COE548: LARGE LANGUAGE MODELS

Topic: Deep Learning Overview Pt. 2

LAU
الجامعة اللبنانيّة الأميركيّة
Lebanese American University

Outline

Deep Learning

- Recap on Neural Networks
- Backpropagation
  - Loss Functions
  - Gradient Descent, Chain Rule
  - Optimizers
- Code Example

Neural Network Overview

Refer to L9 in Lecture Notes

- **Forward Propagation:** Data propagates from the input to the output layer.
- **Loss Function:** Measures the difference between the network's prediction and the actual target values.
- **Backpropagation:** The process of updating the weights of the network by calculating the gradient of the loss function with respect to each weight.
- **Optimization:** The process of gradually adjusting the weights (or parameters) of the neural network to minimize the loss.

Backpropagation

Motivational example

- Say the neural network given by the function $f_{NN}\left(x^{(0)}\right)$ is

$$f_{NN}\left(x^{(0)}\right) = w_k x^{(0)} + b_k$$

Where $x^{(0)} \in \mathbb{R}^1$ is some scalar input, and w_k, b_k are scalar parameters at training iteration k. In backpropagation our aim is to find the values of w and b that best fit our training data.

Note: In my notation I use the superscript parenthesis to denote layer numbers, so 0 is input layer.

Let us assume our data is sampled from a fixed known (non-noisy) function $y = 10x^{(0)}$.

Backpropagation

Motivational example

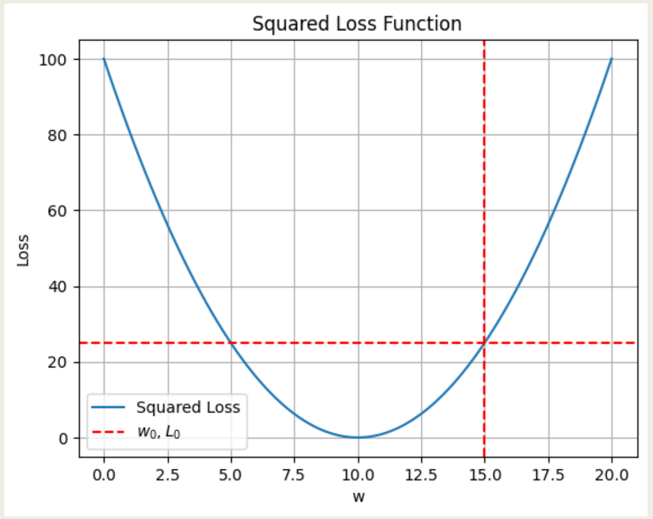
$$\hat{y} = f_{NN}\left(x^{(0)}\right) = w_k x^{(0)} + b_k, \qquad y = 10x^{(0)}$$

As we discussed earlier, we initialize a neural network with random values. For simplicity of the motivational example let's disregard b_k. Let us then set $w_{k=0} = 15$.

For $x^{(0)} = 1$ we have a true output $y = 10$, our neural network will output $\hat{y} = 15$.

Choose the squared loss function $L = (\hat{y} - y)^2$

Then, $L = (15 - 10)^2 = 25$



Backpropagation

- Backpropagation is a method used to compute the gradient of the loss function of a neural network with respect to its weights. This gradient is then propagated back through the network, layer by layer, to update the weights in a way that minimally reduces the error in the output.

- In our example, this means

$$\frac{\partial L}{\partial w_{k=0}} = \frac{\partial \left(w_{k=0}x^{(0)} - 10x^{(0)}\right)^2}{\partial w_{k=0}} = 2\left(w_{k=0}x^{(0)} - 10x^{(0)}\right)x^{(0)}$$

$$\frac{\partial L}{\partial w_{k=0}} = 2(15 - 10) \cdot 1 = 10$$

Backpropagation: Gradient Descent

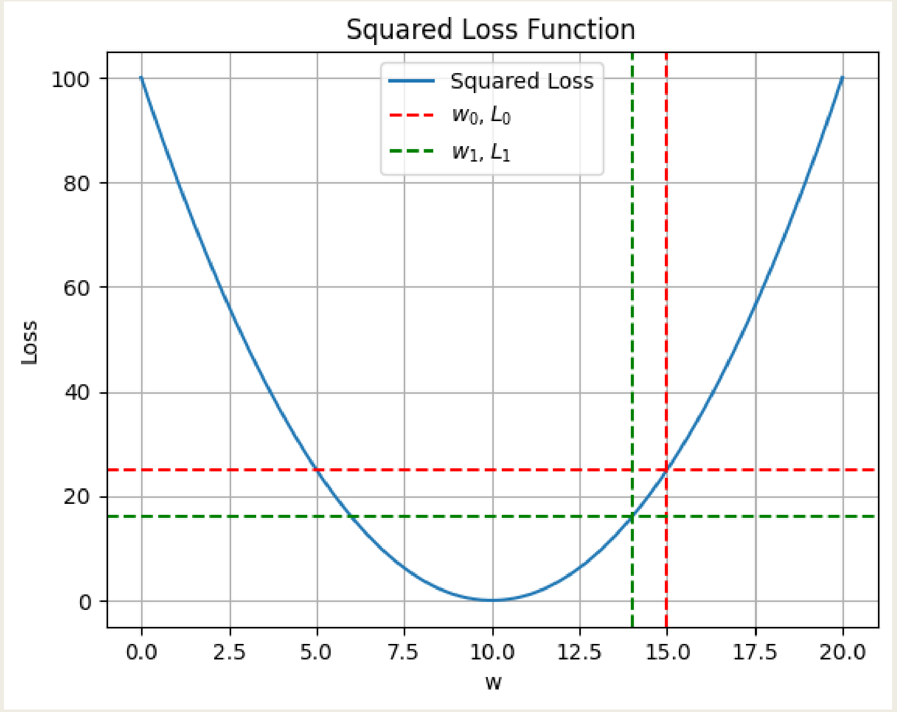
- The algorithm we will use to update any parameter is

$$x_k = x_{k-1} - \alpha \frac{\partial L}{\partial x_{k-1}}$$

Where α is what we call the learning rate. This is called **gradient descent.** It controls how much change we make to the parameters based off the gradient of the loss with respect to that parameter.

In our example, let us choose $\alpha = 0.1$. Thus we have:

$$w_{k=1} = w_{k=0} - 0.1 \frac{\partial L}{\partial w_{k=0}} = 15 - 0.1 \cdot 10 = 14$$

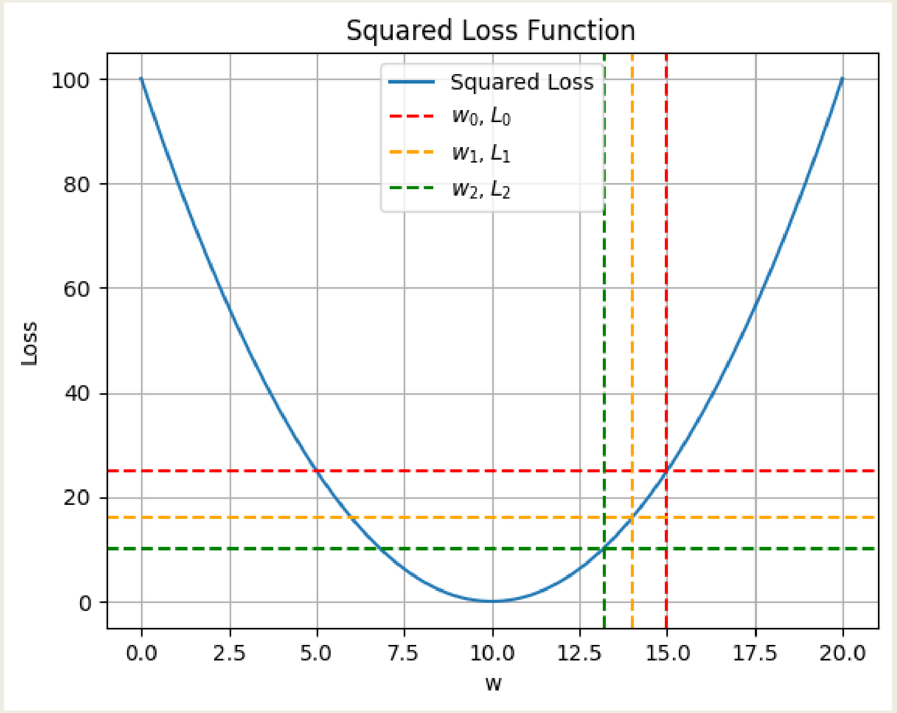


Backpropagation: Gradient Descent

- The algorithm we will use to update any parameter is

$$x_k = x_{k-1} - \alpha \frac{\partial L}{\partial x_{k-1}}$$

Where α is what we call the learning rate. This is called **gradient descent**. It controls how much change we make to the parameters based off the gradient of the loss with respect to that parameter.



In our example, let us choose $\alpha = 0.1$. Thus we have:

$$w_{k=2} = w_{k=1} - 0.1 \frac{\partial L}{\partial w_{k=1}} = 14 - 0.1 \cdot 8 = 13.2$$

Backpropagation: Chain Rule

In a more realistic scenario, our neural network has numerous hidden layers, with non-linear activation functions, and many parameters to update. Thus, backpropagation will have to be applied to each parameter, which means we will use the chain rule. Consider a neural network with L hidden layers:

$$\hat{y} = f_{NN}\big(x^{(0)}\big) = W^{(L)}\underbrace{\sigma\Big(W^{(L-1)}\underbrace{\sigma\Big(\dots W^{(1)}\underbrace{\sigma\big(W^{(0)}x^{(0)}\big)}_{x^{(1)}}}_{x^{(L-1)}}}_{x^{(L)}}\Big)\Big)$$

Then for some loss function, $\mathcal{L}$, we have:

$$\frac{\partial \mathcal{L}}{\partial W^{(l)}} = \frac{\partial \mathcal{L}}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial x^{(L)}}\frac{\partial x^{(L)}}{\partial x^{(L-1)}}\dots\frac{\partial x^{(l+1)}}{\partial W^{(l)}}$$

Backpropagation: Chain Rule

Simple example of chain rule for a neural network with one hidden layer and sigmoid activation function, and scalar input with one neuron per layer:

$$\hat{y} = w^{(1)}x^{(1)} = w^{(1)}\sigma\left(w^{(0)}x^{(0)}\right)$$

Again assume the loss is the squared loss function: $\mathcal{L} = (\hat{y} - y)^2$. Then

$$\frac{\partial \mathcal{L}}{\partial w^{(0)}} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial x^{(1)}} \frac{\partial x^{(1)}}{\partial w^{(0)}}$$

$$\frac{\partial \mathcal{L}}{\partial w^{(0)}} = \frac{\partial (\hat{y} - y)^2}{\partial \hat{y}} \cdot \frac{\partial w^{(1)}x^{(1)}}{\partial x^{(1)}} \cdot \frac{\partial \sigma\left(w^{(0)}x^{(0)}\right)}{\partial w^{(0)}}$$

$$\frac{\partial \mathcal{L}}{\partial w^{(0)}} = 2(\hat{y} - y) \cdot w^{(1)} \cdot \sigma\left(w^{(0)}x^{(0)}\right)\left(1 - \sigma\left(w^{(0)}x^{(0)}\right)\right)$$

By gradient descent we then have $w_k^{(0)} = w_{k-1}^{(0)} - \alpha \frac{\partial \mathcal{L}}{\partial w_{k-1}^{(0)}}$

Backpropagation: Optimizers

- Optimizers are algorithms or methods used to change the attributes of the neural network such as weights and learning rate in order to reduce the losses.
  - *We just introduced a type of gradient descent optimizer in the previous slides.*
- Their purpose:
  - *Minimize (or maximize) a loss function or objective function that measures the performance of the neural network.*
  - *Update the network's weights and biases in an efficient manner.*
- There are multiple types of optimizers however.
  - *The choice of optimizer can significantly affect the speed and quality of the training process.*
  - *Different optimizers improve training by navigating through rough or flat areas in the loss landscape.*

Backpropagation: Optimizers

- In the previous slides we considered one training sample at a time. However, this is not favorable in most applications of neural networks where we have lots of data with varying outputs.
- When updating the parameters based on the gradient computed from a single data point, the update direction and magnitude reflect the error and characteristics of that specific sample rather than the general trends in the data.
- Key issues:
  - *High Variance: Each individual gradient computation might point in a very different direction, depending on the specifics of the single data point. This results in a very noisy update path during training, where the model parameters may oscillate or diverge significantly, rather than smoothly converging towards the global minimum.*

Backpropagation: Optimizers

- In the previous slides we considered one training sample at a time. However, this is not favorable in most applications of neural networks where we have lots of data with varying outputs.
- When updating the parameters based on the gradient computed from a single data point, the update direction and magnitude reflect the error and characteristics of that specific sample rather than the general trends in the data.
- Key issues:
  - *Slow Convergence: While SGD can help escape local minima due to its noisy nature, this same characteristic can also lead to inefficient paths toward the minimum, taking longer to converge or getting stuck in cycles.*

Backpropagation: Optimizers

- In the previous slides we considered one training sample at a time. However, this is not favorable in most applications of neural networks where we have lots of data with varying outputs.
- When updating the parameters based on the gradient computed from a single data point, the update direction and magnitude reflect the error and characteristics of that specific sample rather than the general trends in the data.
  - *Overfitting: There is a risk that the model will start to overfit to the specifics of individual samples rather than learning the more general underlying patterns in the data set. This is especially problematic if the data set contains outliers or noisy data.*
- A solution: Train in batches

Backpropagation: Optimizers

Stochastic/Mini-Batch Gradient Descent

- Stochastic gradient descent (SGD) uses a subset of training data (mini-batch) to calculate an approximation of the gradient, leading to faster iterations.

- SGD formula:

$$w_k = w_{k-1} - \alpha \nabla_{w_{k-1}} \mathcal{L}\left(w_{k-1}; x_{1:n}^{(0)}, y_{1:n}\right)$$

- If $n = 1$, then this is just like the examples we showed earlier (this is commonly known as SGD).
- If $n = N$ (entire dataset), this is what is referred to as gradient descent.
- If $1 < \mathrm{n} < N$, then this is mini-batch gradient descent. The choice of n depends on various variables, such as the data size, the computational cost one can afford, the variability in data, etc.

Backpropagation: Optimizers

Stochastic/Mini-Batch Gradient Descent

- Stochastic gradient descent (SGD) uses a subset of training data (mini-batch) to calculate an approximation of the gradient, leading to faster iterations.

- SGD formula:

$$w_k = w_{k-1} - \alpha \nabla_{w_{k-1}} \mathcal{L}\left(w_{k-1}; x_{1:n}^{(0)}, y_{1:n}\right)$$

$$\nabla_{w_k} \mathcal{L}\left(w_{k-1}, x_{1:n}^{(0)}, y_{1:n}\right) = \frac{1}{n} \sum_{i=1}^{n} \nabla_{w_k} \mathcal{L}(w_k; x_i, y_i)$$

Backpropagation: Optimizers
SGD with momentum

- Momentum is a method used to accelerate SGD in the relevant direction and dampen oscillations. It does this by adding a fraction of the update vector of the past time step to the current update.
- Two parts:
  - *Velocity update:*

$$v_k = \gamma v_{k-1} - \alpha \nabla_{w_{k-1}} \mathcal{L}\left(w_{k-1}; x_{1:n}^{(0)}, y_{1:n}\right)$$

  - *Parameter update:*

$$w_k = w_{k-1} - v_k$$

- Another way you can write it is:

$$w_k = w_{k-1} - \alpha \nabla_{w_{k-1}} \mathcal{L}(w_{k-1}) + \beta(w_{k-1} - w_{k-2})$$

Great explanation of SGDm (if you understand concept of exponential moving average): https://www.youtube.com/watch?v=k8fTYJPd3_I

Backpropagation: Optimizers

Adaptive Gradient Algorithm (AdaGrad)

- Intuition: Adapts the learning rate to the parameters, performing larger updates for infrequent parameters and smaller updates for frequent ones.

$$w_k = w_{k-1} - \frac{\alpha}{\sqrt{G_k + \epsilon}} \cdot \nabla_{w_{k-1}} \mathcal{L}(w_{k-1})$$

- Where G_k is a diagonal matrix where each diagonal element, i, is the sum of squares f the gradients w.r.t $w_{k-1,i}$ up to iteration k.
- ϵ is a term (usually very small) used to avoid division by zero.
- Parameters associated with features that occur infrequently, the accumulated gradient will be smaller, which makes the denominator in the update rule smaller. As a result, these parameters receive a relatively larger update, effectively 'learning more' from fewer examples.

Backpropagation: Optimizers

Root Mean Square Propagation (RMSProp)

- Intuition: Developed to resolve AdaGrad’s radically diminishing learning rates by using a moving average of squared gradients to normalize the gradient itself.

$$v_k = \beta v_{k-1} + (1-\beta)\nabla^2_{\mathrm{w_k}}\mathcal{L}(w_k)$$

$$w_k = w_{k-1} - \frac{\alpha}{\sqrt{v_k + \epsilon}} \cdot \nabla_{w_{k-1}}\mathcal{L}(w_{k-1})$$

- ϵ is a term (usually very small) used to avoid division by zero.
- For data with sparse features, some parameters may only receive updates sporadically. RMSprop ensures that the scaling factor for such infrequent updates does not grow too large, unlike Adagrad, where old gradients could dominate the sum indefinitely.

Backpropagation: Optimizers

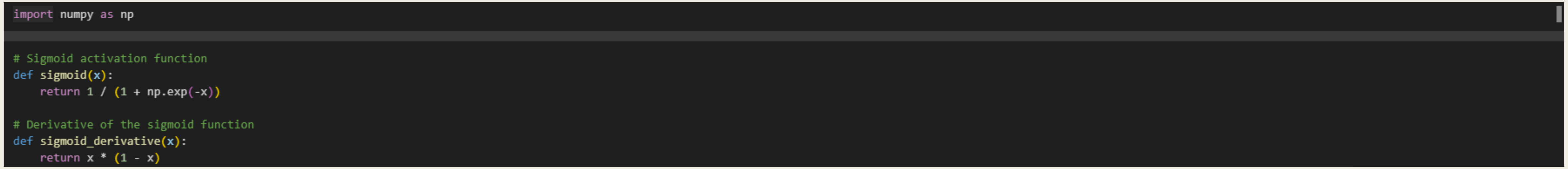
Adaptive Moment Estimation (Adam)

- Intuition: Combines elements of RMSprop and momentum. Not only stores an exponentially decaying average of past squared gradients v_k, but also keeps an exponentially decaying average of past gradients m_k, similar to momentum.

$$m_k = \beta_1 m_{k-1} + (1-\beta_1)\nabla_{w_k}\mathcal{L}(w_k)$$
$$v_k = \beta_2 v_{k-1} + (1-\beta_2)\nabla^2_{\mathrm{w_k}}\mathcal{L}(w_k)$$
$$\widehat{m}_k = \frac{m_k}{1-\beta_1^k}, \hat{v}_k = \frac{v_k}{1-\beta_2^k}$$
$$w_k = w_{k-1} - \frac{\alpha}{\sqrt{\hat{v}_k + \epsilon}} \cdot \widehat{m}_k$$

- ϵ is a term (usually very small) used to avoid division by zero, $\widehat{m}_k$ and $\hat{v}_k$ are bias-corrected versions of m_k and v_k.

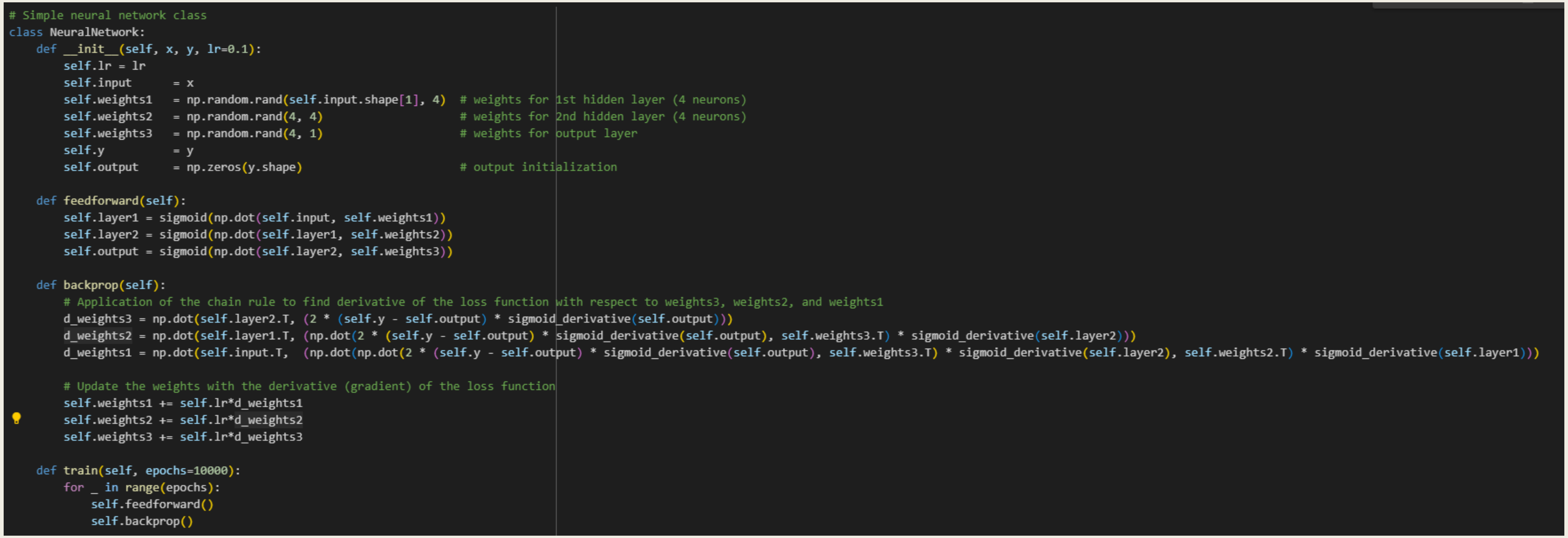
Code Example



```python
import numpy as np

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

# Derivative of the sigmoid function
def sigmoid_derivative(x):
    return x * (1 - x)
```



```python
# Simple neural network class
class NeuralNetwork:
    def __init__(self, x, y, lr=0.1):
        self.lr = lr
        self.input      = x
        self.weights1   = np.random.rand(self.input.shape[1], 4)  # weights for 1st hidden layer (4 neurons)
        self.weights2   = np.random.rand(4, 4)                    # weights for 2nd hidden layer (4 neurons)
        self.weights3   = np.random.rand(4, 1)                    # weights for output layer
        self.y          = y
        self.output     = np.zeros(y.shape)                       # output initialization

    def feedforward(self):
        self.layer1 = sigmoid(np.dot(self.input, self.weights1))
        self.layer2 = sigmoid(np.dot(self.layer1, self.weights2))
        self.output = sigmoid(np.dot(self.layer2, self.weights3))

    def backprop(self):
        # Application of the chain rule to find derivative of the loss function with respect to weights3, weights2, and weights1
        d_weights3 = np.dot(self.layer2.T, (2 * (self.y - self.output) * sigmoid_derivative(self.output)))
        d_weights2 = np.dot(self.layer1.T, (np.dot(2 * (self.y - self.output) * sigmoid_derivative(self.output), self.weights3.T) * sigmoid_derivative(self.layer2)))
        d_weights1 = np.dot(self.input.T,  (np.dot(np.dot(2 * (self.y - self.output) * sigmoid_derivative(self.output), self.weights3.T) * sigmoid_derivative(self.layer2), self.weights2.T) * sigmoid_derivative(self.layer1)))

        # Update the weights with the derivative (gradient) of the loss function
        self.weights1 += self.lr*d_weights1
        self.weights2 += self.lr*d_weights2
        self.weights3 += self.lr*d_weights3

    def train(self, epochs=10000):
        for _ in range(epochs):
            self.feedforward()
            self.backprop()
```