Simple Neural Network

Neural Network

Neural networks can be used for three main tasks: ***classification***, ***regression*** (prediction of a scalar), clustering (ordering unstructured data into groups), and more.

A neural network has a vast number of adjustable parameters, represented by weights and biases. The goal of a neuron network is to train it to fit the data instead of just memorizing it – this is called ***generalization***. Training is done by slowly adjusting weights and biases on many examples of data, and calculating the loss (error) of the output.

Overfitting occurs when the network only learns to fit the training data but is incapable of relating input-output dependencies, and thus the parameters are not tweaked correctly.

Layers

A neural network consists of a number of layers: the input layer, the output layer, and so-called hidden layers.

The ***input layer*** represents the input samples, like pixel values or sensor data. This input needs to be preprocessed so that it is normalized, scaled, and in numeric form.

The ***output layer*** represents the output value. With classification the output layer typically has as many neurons as the amount of classes, but it can also be binary classification (true or false).

Any layers between these two are called ***hidden layers***. A neural network is ***deep*** when there are two or more such layers. Hidden layers consist of neurons with a vast number of tweakable parameters: the weights and biases.

Dense layers are those where each neuron in the layer is connected to each neuron in the previous and next layer. This type of layer is also called ***linear*** or ***fully-connected***.

Parameters

When training, the weights of a layer are initialized randomly to a small non-zero value, and the biases set to zero. Otherwise, in a pre-trained model the parameters are initialized to the previously determined output of that model.

If a neuron no longer affects the model's result on any input and returns 0 on most inputs, then the neuron or even the network is essentially untrainable, or "dead". In this case, tweaking the bias to some non-zero value will lead to more favorable results.

Layer inputs use data batches because the output of a layer should be sample-related and not neuron-related. Samples are passed further through the network, and the next layer will also expect a batch of inputs.

- $\mathbf{X}$: matrix of input samples. One input sample is given as $x \in \mathbf{X}$.
- $\boldsymbol{y}$: target classification labels.
- $\hat{\boldsymbol{y}}$: predicted classification (the values of the output layer).
- $\boldsymbol{\Theta}$: matrix of outputs of a hidden layer. The output of one neuron is $\theta \in \boldsymbol{\Theta}$

Arrays

An n-dimensional array is homologous if all arrays along the 'row' dimension are of equal length.

The shape of an array is determined by the dimensions, so a 3D array with 3 matrices of 2 lists of 4 elements has a shape of (3, 2, 4). Shape is an important factor for mathematical operations on arrays.

- An array is an ordered homologous container for numbers. - A vector is a list or 1-dimensional array. - A matrix is a 2-dimensional array. Each element of the array can be accessed using a tuple of indices as a key. - A tensor is an n-dimensional object that can be represented as an array.

Forward pass

The goal of the forward pass is to run the input samples through the neural network and to calculate the loss, which is a representation of the error of the model.

A dense layer applies the following linear formula:

$$\boldsymbol{\Theta}_i = \boldsymbol{\Theta}_{i-1} \cdot \mathbf{W}_i + \boldsymbol{b}_i$$

where $\boldsymbol{\Theta}_i$ is the output matrix of a layer i, $\mathbf{W}$ is the weight matrix of this layer, and $\boldsymbol{b}$ is a vector of biases, one for each neuron in the layer.

Matrix $\boldsymbol{\Theta}_{i-1}$ can be output from a previous hidden layer, or the input sample matrix $\mathbf{X}$ for the first layer.

Activation

After calculating the linear formula above, an activation function is applied to its output to simulate whether a neuron is activated for a certain input. Just like weights and biases, activation functions are essential parts of each neuron of a layer.

A neural network typically has two different activation functions; one used in the hidden layers, and a different one in the output layer.

If an activation function is nonlinear, it will allow neural networks with two or more hidden layers to map nonlinear functions. Some examples of nonlinear activation functions are:

- Sigmoid
- Rectified Linear Unit (ReLU)
- Softmax

Sigmoid

The Sigmoid or logistic function is one of the original granular activation functions used in neural networks:

$$\text{Sigmoid}(x) = \frac{1}{1 + \mathrm{e}^{-x}}$$

It is less common in modern networks, where it has been largely replaced by the Rectified Linear activation function.

Rectified Linear Unit (ReLU)

The rectified function is a simple but powerful when using them in two (or more) successive layers because it can then approximate nonlinear functions. While simple, efficient, and nearly linear, it remains nonlinear due to the bend after 0.

$$\text{ReLU}(x) = \max(0, x)$$

Softmax

The Softmax activation function is used for multiclass classification problems and is applied to the output layer. It can take in non-normalized, or uncalibrated, inputs and produce a normalized distribution of probabilities for each class that will add up to 1. The predicted class is associated with the output neuron that returned the largest confidence score.

Softmax takes as input a vector $\boldsymbol{z} = (z_1, \ldots, z_K) \in \mathbb{R}$ of K real numbers, and normalizes it into a probability distribution consisting of K probabilities proportional to the exponentials of the input numbers:

$$\text{Softmax}(\boldsymbol{z})_i = \frac{\exp(\boldsymbol{z}_i)}{\sum_{k=1}^{K} \exp(\boldsymbol{z}_k)} \text{ for } i = 1, \ldots, K.$$

Exponentiation leads to non-negative values for each of the inputs. Due to the monotonic nature of the exponential function, higher input values also give higher outputs, so the predicted class will not change after applying exponentiation. Converting these numbers to a probability distribution that describe a vector of confidences is done via normalization in the denominator.

Softmax also includes a subtraction of the largest of the inputs before doing the exponentiation step. This mitigates two pervasive challenges with neural networks: dead neurons and exploding values, which render a network useless over time. The exponential function in the Softmax activation is actually one of the sources of exploding values. By subtracting the highest input value from all inputs, the maximum value will be 0, and thus the exponentiation will only produce outputs between 0 and 1.

Loss

A ***loss function*** or ***cost function*** quantifies the error of a model and the objective is to reduce it to 0 as closely as possible. By calculating loss, we strive to increase correct confidence and decrease misplaced confidence.

Neural networks that do regression have a loss function that calculate the mean squared error.

Categorical Cross-Entropy

For classification, the output of the network is a probability distribution, and a useful function when training a classification problem with classes is ***categorical cross-entropy***. It compares a ground-truth probability vector $\boldsymbol{y}$ and some predicted distribution $\hat{\boldsymbol{y}}$. It is one of the most commonly used loss functions with a softmax activation on the output layer.

The formula for calculating the categorical cross-entropy of $\boldsymbol{y}$ (actual/desired distribution) and $\hat{\boldsymbol{y}}$ (predicted distribution) is:

$$L_i = -\sum_j \boldsymbol{y}_{i,j} \log(\hat{\boldsymbol{y}})$$

Where L_i denotes sample loss value, i is the i-th sample in the set, j is the label/output index, $\boldsymbol{y}$ denotes the target values, and $\hat{\boldsymbol{y}}$ denotes the predicted values.

By encoding the target values $\boldsymbol{y}_{i,j}$ as one-hot encoded vectors, this equation can be simplified. One-hot means that one of the values in a vector is "hot" with a value of 1, and the rest are "cold" with values of 0. As a result, the summation over the targets $\hat{\boldsymbol{j}}$ zeroes out to just one multiplication by 1, simplifying the equation to:

$$L_i = -\log(\hat{\boldsymbol{y}}_{i,k})$$

Where L_i denotes sample loss value, i is the i-th sample in a set, k is the index of the target label (ground-true label), $\hat{\boldsymbol{y}}$ denotes the predicted values.

There is a problem with the categorical cross-entropy when a confidence value equals 0; this will lead to the invalid calculation of $\log(0)$, which is undefined. To prevent loss from being exactly 0 or negative, the value is clipped from both sides by the same infinitesimally small number.

Accuracy

While loss is a useful metric for optimizing a model, accuracy describes how often the largest confidence is the correct class in terms of a fraction. To compute the accuracy, the argmax values from the Softmax outputs $\hat{\boldsymbol{y}}$ are compared to the target classification $\boldsymbol{y}$:

$$\text{Accuracy}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \frac{1}{\|\hat{\boldsymbol{y}}\|} \sum_{i=1}^{\|\hat{\boldsymbol{y}}\|} (1 \text{ if } \text{argmax}(\hat{\boldsymbol{y}}_{\boldsymbol{i}}) = \boldsymbol{y}_i)$$

In other words, it is an arithmetic mean of all the predictions $\hat{\boldsymbol{y}}_i$ that correspond to the correct classification $\boldsymbol{y}_i$ (in which the result is 1).

Backpropagation

Calculating the partial derivatives of all the functions from the loss function back through all the layers in the neural network is called backpropagation. In this backwards pass, ***gradient descent*** is performed by calculating derivatives of single-parameter functions and gradients of multivariate functions. The end goal is to reduce the loss metric as much as possible by optimizing the model.

A partial derivative measures how much impact a single input has on a function's output. The gradient (denoted with a ∇ symbol) is a vector of the size of inputs containing all of the partial derivative solutions with respect to each of the inputs and shows the direction of the steepest ascent. By applying a negative fraction to a gradient the loss can be decreased.

In a dense layer, a single neuron in the current layer is connected to all neurons in the next layer. During backpropagation, each neuron from the current layer will receive a vector of partial derivatives for all the neurons in the next layer. Each neuron will output a gradient of the partial derivatives with respect to all of their inputs.

Rectified Linear Unit (ReLU) Derivative

The forward pass with the ReLU function for one neuron with inputs $\mathbf{X}$, weights $\mathbf{W}$, and biases $\boldsymbol{b}$ is:

$$
\begin{aligned}
&y = \text{ReLU}(\mathbf{X} \cdot \mathbf{W} + \boldsymbol{b}) \text{ or} \\
&y = \text{ReLU}(x_1 w_1 + x_2 w_2 + x_3 w_3 + \boldsymbol{b})
\end{aligned}
$$

This equation contains three nested functions: max, a summation and multiplications. The partial derivative to any input x_i (or any other of the inputs) is obtained by applying the ***chain rule*** to the nested max, sum, and mul functions:

$$
\begin{aligned}
\frac{\partial}{\partial x_i}[\text{ReLU}(\text{sum}(\text{mul}(x_1, w_1), \text{mul}(x_i, w_i), \text{mul}(x_n, w_n)))] = \\
\frac{d\text{ReLU}(\dots)}{d\text{sum}(\dots)} \cdot \frac{\partial \text{sum}(\dots)}{\partial \text{mul}(x_i, w_i)} \cdot \frac{\partial \text{mul}(x_i, w_i)}{\partial x_i}
\end{aligned}
$$

Firstly, the partial derivative of the sum operation is always 1, no matter the inputs:

$$
\begin{aligned}
f(x, y) = x + y \quad \rightarrow \quad & \frac{\partial}{\partial x} f(x, y) = 1 \\
& \frac{\partial}{\partial y} f(x, y) = 1
\end{aligned}
$$

Secondly, the derivative for a product (mul operation) is whatever the input is being multiplied by:

$$
\begin{aligned}
f(x, y) = x \cdot y \quad \rightarrow \quad & \frac{\partial}{\partial x} f(x, y) = y \\
& \frac{\partial}{\partial y} f(x, y) = x
\end{aligned}
$$

Thirdly, derivative of ReLU with respect to input x is 1 if $x > 0$, and 0 otherwise:

$$
\frac{\partial}{\partial \text{ReLU}(x)} = \frac{\partial}{\partial x} \max(0, x) = 1 \, (x > 0)
$$

And thus finally – following the chain rule – the full derivative of the ReLU function with respect to input x_i becomes:

$$\frac{\partial}{\partial x_i}\text{ReLU}(x_i) = \partial_v \cdot 1\,(z > 0) \cdot w_i, \text{ with } z = \boldsymbol{x} \cdot \boldsymbol{w} + b$$

Where ∂_v is the derivative calculated in the next layer, and z is the weighted sum of the inputs and the bias.

Combining all the partial derivatives above for each input $x_i \in X$ make up the gradients and can be represented as ∇_x for the inputs, ∇_w for the weights, and ∇_b for the bias:

$$\nabla_x = \begin{bmatrix} \frac{\partial}{\partial x_1}\text{ReLU}(x_1) \\ \frac{\partial}{\partial x_i}\text{ReLU}(x_i) \\ \frac{\partial}{\partial x_n}\text{ReLU}(x_n) \end{bmatrix}$$

$$\nabla_w = \begin{bmatrix} \frac{\partial}{\partial w_1}\text{ReLU}(w_1) \\ \frac{\partial}{\partial w_i}\text{ReLU}(w_i) \\ \frac{\partial}{\partial w_n}\text{ReLU}(w_n) \end{bmatrix}$$

$$\nabla_b = \begin{bmatrix} \frac{\partial}{\partial b}\text{ReLU}(b) \end{bmatrix}$$

Categorical Cross-Entropy Derivative

To calculate the gradient we need to use the partial derivative of the full forward function, which is:

$$L_i = -\sum_j \boldsymbol{y}_{i,j} \log(\hat{\boldsymbol{y}}_{i,j})$$

Where L_i denotes sample loss value, i is the i-th sample in a set, j is the label index, $\boldsymbol{y}$ are the target values, and $\hat{\boldsymbol{y}}$ are the predicted values.

Let's define the gradient equation as the partial derivative of the loss function with respect to each of its inputs and solve it using the chain rule:

$$\begin{aligned}
\frac{\partial L_i}{\partial \hat{\boldsymbol{y}}_{i,j}} &= \frac{\partial}{\partial \hat{\boldsymbol{y}}_{i,j}}[-\sum_j \boldsymbol{y}_{i,j} \log(\hat{\boldsymbol{y}}_{i,j})] \\
&= -\sum_j \boldsymbol{y}_{i,j} \cdot \frac{\partial}{\partial \hat{\boldsymbol{y}}_{i,j}} \log(\hat{\boldsymbol{y}}_{i,j}) \\
&= -\sum_j \boldsymbol{y}_{i,j} \cdot \frac{1}{\hat{\boldsymbol{y}}_{i,j}} \cdot \frac{\partial}{\partial \hat{\boldsymbol{y}}_{i,j}} \hat{\boldsymbol{y}}_{i,j} \\
&= -\sum_j \boldsymbol{y}_{i,j} \cdot \frac{1}{\hat{\boldsymbol{y}}_{i,j}} \cdot 1 \\
&= -\sum_j \frac{\boldsymbol{y}_{i,j}}{\hat{\boldsymbol{y}}_{i,j}}
\end{aligned}$$

The summation can be omitted because the partial derivative is calculated with respect to the $\boldsymbol{y}$, predicted values at the i-th sample and given index j. Thus the sum is performed over a single element at j. The derivative of this loss function with respect to its inputs then equals the negative ground-truth vector ($\boldsymbol{y}$), divided by the vector of the predicted values ($\hat{\boldsymbol{y}}$):

$$\frac{\partial L_i}{\partial \hat{\boldsymbol{y}}_{i,j}} = -\frac{\boldsymbol{y}_{i,j}}{\hat{\boldsymbol{y}}_{i,j}}$$

Softmax Derivative

The definition of the partial derivative of the Softmax function is:

$$\mathbf{S}_{i,j} = \frac{\exp(\boldsymbol{z}_{i,j})}{\sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k})} \quad \rightarrow \quad \frac{\partial \mathbf{S}_{i,j}}{\partial \boldsymbol{z}_{i,m}} = \frac{\partial \frac{\exp(\boldsymbol{z}_{i,j})}{\sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k})}}{\partial \boldsymbol{z}_{i,m}}$$

Where $\mathbf{S}_{i,j}$ denotes j-th Softmax's output of the i-th sample, $\boldsymbol{z}$ is the input array which is a list of output vectors from the previous layer, $\boldsymbol{z}_{i,j}$ is the j-th Softmax's input of the i-th sample, K is the number of classifications, and $\boldsymbol{z}_{i,m}$ is the m-th Softmax's input of the i-th sample.

The Softmax function equals the exponentiated input divided by the sum of all exponentiated inputs, so we need to exponentiate all of the values first and then divide each of them by the sum of all of them to perform the normalization. Because each input of the Softmax impacts the each of the outputs, each partial derivative of each output with respect to each input has to be calculated.

Applying the chain rule and solving with respect to input $\boldsymbol{z}_{i,k}$ yields:

$$\begin{aligned}
\frac{\partial \mathbf{S}_{i,j}}{\partial \boldsymbol{z}_{i,m}} &= \frac{\partial \frac{\exp(\boldsymbol{z}_{i,j})}{\sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k})}}{\partial \boldsymbol{z}_{i,m}} \\
&= \frac{\frac{\partial}{\partial \boldsymbol{z}_{i,m}} \exp(\boldsymbol{z}_{i,j}) \cdot \sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k}) - \exp(\boldsymbol{z}_{i,j}) \cdot \frac{\partial}{\partial \boldsymbol{z}_{i,m}} \sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k})}{[\sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k})]^2} \\
&= \frac{\frac{\partial}{\partial \boldsymbol{z}_{i,m}} \exp(\boldsymbol{z}_{i,j}) \cdot \sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k}) - \exp(\boldsymbol{z}_{i,j}) \cdot \frac{\partial}{\partial \boldsymbol{z}_{i,m}} \sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k})}{[\sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k})]^2}
\end{aligned}$$

Now there are two possibilities for the partial derivative on the left side of the subtraction operation that lead to different results: one where $j = m$ and one where $j \neq m$.

In the case of $j = m$:

$$\begin{aligned}
\frac{\partial \mathbf{S}_{i,j}}{\partial \boldsymbol{z}_{i,m}} &= \frac{\exp(\boldsymbol{z}_{i,j}) \cdot \sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k}) - \exp(\boldsymbol{z}_{i,j}) \cdot \exp(\boldsymbol{z}_{i,m})}{[\sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k})]^2} \\
&= \frac{\exp(\boldsymbol{z}_{i,j}) \cdot (\sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k}) - \exp(\boldsymbol{z}_{i,m}))}{\sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k}) \cdot \sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k})} \\
&= \frac{\exp(\boldsymbol{z}_{i,j})}{\sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k})} \cdot \frac{\sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k}) - \exp(\boldsymbol{z}_{i,m})}{\sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k})} \\
&= \frac{\exp(\boldsymbol{z}_{i,j})}{\sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k})} \cdot \frac{\sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k})}{\sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k})} - \frac{\exp(\boldsymbol{z}_{i,m})}{\sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k})}
\end{aligned}$$

Looking closely, both the left part and right part of the equation are just the definitions of the Softmax function itself:

$$\frac{\partial \mathbf{S}_{i,j}}{\partial \boldsymbol{z}_{i,m}} = \mathbf{S}_{i,j} \cdot (1 - \mathbf{S}_{i,m}) \quad \text{when } j = m$$

In the case of $j \neq m$:

$$\begin{aligned}
\frac{\partial \mathbf{S}_{i,j}}{\partial \boldsymbol{z}_{i,m}} &= \frac{\frac{\partial}{\partial \boldsymbol{z}_{i,m}} \exp(\boldsymbol{z}_{i,j}) \cdot \sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k}) - \exp(\boldsymbol{z}_{i,j}) \cdot \frac{\partial}{\partial \boldsymbol{z}_{i,m}} \sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k})}{[\sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k})]^2} \\
&= \frac{0 \cdot \sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k}) - \exp(z_{i,j}) \cdot \exp(\boldsymbol{z}_{i,m})}{[\sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k})]^2} \\
&= \frac{-\exp(z_{i,j}) \cdot \exp(\boldsymbol{z}_{i,m})}{\sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k}) \cdot \sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k})} \\
&= -\frac{\exp(\boldsymbol{z}_{i,j})}{\sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k})} \cdot \frac{\exp(\boldsymbol{z}_{i,m})}{\sum_{k=1}^{K} \exp(\boldsymbol{z}_{i,k})}
\end{aligned}$$

For this case the Softmax function can also be identified in both the left and right parts of the equation:

$$\frac{\partial \mathbf{S}_{i,j}}{\partial \boldsymbol{z}_{i,m}} = -\mathbf{S}_{i,j} \cdot \mathbf{S}_{i,m} \quad \text{when } j \neq m$$

Thus, the two derivatives calculated above can be represented as:

$$\frac{\partial \mathbf{S}_{i,j}}{\partial \boldsymbol{z}_{i,m}} = \begin{cases} \mathbf{S}_{i,j} \cdot (1 - \mathbf{S}_{i,m}) & \text{when } \; j = m \\ \mathbf{S}_{i,j} \cdot (0 - \mathbf{S}_{i,m}) & \text{when } \; j \neq m \end{cases}$$

By applying the ***Kronecker delta function*** the equation simplifies even further:

$$\mathbf{S}_{i,j}\delta_{j,k} - \mathbf{S}_{i,j}\mathbf{S}_{i,k} \quad \text{where} \quad \delta_{i,j} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$$

The result of this is a Jacobian matrix; an array of partial derivatives in all of the of both input vectors. This is then done for every output of the Softmax function with respect to each input separately (a batch of samples), and thus the result becomes a list of Jacobian matrices, or a 3D matrix. This result then needs to be condensed to a single partial derivative value by taking each row of the Jacobian matrix and multiplying it with the corresponding value in the loss function's gradient.

Softmax Cross-Entropy

By combining the ***Softmax activation*** and the ***Categorical Cross-Entropy loss*** functions, their derivative can be greatly simplified, leading to a significant performance boost. The simplification is possible due to the the inputs of the cross-entropy function being equal to the outputs of the Softmax function, and using the properties of the one-hot encoded vector to reduce a summation to a scalar value.

The chain rule can be applied to calculate the partial derivative of the Categorical Cross-Entropy loss function with respect to the Softmax's inputs:

$$
\begin{aligned}
\frac{\partial L_i}{\partial \boldsymbol{z}_{i,m}} &= \frac{\partial L_i}{\partial \hat{\boldsymbol{y}}_{i,j}} \cdot \frac{\partial \mathbf{S}_{i,j}}{\partial \boldsymbol{z}_{i,m}} \\
&= \frac{\partial L_i}{\partial \hat{\boldsymbol{y}}_{i,j}} \cdot \frac{\partial \hat{\boldsymbol{y}}_{i,j}}{\partial \boldsymbol{z}_{i,m}} \\
&= -\sum_j \frac{\boldsymbol{y}_{i,j}}{\hat{\boldsymbol{y}}_{i,j}} \cdot \frac{\partial \hat{\boldsymbol{y}}_{i,j}}{\partial \boldsymbol{z}_{i,m}} \\
&= -\frac{\boldsymbol{y}_{i,m}}{\hat{\boldsymbol{y}}_{i,m}} \cdot \frac{\partial \hat{\boldsymbol{y}}_{i,m}}{\partial \boldsymbol{z}_{i,m}} - \sum_{j \neq m} \frac{\boldsymbol{y}_{i,j}}{\hat{\boldsymbol{y}}_{i,j}} \cdot \frac{\partial \hat{\boldsymbol{y}}_{i,j}}{\partial \boldsymbol{z}_{i,m}} \\
&= -\frac{\boldsymbol{y}_{i,m}}{\hat{\boldsymbol{y}}_{i,m}} \cdot \hat{\boldsymbol{y}}_{i,j} \cdot (1 - \hat{\boldsymbol{y}}_{i,m}) - \sum_{j \neq m} \frac{\boldsymbol{y}_{i,j}}{\hat{\boldsymbol{y}}_{i,j}} (-\boldsymbol{y}_{i,j} \boldsymbol{y}_{i,m}) \\
&= -\boldsymbol{y}_{i,m} \cdot (1 - \hat{\boldsymbol{y}}_{i,m}) + \sum_{j \neq m} \boldsymbol{y}_{i,j} \hat{\boldsymbol{y}}_{i,m} \\
&= -\boldsymbol{y}_{i,m} + \boldsymbol{y}_{i,m} \hat{\boldsymbol{y}}_{i,m} + \sum_{j \neq m} \boldsymbol{y}_{i,j} \hat{\boldsymbol{y}}_{i,m} \\
&= -\boldsymbol{y}_{i,m} + \sum_j \boldsymbol{y}_{i,j} \hat{\boldsymbol{y}}_{i,m} \\
&= -\boldsymbol{y}_{i,m} + \hat{\boldsymbol{y}}_{i,m} \\
&= \hat{\boldsymbol{y}}_{i,m} - \boldsymbol{y}_{i,m}
\end{aligned}
$$

After applying the chain rule to the partial derivatives of cross-entropy loss function L_i and of the Softmax function $\mathbf{S}_{i,j}$, the whole equation simplifies significantly to the subtraction of the predicted labels $\hat{\boldsymbol{y}}$ and the ground-truth labels $\boldsymbol{y}$.

Optimizers

During backpropagation the calculated gradient points to the current steepest loss ascent, and taking the negative of the gradient vector flips it toward the current steepest descent. The goal of an optimizer is to adjust a model's parameters to approach the ***global minimum*** of the loss gradient. When there are millions of dimensions (parameters) in a function, gradient descent is the best known way to search for a global minimum.

As long as the loss is not very close to 0 the model stopped learning and it has become stuck in a ***local minimum***. However, if a loss of 0 is reached, this is a reason to be suspicious, because it usually means that the model has been overfitted to the training data. Conversely, ***generalization*** is the ability of the model to correctly predict unseen data, or to approximate the "trend" of the data closely.

An ***epoch*** is a full cycle of a forward then backward pass and then optimization through all of the training data. A neural network will usually be trained for multiple epochs.

Most optimizers are variants of ***Stochastic Gradient Descent*** (SGD).

Stochastic Gradient Descent

Stochastic Gradient Descent historically refers to an optimizer that fits a single sample at a time, but modern variants is seen as one that assumes a batch of data, whether that batch happens to be a single sample, every sample in a dataset, or some subset of the full dataset at a time.

Stochastic Gradient Descent uses a factor called the ***learning rate*** that is applied to the gradients before subtracting them from the weight and bias parameters. The learning rate is a so-called ***hyper-parameter***, a global parameter that is configured for the optimizer.

Gradient descent is the simplest optimization algorithm which computes gradients of loss function with respect to model parameters and updates them by using the following formula:

$$\theta_t = \theta_{t-1} - \eta \nabla_\theta$$

Where θ is a parameter (weight or bias), ∇_θ is the gradient of θ, η is the learning rate, and t is the step/iteration/epoch index.

Learning Rate

The learning rate η is a scalar value that modifies the influence of the gradient descent. By adjusting this parameter correctly, the global minimum can be closely approximated.

By taking small steps in the direction of the gradient we can ensure that the steepest descent is followed. However, steps that are too small can cause ***learning stagnation***, and thus if the learning rate is too small, the optimizer may get stuck in a local minimum. Having a learning rate that is too high will lead to an unstable model where it may not be able to find the global minimum, or further epochs may cause a ***gradient explosion***. This is a situation where parameter updates cause the loss and gradient to increase instead of decrease.

Choosing the right hyper-parameters is critical and challenging. It is usually best to start with the optimizer defaults, perform a few steps, and observe the training process after tuning. In most cases it is beneficial start with a higher learning rate and decrease it during training using a ***learning rate decay***.

Learning Rate Decay

One method to add decay is to decrease the learning rate manually or logically in response to the loss across epochs, as when it begins to level out or starts jumping over large deltas.

Another way is to implement a ***decay rate*** which steadily decays the learning rate per batch or epoch. A step-wise decay rate is also referred to as ***1/t decaying*** or ***exponential decaying***.

The learning rate decay is defined as the reciprocal of the step count fraction:

$$\eta = \eta_0 \frac{1}{1 + \beta\, t}$$

Where η is the learning rate for the current step t, η_0 is the initial learning rate, and β is the decay rate. The formula will make the learning rate smaller with time, because the denominator will get higher with each step.

If the learning rate decays too quickly and becomes too small, the model gets trapped in a local minimum. Therefore, the decay is usually a small number like 0.001 or 0.00001.

Stochastic Gradient Descent with Momentum

Adding ***momentum*** to an optimizer adds "inertia" to the steps taken towards the global minimum and can mitigate learning stagnation and gradient explosion. Momentum retains the direction of the gradient from a previous step and uses it to influence the next update's direction.

The momentum γ represents the fraction of the previous parameter update to retain, and subtracting the actual gradient, multiplied by the learning rate, from it:

$$\begin{aligned} v_t &= \gamma\, v_{t-1} + \eta\, \nabla\theta_t \\ \theta_t &= \theta_{t-1} - v_t \end{aligned}$$

Where θ_t are the weights for the current step t, $\gamma \in \{0, 1\}$ is the momentum, θ_{t-1} is are the weights of the previous step, η is the learning rate for step t, and $\nabla\theta_t$ is the gradient, or derivative of loss with respect to weight. In practice, this equation is similarly applied to determine the adjusted biases.

The Momentum optimizer is usually one of two main initial choices, the other being ***Adam***.

AdaGrad

Adaptive Gradient or AdaGrad uses an adaptive per-parameter learning rate rather than a global one.

The concept is that some parameters (weights/biases) matter more to the loss than others, and thus some parameters should be updated more significantly than others in each epoch. In AdaGrad the parameters will have a different learning rate based on whether the features are ***dense*** or ***sparse***. That is, if gradients corresponding to a certain parameter are large, then the respective learning rate will be small, and conversely, for smaller gradients the learning rate will be bigger. AdaGrad can deal with vanishing and exploding gradient problems this way.

The ***sum of squared gradients*** captures how much a parameter has been updated already. Instead of keeping track of the sum of gradients like momentum, AdaGrad keeps track of the ***sum of squared gradients*** in a ***cache*** to adapt the gradient in a certain direction.

During the update step, AdaGrad scales the learning rate by dividing it by the square root of the accumulated gradients:

$$\begin{aligned} v_t &= v_{t-1} + {\nabla\theta_t}^2 \\ \theta_t &= \theta_{t-1} - \frac{\eta}{\sqrt{v_t} + \epsilon} \nabla\theta_t \end{aligned}$$

Where v_t is the sum of squared gradients at step t, θ_t are the weights at step t, $\nabla\theta_t$ are the gradients from all previous steps, η is the initial learning rate, and ϵ is a small positive term to prevent possible division by zero.

The advantage of AdaGrad is that there is no need to manually adjust the learning rate during training, but the learning rate constantly decays over time and therefore converges slowly during later steps.

RMSProp

RMSProp stands for ***Root Mean Square Propagation***, another adaptive optimizer that calculates a learning rate per parameter. It was designed as an improvement over AdaGrad's issue of learning rate decay and its formula is similar:

$$v_t = \rho\, v_{t-1} + (1-\rho)\nabla\theta_t{}^2$$
$$\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{v_t} + \epsilon}\nabla\theta_t$$

Where the new hyper-parameter ρ is the cache memory decay rate. The value of this parameter can be set much lower than a standard learning rate because RMSProp carries over a lot of momentum of gradient and small gradient updates are enough to keep it going. Therefore, the recommended default is $\rho = 0.001$.

Instead of storing a cumulated sum of squared gradients, the ***exponentially moving average*** v_t is calculated for squared gradients $\nabla\theta^2$. RMSProp converges faster than AdaGrad because the exponentially moving average puts more emphasis on recent gradient values rather than equally distributing importance between all gradients. Additionally, the learning rate does not always decay with the increase of iterations.

Adam

Adam is short for ***Adaptive Moment Estimation*** and combines RMSProp with Momentum. In addition to storing an exponentially decaying average of past squared gradients v_t like RMSprop, it also keeps an exponentially decaying average of past gradients m_t, similar to momentum:

$$v_t = \beta_1 v_{t-1} + (1-\beta_1)\nabla\theta_t$$
$$s_t = \beta_2 s_{t-1} + (1-\beta_2)\nabla\theta_t{}^2$$

Where β_1 and β_2 are the decay rates that are recommended to be set at $\beta_1 = 0.9$ and $\beta_2 = 0.999$. Moving averages v_t and s_t are estimates of the ***first moment*** (the mean) and the ***second moment*** (the uncentered variance) of the gradients respectively, hence the name of this optimizer.

Initially, and the decay rates are small (close to 1), the moments are biased towards zero. Those biases can be mitigated by computing the ***bias-corrected first and second moment estimates***:

$$\hat{v}_t = \frac{v_t}{1-\beta_1^t}$$
$$\hat{s}_t = \frac{s_t}{1-\beta_2^t}$$

And then use these estimates to update the parameters according to Adam's update rule:

$$\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{\hat{s}_t} + \epsilon}\hat{v}_t$$

Hyper-parameters

In general it is usually a good idea to try the Adam optimizer, but sometimes simple SGD or Momentum performs better than Adam. A general starting learning rate for SGD is 1.0, with a decay down to 0.1. For Adam, a good starting learning rate is 0.001, decaying down to 0.0001. Keep in mind that different problems may require different values, but these are decent starting values.

Validation

The expectation of a well-trained neural network is that it can learn a representation of a clear set of rules and use this representation to predict classes of additional generated data.

Unfortunately, the complexity of neural networks is both a strength and a weakness. Having a massive amount of tunable parameters means that they are exceptional at closely fitting to the training data. An overfit model is exceptional at predicting the data that it has already seen, but does significantly worse on new, similar data.

Avoiding Overfitting

To avoid overfitting, it is essential to have both ***training data*** and ***testing data*** as separate sets. This way, some data is reserved and withheld from the training data to test the model's effectiveness.

Overfitting can be recognized when testing data results begin to significantly diverge in trend from training data. Having a loss difference of 10% or more between the training and testing phases is a common sign of serious overfitting. In general, the goal is to have the testing loss identical to the training loss, even if that means higher loss and lower accuracy on the training data. Similar performance on both datasets means that model has generalized.

Divergence of the training and testing data can often take a long time. One general rule to follow when selecting initial model hyperparameters is to find the smallest model possible that still learns. With fewer neurons it's easier for a neural network to generalize compared to memorizing the data. The process of trying different model settings is called ***hyperparameter tuning***.

Other ways to avoid overfitting are ***regularization*** or using a so-called ***dropout layer***.

Validating

Hyperparameter tuning can be performed using another dataset called ***validation data***. When the training data is large, a part of it can separated and used for validation purposes.

If the training data is sparse, there are two options. One is to temporarily split the training data into a smaller training dataset and a validation dataset for hyperparameter tuning. Then train the model on all of the training data when the hyperparameters are finalized.

The second is k***-fold cross-validation***: split the training dataset into k amount of parts and train the model on all except one of those parts and validate it on the remaining part. Then repeat this process for all of the parts, choosing a different one used for validation each time.

When validating it is common to loop over different hyperparameter sets while running the training phase multiple times, applying different settings each run, and reviewing the results to choose the best set of hyperparameters.

Preprocessing

Preprocessing refers to the technique of preparing the training dataset (or samples) for use in the model. This includes but is not limited to: standardization, scaling, variance scaling, mean removal, non-linear transformations, and scaling to outliers. Neural networks usually perform best on data consisting of numbers in a range of $[0, 1]$ or $[-1, 1]$. For example for pixel values in range $[0, 255]$ they would be scaled to either of these ranges.

There are two main reasons that these ranges are preferred:

- Centering data on zero helps with model training because it will attenuate weight biasing in some direction. This is because many activation functions actually behave quite well within this described range.
- In multiplication operations with numbers in the $[-1, 1]$ range the result becomes a small value, a fraction. Big numbers on the other hand may cause instability or numerical overflow.

Not all values have to strictly be only in the $[-1, 1]$ range because the model will perform well with data slightly outside of this range or with a few major outliers.

All datasets should be transformed in the same way, be it the training, validation, or testing data. This transformer should be saved along with the model and datasets and also applied during ***prediction***.

Data augmentation can be used in cases where the number of training samples is low. The idea is to copy the existing samples, make a slight modification to it that still retains all the required features that the model should be able to handle, and use it as an additional sample. An example would be a photo that can be safely rotated, cropped, or scaled.

The amount of required samples to train a model may vary greatly, and depends on the data complexity and model size. Usually a few thousand (or tens of thousands) per class or feature will be necessary.

1 Regularization

Regularization methods are used to reduce generalization error.

L1 and L2 Regularization

In ***L1*** and ***L2*** regularization a ***penalty*** value is calculated and added to the loss to penalize the model for large weights and biases. Large weights might indicate that a neuron is attempting to memorize a feature but in general it is better to have many neurons contributing to a model's output.

L1 regularization's penalty is the sum of all absolute values for the weights and biases:

$$L_{1_\theta} = \lambda \sum_m |\theta_m|$$

Where λ is a value representing the impact that this regularization carries and is usually a small value like 0.0005, and parameter m is an arbitrary iterator over all of the parameters (weights or biases) in a model.

L2 regularization's penalty is the sum of the squared weights and biases:

$$L_{2_\theta} = \lambda \sum_m \theta_m^2$$

L1 defines a linear penalty because the loss returned by this function is directly proportional to parameter values, whereas L2 defines a non-linear approach that penalizes larger parameters more than smaller ones. Because L1 penalizes small weights more than L2, it is rarely used alone and usually combined with L2, or not used at all. Note that regularization functions of this type drive the sum of parameters towards 0, which can help in cases of exploding gradients.

The overall loss is now defined as:

$$Q_i = L_i + L_{1_\theta} + L_{2_\theta}$$

Backward pass

For the backward pass we need to know the derivatives of L_{2_θ} and L_{1_θ}.

$$\begin{aligned}
\frac{\partial L_{2_\theta}}{\partial \theta_m} &= \frac{\partial}{\partial \theta_m}\left[\lambda \sum_m \theta_m^2\right] \\
&= \lambda \frac{\partial}{\partial \theta_m} \theta_m^2 \\
&= 2\lambda\theta_m
\end{aligned}$$

For L_{1_θ} the derivative of the absolute value piecewise function must be computed, which effetively multiplies a value by -1 if it is less than 0, and by 1 otherwise:

$$\begin{aligned}\frac{\partial L_{1_\theta}}{\partial \theta_m} &= \frac{\partial}{\partial \theta_m} \lambda \sum_m |\theta_m| \\ &= \lambda \frac{\partial}{\partial \theta_m} |\theta_m| \\ &= \lambda \begin{cases} 1 & \theta_m > 0 \\ -1 & \theta_m < 0 \end{cases}\end{aligned}$$

Regularization terms are usually only added to the hidden layers and not the output layer. In general, regularization allows to create much larger models (with many more neurons) without fear of overfitting.

Dropout

Another option for regularization is adding a ***Dropout Layer***. Dropout works by randomly disabling neurons at a given rate during every forward pass, forcing the network to learn how to make accurate predictions with only a random part of neurons remaining. It forces the model to use more neurons for the same purpose, resulting in a higher chance of learning the underlying function, and becoming much less dependent on any specific arrangement of neurons.

Dropout mitigates ***co-adoption*** and overfitting, and becomes more robust against ***noise*** and other perturbations in the training data. Co-adoption happens when neurons depend on the output values of other neurons and do not learn the underlying function by themselves.

A dropout layer "turns off" neurons with a filter with numbers drawn from a ***Bernoulli distribution***. This is a binary probability distribution which gives a value of 1 with a probability of p and a value of 0 with a probability of q. A random draw r_i from this distribution is given as:

$$\begin{aligned} r_i \sim \text{Bernoulli}(p) \text{ where} \\ P(r_i = 1) = p \\ P(r_i = 0) = q = 1 - P(r_i = 1)\end{aligned}$$

Where $P(r_i = x)$ represents the probability that r_i equals x.

The hyperparameter for a dropout layer is the ***dropout rate*** q; the value in range $\{0..1\}$ for the percentage of neurons to disable in a given layer. A Bernoulli distribution is a special case of a Binomial probability distribution $Pr(k, n, p)$ with $n = 1$, and is given as:

$$Pr(k, n, p) = \binom{n}{k} p^k \left(1 - p\right)^{n-k}$$

Where n is the number of trials (or experiments), k is the number of successes, and p the success probability for each trial. The output of a layer will be multiplied by the chosen neurons according to the probability distribution, where those that have not been selected are set to zero:

$$\theta = \theta \, \frac{Pr(1, 1 - p, \|\theta\|)}{(1 - p)}$$

Where θ are the output parameters of a layer, and $\|\theta\|$ is the size of the this array. The division by $(1 - p)$ scales the data after a dropout, to mimic the mean of the sum when all of the neurons output their values.

Backward Pass

For the backward pass we need the partial derivative of the dropoout operations. When the value of r_i equals 1, then its function and derivative become the neuron's output θ, compensated for the loss value by $1 - p$, where p is the dropout rate:

$$\frac{\partial}{\partial\theta}\left[\frac{z}{1-p}\right] = \frac{1}{1-p}\cdot\frac{\partial}{\partial\theta}\theta = \frac{1}{1-p}$$

And when $r_i = 0$ the dropout filter returns 0, and thus the derivative is also 0. Both cases combined yields:

$$Dr_i = \begin{cases} \frac{\theta_i}{1-p} & r_i = 1 \\ 0 & r_i = 0 \end{cases} \rightarrow \frac{\partial}{\partial\theta_i} Dr_i = \begin{cases} \frac{1}{1-p} & r_i = 1 \\ 0 & r_i = 0 \end{cases} = \frac{r_i}{1-p}$$

Where Dr_i is the dropout for given input at index i.