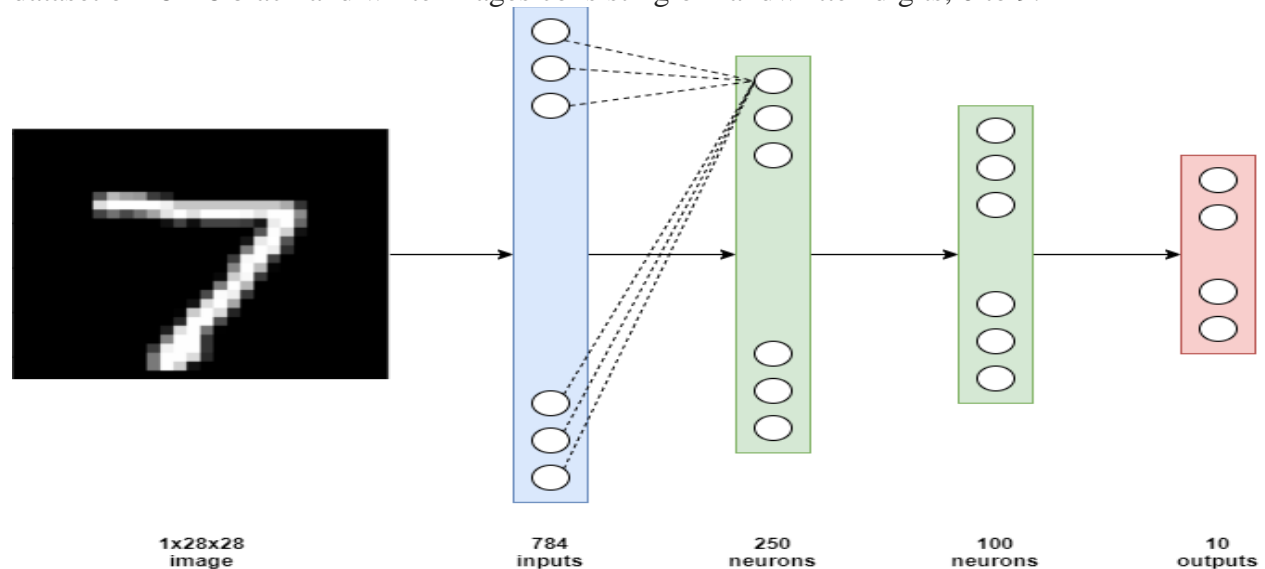


WEEK 5 : MULTILAYER PERCEPTRON ALGORITHM TO PERFORM IMAGE CLASSIFICATION

we'll start with one of the most basic neural network architectures, a multilayer perceptron (MLP), also known as a feedforward network. The dataset we'll be using is the famous MNIST dataset, a dataset of 28x28 black and white images consisting of handwritten digits, 0 to 9.



We'll process the dataset, build our model, and then train our model. Afterwards, we'll do a short dive into what the model has actually learned.

Data Processing

Let's start by importing all the modules we'll need. The main ones we need to import are:

- torch for general PyTorch functionality
- torch.nn and torch.nn.functional for neural network based functions
- torch.optim for our optimizer which will update the parameters of our neural network
- torch.utils.data for handling the dataset
- torchvision.transforms for data augmentation
- torchvision.datasets for loading the dataset
- sklearn's metrics for visualizing a confusion matrix
- sklearn's decomposition and manifold for visualizing the neural network's representations in two dimensions
- matplotlib for plotting

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torch.utils.data as data

import torchvision.transforms as transforms
import torchvision.datasets as datasets

from sklearn import metrics
from sklearn import decomposition
```

```
from sklearn import manifold
from tqdm.notebook import trange, tqdm
import matplotlib.pyplot as plt
import numpy as np

import copy
import random
import time
```

To ensure we get reproducible results we set the random seed for Python, Numpy and PyTorch.

```
SEED = 1234
random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)
torch.cuda.manual_seed(SEED)
torch.backends.cudnn.deterministic = True
```

The first thing we'll do is load our dataset.

This will automatically download the training set for the MNIST dataset and save it in a folder called .data. It will create the folder if it does not exist.

```
ROOT = '.data'
```

```
train_data = datasets.MNIST(root=ROOT,
                             train=True,
                             download=True)
```

Next, we want to *normalize* our data. This means we want it to have a mean of zero and a standard deviation of one.

Why do we want to do this? Normalizing our data allows our models to train faster and to also help them avoid local minima, i.e. train more reliably.

We normalize our data by subtracting the mean and dividing by the standard deviation of our dataset. First, we need to calculate the mean and standard deviation. **Note:** it is important that the mean and standard deviation are only calculated over the training set and not the test set. We do not want to use any information at all from the test set and should only look at it when we are calculating our test loss.

To calculate the means and standard deviations we get the actual data (the images) using the .data attribute of our training data, convert them into floating point numbers, and then use the built-in mean and std functions to calculate the mean and standard deviation, respectively. The image data has values between 0-255, which we want to scale between 0-1, so we divide by 255.

```
mean = train_data.data.float().mean() / 255
std = train_data.data.float().std() / 255
```

```
print(f'Calculated mean: {mean}')
```

```
print(f'Calculated std: {std}')
```

Calculated mean: 0.13066048920154572

Calculated std: 0.30810779333114624

Now we've calculated our mean and standard deviation how do we actually use them? We use Torchvision's transforms.

A transform states how our data should be augmented and processed. Data augmentation involves manipulating the available training data in a way that artificially creates more training examples. We use transforms.Compose to build a list of transformations that will be applied to the image.

The transforms we use are:

- RandomRotation - randomly rotates the image between (-x, +x) degrees, where we have set x = 5. Note, the fill=(0,) is due to a [bug](#) in some versions of torchvision.
- RandomCrop - this first adds padding around our image, 2 pixels here, to artificially make it bigger, before taking a random 28x28 square crop of the image.
- ToTensor() - this converts the image from a PIL image into a PyTorch tensor.
- Normalize - this subtracts the mean and divides by the standard deviations given.

The first two transformations have to be applied before ToTensor as they should both be applied on a PIL image. Normalize should only be applied to the images after they have been converted into a tensor. See the Torchvision documentation for transforms that should be applied to PIL images and transforms that should be applied on tensors.

We have two lists of transforms, a train and a test transform. The train transforms are to artificially create more examples for our model to train on. We do not augment our test data in the same way, as we want a consistent set of examples to evaluate our final model on. The test data, however, should still be normalized.

```
train_transforms = transforms.Compose([
    transforms.RandomRotation(5, fill=(0,)),
    transforms.RandomCrop(28, padding=2),
    transforms.ToTensor(),
    transforms.Normalize(mean=[mean], std=[std])
])
```

```
test_transforms = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=[mean], std=[std])
])
```

Now we have defined our transforms we can then load the train and test data with the relevant transforms defined.

```
train_data = datasets.MNIST(root=ROOT,
    train=True,
    download=True,
    transform=train_transforms)
```

```
test_data = datasets.MNIST(root=ROOT,
                           train=False,
                           download=True,
                           transform=test_transforms)
```

We can simply check the len of the datasets to see how many examples are within each.

```
print(f'Number of training examples: {len(train_data)}')
print(f'Number of testing examples: {len(test_data)}')
```

Number of training examples: 60000
Number of testing examples: 10000

We can get a look at some of the images within our dataset to see what we're working with. The function below plots a square grid of images. If you supply less than a complete square number of images it will ignore the last few.

```
def plot_images(images):

    n_images = len(images)

    rows = int(np.sqrt(n_images))
    cols = int(np.sqrt(n_images))

    fig = plt.figure()
    for i in range(rows*cols):
        ax = fig.add_subplot(rows, cols, i+1)
        ax.imshow(images[i].view(28, 28).cpu().numpy(), cmap='bone')
        ax.axis('off')
```

Let's load 25 images. These will have been processed through our transforms, so will be randomly rotated and cropped.

It's a good practice to see your data with your transforms applied, so you can ensure they look sensible. For example, it wouldn't make sense to flip the digits horizontally or vertically unless you are expecting to see what in your test data.

```
N_IMAGES = 25
images = [image for image, label in [train_data[i] for i in range(N_IMAGES)]]
plot_images(images)
```

The MNIST dataset comes with a training and test set, but not a validation set. We want to use a validation set to check how well our model performs on unseen data. Why don't we just use the test data? We should only be measuring our performance over the test set once, after all training is done. We can think of the validation set as a proxy test set we are allowed to look at as much as we want.

Furthermore, we create a validation set, taking 10% of the training set. **Note:** the validation set should always be created from the training set. Never take the validation set from the test set. When

researchers publish research papers they should be comparing performance across the test set and the only way to ensure this is a fair comparison is for all researchers to use the same test set. If the validation set is taken from the test set, then the test set is not the same as everyone else's and the results cannot be compared against each other.

First, we have to define the exact number of examples that we want to be in each split of the training/validation sets.

```
VALID_RATIO = 0.9
```

```
n_train_examples = int(len(train_data) * VALID_RATIO)
n_valid_examples = len(train_data) - n_train_examples
```

Then, we use the `random_split` function to take a random 10% of the training set to use as a validation set. The remaining 90% will stay as the training set.

```
train_data, valid_data = data.random_split(train_data,
                                           [n_train_examples, n_valid_examples])
```

We can print out the number of examples again to check our splits are correct.

```
print(f'Number of training examples: {len(train_data)}')
print(f'Number of validation examples: {len(valid_data)}')
print(f'Number of testing examples: {len(test_data)}')
```

Number of training examples: 54000

Number of validation examples: 6000

Number of testing examples: 10000

One thing to consider is that as the validation set has been created from the training set it has the same transforms as the training set, with the random rotating and cropping. As we want our validation set to act as a proxy for the test set, it should also be fixed, without any random augmentation.

First, let's see what 25 of the images within the validation set look like with the training transforms:

```
N_IMAGES = 25
images = [image for image, label in [valid_data[i] for i in range(N_IMAGES)]]
plot_images(images)
```

We can now simply replace the validation set's transform by overwriting it with our test transforms from above.

As the validation set is a Subset of the training set, if we change the transforms of one, then by default Torchvision will change the transforms of the other. To stop this from happening, we make a deepcopy of the validation data.

```
valid_data = copy.deepcopy(valid_data)
valid_data.dataset.transform = test_transforms
```

To double check we've correctly replaced the training transforms, we can view the same set of images and notice how they're more central (no random cropping) and have a more standard orientation (no random rotations).

```
N_IMAGES = 25
images = [image for image, label in [valid_data[i] for i in range(N_IMAGES)]]
plot_images(images)
```

Next, we'll define a DataLoader for each of the training/validation/test sets. We can iterate over these, and they will yield batches of images and labels which we can use to train our model.

We only need to shuffle our training set as it will be used for stochastic gradient descent, and we want each batch to be different between epochs. As we aren't using the validation or test sets to update our model parameters, they do not need to be shuffled.

Ideally, we want to use the biggest batch size that we can. The 64 here is relatively small and can be increased if our hardware can handle it.

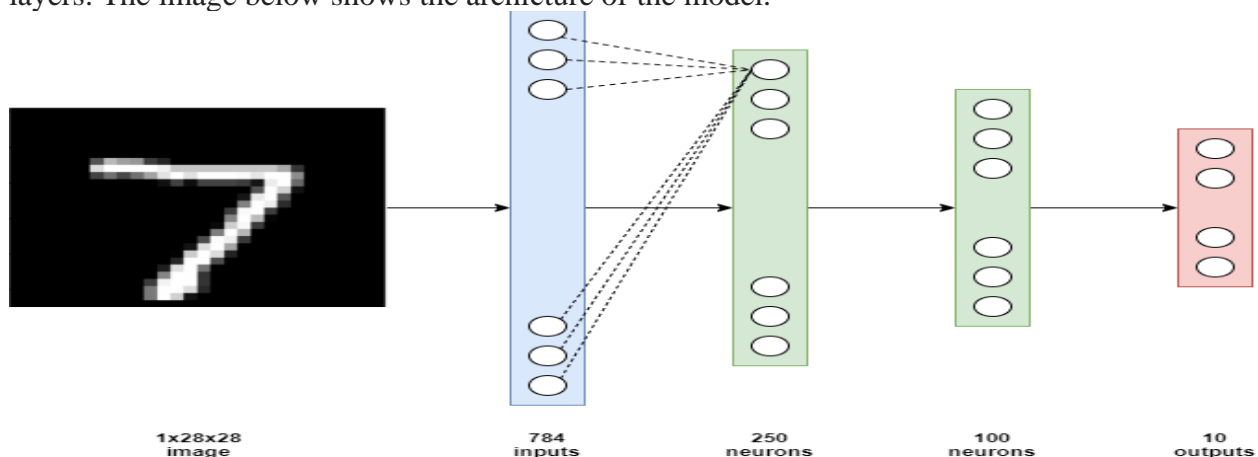
```
BATCH_SIZE = 64
train_iterator = data.DataLoader(train_data,
                                shuffle=True,
                                batch_size=BATCH_SIZE)

valid_iterator = data.DataLoader(valid_data,
                                batch_size=BATCH_SIZE)

test_iterator = data.DataLoader(test_data,
                                batch_size=BATCH_SIZE)
```

Defining the Model

Our model will be a neural network, specifically a multilayer perceptron (MLP) with two hidden layers. The image below shows the architecture of the model.



Specifically, first we will flatten our 1x28x28 (1 color channel, 28 pixels height and width) image into a 784 element vector, also called 784 *features*. We flatten our input, as MLPs cannot handle two or three-dimensional data. Next, the 784 dimensional input is passed through the first hidden layer to transform it into 250 dimensions. Then, another hidden layer, which will transform it to

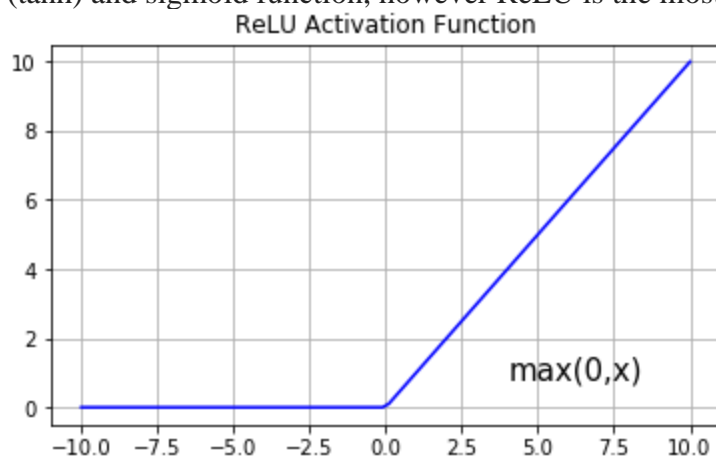
100 dimensions. Finally, an output layer which will transform it into a 10 dimensional vector. The output dimension should equal the number of classes within your data. Here we have ten digits, 0 - 9, so need our output to be 10 dimensions.

The transformation between 784 to 250, 250 to 100 and 100 to 10 dimensions are done by Linear layers. These are also known as fully connected or affine layers. In these layers, every element in one layer is connected to every element in the next. We can think of these elements as *neurons*, as this architecture is inspired by how the human brain is made of millions of interconnected nodes, also called neurons.

Each connection between a neuron in one layer and a neuron in the next has a *weight* associated with it. The input to one neuron is the sum of the weighted values of all neurons in the previous layer connected to it, plus a weighted bias term, where the bias value is always 1. The neuron then applies an *activation function* to this weighted sum. This activation function is a non-linear function that allows the neural network to learn non-linear functions between inputs and outputs.

We define our MLP below, which consists of three linear layers. We first take the input batch of images and flatten them, so they can be passed into the linear layers. We then pass them through the first linear layer, `input_fc`, which calculates the weighted sum of the inputs, and then apply the *ReLU* (rectified linear unit) activation function elementwise. This result is then passed through another linear layer, `hidden_fc`, again applying the same activation function elementwise. Finally, we pass this through the final linear layer, `output_fc`. We return not only the output but also the second hidden layer as we will do some analysis on it later.

The ReLU activation function is a popular non-linear function that is simply $\max(0, x)$, where x is the weighted sum of the inputs to that neuron. Other activation functions used are hyperbolic tan (\tanh) and sigmoid function, however ReLU is the most commonly used.



One thing to note is that we do not use an activation function on the input directly or on the output. You should never use activation functions directly on the input, i.e. `F.relu(x)`. PyTorch combines activation functions to be applied on the output with the functions which calculate the *loss*, also known as *error* or *cost*, of a neural network. This is done for numerical stability.

Why did we choose hidden dimensions of 250 and 100 elements? Why did we only have two hidden layers? There is no magic formula to tell us how many layers to use and how many neurons to have in each layer, and there is most probably a better set of values. However, the general idea is that neural networks extract features from data. Layers closer to the input learn to extract general features (e.g. lines, curves, edges), whilst later layers combine the features extracted from the previous layer into more high level features (e.g. the intersection of two lines making a cross, multiple curves make a circle). We force our neural network to learn these features by reducing

the number of neurons in each layer. This way, it has to learn to compress information by extracting only the useful and general features. Thus, we want a neural network with multiple layers and some sort of information compression (reduced number of neurons in subsequent layers).

```
class MLP(nn.Module):
    def __init__(self, input_dim, output_dim):
        super().__init__()

        self.input_fc = nn.Linear(input_dim, 250)
        self.hidden_fc = nn.Linear(250, 100)
        self.output_fc = nn.Linear(100, output_dim)

    def forward(self, x):

        # x = [batch size, height, width]

        batch_size = x.shape[0]

        x = x.view(batch_size, -1)

        # x = [batch size, height * width]

        h_1 = F.relu(self.input_fc(x))

        # h_1 = [batch size, 250]

        h_2 = F.relu(self.hidden_fc(h_1))

        # h_2 = [batch size, 100]

        y_pred = self.output_fc(h_2)

        # y_pred = [batch size, output dim]

        return y_pred, h_2
```

We'll define our model by creating an instance of it and setting the correct input and output dimensions.

```
INPUT_DIM = 28 * 28
OUTPUT_DIM = 10

model = MLP(INPUT_DIM, OUTPUT_DIM)
```

We can also create a small function to calculate the number of trainable parameters (weights and biases) in our model - in case all of our parameters are trainable.

```
def count_parameters(model):  
    return sum(p.numel() for p in model.parameters() if p.requires_grad)
```

The first layer has 784 neurons connected to 250 neurons, so $784 \cdot 250$ weighted connections plus 250 bias terms.

The second layer has 250 neurons connected to 100 neurons, $250 \cdot 100$ weighted connections plus 100 bias terms.

The third layer has 100 neurons connected to 10 neurons, $100 \cdot 10$ weighted connections plus 10 bias terms.

$784 \cdot 250 + 250 + 250 \cdot 100 + 100 + 100 \cdot 10 + 10 = 222,360$

```
[ ]  
print(f'The model has {count_parameters(model):,} trainable parameters')  
The model has 222,360 trainable parameters
```

Training the Model

Next, we'll define our optimizer. This is the algorithm we will use to update the parameters of our model with respect to the loss calculated on the data.

We aren't going to go into too much detail on how neural networks are trained (see [this](#) article if you want to know how) but the gist is:

- pass a batch of data through your model
- calculate the loss of your batch by comparing your model's predictions against the actual labels
- calculate the gradient of each of your parameters with respect to the loss
- update each of your parameters by subtracting their gradient multiplied by a small *learning rate* parameter

We use the *Adam* algorithm with the default parameters to update our model. Improved results could be obtained by searching over different optimizers and learning rates, however default Adam is usually a good starting off point. Check out [this](#) article if you want to learn more about the different optimization algorithms commonly used for neural networks.

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

Then, we define a *criterion*, PyTorch's name for a loss/cost/error function. This function will take in your model's predictions with the actual labels and then compute the loss/cost/error of your model with its current parameters.

CrossEntropyLoss both computes the *softmax* activation function on the supplied predictions as well as the actual loss via *negative log likelihood*.

Briefly, the softmax function is:

$\text{softmax}(x) = \frac{e^{x_i}}{\sum_j e^{x_j}}$

This turns out 10 dimensional output, where each element is an unbounded real number, into a probability distribution over 10 elements. That is, all values are between 0 and 1, and together they all sum to 1.

Why do we turn things into a probability distribution? So we can use negative log likelihood for our loss function, as it expects probabilities. PyTorch calculates negative log likelihood for a single example via:

negative log likelihood (y^{\wedge}, y) = $-\log(\text{softmax}(y^{\wedge})[y])$

y^{\wedge} is the R10 output, from our neural network, whereas y is the label, an integer representing the class. The loss is the negative log of the class index of the softmax. For example:

$y^{\wedge} = [5, 1, 1, 1, 1, 1, 1, 1, 1, 1]$

$\text{softmax}(y^{\wedge}) = [0.8585, 0.0157, 0.0157, 0.0157, 0.0157, 0.0157, 0.0157, 0.0157, 0.0157, 0.0157]$

If the label was class zero, the loss would be:

negative log likelihood ($y^{\wedge}, 0$) = $-\log(0.8585) = 0.153\dots$

If the label was class five, the loss would be:

negative log likelihood ($y^{\wedge}, 5$) = $-\log(0.0157) = 4.154\dots$

So, intuitively, as your model's output corresponding to the correct class index increases, your loss decreases.

```
criterion = nn.CrossEntropyLoss()
```

We then define device. This is used to place your model and data on to a GPU, if you have one.

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

We place our model and criterion on to the device by using the .to method.

```
model = model.to(device)
criterion = criterion.to(device)
```

Next, we'll define a function to calculate the accuracy of our model. This takes the index of the highest value for your prediction and compares it against the actual class label. We then divide how many our model got correct by the amount in the batch to calculate accuracy across the batch.

```
def calculate_accuracy(y_pred, y):
    top_pred = y_pred.argmax(1, keepdim=True)
    correct = top_pred.eq(y.view_as(top_pred)).sum()
    acc = correct.float() / y.shape[0]
    return acc
```

We finally define our training loop.

This will:

- put our model into train mode
- iterate over our dataloader, returning batches of (image, label)
- place the batch on to our GPU, if we have one
- clear the gradients calculated from the last batch
- pass our batch of images, x , through to model to get predictions, y_{pred}
- calculate the loss between our predictions and the actual labels
- calculate the accuracy between our predictions and the actual labels
- calculate the gradients of each parameter

- update the parameters by taking an optimizer step
- update our metrics

Some layers act differently when training and evaluating the model that contains them, hence why we must tell our model we are in "training" mode. The model we are using here does not use any of those layers, however it is good practice to get used to putting your model in training mode.

```
def train(model, iterator, optimizer, criterion, device):

    epoch_loss = 0
    epoch_acc = 0

    model.train()

    for (x, y) in tqdm(iterator, desc="Training", leave=False):

        x = x.to(device)
        y = y.to(device)

        optimizer.zero_grad()

        y_pred, _ = model(x)

        loss = criterion(y_pred, y)

        acc = calculate_accuracy(y_pred, y)

        loss.backward()

        optimizer.step()

        epoch_loss += loss.item()
        epoch_acc += acc.item()

    return epoch_loss / len(iterator), epoch_acc / len(iterator)
```

The evaluation loop is similar to the training loop. The differences are:

- we put our model into evaluation mode with model.eval()
- we wrap the iterations inside a with torch.no_grad()
- we do not zero gradients as we are not calculating any
- we do not calculate gradients as we are not updating parameters
- we do not take an optimizer step as we are not calculating gradients

torch.no_grad() ensures that gradients are not calculated for whatever is inside the with block. As our model will not have to calculate gradients, it will be faster and use less memory.

```
def evaluate(model, iterator, criterion, device):
```

```
epoch_loss = 0
epoch_acc = 0

model.eval()

with torch.no_grad():

    for (x, y) in tqdm(iterator, desc="Evaluating", leave=False):

        x = x.to(device)
        y = y.to(device)

        y_pred, _ = model(x)

        loss = criterion(y_pred, y)

        acc = calculate_accuracy(y_pred, y)

        epoch_loss += loss.item()
        epoch_acc += acc.item()

    return epoch_loss / len(iterator), epoch_acc / len(iterator)
```

The final step before training is to define a small function to tell us how long an epoch took.

```
def epoch_time(start_time, end_time):
    elapsed_time = end_time - start_time
    elapsed_mins = int(elapsed_time / 60)
    elapsed_secs = int(elapsed_time - (elapsed_mins * 60))
    return elapsed_mins, elapsed_secs
```

We're finally ready to train!

During each epoch we calculate the training loss and accuracy, followed by the validation loss and accuracy. We then check if the validation loss achieved is the best validation loss we have seen. If so, we save our model's parameters (called a state_dict).

```
EPOCHS = 10
```

```
best_valid_loss = float('inf')
```

```
for epoch in range(EPOCHS):
```

```
    start_time = time.monotonic()
```

```
    train_loss, train_acc = train(model, train_iterator, optimizer, criterion, device)
```

```
    valid_loss, valid_acc = evaluate(model, valid_iterator, criterion, device)
```

```

if valid_loss < best_valid_loss:
    best_valid_loss = valid_loss
    torch.save(model.state_dict(), 'tut1-model.pt')

end_time = time.monotonic()

epoch_mins, epoch_secs = epoch_time(start_time, end_time)

print(f'Epoch: {epoch+1:02} | Epoch Time: {epoch_mins}m {epoch_secs}s')
print(f'\tTrain Loss: {train_loss:.3f} | Train Acc: {train_acc*100:.2f}%')
print(f'\tVal. Loss: {valid_loss:.3f} | Val. Acc: {valid_acc*100:.2f}%')
Epoch: 01 | Epoch Time: 0m 10s
    Train Loss: 0.425 | Train Acc: 86.83%
    Val. Loss: 0.194 | Val. Acc: 93.94%
Epoch: 02 | Epoch Time: 0m 9s
    Train Loss: 0.172 | Train Acc: 94.70%
    Val. Loss: 0.119 | Val. Acc: 96.23%
Epoch: 03 | Epoch Time: 0m 9s
    Train Loss: 0.139 | Train Acc: 95.69%
    Val. Loss: 0.117 | Val. Acc: 96.30%
Epoch: 04 | Epoch Time: 0m 9s
    Train Loss: 0.118 | Train Acc: 96.33%
    Val. Loss: 0.100 | Val. Acc: 96.87%
Epoch: 05 | Epoch Time: 0m 9s
    Train Loss: 0.108 | Train Acc: 96.69%
    Val. Loss: 0.092 | Val. Acc: 97.17%
Epoch: 06 | Epoch Time: 0m 9s
    Train Loss: 0.096 | Train Acc: 96.90%
    Val. Loss: 0.085 | Val. Acc: 97.29%
Epoch: 07 | Epoch Time: 0m 9s
    Train Loss: 0.093 | Train Acc: 97.07%
    Val. Loss: 0.084 | Val. Acc: 97.51%
Epoch: 08 | Epoch Time: 0m 9s
    Train Loss: 0.087 | Train Acc: 97.30%
    Val. Loss: 0.083 | Val. Acc: 97.58%
Epoch: 09 | Epoch Time: 0m 9s
    Train Loss: 0.082 | Train Acc: 97.43%
    Val. Loss: 0.059 | Val. Acc: 98.15%
Epoch: 10 | Epoch Time: 0m 10s
    Train Loss: 0.079 | Train Acc: 97.54%
    Val. Loss: 0.060 | Val. Acc: 98.20%

```

Afterwards, we load our the parameters of the model that achieved the best validation loss and then use this to evaluate our model on the test set.

```
model.load_state_dict(torch.load('tut1-model.pt'))
```

```
test_loss, test_acc = evaluate(model, test_iterator, criterion, device)
```

Our model achieves 98% accuracy on the test set.

This can be improved by tweaking hyperparameters, e.g. number of layers, number of neurons per layer, optimization algorithm used, learning rate, etc.

```
print(f'Test Loss: {test_loss:.3f} | Test Acc: {test_acc*100:.2f}%')
```

Test Loss: 0.052 | Test Acc: 98.37%

Examining the Model

Now we've trained our model, there are a few things we can look at. Most of these are simple exploratory analysis, but they can offer some insights into your model.

An important thing to do is check what examples your model gets wrong and ensure that they're reasonable mistakes.

The function below will return the model's predictions over a given dataset. It will return the inputs (image) the outputs (model predictions) and the ground truth labels.

```
def get_predictions(model, iterator, device):
```

```
    model.eval()
```

```
    images = []
```

```
    labels = []
```

```
    probs = []
```

```
    with torch.no_grad():
```

```
        for (x, y) in iterator:
```

```
            x = x.to(device)
```

```
            y_pred, _ = model(x)
```

```
            y_prob = F.softmax(y_pred, dim=-1)
```

```
            images.append(x.cpu())
```

```
            labels.append(y.cpu())
```

```
            probs.append(y_prob.cpu())
```

```
    images = torch.cat(images, dim=0)
```

```
    labels = torch.cat(labels, dim=0)
```

```
    probs = torch.cat(probs, dim=0)
```

```
    return images, labels, probs
```

We can then get these predictions and, by taking the index of the highest predicted probability, get the predicted labels.

```
images, labels, probs = get_predictions(model, test_iterator, device)

pred_labels = torch.argmax(probs, 1)
```

Then, we can make a confusion matrix from our actual labels and our predicted labels.

```
def plot_confusion_matrix(labels, pred_labels):

    fig = plt.figure(figsize=(10, 10))
    ax = fig.add_subplot(1, 1, 1)
    cm = metrics.confusion_matrix(labels, pred_labels)
    cm = metrics.ConfusionMatrixDisplay(cm, display_labels=range(10))
    cm.plot(values_format='d', cmap='Blues', ax=ax)
```

The results seem reasonable enough, the most confused predictions-actuals are: 3-5 and 2-7.

```
plot_confusion_matrix(labels, pred_labels)
```

Next, for each of our examples, we can check if our predicted label matches our actual label.

```
corrects = torch.eq(labels, pred_labels)
```

We can then loop through all of the examples over our model's predictions and store all the examples the model got incorrect into an array.

Then, we sort these incorrect examples by how confident they were, with the most confident being first.

```
incorrect_examples = []

for image, label, prob, correct in zip(images, labels, probs, corrects):
    if not correct:
        incorrect_examples.append((image, label, prob))

incorrect_examples.sort(reverse=True,
                        key=lambda x: torch.max(x[2], dim=0).values)
```

We can then plot the incorrectly predicted images along with how confident they were on the actual label and how confident they were at the incorrect label.

```
def plot_most_incorrect(incorrect, n_images):

    rows = int(np.sqrt(n_images))
```

```

cols = int(np.sqrt(n_images))

fig = plt.figure(figsize=(20, 10))
for i in range(rows*cols):
    ax = fig.add_subplot(rows, cols, i+1)
    image, true_label, probs = incorrect[i]
    true_prob = probs[true_label]
    incorrect_prob, incorrect_label = torch.max(probs, dim=0)
    ax.imshow(image.view(28, 28).cpu().numpy(), cmap='bone')
    ax.set_title(f'true label: {true_label} ({true_prob:.3f})\n'
                f'pred label: {incorrect_label} ({incorrect_prob:.3f})')
    ax.axis('off')
fig.subplots_adjust(hspace=0.5)

```

Below we can see the 25 images the model got incorrect and was most confident about. A lot of these digits are irregular, so it's difficult for the model to do well on these. The images that do look fine, if you squint you can sort of see why the model got it wrong. Why is the neural network so confident on the irregular digits? Surely if it's a weird looking digit then the output of the softmax should be close to evenly distributed across a few digits the model isn't sure about, right? Well, no. The model has been trained to only be incredibly confident about its predictions, and thus when it sees an image it will always be confident about what it is.

```
N_IMAGES = 25
```

```
plot_most_incorrect(incorrect_examples, N_IMAGES)
```

Another thing we can do is get the output and intermediate representations from the model and try to visualize them.

The function below loops through the provided dataset and gets the output from the model and the intermediate representation from the layer before that, the second hidden layer.

```
def get_representations(model, iterator, device):
```

```
    model.eval()
```

```
    outputs = []
    intermediates = []
    labels = []
```

```
    with torch.no_grad():
```

```
        for (x, y) in tqdm(iterator):
```

```
            x = x.to(device)
```

```
            y_pred, h = model(x)
```



```
    outputs.append(y_pred.cpu())
    intermediates.append(h.cpu())
    labels.append(y)

outputs = torch.cat(outputs, dim=0)
intermediates = torch.cat(intermediates, dim=0)
labels = torch.cat(labels, dim=0)

return outputs, intermediates, labels
```

We run the function to get the representations.

```
outputs, intermediates, labels = get_representations(model,
                                                    train_iterator,
                                                    device)
```

The data we want to visualize is in ten dimensions and 100 dimensions. We want to get this down to two dimensions, so we can actually plot it.

The first technique we'll use is PCA (principal component analysis). First, we'll define a function to calculate the PCA of our data, and then we'll define a function to plot it.

```
def get_pca(data, n_components=2):
    pca = decomposition.PCA()
    pca.n_components = n_components
    pca_data = pca.fit_transform(data)
    return pca_data

def plot_representations(data, labels, n_images=None):
    if n_images is not None:
        data = data[:n_images]
        labels = labels[:n_images]
    fig = plt.figure(figsize=(10, 10))
    ax = fig.add_subplot(111)
    scatter = ax.scatter(data[:, 0], data[:, 1], c=labels, cmap='tab10')
    handles, labels = scatter.legend_elements()
    ax.legend(handles=handles, labels=labels)
```

First, we plot the representations from the ten dimensional output layer, reduced down to two dimensions.

```
output_pca_data = get_pca(outputs)
plot_representations(output_pca_data, labels)
```

Next, we'll plot the outputs of the second hidden layer.

The clusters seem similar to the one above. In fact if we rotated the below image anti-clockwise it wouldn't be too far off the PCA of the output representations.

```
intermediate_pca_data = get_pca(intermediates)
plot_representations(intermediate_pca_data, labels)
```

An alternative to PCA is t-SNE (t-distributed stochastic neighbor embedding). This is commonly thought of as being "better" than PCA, although it can be [misinterpreted](#).

```
def get_tsne(data, n_components=2, n_images=None):
    if n_images is not None:
        data = data[:n_images]
    tsne = manifold.TSNE(n_components=n_components, random_state=0)
    tsne_data = tsne.fit_transform(data)
    return tsne_data
```

t-SNE is very slow, so we only compute it on a subset of the representations. The classes look very well separated, and it is possible to use [k-NN](#) on this representation to achieve decent accuracy.

```
N_IMAGES = 5_000

output_tsne_data = get_tsne(outputs, n_images=N_IMAGES)
plot_representations(output_tsne_data, labels, n_images=N_IMAGES)
```

We plot the intermediate representations on the same subset. Again, the classes look well separated, though less so than the output representations. This is because these representations are intermediate features that the neural network has extracted and will use them in further layers to weigh up the evidence of what digit is in the image. Hence, in theory, the classes should become more separated the closer we are to the output layer, which is exactly what we see here.

```
intermediate_tsne_data = get_tsne(intermediates, n_images=N_IMAGES)
plot_representations(intermediate_tsne_data, labels, n_images=N_IMAGES)
```

Another experiment we can do is try and generate fake digits. The function below will repeatedly generate random noise and feed it through the model and find the most confidently generated digit for the desired class.

```
def imagine_digit(model, digit, device, n_iterations=50_000):

    model.eval()

    best_prob = 0
    best_image = None
```

```
with torch.no_grad():  
  
    for _ in range(n_iterations):  
  
        x = torch.randn(32, 28, 28).to(device)  
  
        y_pred, _ = model(x)  
  
        preds = F.softmax(y_pred, dim=-1)  
  
        _best_prob, index = torch.max(preds[:, digit], dim=0)  
  
        if _best_prob > best_prob:  
            best_prob = _best_prob  
            best_image = x[index]  
  
    return best_image, best_prob
```

Let's try and generate a perfect three.

DIGIT = 3

```
best_image, best_prob = imagine_digit(model, DIGIT, device)
```

Looking at the best probability achieved, we have a digit that the model is 100% confident is a three.

```
print(f'Best image probability: {best_prob.item()*100:.2f}%')  
Best image probability: 99.98%
```

Unfortunately, the imagined perfect three just looks like random noise.

As mentioned before, the model has only been trained to be incredibly confident with its predictions, so when faced with random noise will try and classify it as something.

It is also possible that the model is *overfitting* on the training set - that there is a common pattern in handwritten threes within the training set, but it's not the pattern we want our model to learn.

```
plt.imshow(best_image.cpu().numpy(), cmap='bone')  
plt.axis('off');
```

Finally, we can plot the weights in the first layer of our model.

The hope is that there's maybe one neuron in this first layer that's learned to look for certain patterns in the input and thus has high weight values indicating this pattern. If we then plot these weights, we should see these patterns.

```
def plot_weights(weights, n_weights):
```

```
rows = int(np.sqrt(n_weights))
cols = int(np.sqrt(n_weights))

fig = plt.figure(figsize=(20, 10))
for i in range(rows*cols):
    ax = fig.add_subplot(rows, cols, i+1)
    ax.imshow(weights[i].view(28, 28).cpu().numpy(), cmap='bone')
    ax.axis('off')
```

Looking at these weights we see a few of them look like random noise but some of them do have weird patterns within them. These patterns show "ghostly" image looking shapes, but are clearly not images.

N_WEIGHTS = 25

```
weights = model.input_fc.weight.data
```

```
plot_weights(weights, N_WEIGHTS)
```

Conclusions

we have shown:

- loading Torchvision datasets
- loading transforms to augment and normalize our data
- defining a MLP
- training a model to achieve >97% accuracy
- viewing our model's mistakes
- visualizing our data in lower dimensions with PCA and t-SNE
- generating fake digits
- viewing the learned weights of our model