**PyTorch Tutorial: Building a Simple Neural Network From Scratch**

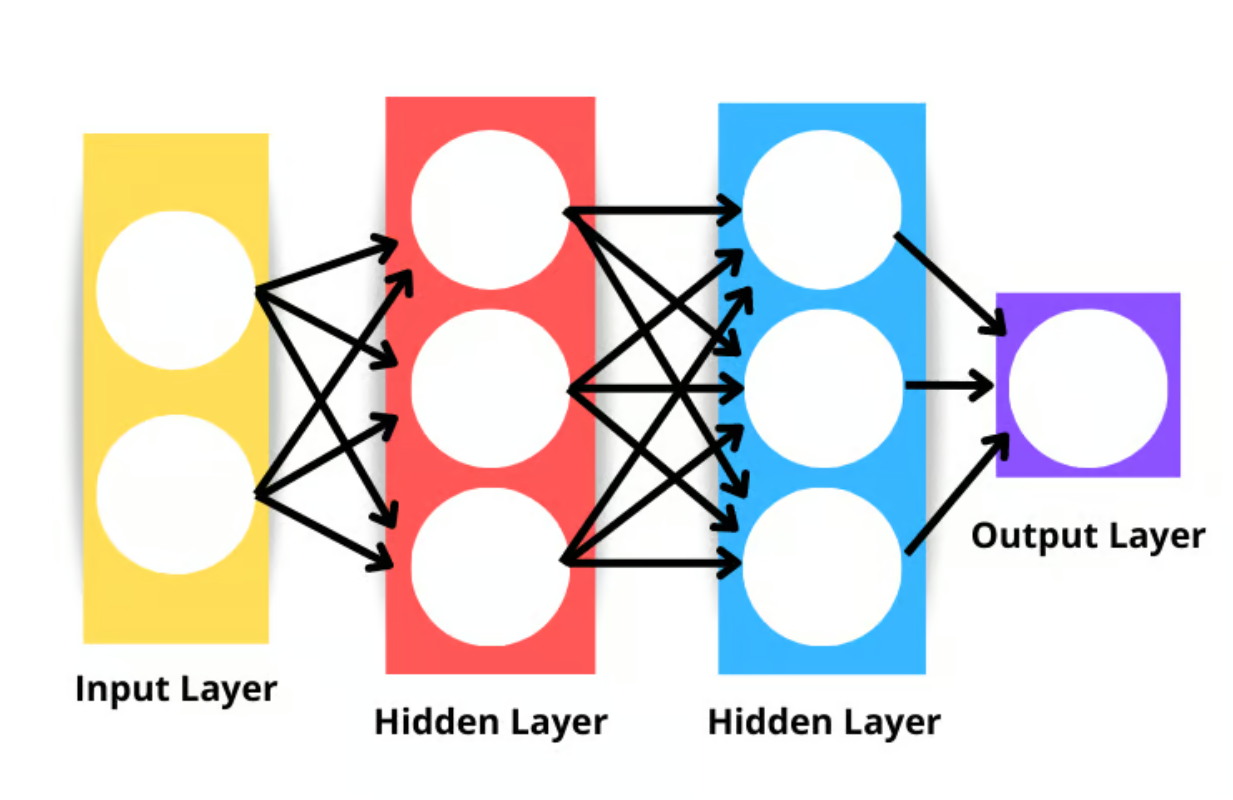
**What are neural networks?**

Neural networks are also called artificial neural networks (ANNs). The architecture forms the foundation of Deep Learning, which is merely a subset of machine learning concerned with algorithms that take inspiration from the structure and function of the human brain.  Put simply, neural networks form the basis of architectures that mimic how biological neurons signal to one another.

Consequently, you’ll often find resources that spend the first five minutes mapping out the human brain’s neural structure to help you conceptualize how a neural network works visually. But when you don’t have an extra five minutes to spare, it’s easier to define a neural network as a function that maps inputs to desired outputs.

The generic neural network architecture consists of the following:

* **Input layer**: Data is fed into the network through the input layer. The number of neurons in the input layer is equivalent to the number of features in the data. The input layer is technically not regarded as one of the layers in the network because no computation occurs at this point.
* **Hidden layer**: The layers between the input and output layers are called hidden layers. A network can have an arbitrary number of hidden layers - the more hidden layers there are, the more complex the network.
* **Output layer**: The output layer is used to make a prediction.
* **Neurons**: Each layer has a collection of neurons interacting with neurons in other layers.
* **Activation function**: Performs non-linear transformations to help the model learn complex patterns from the data.



**How do neural networks work?**

Let’s break down the algorithm into smaller components to understand better how neural networks work.

**Weight initialization**

Weight initialization is the first component in the neural network architecture. The initial weights we set to define the start point for the optimization process of the neural network model.

How we set our weights is important, especially when building deep networks. This is because deep networks are more liable to suffer from the exploding or vanishing gradient problem.  vanishing and exploding gradient problems are two concepts beyond this article's scope, but they both describe a scenario in which the algorithm fails to learn.

Although weight initialization does not completely solve the vanishing or exploding gradient problem, it certainly does contribute to its prevention.

Here are a few common weight initialization approaches:

**Zero initialization**

Zero initialization means that weights are initialized as zero. This is not a good solution as our neural network would fail to break symmetry - it will not learn.

Whenever a constant value is used to initialize the weights of a neural network, we can expect it to perform poorly since all the layers will learn the same thing. If all the outputs of the hidden units have the same influence on the cost, then the gradients will be identical.

**Random initialization**

Random initialization breaks the symmetry, which means it’s better than zero initialization, but some factors may dictate the model's overall quality.

For example, if the weights are randomly initialized with large values, then we can expect each matrix multiplication to result in a significantly larger value. When a sigmoid activation function is applied in such scenarios, the result is a value close to one, which slows down the learning rate.

Another scenario in which random initialization may cause problems is if the weights are randomly initialized to small values. In this case, each matrix multiplication will produce significantly smaller values, and applying a sigmoid function will output a value closer to zero, which also slows down the rate of learning.

**Xavier/Glorot initialization**

A Xavier or Glorot initialization - it goes by either name - is a heuristical approach used to initialize weights. It’s common to see this initialization approach whenever a tanh or sigmoid activation function is applied to the weighted average. The approach was first proposed in 2010 in a research paper titled [**Understanding the difficulty of training deep feedforward neural networks**](http://proceedings.mlr.press/v9/glorot10a.html) by Xavier Glorot and Yoshua Bengio. This initialization technique aims to keep the variance across the network equal to prevent gradients from exploding or vanishing.

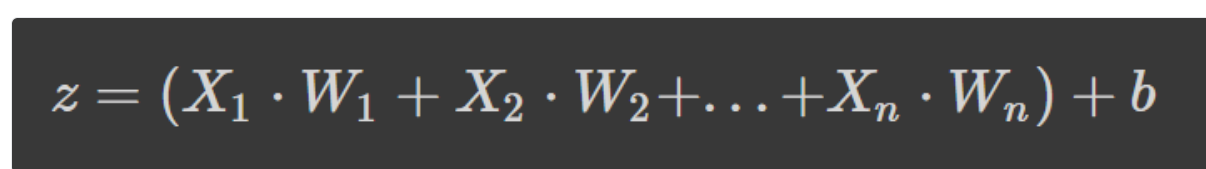
**He/Kaiming initialization**

The He or Kaiming initialization is another heuristic approach. The difference with the He and Xavier heuristic is that He initialization uses a different scaling factor for the weights that consider the non-linearity of activation functions.

Thus, when the ReLU activation function is used in the layers, He initialization is the recommended approach. You can learn more about this approach in [**Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification**](https://arxiv.org/abs/1502.01852) by He et al.

**Forward propagation**

Neural networks work by taking a weighted average plus a bias term and applying an activation function to add a non-linear transformation. In the weighted average formulation, each weight determines the importance of each feature (i.e., how much it contributes to predicting the output).



The formula above is the weighted average plus a bias term where,

* z is the weighted sum of a neuron's input
* Wn denotes the weights
* Xn denotes the independent variables, and
* b is the bias term.

If the formula looks familiar, that’s because it is linear regression. Without introducing non-linearity into the neurons, we would have linear regression, which is a simple model. The non-linear transformation allows our neural network to learn complex patterns.

**Activation functions**

We’ve already alluded to some activation functions in the [**weight initialization**](https://docs.google.com/document/d/1bBGH7BoM_9EO2mQDFv1-RfYtduqvGvqs50eftV3Zu24/edit?pli=1#heading=h.h15smvdxe10p) section, but now you know their importance of them in a neural network architecture.

Let’s delve deeper into some common activation functions you’re likely to see when you read research papers and other people's code.

**Sigmoid**

The sigmoid function is characterized by an “S”-shaped curve that is bounded between the values zero and one. It’s a differentiable function, meaning the slope of the curve can be found at any two points, and monotonic, which means it’s neither entirely increasing nor decreasing. You would typically use the sigmoid activation function for binary classification problems.

Here’s how you can visualize your own sigmoid function using Python:

**# Sigmoid function in Python**

import matplotlib.pyplot as plt

import numpy as np

x = np.linspace(-5, 5, 50)

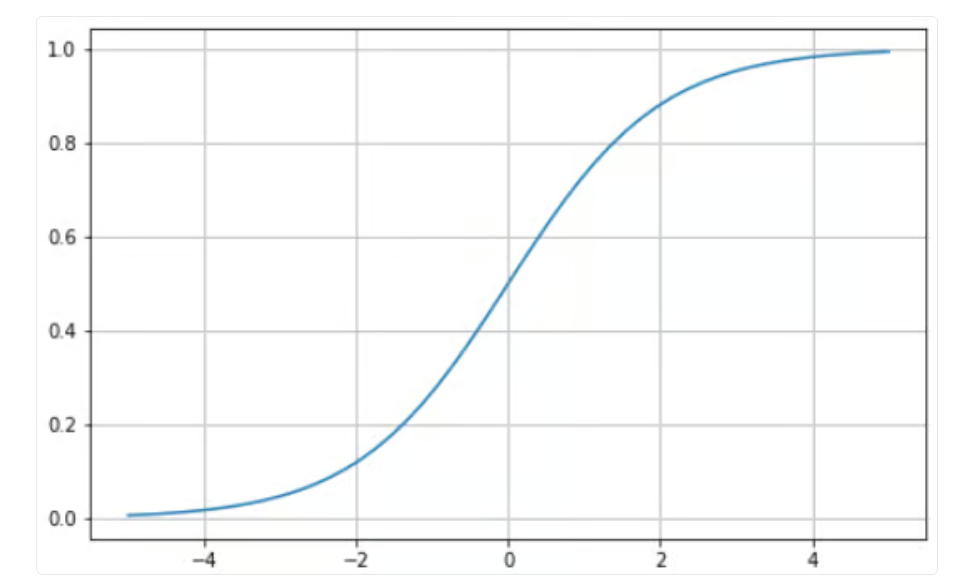
z = 1/(1 + np.exp(-x))

plt.subplots(figsize=(8, 5))

plt.plot(x, z)

plt.grid()

plt.show()



**Tanh**

The hyperbolic tangent (tanh) has the same “S”- shaped curve as the sigmoid function, except the values are bounded between -1 and 1. Thus, small inputs are mapped closer to -1, and larger inputs are mapped closer to 1.

A black background with white text

Description automatically generated

Here’s an example tanh function visualized using Python:

**# tanh function in Python**

import matplotlib.pyplot as plt

import numpy as np

x = np.linspace(-5, 5, 50)

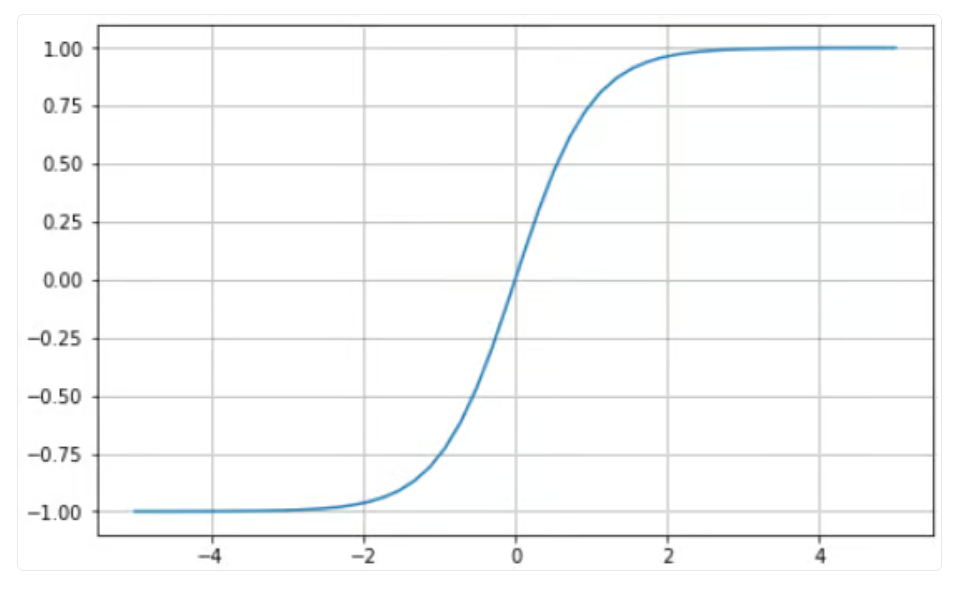
z = np.tanh(x)

plt.subplots(figsize=(8, 5))

plt.plot(x, z)

plt.grid()

plt.show()



**Softmax**

The softmax function is generally used as an activation function in the output layer. It’s a generalization of the sigmoid function to multiple dimensions. Thus, it’s used in neural networks to predict class membership on more than two labels.

**Rectified Linear Unit (ReLU)**

Using the sigmoid or tanh function to build deep neural networks is risky since they are more likely to suffer from the vanishing gradient problem. The rectified linear unit (ReLU) activation function came in as a solution to this problem and is often the default activation function for several neural networks.

Here’s a visual example of the ReLU function using Python:

# ReLU in Python

import matplotlib.pyplot as plt

import numpy as np

x = np.linspace(-5, 5, 50)

z = [max(0, i) for i in x]

plt.subplots(figsize=(8, 5))

plt.plot(x, z)

plt.grid()

plt.show()

A graph with a line

Description automatically generated

ReLU is bounded between zero and infinity: notice that for input values less than or equal to zero, the function returns zero, and for values above zero, the function returns the input value provided (i.e., if you input two the two will be returned). Ultimately, the ReLU function behaves extremely similar to a linear function, making it much easier to optimize and implement.

The process from the input to the output layer is known as the forward pass or forward propagation. During this phase, the outputs generated by the model are used to compute a cost function to determine how the neural network is performing after each iteration. This information is then passed back through the model to correct the weights such that the model can make better predictions in a process known as backpropagation.

**Backpropagation**

At the end of the first forward pass, the network makes predictions using the initialized weights, which are not tuned. Thus, it’s highly likely that the predictions the model makes will not be accurate. Using the loss calculated from forward propagation, we pass information back through the network to fine-tune the weights in a process known as backpropagation.

Ultimately, we are using the optimization function to help us identify the weights that may reduce the error rate, making the model more reliable and increasing its ability to generalize to new instances.

**PyTorch Tutorial: A step-by-step walkthrough of building a neural network from scratch**

In this article section, we will build a simple artificial neural network model using the PyTorch library.

PyTorch is one of the most popular libraries for deep learning. It provides a much more direct debugging experience than TensorFlow. It has several other perks such as distributed training, a robust ecosystem, cloud support, allowing you to write production-ready code, etc.

**Data definition & preparation**

The dataset we will be using in our tutorial is make\_circles from scikit-learn - see the [**documentation**](https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_circles.html). It’s a toy dataset containing a large circle with a smaller circle in a two-dimensional plane and two features. For our demonstration, we used 10,000 samples and added a 0.05 standard deviation of Gaussian noise to the data.

Before we build our neural network, it’s good practice to split our data into training and testing sets so we can evaluate the model's performance on unseen data.

**CODE**

**import matplotlib.pyplot as plt**

**from sklearn.datasets import make\_circles**

**from sklearn.model\_selection import train\_test\_split**

**# Create a dataset with 10,000 samples.**

**X, y = make\_circles(n\_samples = 10000,**

**noise= 0.05,**

**random\_state=26)**

**X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=.33, random\_state=26)**

**# Visualize the data.**

**fig, (train\_ax, test\_ax) = plt.subplots(ncols=2, sharex=True, sharey=True, figsize=(10, 5))**

**train\_ax.scatter(X\_train[:, 0], X\_train[:, 1], c=y\_train, cmap=plt.cm.Spectral)**

**train\_ax.set\_title("Training Data")**

**train\_ax.set\_xlabel("Feature #0")**

**train\_ax.set\_ylabel("Feature #1")**

**test\_ax.scatter(X\_test[:, 0], X\_test[:, 1], c=y\_test)**

**test\_ax.set\_xlabel("Feature #0")**

**test\_ax.set\_title("Testing data")**

**plt.show()**

**Explanation of code**

This code imports the necessary libraries for creating a dataset, splitting it into training and testing sets, and visualizing the data.

• The make\_circles function from sklearn.datasets is used to create a dataset with 10,000 samples.

• The n\_samples parameter specifies the number of samples to generate, noise adds random noise to the data, and random\_state sets the seed for the random number generator to ensure reproducibility.

• The train\_test\_split function from sklearn.model\_selection is used to split the dataset into training and testing sets.

• The test\_size parameter specifies the proportion of the dataset to include in the testing set, and random\_state sets the seed for the random number generator to ensure reproducibility.

• The fig, (train\_ax, test\_ax) = plt.subplots(ncols=2, sharex=True, sharey=True, figsize=(10, 5)) line creates a figure with two subplots, one for the training data and one for the testing data.

• The ncols=2 parameter specifies the number of columns in the figure, sharex=True and sharey=True ensure that the x and y axes are shared between the subplots, and figsize=(10, 5) sets the size of the figure.

• The train\_ax.scatter and test\_ax.scatter functions are used to plot the training and testing data, respectively.

• The c parameter specifies the color of the points based on the target variable (y\_train or y\_test).

• The set\_title, set\_xlabel, and set\_ylabel functions are used to set the title and axis labels for the training data subplot.

• Finally, plt.show() is used to display the figure.



The next step is to convert the training and testing data from [**NumPy arrays**](https://numpy.org/doc/stable/reference/generated/numpy.array.html) to **[PyTorch tensors](https://pytorch.org/docs/stable/tensors.html" \t "_blank)**. To do this we are going to create a custom dataset for our training and test files. We are also going to leverage PyTorch’s **[Dataloader](https://pytorch.org/docs/stable/data.html" \t "_blank)** module so we can train our data in batches. **Here’s the code:**

**import warnings**

**warnings.filterwarnings("ignore")**

**!pip install torch -q**

**import torch**

**import numpy as np**

**from torch.utils.data import Dataset, DataLoader**

**# Convert data to torch tensors**

**class Data(Dataset):**

**def \_\_init\_\_(self, X, y):**

**self.X = torch.from\_numpy(X.astype(np.float32))**

**self.y = torch.from\_numpy(y.astype(np.float32))**

**self.len = self.X.shape[0]**

**def \_\_getitem\_\_(self, index):**

**return self.X[index], self.y[index]**

**def \_\_len\_\_(self):**

**return self.len**

**batch\_size = 64**

**# Instantiate training and test data**

**train\_data = Data(X\_train, y\_train)**

**train\_dataloader = DataLoader(dataset=train\_data, batch\_size=batch\_size, shuffle=True)**

**test\_data = Data(X\_test, y\_test)**

**test\_dataloader = DataLoader(dataset=test\_data, batch\_size=batch\_size, shuffle=True)**

**# Check it's working**

**for batch, (X, y) in enumerate(train\_dataloader):**

**print(f"Batch: {batch+1}")**

**print(f"X shape: {X.shape}")**

**print(f"y shape: {y.shape}")**

**break**

**"""**

**Batch: 1**

**X shape: torch.Size([64, 2])**

**y shape: torch.Size([64])**

**"""**

**EXPLANATION**

This code imports necessary libraries such as warnings, torch, numpy, and DataLoader.

• It defines a class called Data that converts input data into torch tensors.

• The class has three methods: \_\_init\_\_ that initializes the class with input data, \_\_getitem\_\_ that returns the data at a specific index, and \_\_len\_\_ that returns the length of the data.

• The code then instantiates training and test data using the Data class and DataLoader.

• The DataLoader is used to load the data in batches of size 64 and shuffle the data.

• Finally, the code checks if the DataLoader is working by printing the shape of the first batch of data.

• The warnings library is used to ignore any warnings that may arise during the execution of the code.

• The !pip install torch -q command installs the torch library without displaying any output.

• The numpy library is used to convert input data into numpy arrays before converting them into torch tensors.

• Overall, this code converts input data into torch tensors and loads them into batches using DataLoader for efficient training and testing of machine learning models.

…………………………………………………………………………………………………………………..

**Neural network implementation & model training**

We are going to implement a simple two-layer neural network that uses the ReLU activation function (torch.nn.functional.relu). To do this we are going to create a class called NeuralNetwork that inherits from the nn.Module which is the base class for all neural network modules built in PyTorch.

import torch

from torch import nn

from torch import optim

input\_dim = 2

hidden\_dim = 10

output\_dim = 1

class NeuralNetwork(nn.Module):

def \_\_init\_\_(self, input\_dim, hidden\_dim, output\_dim):

super(NeuralNetwork, self).\_\_init\_\_()

self.layer\_1 = nn.Linear(input\_dim, hidden\_dim)

nn.init.kaiming\_uniform\_(self.layer\_1.weight, nonlinearity="relu")

self.layer\_2 = nn.Linear(hidden\_dim, output\_dim)

def forward(self, x):

x = torch.nn.functional.relu(self.layer\_1(x))

x = torch.nn.functional.sigmoid(self.layer\_2(x))

return x

model = NeuralNetwork(input\_dim, hidden\_dim, output\_dim)

print(model)

"""

NeuralNetwork(

(layer\_1): Linear(in\_features=2, out\_features=10, bias=True)

(layer\_2): Linear(in\_features=10, out\_features=1, bias=True)

)

"""

**This code defines a neural network model using the PyTorch library.**

• First, the necessary libraries are imported: torch, nn (short for neural network), and optim (short for optimizer).

• Next, the input, hidden, and output dimensions of the neural network are defined.

• Then, a class called NeuralNetwork is defined, which inherits from the nn.Module class.

• This class has an \_\_init\_\_ method that initializes the layers of the neural network.

• The first layer is a linear layer (nn.Linear) that takes in the input dimension and outputs to the hidden dimension.

• The weights of this layer are initialized using the Kaiming uniform initialization method with a ReLU nonlinearity.

• The second layer is another linear layer that takes in the hidden dimension and outputs to the output dimension.

• The forward method of the NeuralNetwork class defines the forward pass of the neural network.

• It takes in an input tensor x and passes it through the first layer with a ReLU activation function (torch.nn.functional.relu).

• The output of this layer is then passed through the second layer with a sigmoid activation function (torch.nn.functional.sigmoid).

• The final output of the neural network is returned.

• Finally, an instance of the NeuralNetwork class is created with the specified input, hidden, and output dimensions, and the model is printed to show the structure of the neural network.

And that’s all.

To train the model we must define a loss function to use to calculate the gradients and an optimizer to update the parameters. For our demonstration, we are going to use [**binary crossentropy**](https://pytorch.org/docs/stable/generated/torch.nn.BCEWithLogitsLoss.html) and [**stochastic gradient descent**](https://pytorch.org/docs/stable/generated/torch.optim.SGD.html) with a learning rate of 0.1.

learning\_rate = 0.1

loss\_fn = nn.BCELoss()

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

This code sets the learning rate to 0.1, which is a hyperparameter that determines the step size at each iteration while optimizing the model.

• The code also defines the loss function as Binary Cross Entropy (BCE) Loss, which is commonly used for binary classification problems.

• Finally, the code initializes the optimizer as Stochastic Gradient Descent (SGD) with the model parameters and the learning rate as inputs.

• The optimizer is responsible for updating the model parameters during training to minimize the loss function.

**Lets Train Model**

**num\_epochs = 100**

**loss\_values = []**

**for epoch in range(num\_epochs):**

**for X, y in train\_dataloader:**

**# zero the parameter gradients**

**optimizer.zero\_grad()**

**# forward + backward + optimize**

**pred = model(X)**

**loss = loss\_fn(pred, y.unsqueeze(-1))**

**loss\_values.append(loss.item())**

**loss.backward()**

**optimizer.step()**

print("Training Complete")

"""

Training Complete

"""

**Explanation**

This code trains a machine learning model using a training dataset.

• The first line sets the number of epochs to 100.

• An epoch is a complete pass through the entire training dataset.

• The for loop iterates through each epoch.

• Within each epoch, the for loop iterates through each batch of data in the train\_dataloader.

• The optimizer.zero\_grad() line sets the gradients of all model parameters to zero.

• This is necessary because PyTorch accumulates gradients by default, so we need to reset them before each iteration.

• The pred = model(X) line computes the model's predictions for the input data X.

• The loss = loss\_fn(pred, y.unsqueeze(-1)) line computes the loss between the model's predictions and the true labels y.

• The unsqueeze(-1) method adds an extra dimension to y to match the shape of pred.

• The loss\_values.append(loss.item()) line adds the current loss value to a list of loss values.

• The loss.backward() line computes the gradients of the loss with respect to all model parameters.

• The optimizer.step() line updates the model parameters based on the computed gradients.

• After all epochs and batches have been processed, the code prints "Training Complete".

**Let’s track loss value**

**step = range(len(loss\_values))**

**fig, ax = plt.subplots(figsize=(8,5))**

**plt.plot(step, np.array(loss\_values))**

**plt.title("Step-wise Loss")**

**plt.xlabel("Epochs")**

**plt.ylabel("Loss")**

**plt.show()**

**A blue line graph with text

Description automatically generated**

The visualization above shows the loss of our model over 100 epochs. Initially, the loss starts at 0.7 and gradually decreases - this informs us that our model has been improving its predictions over time. However, the model seems to plateau around the 60 epoch mark, which may be down to a variety of reasons, such as the model may be in the region of a local or global minimum of the loss function.

Nonetheless, the model has been trained and is ready to make predictions on new instances - let’s look at how to do that in the next section.

**Predictions & model evaluation**

Making predictions with our PyTorch neural network is quite simple.

**import itertools # Import this at the top of your script**

**# Initialize required variables**

**y\_pred = []**

**y\_test = []**

**correct = 0**

**total = 0**

**"""**

**We're not training so we don't need to calculate the gradients for our outputs**

**"""**

**with torch.no\_grad():**

**for X, y in test\_dataloader:**

**outputs = model(X) # Get model outputs**

**predicted = np.where(outputs.numpy() < 0.5, 0, 1) # Convert to NumPy and apply threshold**

**predicted = list(itertools.chain(\*predicted)) # Flatten predictions**

**y\_pred.append(predicted) # Append predictions**

**y\_test.append(y.numpy()) # Append true labels as NumPy**

**total += y.size(0) # Increment total count**

**correct += (predicted == y.numpy()).sum().item() # Count correct predictions**

**print(f'Accuracy of the network on the 3300 test instances: {100 \* correct // total}%')**

**"""**

**Accuracy of the network on the 3300 test instances: 97%**

**"""**

**Note:** Each run of the code would produce a different output so you may not get the same results.

The code above loops through the test batches, which are stored in the test\_dataloader variable, without calculating the gradients. We then predict the instances in the batch and store the results in a variable called outputs. Next, we determine set all the values less than 0.5 to 0 and those equal to or greater than 0.5 to 1. These values are then appended to a list for our predictions.

After that, we add the actual predictions of the instances in the batch to a variable named total. Then we calculate the number of correct predictions by identifying the number of predictions equal to the actual classes and totaling them. The total number of correct predictions for each batch is incremented and stored in our correct variable.

To calculate the accuracy of the overall model, we multiply the number of correct predictions by 100 (to get a percentage) and then divide it by the number of instances in our test set. Our model had 97% accuracy. We dig in further using the confusion matrix and scikit-learn’s classification\_report to get a better understanding of how our model performed.

**from sklearn.metrics import classification\_report**

**from sklearn.metrics import confusion\_matrix**

**import seaborn as sns**

**y\_pred = list(itertools.chain(\*y\_pred))**

**y\_test = list(itertools.chain(\*y\_test))**

**print(classification\_report(y\_test, y\_pred))**

**"""**

**precision recall f1-score support**

**0.0 0.98 0.97 0.98 1635**

**1.0 0.98 0.98 0.98 1665**

**accuracy 0.98 3300**

**macro avg 0.98 0.98 0.98 3300**

**weighted avg 0.98 0.98 0.98 3300**

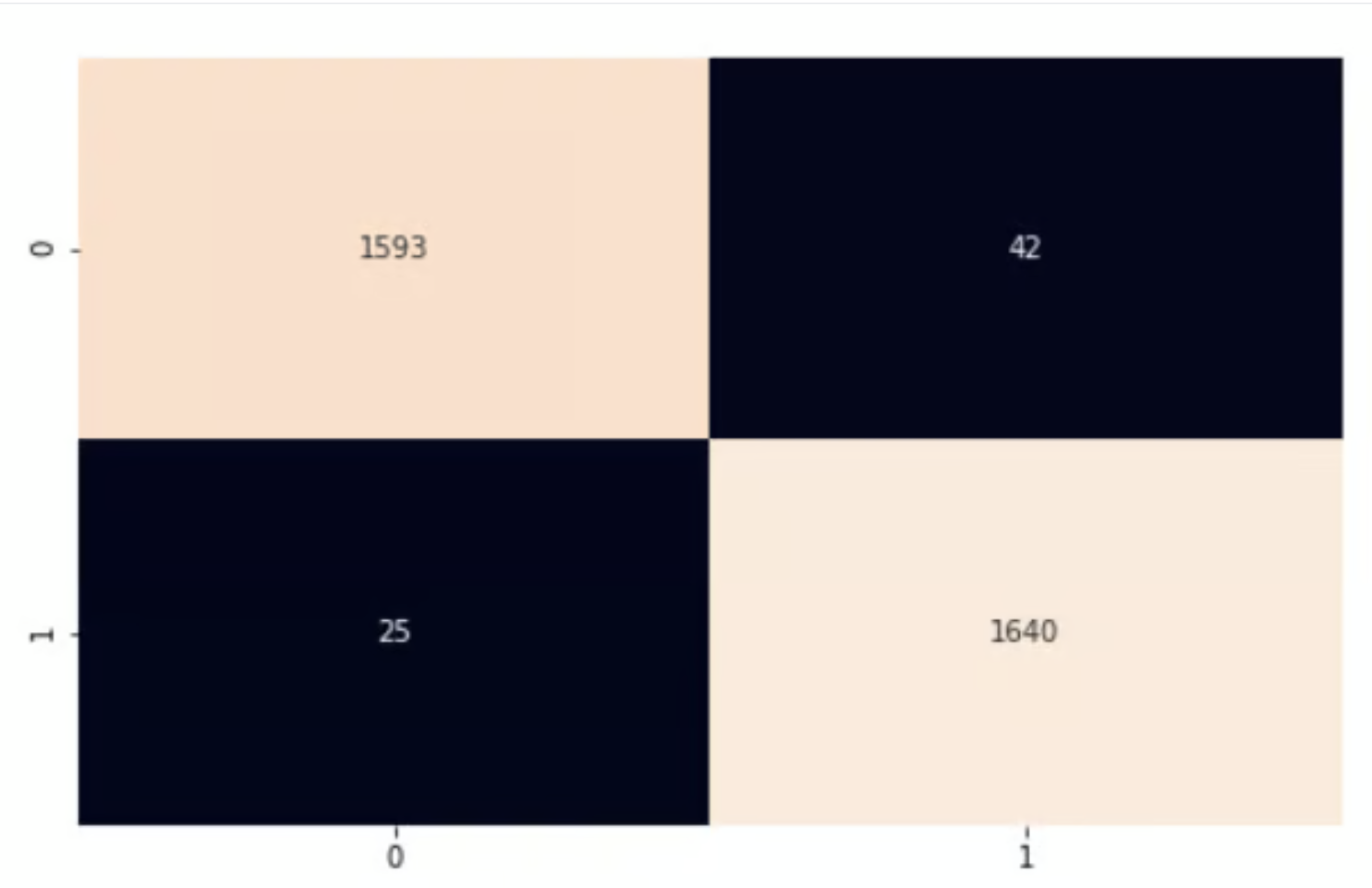
**"""**

**cf\_matrix = confusion\_matrix(y\_test, y\_pred)**

**plt.subplots(figsize=(8, 5))**

**sns.heatmap(cf\_matrix, annot=True, cbar=False, fmt="g")**

**plt.show()**

****

Our model is performing pretty well. I encourage you to explore the code and make some changes to help make what we’ve covered in this article stick.

In this PyTorch tutorial, we covered the foundational basics of neural networks and used PyTorch, a Python library for deep learning, to implement our network. We used the circle's dataset from scikit-learn to train a two-layer neural network for classification. We then made predictions on the data and evaluated our results using the accuracy metric.