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 $y^{(i)} \in \{1,\ldots,K\}$
- Back-propagation
  - Error derivatives
  - Additional derivatives
  - Example
    - Back-propagation procedure
  - Back-propagating with all training examples
    - Epochs and batches
    - Types of gradient descent
      - Mini-batch (stochastic) gradient descent
      - Stochastic gradient descent
      - Batch gradient descent

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 $\phi\left(\mathbf{x}^{(i)}\right)$. 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 $\boldsymbol{\theta}^{(1)}$ 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**.

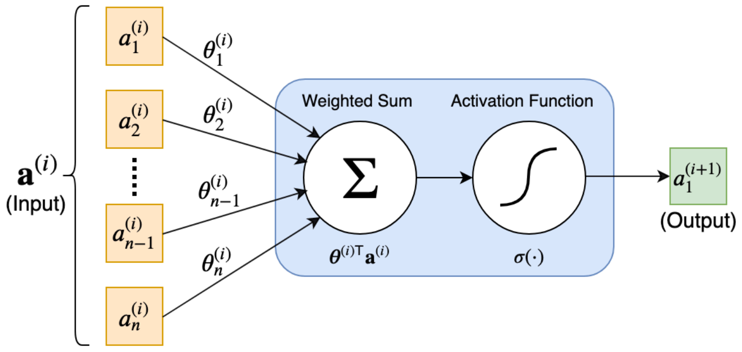


Figure 1: An artificial neuron (depicted in blue).

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 $\boldsymbol{\theta}^{(i)}$) . 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.

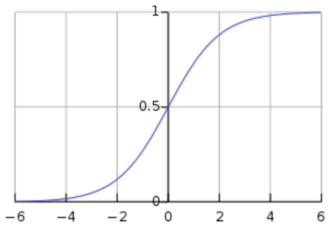


Figure 2: The logistic sigmoid activation function. (source)

   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 $\boldsymbol{\theta}^{(i+1)}$.

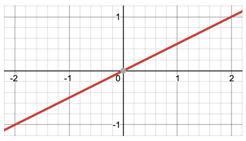
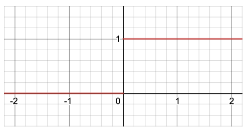
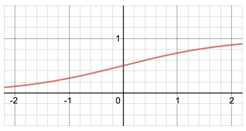
    - The outputs of the neurons in the final hidden layer are passed to the neurons in the **output layer**.
- **Output layer**:

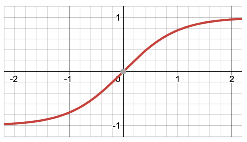
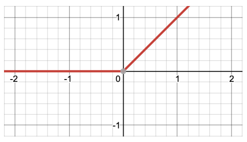
- Functions the same way as a hidden layer—each output takes a weighted sum of its inputs and passes it through an activation function.
- The number of output neurons is often the number of classes (in the case of a classification problem). However, in the case of binary classification, one output neuron 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 L layers, L 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	$\sigma_{\text{Step}}(x;k)=\begin{cases}1 & x\geq k\\ 0 & x<k\end{cases}$		• Can be used to make a decision between binary classes 0 and 1, given a real-valued argument. For example, since σ_{Logistic} yields a probability (where we would have to use $\arg\max$ to determine the class), we can determine the class by using $\sigma_{\text{Step}}(x;k=\frac{1}{2})$ in the output layer.
Logistic (sigmoid) function	$\sigma_{\text{Logistic}}(x)=\frac{1}{1+e^{-x}}$		• Typically used in the output layer of a neural network performing binary classification. • Suffers from the issue of vanishing gradients as $\|x\|$ increases. • Differentiable across entire domain.
Hyperbolic	$\sigma_{\tanh}(x)=\tanh(x)$		• Can mathematically be

tangent function			represented as a shifted and scaled version of σ_{Logistic} and is similarly used for binary classification tasks. • However, this cannot be directly used for negative log likelihood cost functions due to the function range being $(-1, 1)$. • Suffers from the same vanishing gradient issue as σ_{Logistic}.
Rectified linear unit (ReLU)	$\sigma_{\text{ReLU}}(x) = \max(0, x)$		ReLU is generally preferred to sigmoidal activation functions such as σ_{Logistic} and $\sigma_{\tanh}$ in deeper neural networks, mainly because: • its simpler gradients (during backpropagation) allow for each layer to be trained quicker, • it does not suffer from the vanishing gradient problem.

Softmax function

The softmax function is a special type of activation function which takes a vector of K real numbers as input, and normalizes it into a discrete probability distribution consisting of K probabilities.

The standard softmax function $\sigma : \mathbb{R}^K \to \mathbb{R}^K$ is defined by the formula:

$$\sigma_{\text{Soft}}(i, \mathbf{z}) = \frac{\exp z_i}{\sum_{j=1}^{K} \exp z_j} \qquad \text{for } i = \{1, \ldots, K\} \text{ and } \mathbf{z} = (z_1, \ldots, z_k) \in \mathbb{R}^K$$

Multi-layer neural networks

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

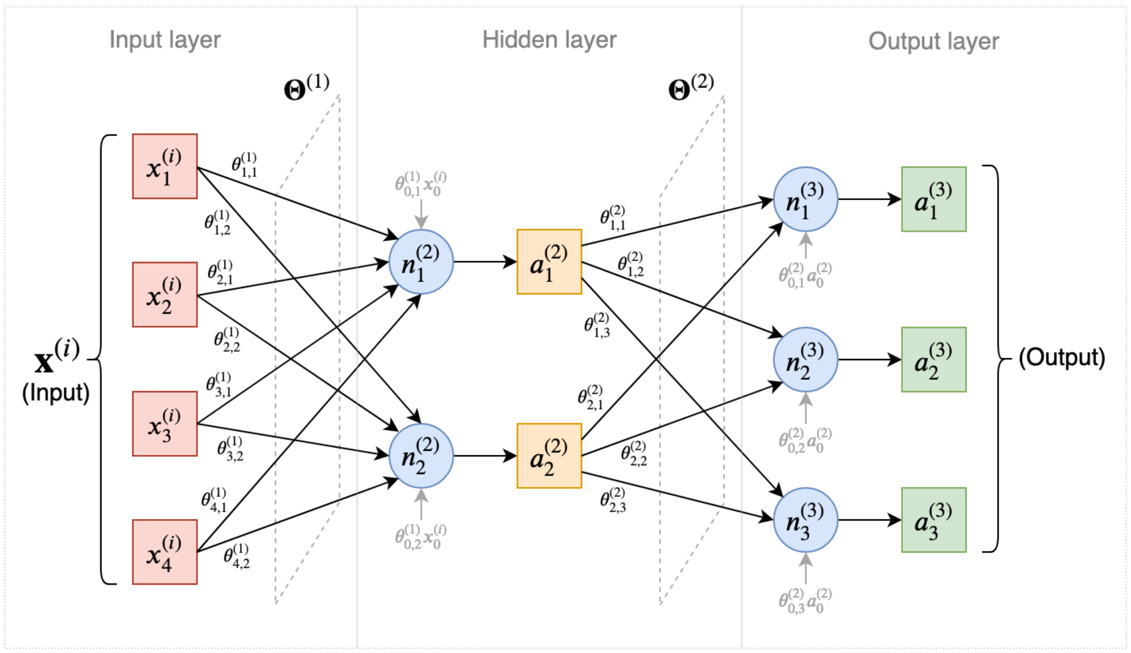


Figure 3: A multi-layer neural network consisting of one hidden layer with two neurons.

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 $x_0^{(i)}$.

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

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

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

$$\begin{array}{c|cccccc}
\mathbf{\Theta}^{(i)} & \boldsymbol{\theta}_{0,\cdot}^{(i)} & \boldsymbol{\theta}_{1,\cdot}^{(i)} & \boldsymbol{\theta}_{2,\cdot}^{(i)} & \cdots & \boldsymbol{\theta}_{N_i,\cdot}^{(i)} \\
\hline
\boldsymbol{\theta}_{\cdot,1}^{(i)\mathsf{T}} & \theta_{0,1}^{(i)} & \theta_{1,1}^{(i)} & \theta_{2,1}^{(i)} & \cdots & \theta_{N_i,1}^{(i)} \\
\boldsymbol{\theta}_{\cdot,2}^{(i)\mathsf{T}} & \theta_{0,2}^{(i)} & \theta_{1,2}^{(i)} & \theta_{2,2}^{(i)} & \cdots & \theta_{N_i,2}^{(i)} \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
\boldsymbol{\theta}_{\cdot,N_{i+1}}^{(i)\mathsf{T}} & \theta_{0,N_{i+1}}^{(i)} & \theta_{1,N_{i+1}}^{(i)} & \theta_{2,N_{i+1}}^{(i)} & \cdots & \theta_{N_i,N_{i+1}}^{(i)}
\end{array}$$

> **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 $i+1$.
> - $\boldsymbol{\theta}_{\cdot,k}^{(i)} = \left(\theta_{0,k}^{(i)}, \dots, \theta_{N_i,k}^{(i)}\right)^\mathsf{T}$ represents the weights from each neuron in layer i, to neuron k in layer $i+1$.

> - $\boldsymbol{\theta}_{j,\cdot}^{(i)} = \left(\theta_{j,1}^{(i)}, \dots, \theta_{j,N_{i+1}}^{(i)}\right)^\mathsf{T}$ 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:

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

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

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

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

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

> **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:
>
> $$a_j^{(i)} = \sigma_j^{(i)}\left(\boldsymbol{\theta}_{\cdot,j}^{(i-1)\mathsf{T}}\mathbf{a}^{(i-1)}\right)$$
>
> 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 $\boldsymbol{\Theta}^{(i-1)}$ with the outputs of the previous layer, and a vector-valued activation function $\boldsymbol{\sigma}^{(i)}$:
>
> $$\begin{aligned}\mathbf{a}^{(i)} &= \boldsymbol{\sigma}^{(i)}\Big(\boldsymbol{\Theta}^{(i-1)}\mathbf{a}^{(i-1)}\Big)^\mathsf{T} \\ &= \Bigg(\underbrace{\sigma^{(i)}\left(\boldsymbol{\theta}_{\cdot,1}^{(i-1)\mathsf{T}}\mathbf{a}^{i-1}\right)}_{a_1^{(i)}}, \dots, \underbrace{\sigma^{(i)}\left(\boldsymbol{\theta}_{\cdot,N_i}^{(i-1)\mathsf{T}}\mathbf{a}^{i-1}\right)}_{a_{N_i}^{(i)}}\Bigg)^\mathsf{T}\end{aligned}$$

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:

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

$$\mathbf{a}^{(2)} = \begin{pmatrix} a_0^{(2)} \\ \sigma^{(2)}\left(\boldsymbol{\theta}_{\cdot,1}^{(1)\mathsf{T}}\mathbf{a}^{(1)}\right) \\ \sigma^{(2)}\left(\boldsymbol{\theta}_{\cdot,2}^{(1)\mathsf{T}}\mathbf{a}^{(1)}\right) \end{pmatrix}, \quad \boldsymbol{\Theta}^{(2)} = \begin{pmatrix} \theta_{0,1}^{(2)} & \theta_{1,1}^{(2)} & \theta_{2,1}^{(2)} \\ \theta_{0,2}^{(2)} & \theta_{1,2}^{(2)} & \theta_{2,2}^{(2)} \\ \theta_{0,3}^{(2)} & \theta_{1,3}^{(2)} & \theta_{2,3}^{(2)} \end{pmatrix}$$

$$\mathbf{a}^{(3)} = \begin{pmatrix} \sigma^{(3)}\left(\boldsymbol{\theta}_{\cdot,1}^{(2)\mathsf{T}}\mathbf{a}^{(2)}\right) \\ \sigma^{(3)}\left(\boldsymbol{\theta}_{\cdot,2}^{(2)\mathsf{T}}\mathbf{a}^{(2)}\right) \\ \sigma^{(3)}\left(\boldsymbol{\theta}_{\cdot,3}^{(2)\mathsf{T}}\mathbf{a}^{(2)}\right) \end{pmatrix}$$

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 $\boldsymbol{\Theta}^{(i)}$ 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.

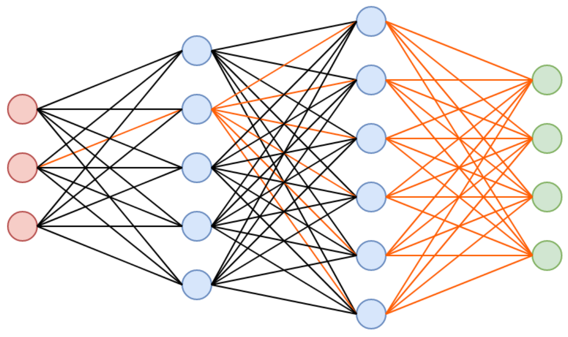


Figure 4: The propagating effect of modifying one weight, on subsequent layers.

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 $C(\mathbf{\Theta})$ 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 $C(\mathbf{\Theta})$:

$$\hat{\mathbf{\Theta}} = \arg\min_{\mathbf{\Theta}} C(\mathbf{\Theta})$$

> **Where**: $\mathbf{\Theta}$ represents all of the weights in the neural network, and can therefore be represented as a vector of the weight matrices for each layer:
>
> $$\mathbf{\Theta} = \left(\mathbf{\Theta}^{(1)}, \ldots, \mathbf{\Theta}^{(L)}\right)$$

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 $y^{(i)} \in \{1, \ldots, K\}$ (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.

$$C(\mathbf{\Theta}) = \sum_{i=1}^{N} \left(y^{(i)} - \hat{y}^{(i)}\right)^2$$

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:

$$C(\mathbf{\Theta}) = \frac{1}{2N} \sum_{i=1}^{N} \left(y^{(i)} - \hat{y}^{(i)}\right)^2$$

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:

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

> **Where**:

- $p_1^{(i)} = \mathbb{P}\left(\hat{y}^{(i)} = 1 \middle| \mathbf{x}^{(i)}; \boldsymbol{\Theta}\right) = a_1^{(L+1)} = \sigma\left(\boldsymbol{\Theta}^{(L)\top}\mathbf{a}^{(L)}\right)$ represents the probability of the output variable $\hat{y}^{(i)}$ taking the value 1.

  Observe that for binary classification, the final layer $L+1$ has only one neuron $n_1^{(L+1)}$ (with output $a_1^{(L+1)}$) which represents $p_1^{(i)}$. As a result, the weight matrix from layer L to layer $L+1$ is a $(1 \times N_L)$ vector:

  $$\boldsymbol{\Theta}^{(L)} = \left(\theta_{0,1}^{(L)}, \theta_{1,1}^{(L)}, \ldots, \theta_{N_L,1}^{(L)}\right)$$

  Note: For binary classification, $p_0^{(i)} + p_1^{(i)} = 1$ and therefore $p_0^{(i)} = 1 - p_1^{(i)}$. Due to this property, we don't require a second neuron in the final layer as we can just use the complement of $p_1^{(i)}$.

- $\hat{y}^{(i)}$ is a random variable representing the **predicted output** for the i^{th} training sample, $\mathbf{x}^{(i)}$ such that $\hat{y}^{(i)} \in \left\{p_0^{(i)}, p_1^{(i)}\right\}$.

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

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

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^{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:

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

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**:

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

> **Where**:
>
> - There are K neurons in the output layer, with each output $a_c^{(L+1)} : c \in \{1, \dots, K\}$ representing the probability $p_c^{(i)}$ for the feature vector $\mathbf{x}^{(i)}$ being in class c.
>
>   Therefore, the weight matrix $\mathbf{\Theta}^{(L)}$ from layer L to the final layer $L+1$, has dimensions $(K \times N_L)$.
>
>   Recall that in multinomial classification (as seen in the *Logistic Regression* notes), for each class c, we classify a feature vector $\mathbf{x}^{(i)}$ as k or not-c through the use of the **softmax function** which generates a probability of $\mathbf{x}^{(i)}$ being in class c:
>
> $$\sigma_{\text{Soft}}\left(c, \mathbf{x}^{(i)}; \mathbf{\Theta}\right) = \frac{\exp\left(\boldsymbol{\theta}_{\cdot,c}^{(L)\top} \mathbf{a}^{(L)}\right)}{\sum_{k=1}^{K} \exp\left(\boldsymbol{\theta}_{\cdot,k}^{(L)\top} \mathbf{a}^{(L)}\right)}$$
>
>   We assign $\mathbf{x}^{(i)}$ to the class which yields the highest probability (as generated by the softmax function).
>
> - $p_c^{(i)} = \mathbb{P}\left(y^{(i)} = c \middle| \mathbf{x}^{(i)}; \mathbf{\Theta}\right) = \sigma_{\text{Soft}}\left(c, \mathbf{x}^{(i)}; \mathbf{\Theta}\right)$ represents the probability of feature vector $\mathbf{x}^{(i)}$ being assigned to class c.
>
> - $\mathbf{y}^{(i)} = \left(y_1^{(i)}, \dots, y_K^{(i)}\right)^\top$ is a **one-hot encoded** vector representing the actual class of the i^{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 $y^{(i)}$ are set to 0, giving: $\mathbf{y}^{(i)} = (0, 0, 0, 1, 0, 0)^\top$. One-hot encoded vectors are frequently used in machine learning as they allow for selective or conditional calculations, similar to indicator variables.

Back-propagation

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 behind back-propagation is to choose an appropriate cost function and then systematically modify the weights of various neurons in order to minimize this cost function. This method is similar to gradient descent, but it uses the chain rule in order to calculate the gradient vector.

The purpose of back-propagation is to calculate all of the error derivatives of the neural network.

Error derivatives

The back-propagation algorithm decides how much to update each weight of the network after comparing the predicted output with the desired output for a **particular** example. For this, we need to compute how the error changes with respect to each weight—that is $\frac{\partial}{\partial \theta^{(l)}_{j,k}} C(\boldsymbol{\Theta})$.

Once we have these error derivatives, the weights can be updated using a simple update rule for $l \in \{1, \ldots, L\}$:

$$\theta^{(l)}_{j,k} \leftarrow \theta^{(l)}_{j,k} - \eta \frac{\partial C}{\partial \theta^{(l)}_{j,k}}$$

Or as a per-layer weight matrix update:

$$\boldsymbol{\Theta}^{(l)} \leftarrow \boldsymbol{\Theta}^{(l)} - \eta \boldsymbol{\nabla}_{\boldsymbol{\Theta}^{(l)}} C(\boldsymbol{\Theta})$$

> **Where**:
>
> $$\boldsymbol{\nabla}_{\boldsymbol{\Theta}^{(l)}} C(\boldsymbol{\Theta}) = \begin{pmatrix} \nabla_{\boldsymbol{\theta}^{(l)}_{0,\cdot}} C(\boldsymbol{\Theta}) \\ \nabla_{\boldsymbol{\theta}^{(l)}_{1,\cdot}} C(\boldsymbol{\Theta}) \\ \vdots \\ \nabla_{\boldsymbol{\theta}^{(l)}_{N_l,\cdot}} C(\boldsymbol{\Theta}) \end{pmatrix}^{\mathsf{T}} = \begin{pmatrix} \nabla_{\boldsymbol{\theta}^{(l)}_{0,\cdot}} C(\boldsymbol{\Theta}) & \nabla_{\boldsymbol{\theta}^{(l)}_{1,\cdot}} C(\boldsymbol{\Theta}) & & \nabla_{\boldsymbol{\theta}^{(l)}_{N_l,\cdot}} C(\boldsymbol{\Theta}) \\ \frac{\partial C}{\partial \theta^{(l)}_{0,1}} & \frac{\partial C}{\partial \theta^{(l)}_{1,1}} & \cdots & \frac{\partial C}{\partial \theta^{(l)}_{N_l,1}} \\ \frac{\partial C}{\partial \theta^{(l)}_{0,2}} & \frac{\partial C}{\partial \theta^{(l)}_{1,2}} & \cdots & \frac{\partial C}{\partial \theta^{(l)}_{N_l,2}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial C}{\partial \theta^{(l)}_{0,N_{l+1}}} & \frac{\partial C}{\partial \theta^{(l)}_{1,N_{l+1}}} & \cdots & \frac{\partial C}{\partial \theta^{(l)}_{N_l,N_{l+1}}} \end{pmatrix}$$

Additional derivatives

To help compute $\frac{\partial C}{\partial \theta^{(l)}_{j,k}}$, we store two additional derivatives for each neuron:

- How the error changes with the **total (weighted) input of the neuron**: $\frac{\partial C}{\partial z^{(l)}_j}$.

  > **Where**: $z^{(l)}_j = \boldsymbol{\theta}^{(l-1)\mathsf{T}}_{\cdot,j} \mathbf{a}^{(l-1)}$, the input for neuron $n^{(l)}_j$.

- How the error changes with the **total output of the neuron**: $\frac{\partial C}{\partial a^{(l)}_j}$.

  > **Where**: $a^{(l)}_j = \sigma^{(l)}\left(\boldsymbol{\theta}^{(l-1)\mathsf{T}}_{\cdot,j} \mathbf{a}^{(l-1)}\right) = \sigma^{(l)}\left(z^{(l)}_j\right)$

Example

Consider a single training example $\left(\mathbf{x}^{(i)}, y^{(i)}\right)$ with a predicted output $\hat{y}^{(i)}$ from the following neural network:

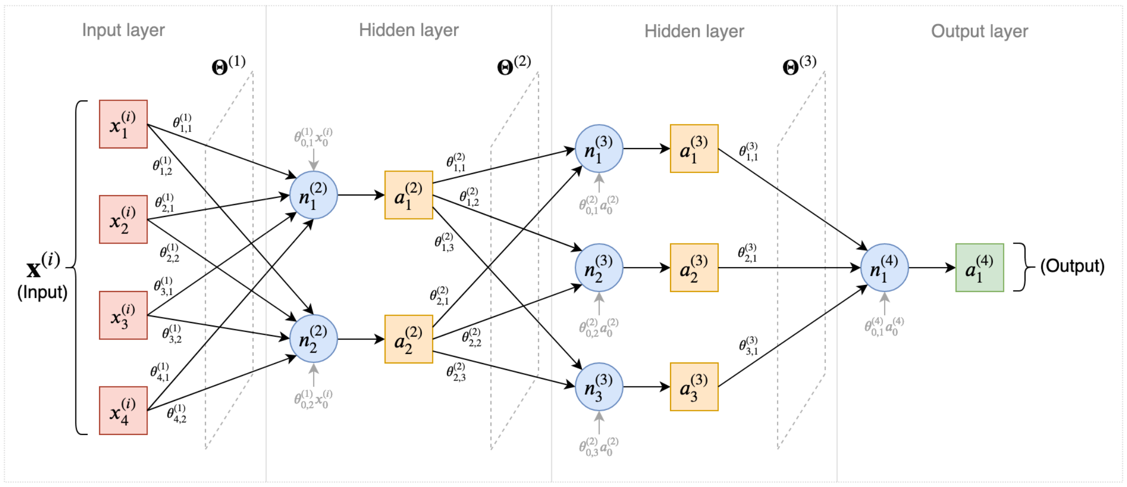


Figure 5: Modified version of the neural network shown in *Figure 3*, with an additional layer consisting of one output neuron.

The neural network has the following properties:

- The output $y^{(i)}$ is given by the output of the final neuron.
- All of the activation functions are σ_{Logistic}.
- $\mathbf{x}^{(i)} \in \mathbb{R}^4$ and $y^{(i)} \in (0, 1)$.
- The cost for one training example is given by:

$$C(\mathbf{\Theta}) = \frac{1}{2}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$$

The neural network can be defined by the following matrices and vectors (as seen before):

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

$$\mathbf{a}^{(2)} = \begin{pmatrix} a_0^{(2)} \\ \sigma\left(\boldsymbol{\theta}_{\cdot,1}^{(1)\mathsf{T}}\mathbf{a}^{(1)}\right) \\ \sigma\left(\boldsymbol{\theta}_{\cdot,2}^{(1)\mathsf{T}}\mathbf{a}^{(1)}\right) \end{pmatrix}, \quad \mathbf{\Theta}^{(2)} = \begin{pmatrix} \theta_{0,1}^{(2)} & \theta_{1,1}^{(2)} & \theta_{2,1}^{(2)} \\ \theta_{0,2}^{(2)} & \theta_{1,2}^{(2)} & \theta_{2,2}^{(2)} \\ \theta_{0,3}^{(2)} & \theta_{1,3}^{(2)} & \theta_{2,3}^{(2)} \end{pmatrix}$$

$$\mathbf{a}^{(3)} = \begin{pmatrix} a_0^{(3)} \\ \sigma\left(\boldsymbol{\theta}_{\cdot,1}^{(2)\mathsf{T}}\mathbf{a}^{(2)}\right) \\ \sigma\left(\boldsymbol{\theta}_{\cdot,2}^{(2)\mathsf{T}}\mathbf{a}^{(2)}\right) \\ \sigma\left(\boldsymbol{\theta}_{\cdot,3}^{(2)\mathsf{T}}\mathbf{a}^{(2)}\right) \end{pmatrix}, \quad \mathbf{\Theta}^{(3)} = \begin{pmatrix} \theta_{0,1}^{(3)} & \theta_{1,1}^{(3)} & \theta_{2,1}^{(3)} & \theta_{3,1}^{(3)} \end{pmatrix}$$

$$\mathbf{a}^{(4)} = a_1^{(4)} = \sigma\left(\boldsymbol{\theta}_{\cdot,1}^{(3)\mathsf{T}}\mathbf{a}^{(3)}\right)$$

Back-propagation procedure

To begin back-propagating to find the cost derivatives, we start at the end of the network, with our predicted output $\hat{y}^{(i)}$.

1. Given our cost function $C(\boldsymbol{\Theta}) = \frac{1}{2}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$, we have:

$$\frac{\partial C}{\partial \hat{y}^{(i)}} = y^{(i)} - \hat{y}^{(i)}$$

   Note: Recall that $\hat{y}^{(i)} = a_1^{(4)}$, so we can also say that $\frac{\partial C}{\partial \hat{y}^{(i)}} = \frac{\partial C}{\partial a_1^{(4)}}$.

2. Now that we have $\frac{\partial C}{\partial a_1^{(4)}}$, we can find $\frac{\partial C}{\partial z_1^{(4)}}$ through the use of the chain rule:

$$\frac{\partial C}{\partial z_1^{(4)}} = \frac{\partial C}{\partial a_1^{(4)}} \frac{\partial a_1^{(4)}}{\partial z_1^{(4)}}$$

   Observe that $\frac{\partial a_1^{(4)}}{\partial z_1^{(4)}}$ may be expressed as:

$$\begin{aligned}\frac{\partial a_1^{(4)}}{\partial z_1^{(4)}} &= \frac{\partial}{\partial z_1^{(4)}}\sigma\left(\boldsymbol{\theta}_{\cdot,1}^{(3)\mathsf{T}}\mathbf{a}^{(3)}\right)\\ &= \frac{\partial}{\partial z_1^{(4)}}\sigma\left(z_1^{(4)}\right)\end{aligned}$$

   The logistic function has the nice property that its derivative is defined as $\frac{\partial}{\partial z}\sigma(z) = \sigma(z)\big(1-\sigma(z)\big)$. This allows us to further simplify $\frac{\partial a_1^{(4)}}{\partial z_1^{(4)}}$:

$$\frac{\partial a_1^{(4)}}{\partial z_1^{(4)}} = \sigma\left(z_1^{(4)}\right)\left(1-\sigma\left(z_1^{(4)}\right)\right)$$

   However, to maintain some generality over the various activation functions, we will continue to write this as $\sigma'\left(z_1^{(4)}\right)$.

   In *Step 1* we found that $\frac{\partial C}{\partial a_1^{(4)}} = y^{(i)} - \hat{y}^{(i)}$ and can therefore write $\frac{\partial C}{\partial z_1^{(4)}}$ as:

$$\begin{aligned}\frac{\partial C}{\partial z_1^{(4)}} &= \left(y^{(i)} - \hat{y}^{(i)}\right)\frac{\partial a_1^{(4)}}{\partial z_1^{(4)}}\\ &= \left(y^{(i)} - \sigma\left(z_1^{(4)}\right)\right)\frac{\partial a_1^{(4)}}{\partial z_1^{(4)}}\end{aligned}$$

   Substituting the expression for $\frac{\partial a_1^{(4)}}{\partial z_1^{(4)}}$ that we found earlier:

$$\frac{\partial C}{\partial z_1^{(4)}} = \underbrace{\sigma'\left(z_1^{(4)}\right)}_{\frac{\partial a_1^{(4)}}{\partial z_1^{(4)}}}\underbrace{\left(y^{(i)} - \sigma\left(z_1^{(4)}\right)\right)}_{\frac{\partial C}{\partial a_1^{(4)}}}$$

3. Once we have the cost derivative with respect to the total (weighted) input of a neuron, $\frac{\partial C}{\partial z_j^{(l)}}$, we can get the cost derivative with respect to the weights coming into that neuron:

$$\frac{\partial C}{\partial \theta_{j,k}^{(l-1)}} = \frac{\partial C}{\partial z_k^{(l)}} \frac{\partial z_k^{(l)}}{\partial \theta_{j,k}^{(l-1)}}$$

Recall that $z_k^{(l)}$ is defined as:

$$z_k^{(l)} = \sigma\left(\boldsymbol{\theta}_{\cdot,k}^{(l-1)\mathsf{T}} \mathbf{a}^{(l-1)}\right)$$

Then the derivative of the input $z_k^{(l)}$ with respect to the incoming weight $\theta_{j,k}^{(l-1)}$ is given by the following (using the chain rule):

$$\begin{aligned}\frac{\partial z_k^{(l)}}{\partial \theta_{j,k}^{(l-1)}} &= \sigma'\left(\boldsymbol{\theta}_{\cdot,k}^{(l-1)\mathsf{T}} \mathbf{a}^{(l-1)}\right) a_j^{(l-1)} \\ &= \sigma'\left(z_k^{(l)}\right) a_j^{(l-1)}\end{aligned}$$

Applying this to $z_1^{(4)}$ and its cost derivative $\frac{\partial C}{\partial z_1^{(4)}}$ that we already obtained in *Step 2*, we can find the cost derivative of the first weight $\theta_{1,1}^{(3)}$ in $\boldsymbol{\Theta}^{(3)}$:

$$\begin{aligned}\frac{\partial C}{\partial \theta_{1,1}^{(3)}} &= \frac{\partial C}{\partial z_1^{(4)}} \frac{\partial z_1^{(4)}}{\partial \theta_{1,1}^{(3)}} \\ &= \underbrace{\sigma'\left(z_1^{(4)}\right)\left(y^{(i)} - \sigma\left(z_1^{(4)}\right)\right)}_{\frac{\partial C}{\partial z_1^{(4)}}} \underbrace{\sigma'\left(z_1^{(4)}\right) a_1^{(3)}}_{\frac{\partial z_1^{(4)}}{\partial \theta_{1,1}^{(3)}}} \\ &= \sigma'\left(z_1^{(4)}\right)^2 \left(y^{(i)} - \sigma\left(z_1^{(4)}\right)\right) a_1^{(3)}\end{aligned}$$

The same procedure can be used to obtain an error derivative for all of the weights in $\boldsymbol{\Theta}^{(3)}$.

4. Using the chain rule for multivariate functions once again, we can calculate $\frac{\partial C}{\partial a_j^{(3)}}$.

$$\begin{aligned}\frac{\partial C}{\partial z_j^{(3)}} &= \frac{\partial C}{\partial a_j^{(3)}} \frac{\partial a_j^{(3)}}{\partial z_j^{(3)}} \\ &= \frac{\partial C}{\partial a_j^{(3)}} \frac{\partial}{\partial z_j^{(3)}} \sigma\left(z_j^{(3)}\right) \\ &= \frac{\partial C}{\partial a_j^{(3)}} \sigma'\left(z_j^{(3)}\right)\end{aligned}$$

Where $\frac{\partial C}{\partial a_j^{(3)}}$ may also be found using the chain rule and the weight derivatives of the previous layer:

$$\frac{\partial C}{\partial a_j^{(3)}} = \frac{\partial C}{\partial z_1^{(4)}} \sum_{k=1}^{N_4} \frac{\partial z_1^{(4)}}{\partial \theta_{j,k}^{(3)}} \frac{\partial \theta_{j,k}^{(3)}}{\partial a_j^{(3)}}$$

And then I got lost. But basically keep back-propagating until you find all the error derivatives with respect to all weights of the network.

Back-propagating with all training examples

The example above describes the process of finding all of the error derivatives for the weights in the network after a forward pass with just one training example.

To train a neural network using all of the training examples:

1. **Input a set of training examples**
   $\mathcal{D} = \left\{\left(\mathbf{x}^{(i)}, y^{(i)}\right)\right\}_{i=1}^{N}$.
2. **Randomize the weights of the networks**
   Biases should be set to 1.
3. **For each training example**:
   1. **Feedforward**
      For each layer l and each neuron $n_j^{(l)}$ on layer l, compute:
      - $z_j^{(l)} = \boldsymbol{\theta}_{\cdot,j}^{(l-1)} \mathbf{a}^{(l-1)}$
      - $a_j^{(l)} = \sigma^{(l)}\left(z_j^{(l)}\right)$
   2. **Calculate the error and gradient of the output layer**
      For each neuron $n_k^{(N+1)}$ on layer $L+1$, compute the values:
      - $\dfrac{\partial C}{\partial z_k^{(L+1)}}$ (the error derivative w.r.t the weighted input, a.k.a ***the error***)
      - $\dfrac{\partial C}{\partial \theta_{j,k}^{(L)}}$ (the error derivative w.r.t the weight $\theta_{j,k}^{(L)}$ connecting $n_j^{(L)}$ with $n_k^{(L+1)}$)
   3. **Backpropagate the error**
      For each $l \in \{L, L-1 \dots, 2\}$, compute the following values for each neuron $n_k^{(l)}$:
      - $\dfrac{\partial C}{\partial z_k^{(l)}}$ (the error)
      - $\dfrac{\partial C}{\partial \theta_{j,k}^{(l-1)}}$ (the error derivative w.r.t the weight $\theta_{j,k}^{(l-1)}$)
4. **Gradient descent**
   For each weight $\theta_{j,k}^{(l)}$ in the network (where $l \in \{1, \dots, L\}$), update it using the simple update rule $\theta_{j,k}^{(l)} \leftarrow \theta_{j,k}^{(l)} - \eta \frac{\partial C}{\partial \theta_{j,k}^{(l)}}$.

Epochs and batches

The algorithm above forms a training loop for one **epoch**, treating the entire training set as one **batch**.

An **epoch** represents a single forward pass and backward pass of **all** of the training examples through the neural network. Since feeding the entire training data set into the neural network would be time consuming, we divide it into several **batches**, each of equal **batch size**.

A **batch** is therefore just a division of the training set. An epoch requires all of the batches to be passed forward and backward through the neural network. Therefore, the number of **iterations** in an epoch is simply equal to the number of batches.

> **Example**: If we have a training set consisting of 1000 examples and divide it into batches of size 200, we will have 5 batches and will therefore require 5 iterations to complete 1 full epoch.

Types of gradient descent

Mini-batch (stochastic) gradient descent

In **mini-batch (stochastic) gradient descent**, the training set is randomly divided into a number of subsets—each of a specified batch size. The term **stochastic** arises due to the fact that the batches are randomly divided.

The weights of the network are updated after each batch completes a forward and backward pass through the network. After one epoch is complete, we may randomly select the subsets again and perform another training epoch.

Suppose we divide $\mathcal{D}=\left\{(\mathbf{x}^{(i)},y^{(i)})\right\}_{i=1}^{N}$ into B batches such that $\mathcal{D}=\bigcup_{b=1}^{B}\mathcal{D}^{(b)}$.

> **Where**: Each batch is given as $\mathcal{D}^{(b)}=\left\{(\mathbf{x}^{(b,i)},y^{(b,i)})\right\}_{i=1}^{\frac{N}{B}}$. Note that the size of $\mathcal{D}^{(b)}$ may not always be exactly $\frac{N}{B}$ as the number of batches B often does not perfectly divide the total number of training samples N.
>
> So instead, the size of a training batch is simply denoted as $|\mathcal{D}^{(b)}|$.

The update rules for gradient descent must be slightly modified (from the one seen before) as we now have to average the error derivative over the number of samples in the batch.

$$\theta_{j,k}^{(l)} \leftarrow \theta_{j,k}^{(l)} - \frac{\eta}{|\mathcal{D}^{(b)}|}\sum_{i=1}^{|\mathcal{D}^{(b)}|}\frac{\partial C}{\partial\theta_{j,k}^{(l)}}$$

$$\mathbf{\Theta}^{(l)} \leftarrow \mathbf{\Theta}^{(l)} - \frac{\eta}{|\mathcal{D}^{(b)}|}\sum_{i=1}^{|\mathcal{D}^{(b)}|}\mathbf{\nabla}_{\mathbf{\Theta}^{(l)}}C(\mathbf{\Theta})$$

> **Where**: C is the per-example (non-averaged) error function $C_{(\mathbf{x}^{(b,i)},\mathbf{y}^{(b,i)})}$ evaluated on training example $(\mathbf{x}^{(b,i)},\mathbf{y}^{(b,i)})$. The subscripts are omitted for clarity.

Stochastic gradient descent

Stochastic gradient descent takes the idea of mini-batches to the extreme—it only uses a single example per iteration (batch size 1).

This is simply a special case of mini-batch stochastic gradient descent where $B=N$ and $|\mathcal{D}^{(b)}|=1$. This leads to the following update rules for each example (batch):

$$\theta_{j,k}^{(l)} \leftarrow \theta_{j,k}^{(l)} - \eta\frac{\partial C}{\partial\theta_{j,k}^{(l)}}$$

$$\mathbf{\Theta}^{(l)} \leftarrow \mathbf{\Theta}^{(l)} - \eta\mathbf{\nabla}_{\mathbf{\Theta}^{(l)}}C(\mathbf{\Theta})$$

> **Where**: C is the per-example (non-averaged) error function $C_{(\mathbf{x}^{(b,i)},\mathbf{y}^{(b,i)})}$ evaluated on training example $(\mathbf{x}^{(b,i)},\mathbf{y}^{(b,i)})$. The subscripts are omitted for clarity.

In this form of gradient descent, the weights and biases are updated for each training example, after it has completed a forward and backward pass through the neural network. Therefore, one epoch would consist of N iterations.

Batch gradient descent

In **batch gradient descent**, each epoch treats the entire training data as one batch, meaning the batch size is equal to the number of training examples.

Once again, this is an extreme case of mini-batch stochastic gradient descent where $B = 1$ and $|\mathcal{D}^{(b)}| = N$. This leads to the following update rules:

$$\theta_{j,k}^{(l)} \leftarrow \theta_{j,k}^{(l)} - \frac{\eta}{N}\sum_{i=1}^{N}\frac{\partial C}{\partial \theta_{j,k}^{(l)}}$$

$$\mathbf{\Theta}^{(l)} \leftarrow \mathbf{\Theta}^{(l)} - \frac{\eta}{N}\sum_{i=1}^{N}\mathbf{\nabla}_{\mathbf{\Theta}^{(l)}} C(\mathbf{\Theta})$$

> **Where**: C is the per-example (non-averaged) error function $C_{(\mathbf{x}^{(b,i)},\mathbf{y}^{(b,i)})}$ evaluated on training example $(\mathbf{x}^{(b,i)}, \mathbf{y}^{(b,i)})$. The subscripts are omitted for clarity.

In this form of gradient descent, the weights are not updated until the epoch is complete (all training examples have done a forward and backward pass through the network).

This form of gradient descent is rarely used in practice since it requires the entire data set to be loaded into memory. It may be feasible for problems with few training examples, but neural networks require lots of training data and therefore may not be appropriate in some of these cases anyway.

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
  Softmax function
- *Jason Brownlee*
  Why Training a Neural Network Is Hard
  A Gentle Introduction to Mini-Batch Gradient Descent and How to Configure Batch Size
- *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
- *Morgan Giraud (MetaFlow)*
  ML notes: Why the log-likelihood?
- *Gluon*
  Binary classification with logistic regression
  Gradient descent and stochastic gradient descent from scratch
- *Ankur Gupta (Perfectly Random)*
  Bernoulli Distribution as a tiny Neural Network
- *astroman (BigQuant)*
  Entropy (3.13 Information theory)
- *jingweimo (ScienceNet)*
  Nonlinearity and loss function in multi-tasking problems
- *Google Developers*
  Backpropagation algorithm
  Reducing Loss: Stochastic Gradient Descent
- *D.W. (StackExchange)*
  Why does the neural network logistic regression cost function sum for all layers only for lambda?
- *user329469 (StackExchange)*
  Derivative of neural network function with respect to weights
- *Michael A. Nielsen, "Neural Networks and Deep Learning", Determination Press, 2015*
  Chapter 2: How the backpropagation algorithm works
- *Oleg Shirokikh (StackExchange)*
  Back-propagation in Neural Nets with >2 hidden layers
- *John McGonagle, George Shaikouski, Christopher Williams, Andrew Hsu, Jimin Khim*
  Backpropagation