Performance Comparison of Neural Networks using Keras and PyTorch

CS/EE 147: GPU Computing and Programming Final Project

Team Name:
2EEs_1CS_404_NOT_FOUND!

Team Members:
Estevan Valencia Rivera (861211373)
Yash Patel (SID: 861215968)
Venkat Prathipati (861209938)

Table of Contents:

Project Overview

A graphics processing unit (GPU) is a specialized electronic hardware designed for specialized display functions. Due to their architecture, they have seen usage beyond that. A GPU performs parallel operations and this makes them the perfect hardware for Image Processing, Data analysis, Machine Learning and Neural Networks (NN). The objective of this project was to implement and evaluate the performance of neural networks using two different frameworks: Keras and PyTorch. We extended the dataset implementation in Keras by using our own dataset to recognize faces.

GPU Acceleration

GPU-accelerated computing is the employment of a graphics processing unit (GPU) along with a computer processing unit (CPU) in order to facilitate processing-intensive operations such as deep learning, analytics and engineering applications. In this project the Facial Recognition pipeline uses algorithms and techniques such as the parallel algorithm to normalize data and improve our recognition capabilities. Parallel algorithm can be executed a piece at a time on many different processing devices, and then is combined together again to get the correct result.

Keras uses cuDNN is an NVIDIA library designed specifically to speed up the training aspect of neural networks using a GPU. In PyTorch, the model gets moved onto the GPU and just that can result in a performance gain but in the future we would imagine that moving the data (images) onto the GPU will also result in increased performance. Since NNs are heavily image based and have many number of nodes, we can intuitively state that a GPU will be much better performance than a CPU due to the ability of parallelization of the network.

Implementation Details

Data Collection for Facial Recognition:

Before training our neural networks to detect faces using either framework, we decided to create our own test data to train neural networks. In order to get great training data, we decided to implement the OpenCV library implementation for facial detection. OpenCV is an open source computer vision and machine learning software library. The library has algorithms to detect and recognize faces, identify objects, camera movements and etc.

We used the library to capture images of ourselves through the computers built-in camera. Using color detector to convert image to gray and then resize the image to fit just the

face of the person in the image. We then align the faces in the images by using the Facial Aligner library. Once the program was ready, we took 200 training images for each person and around 10 test images. All the images only have the face in the image with some features such as hair and ears cropped out. The image to the left was the original, and it outputs two separate images(to the right). These training and test images were used for steps to follow.

Neural Networks Using Keras:

Keras is a high level API that is written in Python. Keras can use various backends like Tensor Flow, CNTK, and Theano. For our project we used TensorFlow because it was familiar to us since we had a lecture on it. Another benefit of Keras is that it allows us to easily select between the CPU or GPU. Keras was installed in one of our home desktops with a GTX 1060 GPU.

For any NN you need a dataset to train it. While we were getting familiar with neural networks we used the MNIST Handwritten Digit Dataset. This data set has 60,000 28x28 images that are used to train the NN. It also has 10,000 images to test the NN. For the NN to classify the Handwritten Digits we used a 3 Layer NN. The first layer does not really count as a layer as it only flattens the matrix into a vector. The first real layer is a Dense layer with a size of 128 and an activation function of Rectified Linear (RELU). The second layer is exactly like the first layer a Dense layer with a size of 128 and an activation function of RELU. The final layer is another Dense layer with a size of 10, it is 10 because there are 10 possible digits that could be written. This layer uses a softmax activation function to finally classify the digit.

Dataset	CPU Runtime Avg.	GPU Runtime Avg.
MNIST Digits	3.5 s	5.5 s
MNIST Fashion	4.5 s	6.5 s

Figure 1: Computation Time for 3 layer NN

So we test the above NN on two datasets and on both the CPU was able to train the NN about 1 second faster per epoch than the GPU. We then Created a NN that did not make the accuracy better but it increases the computation needed to be done to train the NN. We added 10 Dense layers each with a size of 1024. This was only done to compare the computation time between the CPU and GPU.

Dataset	CPU Runtime Avg.	GPU Runtime Avg.
MNIST Digits	312 s	31 s
MNIST Fashion	320 s	31 s

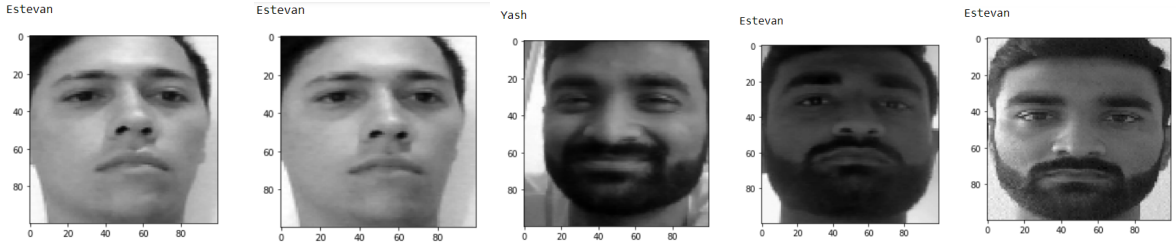
Figure 2: Computation Time for 10 layer NN

As seen in Figure 2, when the computation for the NN gets extremely large like in more complex NN the GPU is much more efficient in training the NN.

Convolutional Neural Networks Using Keras:

We then used Keras again to create facial reconition NN that was able to distinguish between two people. We created our own dataset as described above. The NN consisted of two Convolutional Layer both using the Relu activation function. After each convolutional layer we down sample the image using a Max Pooling layer. We then Flatten the data and pass it through two Dense layer to classify it. It is also passed through a Dropout layer to get rid of some of the feature to prevent over fitting. We did not compare the performance between the GPU and the CPU because it was a relatively small NN in which the CPU performed a little better. The GPU trained each epoch in about 3 seconds, this was also due to the fact that our face data set was pretty small.Only 200 pictures of each person.

We tested the NN with 32 images and of the 32 images the NL classified two images incorrectly. This means that our NN has about a 94% accuracy. There is a possibility of over training but we tested with images taken in different conditions as the training data.



Neural Networks Using PyTorch:

PyTorch is a machine learning library for Python used for a variety of applications such as artificial intelligence, natural language processing and neural networks. PyTorch provides two main high level features: tensor computing with GPU acceleration and construction of neural networks using automatic differentiation. For this project, PyTorch was used to construct a multi-layered Convolutional Neural Network (CNN) and it was trained on a GPU and a CPU to observe performance differences between the two. The CPU used for testing was an Intel i5-6600K and the GPU used was a NVIDIA GeForce GTX 1060 6GB.

To begin creating the model, the first step is to import the following libraries: pytorch, torchvision, and numpy. To keep testing consistent, the same dataset was utilized in both Keras and PyTorch based models. The Fashion Modified National Institute of Standards and Technology (MNIST) dataset created by Zalando Research was used on this CNN. It contains 60,000 28x28 grayscale images for training and 10,000 28x28 grayscale images for testing. Figure **N** shows images from the Fashion MNIST dataset. PyTorch is often installed with a few datasets preloaded and so the next step is to import the test and the training datasets into the workspace. A sample of the dataset is shown in Figure N.

```
grid = torchvision.utils.make_grid(images, nrow=10)`
plt.figure(figsize = (15,15))
plt.imshow(np.transpose(grid,(1,2,0)))
print('labels:',labels)
```

```
labels: tensor([9, 0, 0, 3, 0, 2, 7, 2, 5, 5])
```

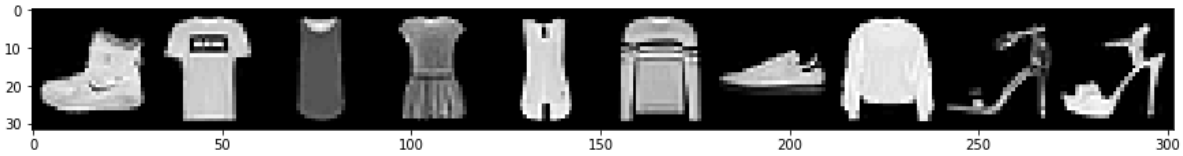


Figure 3: Fashion MNIST Dataset Sample

After importing the libraries and the datasets, the following hyperparameters are initialized: epochs, classes, batch size and learning rate. The importance and description of each can be found in the code snippet provided in the Documentation portion of the report. The design of the model for this project is an extremely common one for this dataset. Figure 4 shows the structure and layers of the CNN, it contains a total of 6 layers: 2 convolutional layers, 2 Max Pooling layers, 1 fully connected layer and 1input layer. The next step in the creation of the network is the loss function that helps a CNN prediction increase through each iteration. For our case, we used the cross-entropy loss function, the graph of which is shown in Figure 5.

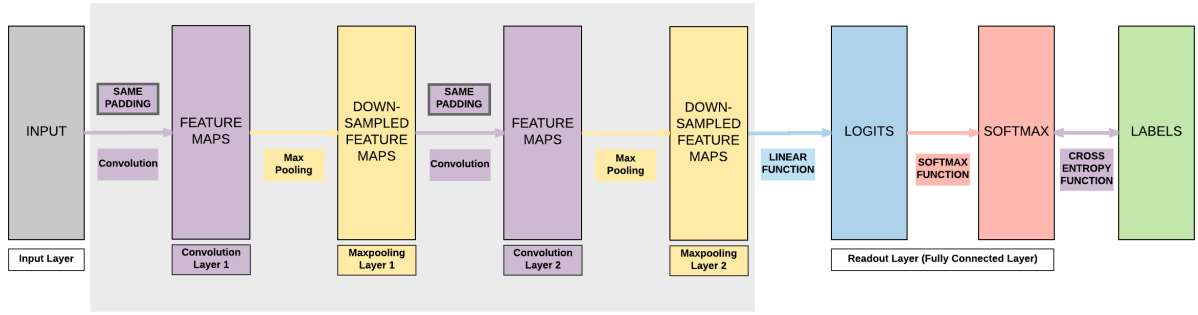


Figure 4: Block Model of CNN

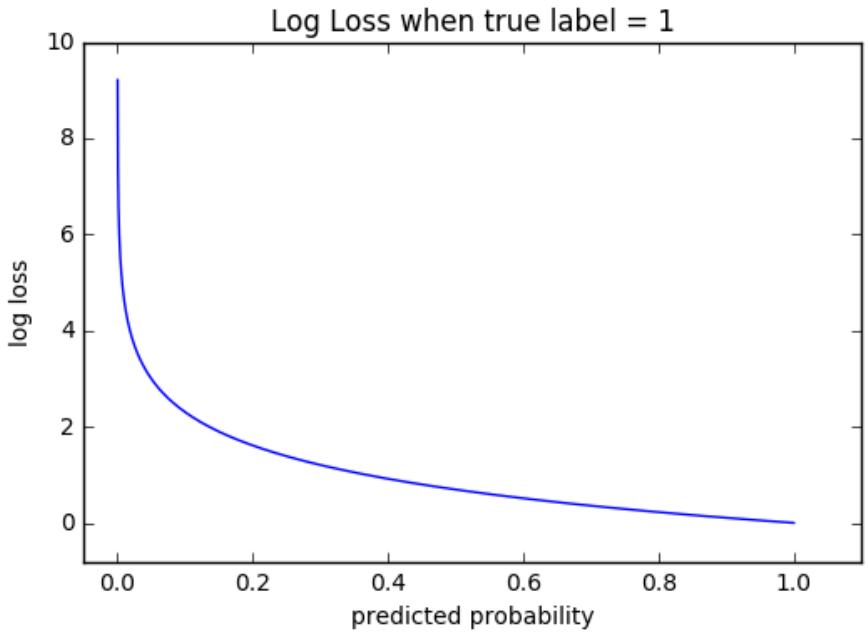


Figure 5: Cross-Entropy Loss Function

The model has been created and now it needs to be initialized and trained. The initialization is extremely easy and can be seen in the code snippet. Training of a CNN is basically performing forward propagation, calculating the loss and then performing backward propagation to use the gradients to update the parameters. After training the model, it was tested against the testing dataset. The final changes to the code were for the syntax when changing the execution device from the CPU to the GPU. Table 6 exhibits the performance increase in not only the training of the network but also the testing of it.

	CPU	**GPU**
Training Time	486.193 s	80.487 s
Model Accuracy	85.9%	88.62%
Testing Time	4.004s	1.169s

Table 6: GPU vs CPU performance of CNN using PyTorch

The results show around 6x time reduction when using a GPU for training a CNN compared to a CPU. We also see a time reduction when using a GPU for testing the CNN. That may prove useful in real world applications like autonomous vehicles where the input is essentially the real world images and having a reduced time would benefit the car in making quicker decisions. We see an increase in model accuracy for the GPU but after running the

scripts several times, it is not definitive if the GPU has an advantage over the CPU. The results we got could be attributed to the random seed for the initialization of the CNN.

Documentation on code execution

Facial Detection using OpenCV(Training Data/Test Data):

```python
import cv2, sys, os, dlib, imutils
from imutils.face_utils import FaceAligner
from imutils.face_utils import rect_to_bb

# Detects face
detector = dlib.get_frontal_face_detector()
# Predicts shape of face and features
predictor = dlib.shape_predictor("shape_predictor_68_face_landmarks.dat")
# Set to align face at the mouth
fa = FaceAligner(predictor, desiredFaceWidth=128)
# Classify face using Haar features
face_cascade = cv2.CascadeClassifier('haarcascade_frontalface_default.xml')
video_capture = cv2.VideoCapture(0)     # Capture Video
save = "C:/Users/venka/picResults/"     # Save image to
i = 0

while True:
    retval, frame = video_capture.read() # Capture frame by frame
    cv2.imshow('Video', frame)           # Show image
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)  # Convert to gray

    # Detect number of faces in original image
    faces = face_cascade.detectMultiScale(
            gray,
            scaleFactor=1.1,
            minNeighbors=3,
            minSize=(35,35)
    )
    rects = detector(gray, 2)
    facecnt = len(faces)
    print("Detected faces: %d" % facecnt) # Returns number of detected faces

    for rect in rects:
        (x, y, w, h) = rect_to_bb(rect)
        imutils.resize(frame[y:y+h, x:x+w], width=128) # Resize image based on rect

        lastImg = fa.align(frame, gray, rect)   # Align image based on features
        lastImg = cv2.resize(lastImg,(200,200)) # Crop Image
        lastImg = lastImg[15:200-15,30:200-30]  # Can change size of image here
        i += 1
        cv2.imwrite(os.path.join(save, "image%d.jpg" % i), lastImg) # Write to directory

    if cv2.waitKey(0) & 0xFF == ord('q'):
        sys.exit()
    cv2.destroyAllWindows()
```

MNIST Fashion and Handwritten Digits Neural Network

```
import os
#os.environ["CUDA_VISIBLE_DEVICES"] = "-1"      #making this -1 runs it on the cpu making it empty runs it on gpu
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

mnist = tf.keras.datasets.fashion_mnist         #change this to download the different MNSIT data
(x_train,y_train), (x_test,y_test) = mnist.load_data()
# In[8]:
x_train = tf.keras.utils.normalize(x_train, axis=1)
x_test = tf.keras.utils.normalize(x_test, axis=1)
# In[9]:
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(128, activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(128, activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(10, activation=tf.nn.softmax)) #output layer should have number of classes
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',metrics=['accuracy'])
model.fit(x_train,y_train,epochs=3)
# In[10]:
val_loss, val_acc = model.evaluate(x_test, y_test)
print(val_loss, val_acc)
```

This can be run as a simple python script, if everything is already downloaded onto the system the program will automatically download the dataset since it is part of the Keras Framework. This can also be run on Jupyter Notebook using the included file. To run it on the CPU you must set **os.environ["CUDA_VISIBLE_DEVICES"] = "-1"** to -1.

Facial Recognition using a Convolutional Neural Network

```
from keras.preprocessing.image import ImageDataGenerator
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import os
import cv2
import random
IMG_SIZE = 100
DATADIR = "C:/Users/Estepan/Documents/147/Data"     #this is the location the test data which has the data in seprate folders
CATEGORIES = ["Estevan","Yash"]                     #these are the folders that hold the taining data
CATEGORIES2 = ["EstevanNew","YashNew"]              #these folders hold the test data
# In[2]: This section imports the Training data labels it and shuffles it
training_data = []
def creat_training_data():
    for category in CATEGORIES:
        path = os.path.join(DATADIR, category)
        class_num = CATEGORIES.index(category)
        for img in os.listdir(path):
            img_array = cv2.imread(os.path.join(path,img), cv2.IMREAD_GRAYSCALE)
            new_array = cv2.resize(img_array, (IMG_SIZE,IMG_SIZE))
            training_data.append([new_array,class_num])

creat_training_data()
random.shuffle(training_data)
X = []
Y = []
for features, label in training_data:
    X.append(features)
    Y.append(label)

X = np.array(X).reshape(-1,IMG_SIZE,IMG_SIZE,1)
X = tf.keras.utils.normalize(X, axis=1)
# In[3]:# In[2]: This section imports the Testing data labels it and shuffles it

test_data =[]
def creat_test_data():
    for category in CATEGORIES2:
        path = os.path.join(DATADIR, category)
        class_num = CATEGORIES2.index(category)
        for img in os.listdir(path):
            img_array = cv2.imread(os.path.join(path,img), cv2.IMREAD_GRAYSCALE)
            new_array = cv2.resize(img_array, (IMG_SIZE,IMG_SIZE))
            test_data.append([new_array,class_num])


creat_test_data()
random.shuffle(test_data)
Xt = []
Yt = []
for features, label in test_data:
    Xt.append(features)
    Yt.append(label)

Xtt = Xt

Xt = np.array(Xt).reshape(-1,IMG_SIZE,IMG_SIZE,1)
Xt = tf.keras.utils.normalize(Xt, axis=1)
# In[4]: This is the NN model used to reconize the faces
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Conv2D(64, kernel_size = (3,3), input_shape =(IMG_SIZE,IMG_SIZE,1)))
model.add(tf.keras.layers.Activation("relu"))
model.add(tf.keras.layers.MaxPooling2D(pool_size=(2,2)))

model.add(tf.keras.layers.Conv2D(64, kernel_size = (3,3)))
model.add(tf.keras.layers.Activation("relu"))
model.add(tf.keras.layers.MaxPooling2D(pool_size=(2,2)))

model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(64))
model.add(tf.keras.layers.Dropout(.1))
model.add(tf.keras.layers.Dense(1))
model.add(tf.keras.layers.Activation("sigmoid"))


model.compile(optimizer='adam', loss='binary_crossentropy',metrics=['accuracy'])
model.fit(X,Y,batch_size =1,epochs=2)
# In[5]: This outputs the testing data with the name of who it thinks it is
predictions = model.predict([Xt])
for i in range(36):
    if predictions[i] > .5:
        print("Yash")
        plt.imshow(Xtt[i],cmap="gray")
        plt.show()
    elif predictions[i] > 0:
        print("Estevan")
        plt.imshow(Xtt[i],cmap="gray")
        plt.show()

val_loss, val_acc = model.evaluate(Xt, Yt)
print(val_loss, val_acc)
```

Since for this we used our own dataset it requires us to load and label the data into the program. The variable DATADIR should be changed to the location of the folder that holds the multiple folders of the different people. CATEGORIES should be changed to the names of the folders that holds the data you want to use to train. CATEGORIES2 should be changed to the names of the folders that holds the data you want to use to test. Once those variables are changed it can be run as a simple python script assuming everything has been installed on the system. This can also be run on Jupyter Notebook using the included file. To run it on the CPU you must set **os.environ["CUDA_VISIBLE_DEVICES"] = "-1"** to -1.

PyTorch CPU Full Code:

Final CPU.py

```
import torch
import torchvision
import numpy as np
import matplotlib.pyplot as plt
import torchvision.transforms as transforms
import torchvision.datasets as datasets
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable
import torch.optim as optim #this is the optimzer that is used to update the weights
import time

print(torch.__version__)
print(torchvision.__version__)


transform = transforms.Compose([transforms.ToTensor()])

train_set = torchvision.datasets.FashionMNIST(
    root='./data/FashionMNIST', #loaction on the disk where the data is
    train = True, #we want this to be the training data
    download = True, #download it if its not located at location of root
    transform = transforms.Compose([
        transforms.ToTensor()
    ]))

test_dataset = datasets.FashionMNIST(root='./data/FashionMNIST',
                           train=False,
                           download=True,
                           transform=transforms.Compose([transforms.ToTensor()]))


#hyper Parameters
num_epochs = 8
num_classes = 10
batch_size = 100
learning_rate = 0.001


train_loader = torch.utils.data.DataLoader(dataset=train_set,
                                           batch_size=batch_size,
                                           shuffle=True)

test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                          batch_size=batch_size,
                                          shuffle=False)

```

```
52
53  class Network(nn.Module):
54      def __init__(self):
55          super(Network, self).__init__()
56
57          self.cnn1 = nn.Conv2d(in_channels=1, out_channels=16, kernel_size=5, stride=1, padding=2)
58          self.cnn2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=5, stride=1, padding=2)
59
60          self.relu1 = nn.ReLU()
61          self.relu2 = nn.ReLU()
62
63          self.maxpool1 = nn.MaxPool2d(kernel_size=2)
64          self.maxpool2 = nn.MaxPool2d(kernel_size=2)
65
66          self.dropout = nn.Dropout(p=0.5)
67
68          self.fc1 = nn.Linear(32*7*7, 10)
69
70      def forward(self, x):
71          t = self.relu1(self.cnn1(x))
72
73          t = self.maxpool1(t)
74
75          t = self.relu2(self.cnn2(t))
76
77          t = self.maxpool2(t)
78
79          t = t.view(t.size(0), -1)
80
81          t = self.dropout(t)
82
83          t = self.fc1(t)
84          return t
```

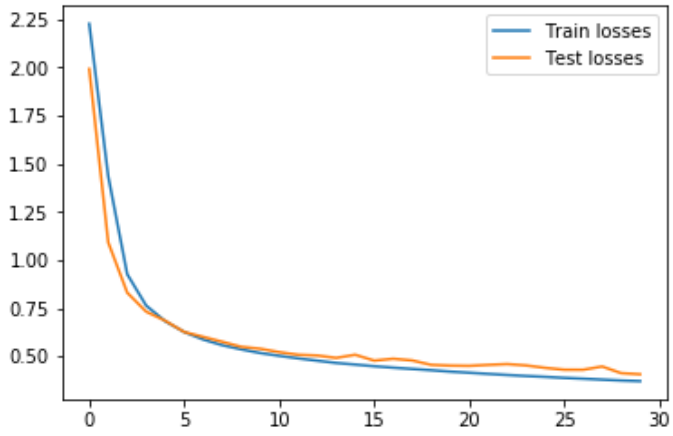
Final CPU.py

```
            t = self.fc1(t)
            return t

                #6 layers, input layer, 2 hidden conv layers, 2 hidden linear layers, and output layers


model = Network()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate) #lr is a parameter that you need to test and tune
#another common optimizer is SGD

iteration = 0
t0 = time.time()
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        images = Variable(images)
        labels = Variable(labels)

        #Clear the gradients
        optimizer.zero_grad()

        #Forward propagation, same as passing the batch to the network for analysis
        outputs = model(images)

        loss = criterion(outputs, labels) #this is what calculates the cross entropy loss

        #Calculates the gradietns and is technically backwards propagation
        loss.backward()

        # Update Weights
        optimizer.step()

        iteration = iteration + 1;

        #Total number of labels
        total = labels.size(0)

        #Obtaining predictions from max value
        _, predicted = torch.max(outputs.data, 1)

        #Calculate the number of correct answers
        correct = (predicted == labels).sum().item()

        #Print loss and accuracy
        if (i + 1) % 100 == 0:
            print('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}, Accuracy: {:.2f}%'
                  .format(epoch + 1, num_epochs, i + 1, len(train_loader), loss.item(),
                          (correct / total) * 100))

print('{} seconds'.format(time.time()-t0))


t0 = time.time()
with torch.no_grad():
    correct = 0
    total = 0
    for images, labels in test_loader:
        images = Variable(images)
        labels = Variable(labels)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    print('Test Accuracy of the model on the 10000 test images: {} %'.format(100 * correct / total))
print('{} Verification in seconds'.format(time.time()-t0))
```

The plot shows how the loss function works. We see the loss at each epoch get reduced which uses backpropagation to update weights to increase accuracy.

```
In [13]: import matplotlib.pyplot as plt
         %matplotlib inline

         plt.plot(train_losses,label = "Train losses")
         plt.plot(test_losses, label = "Test losses")
         plt.legend()
```

```
Out[13]: <matplotlib.legend.Legend at 0x20d78e681d0>
```



PyTorch GPU:

For the sake of space and repetitiveness, the GPU code is omitted due to its similarity with the CPU code. By default PyTorch will utilize the CPU, however to check for a CUDA enabled device on the system, one must run the following command:

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

This only isn't sufficient enough however, you must also move the CNN model onto the GPU and make modifications accordingly as is seen in the snippet below.

In [7]:
```
model = Network()
model = Network().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

In [8]:
```
iteration = 0
t0 = time.time()
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        images = Variable(images.cuda())
        labels = Variable(labels.cuda())

        #Clear the gradients
        optimizer.zero_grad()

        #Forward propagation
        outputs = model(images)

        #Calculating loss with softmax to obtain cross entropy loss
        loss = criterion(outputs, labels)

        #Backward propation
        loss.backward()

        #Updating gradients
        optimizer.step()

        iteration += 1

        #Total number of labels
        total = labels.size(0)

        #Obtaining predictions from max value
        _, predicted = torch.max(outputs.data, 1)

        #Calculate the number of correct answers
        correct = (predicted == labels).sum().item()

        #Print loss and accuracy
        if (i + 1) % 100 == 0:
            print('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}, Accuracy: {:.2f}%'
                  .format(epoch + 1, num_epochs, i + 1, len(train_loader), loss.item(),
                          (correct / total) * 100))

print('{} seconds'.format(time.time()-t0))
```

Sample Output using PyTorch scripts:

```
Epoch [3/8], Step [100/600], Loss: 0.4490, Accuracy: 83.00%
Epoch [3/8], Step [200/600], Loss: 0.2767, Accuracy: 92.00%
Epoch [3/8], Step [300/600], Loss: 0.5174, Accuracy: 80.00%
Epoch [3/8], Step [400/600], Loss: 0.3252, Accuracy: 88.00%
Epoch [3/8], Step [500/600], Loss: 0.2560, Accuracy: 91.00%
Epoch [3/8], Step [600/600], Loss: 0.3836, Accuracy: 87.00%
Epoch [4/8], Step [100/600], Loss: 0.3748, Accuracy: 89.00%
Epoch [4/8], Step [200/600], Loss: 0.4118, Accuracy: 86.00%
Epoch [4/8], Step [300/600], Loss: 0.2170, Accuracy: 90.00%
Epoch [4/8], Step [400/600], Loss: 0.2915, Accuracy: 86.00%
Epoch [4/8], Step [500/600], Loss: 0.2642, Accuracy: 89.00%
Epoch [4/8], Step [600/600], Loss: 0.2552, Accuracy: 95.00%
Epoch [5/8], Step [100/600], Loss: 0.3064, Accuracy: 91.00%
Epoch [5/8], Step [200/600], Loss: 0.2607, Accuracy: 90.00%
Epoch [5/8], Step [300/600], Loss: 0.3609, Accuracy: 92.00%
Epoch [5/8], Step [400/600], Loss: 0.2431, Accuracy: 91.00%
Epoch [5/8], Step [500/600], Loss: 0.3308, Accuracy: 86.00%
Epoch [5/8], Step [600/600], Loss: 0.3157, Accuracy: 89.00%
Epoch [6/8], Step [100/600], Loss: 0.4020, Accuracy: 89.00%
Epoch [6/8], Step [200/600], Loss: 0.2389, Accuracy: 93.00%
Epoch [6/8], Step [300/600], Loss: 0.3601, Accuracy: 88.00%
Epoch [6/8], Step [400/600], Loss: 0.4480, Accuracy: 85.00%
Epoch [6/8], Step [500/600], Loss: 0.4287, Accuracy: 83.00%
Epoch [6/8], Step [600/600], Loss: 0.4576, Accuracy: 87.00%
Epoch [7/8], Step [100/600], Loss: 0.2780, Accuracy: 92.00%
Epoch [7/8], Step [200/600], Loss: 0.2678, Accuracy: 89.00%
Epoch [7/8], Step [300/600], Loss: 0.2599, Accuracy: 93.00%
Epoch [7/8], Step [400/600], Loss: 0.3389, Accuracy: 86.00%
Epoch [7/8], Step [500/600], Loss: 0.2114, Accuracy: 90.00%
Epoch [7/8], Step [600/600], Loss: 0.2277, Accuracy: 90.00%
Epoch [8/8], Step [100/600], Loss: 0.2549, Accuracy: 92.00%
Epoch [8/8], Step [200/600], Loss: 0.2871, Accuracy: 90.00%
Epoch [8/8], Step [300/600], Loss: 0.4062, Accuracy: 81.00%
Epoch [8/8], Step [400/600], Loss: 0.1739, Accuracy: 95.00%
Epoch [8/8], Step [500/600], Loss: 0.3058, Accuracy: 88.00%
Epoch [8/8], Step [600/600], Loss: 0.1862, Accuracy: 94.00%
82.8073239326477 seconds
```

In [9]:

```
t0 = time.time()
with torch.no_grad():
    correct = 0
    total = 0
    for images, labels in test_loader:
        images = Variable(images.cuda())
        labels = Variable(labels.cuda())
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    print('Test Accuracy of the model on the 10000 test images: {} %'.format(100 * correct / total))
print('{} Verification in seconds'.format(time.time()-t0))
```

```
Test Accuracy of the model on the 10000 test images: 88.29 %
1.135472297668457 Verification in seconds
```

Evaluation/Results

In this extremely competitive and revealing Battle of the Threads, CPU is a winner for smaller datasets and neural networks. GPU is however the winner for larger datasets and CNNs or neural networks with a larger number of layers and epochs. While it can be argued that the training images were greyscale and that the dataset was relatively small for our facial recognition, we were able to repeatedly get similar results. After implementing the Fashion MNIST dataset on our CNNs and our own face dataset, we are able to conclusively tell that the GPUs are far better for not only the training portion of a model but also the testing portion. Due to the CPUs sequential calculations and the branching nature of NNs, it isn't surprising to see that the GPU was able to outperform the CPU.

Problems Faced

The primary trouble we faced had to do with understanding the workings of Neural Networks and its implementation on Keras and PyTorch. We had no prior knowledge of ML or CNN nor did we have much programming experience on Python. Implementation of the CNN on PyTorch was especially difficult because the 24 video tutorial we were following didn't end up covering the training part of the model because it is still an ongoing tutorial series. We were able to look at other sources, however, for guidance on how to do the training portion of the project. When coding, PyTorch was a lot more difficult to code because of the granular control it offers to the user. A PyTorch CNN coder must know the input and output sizes at each layer or else the code will not work.

When creating a neural network and training the data, instead of our network generalizing the data, it was overfitting the data. Overfitting is a modeling error which occurs when a function is too closely fit to a limited set of data points. We had to train the data to generalize the test images, instead of overfitting the images, which creates a super high accuracy rate.

Conclusion

When we first started this project we only knew the very basics of neural networks, in that they are used in data analysis and product prediction but not much was known about the innerworkings. By the end of the project we understood the high level and learned a lot about how much GPUs can speed up the process of a complex neural network. Overall this was an interesting and enlightening project.

References

https://www.deeplearningwizard.com/deep_learning/practical_pytorch/pytorch_convolutional_neuralnetwork/

https://keras.io

https://www.tensorflow.org/install/gpu

Presentation

https://docs.google.com/presentation/d/1PSDInOr-Gm7OV7mCscY83sI438KIBbatBSAHAjKyr9U/edit?usp=sharing