Neural Network Notes

Maurits Lam

Contents

Chapter 1

Concepts

1.1 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.

There are several layer types such as dense layers, dropout layers, convolutional layers, or recurrent layers. ***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***.

Dense layers learn feature values and the correlation between them, while ***convolutional*** layers are trained to find and understand features like characteristics or traits.

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.

1.2 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. Two of the most commonly used activation functions, ***Rectified Linear Unit*** (ReLU) and ***Softmax*** are such functions.

Rectified Linear Unit

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.

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

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

$$\text{Softmax}(\theta)_i = \frac{\mathrm{e}^{\theta_i}}{\sum_{k=1}^{K} \mathrm{e}^{\theta_k}} \text{ for } i = 1, \ldots, K.$$

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 $\hat{\boldsymbol{y}}$ 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 $\boldsymbol{y}$:

$$\text{Accuracy}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \frac{1}{\|\hat{\boldsymbol{y}}\|} \sum_{i=1}^{\|\hat{\boldsymbol{y}}\|} \left(\begin{cases} 1 & \text{argmax}(\hat{\boldsymbol{y}}_{\boldsymbol{i}}) = \boldsymbol{y}_i \\ 0 & \text{argmax}(\hat{\boldsymbol{y}}_{\boldsymbol{i}}) \neq \boldsymbol{y}_i \end{cases} \right)$$

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).

1.3 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:

$$\mathbf{S}_{i,j} = \frac{\mathrm{e}^{\theta_{i,j}}}{\sum_{k=1}^{K} \mathrm{e}^{\theta_{i,k}}} \quad \rightarrow \quad \frac{\partial \mathbf{S}_{i,j}}{\partial \theta_{i,m}} = \frac{\partial \frac{\mathrm{e}^{\theta_{i,j}}}{\sum_{k=1}^{K} \mathrm{e}^{\theta_{i,k}}}}{\partial \theta_{i,m}}$$

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

$$\begin{aligned}
\frac{\partial \mathbf{S}_{i,j}}{\partial \theta_{i,m}} &= \frac{\partial \frac{\mathrm{e}^{\theta_{i,j}}}{\sum_{k=1}^{K} \mathrm{e}^{\theta_{i,k}}}}{\partial \theta_{i,m}} \\
&= \frac{\frac{\partial}{\partial \theta_{i,m}} \mathrm{e}^{\theta_{i,j}} \cdot \sum_{k=1}^{K} \mathrm{e}^{\theta_{i,k}} - \mathrm{e}^{\theta_{i,j}} \cdot \frac{\partial}{\partial \theta_{i,m}} \sum_{k=1}^{K} \mathrm{e}^{\theta_{i,k}}}{[\sum_{k=1}^{K} \mathrm{e}^{\theta_{i,k}}]^2} \\
&= \frac{\frac{\partial}{\partial \theta_{i,m}} \mathrm{e}^{\theta_{i,j}} \cdot \sum_{k=1}^{K} \mathrm{e}^{\theta_{i,k}} - \mathrm{e}^{\theta_{i,j}} \cdot \frac{\partial}{\partial \theta_{i,m}} \sum_{k=1}^{K} \mathrm{e}^{\theta_{i,k}}}{[\sum_{k=1}^{K} \mathrm{e}^{\theta_{i,k}}]^2}
\end{aligned}$$

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 $j = m$:

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

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

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

In the case of $j \neq m$:

$$\begin{aligned}
\frac{\partial \mathbf{S}_{i,j}}{\partial \theta_{i,m}} &= \frac{\frac{\partial}{\partial \theta_{i,m}} \mathrm{e}^{\theta_{i,j}} \cdot \sum_{k=1}^{K} \mathrm{e}^{\theta_{i,k}} - \mathrm{e}^{\theta_{i,j}} \cdot \frac{\partial}{\partial \theta_{i,m}} \sum_{k=1}^{K} \mathrm{e}^{\theta_{i,k}}}{[\sum_{k=1}^{K} \mathrm{e}^{\theta_{i,k}}]^2} \\
&= \frac{0 \cdot \sum_{k=1}^{K} \mathrm{e}^{\theta_{i,k}} - \mathrm{e}^{z_{i,j}} \cdot \mathrm{e}^{\theta_{i,m}}}{[\sum_{k=1}^{K} \mathrm{e}^{\theta_{i,k}}]^2}
\end{aligned}$$

$$= \frac{-e^{z_{i,j}} \cdot e^{\theta_{i,m}}}{\sum_{k=1}^{K} e^{\theta_{i,k}} \cdot \sum_{k=1}^{K} e^{\theta_{i,k}}}$$
$$= -\frac{e^{\theta_{i,j}}}{\sum_{k=1}^{K} e^{\theta_{i,k}}} \cdot \frac{e^{\theta_{i,m}}}{\sum_{k=1}^{K} e^{\theta_{i,k}}}$$

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

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

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

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

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

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

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:

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

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 $\boldsymbol{y}$.

1.4 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:

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

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:

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

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.

1.5 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.

1.6 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:

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

$$= \lambda \frac{\partial}{\partial \theta_m} |\theta_m|$$
$$= \lambda \begin{cases} 1 & \theta_m > 0 \\ -1 & \theta_m < 0 \end{cases}$$

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. As such, dropout is a method to disable, or filter out, certain neurons in an attempt to regularize and improve the model's ability to generalize.

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:

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

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.

1.7 Binary logistic regression

Binary logistic regression is a model with an output layer where each neuron separately represents two classes: 0 for one of them, and 1 for the other. This single neuron could represent any combination of two classes, like two distinct classifications or a true/false representation, and there could be many of these output neurons.

In a ***regressor*** type of algorithm like binary logistic regression, the ***sigmoid*** activation function is used for the output layer rather than softmax, and ***binary cross-entropy*** for calculating loss instead of ***categorical cross-entropy***.

Sigmoid Activation Function

The ***Sigmoid*** or ***logistic*** function is one of the classic granular activation functions used in neural networks. It is used with regressors because it scales a range of outputs from negative to positive infinity to be between 0 and 1, and approaches both maximum and minimum values exponentially fast. Its equation is:

$$\sigma_{i,j} = \frac{1}{1 + \mathrm{e}^{-\theta_{i,j}}}$$

Where $\theta_{i,j}$, given indices, represents a singular output value of the layer that this activation function takes as input. Index i is the current sample, and index j is the current output in sample θ_i.

The derivative of the Sigmoid function is needed for the activation function's backwards step. Defining it with respect to its input yields:

$$\begin{aligned}
\sigma'_{i,j} &= \frac{\mathrm{d}}{\mathrm{d}\theta_{i,j}} \left[\frac{1}{1 + \mathrm{e}^{\theta_{i,j}}} \right] \\
&= \frac{\mathrm{d}}{\mathrm{d}\theta_{i,j}} (1 + \mathrm{e}^{-\theta_{i,j}})^{-1} \\
&= -(1 + \mathrm{e}^{-\theta_{i,j}})^{-2} \cdot \left(\frac{\mathrm{d}}{\mathrm{d}\theta_{i,j}} 1 + \frac{\mathrm{d}}{\mathrm{d}\theta_{i,j}} \theta_{i,j} \right) \\
&= -(1 + \mathrm{e}^{-\theta_{i,j}})^{-2} \cdot (\mathrm{e}^{-\theta_{i,j}} \cdot \frac{\mathrm{d}}{\mathrm{d}\theta_{i,j}} [-\theta_{i,j}]) \\
&= -(1 + \mathrm{e}^{-\theta_{i,j}})^{-2} \cdot -(\mathrm{e}^{-\theta_{i,j}}) \\
&= (1 + \mathrm{e}^{-\theta_{i,j}})^{-2} \cdot (\mathrm{e}^{-\theta_{i,j}}) \\
&= \frac{\mathrm{e}^{-\theta_{i,j}}}{(1 + \mathrm{e}^{-\theta_{i,j}})^2} \\
&= \frac{1}{1 + \mathrm{e}^{-\theta_{i,j}}} \cdot \frac{\mathrm{e}^{-\theta_{i,j}}}{1 + \mathrm{e}^{-\theta_{i,j}}} \\
&= \frac{1}{1 + \mathrm{e}^{-\theta_{i,j}}} \cdot \left(\frac{1 + \mathrm{e}^{-\theta_{i,j}}}{1 + \mathrm{e}^{-\theta_{i,j}}} - \frac{1}{1 + \mathrm{e}^{-\theta_{i,j}}} \right) \\
&= \frac{1}{1 + \mathrm{e}^{-\theta_{i,j}}} \cdot \left(1 - \frac{1}{1 + \mathrm{e}^{-\theta_{i,j}}} - \frac{1}{1 + \mathrm{e}^{-\theta_{i,j}}} \right) \\
&= \sigma_{i,j} \cdot (1 - \sigma_{i,j})
\end{aligned}$$

As it turns out, the derivative of the sigmoid function equals this function multiplied by the difference of 1 and the sigmoid function itself.

Binary Cross-Entropy Loss

Similar to categorical cross-entropy loss, in ***binary cross-entropy*** the loss can be determined using the negative log of the predictions. However, rather than only calculating this on the target class, the log-likelihoods of the correct and incorrect classes are summed for each neuron separately. Since class values are either 0 or 1, the incorrect class can be simplified to be a 1-***correct class*** by just inverting its value. The negative log-likelihood of the correct and incorrect classes is then calculated by adding them together:

$$\begin{aligned} L_{i,j} &= (y_{i,j})(-\log(\hat{y}_{i,j})) + (1 - y_{i,j})(-\log(1 - \hat{y}_{i,j})) \\ &= -y_{i,j} \cdot \log(\hat{y}_{i,j}) + (1 - y_{i,j}) \cdot \log(1 - \hat{y}_{i,j}) \end{aligned}$$

Where $L_{i,j}$ is calculated on a single output j, which is a vector of losses containing one value for each output. Because a model can contain multiple binary outputs, and each outputs its own prediction. To obtain the ***sample*** loss we need to calculate the mean of all of these losses from a single sample:

$$L_i = \frac{1}{J} \sum_j L_{i,j}$$

Where index i is the current sample, index j is the current output in this sample, and J is the number of outputs.

For backpropagation, we have to calculate the partial derivative of the sample loss with respect to each input:

$$\frac{\partial L_i}{\partial \hat{y}_{i,j}} = \frac{\partial L_i}{\partial L_{i,j}} \cdot \frac{\partial L_{i,j}}{\partial \hat{y}_{i,j}}$$

The first term is the partial derivative of the sample loss L_i with respect to a single output loss $L_{i,j}$:

$$\frac{\partial L_i}{\partial L_{i,j}} = \frac{\partial}{\partial L_{i,j}}[\frac{1}{J} \sum_j L_{i,j}] = \frac{1}{J} \cdot \frac{\partial}{\partial L_{i,j}} L_{i,j} = \frac{1}{J}$$

The second term is the ***atomic output loss***, a derivative of the single output loss $L_{i,j}$ with respect to the related prediction $\hat{y}_{i,j}$, and is given as:

$$\begin{aligned} \frac{\partial L_{i,j}}{\partial \hat{y}_{i,j}} &= \frac{\partial}{\partial \hat{y}_{i,j}} \left[-y_{i,j} \cdot \log(\hat{y}_{i,j}) - (1 - y_{i,j}) \cdot \log(1 - \hat{y}_{i,j})\right] \\ &= \frac{\partial}{\partial \hat{y}_{i,j}} \left[-y_{i,j} \cdot \log(\hat{y}_{i,j})\right] + \frac{\partial}{\partial \hat{y}_{i,j}} \left[-(1 - y_{i,j}) \cdot \log(1 - \hat{y}_{i,j})\right] \\ &= -y_{i,j} \cdot \frac{\partial}{\partial \hat{y}_{i,j}} \log(\hat{y}_{i,j}) - (1 - y_{i,j}) \cdot \frac{\partial}{\partial \hat{y}_{i,j}} \log(1 - \hat{y}_{i,j}) \\ &= -y_{i,j} \cdot \frac{1}{\hat{y}_{i,j}} \cdot \frac{\partial}{\partial \hat{y}_{i,j}} \hat{y}_{i,j} - (1 - y_{i,j}) \cdot \frac{1}{1 - \hat{y}_{i,j}} \cdot \frac{\partial}{\partial \hat{y}_{i,j}} [1 - \hat{y}_{i,j}] \\ &= -\frac{y_{i,j}}{\hat{y}_{i,j}} - \frac{1 - y_{i,j}}{1 - \hat{y}_{i,j}} \cdot \left(\frac{\partial}{\partial \hat{y}_{i,j}} 1 - \frac{\partial}{\partial \hat{y}_{i,j}} \hat{y}_{i,j}\right) \\ &= -\frac{y_{i,j}}{\hat{y}_{i,j}} - \frac{1 - y_{i,j}}{1 - \hat{y}_{i,j}} \cdot (0 - 1) \\ &= -\left(\frac{y_{i,j}}{\hat{y}_{i,j}} - \frac{1 - y_{i,j}}{1 - \hat{y}_{i,j}}\right) \end{aligned}$$

And so the equation of the partial derivative of a sample loss with respect to a single output loss becomes:

$$\frac{\partial L_i}{\partial \hat{y}_{i,j}} = \frac{\partial L_i}{\partial L_{i,j}} \cdot \frac{\partial L_{i,j}}{\partial \hat{y}_{i,j}} = -\frac{1}{J} \cdot \left(\frac{y_{i,j}}{\hat{y}_{i,j}} - \frac{1 - y_{i,j}}{1 - \hat{y}_{i,j}} \right)$$

This normalization has to be performed due to the fact that each output returns its own derivative. Without normalization each additional input will raise gradients and would require changing the hyper-parameters like the learning rate.

1.8 Regression

Aside from ***classification***, where it is determined what something is, the other main type of of model is ***regression***, which aims to determine a specific value. This requires a model that accepts scalar values, has more granular output, a new way to measure loss, and another output layer activation function.

Activation Function

Regression uses a linear activation function is one that simply returns the input as the output: $f(x) : y = x$. For the backwards pass the derivative becomes $\frac{\mathrm{d}f(x)}{\mathrm{d}x} = 1$.

Loss

The two main methods for calculating error in regression are ***mean squared error*** (MSE) and ***mean absolute error*** (MAE). Mean absolute error is actually just the same as L1 error, and mean squared error is like L2 error.

Mean Squared Error

In mean squared error, the difference between the predicted and true values of single outputs are squared and then averaged:

$$L_i = \frac{1}{J} \sum_j (y_{i,j} - \hat{y}_{i,j})^2$$

Where y is the target value, $\hat{y}$ means predicted value, i is the index of the current sample, j is the index of the current output in this sample, and J is the number of outputs.

The idea is to penalize more harshly the further away the predicted value is from the intended target.

Its partial derivative is given as:

$$\begin{aligned}
\frac{\partial}{\partial \hat{y}_{i,j}} L_i &= \frac{\partial}{\partial \hat{y}_{i,j}} \left[\frac{1}{J} \sum_j (y_{i,j} - \hat{y}_{i,j})^2 \right] \\
&= \frac{1}{J} \cdot \frac{\partial}{\partial \hat{y}_{i,j}} (y_{i,j} - \hat{y}_{i,j})^2 \\
&= \frac{2}{J} \cdot (y_{i,j} - \hat{y}_{i,j}) \cdot \frac{\partial}{\partial \hat{y}_{i,j}} [y_{i,j} - \hat{y}_{i,j}] \\
&= \frac{2}{J} \cdot (y_{i,j} - \hat{y}_{i,j}) \cdot (\frac{\partial}{\partial \hat{y}_{i,j}} y_{i,j} - \frac{\partial}{\partial \hat{y}_{i,j}} \hat{y}_{i,j}) \\
&= \frac{2}{J} \cdot (y_{i,j} - \hat{y}_{i,j}) \cdot (0 - 1) \\
&= -\frac{2}{J} \cdot (y_{i,j} - \hat{y}_{i,j})
\end{aligned}$$

Mean Absolute Error

In mean absolute error the absolute difference between the predicted and true values in a single output is computed and then averaged over all the outputs:

$$L_i = \frac{i}{J} \sum_j |y_{i,j} - \hat{y}_{i,j}|$$

Where y is the target value, $\hat{y}$ is the predicted value, i is the index of the current sample, j is the index of the current output in this sample, and J is the number of outputs.

Mean absolute error, when used as a loss function, penalizes the error linearly. It produces sparser results than MSE and is robust to outliers, which can be both adventageous and disadvantageous. This is why MAE is used less frequently than MSE loss.

The partial derivative for absolute error with respect to the predicted values is:

$$\begin{aligned} \frac{\partial}{\partial \hat{y}_{i,j}} L_i &= \frac{\partial}{\partial \hat{y}_{i,j}} \left[\frac{1}{J} \sum_j |y_{i,j} - \hat{y}_{i,j}| \right] \\ &= \frac{1}{J} \cdot \frac{\partial}{\partial \hat{y}_{i,j}} |y_{i,j} - \hat{y}_{i,j}| \\ &= \frac{1}{J} \cdot \begin{cases} +1 & y_{i,j} - \hat{y}_{i,j} > 0 \\ -1 & y_{i,j} - \hat{y}_{i,j} < 0 \end{cases} \end{aligned}$$

This function is undefined at 0, but in the implementation it should return 0 when $y_{i,j} - \hat{y}_{i,j} = 0$.

Accuracy

To measure performance of the model there are to problems. The first is that each output neuron represents a separate output that cannot be combined to calculate accuracy. The second is that the prediction is a scalar value that cannot be directly compared to the ground truth one due to floating-point errors. In actuality, there is no perfect way to show accuracy with regression so we need to simulate or approximate it.

One method is to use a precision value that is calculated using the standard deviation from the ground truth target values and dividing them by some "limit" λ. The larger this value is, the more strict the accuracy metric will be:

$$\rho = \frac{\sigma(y)}{\lambda} = \frac{1}{\lambda} \cdot \sqrt{\frac{\sum_j (y_j - \mu)^2}{J}}$$

Where j is one output, J is the number of outputs, or the size of the population, and μ is the ***population mean***.

This precision value can be used as an "allowance" for regression outputs when comparing targets and predicted values for accuracy. The comparison is computed by applying the absolute value on the difference between the ground truth and the predictions. Then we check if the difference is smaller than the previously calculated precision:

$$\frac{1}{J} \cdot \sum_j^J (|\hat{y}_j - y_j| < \rho)$$

Glorot

In the backpropagation step, the gradient is calculated using weights, and the learning rate does not effect it. This is why it is important to use the correct weight initialization. In the ***Glorot uniform***

initialization method, the fraction that multiplies the draw from the uniform distribution is not constant but depends on the number of inputs and the number of neurons. Using this technique to initialize weights can improve the model in cases where the model is not learning at all.

1.9 Data sets

The following ***preprocessing*** steps must be undertaken for a training dataset:

1. Balancing
2. Augmentation (if applicable)
3. Grayscaling (for images)
4. Scaling
5. Flattening
6. Shuffling

A dataset used for classification is ***balanced*** if each class occurs with the same frequency. If a dataset is not balanced, a model will likely become biased to predict the class containing the most images. This is because neural networks seek out the steepest and quickest gradient descent to decrease loss, which is most easily found in a local minimum of the most prominent specific class.

In an imbalanced dataset, it is better to trim away samples from the high-frequency classes so that each class has the same amount of samples. Another option for image datasets is to ***augment*** the samples by cropping, rotating, or flipping input images, thus creating new samples to balance out the dataset.

It is common practice to ***grayscale*** input images and to resize them to normalize their pixel dimensions so they will have the same amount of input values. This results in grayscale values in the $[0, 255]$ range for each pixel.

Next, the data needs to be ***scaled*** into a $[0, 1]$ or $[-1, 1]$ range. The manner of how to best scale a dataset takes some experimentation and trial and error. If linear scaling does not suffice because of outliers, some combination of the average value and standard deviation may prove useful. Both the training, testing, and prediction data must be scaled using identical methods. However, any preprocessing transformations must be informed only from the training dataset, and not be derived from the testing dataset.

Since dense layers operate on batches of 1-dimensional vectors, any 2-dimensional features (such as pixels) must be ***flattened*** into a singular row. Note that some convolutional neural networks will allow passing 2D image arrays as-is.

Another important transformation is ***shuffling***: the process of randomizing the order of the input classes. When all classes are clumped together in chunks, the training step will find new local minima for each classification, likely spiking on the final classification in the order and never finding a global minimum. It is important to realize that this preprocessing step is only necessary when training is done in various batches of data, but this is the general use case.

Batches

A batch is a slice of a fixed size from a preprocessed dataset. Training with batches means iterating through the dataset in a chunk of data at a time, performing the forward pass, loss calculation, backward pass, and optimization.

Each batch should be large enough and representative of the dataset, so that the result of training on a batch is a good approximation of the direction towards a global minimum. If the batch size is too small, the direction of gradient descent can fluctuate too much, causing the model to take a long time to train.

Common batch sizes range between 32 and 128 samples. These can be made smaller in case of memory restraints, or larger to increase the speed (but not necessarily effectiveness) of training. Loss and accuracy improvements can usually be seen by increasing the batch size by a factor of 2 or 3. At some point increasing it will have diminishing returns and will become slow.

Each batch of samples being trained is referred to as a ***step***. The outer loop of the training algorithm iterates over the number of epochs and the inner one over the number of steps. Both the batch-wise and epoch-wise loss and accuracy statistics are tracked. The overall epoch-wise loss and accuracy are sample-wise averages.

1.10 Models

During model training, the usual ***evaluation*** process is to analyze its performance, then adjust the hyperparameters, and retrain it using the training and validation data. Then, when the model and hyperparameters that appear to perform the best are found, that model is used on testing data and to make predictions in production.

Running evaluation on the training data *at the end of the training process* will return the final accuracy and loss. There will be some difference between this mean accuracy and loss and those accumulated during an epoch while the model was still learning.

State

A model's state should be able to be saved for future use in ***predictions*** or as a ***checkpoint*** during the training process. There are two options that have different use cases: either only the parameters (weights and biases) are saved, or the whole model with all hyperparameters, layers, and optimizer state.

Saved parameters can be used to initialize a model that has been trained from similar data and then applied to work with specific data. This is called ***transfer learning***. Parameters can also be used to visualize the model, to identify dead neurons, or for ***reinforcement learning***, where weights collected from multiple models are committed to a single network. While it is possible to train the model further using just parameters, it is more optimal to load a full model including the optimizer's state.

Prediction

Prediction, or ***inference***, is the process of loading a trained model and using it to predict the correct classification of a never-before-seen input such as an image.

The output of a trained classification model is an array of ***confidence*** vectors that contain a confidence metric per class. The highest confidence score points to the ***prediction***. With a softmax classifier the predicted class is simply the argmax of each confidence vector.

1.11 Convolutional neural network

A **Convolutional Neural Network** (CNN) is a type of deep neural network designed for processing data with grid-like topology, such as images. Local features like edges, textures, or shapes are extracted by sliding kernels over the image. Spatial patterns are learned through **convolutional** layers and combined with **pooling** layers to reduce the spatial dimensions of feature maps while capturing dominant features and making the model more efficient.

Convolutional Layer

This layer is the backbone of a CNN. Its forward pass applies the **convolution operation** to the input (image or other feature map), producing a **feature map** that highlights specific patterns like edges or textures.

The convolution operation involves applying a **kernel** (or filter) to the input image, which slides over the image and computes the dot product between the kernel and a local region of the image. This results in a feature map that highlights specific patterns or features in the image. Mathematically, the **convolution** operation can be represented as:

$$S(i,j) = \sum_m \sum_n I(i+m, j+n) \cdot K(m,n)$$

where:

- $S(i,j)$: output feature map at location (i,j).
- $I(i,j)$: pixel value at location (i,j).

- $K(m, n)$: weight of the kernel at location (m, n).

The kernel is a small matrix of weights that is learned during training. The size of the kernel determines the size of the local region being analyzed. Common kernel sizes are 3×3, 5×5, or 7×7 pixels.

In the **backward pass**, we need to know how much the input to the convolutional layer contributed to the final loss and adjust the kernel so that it leads to less error in the future. This is done by convolving the original layer's input with the gradient of the loss with respect to the output feature map $S(i, j)$:

$$\frac{\partial L}{\partial K} = \sum_i \sum_j \frac{\partial L}{\partial S(i, j)} \cdot I(i + m, j + n)$$

where:

- L: loss function.
- K: kernel weights.
- $S(i, j)$: output feature map at location (i, j).
- $I(i + m, j + n)$: input image at location $(i + m, j + n)$.

Pooling Layer

The **pooling** layer reduces the spatial dimensions of feature maps while preserving their most important features. This helps to decrease computational complexity and to prevent overfitting. The two most common types of pooling are **max** pooling and **average** pooling.

Max pooling selects the maximum value within a region, emphasizing the most prominent features. The output at a given location is the maximum value within the pooling window. Average pooling computes the mean value of the elements in the window, providing a smoother representation. For a pooling window of size $p \times p$, the output can be expressed as:

$$P(i, j) = \max_{m,n}^{p} I(i + m, j + n)$$

where:

- $P(i, j)$: output feature map at location (i, j).
- $I(i + m, j + n)$: input feature map at location $(i + m, j + n)$.
- p: pooling window size.
- m: index for the height of the pooling window.
- n: index for the width of the pooling window.

During the backward pass, the pooling layer propagates gradients to the input feature map. For max pooling, the gradient is passed only to the input element that had the maximum value in the pooling window. For average pooling, the gradient is distributed equally among all elements in the window. This ensures that the pooling operation contributes to the optimization of the network during training.

Batch Normalization

Batch normalization is a technique to improve the training of deep neural networks by normalizing the inputs to each layer. This helps to stabilize and accelerate training by reducing internal **covariate shift**, which occurs when the distribution of inputs to a layer changes during training. Batch normalization reduces sensitivity to weight initialization and acts as a regularizer, allowing for higher learning rates and possibly reducing the need for dropout layers as well.

In convolutional layers, batch normalization is typically applied to the feature maps along the channel dimension, ensuring that each channel is normalized independently.

During the forward pass, the input feature map is normalized by subtracting the batch mean μ_B and dividing by the batch standard deviation σ_B^2. It then scales and shifts the normalized values using learnable parameters. It can be mathematically expressed as:

$$y = \gamma \frac{x - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} + \beta$$

where:

- x: input feature map.
- μ_B: batch mean.
- σ_B^2: batch variance.
- ϵ: small constant to prevent division by zero.
- γ: learnable scale parameter.
- β: learnable shift parameter.

During the backward pass, gradients are computed with respect to input x, γ, and β, allowing the network to learn the optimal scaling and shifting parameters.

Chapter 2

Terminology

2.1 Mathematics

Tensor Mathematical object that generalizes the concepts of scalars, vectors, and matrices. Tensors can be thought of as multi-dimensional arrays of numbers. There are different types of tensors, characterized by their rank. Tensors of rank greater than 2 are referred to as higher-order tensors.

Shape Indication of the size of each dimension along which the tensor is defined. In general, the shape of a tensor is represented as a tuple of integers, where each integer represents the size of a particular dimension of the tensor. So, for an n-dimensional tensor, the shape would be represented by a tuple of n integers.

Categorical Data Type of data that represents categories or labels, rather than numerical values. It is used to classify items into groups or classes based on qualitative attributes or characteristics. There are two types of categorical data.

Ordinal data represents categories with a natural order or ranking. While the categories are still discrete, they have a meaningful sequence or hierarchy. This kind of data can easily be converted to numerical data with ordinal or integer encoding.

Nominal data represents categories that have no inherent order or ranking. Each category is distinct and unrelated to the others. One-hot encoding can be applied for the class labels.

One-Hot Encoding Technique used to represent categorical variables as binary vectors in machine learning and data processing. It is commonly used when dealing with categorical data, such as class labels, where the values are discrete and unordered.

In one-hot encoding, each category or class is represented as a binary vector where only one element is "hot" (set to 1) while all other elements are "cold" (set to 0).

One-hot encoding is useful because it allows machine learning algorithms to work with categorical data, which are typically represented as strings or integers, in a numerical format that can be fed into models for training and prediction. Additionally, it ensures that the encoded features are equidistant from each other, which can prevent certain models from assuming false ordinal relationships between categories.

Gradient Vector of partial derivatives for a multivariate function that points in the direction in which the function increases most quickly. Its components are the partial derivatives of that function with respect to each of its variables.

In optimization algorithms such as gradient descent, the gradient is used to update the parameters of a model in order to minimize a loss function. By iteratively following the direction of the negative gradient, one can descend to a local minimum of the function.

Jacobian Matrix Jacobian matrix is a matrix of (first-order) partial derivatives. It describes the rate of change of a multivariable function with respect to its input variables.

The Jacobian matrix provides important information about the local behavior of a function near a specific point. It describes how small changes in the input variables result in changes to the output variables.

Jacobian Determinant Measures the volume scaling factor associated with the transformation of a multivariate function near an input point. It provides important information about the local behavior (expansion or contraction) of functions and is closely related to volume scaling and change of variables.

Likelihood Function Represents the probability of the observed data given a set of parameters. It is a fundamental concept in likelihood-based methods, such as maximum likelihood estimation. The likelihood function plays a central role in statistical inference where the goal is to find the set of parameters that maximizes the likelihood function. It is often used to make inferences about these parameters, such as estimating their values or testing hypotheses about them.

Log-Likelihood The natural logarithm of the likelihood function. It is often used in statistical inference and parameter estimation, particularly in cases where the likelihood function involves products of many probabilities, which can be computationally cumbersome to work with directly.

Taking the logarithm of the likelihood function has the advantage that maximizing the log-likelihood is equivalent to maximizing the likelihood itself, and that the log-likelihood function has a single global maximum (it is concave), which efficiently facilitates gradient descent.

Global and Local Minimum A local minimum is a point where the function takes on its smallest value in a small neighborhood around that point. A global minimum is a point where the function takes on its smallest value over its entire domain.

In graphical terms, you can visualize a local minimum as a "valley" where the function is lower than its neighboring points but may not be the absolute lowest point in the entire graph. On the other hand, a global minimum represents the lowest point on the entire graph.

The presence of multiple local minima can make optimization problems challenging, especially in non-convex functions, where gradient-based optimization methods may converge to local minima instead of the global minimum.

Moving/Rolling Average Statistical technique used to smooth out fluctuations in data over time by creating a series of averages of different subsets of the full dataset. It is commonly used in time series analysis to identify trends and patterns in data.

The basic idea behind a moving average is to calculate the average value of a fixed number of consecutive data points (referred to as the "window" or "period") and use this average to represent the underlying trend in the data. As new data becomes available, the moving average is recalculated by shifting the window along the dataset. They help to reduce noise and highlight underlying trends or patterns in the data.

Mean Squared Error Common metric used to measure the average squared difference between ground truth values and predicted values produced by a model. It's widely used in regression analysis to evaluate the performance of a predictive model.

The MSE quantifies the average magnitude of errors made by the model. A lower MSE indicates that the model's predictions are closer to the actual values, while a higher MSE indicates larger discrepancies between the predicted and actual values.

MSE is a useful metric because it penalizes larger errors more heavily than smaller errors due to the squaring operation. However, it has the drawback of being sensitive to outliers.

Mean Absolute Error Metric used to measure the average absolute difference between ground truth values and predicted values. It is widely used in regression analysis to evaluate the performance of a predictive model.

The MAE measures the average absolute difference between the predicted and actual values, and therefore, it quantifies the average magnitude of errors made by the model without considering their direction. A lower MAE indicates that the model's predictions are closer to the actual values.

MAE is less sensitive to outliers compared to Mean Squared Error (MSE) because it does not square the errors. However, it treats positive and negative errors equally, which may not be desirable.

Kronecker Delta The Kronecker delta is a mathematical tool that is defined as equal to 1 if indices given i and j are equal, and it is equal to 0 if the indices i and j are different.

Bernoulli Distribution Discrete probability distribution that models a single random experiment with two possible outcomes: success and failure. Each trial has only two possible outcomes, often denoted as "success" and "failure." These outcomes are mutually exclusive and exhaustive, meaning that one and only one of them must occur in each trial, and the probabilities of success and of failure remain constant from trial to trial.

The Bernoulli distribution is often used as a building block for more complex probability distributions and statistical models. It serves as the foundation for the binomial distribution, which describes the number of successes in a fixed number of independent Bernoulli trials. Examples of Bernoulli experiments include flipping a coin.

Moments The first and second moments are two fundamental moments often used to characterize a probability distribution. The first moment (mean) provides information about the center of the distribution, while the second moment (variance) provides information about the spread or

dispersion of the distribution around the mean. Together, these moments offer key insights into the shape and characteristics of the probability distribution of a random variable.

The mean (μ) is calculated as the weighted average of all possible values of the random variable, where each value is weighted by its respective probability.

The variance (σ^2) is calculated as the weighted average of the squared deviations of all possible values of the random variable from the mean, where each squared deviation is weighted by its respective probability.

Note that a standard deviation (σ) is the square root of the variance. It is a more intuitive measure of dispersion compared to variance because it is expressed in the same units as the original data.

k-Fold Cross-Validation Technique used in machine learning for evaluating the performance of a predictive model. It helps to assess how well a model generalizes to new data by partitioning the available dataset into multiple subsets, called folds, and iteratively training and evaluating the model on different combinations of these folds.

The original dataset is randomly partitioned into K equal-sized subsets or folds. The model is trained on $K-1$ folds (the training set), and evaluated on the left-out fold (the validation set) to compute a loss metric. After K iterations, all metrics are combined to obtain a single estimate of the model's performance.

Kernel A kernel is a small matrix of learnable weights used to extract features from input data. It slides over the input (e.g., an image) and performs element-wise multiplication followed by summation, producing a feature map that highlights specific patterns like edges or textures.

Covariate shift Covariate shift occurs when the distribution of the input data during training differs from the distribution during testing or deployment. This mismatch can degrade the model's performance because a network learns patterns based on the training distribution, which may not generalize well to the new data. Techniques like domain adaptation, data augmentation, or normalization can help mitigate this issue.

2.2 Machine Learning

Machine learning is a subfield of Artificial Intelligence that focuses on the development of algorithms and models that allow computers to learn from data and make predictions or decisions without being explicitly programmed for every task. In essence, machine learning algorithms enable computers to learn patterns and relationships from data and use this knowledge to make decisions or predictions on new, unseen data.

The three main categories of machine learning are supervised learning, unsupervised learning, and reinforcement learning.

Pattern Recognition Pattern recognition is a field within machine learning that focuses on the automated identification of patterns or regularities in data. It involves the development of algorithms and techniques to recognize and classify patterns in various types of data, such as images, text, audio, or time series.

The goal of pattern recognition is to extract meaningful information from raw data and use it to make decisions or predictions.

Neural Network Class of machine learning models inspired by the structure and function of the human brain. They consist of interconnected layers of nodes (neurons) and are capable of learning complex patterns and relationships in data. Examples include feedforward neural networks, convolutional neural networks, and recurrent neural networks.

A model refers to the representation of a system or a process that is learned from data. It encapsulates the relationship between input data and the corresponding output, allowing the model to make predictions or decisions about new, unseen data. A neural network is one type of model.

Once a neural network is trained on a dataset, it can be used to make predictions or decisions about new, unseen data. The process of training a model involves feeding it with labeled data (in supervised learning) and adjusting the model parameters to minimize a loss function, which quantifies the difference between the model's predictions and the true values.

Deep neural networks can have various architectures, including fully connected (or dense) networks, convolutional neural networks for image data, or recurrent neural networks for sequential data. The choice of architecture depends on the nature of the input data and the specific task to be performed.

Deep Learning Deep learning is a subfield of machine learning that focuses on the development and

training of artificial neural networks with multiple layers.

Deep learning models automatically learn hierarchical representations of data, starting from low-level features (such as edges in images or phonemes in speech) and progressing to higher-level features (such as object shapes or spoken words). Deep learning models can scale effectively to large datasets and complex problems and learn to automatically extract useful representations of data, which can be transferred to other tasks or domains with minimal adaptation. This property makes deep learning particularly well-suited for transfer learning and multitask learning.

Feedforward Neural Network Also known as a multilayer perceptron, a feedforward neural network one of the simplest and most common types of artificial neural networks used in machine learning and deep learning. It consists of multiple layers of neurons, where each neuron in one layer is connected to every neuron in the next layer, and information flows in one direction, from the input layer to the output layer, without any feedback loops.

During the training process, the weights and biases of the neurons in the network are iteratively adjusted using optimization algorithms such as gradient descent, so that the network learns to produce the correct outputs for a given set of inputs. This process involves forward propagation (computing a prediction) and backward propagation (computing the gradients of the loss function with respect to the weights and biases), hence the name "feedforward" neural network.

Feedforward neural networks are versatile and can be applied to a wide range of tasks, including classification, regression, and function approximation. However, they may struggle with capturing long-range dependencies in sequential data, such as natural language or time-series data.

Convolutional Neural Network A specialized type of deep neural network designed to process and analyze spatial data, such as images and videos. CNNs are widely used in computer vision tasks, including image classification, object detection, and segmentation. They consist of multiple layers, including convolutional layers, pooling layers, and fully connected layers.

Recurrent Neural Network Type of neural network designed to process and analyze sequential data, such as time series, text, audio, and video. RNNs are characterized by their ability to maintain a memory of previous inputs through hidden states, allowing them to capture temporal dependencies and patterns in the data.

Supervised Learning

Type of machine learning where the goal of is to learn a mapping from input features to output labels or values based on the examples provided in the training data. Each training example consists of an input feature vector and a corresponding output label or value. The algorithm learns from these examples by adjusting its internal parameters to minimize the error between its predictions and the true labels in the training data.

Supervised learning algorithms learn to generalize from the training data to make predictions or decisions on new, unseen data. The quality of the predictions depends on factors such as the choice of algorithm, the quality and quantity of the training data, the selection of features, and the model's capacity to capture complex relationships in the data.

The two main types of supervised learning tasks are classification and regression.

Regression Supervised learning task where the goal is to predict a continuous numerical value based on input features. The input data consists of a set of features or attributes, and each example is associated with a continuous target value. A regressor's task is to learn a mapping from the input features to the output target values based on the examples provided in the training data. The output variable is continuous and represents a quantity.

Classification Supervised learning task where the goal is to predict the categorical label or class of an input data point based on its features. The input data consists of a set of features or attributes, and each example is associated with a class label indicating its category or class. A classifier's task is to learn a mapping from the input features to the output labels based on the examples provided in the training data. The output variable is discrete and represents a category or class.

Classification can be binary or multiclass. Confusingly, binary classification is also called binary logistic regression, where the output is a probability score between 0 and 1, representing the likelihood that an observation belongs to the target class.

Confidence In the context of classification tasks, a confidence score represents the model's confidence in its prediction of the class label for a given input data point. It typically reflects the probability that the predicted class is correct.

In regression tasks a confidence score may represent the level of uncertainty associated with the predicted numerical value. It can indicate the range of values within which the true value is likely to fall.

Logits Logits represent the raw (unnormalized) predictions generated by a model before applying an activation function. They are typically the outputs of the last layer of a neural network.
Logits are real-valued scores assigned by the model to each class, indicating the likelihood or confidence of the input belonging to each class. Unlike probabilities, logits are not constrained to the range [0, 1] or summing up to one. They can be any real number, positive or negative.
After obtaining the logits, they are usually transformed into probabilities using the softmax function. The softmax function normalizes the logits to produce a probability distribution over the classes, ensuring that the probabilities sum up to one.

Inference Once a classification model has been trained on a labeled dataset, it learns the relationship between the input features and the corresponding class labels. During inference, the trained model takes the features of a new data point as input and predicts the class label or category that the data point belongs to.
The process of inference typically involves the following: input preprocessing, prediction, decision, and (optionally) evaluation.

Unsupervised Learning

Unsupervised learning is a type of machine learning where the algorithm learns patterns, structures, or relationships in the data without explicit supervision or labeled output. In unsupervised learning, the algorithm is given input data without corresponding output labels, and its goal is to discover hidden patterns or structure in the data.

The primary objective of unsupervised learning is to find inherent structures within the data, such as clusters, subgroups, or underlying distributions. Unlike supervised learning, where the algorithm is provided with labeled examples and learns to map input data to output labels, unsupervised learning algorithms operate on raw, unlabeled data and are tasked with extracting meaningful information from it.

Common unsupervised learning tasks are clustering and anomaly detection.

Clustering Placeholder
Anomaly Detection Placeholder

Neuron

A neuron is a fundamental building block in a neural network that mimics the behavior of biological neurons in the human brain. Neurons process information by receiving input signals, performing computations, and producing output signals. A neuron typically has weights, a bias, and an activation function.

Neurons are organized into layers, with each layer consisting of one or more neurons. The input layer receives the input data, hidden layers perform intermediate computations, and the output layer produces the final output of the network. The connections between neurons, along with their associated weights, form the structure of the neural network, allowing it to learn complex patterns and relationships in the data.

During the training process, the weights of the neurons are adjusted iteratively using optimization algorithms such as gradient descent, so that the network learns to produce the correct outputs for a given set of inputs. This process of learning from data is known as training the neural network, and it involves adjusting the weights to minimize a loss function that measures the difference between the predicted outputs and the true outputs.

Weights Parameters that determine the strength of the connections between neurons in adjacent layers. Each neuron connected to neurons in the subsequent layer has an associated weight. These weights are adjusted during the training process to minimize the difference between the predicted outputs of the network and the true labels of the training data.

Biases Additional parameters added to each neuron in the network, which allow the network to learn more complex functions. Biases shift the output of each neuron to the left or right, affecting the

activation function's decision boundary. Like weights, biases are learned during training to improve the performance of the network.

Glorot Uniform Also known as Xavier uniform initialization, Glorot Uniform is a method used to initialize the weights of neural network layers in a principled way.
The idea is to scale the initial weights of each layer in such a way that the variance of the activations remains approximately constant across different layers of the network. This helps to ensure that the gradients neither explode nor vanish during the training process, improving training efficiency.
Another initialization technique is Kaiming Initialization or He Initialization.

Model Fitting

Model fitting, also known as model training or model estimation, is the process of adjusting the parameters of a machine learning model to minimize the difference between the model's predictions and the actual target values in the training data. The objective of model fitting is to create a model that accurately captures the underlying patterns or relationships in the data and can generalize well to new, unseen data.

In model fitting, the algorithm iteratively adjusts the parameters of the model using an optimization algorithm to minimize a predefined loss function. The loss function measures the discrepancy between the predicted values of the model and the actual target values in the training data. The goal is to find the values of the model parameters that minimize this discrepancy, effectively "fitting" the model to the training data.

Epoch Represents one complete forward, backward, and optimization pass through the entire training dataset during the training of a machine learning model. Multiple epochs are typically required to train a model effectively, with each epoch allowing the model to learn from and adjust to the training data incrementally.
Choosing the appropriate number of epochs depends on various factors, including the complexity of the model, the size of the dataset, and the desired level of convergence.
After completing all epochs, the training process is complete, and the model can be evaluated on a separate validation or test dataset to assess its performance.

Hyperparameters Hyperparameters are configuration settings that govern the learning process of a model but are not learned from the data. These parameters are set before the learning process begins and remain fixed throughout training. Unlike model parameters, which are learned during training (such as weights in a neural network), hyperparameters control aspects of the learning algorithm itself, such as its complexity, capacity, or optimization behavior.
Some common examples of hyperparameters include the number of epochs, learning rate, learning rate decay, regularization strength, number of hidden layers, activation functions, batch size, and dropout rate.

Convergence Refers to the process by which the parameters of the network, such as weights and biases, gradually adjust during training to minimize the loss function and improve the model's performance on the training data. In other words, convergence occurs when the optimization algorithm used to train the neural network finds parameter values that result in a satisfactory level of performance on the training dataset.
The goal of training a neural network is to find the optimal set of parameters that minimize a predefined loss function, which measures the discrepancy between the model's predictions and the true labels in the training data. Convergence is achieved when the optimization algorithm successfully finds parameter values that lead to a sufficiently low value of the loss function.

Generalization The ability of a trained model to perform well on new, unseen data that it hasn't been exposed to during training. In other words, a model that generalizes well can make accurate predictions or classifications on data it hasn't encountered before, beyond the examples it was trained on.
The ultimate goal of machine learning is to develop models that can generalize well to real-world scenarios and make reliable predictions on unseen data. Generalization is a crucial property of machine learning models because it ensures that they are able to capture underlying patterns and relationships in the data, rather than simply memorizing the training examples.

Memorization Refers to a phenomenon where a model learns to memorize the training data instead of learning the underlying patterns or relationships within the data. Memorization occurs when a model becomes overly complex and captures noise or specific characteristics of the training data that are not representative of the underlying distribution.

Memorization is typically associated with overfitting, which is a situation where a model performs well on the training data but fails to generalize to new, unseen data. Instead of learning the true underlying patterns, an overfitted model has essentially "memorized" the training examples and their corresponding labels.

Overfitting Common problem in machine learning where a model learns to capture noise or random fluctuations in the training data rather than the underlying patterns or relationships. Overfitting occurs when a model becomes overly complex and learns to fit the training data too closely, to the extent that it performs poorly on new, unseen data.

Memorization and overfitting can be mitigated by techniques such as regularization, cross validation, reducing model/parameter complexity, early stopping, or data augmentation.

Underfitting Problem where a model is too simple to capture the underlying patterns or relationships in the data. Underfitting occurs when a model is not able to learn from the training data effectively, resulting in poor performance on both the training data and new, unseen data.

An underfitted model fails to capture the complexity of the underlying data and is unable to make accurate predictions or classifications. It typically exhibits high bias and low variance.

Noise Random variations or disturbances in the input data, model parameters, or the training process itself. By effectively managing noise, neural networks can achieve better performance, robustness, and generalization to new data.

Data noise can include measurement errors, missing values, outliers, or irrelevant features. Data preprocessing techniques such as filtering, smoothing, and outlier detection can help mitigate data noise.

Model noise are random fluctuations or uncertainty in the model parameters or architecture. This can occur due to initialization randomness, numerical instability, or stochastic optimization algorithms. Regularization techniques such as weight decay or dropout can help reduce model noise by promoting smoother or more stable parameter values.

Training noise refers to variations or randomness introduced during the training process. This can include mini-batch sampling noise, learning rate scheduling, or early stopping. Techniques such as batch normalization or adaptive learning rate methods can help stabilize the training process and reduce training noise.

Co-Adaptation Phenomenon where certain neurons or groups of neurons within the network become highly interdependent during training. This interdependence can result in the network relying too heavily on specific subsets of neurons to make predictions, potentially leading to inefficiencies, reduced generalization performance, or overfitting.

Co-adaptation occurs when the optimization process of training the neural network encourages certain neurons to specialize in representing specific patterns or features in the training data. While specialization can be beneficial to some extent, excessive co-adaptation can hinder the network's ability to generalize to new, unseen data because it may become overly reliant on these specialized neurons.

One common example of co-adaptation is the phenomenon known as "dead neurons," where certain neurons become inactive or do not contribute meaningfully to the network's predictions. Dead neurons can occur when a neuron's weights are initialized in such a way that they are unable to effectively update during training, or when the neuron becomes saturated due to the activation function used.

Co-adaptation can be mitigated with regularization, early stopping, cross-validation, or redesigning a network's architecture.

Dead Neurons Neurons within a neural network that do not contribute to the learning process and remain inactive throughout training and inference. Dead neurons can occur due to various reasons and are typically associated with vanishing gradients, which result in the neuron's weights not being updated during training.

Some potential causes include: initialization issues, zero activation, saturated activation, low learning rates. These can be combated by choosing proper activation functions, initialization techniques (like Glorot or He), and applying regularization (such as dropout or batch normalization).

By addressing dead neurons and ensuring that all neurons actively contribute to the learning process, the overall performance and effectiveness of the neural network can be enhanced.

Expoding Gradients Phenomenon that can occur during the training of neural networks, particularly deep neural networks, where the gradients of the loss function with respect to the model parameters become extremely large. This can lead to numerical instability during optimization, causing the model's weights to grow exponentially and the training process to diverge.

Gradient explosion is the opposite of gradient vanishing, where gradients become extremely small, and it tends to occur in deep networks with a large number of layers, particularly during backpropagation, the process of computing gradients and updating the weights of the network.
Several factors can contribute to this explosion: poor weight initialization, unstable architecture, high learning rates, or poorly scaled features.
They can be mitigated with proper weight initialization, gradient clipping, and lowering learning rates.

Transfer Learning Learning technique where knowledge gained from training a model on one task is leveraged to improve performance on a related but different task. Instead of training a model from scratch on the target task, transfer learning involves using a pre-trained model as a starting point and fine-tuning it on the new task.
The core idea behind transfer learning is that features learned by a model on one task may be useful for solving a related task. By transferring knowledge from the source task to the target task, transfer learning can help improve model performance, reduce the amount of labeled data needed for training, and speed up the training process.

Activation Function

An activation function is applied to the output of neurons to introduce nonlinearity into a neural network. They aid in learning complex patterns and relationships in the data by enabling them to model nonlinear mappings between inputs and outputs. Activation functions play a crucial role in determining the output of a neuron and the overall behavior of the network.

The choice of activation function depends on the specific requirements of the task, the characteristics of the data, and the architecture of the neural network.

Linear Also known as the identity activation function, the linear activation function is a simple mathematical function that computes the output as a linear transformation of the input. In other words, the output of a neuron with a linear activation function is directly proportional to its input, with no non-linear transformation applied.

Sigmoid Also known as the logistic sigmoid function, it is a non-linear mathematical function that maps any real-valued number to a value between 0 and 1. It is commonly used in binary classification tasks, where the goal is to predict probabilities of belonging to one of two classes.
The sigmoid function has some limitations. First, the gradients of the sigmoid function become very small for large positive and negative inputs, leading to the vanishing gradient problem. This can make training deep neural networks with sigmoid activations slow and difficult. Second, it saturates when the input is very large or very small, leading to gradients close to zero. This can cause the model to learn slowly and exhibit poor convergence properties.

Rectifier Also known as rectified linear unit (ReLU), a rectifier function is a type of activation function commonly used in deep neural networks. It introduces non-linearity into the network by outputting the input directly if it is positive, and zero otherwise.
The rectifier function has become one of the most widely used activation functions in deep learning due to its simplicity, efficiency, and effectiveness in practice. It is often used in hidden layers of neural networks, especially in convolutional neural networks (CNNs) for computer vision tasks, and in feedforward neural networks for various other applications.
It has several advantageous properties. First, it is non-linear, which allows neural networks to learn complex mappings between inputs and outputs. Non-linear activation functions are essential for modeling non-linear relationships in data. Second, it encourages sparsity in activations, meaning that only a subset of neurons in a layer will be active (i.e., have non-zero outputs) at any given time. This property can lead to more efficient computation and better generalization. Third, the rectifier is also computationally efficient to evaluate and differentiate.

Softmax Function that converts a vector of arbitrary real values into a probability distribution. It is commonly used as the activation function in the output layer of a neural network for multiclass classification tasks, where the goal is to predict the probability that an input belongs to each class.
The softmax function essentially exponentiates each score and normalizes them so that they sum up to 1, thus producing a valid probability distribution. This enables the model to output probabilities for each class, allowing for easier interpretation and comparison of predictions.
Softmax is commonly used as the final activation function in neural networks for tasks such as image classification, natural language processing, and other classification problems with multiple classes.

It is often paired with the categorical cross-entropy loss function, which measures the difference between the predicted probabilities and the true labels, to train the model.

Loss Function

Also known as the cost function or objective function, it quantifies the difference between the predicted values of a model and the true values of the target variable. The loss function measures how well the model is performing on a given dataset by evaluating the discrepancy or error between its predictions and the actual outcomes.

A loss function must be differentiable (has a derivative for each point in its domain) and convex (does not have local minima). Different types of machine learning problems, such as classification, regression, and clustering, may require different loss functions.

Logarithmic Loss Also known as log-loss or cross-entropy loss, this loss function is used in classification tasks to measure the accuracy of a probabilistic classifier's predictions. It quantifies the difference between predicted probabilities and the actual binary or multiclass labels.

Logarithmic loss penalizes confident and incorrect predictions more heavily than confident and correct predictions. It is a strictly proper scoring rule, meaning that minimizing log-loss during training directly optimizes the model's ability to predict probabilities. Therefore, it is commonly used as the loss function when training probabilistic classifiers, particularly in binary and multiclass classification tasks.

Binary Cross-Entropy Loss Specific form of the logarithmic loss function used in binary classification tasks. It measures the difference between predicted probabilities and the actual binary labels (0 or 1) for each instance.

Binary cross-entropy loss is particularly used when training probabilistic classifiers such as logistic regression or neural networks with a sigmoid activation function in the output layer. It is a strictly proper scoring rule, meaning that minimizing binary cross-entropy loss during training directly optimizes the model's ability to predict probabilities.

Categorical Cross-Entropy Loss Also known as multiclass cross-entropy loss, this is a specific form of the logarithmic loss function used in multiclass classification tasks. It measures the difference between predicted probabilities and the actual class labels for each instance.

The categorical cross-entropy loss computes the loss for each instance and class pair and then averages them over all instances. It penalizes confident and incorrect predictions more heavily than confident and correct predictions.

Categorical cross-entropy loss is particularly used when training probabilistic classifiers with the softmax activation function in the output layer. It is a strictly proper scoring rule, meaning that minimizing categorical cross-entropy loss during training directly optimizes the model's ability to predict probabilities for each class.

Mean Absolute Error Also known as L1 loss, mean absolute error (MAE) is a commonly used loss function in regression tasks. It measures the average absolute difference between the predicted values and the actual values.

MAE loss is less sensitive to outliers because it penalizes each error linearly. This means that large errors have the same impact on the loss regardless of their sign.

Mean Squared Error Also known as L2 loss, mean squared error (MSE) is a commonly used loss function in regression tasks. It measures the average squared difference between the predicted values and the actual values.

MSE loss penalizes larger errors more heavily than smaller errors due to the squaring operation. As a result, MSE loss is more sensitive to outliers compared to other loss functions like Mean Absolute Error (MAE) loss.

Optimization

Optimization refers to the process of adjusting the parameters (weights and biases) of the network to minimize a defined loss function. The goal of optimization is to find the set of parameters that results in the best performance of the neural network on a given task, such as classification or regression.

Optimization in neural networks typically involves iterative algorithms that update the parameters based on the gradients of the loss function with respect to those parameters. The gradients indicate the

direction and magnitude of the steepest ascent of the loss function, and optimization algorithms adjust the parameters in the opposite direction of the gradients to descend towards the minimum of the loss function.

Gradient Descent Optimization algorithm used to find a local minimum of a loss function that takes repeated steps in the opposite direction of the (approximate) gradient of the function at the current point, because this is the direction of steepest descent.

Stochastic Gradient Descent Optimizer that picks a random sample from the dataset and runs a training epoch and it updates each training example's parameters one at a time. While frequent updates can offer more detail and speed, it can result in noisy gradients, but this can sometimes be helpful to escape local minima.

Batch Gradient Descent Optimizer that sums error for each point in a training set, updating the model only after a training epoch; after all training examples have been evaluated. Batch gradient descent also usually produces a stable error gradient and convergence, but sometimes that convergence point isn't the most ideal, finding the local minimum versus the global one.

Mini-Batch Gradient Descent Optimizer that combines concepts from both stochastic gradient descent and batch gradient descent. It splits the training dataset into small batch sizes and performs updates on each of those batches.

Stochastic Gradient Descent with Momentum Optimizer that extends the basic SGD algorithm by incorporating a momentum term, which helps accelerate convergence and reduce oscillations during training. In it, the update rule is modified to include a momentum term accumulates a fraction of the previous gradients to determine the update direction.

The momentum term determines how much of the previous gradients to retain when computing the current update direction. A higher momentum value means that the algorithm will rely more on the accumulated gradient history, leading to smoother and more stable updates that help the optimization process escape from local minima and plateaus more effectively.

SGD with momentum helps overcome some of the limitations of standard SGD, such as slow convergence and oscillations in the parameter space. It is particularly useful for training deep neural networks with high-dimensional parameter spaces, where the optimization landscape may be complex and non-convex.

AdaGrad Also known as Adaptive Gradient, this optimizer adapts the learning rate of each parameter during training based on the historical gradients observed for that parameter.

In other words, the learning rate of each parameter for each step will be based on the magnitude of the gradients that have been observed for that parameter in the past. Parameters that have received large gradients in the past will have their learning rates decreased, while parameters that have received small gradients will have their learning rates increased. This allows AdaGrad to automatically adapt the learning rates of different parameters based on their individual behavior during training.

AdaGrad effectively reduces the learning rate for frequently occurring parameters and increases the learning rate for infrequently occurring parameters, which can help improve convergence and stability during training.

However, one drawback of AdaGrad is that the learning rates can become too small over time, leading to premature convergence or slow progress in later stages of training. This issue has led to the development of variations of AdaGrad, such as RMSprop and Adam, which address some of its limitations while retaining its adaptive learning rate capabilities.

AdaDelta Adaptive learning rate optimization algorithm that is designed to address some of the limitations of AdaGrad, particularly the diminishing learning rates problem.

Like AdaGrad, AdaDelta (Adaptive Delta) adapts the learning rate for each parameter based on the historical gradients observed for that parameter. However, AdaDelta addresses the issue of diminishing learning rates by dynamically adapting the learning rate based on a moving average of the past gradients, rather than accumulating all past squared gradients.

The key idea behind AdaDelta is to compute an exponentially decaying average of past squared gradients, referred to as the root mean square (RMS) of the gradients, and use this average to scale the learning rate for each parameter.

AdaDelta effectively adapts the learning rate for each parameter based on the magnitude of the gradients observed for that parameter, allowing it to converge more quickly and robustly compared to fixed learning rate methods like AdaGrad. It also eliminates the need for manually tuning the learning rate or choosing a global learning rate schedule, making it easier to use in practice.

RMSProp Root Mean Square Propagation is an optimization algorithm used for training neural net-

works. It addresses the limitation of AdaGrad's diminishing learning rates by using a moving average of the squared gradients to scale the learning rates of the model parameters.
In RMSprop, the learning rate is adaptively adjusted for each parameter based on the magnitude of the gradients observed for that parameter. Parameters that receive large gradients will have their learning rates decreased, and parameters that receive small gradients will have their learning rates increased.
While similar to AdaDelta, the key differences is in the way they compute the moving average of past squared gradients and in the update rule for adjusting the model parameters.

Adam Short for Adaptive Moment Estimation, this popular optimizer combines ideas from two other optimization algorithms, RMSprop and momentum, to achieve efficient and robust optimization.
Adam maintains two moving averages: the first moment (the mean) of the gradients and the second moment (the variance) of the gradients. These moving averages are used to adaptively adjust the learning rates for each parameter during training.
Adam adapts the learning rates for each parameter based on the magnitude of the gradients and the past gradients' history. It effectively combines the benefits of momentum (which helps accelerate convergence) and RMSprop (which adapts the learning rates based on the magnitude of the gradients) to optimize deep neural networks efficiently.

Regularization

Regularization refers to techniques used to prevent overfitting, which occurs when a model learns to fit the training data too closely, capturing noise and irrelevant patterns that do not generalize well to new, unseen data. Regularization methods impose constraints on the model's parameters during training, discouraging overly complex models that may memorize the training data rather than learning underlying patterns.

Some regularization techniques are: L1 and L2 regularization, dropout, and early stopping.

L1 Regularization Regularization technique also called a lasso in which a penalty term proportional to the absolute values of the model's parameters is added to the loss function. This encourages sparsity in the parameter values, effectively selecting only the most important features and reducing the overall complexity of the model.
L1 regularization is rarely used on its own.

L2 Regularization Regularization technique also known as a ridge in which a penalty term proportional to the squared values of the model's parameters is added to the loss function. This encourages smaller parameter values, effectively preventing any single parameter from becoming too large and dominating the model's behavior. L2 regularization is particularly effective at preventing weights from growing too large and helps to improve the generalization performance of the model.

Elastic Net Regularization Technique that combines L1 and L2 regularization by adding a penalty term that is a linear combination of both L1 and L2 penalties. This allows for the benefits of both L1 and L2 regularization, effectively promoting sparsity while also preventing overly large parameter values.

Dropout Technique where randomly selected neurons are ignored or "dropped out" during training with a certain probability. This forces the network to learn more robust features and prevents individual neurons from becoming too specialized or co-adapting to each other.

Early Stopping Involves monitoring the model's performance on a separate validation dataset during training and stopping the training process when the validation performance starts to degrade. This prevents the model from overfitting by halting training before it starts to memorize the training data too closely.

Batch normalization Batch normalization is a technique to improve training stability and speed by normalizing layer inputs. It reduces internal covariate shift, making the network less sensitive to weight initialization and allowing higher learning rates. It also acts as a regularizer, potentially reducing the need for dropout. However, it adds computational overhead and complexity, and its effectiveness may vary depending on the task and dataset.

Dataset

Collections of data points used for training, validation, and testing the models. These datasets are typically organized into features (input variables) and labels (target variables) and are used to train the

neural network to learn patterns and relationships between the features and labels.

It is important to use high-quality datasets that are representative of the problem domain and provide sufficient diversity and coverage to train robust and generalizable models.

Features Input variables or attributes that are used to represent the data being processed by the network. Features are essentially the characteristics or properties of the data that are relevant to the task at hand. They provide the raw information that the neural network processes to make predictions or perform a specific task.

Features can take many forms, depending on the nature of the data and the problem being solved. In image classification tasks features may represent pixel values or higher-level visual patterns extracted from images. In natural language processing tasks, features may represent words, phrases, or other linguistic elements extracted from text data. In tabular data analysis, features may represent numerical or categorical variables describing different aspects of the data.

Training Data Dataset used to train a machine learning model. It consists of a collection of input-output pairs, where each input is associated with a corresponding target output. The purpose of training data is to teach the model to learn patterns or relationships between the input features and the target outputs, so that it can make accurate predictions or classifications on new, unseen data.

In supervised learning, the training data consists of labeled examples, where each input is paired with the correct output. The model learns to map inputs to outputs by observing these examples and adjusting its parameters to minimize the difference between its predictions and the true labels in the training data.

Testing Data Also known as validation or evaluation data, it is a separate dataset used to assess the performance of a trained machine learning model. Unlike training data which is used to train the model's parameters, testing data is used to evaluate how well the trained model generalizes to new, unseen data.

The primary purpose of testing data is to estimate the model's performance on real-world data that it has not been exposed to during training. By evaluating the model on a separate dataset, practitioners can assess its generalization ability and identify any issues such as overfitting or underfitting.

Testing data should be representative of the same distribution as the training data to ensure fair evaluation.

Batch Subset of the training data that is used together to compute the gradients of the model parameters during the optimization process. Instead of updating the model's parameters based on the gradients computed from individual training samples, batches allow for more efficient computation by aggregating gradients over multiple samples.

The size of the batch, known as the batch size, is a hyperparameter that can be adjusted based on factors such as the computational resources available, the size of the dataset, and the characteristics of the optimization problem. Common batch sizes range from small values (e.g., 32 or 64) to larger values (e.g., 128, 256, or even larger) depending on the specifics of the training process.

Augmentation Technique used to artificially increase the size of a training dataset by applying various transformations to the existing data samples. These transformations include operations such as rotation, translation, scaling, flipping, cropping, and changing brightness or contrast. By applying such transformations, data augmentation generates new training samples that are variations of the original data, thus providing more diverse examples for the neural network to learn from.

Data augmentation is particularly useful in scenarios where the size of the training dataset is limited or when the dataset lacks diversity. It helps to improve the generalization performance of the neural network by exposing it to a wider range of variations and reducing overfitting to the training data. Additionally, data augmentation can help to make the model more robust to variations in input data that it may encounter during deployment.

Preprocessing The steps taken to prepare a raw dataset before feeding it into the neural network for training. Preprocessing is an essential part of the machine learning pipeline and can have a significant impact on the performance of the model.

Some common steps include normalization, balancing, scaling, flattening, and shuffling.

Normalization typically scales the features to have a mean of 0 and a standard deviation of 1, while standardization scales the features to have a mean of 0 and a variance of 1.

Balancing addresses the issue of class imbalance in classification, where where certain classes may occur much less frequently than others in the dataset.

Scaling constraints input features to a specific range, such as [0, 1] or [-1, 1]. Feature scaling can

help improve the convergence of optimization algorithms and prevent numerical instabilities.
Flattening is the conversion of a multidimensional tensor into a one-dimensional vector. For image data flattening is used to convert the two-dimensional pixel grid of an image into a one-dimensional vector that can be fed into a neural network's input layer.
Shuffling involves randomly reordering the data samples in a dataset, typically to improve the performance and generalization of machine learning models. It helps to reduce bias, avoid overfitting, and improving optimization. It is typically performed randomly before each epoch of training so that the model sees a different order of data samples in each epoch, helping to prevent it from memorizing the training data order.

Chapter 3

Formulas

3.1 Definitions

- X: vector of input features, where x_i denotes the i-th input feature
- y: true or target output associated with the input data
- $\hat{y}$: predicted output produced by the neural network
- W: weight matrix with dimension $n \times m$, where m is the number of neurons in the dense layer
- b: bias vector with dimension m
- z: weighted sum of inputs plus the bias term (the logits or pre-activations)
- $h = \sigma(z)$: layer output vector after applying activation function σ element-wise over z

3.2 Forward propagation

$$\hat{y} = \sigma_L((X \cdot W_1 + b_1) \cdot W_2 + b_2 \cdot \ldots \cdot W_L + b_L)$$

Where:

- L: output layer
- σ_L: activation function applied at the output layer L
- X: vector of input features
- W_l: weight matrix of layer l
- b_l: bias vector of layer l

3.2.1 Layers

Dense

The formula for the operation performed by a single dense layer is a neural network is:

$$z = W \cdot x + b$$

Where:

- z: weighted sum of inputs plus the bias term
- W: weight matrix of the dense layer with dimension $n \times m$, where m is the number of neurons in the dense layer
- x: input vector to the dense layer with dimension n
- b: bias vector of the dense layer with dimension m

Or in matrix notation:

$$z_j = \sum_{i=1}^{n} w_{ij} \cdot x_i + b_j$$

Where:

- z_j: j-th element of the output vector z
- n: number of neurons in the layer
- w_{ij}: the weight connecting the i-th input neuron to the j-th neuron in the dense layer
- x_i: the i-th element of input vector x
- b_j: the bias term for the j-th neuron in the dense layer

3.2.2 Activators

Linear

$$\sigma(x) = x$$

Where:

- $\sigma(x)$: linear activation function
- x: input vector

Sigmoid

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

Where:

- $\sigma(x)$: sigmoid activation function
- e: Euler's number, or the base of the natural logarithm
- x: input vector

Rectifier

$$\sigma(x) = \max(0, x)$$

Where:

- $\sigma(x)$: rectifier or Rectified Linear Unit activation function
- $\max(0, x)$: function that returns 0 or x depending on which one is larger
- x: input vector

Softmax

$$\sigma_i(z) = \frac{\mathrm{e}^{z_i}}{\sum_{j=1}^{N} \mathrm{e}^{z_{ij}}}$$

Where:

- $\sigma(z)_i$: probability assigned to the i-th class after applying the Softmax function
- z_i: logit corresponding to the i-th class
- e: Euler's number, or the base of the natural logarithm
- N: total number of classes

3.2.3 Loss

Mean Absolute Error

$$\mathcal{L} = \frac{1}{n} \sum_{i=1}^{n} |\hat{y}_i - y_i|$$

Where:

- $\hat{y}_i$: predicted value for the i-th sample
- y_i: true value for the i-th sample
- $|\cdot|$: absolute value

Mean Squared Error

$$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

Where:

- $\hat{y}_i$: predicted value for the i-th sample
- y_i: true value for the i-th sample

Binary Cross-Entropy

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}(y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i))$$

Where:

- N: number of classes
- y_i: truth label for the i-th sample
- $\hat{y}_i$: predicted value (0 or 1) for the i-th sample

Categorical Cross-Entropy

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N} y_i \log(\hat{y}_i)$$

Where:

- N: number of samples
- y_i: truth label for the i-th sample
- $\hat{y}_i$: predicted value (0 or 1) for the i-th sample

When y is expressed as one-hot vectors, the equation simplifies to:

$$\mathcal{L} = -\frac{1}{N}\log(\hat{y}_{i,k})$$

Where k is the index of the target label.

3.3 Backward propagation

The backpropagation process involves the following steps:

1. Compute the error signals or gradient of the loss function with respect to the activations for each layer.
2. Compute the parameters (weights and biases) for each layer using the error signal of that layer and the activations of the previous layer.
3. Update the parameters (weights and biases) using an optimizer (e.g. gradient descent).

3.3.1 Error

An error signal of a layer l is the gradient of the loss function with respect to the activations a and its formula thus depends on the chosen activation and loss functions.

For the final layer $l = L$ the error signal is given as:

$$\begin{aligned}\delta_L &= \frac{\partial \mathcal{L}}{\partial a_L} \odot \sigma'_L(z_L) \\ &= \mathcal{L}'_L \odot \sigma'_L(z_L)\end{aligned}$$

Where:

- $\odot$ denotes element-wise multiplication
- $\frac{\partial \mathcal{L}}{\partial a_L}$ or $\mathcal{L}'_L$: partial derivative of the loss function with respect to the activations of layer L
- a_L: activations of layer L
- σ'_L: derivative of activation function σ of layer L
- z_L: pre-activation matrix or logits of layer L ($z_L = W_L \cdot a_L + b_L$)

The error signal at hidden layers is computed based on the error signals of the subsequent layers and the weights connecting the current layer to the following ones. The formula involves backpropagating the error signal from the subsequent layer (δ_{l+1}) through the weights, combined with the derivative of the activation function in the current layer σ'_l:

$$\delta_l = (W_l^T \cdot \delta_{l+1}) \odot \sigma'_l(z_l)$$

Where:

- $\cdot$ denotes the dot product
- W_l: weight matrix of layer l
- σ'_l: derivative of activation function σ of layer l
- z_l: pre-activation matrix or logits of layer l ($z_l = W_l \cdot a_{l+1} + b_l$)

3.3.2 Layers

The gradients of the weights and biases of a layer l are calculated using the error signal of the current layer and the activations of the preceding layer $(l+1)$:

$$\nabla W_l = \frac{\partial \mathcal{L}}{\partial W_l} = a_{l+1} \cdot \delta_l$$
$$\nabla b_l = \frac{\partial \mathcal{L}}{\partial b_l} = \sum_{i=1}^{n} \delta_{l(ij)}$$

Where $\delta_{l(ij)}$ denotes the error signal (gradient) for the j-th neuron in layer l for the i-th sample in the batch. The summation sums up all the rows of the error signal matrix along the columns.

3.3.3 Activators

In the backward step for an activation function σ its derivative with respect to the logits z (pre-activation values of a layer) must be calculated and is represented as:

$$\frac{\partial \mathcal{L}}{\partial \sigma} = \sigma'(z)$$

Linear

$$\sigma'(z) = 1$$

The backward pass for a linear activation function simply involves passing the error signal backward unchanged through the layer.

Sigmoid

$$\sigma'(z) = \sigma(z) \cdot (1 - \sigma(z))$$

Rectified Linear Unit

$$\sigma'(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases}$$

Softmax

$$\sigma'(z) = \frac{\partial \hat{y}_i}{\partial z_j} = \begin{cases} \hat{y}_i - \hat{y}_i \hat{y}_j & \text{when } i = j \\ -\hat{y}_i \hat{y}_j & \text{when } i \neq j \end{cases}$$

Where:

- $\hat{y}_i$: i-th element of the Softmax function predictions $\hat{y}$
- z_j: j-th element of the Softmax function input vector z

3.3.4 Loss

Mean Absolute Error

$$\mathcal{L}'(\hat{y}_i) = \frac{\partial \mathcal{L}}{\partial \hat{y}_i} = \frac{1}{n} \text{sign}(y_i - \hat{y}_i)$$

Where

- $\hat{y}_i$: i-th element of the Softmax function predictions $\hat{y}$
- y_i: i-th element of the target label vector y
- $\text{sign}(x)$: -1 if $x < 0, 0$ if $x = 0,$ and 1 if $x > 0$

Mean Squared Error

$$\mathcal{L}'(\hat{y}_i) = \frac{\partial \mathcal{L}}{\partial \hat{y}_i} = -\frac{1}{n} 2(\hat{y}_i - y_i)$$

Where:

- $\hat{y}_i$: i-th element of the Softmax function predictions $\hat{y}$
- y_i: i-th element of the target label vector y

Binary Cross-Entropy

$$\mathcal{L}'(\hat{y}_i) = \frac{\partial \mathcal{L}}{\partial \hat{y}_i} = -\frac{1}{n} \left[\frac{y_i}{\hat{y}_i} - \frac{1 - y_i}{1 - \hat{y}_i} \right]$$

Where:

- $\hat{y}_i$: i-th element of the Softmax function predictions $\hat{y}$
- y_i: i-th element of the target label vector y

Categorical Cross-Entropy

$$\mathcal{L}'(\hat{y}_i) = \frac{\partial \mathcal{L}}{\partial \hat{y}_i} = -\frac{1}{n} \frac{y_i}{\hat{y}_i}$$

Where:

- $\hat{y}_i$: i-th element of the Softmax function predictions $\hat{y}$
- y_i: i-th element of the target label vector y

Softmax Categorical Cross-Entropy

$$\mathcal{L}'(z_i) = \frac{\partial \mathcal{L}}{\partial z_i} = \hat{y}_i - y_i$$

Where:

- z_i: i-th element of the Softmax function input vector z
- $\hat{y}_i$: i-th element of the Softmax function predictions $\hat{y}$
- y_i: i-th element of the target label vector y

3.4 Optimizers

3.4.1 Stochastic gradient descent

$$\theta_{t+1} = \theta_t - \eta \nabla \mathcal{L}(\theta_t)$$

Where:

- θ_t: value of the parameter at time step t
- η: learning rate
- $\nabla \mathcal{L}(\theta_t)$: gradient of the loss function $\mathcal{L}$ with respect to parameter θ at time step t

3.4.2 Stochastic gradient descent with momentum

$$\begin{aligned} v_{t+1} &= \gamma \cdot v_t + (1 - \gamma) \nabla \mathcal{L}(\theta_t) \\ \theta_{t+1} &= \theta_t - \eta \cdot v_{t+1} \end{aligned}$$

Where:

- θ_t: value of the parameter at time step t
- η: learning rate
- $\nabla \mathcal{L}(\theta_t)$: gradient of the loss function $\mathcal{L}$ with respect to parameter θ at time step t
- v_t: moving average of the gradients at time step t
- γ: momentum hyperparameter (value between 0 and 1)

3.4.3 AdaGrad: adaptive gradient

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

Where:

- θ_t: value of the parameter at time step t
- η: learning rate
- $\nabla \mathcal{L}(\theta_t)$: gradient of the loss function $\mathcal{L}$ with respect to parameter θ at time step t
- v_t: sum of squares of past gradients up to time step t
- ϵ: small constant (usually on the order of 10^8) to avoid division by zero

3.4.4 AdaDelta: adaptive learning rate

$$\begin{aligned} v_{t+1} &= \rho v_t + (1 - \rho) \nabla \mathcal{L}(\theta_t)^2 \\ \Delta\theta_{t+1} &= -\frac{\sqrt{\Delta\theta_t} + \epsilon}{\sqrt{v_{t+1}} + \epsilon} \nabla \mathcal{L}(\theta_t) \\ \theta_{t+1} &= \theta_t + \Delta\theta_{t+1} \end{aligned}$$

Where:

- θ_t: value of the parameter at time step t
- $\nabla \mathcal{L}(\theta_t)$: gradient of the loss function $\mathcal{L}$ with respect to parameter θ at time step t
- v_t: exponentially decaying moving average of squared gradients at time step t
- ρ: decay rate for the moving average, typically close to 1 (e.g. 0.95)
- ϵ: small constant to avoid division by zero
- $\Delta\theta_{t+1}$: update term for the parameter at time step $t+1$, computed based on the root mean square of the parameter updates
- $\Delta\theta_t$: exponentially decaying moving average of squared parameter updates at time step t

3.4.5 RMSProp: root mean square propagation

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

Where:

- θ_t: value of the parameter at time step t
- η: learning rate
- $\nabla\mathcal{L}(\theta_t)$: gradient of the loss function $\mathcal{L}$ with respect to parameter θ at time step t
- v_t: exponentially decaying moving average of squared gradients at time step t
- ρ: decay rate, typically set to a value close to 1 (e.g. 0.9)
- ϵ: small constant to avoid division by zero

3.4.6 Adam: adaptive moment estimation

$$m_{t+1} = \beta_1 m_t + (1-\beta_1)\nabla\mathcal{L}(\theta_t)$$
$$v_{t+1} = \beta_2 v_t + (1-\beta_2)\nabla\mathcal{L}(\theta_t)^2$$
$$\hat{m}_{t+1} = \frac{m_{t+1}}{1-\beta_1^t}$$
$$\hat{v}_{t+1} = \frac{v_{t+1}}{1-\beta_2^t}$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_{t+1}} + \epsilon}\hat{m}_{t+1}$$

Where:

- θ_t: value of the parameter at time step t
- η: learning rate
- $\nabla\mathcal{L}(\theta_t)$: gradient of the loss function $\mathcal{L}$ with respect to parameter θ at time step t
- m_t and v_t are the first and second moments (moving averages) of the gradients at time step t
- $\hat{m}_{t+1}$ and $\hat{v}_{t+1}$ are bias-corrected estimates of the first and second moments
- β_1 and β_2 are the decay rates for the moving averages, typically close to 1 (e.g. 0.9 and 0.999, respectively)
- ϵ: small constant to avoid division by zero

3.5 Regularization

3.5.1 L1 regularization

L1 regularization adds the sum of the absolute values of the parameters to the loss function. The L1 regularization terms for a parameter θ (weights or bias) is:

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

Where:

- L_{1_θ}: L1 regularization penalty for parameter θ
- λ: lambda term that controls the impact of the regularization penalty
- θ_m: m-th element in the vector or matrix of parameters θ

The gradient of the L1 regularization term with respect to the parameters is simply the sign of the parameters $\text{sign}(\theta)$. Therefore, the total gradient of the loss function with respect to the parameters of the dense layer are:

$$L'_{1_\theta} = \frac{\partial\mathcal{L}}{\partial\theta_m} = \nabla\theta_m + \lambda\,\text{sign}(\theta_m)$$

Where:

- $\mathcal{L}$: loss function
- $\nabla\theta_m$: gradient of the loss function with respect to the m-th parameters without regularization
- λ: lambda term that controls the impact of the regularization penalty
- θ_m: m-th element in the vector or matrix of parameters θ

3.5.2 L2 regularization

L2 regularization adds the sum of the squared values of the parameters to the loss function. The L2 regularization term is:

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

Where:

- L_{2_θ}: L2 regularization penalty for parameter θ
- λ: lambda term that controls the impact of the regularization penalty
- θ_m: m-th element in the vector or matrix of parameters θ

The gradient of the L2 regularization term with respect to the parameters is simply the sign of the parameters is:

$$L'_{2_\theta} = \frac{\partial \mathcal{L}}{\partial \theta_m} = \nabla\theta_m + 2\lambda\theta_m$$

Where:

- $\mathcal{L}$: loss function
- $\nabla\theta_m$: gradient of the loss function with respect to the m-th parameters without regularization
- λ: lambda term that controls the impact of the regularization penalty
- θ_m: m-th element in the vector or matrix of parameters θ

3.5.3 Dropout

In a dropout layer, during a forward pass, neurons are randomly dropped out with certain probability p during tyraining. This helps prevent overfitting by forcing the network to learn redundant representations.

A binomial experiment consists of a fixed number n of statistically independent Bernoulli trials, each with a probability of success p, and counts the number of successes. A random variable corresponding to a binomial experiment denoted by $B(n, p)$ has a binomial distribution. The probability of exactly k successes in experiment $B(n, p)$ is given by:

$$P(k) = B(n, p) = \binom{n}{k} p^k (1-p)^{n-k}$$

Where:

- $\binom{n}{k}$: binomial coefficient
- n: number of trials
- k: number of successes
- p: probability of success
- $(1-p)$: probability of failure

The Bernoulli distribution is a special case of the binomial distribution with $n = 1$ and $k \in 0, 1$:

$$B(1, p) = p^k (1-p)^{1-k}$$

Or equivalently:

$$B(1, p) = \begin{cases} p & \text{if } x = 1 \\ 1-p & \text{if } x = 0 \end{cases}$$

Draw a random binary mask from the Bernoulli distribution described above with probability p to generate a sequence of binary values (0s and 1s). Then the formula for the operation of a dropout layer is given as:

$$z = x \cdot B(1, p) \frac{1}{1-p}$$

Where:

- z: output after dropout
- x: input vector
- $B(1, p)$: Bernoulli distribution with probability p
- p: probability of success

When dropout is applied during training, a fraction p of neuron outputs are randomly set to zero. Thus, the remaining fraction $1 - p$ of neuron outputs are scaled up by a factor of $\frac{1}{1-p}$ to compensate for the dropped neurons. During inference, dropout is not applied.

During the backward pass, the activations of the remaining neurons need to be scaled to ensure that the expected output of the layer remains the same during training and inference. The formula for the backwards pass of the dropout layer is given as:

$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial \hat{x}} \odot \frac{B(1, p)}{1 - p}$$

Where:

- $\odot$ represents element-wise multiplication
- $\mathcal{L}$: loss with respect to the output of the dropout layer
- x: input to the dropout layer
- $\hat{x}$: output of the dropout layer
- $\frac{\partial \mathcal{L}}{\partial x}$: gradient of the loss with respect to the input to the dropout layer
- $\frac{\partial \mathcal{L}}{\partial \hat{x}}$: gradient of the loss with respect to the output of the dropout layer
- $\frac{B(1,p)}{1-p}$: binary mask generated during the forward pass, with elements set to 1 for the neurons that are retained and 0 for the neurons that were dropped