HyperionDev

Learning Algorithms

Task

Visit our website

Introduction

To train a neural network, we use gradient descent as a learning algorithm to optimise the weights and biases. Backpropagation efficiently calculates the gradients in gradient descent. Regularisation techniques like L1 and L2 regularisation prevent overfitting. While libraries automate many of these processes, understanding their details is crucial for customising training processes, debugging issues, and exploring innovative techniques.

Gradient descent

Gradient descent is a popular optimisation algorithm used to train neural networks. Recall it works by iteratively adjusting the model's parameters in the direction that minimizes a loss function. This is achieved by calculating the gradient of the loss function with respect to the parameters and taking a step in the opposite direction. In essence, gradient descent helps the model learn the optimal values for its weights and biases to make accurate predictions. Here we will explore some variations of gradient descent.

Stochastic gradient descent

Stochastic gradient is a more efficient variant of traditional gradient descent, particularly suited for training large models like neural networks with millions of parameters. For simple models with only a couple of parameters, like the Gabor model, finding the global minimum might be feasible through deep search or multiple gradient descent starts. However, for more complex models, these methods are impractical.

Gradient descent typically settles at a minimum that heavily depends on where it started, with no guarantee it is the global minimum or even a satisfactory one. Stochastic gradient descent addresses this limitation by introducing randomness into the gradient at each iteration. This means the path taken doesn't always strictly follow the steepest descent; it can move in less optimal directions or even slightly uphill. This variability gives Stochastic gradient descent the ability to escape local minima and explore different areas of the loss function, potentially finding better minima across the landscape (as illustrated in figure 17).

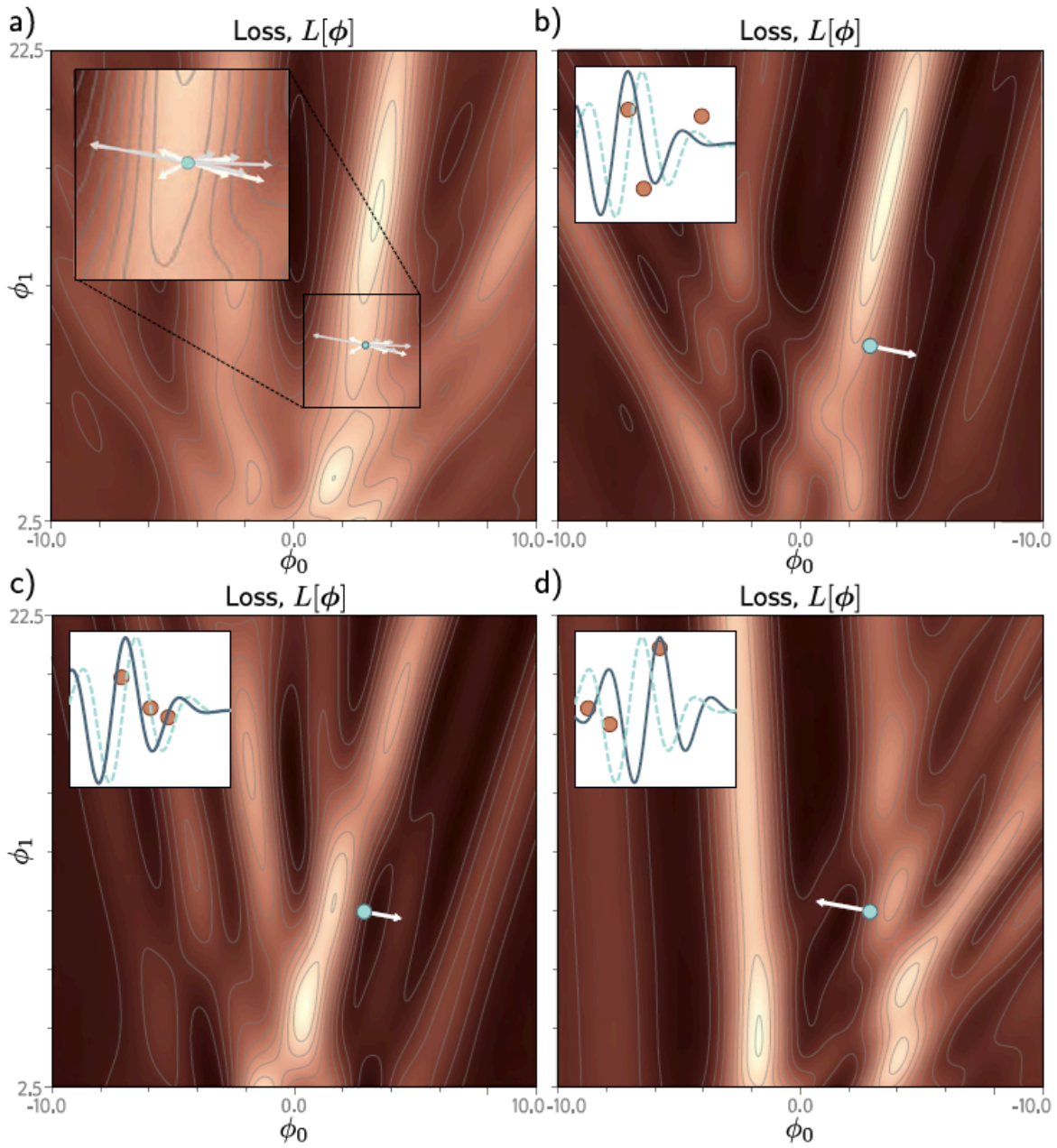


Figure 17: *The perspective on stochastic gradient descent (for the Gabor model) using a batch size of three. Panel (a) depicts the loss function for the entire training dataset, highlighting a probability distribution of potential parameter changes at each iteration, as illustrated by the inset samples. These variations are from different combinations of the three batch elements. In panel (b), the loss function for one specific batch is shown, where the stochastic gradient descent algorithm moves downhill based on this function over a distance determined by the learning rate and the local gradient's magnitude. This adjustment results in the model (dashed line in the inset) adapting to more closely fit the batch data (solid line). Panel (c) demonstrates that a different batch leads to an alternate loss function, prompting a unique update. In panel (d), the algorithm descends in terms of the batch-specific loss function but ascends relative to the global loss function shown in panel (a). This mechanism allows stochastic gradient descent to potentially escape local minima. (Prince, 2024)*

Mini-batch gradient descent

In the context of stochastic gradient descent, randomness is incorporated by selecting a random subset of the training data, known as a mini-batch, at each iteration to compute the gradient. The update formula for the model parameters ϕ_t at iteration t is given by:

$$\phi_{t+1} = \phi_t - \alpha \sum_{i \in B_t} \frac{\partial l_i[\phi_t]}{\partial \phi}$$

where B_t represents the indices of the input/output pairs in the current batch, l_i is the loss for the i-th pair, and α is the learning rate that, together with the gradient magnitude, decides the step size for each update. The learning rate is predetermined and remains constant throughout the process, independent of the function's local properties.

Batches are typically sampled from the dataset without replacement, and the algorithm processes all the data before resampling from the entire dataset. A complete cycle through the dataset is called an **epoch**. Batch sizes can vary from one example to the entire dataset – the latter being equivalent to full-batch gradient descent, which mimics traditional gradient descent.

An alternative view of stochastic gradient descent suggests that at each iteration, it calculates the gradient of a varying loss function dependent on the specific batch and the model. This perspective sees stochastic gradient descent as performing deterministic gradient descent on a continually shifting loss function. Despite this variability, the average loss and gradient calculations across iterations align with those of traditional gradient descent.

Stochastic gradient descent possesses several beneficial properties. Firstly, even though it introduces noise into the update trajectory, it consistently improves the fit for a subset of data with each iteration, making the updates effective albeit not always optimal. Secondly, it processes each training example equally by drawing them without replacement and cycling through the dataset. Thirdly, it is more efficient computationally to calculate gradients from just a subset of data rather than the full dataset. Fourthly, stochastic gradient descent has the potential to escape local minima and lastly, it decreases the likelihood of getting stuck near saddle points since different batches likely exhibit significant gradients at any given point on the loss function. Additionally, there is evidence suggesting that stochastic gradient descent helps neural networks generalise better to new data. Stochastic gradient descent does not "converge" in the traditional sense but aims for a state where all data points are well represented by the model, resulting in small gradients regardless of the batch used, hence leading to minimal changes in the parameters.

In practice, stochastic gradient descent is often managed with a learning rate schedule, starting with a higher learning rate (α) which is reduced by a constant factor every set number of epochs (N). An epoch represents a complete pass through the entire training dataset, where every training example has been used once to update the model parameters. One epoch indicates that the learning algorithm has processed all the training data one time; subsequent epochs involve repeatedly processing the dataset to progressively improve the model. This strategy allows for exploration of the parameter space in the early stages and allows for more precise adjustments later on, improving the fine-tuning of the model parameters.

Backpropagation

Backpropagation represents an algorithm used for training neural networks, particularly effective for optimising the weights and biases through the network layers. This method is based on the principle of gradient descent, where the aim is to minimise the loss or error function, which quantifies the difference between the predicted outputs of the network and the actual target values.

Did you know

The backpropagation algorithm, first introduced in the 1970s, only gained widespread recognition after 1986 with a paper by David Rumelhart, Geoffrey Hinton, and Ronald Williams. This paper highlighted how backpropagation greatly accelerated the learning process in neural networks compared to previous methods, enabling the resolution of problems that were once considered unsolvable. Today, backpropagation represents a fundamental method for training neural networks.

Math annotations

First, let's review a matrix-based method for calculating outputs from a neural network. Understanding this algorithm will make the complex notation of backpropagation more approachable.

To start, let's clarify the notation for weights in a network. The term ω_{jk}^{l} refers to the weight from the k^{th} neuron in the $(l-1)^{th}$ layer to the j^{th} neuron in the l^{th} layer. This notation might seem complex initially, but with practice, it becomes straightforward. For instance, the following diagram illustrates the weight for the connection from the fourth neuron in the second layer to the second neuron in the third layer of a network.

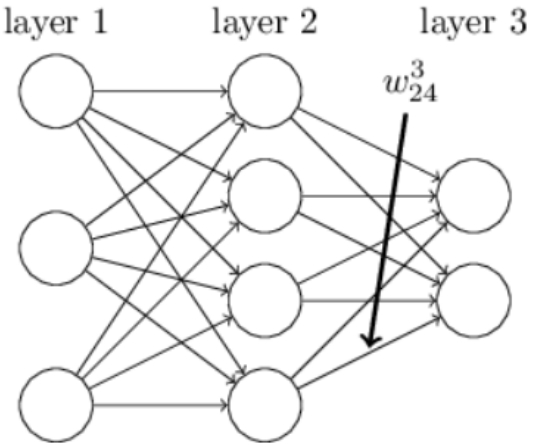


Figure 5: *Representing that ω_{jk}^{l} is the wight from the k^{th} neuron in the $(l-1)^{th}$ layer to the j^{th} neuron in the l^{th} layer (Nielsen, 2019)*

The notation's order, j and k might seem reversed, but this will be explained shortly. Similar notations apply to the network's biases and activations. For biases b_{j}^{l}, denotes the bias of the j^{th} neuron in the l^{th} layer, and a_{j}^{l} represents its activation. The activation of the j^{th} neuron in the l^{th} layer, and a_{j}^{l} is determined by the activations from the previous layer through the equation:

$$a_{j}^{l} = \sigma\left(\sum_{k} \omega_{jk}^{l} a_{k}^{l-1} + b_{j}^{l}\right)$$

In the following figure these notations are presented:

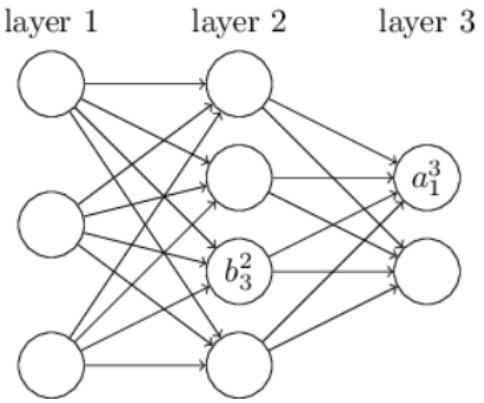


***Figure 6**: An example of the introduced notations (Nielsen, 2019)*

To express this in a matrix form, we define a weight matrix W^l for each layer, with each entry ω_{jk}^l representing the weight from neurons in layer $l-1$ to layer l. A bias vector b^l and an activation vector a^l are defined similarly for each layer.

Using matrix notation, this activation equation becomes more streamlined:

$$a^l = \sigma\left(\omega^l a^{l-1} + b^l\right)$$

Take note

The vectorised form allows us to think globally about how activations are related across layers: simply apply the weight matrix to the activations of the previous layer, add the bias vector, and apply the activation function. This perspective avoids the heavy detail of individual neuron connections and exploits matrix operations for efficient computation, which is vital for implementing neural networks effectively.

In practice, when computing a^l using the above equation, we also calculate:

$$z^l \equiv W^l a^{l-1} + b^l,$$

known as the weighted input to the neurons in layer l. This concept of weighted input is important in neural network calculations and will be used extensively in discussions of backpropagation. In terms of weighted inputs, the activation for each layer can be

rewritten as $a^l = \sigma(z^l)$. This approach simplifies many calculations and helps us understand the network's behaviour more holistically.

Step-by-step guide

To start with backpropagation, it is required to calculate errors on the network outputs that should be propagated backward through the network, from the output layer back to the input layer. Error calculation begins by comparing the network's predictions against the actual target values. The difference between these values indicates how much error the network produced for that particular instance. This process is repeated for each input in the training set to understand the network's performance across a variety of cases.

Then the backward movement starts involving several key steps:

- **Gradient of the cost function**: The derivative of the cost function with respect to the output of the network is calculated. This gradient indicates the direction and rate at which the weights need to adjust to minimise the loss.
- **Local gradients**: For each neuron, local gradients are computed. These are partial derivatives of the neuron's output with respect to its inputs, which are then used to determine the effect of each weight on the loss. Partial derivatives expand the concept of a derivative to multiple variable functions. Keeping the other variables fixed, they track how a function changes as one of its inputs varies. Partial derivatives enable us to calculate the gradient of the loss function with regard to every weight and bias in the network. In particular, when we calculate the gradient of the loss function, we're determining how much the loss would change if we made a small change to one weight or bias while keeping all others fixed. Changing the parameters of the network in a way that reduces the loss depends on this fact. Holding the other weights constant, the partial derivative of the neuron's output with respect to w_i tells us how the output of the neuron changes. This helps us to analyse how every individual weight affects the total error, therefore guiding the changes required to maximise the performance of the network. Implementing backpropagation in neural networks depends on the computation of these partial derivatives.
- **Error responsibility**: The error is distributed to all weights in the network proportional to their contribution to the total error. This distributed error is used to update the weights, aiming to reduce the error in future predictions.
- **Update the weights and biases**: The final step uses the calculated gradients to update the weights and biases. Typically, a learning rate is used to control how much the weights are changed during each iteration. This rate helps in making sure the network gradually converges to a minimum of the loss function, avoiding large jumps which can lead to unstable training dynamics.

Let's explore this process in more detail now.

Error calculation

To properly quantify output errors, loss functions are commonly used to transform them into a single, comprehensive value that reflects the performance of the entire network. Common loss functions include the mean squared error (MSE) for regression tasks, which calculates the average of the squares of the differences between predicted and actual values. For classification tasks, cross-entropy loss is often used; it measures the performance of a classification model whose output is a probability value between 0 and 1. The loss increases as the predicted probability diverges from the actual label, providing a strong gradient when the model is wrong.

The loss functions provide a metric that backpropagation and optimisation algorithms such as gradient descent, Stochastic Gradient Descent (SGD), and Mini-Batch Stochastic Gradient Descent (MB-SGD) use to adjust weights and biases, thereby improving prediction accuracy.

Cost function

To effectively use the backpropagation algorithm, we need to understand the underlying assumptions about the cost function, represented as C, which measures the performance of a neural network. It quantifies how well the neural network is performing across the entire dataset. In essence, the cost function is the aggregation of individual loss values for all training examples. For example, in the case of mean squared error, the cost function would be the average of the MSE values for all training examples.

This average cost function allows us to use backpropagation to calculate the derivatives $\frac{\partial C_x}{\partial \omega}$ and $\frac{\partial C_x}{\partial b}$ for individual examples and then average these to find $\frac{\partial C}{\partial \omega}$ and $\frac{\partial C}{\partial b}$.

In terms of suitable cost functions, the quadratic cost function is commonly exploited for a neural network with L layers:

$$C = \frac{1}{2n}\sum_x \|y(x) - a^L(x)\|^2,$$

Where:

n - total number of training samples,

x - each training sample,

y - desired output, and

a^L - output from the network for input x.

In order to use cost functions in an optimal way, two critical assumptions will now be considered.

Assumption 1

The first critical assumption for backpropagation is that the cost function can be expressed as the average of the costs computed for each training example:

$$C = \frac{1}{n}\sum_x C_x.$$

In the context of the quadratic cost, the cost for this example is $C_x = \frac{1}{2}\|y - a^L\|^2$. This formulation allows us to use backpropagation to calculate the derivatives $\frac{\partial C_x}{\partial \omega}$ and $\frac{\partial C_x}{\partial b}$ for individual examples and then average these to find $\frac{\partial C}{\partial \omega}$ and $\frac{\partial C}{\partial b}$.

Here, $\frac{\partial C_x}{\partial \omega}$ represents the derivative of the cost for a single example x with respect to a specific weight ω. It tells us how much the cost C_x changes when we make a small change to the weight ω. And $\frac{\partial C_x}{\partial b}$ represents the derivative of the cost for a single example x with respect to a specific bias b, which explains how much the cost C_x changes when we make a small change to the bias b.

By averaging these individual derivatives across all training examples, we obtain: $\frac{\partial C}{\partial \omega}$ - the average derivative of the cost function with respect to the weight ω across all training examples, showing the overall sensitivity of the cost function to changes in the weight. And $\frac{\partial C}{\partial b}$ representing the average derivative of the cost function with respect to the bias b across all training examples, representing the overall sensitivity of the cost function to changes in the bias.

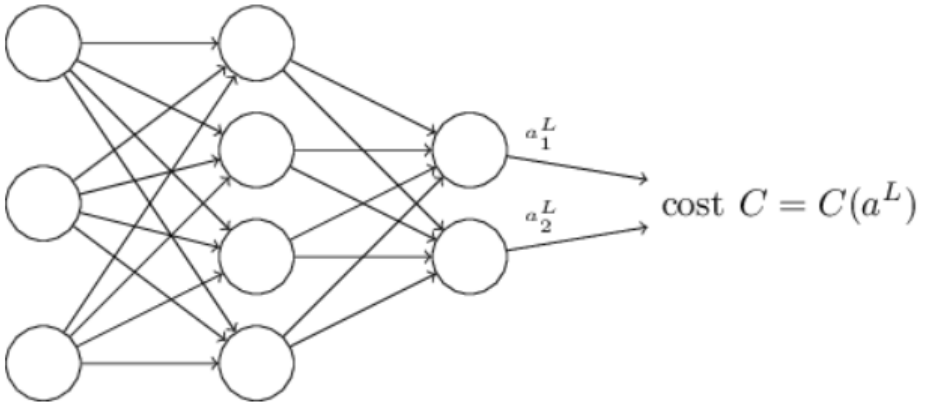


***Figure 7**: Representation of a cost function (Nielsen, 2019)*

Assumption 2

The second assumption is that the cost function must be a function of the network's outputs. For the quadratic cost function, this is satisfied as it can be rewritten for a single training example x as $= \frac{1}{2}\|y - a^L\|^2 = \frac{1}{2}\sum_j \left(y_j - a_j^L\right)^2$, showing that C depends on the output activations a^L.

The desired output y is fixed once the training example x is set, making y a constant parameter in the function, not a variable influenced by the network's weights and biases. Thus, the cost is considered solely a function of the output activations a^L, not of y , which is treated as a parameter that helps define the cost function but is not modified during learning.

Gradient of the cost function

The process begins by evaluating how the output of the network deviates from the desired outcome, a step often completed by computing the error in the output layer. The gradients of the cost function with respect to the output activations are first calculated, providing the basis for determining how much each output neuron contributed to the overall error.

Local gradients

Following the initial computation in the output layer, the backpropagation algorithm proceeds to compute the gradient for each layer in the network moving backwards from the output towards the input. This involves applying the chain rule of calculus repeatedly to propagate the error information back through the network.

For each neuron in each layer, the gradient computation combines the error term propagated from the layer above with the derivative of the activation function used in that neuron. This combination provides a measure of the neuron's impact on the overall error, considering both its activation function's sensitivity and the errors received from the subsequent layer. The gradient computation for each parameter – every weight and bias – is then achieved by linking these error terms with the activations from the previous layer (for weights) or directly associating them with the biases. This results in a set of gradient values for each weight and bias that informs how they should be adjusted to reduce the network's overall error. Finally, these computed gradients are used to update the weights and biases in the gradient descent step.

Error responsibility

Backpropagation presents a complex process within neural networks, and it raises two primary questions:

1. What exactly is the algorithm doing? While we understand that errors are propagated backward from the output, analysing the matrix and vector multiplications can provide greater insight of error responsibility within the network.

2. How was backpropagation discovered? It is one thing to follow an algorithm or its proof, but understanding it deeply enough to have developed it is another challenge altogether.

Let's improve our understanding by considering a small change, $\Delta\omega^{l}_{jk}$, in a weight within the network ω^{l}_{jk}:

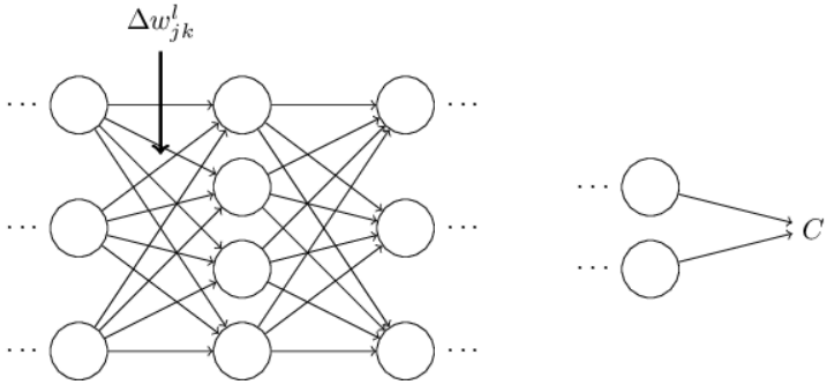


***Figure 8**: Updating the parameters – Step 1 (Nielsen, 2019)*

This minor adjustment affects the output activation of the corresponding neuron:

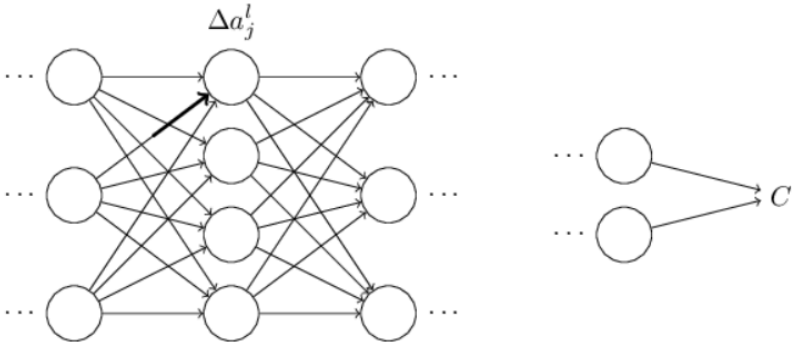


***Figure 9**: Updating the parameters – Step 2 (Nielsen, 2019)*

which then impacts the activations in subsequent layers:

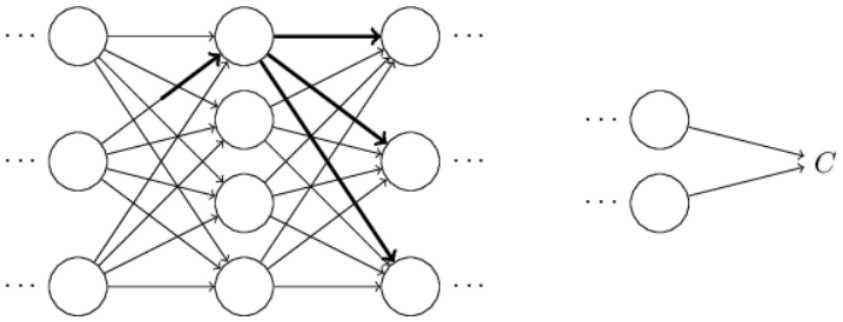


***Figure 9**: Updating the parameters – Step 3 (Nielsen, 2019)*

continuing through the network to the final layer and ultimately influencing the cost function:

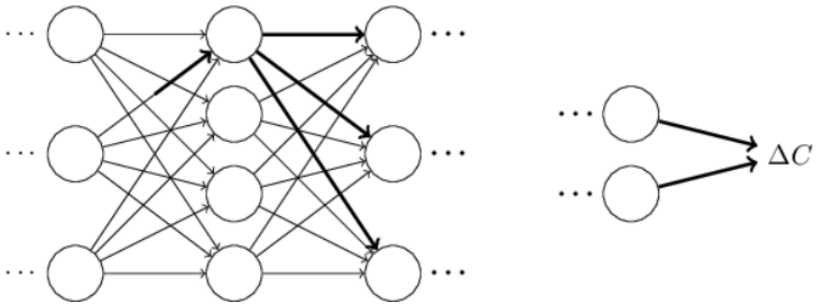


***Figure 10**: Updating the parameters – Step 4 (Nielsen, 2019)*

The relationship between the change in the cost and the change in the weight is expressed as:

$$\Delta C \approx \frac{\partial C}{\partial \omega^{l}_{jk}} \Delta \omega^{l}_{jk}.$$

To track this effect, consider how $\Delta \omega^{l}_{jk}$ causes a change Δa^{l}_{j} in the activation of neuron j^{th} in layer l^{th}, expressed as:

$$\Delta a^{l}_{j} \approx \frac{\partial a^{l}_{j}}{\partial \omega^{l}_{jk}} \Delta \omega^{l}_{jk}$$

This change in turn affects the activations in the following layer, and so forth, cascading through the network. We will now focus on how just a single one of those activations is affected as a_q^{l+1}:

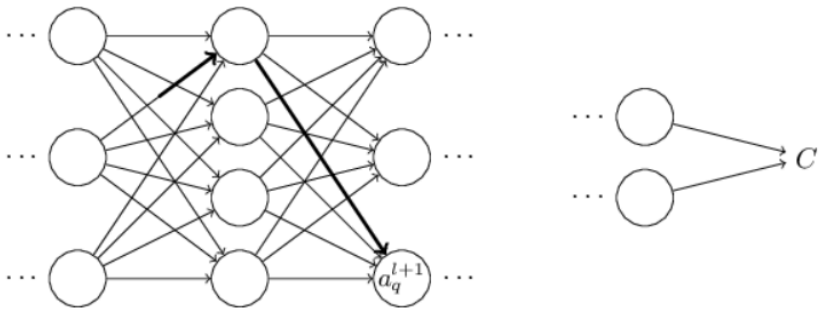


***Figure 11**: Updating the parameters – Step 5 (Nielsen, 2019)*

In fact, this change triggers a subsequent alteration in the next layer's activations. If we alter the weight ω_{jk}^l, it causes a small adjustment in the activation Δa_j^l, which can be approximated by:

$$\Delta a_q^{l+1} \approx \frac{\partial a_q^{l+1}}{\partial a_j^l} \Delta a_j^l.$$

Substituting the expression for Δa_j^l, we get:

$$\Delta a_q^{l+1} \approx \frac{\partial a_q^{l+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial a \omega_{jk}^l} \Delta \omega_{jk}^l$$

This change will cascade through subsequent layers, influencing each layer's activations, and culminate in a change in the network's overall cost C. If this sequence of activations proceeds through layers up to a_m^L, the overall change in cost can be expressed as:

$$\Delta C \approx \frac{\partial C}{\partial a_m^L} \frac{\partial a_m^L}{\partial a_n^{L-1}} \frac{\partial a_n^{L-1}}{\partial a_p^{L-2}} \ldots \frac{\partial a_q^{l+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial \omega_{jk}^l} \Delta \omega_{jk}^l$$

The last formula shows that a change in a single weight affects the cost through a chain of activations, each influencing the next. However, multiple such paths exist where changes can propagate from a weight to the cost, suggesting that the total change in C would be the sum of changes across all paths:

$$\Delta C \approx \sum_{m,n,p,\ldots,q} \frac{\partial C}{\partial a_m^L} \frac{\partial a_m^L}{\partial a_n^{L-1}} \frac{\partial a_n^{L-1}}{\partial a_p^{L-2}} \ldots \frac{\partial a_q^{l+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial \omega_{jk}^l} \Delta \omega_{jk}^l$$

Here, ΔC represents the change in the cost function; $\sum_{m,n,p,\ldots,q}$ summing over all possible paths from the weight ω_{jk}^{l} to the output neuron; $\frac{\partial C}{\partial a_m^L}$ the rate of change of the cost with respect to the activation of the output neuron a_m^L; $\frac{\partial a_m^L}{\partial a_n^{L-1}}$ the rate of change of the activation of the output neuron with respect to the activation of a neuron in the previous layer; $\frac{\partial a_n^{L-1}}{\partial a_p^{L-2}}$ the rate of change of the activation of the neuron in the previous layer with respect to the activation of a neuron in the layer before that, and so on; $\frac{\partial a_q^{l+1}}{\partial a_j^l}$ the rate of change of the activation in layer $l+1$ with respect to the activation in layer l; $\frac{\partial a_j^l}{\partial \omega_{jk}^l}$ the rate of change of the activation in layer l with respect to the weight ω_{jk}^l; ω_{jk}^l represents the small change in the weight.

This equation shows how a small change in a weight affects the cost by propagating through the network's layers.

From this, we derive:

$$\frac{\partial C}{\partial \omega_{jk}^l} \approx \sum_{m,n,p,\ldots,q} \frac{\partial C}{\partial a_m^L} \frac{\partial a_m^L}{\partial a_n^{L-1}} \frac{\partial a_n^{L-1}}{\partial a_p^{L-2}} \ldots \frac{\partial a_q^{l+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial \omega_{jk}^l}$$

Here, $\frac{\partial C}{\partial \omega_{jk}^l}$ represents the gradient of the cost function with respect to the weight ω_{jk}^l and the right side of the equation is a sum over all possible paths from the weight to the output, similar to the previous equation.

This equation summarises the process of calculating the gradient of the cost function with respect to a specific weight. It shows that the gradient is the sum of the products of partial derivatives along all possible paths from the weight to the output.

This equation indicates that the gradient of the cost with respect to a weight is the sum of all the products of rate factors, which are the partial derivatives along each path from that weight to the output cost. The procedure is presented in the following figure for a single path:

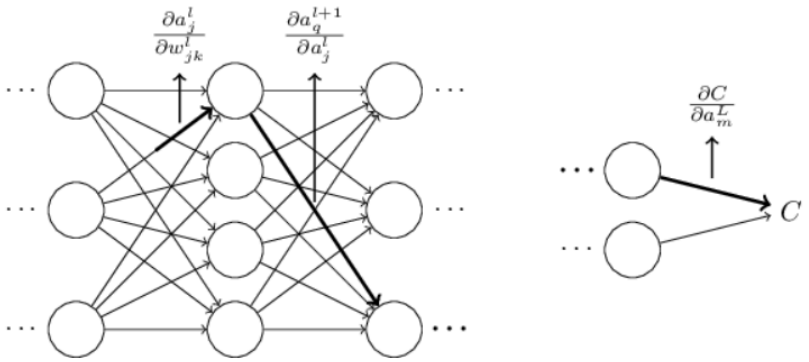


***Figure 12**: Updating the parameters – Step 6 (Nielsen, 2019)*

Update the weights and biases

Now it is finally time to update each weight and bias using gradient descent. During gradient descent, each parameter is adjusted in the opposite direction of its gradient. Mathematically, this is represented by subtracting a fraction of the gradient from the current value of the parameter. This fraction is known as the learning rate, a hyperparameter that controls how big a step is taken towards the minimum of the cost function during each iteration. If the learning rate is too large, the network might overshoot the minimum; if it's too small, the training process can become excessively slow and may get stuck before reaching the minimum.

The general formula for updating a weight is given by:

$$\omega_{new} = \omega_{old} - \eta \frac{\partial C}{\partial \omega}$$

where η is the learning rate and $\frac{\partial C}{\partial \omega}$ is the gradient of the cost function with respect to the weight ω.

Similarly, biases are updated according to:

$$b_{new} = b_{old} - \eta \frac{\partial C}{\partial b},$$

where $\frac{\partial C}{\partial b}$ is the gradient of the cost function with respect to the bias b.

This process is repeated for each parameter in the network across multiple iterations or epochs, gradually going down the cost function as the network learns from the data. The repeated adjustment of weights and biases allows the neural network to fine-tune its predictions, improving its accuracy and effectiveness at performing the desired task. Through this iterative optimisation, neural networks are able to learn complex patterns and relationships within the data, ultimately achieving a level of performance that is finely tuned to the specific requirements of the application. This understanding provides a heuristic explanation of how perturbations in weights affect the network's

behaviour and cost, aligning closely with the backpropagation algorithm. Perturbations in the context of neural network training represent minor weight and bias modifications made within the system. Fine-tuning the parameters of the network depends on these perturbations, which also help to minimise the cost function and increase performance.

Take note

The **main benefit** of exploiting backpropagation is the computation of all partial derivatives of the cost function with respect to every weight and bias through just one forward pass and one backward pass through the network. The backward pass performs computations similar to the forward pass, but it uses transposed versions of the weight matrices. The computational cost of the backward pass is comparable to that of the forward pass through the network. Thus, backpropagation completes in about the time it would take to perform two forward passes – a significant reduction from the million-plus passes required by the earlier method.

Despite its advantages, backpropagation is not without **limitations**, especially in training very deep networks with multiple hidden layers. However, advances in computing power and innovative techniques have now made it feasible to train these complex networks effectively.

Did you know

Backpropagation helps understand how a small change in a weight within a neural network impacts the overall cost by tracking how this change affects each subsequent layer's activations until it alters the final output, providing an efficient way to optimise network performance.

After discussing the theoretical perspective of the backpropagation algorithm, we will now explore a practical example to demonstrate its functionality.

Example

This example will illustrate how a multi-layer perceptron (MLP) can effectively approximate a quadratic function with noise, showcasing the training process through backpropagation. This example is for learning purposes. In practice, you will likely use existing libraries to implement a neural network rather than creating all the required functions yourself.

As always, the first step is to import the required Python libraries.

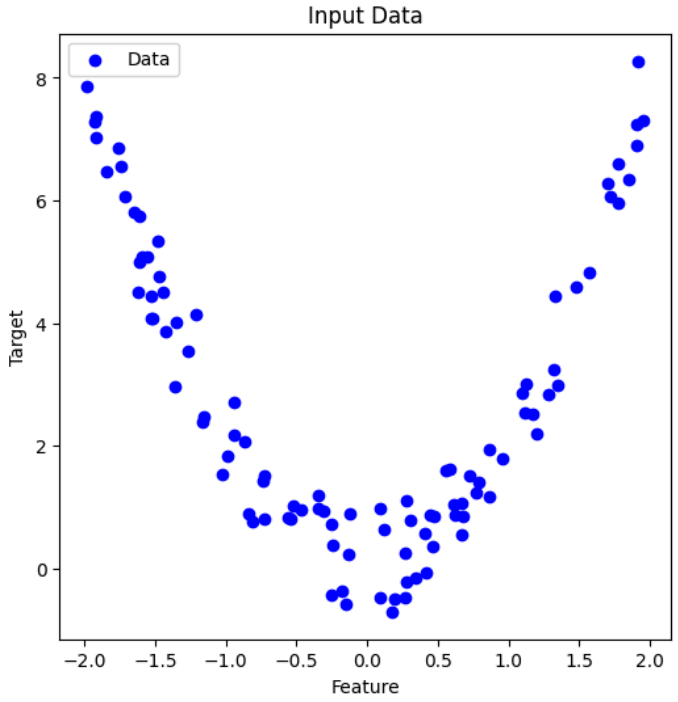
```
import numpy as np
import matplotlib.pyplot as plt
```

We will generate synthetic data ourselves that will represent the desired function. We will do this as follows:

```
np.random.seed(0)

# 100 samples, 1 feature
X = np.random.uniform(-2, 2, size=(100, 1))

# Quadratic function with noise
y = 2 * X**2 + np.random.normal(0, 0.5, size=(100, 1))
```

Let's now plot our data to see what our neural network model will work with.

```
# Plot the data
plt.figure(figsize=(6, 6))
plt.scatter(X, y, color='blue', label='Data')
plt.title('Input Data')
plt.xlabel('Feature')
plt.ylabel('Target')
plt.legend()
plt.show()
```



After looking at the data, we see that it is non-linear and that using a non-linear activation function such as the sigmoid function would be appropriate for our future network. Let's initialise it now and immediately find its derivative, as the derivative of the function will be very necessary for implementing the backpropagation algorithm.

```
# Sigmoid activation function and its derivative
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return sigmoid(x) * (1 - sigmoid(x))
```

Now it is time to set up the parameters and weight matrices for our future MLP model. The following code sequence defines the input size, the number of neurons in the hidden layer, and the output size. Additionally, weight matrices W1 and W2 are initialised with random values.

```
# Initialise parameters
input_size = 1
hidden_size = 10
```

Next, we will set up the parameters for training our model. The number of iterations is set to 1000, and the learning rate is set to 0.01. The loop initialises an empty list for storing the computed loss values and iterates for the specified number of iterations.

```
# Training parameters
num_iter = 1000
learning_rate = 0.01

# Training loop
losses = []
for iter in range(num_iter):
```

Now, it is time to define the forward pass of information through the network. First, the weighted sum `Z1` of the input `X` is computed using weights `W1`, followed by applying the sigmoid activation function to produce `A1`, the outputs of the hidden layer. Next, `Z2` is calculated as the weighted sum of `A1` using weights `W2`, representing the input to the output layer. Finally, `y_pred` is assigned the value of `Z2`, which represents the network's linear output layer for regression.

```
# Forward pass
   Z1 = np.dot(X, W1)
   A1 = sigmoid(Z1)
   Z2 = np.dot(A1, W2)
   y_pred = Z2
```

Once we have calculated the output of the network, it is required next to calculate the mean squared error (MSE) loss between the predicted values (`y_pred`) and the actual target values (`y`) in our regression task. The computed loss is then appended to the losses list, enabling us to track the loss values.

```
   loss = np.mean((y_pred - y)**2)
   losses.append(loss)
```

Finally, the backpropagation phase starts. It begins by computing the gradient `dZ2`, which represents the scaled difference between predicted (`y_pred`) and actual (`y`) values. This gradient is used to compute `dW2`, the gradient of the loss with respect to the weights `W2` connecting the hidden to the output layer.

In a similar manner, `dA1` is computed to propagate gradients back to the hidden layer, followed by `dZ1`, which incorporates the derivative of the sigmoid activation function applied to `Z1`.

At the end, `dW1` is computed to update the weights W1 connecting the input to the hidden layer, facilitating adjustments that optimise the network's performance during training.

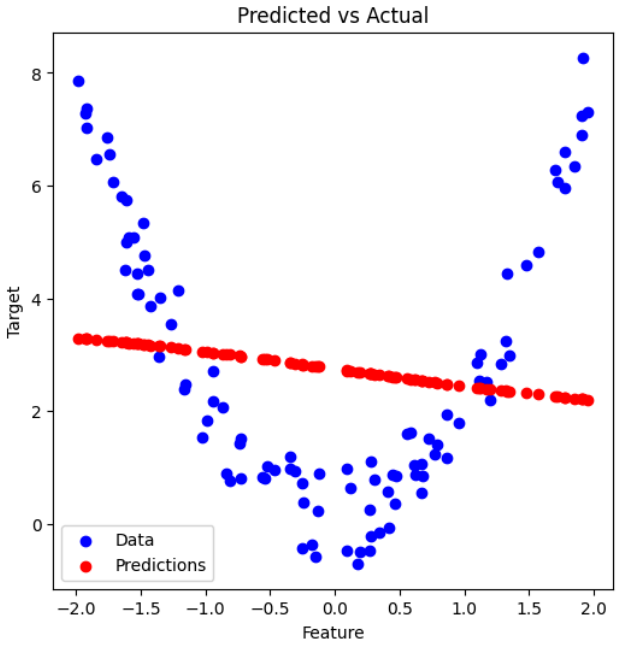
```
    dZ2 = 2 * (y_pred - y) / len(y)
    dW2 = np.dot(A1.T, dZ2)

    dA1 = np.dot(dZ2, W2.T)
    dZ1 = dA1 * sigmoid_derivative(Z1)
    dW1 = np.dot(X.T, dZ1)
```

After these steps, our algorithm can perform weight updates in the MLP during training. The weights `W2` connecting the hidden layer to the output layer are adjusted by subtracting the product of the learning rate and the gradient `dW2`. Similarly, the weights `W1` connecting the input to the hidden layer are updated using the gradient `dW1`.

```
    # Update weights
    W2 -= learning_rate * dW2
    W1 -= learning_rate * dW1
```

To see the results of our backpropagation algorithm on our network, let's plot the predicted vs actual values.

```
# Plot the predicted vs actual values
plt.figure(figsize=(6, 6))
plt.scatter(X, y, color='blue', label='Data')
plt.scatter(X, y_pred, color='red', label='Predictions')
plt.title('Predicted vs Actual')
plt.xlabel('Feature')
plt.ylabel('Target')
plt.legend()
plt.show()
```



As can be seen from the figure above, our model does not show satisfactory generalisation capabilities (the red line is not following the blue one). Now it is your turn to improve the performance of our model. Try increasing the number of neurons in the hidden layer and experimenting with different learning rate values. Why not try some other types of activation functions? It is your call!

Besides experimenting with the number of neurons and learning rate, another key technique to build high-performing neural networks is regularisation.

Regularisation

Regularisation is a technique used in machine learning to prevent overfitting. Recall that overfitting occurs when a model becomes too complex and learns the training data too well, leading to poor performance on new, unseen data. Regularisation introduces a penalty term to the cost function, discouraging the model from learning overly complex patterns. This helps the model generalise better to new data and avoid overfitting.

We previously looked at the cost function expressed as:

$$C = \frac{1}{n}\sum_{x} C_x$$

It aggregates the error over the entire dataset. For example, if we had a dataset containing 300 data points, our cost function would look like this:

$$C = \frac{1}{300}(C_1 + C_2 + \ \ldots \ + C_{299} + C_{300})$$

where C_i represents the error made on each data point. When working with neural networks, overfitting can become a significant issue when the model becomes too complex and captures noise in the training data, rather than general patterns. In other words, our neural network can memorise our training data so well, but perform poorly on unseen data. One way neural networks run into this problem is by having weights and biases that are too large.

Why do large weights and biases cause overfitting? Large weights make the output of the neural network sensitive to small changes in the input. This sensitivity can cause the model to react strongly to irrelevant details in the data, leading to poor generalisation and underperformance on new inputs. A neural network with large weight and biases has more flexibility in creating complex decision boundaries. Too much flexibility results in a model that overfits to the training data's specific characteristics. As a result, this overly complex model will underperform on unseen data.

Given everything we've discussed so far, a good approach to mitigating overfitting would be to control just how large our model parameters (the weights and biases) can become. Let's work with a cost function we are familiar with, the MSE:

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

We know the MSE we will observe depends on the set of weights and biases used in our neural network. We will use θ to represent this set of parameters. Mathematically, we represent this relationship as:

$$J(\theta) = MSE(\theta)$$

The expression tells us that our cost function, J(θ), which represents the MSE depends on θ, the set of parameters. To this expression, we are going to add a **penalty term** which will help to control the size of our parameters through a process called **regularisation**. In this task, we will discuss two approaches to regularisation: L1 regularisation and L2 regularisation.

L1 Regularisation

L1 regularisation, also known as Lasso, adds a penalty proportional to the absolute value of the coefficients to the cost function. The cost function with L1 regularisation is represented as:

$$J(\theta) = \text{MSE}(\theta) + \lambda \sum_{i=1}^{n} |\theta_i|$$

In the term we have added to our cost function, we have:

- λ – the regularisation parameter controlling the strength of the penalty
- θ_i – the model parameters

For the sake of simplicity, let's say λ = 1 (we will discuss how to vary this parameter later). Thus our cost function looks like this:

$$J(\theta) = \text{MSE}(\theta) + \sum_{i=1}^{n} |\theta_i|$$

Notice that this penalty term we have added is proportional to the absolute value of our parameters. That is, if our neural network has the following weights [0.02, -0.3, -0.55], then the cost function would be:

$$J(\theta) = \text{MSE}(\theta) + |(0.02 - 0.3 - 0.55)| = \text{MSE}(\theta) + |-0.83| = \text{MSE}(\theta) + 0.83$$

The penalty term increases the cost function by 0.23. If we were to make the weights even larger, the amount added to the cost function (the penalty) would be even larger since it's proportional to the weights. Remember, the goal is to minimise J(θ), not make it increase. So, the introduction of this penalty term will discourage the model from making the parameters, θ, too large. This will mitigate overfitting.

L2 Regularisation

L2 regularisation, also known as Ridge regression, adds a penalty proportional to the square of the coefficients to the cost function. The cost function with L2 regularisation is represented as:

$$J(\theta) = \text{MSE}(\theta) + \frac{\lambda}{2}\sum_{i=1}^{n}\theta_i^2$$

Just as before, we will make λ = 1 which gives us:

$$J(\theta) = \text{MSE}(\theta) + \sum_{i=1}^{n}\theta_i^2$$

Again, if our neural network has the following weights [0.02, -0.3, -0.55], then the cost function would be:

$$J(\theta) = \text{MSE}(\theta) + (0.02 - 0.3 - 0.55)^2 = \text{MSE}(\theta) + (-0.83)^2 = \text{MSE}(\theta) + 0.6889$$

L2 regularisation adds a penalty term of 0.6889 to the cost function. Just as with L1 regularisation, this penalty is proportional to the size of the weights. So larger weights will result in a greater penalty, causing the cost function to increase by a greater amount. So the model will be discouraged from having weights that are too large, mitigating overfitting.

Select the right technique

The choice between L1 and L2 regularisation depends on the nature of our features. L1 regularisation is particularly useful when we believe that many features are irrelevant, as it tends to eliminate them by setting their corresponding coefficients to zero. L2 regularisation is often preferred when we expect most features to be relevant but still want to avoid overfitting by constraining the size of the coefficients. To determine which regularisation method is most appropriate, it is often helpful to use feature selection techniques.

Regularisation parameter, λ

We have not forgotten about the regularisation parameter, λ. This parameter determines how much to penalise the model for having large weights. Let's revisit the L1 case where the regularised cost function is represented as:

$$J(\theta) = \text{MSE}(\theta) + \lambda \sum_{i=1}^{n} |\theta_i|$$

In our example, we had set λ = 1 and neural network weights to [0.02, -0.3, -0.55] which resulted in the following:

$$J(\theta) = \text{MSE}(\theta) + |(0.02 - 0.3 - 0.55)| = \text{MSE}(\theta) + 0.83$$

Now, let us set λ = 0.5 using the same weights. The regularised cost function will be:

$$J(\theta) = \text{MSE}(\theta) + 0.5 \times |(0.02 - 0.3 - 0.55)| = \text{MSE}(\theta) + 0.5(0.83) = \text{MSE}(\theta) + 0.415$$

Notice that the penalty term added is smaller than before. Adjusting λ has affected the size of the penalty term in the regularised cost function, which is its exact purpose. The role of λ in controlling the strength of this penalty is crucial because:

- Higher values of λ result in stronger regularisation while lower values of λ result in weaker regularisation.
- If λ is too weak, the penalty term will barely have any effect, so the risk of overfitting will not be adequately reduced.
- If λ is too strong, the effect of the penalty term will be too much, resulting in the model's weights being too small, leading to underfitting.

Techniques such as experimentation and cross-validation can help us tune this parameter to find the optimal value to use for our neural network.

Take note

Regularisation can be used to mitigate overfitting in many machine learning models, not just neural networks.

Practical task

1. Make a copy of the backpropagation.ipynb file in your folder and rename it **backpropagation_task.ipynb**.
2. You are tasked with creating a MLP for a binary classification problem using back-propagation. The goal is to classify data points into two classes based on a single feature. The network will consist of an input layer with one feature, a single hidden layer with four nodes and output layer with one node for binary classification.
3. First, initialise the following parameters:
    - Size of the network
    - Number of iterations
    - Learning rate
4. Complete the training loop code to train an MLP.
5. Experiment with different learning rates and the number of iterations. Comment on which values worked well for you.
6. Change the number of nodes in the hidden layer and comment on the effect on your model.
7. Change the activation function to an activation function of your choice. Describe any changes to your model if any.

Important: Be sure to upload all files required for the task submission inside your task folder and then click "Request review" on your dashboard.

Share your thoughts

Please take some time to complete this short feedback **form** to help us ensure we provide you with the best possible learning experience.

Reference list

Prince, S. J. D. (2023). *Understanding Deep Learning*. The MIT Press. **http://udlbook.com**

Nielsen, M. (2019). Neural Networks and Deep Learning. Retrieved from **http://neuralnetworksanddeeplearning.com/**