Data-intensive space engineering

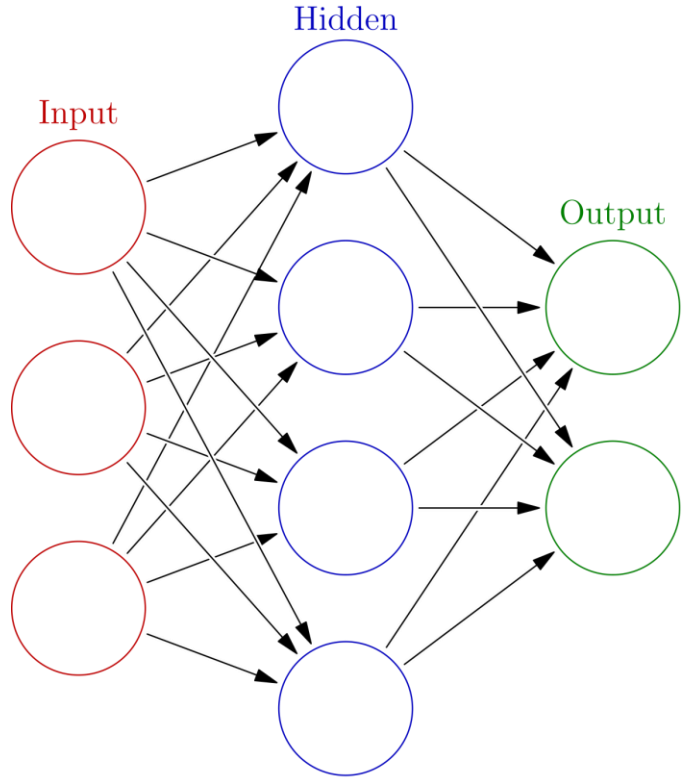
Lecture 7

Carlos Sanmiguel Vila

Introduction to Neural Networks (NNs)

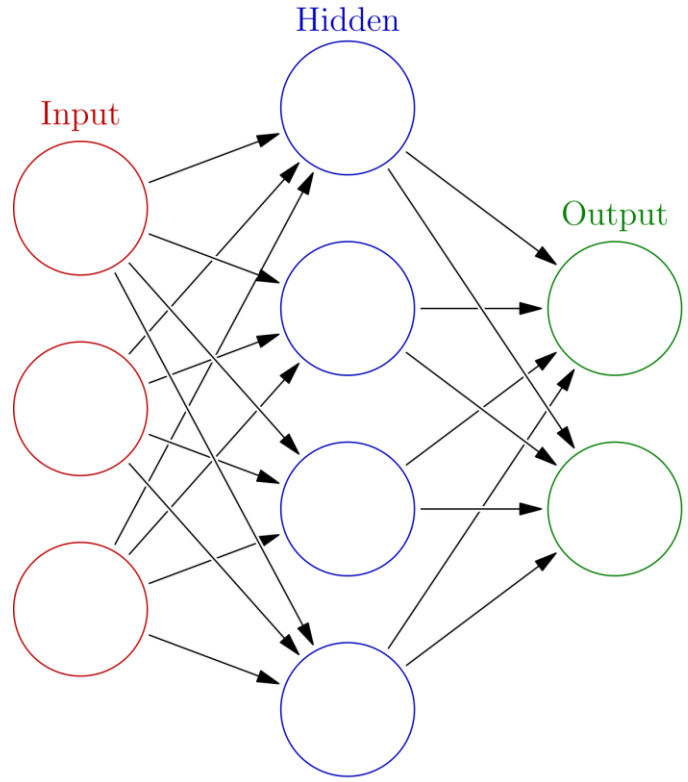
What are Neural Networks?

- A neural network is a computational model inspired by how biological brains work.
- It consists of layers of "neurons" or "nodes" that transform input data through weighted connections.
- The goal is to learn these weights through a process called training, making the model capable of predicting outcomes for unseen data.
- Compare NNs to traditional models (e.g., linear regression) NNs can capture more complex patterns.



Introduction to Neural Networks (NNs)

- A network is made up of layers: Input Layer, Hidden Layers, and Output Layer.
- Each layer consists of neurons (nodes) that process information through weighted connections.
- **Goal**: To learn the weights of these connections, enabling the network to generalize and make predictions on unseen data.
- Neurons in a network receive inputs, apply weights, and pass the result through an activation function.



Input Layer

The input layer is the first layer of the neural network, where raw data is fed into the model.

Each neuron in this layer corresponds to one feature in the input data.

For example, if you are working with images, each pixel in the image might be a feature, and therefore, each pixel value is passed to the neurons in the input layer.

In a tabular dataset, features could include mass, distance, or coordinates. Each of these features would be represented as a neuron in the input layer.

Example: If your dataset has 10 features (e.g., 10 inputs per data point), your input layer will have 10 neurons.

Key Points: The input layer does not apply any transformation or processing on the data; it simply passes it to the next layer (Hidden Layer).

Hidden Layers

The hidden layers perform transformations on the data using learned weights and biases to extract features and patterns. These are the "internal workings" of the network.

Each neuron in a hidden layer takes the output from the previous layer, applies a weighted sum, adds a bias, and then passes the result through an activation function (like ReLU, Tanh, etc.).

The number of neurons in the hidden layers is flexible. You can have multiple hidden layers (this is known as deep learning when you have more than one hidden layer), and each layer can have different numbers of neurons.

These layers are responsible for learning the complex relationships in the data by adjusting the weights and biases during training.

Hidden Layers

These layers are responsible for learning the complex relationships in the data by adjusting the weights and biases during training.

Neurons in the hidden layers compute $z=W\cdot x+b$ where W are the weights, x is the input, and b is the bias term.

The result z is passed through an activation function like ReLU, Tanh, or Sigmoid, which introduces non-linearity to the model, allowing it to learn more complex patterns.

Example: $a = ReLU(z)$ (Activation Function)

Multiple hidden layers allow the network to learn more abstract features in the data.

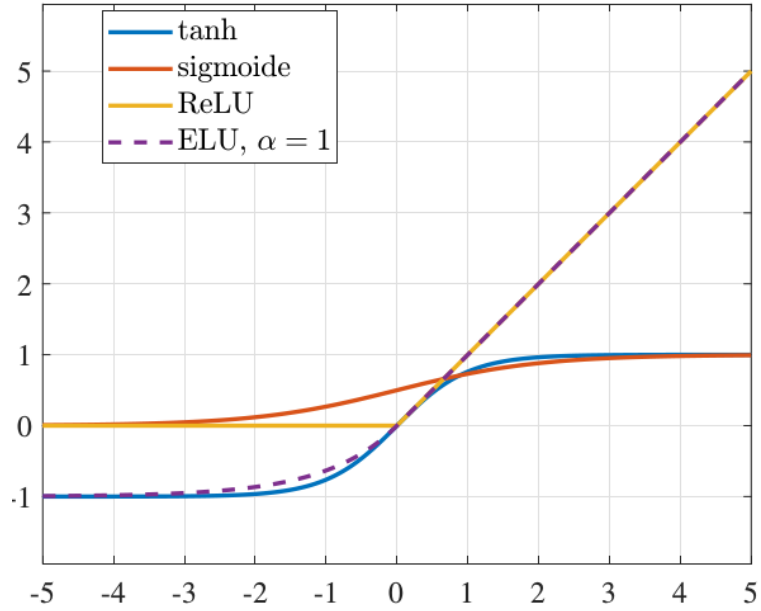
Example: In a neural network trained to classify images of digits (like MNIST), the first hidden layer might learn to detect edges, while deeper hidden layers learn to detect shapes or patterns representing specific digits.

Activation functions

The results obtained from a hidden layer are passed through an activation function to the next layers.

Why Do We Need Activation Functions?

- Activation functions introduce non-linearity to the model, enabling it to capture more complex relationships in data.
- Without an activation function, the neural network would compute a linear transformation, regardless of how many layers it has.



Activation functions

ReLU (Rectified Linear Unit)

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

Advantages: Fast and simple; mitigates the vanishing gradient problem.
Disadvantages: Can cause "dead neurons" where certain neurons stop updating if their output is always 0.

Sigmoid

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

Advantages: Smooth and maps values to a range between 0 and 1, making it suitable for binary classification.
Disadvantages: The gradient becomes very small for large positive or negative inputs (vanishing gradient problem).

Activation functions

Tanh (Hyperbolic Tangent)

$$Tanh(x) = \frac{2}{1 + e^{-2x}} - 1$$

Advantages: Outputs range between -1 and 1, which helps with the zero-centered output, improving convergence in some cases.
Disadvantages: Also suffers from the vanishing gradient problem for large inputs.

ELU (Exponential Linear Unit)

$\text{ELU}(x)=x$ if $x>0$, else $\alpha(e^{x}-1)$ with $0 < \alpha$

Advantages: Allows for negative values which help reduce the "dead neuron" problem of ReLU. It can also push mean activations closer to zero, improving learning.
Disadvantages: Computationally more expensive than ReLU.

Output Layer

- The output layer produces the final predictions of the network, either a **classification** or **regression** result.
- The output layer contains neurons that correspond to the target prediction.
- In **regression problems**, the output layer usually has **one neuron** that predicts a continuous value (e.g., predicting a quantity).
- In **classification problems**, the output layer contains **one neuron per class**. For instance, in a binary classification problem, just one output neuron might predict probabilities (using Sigmoid). In multi-class problems, there are usually **multiple neurons**, one for each class, and the network will output probabilities for each class (using Softmax).

Example: In the MNIST classification task, the output layer would have 10 neurons, each representing one of the digits (0–9). The network would predict the probability of the input image belonging to each class (digit).

The **activation function** in the output layer depends on the task:

- **Sigmoid**: For binary classification (e.g., is an email spam or not).
- **Softmax**: For multi-class classification (e.g., classifying digits 0–9).
- **No activation**: For regression tasks, where the output is a continuous value.

Example Flow of a Neural Network

Input Layer: Suppose we have 3 features in our input data (e.g., $x1$,$x2$,$x3$). These values are passed to the neurons in the input layer.
Hidden Layer: Each neuron in the hidden layer receives a weighted sum of the inputs from the input layer, applies an activation function (like ReLU), and passes the result to the next layer.
Output Layer: The final layer takes the processed values from the hidden layer(s) and produces the output:

- **Classification:** If it's a multi-class problem, each neuron in the output layer gives a probability of the input belonging to a particular class.
- **Regression:** If it's a regression problem, the output will be a single continuous value.

Input Layer: Represents the raw data features.
Hidden Layer(s): Learn complex representations through transformations using weights and activation functions.
Output Layer: Provides the final prediction, whether it's a class label or a continuous value.

How Neural Networks Learn – Forward Propagation

In forward propagation, the input data is passed through the network, layer by layer, until it reaches the output.

The network computes a prediction (output) for each input using the following steps:

- Compute the weighted sum for each neuron in the hidden layers: $z=W\cdot x+b$
- Apply the activation function: $a=f(z)$
- Pass the results to the next layer until the output layer is reached.

Objective: To compute a prediction based on the current weights and biases.

How Neural Networks Learn – Loss Function

Why do we need a loss function?

The loss function measures how far the predicted output is from the actual target.

It's a mathematical way to quantify the error between the network's predictions and the true values.

Popular Loss Functions:

Mean Squared Error (MSE): For regression tasks. $L(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y_i})^2$

Cross-Entropy Loss: For classification tasks. $L(y, \hat{y}) = -\frac{1}{n}\sum_{i=1}^{n} y_i \log(\hat{y_i})$

How Neural Networks Learn – Backpropagation and Gradient Descent

What is Backpropagation?

- Backpropagation is the process of computing the gradient of the loss function with respect to each weight in the network, using the chain rule of calculus.
- The gradients tell us how much each weight and bias in the network contributed to the overall error.

How Does It Work?

1. **Forward Pass**: Compute the output using the current weights and compute the loss.
2. **Backward Pass**: Compute the gradient of the loss function with respect to each weight by applying the chain rule.
3. **Gradient Descent**: Use the computed gradients to update the weights and biases in the direction that minimizes the loss.

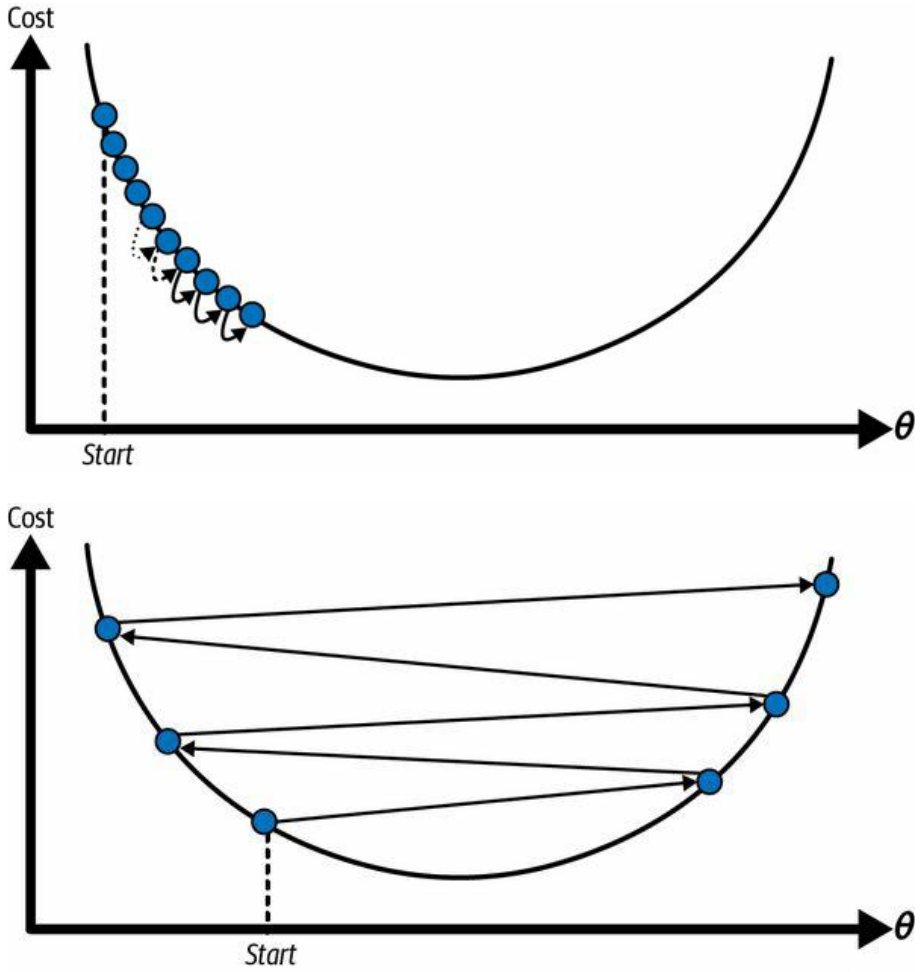
Learning Rate in Gradient Descent

The learning rate η determines the size of the steps we take when updating the weights.

Too Small: The training process will be very slow.

Too Large: The model may overshoot the optimal solution or even fail to converge.

Finding the Right Balance: Experimenting with learning rates is crucial to ensure the model converges efficiently without diverging.



Example: Gradient Update Rule with Two Hidden Layers

Consider a simple neural network with the following structure:

- Input Layer (with 3 features: $x1$,$x2$,$x3$)
- Hidden Layer 1 (with 4 neurons)
- Hidden Layer 2 (with 3 neurons)
- Output Layer (with 1 neuron for a regression task)

The activation function used in both hidden layers is **ReLU**, and there's no activation function in the output layer (because it’s a regression task).

Example: Gradient Update Rule with Two Hidden Layers

1. Forward Propagation Equations:

Let's denote:

- W_1: Weights between **Input Layer** and **Hidden Layer 1**.
- W_2: Weights between **Hidden Layer 1** and **Hidden Layer 2**.
- W_3: Weights between **Hidden Layer 2** and **Output Layer**.

Let's compute the forward pass step by step:

Hidden Layer 1:

$$z_1 = W_1 \cdot x + b_1$$
$$a_1 = ReLU(z_1)$$

Hidden Layer 2:

$$z_2 = W_2 \cdot a_1 + b_2$$
$$a_2 = ReLU\,(z_2)$$

Output Layer:

$$z_3 = W_3 \cdot a_2 + b_3$$

$y = z_3$ (no activation function since it's regression)

Example: Gradient Update Rule with Two Hidden Layers

2. Loss Function:

Assume we're using **Mean Squared Error (MSE)** as the loss function for this regression task. For a single sample:

$$L(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2$$

Where y is the true value, and $\hat{y}$ is the predicted value from the network.

3. Backpropagation and Gradient Update:

During backpropagation, we compute the **gradients** of the loss function with respect to each weight matrix W_1, W_2, W_3. This is done using the **chain rule** of calculus.

The **gradients** are used to update the weights in the direction that minimizes the loss:

$$W_i \leftarrow W_i - \eta \frac{\delta L}{\delta W_i}$$

Where η is the learning rate, and $\frac{\delta L}{\delta W_i}$ represents the gradient of the loss with respect to the weight matrix W_i

Example: Gradient Update Rule with Two Hidden Layers

During backpropagation, we compute the gradients of the loss with respect to each weight and bias.

Output Layer (backpropagation step for W_3):

$$\frac{\delta L}{\delta z_3} = y - \hat{y} \qquad \frac{\delta L}{\delta W_3} = \frac{\delta L}{\delta z_3} a_2 \qquad \frac{\delta L}{\delta b_3} = \frac{\delta L}{\delta z_3}$$

Hidden Layer 2 (backpropagation step for W_2):

$$\frac{\delta L}{\delta z_2} = \frac{\delta L}{\delta z_3} W_3 ReLU'(z_2) \qquad \frac{\delta L}{\delta W_2} = \frac{\delta L}{\delta z_2} a_1 \qquad \frac{\delta L}{\delta b_2} = \frac{\delta L}{\delta z_2}$$

Here, $ReLU'(z_2)$ is the derivative of the ReLU activation function. Since ReLU is 0 for negative inputs and 1 for positive inputs, the gradient either passes through or is set to 0.

Example: Gradient Update Rule with Two Hidden Layers

Backpropagation and Gradient Calculation

Hidden Layer 1 (backpropagation step for W_1):

$$\frac{\delta L}{\delta z_1} = \frac{\delta L}{\delta z_2} W_2 ReLU'(z_1) \qquad \frac{\delta L}{\delta W_1} = \frac{\delta L}{\delta z_1} x \qquad \frac{\delta L}{\delta b_1} = \frac{\delta L}{\delta z_1}$$

Gradient Descent (Updating Weights and Biases)

Now, we use the computed gradients to update the weights and biases in each layer using **gradient descent**. The update rule is:

$$W_i \leftarrow W_i - \eta \frac{\delta L}{\delta W_i}$$

The same process applies for biases:

$$b_i \leftarrow b_i - \eta \frac{\delta L}{\delta b_i}$$

Example: Gradient Update Rule with Two Hidden Layers

Summary of the Process:

- **Forward Propagation:** Input flows through each layer, applying weights, biases, and activation functions to compute the prediction.
- **Loss Computation:** The loss function calculates how far off the prediction is from the actual target.
- **Backpropagation:** The gradients of the loss with respect to each weight and bias are computed layer by layer, starting from the output and working backwards.
- **Gradient Descent:** Weights and biases are updated based on the gradients using the gradient descent update rule.

How Neural Networks Learn – Backpropagation and Gradient Descent

What is Backpropagation?

- Backpropagation is the process of computing the gradient of the loss function with respect to each weight in the network, using the chain rule of calculus.
- The gradients tell us how much each weight and bias in the network contributed to the overall error.

How Does It Work?

1. **Forward Pass**: Compute the output using the current weights and compute the loss.
2. **Backward Pass**: Compute the gradient of the loss function with respect to each weight by applying the chain rule.
3. **Gradient Descent**: Use the computed gradients to update the weights and biases in the direction that minimizes the loss.

Why Use PyTorch for Neural Networks?

- PyTorch is one of the most popular deep learning libraries.
- It allows for:Dynamic Computation Graphs (easy debugging and flexibility).
- GPU Support for faster computations.
- Pythonic syntax (works well with other Python libraries like NumPy).
- Real-World Adoption: Used by companies like Facebook, Tesla, and Microsoft.

Setting Up PyTorch

In Google Colab (with GPU access):

```
!pip install torch torchvision
```

Test the installation:

```
import torch
x = torch.rand(5, 3)
print(x)
```

Verify that PyTorch detects the GPU:

```
if torch.cuda.is_available():
    print("GPU is available")
else:
    print("Using CPU")
```

Introduction to Tensors

Tensors are the basic building blocks in PyTorch (similar to NumPy arrays but with GPU support).
Example:

```
tensor = torch.tensor([[1, 2], [3, 4]])
print(tensor)
```

Basic Operations:

```
y = tensor + tensor
print(y)
```

Comparing Activation Functions in PyTorch

```python
import torch
import torch.nn.functional as F

# Example input tensor
x = torch.tensor([-2.0, -1.0, 0.0, 1.0, 2.0])

# Applying different activation functions
relu_output = F.relu(x)
sigmoid_output = torch.sigmoid(x)
tanh_output = torch.tanh(x)
elu_output = F.elu(x)

print("ReLU Output:", relu_output)
print("Sigmoid Output:", sigmoid_output)
print("Tanh Output:", tanh_output)
print("ELU Output:", elu_output)
```

Example Neural Network in PyTorch

Steps to Define a Neural Network in PyTorch:

1. Define a model class by inheriting from nn.Module.
2. Create layers inside the __init__() method.

```python
import torch
import torch.nn as nn

# Defining the network class
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.fc1 = nn.Linear(10, 50)  # Input to hidden layer (10 input features, 50 n
        self.fc2 = nn.Linear(50, 1)   # Hidden to output layer (1 output for regressio

    def forward(self, x):
        x = torch.relu(self.fc1(x))  # Apply ReLU activation function on the first lay
        x = self.fc2(x)              # Linear output for regression
        return x
```

Example Neural Network in PyTorch

Steps to Define a Neural Network in PyTorch:

3. Define the forward pass in the forward() method.

4. Use optimizers and loss functions for training.

```
model = SimpleNN()  # Create an instance of the network
inputs = torch.randn(64, 10)  # Random batch of 64 samples with 10 features each
outputs = model(inputs)       # Forward pass through the network
print(outputs.shape)          # Should output (64, 1) since we have 1 output neuro
```

```
criterion = nn.MSELoss()  # Loss function for regression
targets = torch.randn(64, 1)  # Random target values for the batch
loss = criterion(outputs, targets)  # Compute the loss
print(loss)
```

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
```

```
optimizer.zero_grad()  # Clear the previous gradients
loss.backward()        # Backpropagation: compute gradients of the loss
optimizer.step()       # Update weights based on gradients
```

Example Neural Network in PyTorch

Now let's integrate all the pieces into a training loop

Key concepts:

- **Epoch**: One full pass through the entire training dataset.
- **Batch**: A subset of the dataset. Instead of passing the entire dataset at once, we break it into batches.
- **Iteration**: Each time a batch is processed, it counts as one iteration.

```python
import matplotlib.pyplot as plt

# List to store loss values for each epoch
loss_values = []

for epoch in range(100):  # Train for 100 epochs
    inputs = torch.randn(64, 10)  # Batch of inputs
    targets = torch.randn(64, 1)  # Batch of target values

    # Forward pass
    outputs = model(inputs)
    loss = criterion(outputs, targets)

    # Backward pass and optimization
    optimizer.zero_grad()  # Clear gradients
    loss.backward()        # Backpropagation
    optimizer.step()       # Update weights

    # Store the loss value for visualization
    loss_values.append(loss.item())

    if epoch % 10 == 0:
        print(f'Epoch {epoch}, Loss: {loss.item()}')

# Plotting the loss over epochs
plt.plot(range(100), loss_values)
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training Loss Over Time')
plt.show()
```

Example Neural Network in PyTorch

Steps to Define a Neural Network in PyTorch:

3. Define the forward pass in the forward() method.

4. Use optimizers and loss functions for training.

```
model = SimpleNN()  # Create an instance of the network
inputs = torch.randn(64, 10)  # Random batch of 64 samples with 10 features each
outputs = model(inputs)       # Forward pass through the network
print(outputs.shape)          # Should output (64, 1) since we have 1 output neuro
```

```
criterion = nn.MSELoss()  # Loss function for regression
targets = torch.randn(64, 1)  # Random target values for the batch
loss = criterion(outputs, targets)  # Compute the loss
print(loss)
```

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
```

```
optimizer.zero_grad()  # Clear the previous gradients
loss.backward()        # Backpropagation: compute gradients of the loss
optimizer.step()       # Update weights based on gradients
```

Example Neural Network in PyTorch

```python
import torch.nn as nn

class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.fc1 = nn.Linear(10, 50)  # Input layer to hidden layer
        self.fc2 = nn.Linear(50, 1)   # Hidden layer to output

    def forward(self, x):
        x = torch.relu(self.fc1(x))   # ReLU activation
        x = self.fc2(x)               # Linear output for regression
        return x

model = SimpleNN()
```

Example Neural Network Training

```python
import torch
import torch.nn as nn
import torch.optim as optim

# Simple Neural Network (with 1 hidden layer)
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.fc1 = nn.Linear(10, 50)  # Input to hidden layer
        self.fc2 = nn.Linear(50, 1)   # Hidden to output layer

    def forward(self, x):
        x = torch.relu(self.fc1(x))   # ReLU activation
        x = self.fc2(x)               # No activation in the output (regression)
        return x

# Create the model, define the loss function and the optimizer
model = SimpleNN()
criterion = nn.MSELoss()  # Using Mean Squared Error Loss for regression
optimizer = optim.SGD(model.parameters(), lr=0.01)  # Using stochastic gradient descen
```

Example Neural Network Training

```
# Example training loop (simplified)
for epoch in range(100):
    inputs = torch.randn(64, 10)  # Random batch of 64 samples
    targets = torch.randn(64, 1)  # Random target values

    # Forward pass
    predictions = model(inputs)
    loss = criterion(predictions, targets)

    # Backward pass
    optimizer.zero_grad()   # Zero out the previous gradients
    loss.backward()         # Compute gradients
    optimizer.step()        # Update the weights

    if epoch % 10 == 0:
        print(f"Epoch {epoch}, Loss: {loss.item()}")
```