3.5 Neural Networks with Pytorch

3.5.1 Introduction of `Pytorch`

In this section, we will introduce PyTorch, an advanced, Python-based platform specifically engineered for the realms of scientific computing, with a strong emphasis on artificial intelligence and deep learning applications. PyTorch has rapidly become a preferred tool among researchers and developers alike, thanks to its dynamic computation graph and user-friendly interface. This framework facilitates not only the rapid prototyping of models but also their efficient scaling and deployment. By offering a comprehensive set of tools and libraries, PyTorch enables practitioners to experiment with complex algorithms and innovate in machine learning and AI research. Its integration with Python, one of the most popular programming languages, further enhances its accessibility and applicability across various scientific domains.

At the heart of PyTorch's utility is the torch.Tensor object, an advanced n-dimensional array that extends beyond the capabilities of similar arrays, such as those provided by NumPy. This extension is particularly evident in torch.Tensor's ability to perform operations on Graphics Processing Units (GPUs), which are crucial for accelerating the computational processes involved in data analysis and model training. The use of GPUs can dramatically reduce the time required for training models, from days to mere hours or even minutes, depending on the complexity of the task and the dataset size. This capability is essential for deep learning applications, where large volumes of data and complex model architectures are the norms. By leveraging torch.Tensor, developers and researchers can more efficiently process data, experiment with model structures, and achieve faster iterations, leading to more innovative outcomes and breakthroughs in their projects.

PyTorch's support for computational graphs represents another foundational aspect of its architecture. Computational graphs are vital for understanding and visualizing the sequence of operations applied to tensors within neural networks. These graphs offer a structured way of looking at how data flows through a model, from input to output, including all the intermediate processing steps. This visual representation is not only beneficial for educational purposes, aiding in the clearer understanding of complex neural network architectures, but it also plays a critical role in the model development process. It allows for the identification and rectification of inefficiencies and errors in the network's design. Moreover, computational graphs facilitate the optimization of models for better performance and efficiency, making them invaluable tools for both novice learners and seasoned AI practitioners.

The feature of automatic differentiation in PyTorch stands out as a pivotal advancement in neural network training. Automatic differentiation automates the calculation of gradients or derivatives, which are essential for the backpropagation process used in training neural networks. This process involves adjusting the weights of the network based on the error between the predicted and actual outputs. Manually calculating these gradients for complex networks can be prohibitively time-consuming and prone to errors. PyTorch's automatic differentiation engine, known as Autograd, significantly simplifies this aspect of model training. It ensures that gradients are accurately and efficiently computed, enabling the optimization of network parameters with minimal effort. In our next lecture, we will dive deeper into how Autograd works and its implications for neural network training, providing you with a solid understanding of this technology and how it can be leveraged to enhance your machine learning projects.

To embark on your journey with PyTorch, the framework is designed to be easily accessible for installation and use. By visiting the official PyTorch website at https://pytorch.org/, you can find detailed instructions for installing PyTorch on various operating systems. This accessibility ensures that you can quickly get started with exploring its functionalities, regardless of your technical background. The website also offers comprehensive documentation, tutorials, and community forums, which are invaluable resources for beginners and experienced users alike. Engaging with these materials can accelerate your learning process, enabling you to master PyTorch's capabilities and apply them to your scientific computing and machine learning projects. Whether you are working on data analysis, model development, or AI research, PyTorch provides a robust and flexible platform to support your endeavors, driving innovation and efficiency in your work.

Integrating PyTorch into your computational workflow opens up a world of possibilities for developing and refining neural network models. Its intuitive design, combined with powerful computational capabilities, democratizes access to advanced machine learning techniques. This makes the development of sophisticated models more approachable for a broader audience, including students, researchers, and industry professionals. As you become more familiar with PyTorch, you'll discover its potential to facilitate groundbreaking research and innovation in artificial intelligence. By leveraging the full spectrum of features and tools that PyTorch offers, you can push the boundaries of what's possible in machine learning, contributing to the advancement of technology and science.

Remark: `PyTorch` is a Python-based tool designed for scientific computing, offering several key features:

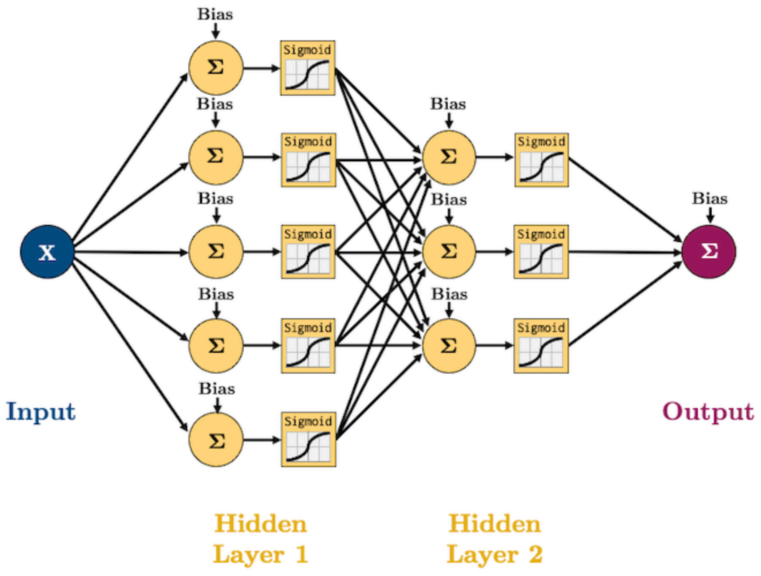
`torch.Tensor:` an n-dimensional array, akin to numpy arrays, with the additional capability to operate on GPUs.

1. Computational graphs: essential for constructing neural networks.
2. Automatic differentiation: facilitates the training of neural networks.
3. For installation, please visit the PyTorch official website: https://pytorch.org.

3.5.2 Developing with `pytoch`: a quick overview

The typical `pytorch` workflow is very similar to what we have learned in `Scikit-Learn`. The workflow looks like:

1. Define your training data: Set up your input tensors and target tensors.
2. Define a network of layers (or model): Construct a model that maps your inputs to your targets using PyTorch modules.
3. Configure the learning process: Choose a loss function, an optimizer, and metrics to monitor, similar to how you would in Keras.
4. Iterate on your training data: Instead of using the fit() method as in Keras, in PyTorch, you manually write the training loop to process the input and target tensors through the model.
5. Evaluate your model performance: After training, assess your model using the validation or test data to check its performance.



So what on Earth does that all mean? Well we are going to build up some intuition one step at a time

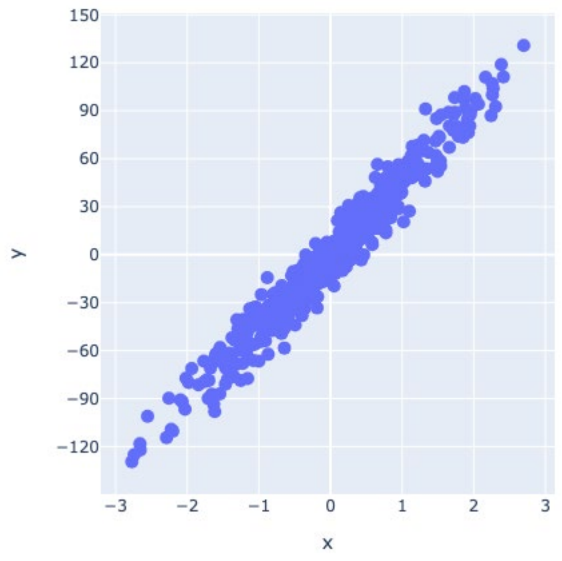
3.5.2.1 Simple Linear Regression with a Neural Network in `PyTorch`

To create a simple regression dataset with 500 observations, you can use Python libraries such as NumPy or Pandas along with Scikit-Learn for data generation. Here's how you can do it:

```
import numpy as np
import pandas as pd
from sklearn.datasets import make_regression
import matplotlib.pyplot as plt
```

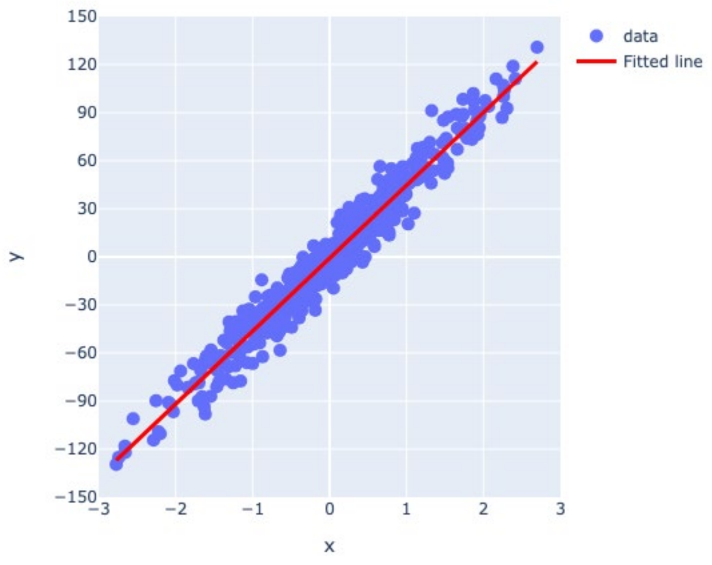
We will use Scikit-Learn's make_regression function to generate a dataset. This function allows you to specify the number of samples, features, and noise level to simulate realistic data.

```
X, y = make_regression(n_samples=500, n_features=1, random_state=0,
noise=10.0)
plot_regression(X, y)
```



We know how to fit a simple linear regression to this data using sklearn:

```
sk_model = LinearRegression().fit(X, y)
plot_regression(X, y, sk_model.predict(X))
```



Here are the parameters of that fitted line:

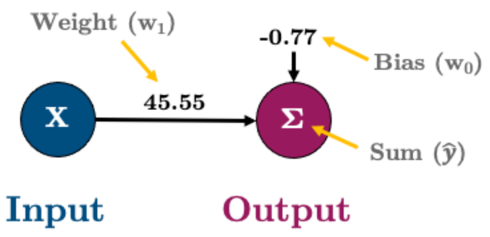
```
print(f"w_0: {sk_model.intercept_:.2f} (bias/intercept)")
print(f"w_1: {sk_model.coef_[0]:.2f}")
```

```
w_0: -0.77  (bias/intercept)
w_1: 45.50
```

As an equation, that looks like this:

$$y = -0.77 + 45.50X$$

in graph form I'll represent it like this:



Let's apply what we've discussed and start working with PyTorch to build a foundational understanding of neural networks. In PyTorch, every neural network model is derived from torch.nn.Module. Recall our discussion on class inheritance from a previous session? Inheritance allows us to utilize pre-written functionalities, eliminating the need to develop common features from scratch.

We will delve deeper into torch.nn.Module in our upcoming lab sessions. For a helpful analogy, consider how models in scikit-learn, such as .fit(), .predict(), and .score(), inherit common methods. Similarly, when we design a neural network in PyTorch, we create our unique architecture but inherit essential functionality from torch.nn.Module.

```
import sys
import numpy as np
import pandas as pd
import torch
from torchsummary import summary
from torch import nn, optim
from torch.utils.data import DataLoader, TensorDataset
from sklearn.datasets import make_regression, make_circles, make_blobs
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression
from utils.plotting import *
```

Now, let's define a model named linearRegression, and I will guide you through the syntax involved:

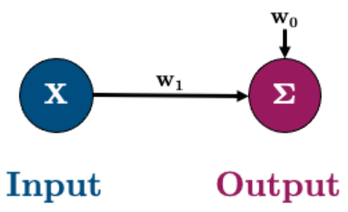
```
class linearRegression(nn.Module):  # our class inherits from nn.Module
and we can call it anything we like
    def __init__(self, input_size, output_size):
        super().__init__()                                  #
super().__init__() makes our class inherit everything from
torch.nn.Module
        self.linear = nn.Linear(input_size, output_size)  # this is a
simple linear layer: wX + b

    def forward(self, x):
        out = self.linear(x)
        return out
```

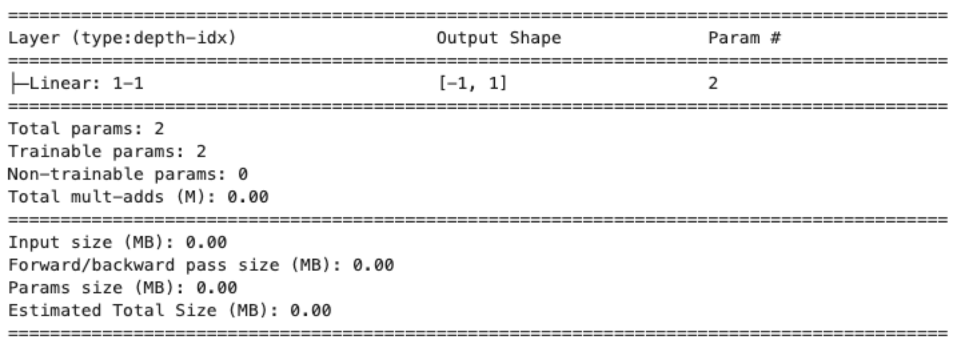
Let's walk through the implementation of the linearRegression class using PyTorch:

1. We start by creating a class linearRegression that extends nn.Module, making it a part of PyTorch's neural network module system.
2. We define a linear layer within the constructor that will compute wX + b, where w is the weight matrix, and b is the bias.
3. The forward method directs how data flows through the model. It's mandatory for PyTorch models and is triggered automatically when you make predictions using the model instance.

This setup makes linearRegression not just a simple linear regression model but also integrates it into the PyTorch ecosystem, leveraging features like automatic differentiation and GPU acceleration. After defining the model class, we can create an instance of that class:



```
summary(model, (1,));
```



```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
├─Linear: 1-1                            [-1, 1]                   2
==========================================================================================
Total params: 2
Trainable params: 2
Non-trainable params: 0
Total mult-adds (M): 0.00
==========================================================================================
Input size (MB): 0.00
Forward/backward pass size (MB): 0.00
Params size (MB): 0.00
Estimated Total Size (MB): 0.00
==========================================================================================
```

In our linearRegression model, we have two key parameters: the weight (w1) and the bias (w0). PyTorch automatically initializes these parameters randomly when you create the model. This initialization is a fundamental part of machine learning algorithms, as it provides the starting point for the training process.

You can view the current values of these parameters by accessing the state_dict of the model. Here's how you can do it:

```
print(model.state_dict())
```

This command will display a dictionary containing the values of the weight and the bias under the keys 'linear.weight' and 'linear.bias', respectively. The state_dict is an essential tool in PyTorch that provides a snapshot of the model's parameters at any point. This is particularly useful for saving and loading models after training.

Before proceeding with training our model, it's important to ensure that our data is in the correct format. Since we are using PyTorch, we need to convert our data from numpy arrays into PyTorch tensors. PyTorch tensors are similar to numpy arrays but have additional capabilities that are essential for training models, such as GPU acceleration.

Here's how you can convert numpy arrays to PyTorch tensors:

```
X_t = torch.tensor(X, dtype=torch.float32)
y_t = torch.tensor(y, dtype=torch.float32)
```

Once the data is converted into tensors, you can proceed with defining your dataloaders and training your model. With our model set up and data properly formatted as PyTorch tensors, we can now begin to interact with the model by asking it to make predictions. This is commonly done by passing input data through the model and receiving output. Here's the basic syntax to achieve this in PyTorch:

```
y_p = model(X_t[0]).item()
print(f"Predicted: {y_p:.2f}")
print(f"   Actual: {y[0]:.2f}")
```

```
Predicted: 0.11
   Actual: 31.08
```

To effectively train our model, it's crucial to define a method for evaluating the performance of the predictions and a strategy for improving the model's parameters based on this evaluation. In PyTorch, this is accomplished through the use of a loss function and an optimizer.

```
LEARNING_RATE = 0.1
criterion = nn.MSELoss()  # loss function
optimizer = torch.optim.SGD(model.parameters(), lr=LEARNING_RATE)  #
optimization algorithm is SGD
```

Defining the Loss Function

The loss function, or "criterion" in PyTorch terminology, quantifies the difference between the predicted values and the actual target values. This measurement tells us how well the model is performing; the goal during training is to minimize this loss. For a regression task like ours, the mean squared error (MSE) is a common choice. It calculates the average of the squares of the differences between the predicted and actual values, providing a simple gauge of prediction error.

Types of Loss Function

Loss functions can vary significantly depending on the type of machine learning problem—regression, classification, or something more specialized. Here are some common loss functions used in different scenarios:

1. Regression Loss Functions

(1) Mean Squared Error (MSE) / L2 Loss: Calculates the square of the difference between predicted and actual values and averages it over all examples. It's sensitive to outliers as it tends to magnify the effects of larger errors.

$$MSE = \frac{1}{N} * sum\left(\left(yi - y^{i}\right)^{2}\right) \ for\ i = 1\ to\ N$$

(2) Mean Absolute Error (MAE) / L1 Loss: Computes the absolute difference between predicted and actual values, averaging over all examples. It is less sensitive to outliers compared to MSE.

$$MAE = \frac{1}{N} * sum\left(\left|yi - y^{i}\right|\right) \ for\ i = 1\ to\ N$$

2. Classification Loss Functions

(1) Cross-Entropy Loss / Log Loss: Measures the performance of a classification model whose output is a probability value between 0 and 1. Cross-entropy loss increases as the predicted probability diverges from the actual label.

$$CE = -sum\left(y_{o,c} * log\left(p_{o,c}\right)\right) \ for\ c = 1\ to\ M$$

where M is the number of classes, y is the binary indicator (0 or 1) if class label, c is the correct classification for observation o, and p is the predicted probability that observation o is of class c

(2) Hinge Loss: Often used for "maximum-margin" classification, most notably for support vector machines (SVMs).

Choosing an Optimization Algorithm

The optimizer adjusts the model parameters (like weights and biases) to reduce the loss calculated by the criterion. For our model, we'll use stochastic gradient descent (SGD), a popular optimization algorithm that iterates over training data and updates parameters in the direction that minimally reduces the loss.

`nn.MSELoss()`: This creates an instance of the mean squared error loss function. In PyTorch, loss functions are implemented as classes, so criterion is an object that when called, computes the MSE between its two arguments.
`optim.SGD()`: This creates an optimizer object for our model's parameters. The `lr` parameter specifies the learning rate, which controls how much the parameters are changed on each update. This learning rate can be crucial for the convergence of the training process: too high a rate can cause overshooting of the minimum, while too low a rate can slow down the convergence or get stuck in local minima.

Before we begin training our model, it’s efficient to organize our data into batches. This is where a "data loader" comes into play. In PyTorch, data loaders are essentially generators that yield batches of data as requested during training. This batched approach is not only memory efficient but also can lead to faster convergence during training.

Here’s how you can set up a data loader for your dataset:

```
BATCH_SIZE = 50
dataset = TensorDataset(X_t, y_t)
dataloader = DataLoader(dataset, batch_size=BATCH_SIZE, shuffle=True)
```

Now that we have our data loader set up, we're ready to train our simple neural network using stochastic gradient descent (SGD) for a few epochs. An epoch is a complete pass through the entire dataset. We will train for 5 epochs, which means the entire dataset will be used five times to update the model parameters.

```
def trainer(model, criterion, optimizer, dataloader, epochs=5,
verbose=True):
    """Simple training wrapper for PyTorch network."""

    for epoch in range(epochs):
        losses = 0
        for X, y in dataloader:
            optimizer.zero_grad()    # Clear gradients w.r.t. parameters
            y_hat = model(X).flatten()  # Forward pass to get output
            loss = criterion(y_hat, y)  # Calculate loss
            loss.backward()       # Getting gradients w.r.t. parameters
            optimizer.step()          # Update parameters
            losses += loss.item()# Add loss for this batch to running
total
```

```
        if verbose: print(f"epoch: {epoch + 1}, loss: {losses /
len(dataloader):.4f}")

trainer(model, criterion, optimizer, dataloader, epochs=5,
verbose=True)
```

```
epoch: 1, loss: 637.3486
epoch: 2, loss: 98.6084
epoch: 3, loss: 93.4924
epoch: 4, loss: 93.8236
epoch: 5, loss: 93.4787
```

After training your PyTorch model, the parameters (weights and biases) should have been updated to minimize the loss function, potentially resulting in different and more accurate predictions compared to the initial state. Comparing the results of your PyTorch model to an equivalent model trained using Scikit-Learn can provide interesting insights into both model behaviors and effectiveness.

```
pd.DataFrame({"w0": [sk_model.intercept_,
model.state_dict()['linear.bias'].item()],
             "w1": [sk_model.coef_[0],
model.state_dict()['linear.weight'].item()]},
            index=['sklearn', 'pytorch']).round(2)
```

	w0	w1
sklearn	-0.77	45.50
pytorch	-0.60	45.55

3.5.2.1 Classfication with PyTorch in MNIST dataset

Let’s consider a concrete example in PyTorch. We'll develop a neural network to classify handwritten digits from the MNIST dataset. This dataset consists of grayscale images of handwritten digits (28 pixels by 28 pixels, totaling 784 pixels per image), categorized into one of ten classes (0 to 9).

The MNIST dataset is a classic dataset in the machine learning community, which has been around for almost as long as the field itself and has been very intensively studied. It's a set of 60,000 training images, plus 10,000 test images, assembled by the National Institute of Standards and Technology (the NIST in MNIST) in the 1980s.

The MNIST dataset comes pre-loaded `pytorch`, in the form of a set of four `Numpy` arrays:

```
import torch
import torchvision
from torchvision import datasets, transforms
from torch import nn, optim
from torch.utils.data import DataLoader
```

1. <u>Define your training data: input tensors and target tensors.</u>

When preparing data for a neural network, particularly image data for tasks like classification, it's crucial to transform the data into a format that the network can process effectively. Here's how you can achieve the transformation described:

(1) Flattening the Image: The images in your dataset are 28×28 pixels, which creates a 3-dimensional array including the batch dimension. These need to be flattened into a 1-dimensional array of 784 pixels (28 multiplied by 28) for each image. This transformation converts the 2D image into a 1D vector that can be fed into the neural network.

(2) Converting to Float: Neural networks perform better with floating-point inputs, especially when using gradient-based optimization methods, as it ensures the data has a uniform type that supports small incremental operations.

(3) Normalization: Scaling the pixel values, which range from 0 to 255, to a range between 0 and 1 helps in speeding up convergence during training. Normalizing the data generally leads to better performance as it ensures that the scale of the inputs is consistent.

(4) Prepare your data by applying transformations, loading it into suitable data structures, and organizing it into training and testing datasets:

```
# Define transformations
transform = transforms.Compose([
   transforms.ToTensor(),
   transforms.Normalize((0.5,), (0.5,))
])

# Load datasets
trainset = datasets.MNIST(root='./data', train=True, download=True,
transform=transform)
trainloader = DataLoader(trainset, batch_size=64, shuffle=True)

testset = datasets.MNIST(root='./data', train=False, download=True,
transform=transform)
testloader = DataLoader(testset, batch_size=64, shuffle=True)
```

➢ `transforms.ToTensor()`: This converts the input data from a numpy array or PIL image format into a PyTorch tensor and scales the pixel values from the range [0, 255] to [0.0, 1.0]. This effectively flattens the 28x28 images into 784-length vectors and changes their type to float.

➢ `transforms.Normalize((0.5,), (0.5,))`: This step normalizes each channel of the input tensor. Here, (0.5,) specifies the mean and (0.5,) the standard deviation for normalization. For grayscale images, there is only one channel. This normalization maps the [0.0, 1.0] range to [-1.0, 1.0], which is a common practice for neural network inputs as it centers the data around zero and scales it to a uniform range.

These preprocessing steps are crucial for neural network performance and are a standard practice in handling image data for machine learning tasks.

2. Define a network of layers (or model) that maps your inputs to your targets.

Define your neural network model by extending `nn.Module,` and set up the layers within its constructor:

```
class NeuralNet(nn.Module):
    def __init__(self):
        super(NeuralNet, self).__init__()
        self.layers = nn.Sequential(
            nn.Linear(28*28, 128),
            nn.ReLU(),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, 10),
            nn.LogSoftmax(dim=1)
        )

    def forward(self, x):
        x = x.view(x.shape[0], -1)
        return self.layers(x)

model = NeuralNet()
```

In PyTorch, you can define models in a flexible manner using the class-based system that leverages Python's object-oriented programming features. Unlike Keras, which has two specific ways of defining models (Sequential and Functional API), PyTorch uses a single, unified approach that provides more flexibility and is akin to Keras's Functional API in terms of the ability to create arbitrary architectures.

3. Configure the learning process by choosing a loss function, an optimizer, and some metrics to monitor.

We are now ready to create our simple neural network model. We will define our model in a function. This is handy if you want to extend the example later and try and get a better score.

```
criterion = nn.NLLLoss()
```

```
optimizer = optim.Adam(model.parameters(), lr=0.003)
```

The learning process is configured in the compilation step, where you specify the optimizer and loss function(s) that the model should use, as well as the metrics you want to monitor during training.

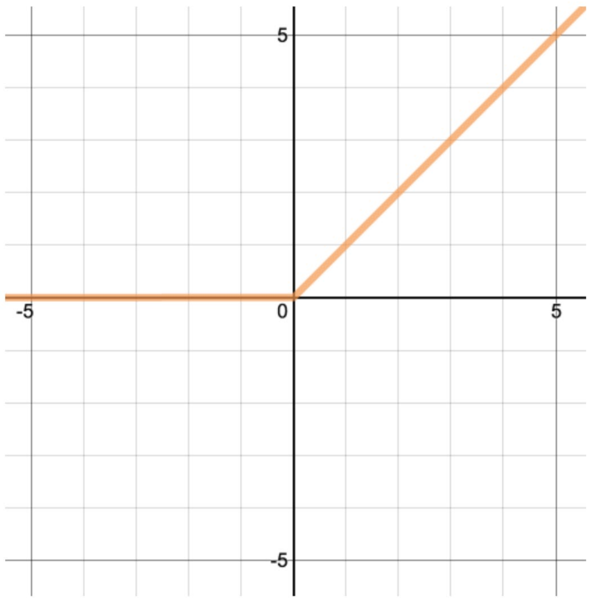
Remark: Activation functions

Rectified Linear Unit (RELU or ReLU)

A rectified linear unit is defined as:

$$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$

Rectified linear unit (RELU) is a more interesting transform that activates a node only if the input is above a certain quantity. While the input is below zero, the output is zero, but when the input rises above a certain threshold, it has a linear relationship with the dependent variable x, as demonstrated in the figure below.



RELU is **the current state of the art for hidden layers** because they have proven to work in many different situations. Because the gradient of a RELU is either zero or a constant, it does not suffer from **vanishing gradient** issues. RELU activation functions have shown to train better in practice than sigmoid activation functions such as tanh.

Softmax

Softmax is a generalization of sigmoidal function and contains **multiple** decision boundaries. The softmax activation function returns the probability distribution over mutually exclusive output classes.

Softmax is the function you will often find at **the output layer** of a classifier. If we have a multiclass modeling problem yet we care only about the best score across these classes, we'd use a softmax output layer to get the highest score of all the classes.

Remark: loss function and optimizer

The `categorical_crossentropy` belongs to the cross entropy loss function, a logarithmic loss function that's used as a feedback signal for learning the weight tensors, and which the training phase will attempt to minimize. For binary classification, this loss function of a real-valued probability prediction $\hat{y} \in (0,1)$ on a data sample with binary label $y \in (0,1)$ is defined as:

$$Loss(y,\hat{y}) = -y\ln(\hat{y}) - (1-y)\ln(1-\hat{y})$$

The cross entropy loss function has foundations in information theory and measures the amount of disagreement between y and $\hat{y}$.

The reduction of the loss happens via **mini-batch stochastic gradient descent**. The exact rules governing a specific use of gradient descent are defined by the `rmsprop` optimizer passed as the first argument.

4. Iterate on your training data

```
# Training loop
epochs = 5
training_losses = []
accuracies = []

for epoch in range(epochs):
    running_loss = 0
    for images, labels in trainloader:
        optimizer.zero_grad()
        output = model(images)
        loss = criterion(output, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    avg_training_loss = running_loss / len(trainloader)
    training_losses.append(avg_training_loss)
```

5. Evaluate your model performance.

```
   # Accuracy calculation
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in testloader:
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    accuracy = 100 * correct / total
    accuracies.append(accuracy)

    print(f"Epoch {epoch+1}: Training Loss: {avg_training_loss:.4f}, 
Accuracy: {accuracy:.2f}%")
```

```
Epoch 1: Training Loss: 0.3232, Accuracy: 93.71%
Epoch 2: Training Loss: 0.1640, Accuracy: 95.89%
Epoch 3: Training Loss: 0.1338, Accuracy: 96.39%
Epoch 4: Training Loss: 0.1195, Accuracy: 96.30%
Epoch 5: Training Loss: 0.1038, Accuracy: 96.33%
```

```
correct = 0
total = 0
with torch.no_grad():
    for images, labels in testloader:
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
# Evaluate the trained model using the test data without gradient 
updates.

print(f'Accuracy of the network on the 10000 test images: {100 * 
correct / total}%')
# Print out the accuracy of the model on the test set.
```

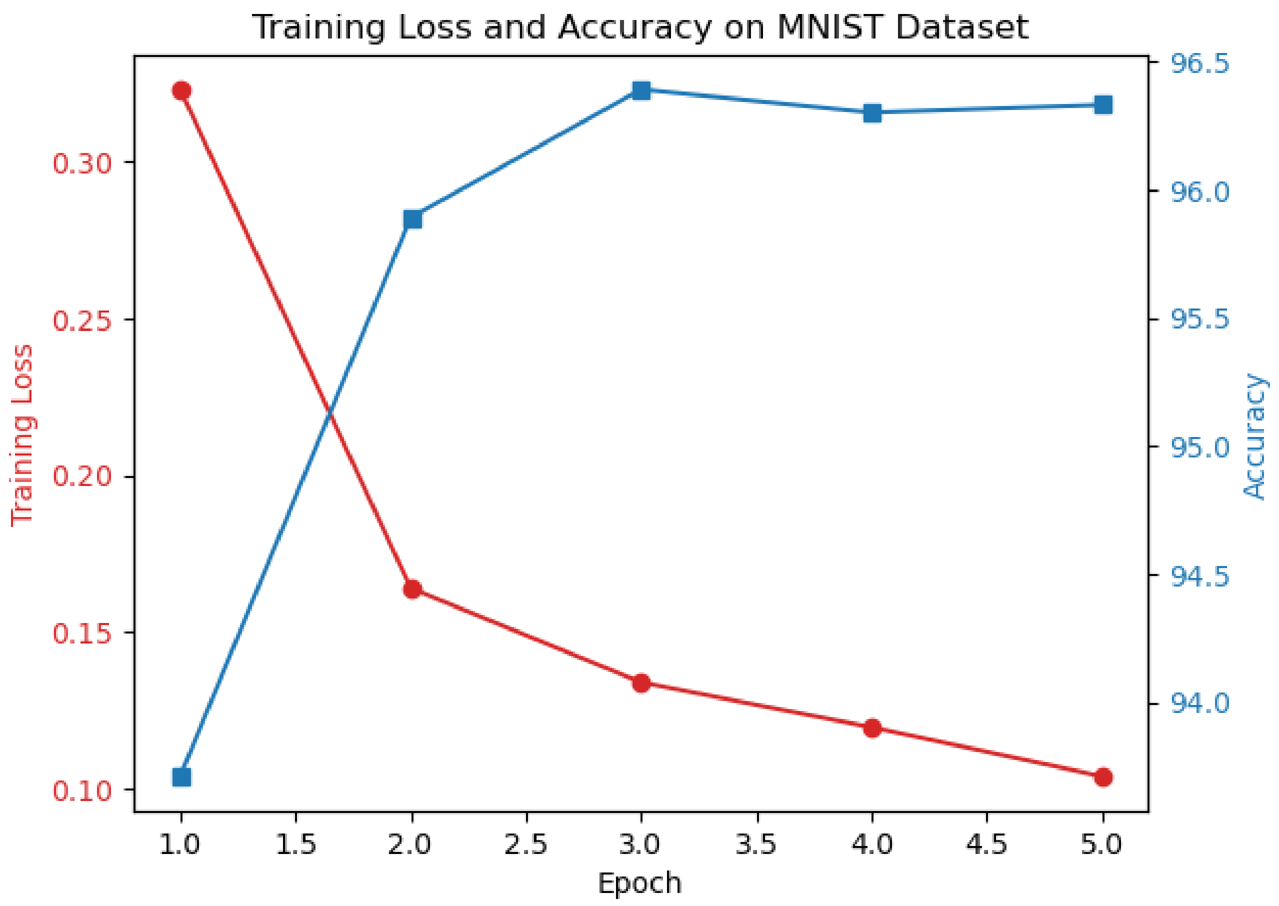
```
Accuracy of the network on the 10000 test images: 96.33%
```

```
# Creating figure
fig, ax1 = plt.subplots()

# Plotting training loss
color = 'tab:red'
ax1.set_xlabel('Epoch')
ax1.set_ylabel('Training Loss', color=color)
ax1.plot(np.arange(1, 6), training_losses, 'o-', color=color)
ax1.tick_params(axis='y', labelcolor=color)
```

```
# Creating a second y-axis for accuracy
ax2 = ax1.twinx()
color = 'tab:blue'
ax2.set_ylabel('Accuracy', color=color)
ax2.plot(np.arange(1, 6), accuracies, 's-', color=color)
ax2.tick_params(axis='y', labelcolor=color)

# Final adjustments
plt.title('Training Loss and Accuracy on MNIST Dataset')
plt.show()
```



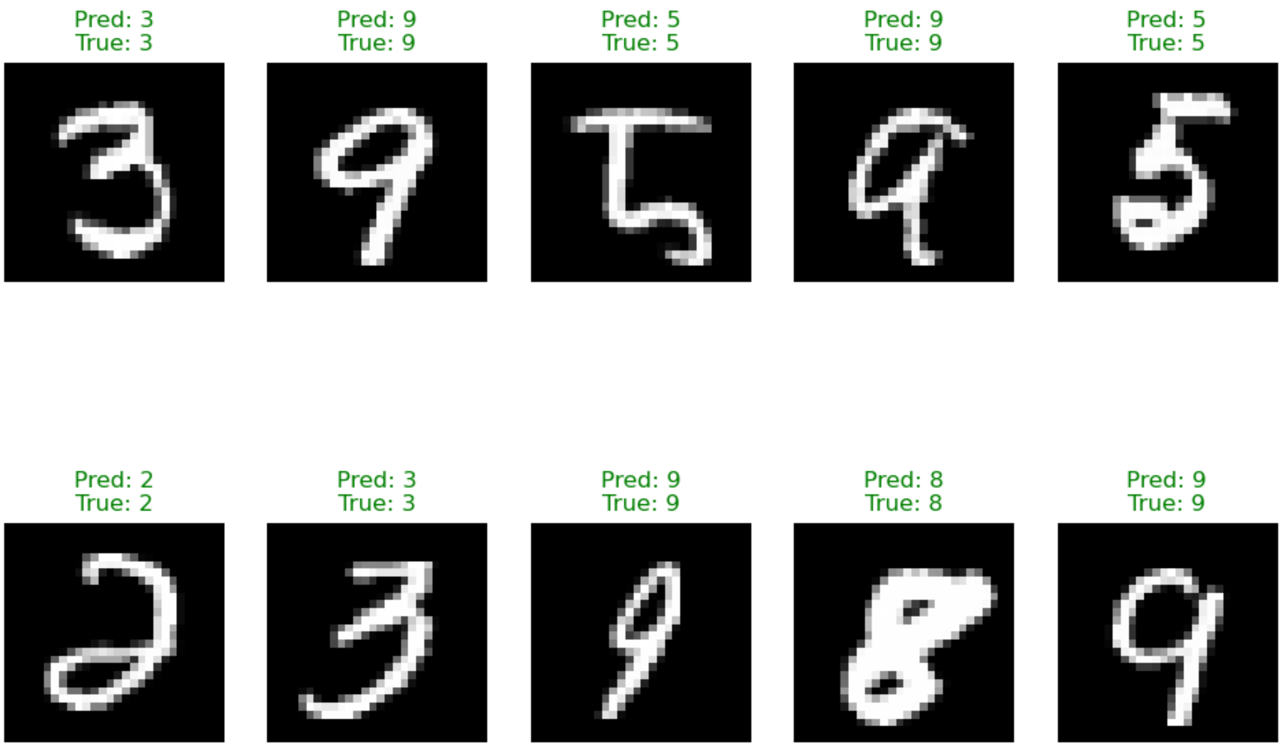
After these 5 epochs, the loss of the network will be sufficiently low that the network will be capable of classifying handwritten digits with high accuracy.

```
def imshow(img):
    img = img / 2 + 0.5  # unnormalize
    plt.imshow(img.numpy().squeeze(), cmap='gray')
```

Now we can use a function to check our prediction

```
def visualize_predictions(dataloader, model, num_images=5):
    dataiter = iter(dataloader)
    images, labels = next(dataiter)
    outputs = model(images)
    _, predicted = torch.max(outputs, 1)

    plt.figure(figsize=(12, 8))
    for idx in range(num_images):
        ax = plt.subplot(2, num_images//2, idx+1)
        imshow(images[idx])
        ax.set_title(f'Pred: {predicted[idx].item()}\nTrue:
{labels[idx].item()}', color=('green' if predicted[idx]==labels[idx]
else 'red'))
        ax.axis('off')
    plt.show()
# Call the function
visualize_predictions(testloader, model, num_images=10)
```



6. Understanding the Confusion Matrix

A confusion matrix is a powerful tool for evaluating the performance of classification models. It provides a visual and quantitative representation of the accuracy of a model by displaying the correct and incorrect predictions broken down by each class. This matrix is typically a table with rows representing the actual classes and columns representing the predicted classes. Each cell in the matrix shows the count of predictions for each class label pair, allowing you to easily identify if the model is confusing two classes (i.e., misclassifying one class as another).

The main diagonal of the confusion matrix shows the number of correct predictions made for each class, while the off-diagonal cells indicate the errors. Common metrics derived from the confusion matrix include precision, recall, and the F1-score, which help provide deeper insights into the classification performance, especially when dealing with imbalanced datasets.

Here’s a simple example of how you might compute and visualize the confusion matrix for a model trained on the MNIST dataset using Python's Scikit-learn library and visualization libraries such as

Matplotlib or Seaborn.

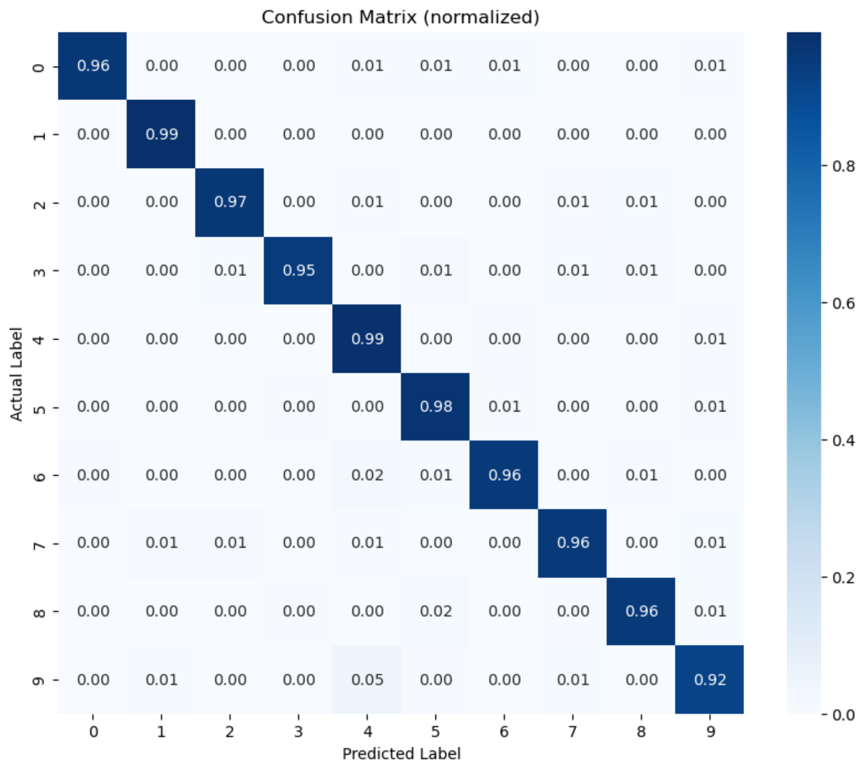
```
from sklearn.metrics import confusion_matrix
import seaborn as sns  # For visualizing the confusion matrix

# Initial setup
all_predictions = []
all_labels = []

# Model evaluation
model.eval()
with torch.no_grad():
    for images, labels in testloader:
        outputs = model(images)
        _, predicted = torch.max(outputs, 1)
        all_predictions.extend(predicted.cpu().numpy())
        all_labels.extend(labels.cpu().numpy())

# Compute confusion matrix
cm = confusion_matrix(all_labels, all_predictions)
# Normalize the confusion matrix by row (i.e., by the number of samples
in each class)
cm_normalized = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]

# Plot the normalized confusion matrix
plt.figure(figsize=(10, 8))
sns.heatmap(cm_normalized, annot=True, fmt=".2f", cmap='Blues',
xticklabels=[0,1,2,3,4,5,6,7,8,9], yticklabels=[0,1,2,3,4,5,6,7,8,9])
plt.ylabel('Actual Label')
plt.xlabel('Predicted Label')
plt.title('Confusion Matrix (normalized)')
plt.show()
```


Confusion Matrix (normalized)
Actual Label
Predicted Label
0: 0.96 0.00 0.00 0.00 0.01 0.01 0.01 0.00 0.00 0.01
1: 0.00 0.99 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
2: 0.00 0.00 0.97 0.00 0.01 0.00 0.00 0.01 0.01 0.00
3: 0.00 0.00 0.01 0.95 0.00 0.01 0.00 0.01 0.01 0.00
4: 0.00 0.00 0.00 0.00 0.99 0.00 0.00 0.00 0.00 0.01
5: 0.00 0.00 0.00 0.00 0.00 0.98 0.01 0.00 0.00 0.01
6: 0.00 0.00 0.00 0.00 0.02 0.01 0.96 0.00 0.01 0.00
7: 0.00 0.01 0.01 0.00 0.01 0.00 0.00 0.96 0.00 0.01
8: 0.00 0.00 0.00 0.00 0.00 0.02 0.00 0.00 0.96 0.01
9: 0.00 0.01 0.00 0.00 0.05 0.00 0.00 0.01 0.00 0.92