1. Image Classification with the MNIST Dataset using CUDA (GPU)

Let's begin by loading the libraries used in this notebook:

```
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
from torch.optim import Adam

# Visualization tools
import torchvision
import torchvision.transforms.v2 as transforms
import torchvision.transforms.functional as F
import matplotlib.pyplot as plt
```

In PyTorch, we can use our GPU in our operations by setting the [device](https://pytorch.org/docs/stable/tensor_attributes.html#torch.device) to `cuda`. The function `torch.cuda.is_available()` will confirm PyTorch can recognize the GPU.

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
torch.cuda.is_available()
```

```
True
```

1.1 The Problem: Image Classification

In traditional programming, the programmer is able to articulate rules and conditions in their code that their program can then use to act in the correct way. This approach continues to work exceptionally well for a huge variety of problems.

Image classification, which asks a program to correctly classify an image it has never seen before into its correct class, is near impossible to solve with traditional programming techniques. How could a programmer possibly define the rules and conditions to correctly classify a huge variety of images, especially taking into account images that they have never seen?

Solution: By training a deep neural network with sufficient data, and providing the network with feedback on its performance via training, the network can identify, though a huge amount of iteration, its own set of conditions by which it can act in the correct way.

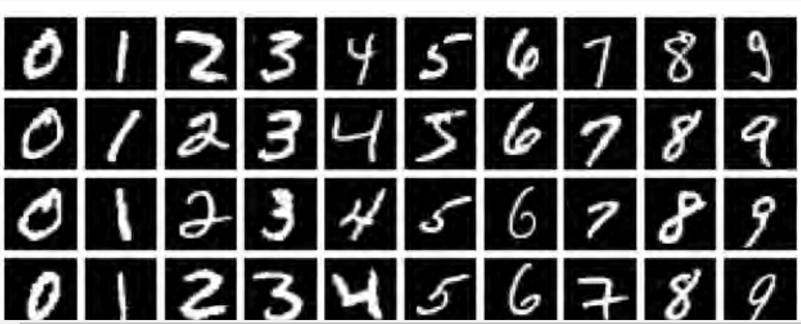
1.2 The MNIST Dataset

In the history of deep learning, the accurate image classification of the [MNIST dataset](http://yann.lecun.com/exdb/mnist/), a collection of 70,000 grayscale images of handwritten digits from 0 to 9, was a major development. While today the problem is considered trivial, doing image classification with MNIST has become a kind of "Hello World" for deep learning.

Here are 40 of the images included in the MNIST dataset:

```
from IPython.display import Image

# Display the image from the local path
Image(filename='/content/mnist1.png', width=600)
```



1.2.1 Training and Validation Data and Labels

When working with images for deep learning, we need both the images themselves, usually denoted as `X`, and also, correct [labels](#) for these images, usually denoted as `Y`. Furthermore, we need `X` and `Y` values both for *training* the model, and then, a separate set of `X` and `Y` values for *validating* the performance of the model after it has been trained.

We can imagine these `X` and `Y` pairs as a set of flash cards. A student can train with one set of flashcards, and to validate the student learned the correct concepts, a teacher might quiz the student with a different set of flash cards.

Therefore, we need 4 segments of data for the MNIST dataset:

1. `x_train`: Images used for training the neural network
2. `y_train`: Correct labels for the `x_train` images, used to evaluate the model's predictions during training
3. `x_valid`: Images set aside for validating the performance of the model after it has been trained
4. `y_valid`: Correct labels for the `x_valid` images, used to evaluate the model's predictions after it has been trained

The process of preparing data for analysis is called Data Engineering. To learn more about the differences between training data and validation data (as well as test data), check out this article by Jason Brownlee.

1.2.2 Loading the Data Into Memory (with Keras)

There are many deep learning frameworks, each with their own merits. In this we will be working with PyTorch 2, and specifically with the Sequential API. The Sequential API has many useful built in functions designed for constructing neural networks. It is also a legitimate choice for deep learning in a professional setting due to its readability and efficiency, though it is not alone in this regard, and it is worth investigating a variety of frameworks.

We will also use the TorchVision library. One of the many helpful features that it provides are modules containing helper methods for many common datasets, including MNIST.

We will begin by loading both the `train` and `valid` datasets for MNIST.

```
train_set = torchvision.datasets.MNIST("./data/", train=True, download=True)
valid_set = torchvision.datasets.MNIST("./data/", train=False, download=True)
```

We stated above that the MNIST dataset contained 70,000 grayscale images of handwritten digits. By executing the following cells, we can see that TorchVision has partitioned 60,000 of these PIL Images for training, and 10,000 for validation (after training).

```
train_set
```

```
Dataset MNIST
    Number of datapoints: 60000
    Root location: ./data/
    Split: Train
```

```
valid_set
```

```
Dataset MNIST
    Number of datapoints: 10000
    Root location: ./data/
    Split: Test
```

Note: The `Split` for `valid_set` is stated as `Test`, but we will be using the data for validation in our exercises. To learn more about the difference between `Train`, `Valid`, and `Test` datasets, please view this article by Kili.

1.2.3 Exploring the MNIST Data

Let's take the first x, y pair from `train_set` and review the data structures:

```
x_0, y_0 = train_set[0]
```

```
x_0
```

```
type(x_0)
```

```
PIL.Image.Image
def __init__()

/usr/local/lib/python3.10/dist-packages/PIL/Image.py
This class represents an image object.  To create
:py:class:`~PIL.Image.Image` objects, use the appropriate factory
functions.  There's hardly ever any reason to call the Image constructor
directly.
```

Is this a 5 or a poorly written 3? We can view the corresponding label to be sure.

```
y_0
```

```
5
```

```
type(y_0)
```

```
int
```

1.3 Tensors

If a vector is a 1-dimensional array, and a matrix is a 2-dimensional array, a tensor is an n-dimensional array representing any number of dimensions. Most modern neural network frameworks are powerful tensor processing tools.

One example of a 3-dimensional tensor could be pixels on a computer screen. The different dimensions would be width, height, and color channel. Video games use matrix mathematics to calculate pixel values in a similar way to how neural networks calculate tensors. This is why GPUs are effective tensor processesing machines.

Let's convert our images into tensors so we can later process them with a neural network. TorchVision has a useful function to convert [PIL Images](#) into tensors with the [ToTensor](#) class:

```
trans = transforms.Compose([transforms.ToTensor()])
x_0_tensor = trans(x_0)
```

```
/usr/local/lib/python3.10/dist-packages/torchvision/transforms/v2/_deprecated.py:42: UserWarning: The transform `ToTensor()` is deprecat
  warnings.warn(
```

[PyTorch tensors](#) have a number of useful properies and methods. We can verify the data type:

```
x_0_tensor.dtype
```

```
torch.float32
```

We can verify the minimum and maximum values. PIL Images have a potential integer range of [0, 255], but the [ToTensor](#) class converts it to a float range of [0.0, 1.0].

```
x_0_tensor.min()
```

```
tensor(0.)
```

```
x_0_tensor.max()
```

```
tensor(1.)
```

We can also view the size of each dimension. PyTorch uses a `C x H x W` convention, which means the first dimension is color channel, the second is height, and the third is width.

Since these images are black and white, there is only 1 color channel. The images are square being 28 pixels tall and wide:

```
x_0_tensor.size()
```

```
torch.Size([1, 28, 28])
```

We can also look at the values directly:

```
x_0_tensor
```

```
          0.4235, 0.0039, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.3176, 0.9412, 0.9922,
          0.9922, 0.4667, 0.0980, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.1765, 0.7294,
          0.9922, 0.9922, 0.5882, 0.1059, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0627,
          0.3647, 0.9882, 0.9922, 0.7333, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.9765, 0.9922, 0.9765, 0.2510, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.1804, 0.5098,
          0.7176, 0.9922, 0.9922, 0.8118, 0.0078, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.1529, 0.5804, 0.8980, 0.9922,
          0.9922, 0.9922, 0.9804, 0.7137, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0941, 0.4471, 0.8667, 0.9922, 0.9922, 0.9922,
          0.9922, 0.7882, 0.3059, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0902, 0.2588, 0.8353, 0.9922, 0.9922, 0.9922, 0.9922, 0.7765,
          0.3176, 0.0078, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0706, 0.6706,
          0.8588, 0.9922, 0.9922, 0.9922, 0.9922, 0.7647, 0.3137, 0.0353,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.2157, 0.6745, 0.8863, 0.9922,
          0.9922, 0.9922, 0.9922, 0.9569, 0.5216, 0.0431, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.5333, 0.9922, 0.9922, 0.9922,
          0.8314, 0.5294, 0.5176, 0.0627, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000]]])
```

By default, a tensor is processed with a [CPU](https://en.wikipedia.org/wiki/Central_processing_unit).

```
x_0_tensor.device
```

```
device(type='cpu')
```

To move it to a GPU, we can use the `.cuda` method.

```
x_0_gpu = x_0_tensor.cuda()
x_0_gpu.device
```

```
device(type='cuda', index=0)
```

The `.cuda` method will fail if a GPU is not recognized by PyTorch. In order to make our code flexible, we can send our tensor `to` the `device` we identified at the start of this notebook. This way, our code will run much faster if a GPU is available, but the code will not break if there is no available GPU.

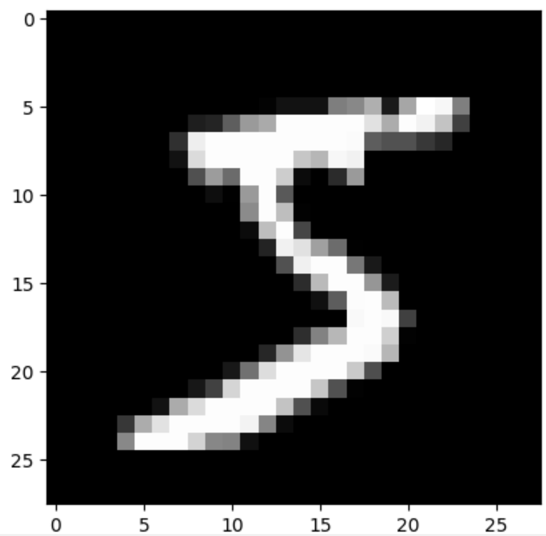
```
x_0_tensor.to(device).device
```

```
device(type='cuda', index=0)
```

Sometimes, it can be hard to interpret so many numbers. Thankfully, TorchVision can convert `C x H x W` tensors back into a PIL image with the [to_pil_image](https://pytorch.org/vision/stable/generated/torchvision.transforms.functional.to_pil_image.html) function.

```
image = F.to_pil_image(x_0_tensor)
plt.imshow(image, cmap='gray')
```

```
<matplotlib.image.AxesImage at 0x7937d0c82cb0>
```



1.4 Preparing the Data for Training

Earlier, we created a `trans` variable to convert an image to a tensor. [Transforms](https://pytorch.org/vision/stable/transforms.html) are a group of torchvision functions that can be used to transform a dataset.

1.4.1 Transforms

The [Compose](https://pytorch.org/vision/stable/generated/torchvision.transforms.Compose.html) fuction combines a list of transforms. We will learn more about transforms in a later notebook, but have copied the `trans` definition below as an introduction.

```
trans = transforms.Compose([transforms.ToTensor()])
```

Before, we only applied `trans` to one value. There are multiple ways we can apply our list of transforms to a dataset. One such way is to set it to a dataset's `transform` variable.

```
train_set.transform = trans
valid_set.transform = trans
```

1.4.2 DataLoaders

If our dataset is a deck of flash cards, a DataLoader defines how we pull cards from the deck to train an AI model. We could show our models the entire dataset at once. Not only does this take a lot of computational resources, but research shows using a smaller batch of data is more efficient for model training.

For example, if our `batch_size` is 32, we will train our model by shuffling the deck and drawing 32 cards. We do not need to shuffle for validation as the model is not learning, but we will still use a `batch_size` to prevent memory errors.

The batch size is something the model developer decides, and the best value will depend on the problem being solved. Research shows 32 or 64 is sufficient for many machine learning problems and is the default in some machine learning frameworks, so we will use 32 here.

```
batch_size = 32

train_loader = DataLoader(train_set, batch_size=batch_size, shuffle=True)
valid_loader = DataLoader(valid_set, batch_size=batch_size)
```

1.5 Creating the Model

It's time to build the model! Neural networks are composed of layers where each layer performs a mathematical operation on the data it receives before passing it to the next layer. To start, we will create a "Hello World" level model made from 4 components:

1. A Flatten used to convert n-dimensional data into a vector.
2. An input layer, the first layer of neurons
3. A hidden layer, another layor of neurons "hidden" between the input and output
4. An output layer, the last set of neurons which returns the final prediction from the model

More information about these layers is available in this blog post by Sarita.

Let's create a `layers` variable to hold our list of layers.

```
layers = []
layers
```

```
[]
```

1.5.1 Flattening the Image

When we looked at the shape of our data above, we saw the images had 3 dimensions: `C x H x W`. To flatten an image means to combine all of these images into 1 dimension. Let's say we have a tensor like the one below. Try running the code cell to see what it looks like before and after being flattened.

```
test_matrix = torch.tensor(
    [[1, 2, 3],
     [4, 5, 6],
     [7, 8, 9]]
)
test_matrix
```

```
tensor([[1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]])
```

```
nn.Flatten()(test_matrix)
```

```
tensor([[1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]])
```

Nothing happened? That's because neural networks expect to recieve a batch of data. Currently, the Flatten layer sees three vectors as opposed to one 2d matrix. To fix this, we can "batch" our data by adding an extra pair of brackets. Since `test_matrix` is now a tensor, we can do that with the shorthand below. `None` adds a new dimension where `:` selects all the data in a tensor.

```
batch_test_matrix = test_matrix[None, :]
batch_test_matrix
```

```
tensor([[[1, 2, 3],
         [4, 5, 6],
         [7, 8, 9]]])
```

```
nn.Flatten()(batch_test_matrix)
```

```
tensor([[1, 2, 3, 4, 5, 6, 7, 8, 9]])
```

Order matters! This is what happens when we do it the other way:

```
nn.Flatten()(test_matrix[:, None])
```

```
tensor([[1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]])
```

Now that we've gotten the hang of the `Flatten` layer, let's add it to our list of `layers`.

```
layers = [
    nn.Flatten()
]
layers
```

```
[Flatten(start_dim=1, end_dim=-1)]
```

1.5.2 The Input Layer

Our first layer of neurons connects our flattened image to the rest of our model. To do that, we will use a Linear layer. This layer will be *densely connected*, meaning that each neuron in it, and its weights, will affect every neuron in the next layer.

In order to create these weights, Pytorch needs to know the size of our inputs and how many neurons we want to create. Since we've flattened our images, the size of our inputs is the number of channels, number of pixels vertically, and number of pixels horizontally multiplied together.

```
input_size = 1 * 28 * 28
```

Choosing the correct number of neurons is what puts the "science" in "data science" as it is a matter of capturing the statistical complexity of the dataset. For now, we will use `512` neurons. Try playing around with this value later to see how it affects training and to start developing a sense for what this number means.

For now, we will use the relu activation function, which in short, will help our network to learn how to make more sophisticated guesses about data than if it were required to make guesses based on some strictly linear function.

```
layers = [
    nn.Flatten(),
    nn.Linear(input_size, 512),  # Input
    nn.ReLU(),  # Activation for input
]
layers
```

```
[Flatten(start_dim=1, end_dim=-1),
 Linear(in_features=784, out_features=512, bias=True),
 ReLU()]
```

1.5.3 The Hidden Layer

Now we will add an additional densely connected linear layer. We will cover why adding another set of neurons can help improve learning in the next lesson. Just like how the input layer needed to know the shape of the data that was being passed to it, a hidden layer's nn.Linear needs to know the shape of the data being passed to it. Each neuron in the previous layer will compute one number, so the number of inputs into the hidden layer is the same as the number of neurons in the previous later.

```
layers = [
    nn.Flatten(),
    nn.Linear(input_size, 512),  # Input
    nn.ReLU(),  # Activation for input
```

```
    nn.Linear(512, 512),  # Hidden
    nn.ReLU()  # Activation for hidden
]
layers
```

```
[Flatten(start_dim=1, end_dim=-1),
 Linear(in_features=784, out_features=512, bias=True),
 ReLU(),
 Linear(in_features=512, out_features=512, bias=True),
 ReLU()]
```

1.5.4 The Output Layer

Finally, we will add an output layer. In this case, since the network is to make a guess about a single image belonging to 1 of 10 possible categories, there will be 10 outputs. Each output is assigned a neuron. The larger the value of the output neuron compared to the other neurons, the more the model predicts the input image belongs to the output neuron's assigned class.

We will not assign the `relu` function to the output layer. Instead, we will apply a `loss function` covered in the next section.

```
n_classes = 10

layers = [
    nn.Flatten(),
    nn.Linear(input_size, 512),  # Input
    nn.ReLU(),  # Activation for input
    nn.Linear(512, 512),  # Hidden
    nn.ReLU(),  # Activation for hidden
    nn.Linear(512, n_classes)  # Output
]
layers
```

```
[Flatten(start_dim=1, end_dim=-1),
 Linear(in_features=784, out_features=512, bias=True),
 ReLU(),
 Linear(in_features=512, out_features=512, bias=True),
 ReLU(),
 Linear(in_features=512, out_features=10, bias=True)]
```

1.5.5 Compiling the Model

A [Sequential](#) model expects a sequence of arguments, not a list, so we can use the [* operator](#) to unpack our list of layers into a sequence. We can print the model to verify these layers loaded correctly.

```
model = nn.Sequential(*layers)
model
```

```
Sequential(
  (0): Flatten(start_dim=1, end_dim=-1)
  (1): Linear(in_features=784, out_features=512, bias=True)
  (2): ReLU()
  (3): Linear(in_features=512, out_features=512, bias=True)
  (4): ReLU()
  (5): Linear(in_features=512, out_features=10, bias=True)
)
```

Much like tensors, when the model is first initialized, it will be processed on a CPU. To have it process with a GPU, we can use `to(device)`.

```
model.to(device)
```

```
Sequential(
  (0): Flatten(start_dim=1, end_dim=-1)
  (1): Linear(in_features=784, out_features=512, bias=True)
  (2): ReLU()
  (3): Linear(in_features=512, out_features=512, bias=True)
  (4): ReLU()
  (5): Linear(in_features=512, out_features=10, bias=True)
)
```

To check which device a model is on, we can check which device the model parameters are on. Check out this stack overflow post for more information.

```
next(model.parameters()).device
```

```
device(type='cuda', index=0)
```

PyTorch 2.0 introduced the ability to compile a model for faster performance. Learn more about it here.

```
model = torch.compile(model)
```

1.6 Training the Model

Now that we have prepared training and validation data, and a model, it's time to train our model with our training data, and verify it with its validation data.

"Training a model with data" is often also called "fitting a model to data." Put another way, it highlights that the shape of the model changes over time to more accurately understand the data that it is being given.

1.6.1 Loss and Optimization

Just like how teachers grade students, we need to provide the model a function in which to grade its answers. This is called a `loss function`. We will use a type of loss function called CrossEntropy which is designed to grade if a model predicted the correct category from a group of categories.

```
loss_function = nn.CrossEntropyLoss()
```

Next, we select an `optimizer` for our model. If the `loss_function` provides a grade, the optimizer tells the model how to learn from this grade to do better next time.

```
optimizer = Adam(model.parameters())
```

1.6.2 Calculating Accuracy

While the results of the loss function are effective in helping our model learn, the values can be difficult to interpret for humans. This is why data scientists often include other metrics like accuracy.

In order to accurately calculate accuracy, we should compare the number of correct classifications compared to the total number of predictions made. Since we're showing data to the model in batches, our accuracy can be calculated along with these batches.

First, the total number of predictions is the same size as our dataset. Let's assign the size of our datasets to `N` where `n` is synonymous with the `batch size`.

```
train_N = len(train_loader.dataset)
valid_N = len(valid_loader.dataset)
```

Next, we'll make a function to calculate the accuracy for each batch. The result is a fraction of the total accuracy, so we can add the accuracy of each batch together to get the total.

```
def get_batch_accuracy(output, y, N):
    pred = output.argmax(dim=1, keepdim=True)
    correct = pred.eq(y.view_as(pred)).sum().item()
    return correct / N
```

1.6.3 The Train Function

Here is where everything comes together. Below is the function we've defined to train our model based on the training data. We will walk through each line of code in more detail later, but take a moment to review how it is structured. Can you recognize the variables we created earlier?

```
def train():
    loss = 0
    accuracy = 0

    model.train()
    for x, y in train_loader:
        x, y = x.to(device), y.to(device)
        output = model(x)
        optimizer.zero_grad()
        batch_loss = loss_function(output, y)
        batch_loss.backward()
        optimizer.step()

        loss += batch_loss.item()
        accuracy += get_batch_accuracy(output, y, train_N)
    print('Train - Loss: {:.4f} Accuracy: {:.4f}'.format(loss, accuracy))
```

1.6.4 The Validate Function

Similarly, this is the code for validating the model with data it did not train on. Can you spot some differences with the `train` function?

```
def validate():
    loss = 0
    accuracy = 0

    model.eval()
    with torch.no_grad():
        for x, y in valid_loader:
            x, y = x.to(device), y.to(device)
            output = model(x)

            loss += loss_function(output, y).item()
            accuracy += get_batch_accuracy(output, y, valid_N)
    print('Valid - Loss: {:.4f} Accuracy: {:.4f}'.format(loss, accuracy))
```

1.6.5 The Training Loop

To see how the model is progressing, we will alternated between training and validation. Just like how it might take a student a few times going through their deck of flash cards to learn all the concepts, the model will go through the training data multiple times to get a better and better understanding.

An `epoch` is one complete pass through the entire dataset. Let's train and validate the model for 20 `epochs` to see how it learns.

```
import time
import torch._dynamo
torch._dynamo.config.suppress_errors = True

# Measure runtime for the training loop
start_time = time.time()  # Start time
epochs = 10

for epoch in range(epochs):
    print('Epoch: {}'.format(epoch))
    train()
    validate()

# End time
end_time = time.time()
```

```
W1109 19:39:36.091000 20008 torch/_dynamo/convert_frame.py:1125]   File "/usr/local/lib/python3.10/dist-packages/torch/_dynamo/symbol
W1109 19:39:36.091000 20008 torch/_dynamo/convert_frame.py:1125]     super().run()
W1109 19:39:36.091000 20008 torch/_dynamo/convert_frame.py:1125]   File "/usr/local/lib/python3.10/dist-packages/torch/_dynamo/symbol
W1109 19:39:36.091000 20008 torch/_dynamo/convert_frame.py:1125]     while self.step():
W1109 19:39:36.091000 20008 torch/_dynamo/convert_frame.py:1125]   File "/usr/local/lib/python3.10/dist-packages/torch/_dynamo/symbol
W1109 19:39:36.091000 20008 torch/_dynamo/convert_frame.py:1125]     self.dispatch_table[inst.opcode](self, inst)
W1109 19:39:36.091000 20008 torch/_dynamo/convert_frame.py:1125]   File "/usr/local/lib/python3.10/dist-packages/torch/_dynamo/symbol
W1109 19:39:36.091000 20008 torch/_dynamo/convert_frame.py:1125]     self._return(inst)
W1109 19:39:36.091000 20008 torch/_dynamo/convert_frame.py:1125]   File "/usr/local/lib/python3.10/dist-packages/torch/_dynamo/symbol
W1109 19:39:36.091000 20008 torch/_dynamo/convert_frame.py:1125]     self.output.compile_subgraph(
W1109 19:39:36.091000 20008 torch/_dynamo/convert_frame.py:1125]   File "/usr/local/lib/python3.10/dist-packages/torch/_dynamo/output
W1109 19:39:36.091000 20008 torch/_dynamo/convert_frame.py:1125]     self.compile_and_call_fx_graph(tx, list(reversed(stack_values)),
W1109 19:39:36.091000 20008 torch/_dynamo/convert_frame.py:1125]   File "/usr/local/lib/python3.10/dist-packages/torch/_dynamo/output
W1109 19:39:36.091000 20008 torch/_dynamo/convert_frame.py:1125]     compiled_fn = self.call_user_compiler(gm)
W1109 19:39:36.091000 20008 torch/_dynamo/convert_frame.py:1125]   File "/usr/local/lib/python3.10/dist-packages/torch/_dynamo/output
W1109 19:39:36.091000 20008 torch/_dynamo/convert_frame.py:1125]     return self._call_user_compiler(gm)
W1109 19:39:36.091000 20008 torch/_dynamo/convert_frame.py:1125]   File "/usr/local/lib/python3.10/dist-packages/torch/_dynamo/output
W1109 19:39:36.091000 20008 torch/_dynamo/convert_frame.py:1125]     raise BackendCompilerFailed(self.compiler_fn, e) from e
W1109 19:39:36.091000 20008 torch/_dynamo/convert_frame.py:1125] torch._dynamo.exc.BackendCompilerFailed: backend='inductor' raised:
W1109 19:39:36.091000 20008 torch/_dynamo/convert_frame.py:1125] RuntimeError: Cannot find a working triton installation. Either the
W1109 19:39:36.091000 20008 torch/_dynamo/convert_frame.py:1125]
W1109 19:39:36.091000 20008 torch/_dynamo/convert_frame.py:1125] Set TORCH_LOGS="+dynamo" and TORCHDYNAMO_VERBOSE=1 for more informat
W1109 19:39:36.091000 20008 torch/_dynamo/convert_frame.py:1125]
Train - Loss: 379.6576 Accuracy: 0.9383
Valid - Loss: 30.7895 Accuracy: 0.9690
Epoch: 1
Train - Loss: 152.9670 Accuracy: 0.9751
Valid - Loss: 21.6914 Accuracy: 0.9787
Epoch: 2
Train - Loss: 106.3177 Accuracy: 0.9819
Valid - Loss: 24.5561 Accuracy: 0.9773
Epoch: 3
Train - Loss: 81.5764 Accuracy: 0.9868
Valid - Loss: 31.3437 Accuracy: 0.9729
Epoch: 4
Train - Loss: 66.9315 Accuracy: 0.9886
Valid - Loss: 24.4810 Accuracy: 0.9778
Epoch: 5
Train - Loss: 57.9145 Accuracy: 0.9902
Valid - Loss: 23.9061 Accuracy: 0.9810
Epoch: 6
Train - Loss: 45.0989 Accuracy: 0.9918
Valid - Loss: 28.9589 Accuracy: 0.9786
Epoch: 7
Train - Loss: 45.0430 Accuracy: 0.9924
Valid - Loss: 23.3764 Accuracy: 0.9831
Epoch: 8
Train - Loss: 38.3158 Accuracy: 0.9937
Valid - Loss: 28.7644 Accuracy: 0.9789
Epoch: 9
Train - Loss: 35.3888 Accuracy: 0.9944
Valid - Loss: 27.2757 Accuracy: 0.9809
```

```
print(f"Training completed in {end_time - start_time:.2f} seconds.")
```

```
Training completed in 140.86 seconds.
```

We're already close to 100%! Let's see if it's true by testing it on our original sample. We can use our model like a function:

```
prediction = model(x_0_gpu)
prediction
```

```
tensor([[-36.4821, -12.6168, -28.1037,   8.9082, -29.2411,  16.4251, -28.8105,
         -15.4103, -15.6650,  -4.2307]], device='cuda:0',
       grad_fn=<AddmmBackward0>)
```

There should be ten numbers, each corresponding to a different output neuron. Thanks to how the data is structured, the index of each number matches the corresponding handwritten number. The 0th index is a prediction for a handwritten 0, the 1st index is a prediction for a handwritten 1, and so on.

We can use the `argmax` function to find the index of the highest value.

```
prediction.argmax(dim=1, keepdim=True)
```

```
tensor([[5]], device='cuda:0')
```

Did it get it right?

```
y_0
```

```
5
```

1.7 Clear the Memory

```
import IPython
app = IPython.Application.instance()
app.kernel.do_shutdown(True)
```

```
{'status': 'ok', 'restart': True}
```