systems that are perhaps the most commonly used learning method for many modern machine learning tasks.

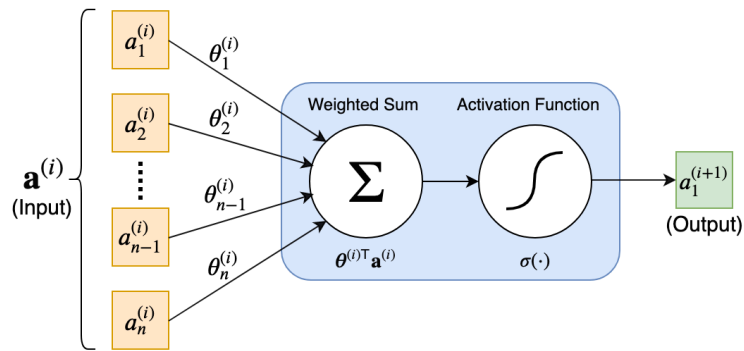
Neural networks are often used for machine learning tasks where not much meaning or knowledge can be derived from the individual features of the training data. Instead, neural networks rely on a system of nodes called **neurons**, which are arranged in collections called **layers**, where the neurons in adjacent layers are connected. These connections have varied strengths, determined by a scalar referred to as a **weight**.

The purpose of the neurons on different layers is to learn the underlying (mostly non-linear) relationships between the original features and the output. Nevertheless, neural networks are often still treated as black boxes when compared to other machine learning models, since it isn't always straightforward to understand what the individual layers and neurons represent.

## Layers and neurons

A neural network may consist of many **layers**. Namely, there are three distinct types of layers:

- **Input layer**:
  - Consists of the features of the input vector  $\mathbf{x}^{(i)}$ , or transformed feature vector  $\phi(\mathbf{x}^{(i)})$ . This vector is also sometimes denoted as  $\mathbf{a}^{(1)}$  for consistency, as the notation  $\mathbf{a}^{(i)}$  in neural networks represents the output vector from the neurons in layer  $i$ , where indexing starts from 1, with the input layer.
  - This layer **does not** pass the features through a weighted sum or activation function. The inputs are simply passed on to the next layer, with the corresponding weights  $\theta^{(1)}$  for this layer.
  - There may only be one input layer in a neural network.
- **Hidden layer**:
  - Consists of **neurons**, each made up of two functions: a **weighted sum** and an **activation function**.



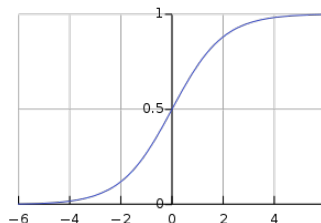
**Figure 1:** An artificial neuron (depicted in blue).

1. The **weighted sum** is computed from the inputs  $\mathbf{a}^{(i)}$ , which are the outputs from the previous layer (along with their corresponding weights  $\theta^{(i)}$ ). This is calculated as the dot product between these two vectors.

Observe that this dot product may be an arbitrary real-valued number, depending on the weights—therefore, this result is passed through the activation function to transform it.

2. The purpose of an **activation function** is to introduce non-linearity to a neural network—it transforms the value of the weighted sum and is chosen specifically for the type of task being performed.

**Example:** In the case of logistic regression, we saw that the logistic function  $\sigma(x) = \frac{1}{1+e^{-x}}$  can be used to shrink  $\mathbb{R}$  into the range  $(0, 1)$ . This is an example of a commonly used activation function.



**Figure 2:** The logistic sigmoid activation function. (source)

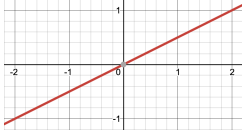
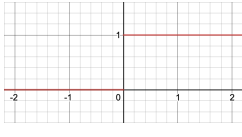
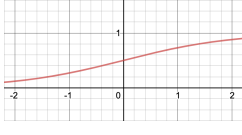
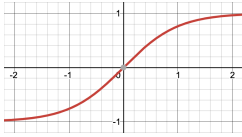
There are many other activation functions which will be described later.

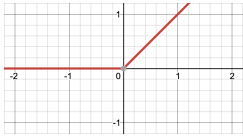
- The output of one neuron is a **single value**, which is sent to each of the neurons on the next layer with new weights  $\theta^{(i+1)}$ .
- The outputs of the neurons in the final hidden layer are passed to the neurons in the **output layer**.
- **Output layer:**
  - Functions the same way as a hidden layer—each output takes a weighted sum of its inputs and passes it through an activation function.
  - The number of output neurons is often the number of classes (in the case of a classification problem). However, in the case of binary classification, one output neuron is enough since we can represent the probability of the other class as the complement of the probability that is output by the neuron.

**Note:** When we say a neural network consists of  $n$  layers,  $n$  is the sum of hidden layers and the output layer.

# Activation functions

As explained previously, the purpose of an **activation function** is to introduce non-linearity to a neural network—it transforms the value of the weighted sum and is chosen specifically for the type of task being performed. Below is a list of some commonly used activation functions:

Name	Function	Plot	Comments
Linear function	$\sigma_{\text{Linear}}(x; k) = kx$		
Threshold (step) function	$\sigma_{\text{Step}}(x; k) = \begin{cases} 1 & x \geq k \\ 0 & x < k \end{cases}$		<ul style="list-style-type: none"><li>• Can be used to make a decision between binary classes 0 and 1, given a real-valued argument. For example, since <math>\sigma_{\text{Logistic}}</math> yields a probability (where we would have to use <math>\arg \max</math> to determine the class), we can determine the class by using <math>\sigma_{\text{Step}}(x; k = \frac{1}{2})</math> in the output layer.</li></ul>
Logistic (sigmoid) function	$\sigma_{\text{Logistic}}(x) = \frac{1}{1+e^{-x}}$		<ul style="list-style-type: none"><li>• Typically used in the output layer of a neural network performing binary classification.</li><li>• Suffers from the issue of vanishing gradients as <math> x </math> increases.</li><li>• Differentiable across entire domain.</li></ul>
Hyperbolic tangent function	$\sigma_{\tanh}(x) = \tanh(x)$		<ul style="list-style-type: none"><li>• Can mathematically be represented as a shifted and scaled version of <math>\sigma_{\text{Logistic}}</math> and is similarly used for binary classification tasks.</li><li>• However, this cannot be directly used for negative log likelihood cost functions due to the</li></ul>

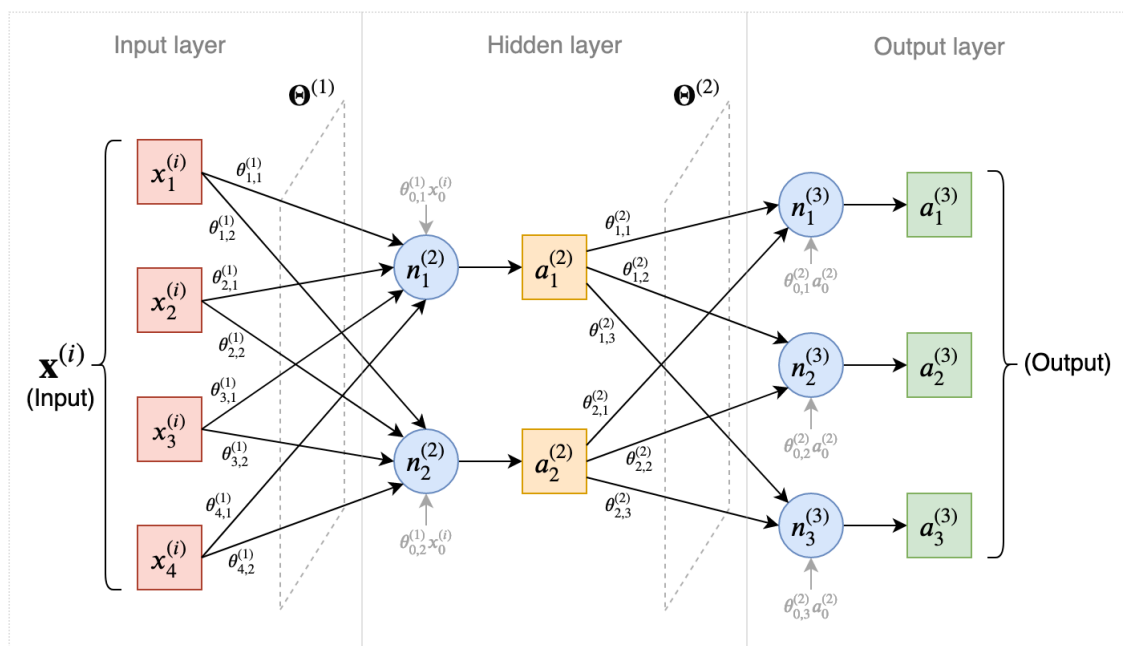
			<p>function range being <math>(-1, 1)</math>.</p> <ul style="list-style-type: none"> <li>Suffers from the same vanishing gradient issue as <math>\sigma_{\text{Logistic}}</math>.</li> </ul>
Rectified linear unit (ReLU)	$\sigma_{\text{ReLU}}(x) = \max(0, x)$		<p>ReLU is generally preferred to sigmoidal activation functions such as <math>\sigma_{\text{Logistic}}</math> and <math>\sigma_{\text{tanh}}</math> in deeper neural networks, mainly because:</p> <ul style="list-style-type: none"> <li>its simpler gradients (during backpropagation) allow for each layer to be trained quicker,</li> <li>it does not suffer from the vanishing gradient problem.</li> </ul>

## Softmax function

Some special comments about softmax

## Multi-layer neural networks

A **multi-layer neural network** is a neural network that contains at least one hidden layer.



**Figure 3:** A multi-layer neural network consisting of one hidden layer with two neurons.

As seen in *Figure 3*, each additional layer requires connections from the neurons in the layer to the neurons in the next layer, as well as the neurons in the previous layer. The connections between each layer depend on the outputs from the previous layer, and some weights for each neuron in the previous layer.

As a result, we require more structures and indexing to formally represent a neural network. Using *Figure 3* as an example:

- $\mathbf{x}^{(i)}$  represents an arbitrary input feature vector with a bias term  $x_0^{(i)}$ .

$$\mathbf{x}^{(i)} = \left( x_0^{(i)}, \dots, x_D^{(i)} \right)^T$$

**Note:**  $\mathbf{x}^{(i)}$  is also treated as the first output—the output of the input layer,  $\mathbf{a}^{(1)}$ .

- $\Theta^{(i)}$  is the **weight matrix** representing the weights of the connections between the neurons in layer  $i$  and the neurons in layer  $i + 1$ .

$\Theta^{(i)}$	$\theta_{0,\cdot}^{(i)}$	$\theta_{1,\cdot}^{(i)}$	$\theta_{2,\cdot}^{(i)}$	$\dots$	$\theta_{N_i,\cdot}^{(i)}$
$\theta_{\cdot,1}^{(i)T}$	$\theta_{0,1}^{(i)}$	$\theta_{1,1}^{(i)}$	$\theta_{2,1}^{(i)}$	$\dots$	$\theta_{N_i,1}^{(i)}$
$\theta_{\cdot,2}^{(i)T}$	$\theta_{0,2}^{(i)}$	$\theta_{1,2}^{(i)}$	$\theta_{2,2}^{(i)}$	$\dots$	$\theta_{N_i,2}^{(i)}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$\theta_{\cdot,N_{i+1}}^{(i)T}$	$\theta_{0,N_{i+1}}^{(i)}$	$\theta_{1,N_{i+1}}^{(i)}$	$\theta_{2,N_{i+1}}^{(i)}$	$\dots$	$\theta_{N_i,N_{i+1}}^{(i)}$

**Where:**

- $N_i$  is the number of neurons in layer  $i$ .
- $\theta_{j,k}^{(i)}$  represents the weight from neuron  $j$  in layer  $i$ , to neuron  $k$  in layer  $i + 1$ .
- $\theta_{\cdot,k}^{(i)} = \left( \theta_{0,k}^{(i)}, \dots, \theta_{N_i,k}^{(i)} \right)^T$  represents the weights from each neuron in layer  $i$ , to neuron  $k$  in layer  $i + 1$ .
- $\theta_{j,\cdot}^{(i)} = \left( \theta_{j,1}^{(i)}, \dots, \theta_{j,N_{i+1}}^{(i)} \right)^T$  represents the weights from neuron  $j$  in layer  $i$ , to each neuron in layer  $i + 1$ .

Observe that there is no connection to the bias unit of the next layer—this bias unit is introduced independently and is not affected by the activations of the neurons from the previous layer. As a result, this weight matrix has dimensions:

$$\dim(\Theta^{(i)}) = N_{i+1} \times (N_i + 1)$$

- $\mathbf{a}^{(i)}$  represents a vector consisting of the outputs from the neurons in layer  $i$ .

$$\mathbf{a}^{(i)} = \left( a_1^{(i)}, \dots, a_{N_i}^{(i)} \right)^T$$

- $\mathbf{n}^{(i)}$  represents the vector of neurons in layer  $i$ .

$$\mathbf{n}^{(i)} = \left( n_1^{(i)}, \dots, n_{N_i}^{(i)} \right)^T$$

**Where:** Each neuron  $n_j^{(i)}$  comprises an activation function  $\sigma_j^{(i)}$  such that the output  $a_j^{(i)}$  of this neuron is given by:

$$a_j^{(i)} = \sigma_j^{(i)} \left( \theta_{\cdot,j}^{(i-1)} \mathbf{a}^{(i-1)} \right)$$

That is, the dot product between the weights from each neuron in layer  $i - 1$  to neuron  $j$  in layer  $i$ , and the outputs of the neurons in the previous layer, applied to the activation function  $\sigma_j^{(i)}$ .

However, it is usually the case that all of the sigmoid functions in one layer are the same—so we can just refer to this as  $\sigma^{(i)}$ .

It is also possible to compute these activations all at once, using matrix multiplication of the weight matrix  $\Theta^{(i-1)}$  with the outputs of the previous layer, and a vector-valued activation function  $\sigma^{(i)}$ :

$$\begin{aligned}\mathbf{a}^{(i)} &= \sigma^{(i)} \left( \Theta^{(i-1)} \mathbf{a}^{(i-1)} \right)^T \\ &= \left( \underbrace{\sigma^{(i)} \left( \theta_{\cdot,1}^{(i-1)} \mathbf{a}^{(i-1)} \right)}_{a_1^{(i)}}, \dots, \underbrace{\sigma^{(i)} \left( \theta_{\cdot, N_i}^{(i-1)} \mathbf{a}^{(i-1)} \right)}_{a_{N_i}^{(i)}} \right)^T\end{aligned}$$

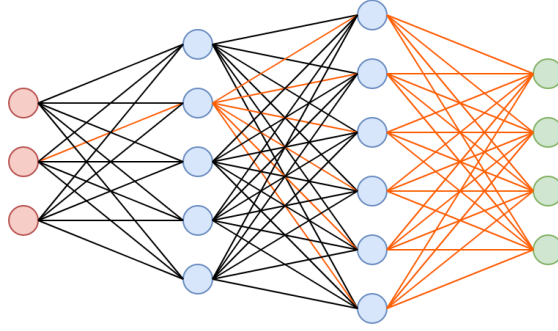
As an example of these forms of representation, the multi-layer neural network in *Figure 3* can be represented by the following matrices and vectors:

$$\begin{aligned}\mathbf{a}^{(1)} = \mathbf{x}^{(i)} &= \begin{pmatrix} x_0^{(i)} \\ x_1^{(i)} \\ x_2^{(i)} \\ x_3^{(i)} \\ x_4^{(i)} \end{pmatrix}, \quad \Theta^{(1)} = \begin{pmatrix} \theta_{0,1}^{(1)} & \theta_{1,1}^{(1)} & \theta_{2,1}^{(1)} & \theta_{3,1}^{(1)} & \theta_{4,1}^{(1)} \\ \theta_{0,2}^{(1)} & \theta_{1,2}^{(1)} & \theta_{2,2}^{(1)} & \theta_{3,2}^{(1)} & \theta_{4,2}^{(1)} \end{pmatrix} \\ \mathbf{a}^{(2)} &= \begin{pmatrix} a_0^{(2)} \\ \sigma^{(2)} \left( \theta_{\cdot,1}^{(1)} \mathbf{a}^{(1)} \right) \\ \sigma^{(2)} \left( \theta_{\cdot,2}^{(1)} \mathbf{a}^{(1)} \right) \end{pmatrix}, \quad \Theta^{(2)} = \begin{pmatrix} \theta_{0,1}^{(2)} & \theta_{1,1}^{(2)} & \theta_{2,1}^{(2)} \\ \theta_{0,2}^{(2)} & \theta_{1,2}^{(2)} & \theta_{2,2}^{(2)} \\ \theta_{0,3}^{(2)} & \theta_{1,3}^{(2)} & \theta_{2,3}^{(2)} \end{pmatrix} \\ \mathbf{a}^{(3)} &= \begin{pmatrix} \sigma^{(3)} \left( \theta_{\cdot,1}^{(2)} \mathbf{a}^{(2)} \right) \\ \sigma^{(3)} \left( \theta_{\cdot,2}^{(2)} \mathbf{a}^{(2)} \right) \\ \sigma^{(3)} \left( \theta_{\cdot,3}^{(2)} \mathbf{a}^{(2)} \right) \end{pmatrix}\end{aligned}$$

## Training a multi-layer neural network

As with other machine learning algorithms such as linear and logistic regression, in order to train a neural network we need to find the weights that minimize some error function. However, this is more of a challenge in neural networks for three reasons:

1. There are far more weights to optimize—with one entire weight matrix  $\Theta^{(i)}$  between layers  $i$  and  $i + 1$ .
2. The change of a single weight connecting a neuron  $n_j^{(i)}$  in layer  $i$  to  $n_k^{(i+1)}$  in layer  $i + 1$  will lead to all of the neurons in subsequent layers being affected due to propagation, as seen in *Figure 4* below.



**Figure 4:** The propagating effect of modifying one weight, on subsequent layers.

3. The error functions used in neural networks are non-convex, meaning that optimization methods are not guaranteed to converge to a global minimum.

Sometimes however, converging to a local minimum may be good enough and produce weights that the neural network can perform well with.

## Error functions

The purpose of an **error function** is to evaluate the performance of the neural network when given some training examples. As neural networks are mostly commonly used in the supervised learning setting, we have access to the actual outputs of the training examples.

The error function  $C(\theta)$  is a measure of the inconsistency between the predicted outputs  $\hat{y}^{(i)}$  and the actual outputs  $y^{(i)}$  for a set of training examples. Therefore, a machine learning model's robustness and performance increases as the error function decreases. The optimal weights are thus the ones that minimize the error function  $C(\theta)$ :

$$\hat{\theta} = \arg \min_{\theta} C(\theta)$$

Neural networks may be used to predict the value of  $y^{(i)} \in \mathbb{R}$  (like linear regression),  $y^{(i)} \in \{0, 1\}$  (binary classification), or  $y^{(i)} \in \{1, \dots, K\}$  (multi-class/multinomial classification). As a result, the error function used in a neural network must be chosen depending on the nature of the output variable.

### Real-valued output $y^{(i)} \in \mathbb{R}$

Similarly to linear regression, the **residual sum of squares (RSS)** error function can be used for neural networks.

$$\begin{aligned} C(\theta) &= \sum_{i=1}^N \left( y^{(i)} - \hat{y}^{(i)} \right)^2 \\ &= \sum_{i=1}^N \left( y^{(i)} - \theta^T \mathbf{x}^{(i)} \right)^2 \end{aligned}$$

However for the back-propagation training algorithm, it is normally preferable for the error function to be represented as a mean of the samples. For this reason, **mean squared error (MSE)** is used more often. Additionally, this quantity is scaled by  $\frac{1}{2}$  to make derivatives cleaner:

$$\begin{aligned}
C(\boldsymbol{\theta}) &= \frac{1}{2N} \sum_{i=1}^N \left( y^{(i)} - \hat{y}^{(i)} \right)^2 \\
&= \frac{1}{2N} \sum_{i=1}^N \left( y^{(i)} - \boldsymbol{\theta}^T \mathbf{x}^{(i)} \right)^2
\end{aligned}$$

**Binary output**  $y^{(i)} \in \{0, 1\}$

As we have seen in logistic regression for binary classification, the output variable can be modeled by the Bernoulli distribution. As a result, the most appropriate cost function to use in this case was the **negative log-likelihood**. Minimizing this would yield the maximum likelihood estimate for the training data,  $\hat{\boldsymbol{\theta}}$ .

For neural networks, we typically use **cross entropy**. Cross entropy is a measure of dissimilarity between two discrete probability distributions. As this is a binary classification problem, we have a Bernoulli-distributed output variable for each sample:

$$\text{Output/Predicted: } \hat{y}^{(i)} \sim \text{Bernoulli} \left( p_1^{(i)} \right)$$

**Where:**

- $p_1^{(i)} = \mathbb{P} \left( \hat{y}^{(i)} = 1 \mid \mathbf{x}^{(i)}; \boldsymbol{\theta} \right) = \sigma \left( \boldsymbol{\theta}^T \mathbf{x}^{(i)} \right)$  represents the probability of the output variable  $\hat{y}^{(i)}$  taking the value 1.  
**Note:** For binary classification,  $p_0^{(i)} + p_1^{(i)} = 1$  and therefore  $p_0^{(i)} = 1 - p_1^{(i)}$ .
- $\hat{y}^{(i)}$  is a random variable representing the **predicted output** for the  $i^{\text{th}}$  training sample,  $\mathbf{x}^{(i)}$  such that  $\hat{y}^{(i)} \in \{p_0^{(i)}, p_1^{(i)}\}$ .

In binary classification problems, we use an average of the per-example **binary cross entropies (BCE)** as the cost function:

$$\begin{aligned}
C(\boldsymbol{\theta}) &= \frac{1}{N} \sum_{i=1}^N H \left( y^{(i)}, \hat{y}^{(i)} \right) \\
&= -\frac{1}{N} \sum_{i=1}^N \left[ (1 - y^{(i)}) \log p_0^{(i)} + y^{(i)} \log p_1^{(i)} \right] \\
&= -\frac{1}{N} \sum_{i=1}^N \left[ (1 - y^{(i)}) \log (1 - p_1^{(i)}) + y^{(i)} \log p_1^{(i)} \right]
\end{aligned}$$

**Where:**

- $H(X, Y)$  is the cross entropy function which measures the dissimilarity between random variables  $X$  and  $Y$ .
- $y^{(i)}$  is a random variable representing the **true output** for the  $i^{\text{th}}$  training sample,  $\mathbf{x}^{(i)}$ .  
**Note:** Unlike  $\hat{y}^{(i)}$ , the  $y^{(i)}$  variable takes the value of the actual binary label since we don't have probabilities for the true outputs.

Observe that this is equivalent to taking the mean of the negative log likelihood function for the Bernoulli distribution:



$$\begin{aligned}
-\frac{1}{N} \mathcal{L}(\boldsymbol{\theta}; \mathbf{y}) &= -\frac{1}{N} \log L(\boldsymbol{\theta}; \mathbf{y}) \\
&= -\frac{1}{N} \log \prod_{i=1}^N \left(1 - p_1^{(i)}\right)^{1-y^{(i)}} \left(p_1^{(i)}\right)^{y^{(i)}} \\
&= -\frac{1}{N} \sum_{i=1}^N \log \left[ \left(1 - p_1^{(i)}\right)^{1-y^{(i)}} \left(p_1^{(i)}\right)^{y^{(i)}} \right] \\
&= -\frac{1}{N} \sum_{i=1}^N \left[ \log \left(1 - p_1^{(i)}\right)^{1-y^{(i)}} + \log \left(p_1^{(i)}\right)^{y^{(i)}} \right] \\
&= -\frac{1}{N} \sum_{i=1}^N \left[ (1 - y^{(i)}) \log \left(1 - p_1^{(i)}\right) + y^{(i)} \log \left(p_1^{(i)}\right) \right]
\end{aligned}$$

**Multinomial output**  $y^{(i)} \in \{1, \dots, K\}$

For multinomial classification problems, multinomial cross entropy is used as the error function for neural networks. This is a generalization of the previously seen binary cross entropy function, allowing for the use of the softmax function and multiple classes through **one-hot encoding**:

$$C(\boldsymbol{\Theta}) = -\frac{1}{N} \sum_{i=1}^N \sum_{c=1}^K y_c^{(i)} \log p_c^{(i)}$$

Where:

- $\boldsymbol{\Theta}$  is a matrix of weights, with each column representing the weight vector  $\boldsymbol{\theta}^{(c)}$  for class  $c$ . Recall that in multinomial classification (as seen in the *Logistic Regression* notes), for each class  $k$ , we classify a feature vector  $\mathbf{x}^{(i)}$  as  $k$  or not- $k$  through the use of the **softmax function**, which generates a probability for that class:

$$\sigma_c(\mathbf{x}^{(i)}; \boldsymbol{\Theta}) = \frac{\exp(\boldsymbol{\theta}^{(c)\top} \mathbf{x}^{(i)})}{\sum_{k=1}^K \exp(\boldsymbol{\theta}^{(k)\top} \mathbf{x}^{(i)})}$$

We assign  $\mathbf{x}^{(i)}$  to the class which had the highest probability.

- $p_c^{(i)} = \mathbb{P}(y^{(i)} = c | \mathbf{x}^{(i)}; \boldsymbol{\Theta}) = \sigma_c(\mathbf{x}^{(i)}; \boldsymbol{\Theta})$  represents the probability of feature vector  $\mathbf{x}^{(i)}$  being assigned to class  $c$ .
- $\mathbf{y}^{(i)} = (y_1^{(i)}, \dots, y_K^{(i)})^\top$  is a **one-hot encoded** vector representing the actual class of the  $i^{\text{th}}$  training sample. For example, if we have data with  $K = 6$  classes, and training sample  $\mathbf{x}^{(i)}$  is assigned to class 4, then  $y_4^{(i)}$  is set to 1, and the rest of the elements of  $\mathbf{y}^{(i)}$  are set to 0, giving:  $\mathbf{y}^{(i)} = (0, 0, 0, 1, 0, 0)^\top$ . One-hot encoded vectors are frequently seen in machine learning as they allow for selective or conditional calculations, similar to indicator variables.

## Using the error functions in backpropagation

As shown earlier in *Figure 4*, the modification of a single weight in any layer will have an effect on the output of the final layer. As a result, updating weights through gradient descent as we did in the *Logistic Regression* notes, will not work in a neural network since we have multiple weight matrices.

Instead, the **back-propagation** algorithm is used to extend gradient descent to the context of hidden layers and neurons. The idea of back-propagation is to choose an appropriate cost function and then systematically and sequentially modify the weights of various neurons in order to minimize this cost function.

## Feedforward neural networks

## Resources

- Charles Sutton, Nigel Goddard (School of Informatics, University of Edinburgh)  
[Introductory Applied Machine Learning: Neural Networks - Training an ANN](#)
- Hiroshi Shimodaira, Iain Murray, Steve Renals (School of Informatics, University of Edinburgh)  
[Algorithms, Data Structures and Learning: Single Layer Neural Networks](#)
- Gordon Ross (School of Mathematics, University of Edinburgh)  
[Statistical Learning: Neural Networks](#)
- Sagar Sharma (Towards Data Science)  
[The Fundamentals of Neural Networks: What the Hell is Perceptron?](#)
- Rubio95R (Stack Exchange)  
[Logistic Regression with Neural Network mindset vs. a shallow Neural Network](#)
- Rohan Varma  
[Creating Neural Networks in Tensorflow](#)
- cdeterman (Stack Exchange)  
[Activation Function for First Layer Nodes in an ANN](#)
- SimpliLearn  
[Multilayer Artificial Neural Network](#)
- Sebastian Raschka  
[Machine Learning FAQ: What is the relation between Logistic Regression and Neural Networks and when to use which?](#)
- James Isaac (Medium)  
[Understanding Deep Neural Networks from First Principles: Logistic Regression](#)
- Jiri Kriz (Nosco)  
[Neural Networks: Forward Propagation](#)
- The Microsoft Cognitive Toolkit  
[CNTK 103: Part B - Logistic Regression with MNIST](#)
- GeeksForGeeks  
[Activation functions in Neural Networks](#)
- Pablo Ruiz (Towards Data Science)  
[Neural Networks I: Notation and building blocks](#)
- Andrew Ng, Kian Katanforoosh (Computer Science Department, Stanford University)  
[CS230: Deep Learning Representations](#)
- Alex S. Holehouse  
[Neural Networks: Representation](#)
- DeepAI  
[Neural Network](#)
- Wikipedia  
[Activation function](#)
- Jason Brownlee  
[Why Training a Neural Network Is Hard](#)
- ML4A (Machine Learning For Artists)

How neural networks are trained

- *Isaac Changhau*

Loss Functions in Neural Networks

- *Kevin P. Murphy*

Machine Learning: A Probabilistic Perspective

- —

[https://gluon.mxnet.io/chapter02\\_supervised-learning/logistic-regression-gluon.html](https://gluon.mxnet.io/chapter02_supervised-learning/logistic-regression-gluon.html)

<https://www.perfectlyrandom.org/2019/04/27/bernoulli-distribution-as-a-tiny-nn/>

<https://bigquant.com/community/t/topic/121439>

<http://wap.sciencenet.cn/blog-578676-1118819.html>