Tutorial 2 - Multi-Class ANNs

Project Motivation

In the lectures we have seen how Multi-Class Artificial Neural Networks (ANNs) can be trained and tested to classify handwritten digits. To truly appreciate this capability we will see if we can apply our ANN model to work with real images.

For example given a blank white sheet of paper with a hand written digit on it, how could we use an ANN model to correctly identify the digit. That will be our objective in this tutorial and along the way you will learn a few tricks to speed the training of your models.

MNIST Multi-Class Classification

To begin, let us load the MNIST dataset and divide it for training and validation. Note that we have left some of the 60,000 samples for final testing of the model.

In [6]:
```
# obtain data
from torchvision import datasets, transforms

mnist_data = datasets.MNIST('data', train=True, download=True, transform=transforms.ToTensor())
mnist_data = list(mnist_data)
mnist_train = mnist_data[:4096]
mnist_val   = mnist_data[4096:5120]
```

Multi-Class ANN Architecture

In this example we will be using a 3-layer ANN with ReLU activation functions applied on the first and second hidden layers. The softmax activation will be used for outputting class probabilities and is not included in the architecture setup.

In [7]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F

import matplotlib.pyplot as plt # for plotting
import torch.optim as optim #for gradient descent

torch.manual_seed(1) # set the random seed

class MNISTClassifier(nn.Module):
    def __init__(self):
        super(MNISTClassifier, self).__init__()
        self.layer1 = nn.Linear(784, 50)
        self.layer2 = nn.Linear(50, 20)
        self.layer3 = nn.Linear(20, 10)
    def forward(self, img):
        flattened = img.view(-1, 28 * 28)
        activation1 = F.relu(self.layer1(flattened))
        activation2 = F.relu(self.layer2(activation1))
        output = self.layer3(activation2)
        return output

model = MNISTClassifier()

print('done')
```

done

Function to Obtain Accuracy

The get_accuracy function is used to compute the accuracy on training or validation data.

In [8]:
```
def get_accuracy(model, train=False):
    if train:
        data = mnist_train
    else:
        data = mnist_val

    correct = 0
    total = 0
    for imgs, labels in torch.utils.data.DataLoader(data, batch_size=64):
        output = model(imgs)
        #select index with maximum prediction score
        pred = output.max(1, keepdim=True)[1]
        correct += pred.eq(labels.view_as(pred)).sum().item()
        total += imgs.shape[0]
    return correct / total

print ('done')
```

done

Function to perform Training and Validation

The train function puts everything together. You can provide arguments to adjust the batch size and number of training epochs.

In [9]:
```
def train(model, data, batch_size=64, num_epochs=1 , print_stat = 1):
    train_loader = torch.utils.data.DataLoader(data, batch_size=batch_size)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

    iters, losses, train_acc, val_acc = [], [], [], []

    # training
    n = 0 # the number of iterations
    for epoch in range(num_epochs):
        for imgs, labels in iter(train_loader):
            out = model(imgs)             # forward pass
            loss = criterion(out, labels) # compute the total loss
            loss.backward()               # backward pass (compute parameter updates)
            optimizer.step()              # make the updates for each parameter
            optimizer.zero_grad()         # a clean up step for PyTorch

            # save the current training information
            iters.append(n)
            losses.append(float(loss)/batch_size)             # compute *average* loss
            train_acc.append(get_accuracy(model, train=True)) # compute training accuracy
            val_acc.append(get_accuracy(model, train=False))  # compute validation accuracy
            n += 1

    if print_stat:
      # plotting
      plt.title("Training Curve")
      plt.plot(iters, losses, label="Train")
      plt.xlabel("Iterations")
      plt.ylabel("Loss")
      plt.show()

      plt.title("Training Curve")
      plt.plot(iters, train_acc, label="Train")
      plt.plot(iters, val_acc, label="Validation")
      plt.xlabel("Iterations")
      plt.ylabel("Training Accuracy")
      plt.legend(loc='best')
      plt.show()

      print("Final Training Accuracy: {}".format(train_acc[-1]))
      print("Final Validation Accuracy: {}".format(val_acc[-1]))


print('done')
```

done

(optional) Sanity Check

Verify that the model is able to overfit on a single batch of data.

In [10]:
```
#overfitting the model (sanity check)
debug_data = mnist_train[:64]
model = MNISTClassifier()
train(model, debug_data, num_epochs=50, print_stat=0)
```

In [11]:
```
#obtain accuracy on 64 samples
correct = 0
total = 0
for imgs, labels in torch.utils.data.DataLoader(debug_data, batch_size=64):
    output = model(imgs)
    #select index with maximum prediction score
    pred = output.max(1, keepdim=True)[1]
    correct += pred.eq(labels.view_as(pred)).sum().item()
    total += imgs.shape[0]
print('Accuracy on batch of 64: ', correct / total)
```

```
Accuracy on batch of 64:  0.34375
```

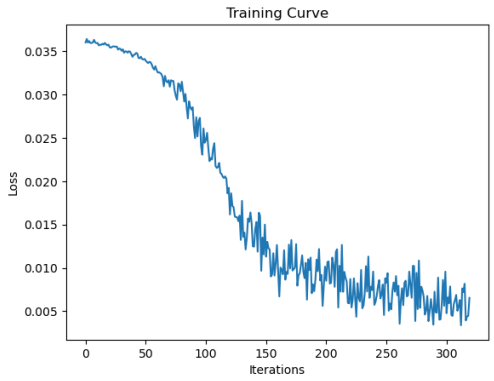
If we set `print_stat=1`, note the Final Training and Validation accuracy is obtained on the full training data and validation data. It does not reflect the performance on the 64 samples that were overfit.

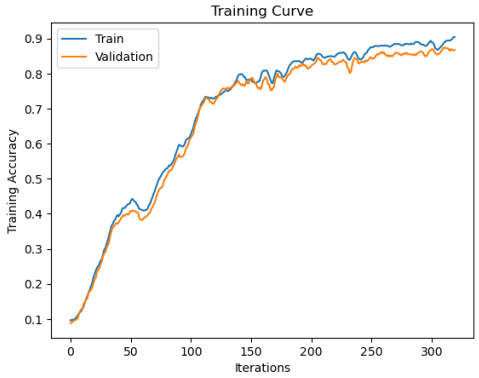
Run Training and Validation

Now that we've validated that our model can overfit a relatively small amount of training data (i.e. 64 samples), we can proceed to train our model on all of the training data.

We will be training our model over 5 epochs (how many training iterations is that?) to ensure that we can complete this tutorial in a reasonable time. In your free time you are welcome to explore the model accuracy as you increase the number of epochs.

In [12]:
```
#proper model
model = MNISTClassifier()
train(model, mnist_train, num_epochs=5)
```






```
Final Training Accuracy: 0.904296875
Final Validation Accuracy: 0.8681640625
```

(optional) Additional Training

At this stage we can consider adjusting our model architecture:

- number of hidden layers,
- hidden units,
- activation functions,

- optimizers,
- learning rate,
- momentum,
- batch size,
- training iterations,

to evaluate the performance of our ANN model. Can we do better?

Tip: Once you have searched through the hyperparameters and found model parameters that work reasonably well. You may want to save your model so that you don't have to retrain the model next time you open the Google Colab file.

In [13]:
```
# save the model for next time
torch.save(model.state_dict(), "saved_model")
```

In [14]:
```
#Load a saved model
model = MNISTClassifier() # step 1: Initialize the model
model.load_state_dict(torch.load("saved_model")) # Step 2: Load the trained parameters
```

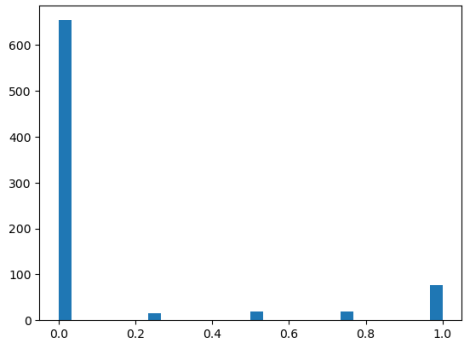
Out[14]:
```
<All keys matched successfully>
```

Test one image

At this point we have trained our model and observed accuracy scores on the training data and validation data. We haven't really looked at the data. For the next stage of the tutorial we will try to understand what the data looks like and consider what is required to classify new images obtained from the internet, or even our cell phone camera.

In [15]:
```
#Laod new image for testing
mnist_sample = mnist_data[19120] #samples with indices > 5120 can be used for testing
img, label = list(mnist_sample) #obtain a single image and label

#plot sample image
print('image dimensions: ', img.shape)
plt.imshow(img.view(-1,28),cmap='Greys') #make image 28 x 28 (not 1 x 28 x 28 as required by model)

#test new image
out = model(img)
prob = F.softmax(out, dim=1)
print('output dimensions: ', out.shape)
print('output probabilities: ', prob, 'sum: ', torch.sum(prob))

#print max index and compare with label
print('output: ', prob.max(1, keepdim=True)[1].item(), 'with a probability of', prob.max(1, keepdim=T
print('label: ', label)
```

```
image dimensions:  torch.Size([1, 28, 28])
output dimensions:  torch.Size([1, 10])
output probabilities:  tensor([[1.2843e-04, 1.3608e-06, 9.3711e-04, 3.1670e-04, 6.8963e-02, 3.8652e-
03,
         9.5960e-05, 1.1483e-01, 6.8463e-03, 8.0402e-01]],
       grad_fn=<SoftmaxBackward0>) sum:  tensor(1., grad_fn=<SumBackward0>)
output:  9 with a probability of 0.8040158748626709
label:  9
```

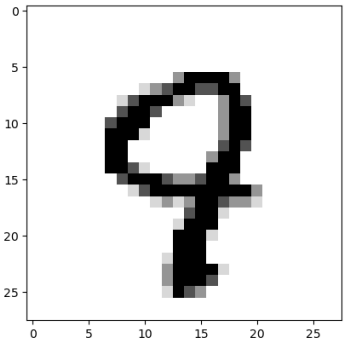


Now that we know a little bit about our data we should be able to generate new samples of the data.

Trying a new sample

In [19]:
```
import numpy as np
import matplotlib.pyplot as plt

#Load an image with black and white matching the MNIST data
img_new = plt.imread('https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcT5ZR8ImWkYVd2FRMZgUvCdNkH
print('Image Dimensions', img_new.shape)

def rgb2gray(rgb):
    return np.dot(rgb[...,:3], [0.299, 0.587, 0.144])

img_gray = rgb2gray(img_new)

plt.title("New Image")
plt.imshow(img_gray ,  cmap='Greys')
plt.show()

# compare to original MNIST image
plt.title("Original MNIST")
plt.imshow(img.view(-1,28) ,  cmap='Greys')
```

```
C:\Users\Rafiu Hossain\AppData\Local\Temp\ipykernel_7440\3958815606.py:5: MatplotlibDeprecationWarni
ng: Directly reading images from URLs is deprecated since 3.4 and will no longer be supported two mi
nor releases later. Please open the URL for reading and pass the result to Pillow, e.g. with ``np.ar
ray(PIL.Image.open(urllib.request.urlopen(url)))``.
  img_new = plt.imread('https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcT5ZR8ImWkYVd2FRMZgUvCd
NkHx0uKjjSAtTEJ0U-x0SPWQFxqnbg')
Image Dimensions (248, 203, 4)
```



Exploring the MNIST data

Before we can load new data for testing we should understand what preprocessing went into making the training data. We will explore:

- Data Type
- Data Dimensions
- Data Normalization
- Orientation

In [16]:
```
# data type
print(img.dtype)
```

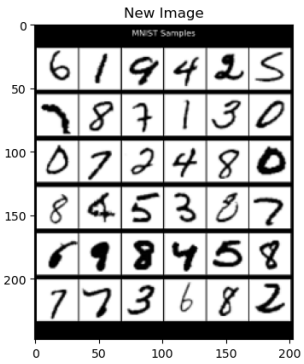
```
torch.float32
```

In [17]:
```
# data dimensions
print(img.shape)
```

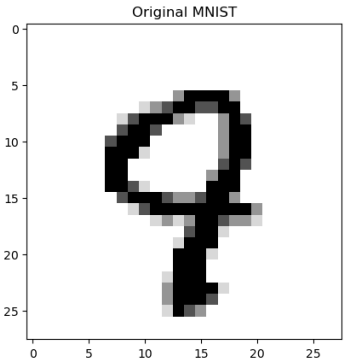
```
torch.Size([1, 28, 28])
```

In [18]:
```
# max and min values
print('min val:', torch.min(img).item(),  '  max val:', torch.max(img).item())

#histogram of values
plt.hist(img.view(-1,28*28), bins=30)
plt.show()
```

```
min val: 0.0   max val: 1.0
```



Out[19]:
```
<matplotlib.image.AxesImage at 0x25bdda1b940>
```

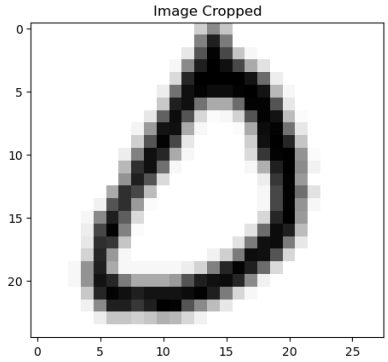


Cropping the Image

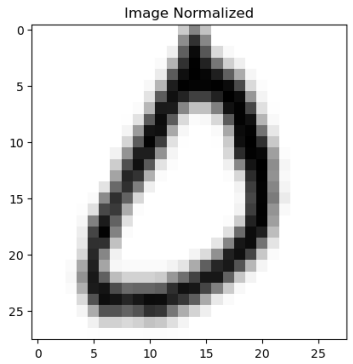
The images used to train our model were centered on the handwritten digit and resized to 28 x 28 pixesl. We will need to do the same to our new data in order for it work with our Multi-Class ANN model.

In [20]:
```
#cropping
img_cropped = img_gray[95:120,5:33]
plt.title("Image Cropped")
plt.imshow(img_cropped, cmap='Greys')
plt.show()
```

```
#resize image
from skimage.transform import rescale, resize, downscale_local_mean
img_resized = resize(img_cropped, (28,28), anti_aliasing=True)

#plot resized image
plt.title("Image Resized")
plt.imshow(img_resized , cmap='Greys')
plt.show()

#image max and min values
print(np.amax(img_resized))
print(np.amin(img_resized))

#normalize to range 0 to 1
img_resized = img_resized / np.amax(img_resized)
plt.title("Image Normalized")
plt.imshow(img_resized, cmap='Greys')
plt.show()

#verify max and min values
print(np.amax(img_resized))
print(np.amin(img_resized))
```




```
1.0299999999999998
0.0
```



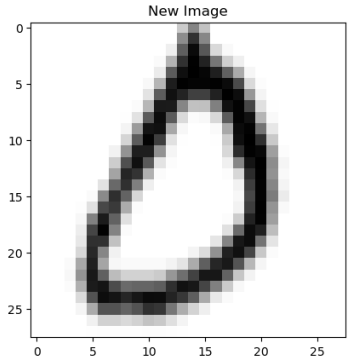

```
1.0
0.0
```

If required, how could you invert the colours?

Testing a New External Image

In [21]:
```
#test new external image

#plot resized image
plt.title("New Image")
```

```
plt.imshow(img_resized, cmap='Greys')
plt.show()

#convert image to torch tensor
img_new = torch.tensor(img_resized)
print('Initial Dimensions: ', img_new.shape)

#make our image match the model dimensions 1 x 28 x 28 and tensor type
img_new = img_new.unsqueeze(0).type(torch.FloatTensor)
print('Updated Dimensions: ', img_new.shape)

#perform forward pass on ANN model and generate an output
out = model(img_new)
prob = F.softmax(out, dim=1)

#examine output properties
print('output dimensions: ', out.shape)
print('output probabilities: ', prob, 'sum: ', torch.sum(prob))

#print max index
print('Predicted Output: ', prob.max(1, keepdim=True)[1].item(), 'with a probability of', prob.max(1,
```




```
Initial Dimensions:  torch.Size([28, 28])
Updated Dimensions:  torch.Size([1, 28, 28])
output dimensions:  torch.Size([1, 10])
output probabilities:  tensor([[8.5894e-01, 3.2328e-07, 5.6255e-03, 3.8166e-02, 1.3929e-07, 9.3705e-
02,
         3.8546e-06, 3.2770e-03, 1.7372e-04, 1.0509e-04]],
       grad_fn=<SoftmaxBackward0>) sum:  tensor(1.0000, grad_fn=<SumBackward0>)
Predicted Output:  0 with a probability of 0.8589437007904053
```

How did our Multi-Class ANN model perform? Was it successful on a new image?

As a little exercise try to load new images and see if the model can classify them correctly. To start you can modify the code above to crop another portion of the image. Once you are comfortable with that, you can look for new images online, or make your own.

Congratulations! You have just completed a project for handwritten digit recognition

Tutorial Challenges

There are just some questions for you to do on your own time.

Classifications

Obtain all classifications on the validation dataset

In [22]:
```
#write code to obtain all predictions on the validation data set
```

Tutorial Challenge 1: Incorrect Classifications

Go through all the predictions made on the validation dataset. How many digits were incorrectly classified? Are there any images that the ANN should have been able to classify correctly? View the images that were misclassified to speculate why they were misclassified. For example, you may find some **qualitative results** such as 2's are often mistaken as 3's or 1's are mistaken as 7's. Are there any samples that you would find difficult to classify?

In [23]:
```
#write code to visualize some of the incorrectly classified images
```

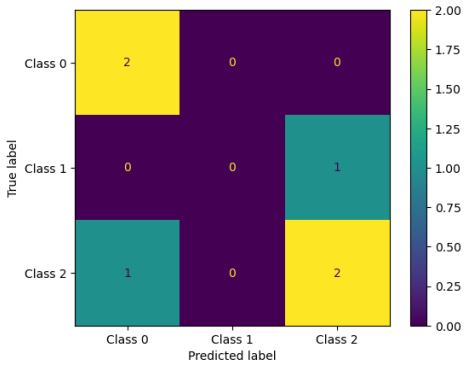
Confusion Matrix

The confusion matrix provide a nice table to visualize the classification performance of your model. Provided below is an example on a 6 sample toy dataset. In the final output the diagonal represents how many samples were correctly classified. For example: if we examine the third row of the result we will see that:

- 1 sample was **incorrectly** classified as class 0
- 0 samples were **incorrectly** classified as class 1
- 2 samples were **correctly** classified as class 2

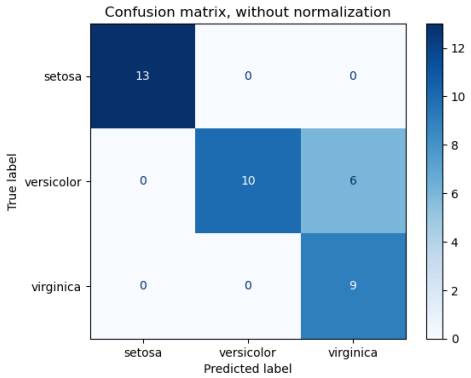
In [24]:
```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

y_true = [2, 0, 2, 2, 0, 1]
y_pred = [0, 0, 2, 2, 0, 2]

cm = confusion_matrix(y_true, y_pred)
print("Confusion Matrix:")
print (cm)

disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=["Class 0", "Class 1", "Class 2"])
disp.plot()
plt.show()
```

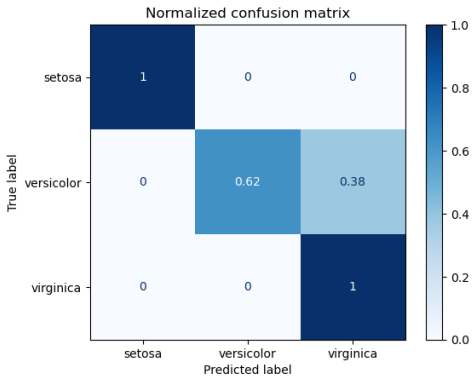
```
Confusion Matrix:
[[2 0 0]
 [0 0 1]
 [1 0 2]]
```



In [25]:
```
#Example from: https://scikit-learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.ht
import numpy as np
import matplotlib.pyplot as plt

from sklearn import svm, datasets
from sklearn.model_selection import train_test_split
from sklearn.metrics import ConfusionMatrixDisplay

# import some data to play with
iris = datasets.load_iris()
X = iris.data
y = iris.target
class_names = iris.target_names

# Split the data into a training set and a test set
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

# Run classifier, using a model that is too regularized (C too low) to see
# the impact on the results
classifier = svm.SVC(kernel="linear", C=0.01).fit(X_train, y_train)

np.set_printoptions(precision=2)

# Plot non-normalized confusion matrix
titles_options = [
    ("Confusion matrix, without normalization", None),
    ("Normalized confusion matrix", "true"),
]
for title, normalize in titles_options:
    disp = ConfusionMatrixDisplay.from_estimator(
        classifier,
        X_test,
        y_test,
        display_labels=class_names,
        cmap=plt.cm.Blues,
        normalize=normalize,
    )
    disp.ax_.set_title(title)
    print(title)
    print(disp.confusion_matrix)

plt.show()
```

```
Confusion matrix, without normalization
[[13  0  0]
 [ 0 10  6]
 [ 0  0  9]]
Normalized confusion matrix
[[1.   0.   0.  ]
 [0.   0.62 0.38]
 [0.   0.   1.  ]]
```





Tutorial Challenge 2: Confusion Matrix

Continuing with our Multi-Class ANN problem. Obtain all the model predictions along with the labels (ground truths) and feed them into a confusion matrix. What insights can you obtain about the performance of your model? Are there certain digits that your model is better able to classify?

In [26]:
```
#Write code to obtain a confusion matrix of our multi-class ANN model
```