ECE421S – Introduction to Machine Learning

Assignment 2

Neural Networks

Hard Copy Due: March 13, 2019 @ BA3014, 4:00-5:00 PM EST

Code Submission: ece421ta2019@gmail.com

March 13, 2019 @ 5:00 PM EST

General Notes:

- Attach this cover page to your hard copy submission
- For assignment related questions, please contact Matthew Wong (matthewck.wong@mail.utoronto.ca)
- For general questions regarding Python or Tensorflow, please contact Tianrui Xiao (tianrui.xiao@mail.utoronto.ca) or see him in person in his office hours, Tuesdays, 4:00-6:00 PM in BA-3128 (Robotics Lab)

Please circle section to which you would like the assignment returned

Tutorial Sections

001	002	003	004
005	006	007	Graduate

Group Members	
Names	StudentID
Alexander APOSTOLOV	1005644279
Anthony KEMMEUGNE	1004686789

ECE421 Assignment 2: Neural Network

Alexander APOSTOLOV (1005644279) and Anthony KEMMEUGNE (1004686789)

Participation percentage 50%-50%

1. Neural Network using Numpy

1.1. Helper Function

To avoid confusion, we define the following matrices (as in the lecture). Let the input to the Neural Network be Xi and the output of the hidden layer ReLu functions be Xh and the output softmax functions be Xo (let h be the number of hidden units and w the number of input units and K the number of classes):

$$Xi = \begin{bmatrix} xi_1^{(1)} & \cdots & xi_1^{(1)} \\ \vdots & \ddots & \vdots \\ xi_1^{(N)} & \cdots & xi_w^{(N)} \end{bmatrix}$$

$$Xh = \begin{bmatrix} xh_1^{(1)} & \cdots & xh_h^{(1)} \\ \vdots & \ddots & \vdots \\ xh_1^{(N)} & \cdots & xh_h^{(N)} \end{bmatrix}$$

$$Xo = \begin{bmatrix} xo_1^{(1)} & \cdots & xo_K^{(1)} \\ \vdots & \ddots & \vdots \\ xo_1^{(N)} & \cdots & xo_K^{(N)} \end{bmatrix}$$

Let's define the input to the hidden layer ReLu functions to be Sh and let the So be the input to the softmax functions:

$$Sh = \begin{bmatrix} sh_1^{(1)} & \cdots & sh_h^{(1)} \\ \vdots & \ddots & \vdots \\ sh_1^{(N)} & \cdots & sh_h^{(N)} \end{bmatrix}$$

$$So = \begin{bmatrix} so_1^{(1)} & \cdots & so_K^{(1)} \\ \vdots & \ddots & \vdots \\ so_1^{(N)} & \cdots & so_K^{(N)} \end{bmatrix}$$

$$\text{Let T} = \begin{bmatrix} t_1^{(1)} & \cdots & t_K^{(1)} \\ \vdots & \ddots & \vdots \\ t_1^{(N)} & \cdots & t_K^{(N)} \end{bmatrix} \text{ be the matrix with the labels}$$

and let $a^{(n)}$ be the n^{th} line of the A matrix and a_k the k^{th} column of A

5. gradCE(): the average cross-entropy function is given by:

$$\text{averageCE} = -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} t_k^{(n)} \log\left(xo_k^{(n)}\right)$$

and the cross entropy function for datapoint n is given by

$$CE^{(n)} = -\sum_{k=1}^{K} t_k^{(n)} \log\left(xo_k^{(n)}\right)$$

We first find the formula for the derivative of the cross entropy with respect to the softmax of a prediction for datapoint n:

$$\frac{\partial CE^{(n)}}{\partial xo_k^{(n)}} = -\, t_k^{(n)} \frac{1}{xo_k^{(n)}}$$

so the gradient matrix is given by:

$$\nabla CE^{(n)} = \begin{bmatrix} \frac{\partial CE}{\partial xo_1^{(n)}} \\ \vdots \\ \frac{\partial CE}{\partial xo_K^{(n)}} \end{bmatrix} = -t^{(n)} \otimes \widetilde{xo^{(n)}},$$

$$\text{where } \widetilde{xo^{(n)}} = \begin{bmatrix} \frac{1}{xo_1^{(n)}} \\ \vdots \\ \frac{1}{xo_K^{(n)}} \end{bmatrix}$$

and $\otimes$ is the elementwise multiplication of two matrices with the same dimensions

Similarly, we have that:

$$\frac{\partial \text{averageCE}}{\partial xo_k} = \frac{1}{N}\sum_{n=1}^{N} \frac{\partial CE^{(n)}}{\partial xo_k^{(n)}} \text{ , } so$$

$$\nabla averageCE = \begin{bmatrix} \frac{\partial \text{averageCE}}{\partial xo_1} \\ \vdots \\ \frac{\partial \text{averageCE}}{\partial xo_1} \end{bmatrix}$$

Here is a snippet of our code implementation of $\nabla averageCE$:

```
#This function accepts two arguments, the targets (e.g. labels) and predictions.
#It returns the gradient of the average cross entropy loss with respect to the softmax of the
#predictions
def gradAverageCE(target, prediction):
    S = np.apply_along_axis(softmax, 1, prediction)
    S_reciprocal = np.reciprocal(S)
    N = target.shape[0]
    result = (-1.0/N)*np.multiply(target, S_reciprocal)
    result[np.isnan(result)]=0.0
    return result
```

1.2 Backpropagation Derivation

The loss function L here is:

$$L = -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} t_k^{(n)}\, log\left(xo_k^{(n)}\right)$$

To avoid confusion, note that $\mathrm{Xo} = \mathrm{S}$ of the handout

Let's rewrite the loss with these:

$$L = -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} t_k^{(n)}\, log\left(xo_k^{(n)}\right) = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_k^{(n)}\, log\left(\frac{e^{so_k^{(n)}}}{\sum_{k=1}^{K} e^{so_k^{(n)}}}\right)$$

$$= -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} t_k^{(n)}\left[log\left(e^{so_k^{(n)}}\right) - \log\left(\sum_{k=1}^{K} e^{so_k^{(n)}}\right)\right]$$

$$= -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} t_k^{(n)}\left[so_k^{(n)} - \log\left(\sum_{k=1}^{K} e^{so_k^{(n)}}\right)\right]$$

Let's calculate

$$\frac{\boldsymbol{\partial L}}{\boldsymbol{\partial so_i^{(n)}}} = \delta o_i^{(n)} = -\frac{1}{N}\left[t_i^{(n)} - \sum_{j=1}^{K}\frac{t_j^{(n)} e^{so_i^{(n)}}}{\sum_{k=1}^{K} e^{so_k^{(n)}}}\right] = \frac{1}{N}\left[-t_i^{(n)} + \sum_{j=1}^{K} t_j^{(n)}\; xo_i^{(n)}\right]$$

$$= \frac{\mathbf{1}}{\boldsymbol{N}}\left[\boldsymbol{xo_i^{(n)}} - \boldsymbol{t_i^{(n)}}\right] \quad \langle 1\rangle$$

from the above, it follows:

$$\frac{\boldsymbol{\partial L}}{\boldsymbol{\partial wo_{i,j}}} = \sum_{n=1}^{N} \frac{\partial L}{\partial so_j^{(n)}} * \frac{\partial so_j^{(n)}}{\partial wo_{i,j}} = \sum_{n=1}^{N} \delta o_j^{(n)} * \frac{\partial so_j^{(n)}}{\partial wo_{i,j}} = \boldsymbol{\sum_{n=1}^{N} \delta o_j^{(n)} * xh_i^{(n)}}$$

$$= \frac{1}{N}\sum_{n=1}^{N}\left[xo_j^{(n)} - t_j^{(n)}\right] * xh_i^{(n)}$$

So we get:

$$\frac{\boldsymbol{\partial L}}{\boldsymbol{\partial Wo}} = \begin{bmatrix} \frac{\partial L}{\partial wo_{1,1}} & \cdots & \frac{\partial L}{\partial wo_{1,K}} \\ \vdots & \ddots & \vdots \\ \frac{\partial L}{\partial wo_{h,1}} & \cdots & \frac{\partial L}{\partial wo_{h,K}} \end{bmatrix} = \frac{1}{N} Xh^T [Xo - T] \in \mathbb{R}^{(h \times K)} \left(= \mathbb{R}^{(1000 \times 10)}\right)$$

from ⟨1⟩, it follows:

$$\frac{\boldsymbol{\partial L}}{\boldsymbol{\partial bo_j}} = \sum_{n=1}^{N} \frac{\partial L}{\partial so_j^{(n)}} * \frac{\partial so_j^{(n)}}{\partial bo_j} = \sum_{n=1}^{N} \delta o_j^{(n)} * \frac{\partial so_j^{(n)}}{\partial bo_j} = \boldsymbol{\sum_{n=1}^{N} \delta o_j^{(n)}} = \frac{1}{N}\sum_{n=1}^{N}\left[xo_j^{(n)} - t_j^{(n)}\right]$$

So, we get:

$$\frac{\boldsymbol{\partial L}}{\boldsymbol{\partial bo}} = \begin{bmatrix} \frac{\partial L}{\partial bo_1} & \cdots & \frac{\partial L}{\partial bo_K} \end{bmatrix} = \frac{1}{N}\left[\sum_{n=1}^{N}\left[xo_1^{(n)} - t_1^{(n)}\right] \quad \cdots \quad \sum_{n=1}^{N}\left[xo_K^{(n)} - t_K^{(n)}\right]\right]$$

$$= \left[\sum_{n=1}^{N} \delta o_1^{(n)} \quad \cdots \quad \sum_{n=1}^{N} \delta o_K^{(n)}\right] \in \mathbb{R}^{(1 \times K)} \left(= \mathbb{R}^{(1 \times 10)}\right)$$

In class we have seen:

$$\frac{\boldsymbol{\partial L}}{\boldsymbol{\partial sh_i^{(n)}}} = \delta h_i^{(n)} = \sum_{j=1}^{K} \frac{\partial L}{\partial so_j^{(n)}} \frac{\partial so_j^{(n)}}{\partial xh_i^{(n)}} \frac{\partial xh_i^{(n)}}{\partial sh_i^{(n)}} = \boldsymbol{\sum_{j=1}^{K} \delta o_j^{(n)} * wo_{i,j} * \theta_h{}' \left(xh_i^{(n)}\right)}$$

where θ_h is the activation function of the hidden layer.

We have that:

$$\theta_h{}'(x) == ReLU'(x) = \begin{cases} 1, & x > 0 \\ 0, & x \le 0 \end{cases}$$

From the above it follows :

$$\frac{\boldsymbol{\partial L}}{\boldsymbol{\partial wh_{i,j}}} = \sum_{n=1}^{N} \frac{\partial L}{\partial sh_j^{(n)}} \frac{\partial sh_j^{(n)}}{\partial wh_{i,j}} = \boldsymbol{\sum_{n=1}^{N} \delta h_j^{(n)} * xi_i} = \sum_{n=1}^{N}\sum_{a=1}^{K} \frac{\partial L}{\partial so_a^{(n)}} * wo_{j,a} * \theta_h{}' \left(xh_j^{(n)}\right) * xi_i$$

$$= xi_i * \sum_{n=1}^{N}\sum_{a=1}^{K} \frac{\partial L}{\partial so_a^{(n)}} * wo_{j,a} * \theta_h{}' \left(xh_j^{(n)}\right)$$

From the above and from ⟨1⟩, it follows:

$$\frac{\partial \boldsymbol{L}}{\partial \boldsymbol{Wh}} = \begin{bmatrix} \frac{\partial L}{\partial wh_{1,1}} & \cdots & \frac{\partial L}{\partial wo_{1,h}} \\ \vdots & \ddots & \vdots \\ \frac{\partial L}{\partial wo_{w,1}} & \cdots & \frac{\partial L}{\partial wo_{w,h}} \end{bmatrix} = Xi^T \cdot \left[\left(\frac{1}{N}[Xo - T]\right) \cdot Wo^T\right] \otimes \ \theta_h{}'(Xh)$$

$$\in \ \mathbb{R}^{(w\times h)}\left(= \mathbb{R}^{(784\times 1000)}\right)$$

Similarly:

$$\frac{\partial \boldsymbol{L}}{\partial \boldsymbol{bh_i}} = \sum_{n=1}^{N} \frac{\partial L}{\partial sh_j^{(n)}} \frac{\partial sh_j^{(n)}}{\partial bh_i} = \sum_{\boldsymbol{n=1}}^{\boldsymbol{N}} \boldsymbol{\delta h_j^{(n)}} = \sum_{n=1}^{N}\sum_{a=1}^{K} \frac{\partial L}{\partial so_a^{(n)}} * wo_{j,a} * \theta_h{}'\left(xh_j^{(n)}\right)$$

$$\frac{\partial \boldsymbol{L}}{\partial \boldsymbol{bh}} = \begin{bmatrix} \frac{\partial L}{\partial bh_1} & \cdots & \frac{\partial L}{\partial bh_h} \end{bmatrix}$$

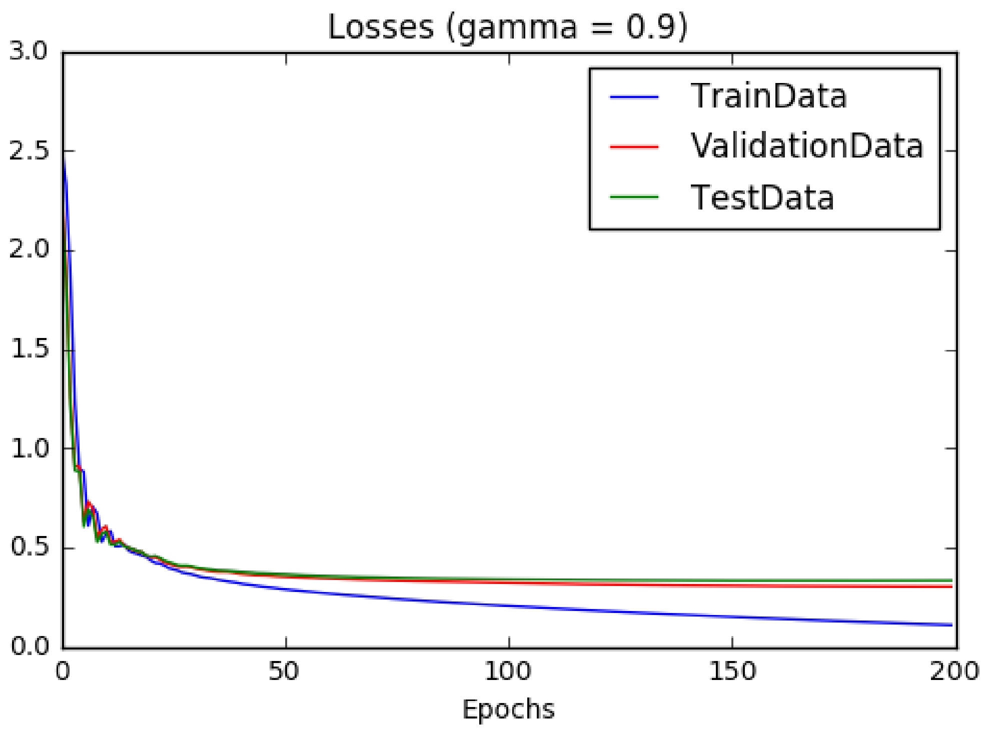
$$= \left[\sum_{n=1}^{N}\sum_{a=1}^{K} \frac{\partial L}{\partial so_a^{(n)}} * wo_{1,a} * \theta_h{}'\left(xh_1^{(n)}\right) \quad \cdots \quad \sum_{n=1}^{N}\sum_{a=1}^{K} \frac{\partial L}{\partial so_a^{(n)}} * wo_{h,a} * \theta_h{}'\left(xh_h^{(n)}\right)\right]$$

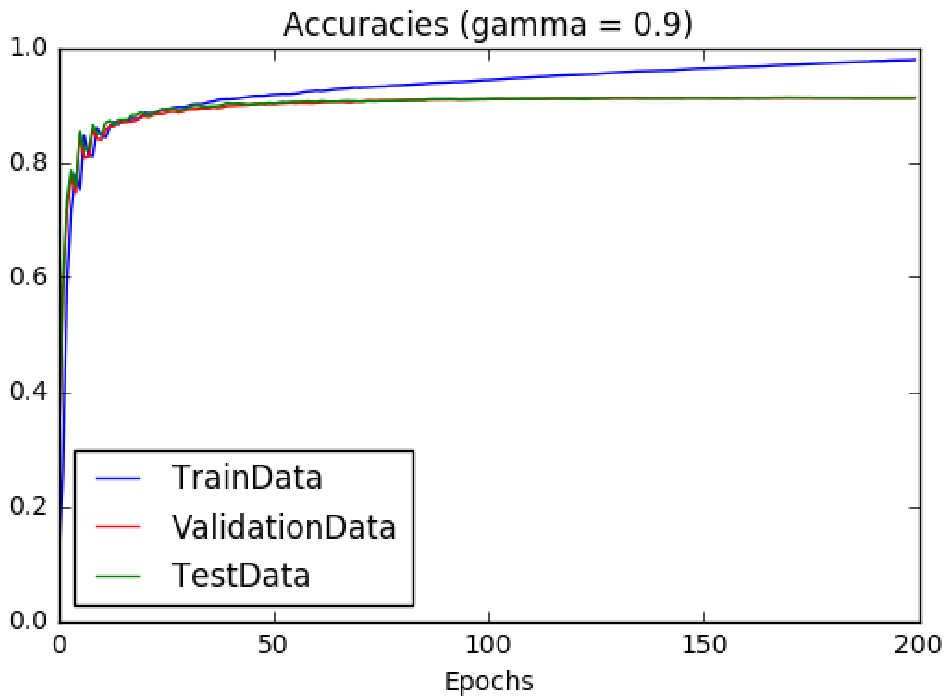
$$\in \ \mathbb{R}^{(1\times h)}\left(= \mathbb{R}^{(1\times 1000)}\right)$$

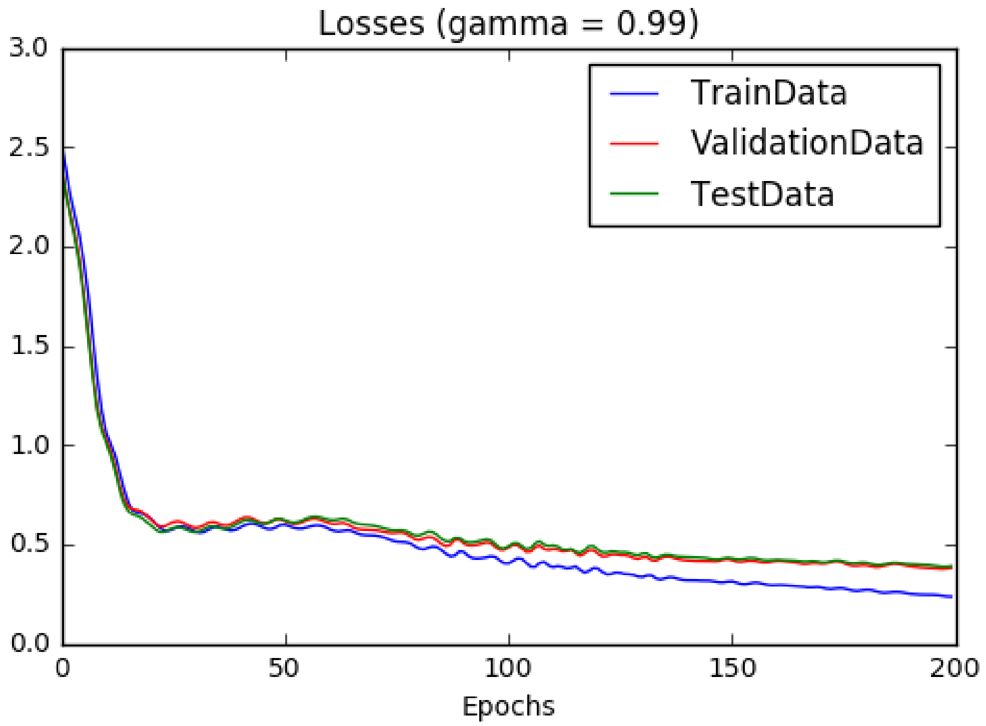
Note that this is the sum of the columns of $\left[\left(\frac{1}{N}[Xo - T]\right) \cdot Wo^T\right] \otimes \ \theta_h{}'(Xh)$

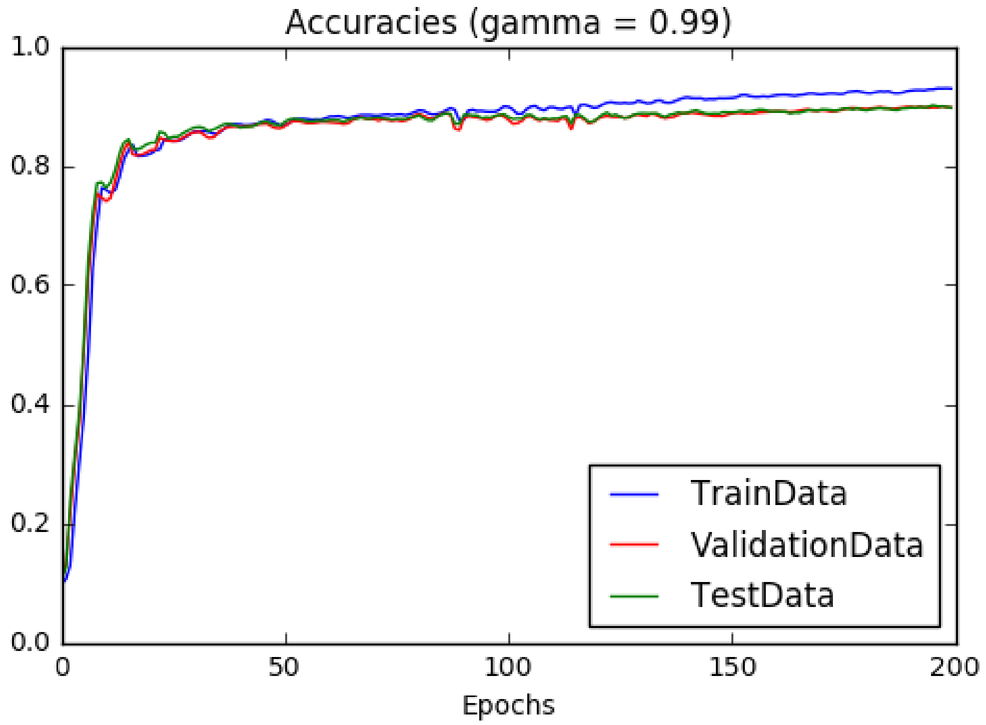
1.3 Learning

Here are the plots of the Loss and the accuracy with γ=0.9 and γ=0.99







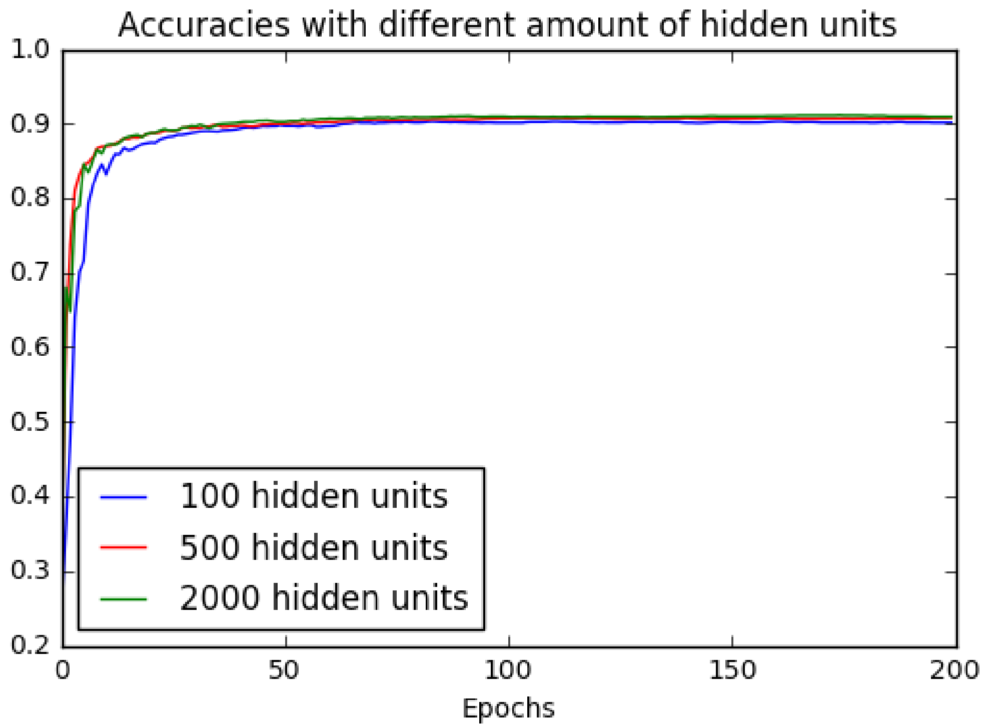


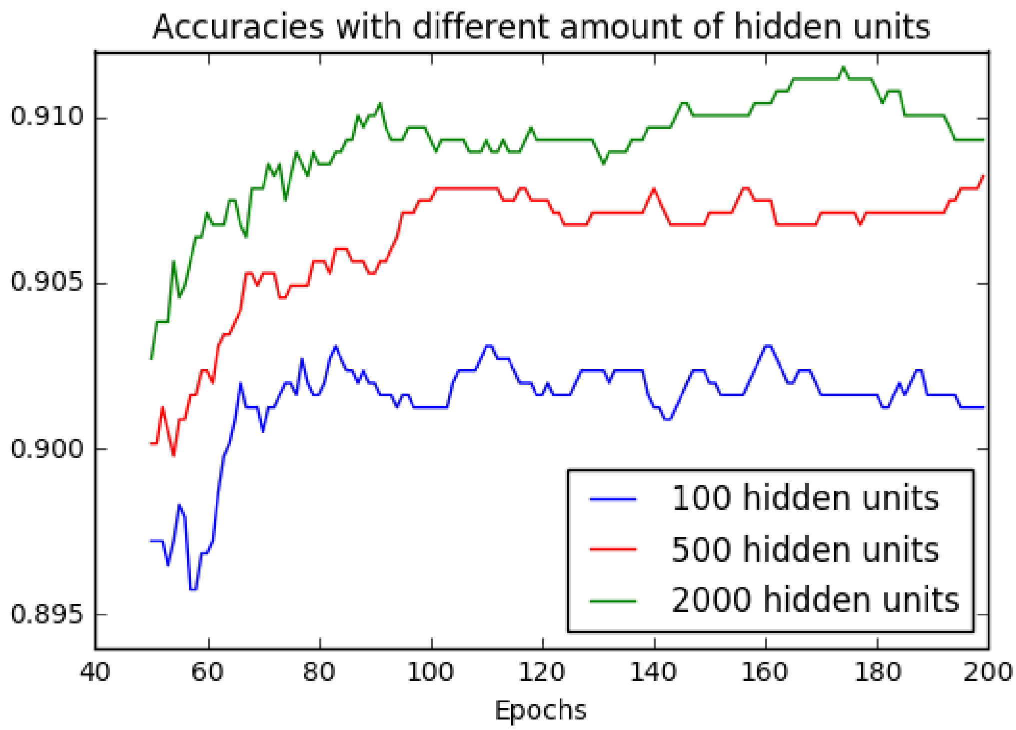
We see that with both parameter values we get really good results very fast, in less than 50 epochs. However, when gamma is 0.9, we get better final results. In both cases, we see some overfitting starting to happen, the training accuracy increases whithout the validation and test accuracy to increase that much or even at all. This trend is stronger when gamma=0.9.

1.4 Hyperparameter Investigation

1.Number of hidden units

We have run the model with different amounts of hidden units and here are the plots of the accuracies of the test dataset. The second graph zooms on the plot after epoch 50:





We see that the three models have very similar behaviours and that they reach 90% accuracy really fast, in less than 50 steps. It seems, however that the more hidden units we use, the better the accuracy. This is what one would expect since the bigger the number of hidden units, the bigger the complexity of the dataset we can interpret with the neural network. However, the final accuracies are really similar, between 500 and 2000 hidden units, so we can think that for this dataset, it might be an overkill to use more than 500 hidden units (since the more hidden units we use, the bigger the complexity of the NN and the longer we need to train it).

2.Early stopping

We will focus on gamma=0.9. By looking at the two plots above, we see that after epoch 60, the testing and validation loss doesn't seem to decrease significantly, whereas the training one does. This might be a sign that the NN is doing unnecessary computations and even overfitting the training data. So, we can choose the early stopping point at 60 epochs to avoid overkill, as we can see from the table below:

	Validation Loss	Test Loss
Epoch 40:	0.36633713248995864	0.37604281395359807
Epoch 50:	0.3518234723584119	0.3643221209515217
Epoch 60:	0.3421867139385551	0.3549265859185476
Epoch 100:	0.3193604053361294	0.33870520224351347
Epoch 200:	0.30149358234838475	0.33268574581248406

Here, we make the assumption that we define the early stopping point as the moment after which the test and validation loss stop decreasing significantly. If we were to use another assumption, which defines it as the moment when the testing or validation loss start increasing over a given interval, we would not be able to define an early stopping point before the 200^{th} since the training and validation losses are always (slightly) decreasing.

2. Neural Network in TensorFlow

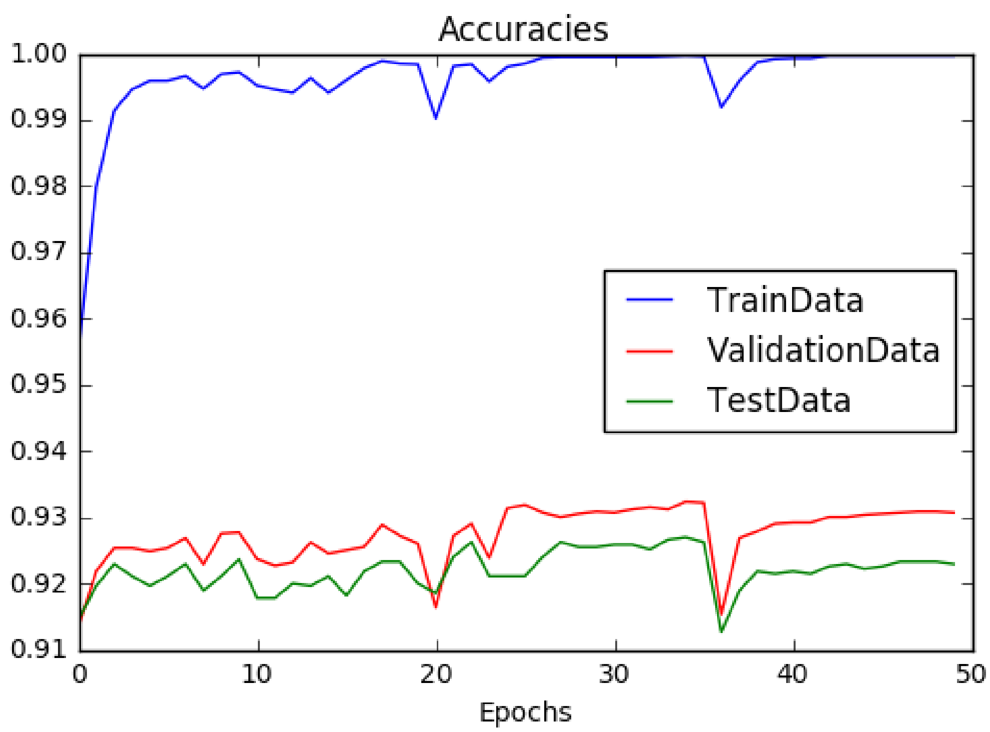
2.1 Model Implementation

Below, you will find the Python code used to model the required CNN:

```
def cnn(learning_rate, epochs=50, batch_size=32, L2_loss=False, regularization=0.0, dropLayer=False, keep = 1.0, onlyFinal=False):
    tf.set_random_seed(421)
    initializer = tf.contrib.layers.xavier_initializer()

    wc= tf.Variable(initializer([3,3,1,32]))
    bc =tf.Variable(initializer([32]))

    sig1 = np.sqrt(2/((14*14*32)+784))
    sig2 = np.sqrt(2/(784+10))
    w1 = tf.Variable(tf.random_normal([14*14*32, 784], stddev = sig1), name='w1')
    b1 = tf.Variable(tf.random_normal([784] ,stddev = sig1), name = 'b1')

    w2 = tf.Variable(tf.random_normal([784, 10],stddev = sig2), name='w2')
    b2 = tf.Variable(tf.random_normal([10], stddev = sig2),  name = 'b2')


    x = tf.placeholder(tf.float32, shape=(None, 784), name='x')
    y = tf.placeholder(tf.float32, shape=(None, 10), name='y')
    reg = tf.placeholder(tf.float32, name='reg')

    input_layer = tf.reshape(x, shape=[-1, 28, 28, 1])
    conv = tf.nn.conv2d(input=input_layer, filter=wc, strides=[1,1,1,1], padding="SAME")
    conv = tf.nn.bias_add(conv, bc)

    relu1 = tf.nn.relu(conv)

    batch_mean, batch_var = tf.nn.moments(relu1,[0])
    normal = tf.nn.batch_normalization(relu1, batch_mean, batch_var, offset = None, scale = None, variance_epsilon = 1e-3)

    maxpool = tf.nn.max_pool(normal, ksize=[1,2,2,1], strides=[1,2,2,1], padding="SAME")

    flat = tf.reshape(maxpool, [-1, 14*14*32])

    full = tf.add(tf.matmul(flat, w1), b1)

    if(dropLayer):
        drop_layer = tf.nn.dropout(full, keep_prob=keep)
        relu2 = tf.nn.relu(drop_layer)
    else:
        relu2=tf.nn.relu(full)


    out = tf.add(tf.matmul(relu2, w2), b2)

    softmax_layer = tf.nn.softmax(out)

    if(L2_loss):
        loss_op = (tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(
        logits=out, labels=y)) +
        reg*tf.nn.l2_loss(wc) +
        reg*tf.nn.l2_loss(w1) +
        reg*tf.nn.l2_loss(w2))
    else:
        loss_op = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(logits=out, labels=y))

    optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
    train_op = optimizer.minimize(loss_op)

    correct_pred = tf.equal(tf.argmax(softmax_layer, 1), tf.argmax(y, 1))
    accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))

    init = tf.global_variables_initializer()

    dimw = 784
    #initialize the datasets
    trainData, validData, testData, trainTarget, validTarget, testTarget = loadData()
    newtrain, newvalid, newtest = convertOneHot(trainTarget, validTarget, testTarget)
    num_examples = trainTarget.shape[0]
    num_examples_valid = validTarget.shape[0]
    num_examples_test = testTarget.shape[0]
    X = np.zeros((num_examples,dimw))
    Xvalid = np.zeros((num_examples_valid,dimw))
    Xtest = np.zeros((num_examples_test,dimw))
    for i in range(0,num_examples):
        X[i]=trainData[i].flatten()
    for i in range(0,num_examples_valid):
        Xvalid[i]=validData[i].flatten()
    for i in range(0,num_examples_test):
        Xtest[i]=testData[i].flatten()


    #prepareTables
    trainLoss = np.zeros(epochs)
    trainAccuracy = np.zeros(epochs)
    validationLoss = np.zeros(epochs)
    validationAccuracy = np.zeros(epochs)
    testLoss = np.zeros(epochs)
    testAccuracy = np.zeros(epochs)


    with tf.Session() as sess:
        sess.run(init)
        number_of_batches = num_examples//batch_size

        for step in range(0, epochs):
            flat_x_shuffled,trainingLabels_shuffled = shuffle(X, newtrain)

            for minibatch_index in range(0,number_of_batches):
                #select miniatch and run optimizer
                minibatch_x = flat_x_shuffled[minibatch_index*batch_size: (minibatch_index + 1)*batch_size, :]
                minibatch_y = trainingLabels_shuffled[minibatch_index*batch_size: (minibatch_index + 1)*batch_size, :]
                sess.run(train_op, feed_dict={x: minibatch_x, y: minibatch_y, reg: regularization})

            if(step==epochs-1 or not(onlyFinal)):
                lossTrain, accTrain = sess.run([loss_op, accuracy], feed_dict={x: flat_x_shuffled, y: trainingLabels_shuffled, reg: regul
                lossValid, accValid = sess.run([loss_op, accuracy], feed_dict={x: Xvalid, y: newvalid, reg: regularization})
                lossTest, accTest = sess.run([loss_op, accuracy], feed_dict={x: Xtest, y: newtest, reg: regularization})
                trainLoss[step]=lossTrain
                trainAccuracy[step]=accTrain
                validationLoss[step]=lossValid
                validationAccuracy[step]=accValid
                testLoss[step]=lossTest
                testAccuracy[step]=accTest
                print("Step " + str(step+1) + ", Train Loss= " + \
                            "{:.8f}".format(lossTrain) + ", Training Accuracy= " + \
                            "{:.8f}".format(accTrain))
            else:
                print("Step " + str(step+1) + " (no data, only final losses and accuracies will be saved)")
        print("Optimization Finished!")
    return trainLoss, trainAccuracy, validationLoss, validationAccuracy, testLoss, testAccuracy
```

2.2 Model Training

We trained the model by minimizing the cross-entropy loss with an Adam optimizer with a learning rate of 1*10^-4 with SGD with minibatch size of 32 for 50 epochs. Here are the losses and accuracies over the training process:



Here, we see that this model gets good results extremely fast (in less than 4 epochs). However we see clear signs of overfitting. Our model perform well on the training set but fell to generalize. After epoch 3, we see that the validation and testing loss are increasing whereas the training loss is decreasing.

This is a model, where an early stoppping point would actually make a difference in improving the learning . We could fix the early stopping point at epoch 4.

2.3 Hyperparameter Investigation

1. L2 Normalization

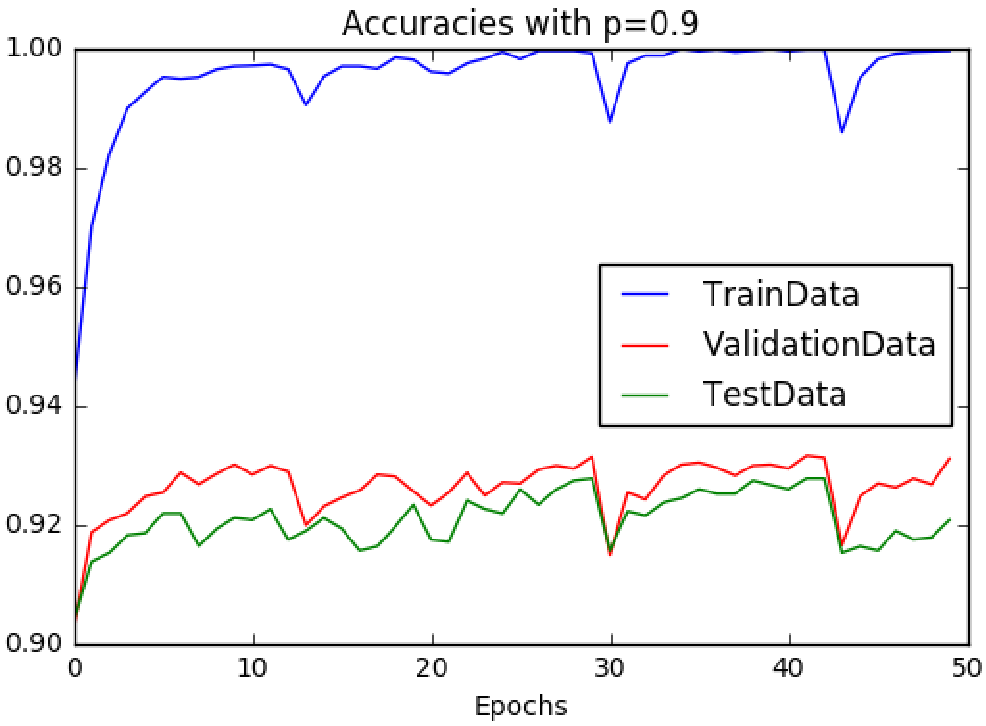
We have implemented L2 regularization with regularization parameter 0.01, 0.1, 0.5 while holding all parameters as before and here are the final accuracies for each scenario:

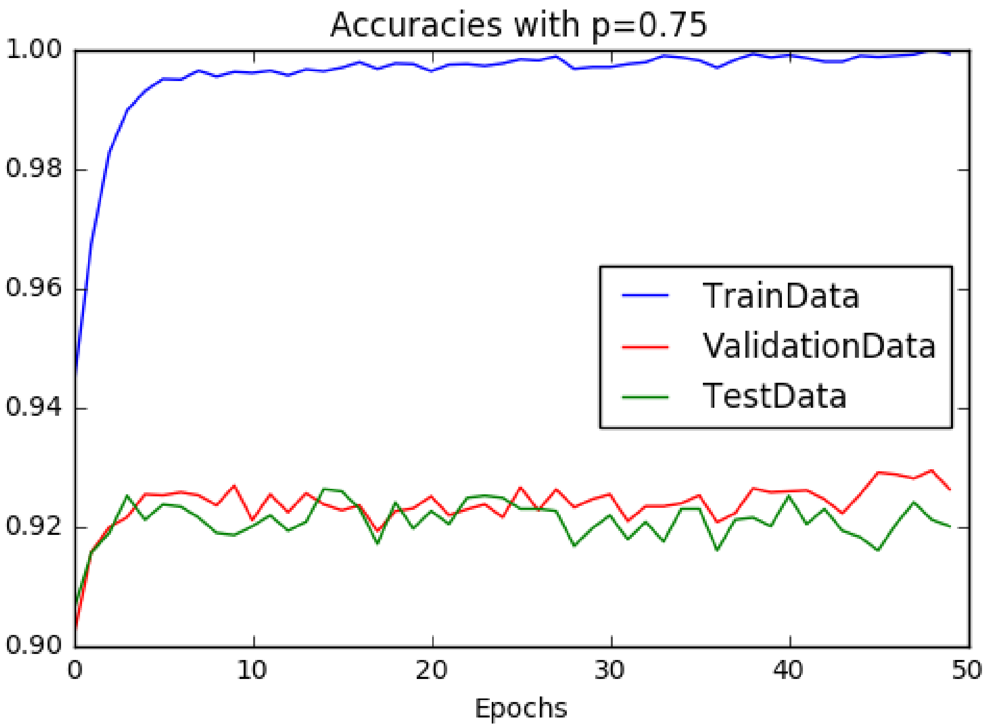
	λ=0.01	λ=0.1	λ=0.5
Training Accuracy	0.9888	0.9182	0.8719
Validation Accuracy	0.9288	0.9040	0.8658
Test Accuracy	0.9258	0.9119	0.8763

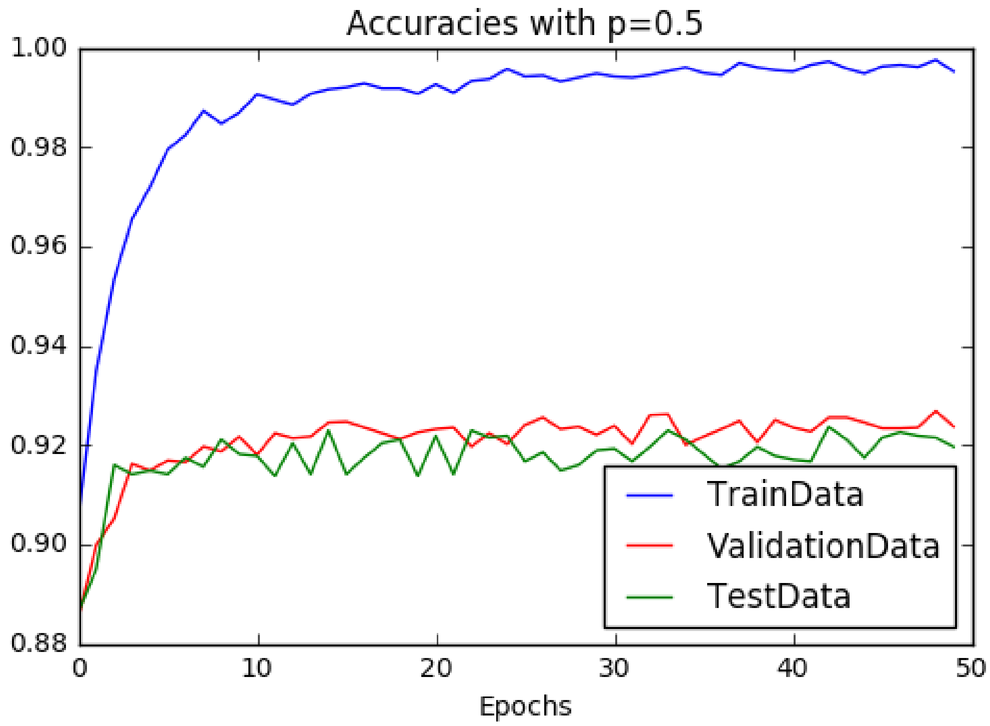
We see that when we use L2 regularization, we decrease the overfitting effect. Indeed, we see that the higher the regularization parameter, the closer all the accuracies stay together, which is a sign that the overfitting effects are minimized. This is what one would expect, since L2 regularization is used to avoid overfitting by including the squared norm of the weights in the loss. It prevents weights from getting too big just to fit the training dataset. However, we see that the bigger the regularization coefficient, the worst the accuracies for all the datasets. This can be explained by the fact that since we are adding a weighted sum of the squared norm of the weights, the model will want to keep the weights as low as possible, which might have the effect of making the model less accurate. So, when we choose the regularization parameter, we have to make a trade-off between minimizing the overfitting and the final accuracies. Between these three scenarios, the best one seems to be the second one, because the overfitting effect is almost absent and the accuracies of the test and validation sets are as close to the first scenario (which we don't choose because there is a lot of overfitting).

2. Dropout

We have added a dropout layer with different probabilities of keeping each unit (0.9, 0.75 and 0.5). The goal of this method is to avoid any node having too much influence on the final result. The method just randomly and independently drops some of them at each training iteration. Here are the plots of the accuracies in each scenario:







We see that when we use a lower keep probability p, we smooth the accuracies over the training process for all accuracies. Besides, the lower the keep probability p is, the closer the accuracies on the different data set get. This is because the dropout technique helps reducing overly big interdependences amongst neurons and therefore prevent over-fitting. Even though the training accuracy is higher than the two others, the validation and test accuracies don't seem to decrease significantly over the training process (they just slightly fluctuate). This means that we never increase the accuracy of the training set at the detriment of the two other sets. Besides we noticed that for p = 0.5 the accuracies increase slower than when we used p = 0.75 even though the curves shapes stay approximately the same. This is probably because since at every iteration almost half the nodes of the dropout layer are dropped, it takes more time to get to train every single one of them. We also see that this method would benefit from an early stopping at roughly epoch 10, since after that the accuracies just fluctuate around the same value, so it is just an overkill of computational time and power.

To conclude, we can say that by training a convolutional neural network, we get better results than training a simple neural network. However a convolutional neural network requires significantly more computational power for the training, and is more sensitive to overfitting.

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Mar  7 14:24:10 2019

@author: anthonykemmeugne
@author: AlexanderApostolov

"""
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
import math
import pickle
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'

# Load the data
def loadData():
    with np.load("notMNIST.npz") as data:
        Data, Target = data["images"], data["labels"]
        np.random.seed(521)
        randIndx = np.arange(len(Data))
        np.random.shuffle(randIndx)
        Data = Data[randIndx] / 255.0
        Target = Target[randIndx]
        trainData, trainTarget = Data[:10000], Target[:10000]
        validData, validTarget = Data[10000:16000], Target[10000:16000]
        testData, testTarget = Data[16000:], Target[16000:]
    return trainData, validData, testData, trainTarget, validTarget, testTarget

def convertOneHot(trainTarget, validTarget, testTarget):
    newtrain = np.zeros((trainTarget.shape[0], 10))
    newvalid = np.zeros((validTarget.shape[0], 10))
    newtest = np.zeros((testTarget.shape[0], 10))

    for item in range(0, trainTarget.shape[0]):
        newtrain[item][trainTarget[item]] = 1
    for item in range(0, validTarget.shape[0]):
        newvalid[item][validTarget[item]] = 1
    for item in range(0, testTarget.shape[0]):
        newtest[item][testTarget[item]] = 1
    return newtrain, newvalid, newtest

#Shuffles the train Data (to be used at the end of each epoch)
def shuffle(trainData, trainTarget):
    np.random.seed(421)
    randIndx = np.arange(len(trainData))
    target = trainTarget
    np.random.shuffle(randIndx)
    data, target = trainData[randIndx], target[randIndx]
    return data, target


def softmax(z):
    expon = np.exp(z)
    A = expon/np.sum(expon, axis=1, keepdims=True)
    return A

#This function accepts 3 arguments: a weight, an input, and a bias matrix and
#returns the product between the weights and input, plus the biases.
```

```python
def compute(X_prev, W, b):
    pre = np.matmul(X_prev,W)
    return np.add(pre,b)

def relu(x):
    return np.maximum(x, 0)

#This function accepts two arguments, the targets (e.g. labels - not onehot encoded!!!) and pred
#tions. It returns a number, the AVERAGE cross entropy loss.
def averageCE(target, prediction):
    N = prediction.shape[0]
    #we don't need to sum the logs for incorrect predictions
    correct_logprobs = -np.log(prediction[range(N),target])
    loss = np.sum(correct_logprobs)/N
    return loss

#This function accepts two arguments, the targets (e.g. labels) and predictions.
#It returns the gradient of the average cross entropy loss with respect to the softmax of the
#predictions
def gradAverageCE(target, prediction):
    S = np.apply_along_axis(softmax, 1, prediction)
    S_reciprocal = np.reciprocal(S)
    N = target.shape[0]
    result = (-1.0/N)*np.multiply(target, S_reciprocal)
    result[np.isnan(result)]=0.0
    return result



def trainNN(h, epochs, takeOnlyTestStats=False, gamma=0.9):
    dimw = 784 #number of input nodes
    K = 10 # number of classes

    #initializing the weights and biases following the Xaiver initialization scheme (zero-mean (
    W = (math.sqrt(2/(dimw+h))) * np.random.randn(dimw,h)
    b = np.zeros((1,h))
    W2 =(math.sqrt(2/(h+K))) * np.random.randn(h,K)
    b2 = np.zeros((1,K))

    #initialize the datasets
    trainData, validData, testData, trainTarget, validTarget, testTarget = loadData()
    newtrain, newvalid, newtest = convertOneHot(trainTarget, validTarget, testTarget)
    num_examples = trainTarget.shape[0]
    num_examples_valid = validTarget.shape[0]
    num_examples_test = testTarget.shape[0]
    X = np.zeros((num_examples,dimw))
    Xvalid = np.zeros((num_examples_valid,dimw))
    Xtest = np.zeros((num_examples_test,dimw))

    #initialize the velocity to 1e-5
    vnewW = np.zeros((dimw,h))+1e-5
    vnewW2 = np.zeros((h,K))+1e-5
    vnewb = np.zeros((1,h))+1e-5
    vnewb2 = np.zeros((1,K))+1e-5
    alpha = 1-gamma

    #prepareTables
    trainLoss = np.zeros(epochs)
    trainAccuracy = np.zeros(epochs)
    validationLoss = np.zeros(epochs)
    validationAccuracy = np.zeros(epochs)
```

```python
testLoss = np.zeros(epochs)
testAccuracy = np.zeros(epochs)

for i in range(0,num_examples):
    X[i]=trainData[i].flatten()
for i in range(0,num_examples_valid):
    Xvalid[i]=validData[i].flatten()
for i in range(0,num_examples_test):
    Xtest[i]=testData[i].flatten()


for i in range(epochs):
    #shuffle data at each epoch
    X,trainTarget=shuffle(X,trainTarget)

    #forward propagation
    z = compute(X,W,b);
    X_layer1 = relu(z)
    out = compute(X_layer1,W2,b2)
    prediction=np.argmax(out, axis=1)
    Sk = softmax(out)
    loss = averageCE(trainTarget,Sk)

    if i % 10 == 0:
        print("iteration %d: loss %f" % (i, loss))

    #Calcultate delta outter
    deltaO = Sk
    deltaO[range(num_examples),trainTarget] -= 1
    deltaO /= num_examples

    #Backpropagate to outter layer
    dWo = np.matmul(np.transpose(X_layer1), deltaO)
    dbo = np.sum(deltaO, axis=0, keepdims=True)

    #Calculate delta hidden
    deltah = np.matmul(deltaO, np.transpose(W2))
    # backprop the ReLU non-linearity, effect of multiplying by the derivative of ReLu
    deltah[X_layer1 <= 0] = 0

    #Backpropagate to hidden layer
    dWh = np.dot(X.T, deltah)
    dbh = np.sum(deltah, axis=0, keepdims=True)

    #Update the velocities, weights and biases
    vnewW = gamma*vnewW+alpha*dWh
    vnewb = gamma*vnewb+alpha*dbh
    vnewW2 = gamma*vnewW2+alpha*dWo
    vnewb2 = gamma*vnewb2+alpha*dbo
    W += -vnewW
    b += -vnewb
    W2 += -vnewW2
    b2 += -vnewb2




    trainLoss[i]=loss
    trainAccuracy[i]=np.mean(prediction==trainTarget)
    if(takeOnlyTestStats==False):
        z_valid = compute(Xvalid,W,b);
```

```
            hidden_layer_valid = relu(z_valid)
            scores_valid = compute(hidden_layer_valid,W2,b2)
            prediction_valid=np.argmax(scores_valid, axis=1)
            probs_valid = softmax(scores_valid)
            loss_valid = averageCE(validTarget,probs_valid)
            validationLoss[i]=loss_valid
            validationAccuracy[i]=np.mean(prediction_valid==validTarget)

        z_test = compute(Xtest,W,b);
        hidden_layer_test = relu(z_test)
        scores_test = compute(hidden_layer_test,W2,b2)
        prediction_test=np.argmax(scores_test, axis=1)
        probs_test = softmax(scores_test)
        loss_test = averageCE(testTarget,probs_test)
        testLoss[i]=loss_test
        testAccuracy[i]=np.mean(prediction_test==testTarget)

    return W, b, W2, b2, trainLoss, trainAccuracy, validationLoss, validationAccuracy, testLoss,


def getDataExercise13():
    print("Getting data for exercise 1.3...")
    epochs = 200
    hiddenUnitSize = 1000
    W_hid, b_hid, W_out, b_out, trainLoss, trainAccuracy, validationLoss, validationAccuracy, te
    W_hid2, b_hid2, W_out2, b_out2, trainLoss2, trainAccuracy2, validationLoss2, validationAccu
    with open('exercise13.pkl', 'wb') as f:
        pickle.dump([W_hid, b_hid, W_out, b_out, trainLoss, trainAccuracy, validationLoss, vali
    return

def plotExercise13():
    with open('exercise13.pkl', 'rb') as f:
        _, _, _, _, trainLoss, trainAccuracy, validationLoss, validationAccuracy, testLoss, test

    startIndex = 0
    endIndex = 200
    x = range(startIndex, endIndex)
    plt.title("Losses (gamma = 0.9)")
    plt.plot(x,trainLoss[startIndex:endIndex], '-b', label='TrainData')
    plt.plot(x,validationLoss[startIndex:endIndex], '-r', label='ValidationData')
    plt.plot(x,testLoss[startIndex:endIndex], '-g', label='TestData')
    plt.legend(loc='best')
    plt.xlabel('Epochs')
    plt.show()

    plt.title("Accuracies (gamma = 0.9)")
    plt.plot(x,trainAccuracy[startIndex:endIndex], '-b', label='TrainData')
    plt.plot(x,validationAccuracy[startIndex:endIndex], '-r', label='ValidationData')
    plt.plot(x,testAccuracy[startIndex:endIndex], '-g', label='TestData')
    plt.legend(loc='best')
    plt.xlabel('Epochs')
    plt.show()

    #GAMMA=0.99
    startIndex = 0
    endIndex = 200
    x = range(startIndex, endIndex)
    plt.title("Losses (gamma = 0.99)")
    plt.plot(x,trainLoss2[startIndex:endIndex], '-b', label='TrainData')
    plt.plot(x,validationLoss2[startIndex:endIndex], '-r', label='ValidationData')
    plt.plot(x,testLoss2[startIndex:endIndex], '-g', label='TestData')
```

```
    plt.legend(loc='best')
    plt.xlabel('Epochs')
    plt.show()

    plt.title("Accuracies (gamma = 0.99)")
    plt.plot(x,trainAccuracy2[startIndex:endIndex], '-b', label='TrainData')
    plt.plot(x,validationAccuracy2[startIndex:endIndex], '-r', label='ValidationData')
    plt.plot(x,testAccuracy2[startIndex:endIndex], '-g', label='TestData')
    plt.legend(loc='best')
    plt.xlabel('Epochs')
    plt.show()
    return

def getDataExercise14():
    print("Getting data for exercise 1.4...")
    epochs = 200

    hiddenUnitSize = 100
    _, _, _, _, _, _, _, _, testLoss_1, testAccuracy_1 = trainNN(hiddenUnitSize, epochs, takeOn

    hiddenUnitSize = 500
    _, _, _, _, _, _, _, _, testLoss_2, testAccuracy_2 = trainNN(hiddenUnitSize, epochs, takeOn

    hiddenUnitSize = 2000
    _, _, _, _, _, _, _, _, testLoss_3, testAccuracy_3 = trainNN(hiddenUnitSize, epochs, takeOn
    with open('exercise14.pkl', 'wb') as f:
        pickle.dump([testLoss_1, testAccuracy_1, testLoss_2, testAccuracy_2, testLoss_3, testAcc
    return

def plotExercise14():
    with open('exercise14.pkl', 'rb') as f:
        testLoss_1, testAccuracy_1, testLoss_2, testAccuracy_2, testLoss_3, testAccuracy_3 = pic

    startIndex = 0
    endIndex = 200
    x = range(startIndex, endIndex)
    plt.title("Losses with different amount of hidden units")
    plt.plot(x,testLoss_1[startIndex:endIndex], '-b', label='100 hidden units')
    plt.plot(x,testLoss_2[startIndex:endIndex], '-r', label='500 hidden units')
    plt.plot(x,testLoss_3[startIndex:endIndex], '-g', label='2000 hidden units')
    plt.legend(loc='best')
    plt.xlabel('Epochs')
    plt.show()

    plt.title("Accuracies with different amount of hidden units")
    plt.plot(x,testAccuracy_1[startIndex:endIndex], '-b', label='100 hidden units')
    plt.plot(x,testAccuracy_2[startIndex:endIndex], '-r', label='500 hidden units')
    plt.plot(x,testAccuracy_3[startIndex:endIndex], '-g', label='2000 hidden units')
    plt.legend(loc='best')
    plt.xlabel('Epochs')
    plt.show()
    return

def checkSomeValues(start, end):
    if(start<0 or end >10000):
        print("Incorrect bounds")
        return
    with open('exercise13.pkl','rb') as f:
           W_hid, b_hid, W_out, b_out, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _ = pickle.l
    trainData, validData, testData, trainTarget, validTarget, testTarget = loadData()
    X = np.zeros((10000,784))
```

```
    for i in range(0,10000):
            X[i]=trainData[i].flatten()
    for i in range(start,end):

        print("Image: ", i ,": \n")
        plt.imshow(trainData[i], cmap="gray")
        plt.show()
        z = compute(X[i],W_hid,b_hid);
        hidden_layer = relu(z)
        scores = compute(hidden_layer,W_out,b_out)
        probs = softmax(scores)
        print("Prediction : ", probs)
        x=np.array(['A','B','C','D','E','F','G','H','I','J'])
        print("Predicted value :", x[np.argmax(probs)])
        print("Real value : ", x[trainTarget[i]])

getDataExercise13()
getDataExercise14()

plotExercise13()
plotExercise14()

checkSomeValues(1, 10)
```

```python
# -*- coding: utf-8 -*-
"""
Created on Mon Mar 11 16:02:33 2019
@author: anthonykemmeugne
@author: AlexanderApostolov
"""

import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
import pickle
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'

# Load the data
def loadData():
    with np.load("notMNIST.npz") as data:
        Data, Target = data["images"], data["labels"]
        np.random.seed(521)
        randIndx = np.arange(len(Data))
        np.random.shuffle(randIndx)
        Data = Data[randIndx] / 255.0
        Target = Target[randIndx]
        trainData, trainTarget = Data[:10000], Target[:10000]
        validData, validTarget = Data[10000:16000], Target[10000:16000]
        testData, testTarget = Data[16000:], Target[16000:]
    return trainData, validData, testData, trainTarget, validTarget, testTarget

def convertOneHot(trainTarget, validTarget, testTarget):
    newtrain = np.zeros((trainTarget.shape[0], 10))
    newvalid = np.zeros((validTarget.shape[0], 10))
    newtest = np.zeros((testTarget.shape[0], 10))

    for item in range(0, trainTarget.shape[0]):
        newtrain[item][trainTarget[item]] = 1
    for item in range(0, validTarget.shape[0]):
        newvalid[item][validTarget[item]] = 1
    for item in range(0, testTarget.shape[0]):
        newtest[item][testTarget[item]] = 1
    return newtrain, newvalid, newtest

#Shuffles the train Data (to be used at the end of each epoch)
def shuffle(trainData, trainTarget):
    np.random.seed(421)
    randIndx = np.arange(len(trainData))
    target = trainTarget
    np.random.shuffle(randIndx)
    data, target = trainData[randIndx], target[randIndx]
    return data, target

def cnn(learning_rate, epochs=50, batch_size=32, L2_loss=False, regularization=0.0, dropLayer=Fa
    tf.set_random_seed(421)
    #Xavier initializer
    initializer = tf.contrib.layers.xavier_initializer()

    wc= tf.Variable(initializer([3,3,1,32]))
    bc =tf.Variable(initializer([32]))
    #zero-mean Gaussians initialization with variance 2/unitsin+unitsout
    sig1 = np.sqrt(2/((14*14*32)+784))
    sig2 = np.sqrt(2/(784+10))
    w1 = tf.Variable(tf.random_normal([14*14*32, 784], stddev = sig1), name='w1')
```

```
b1 = tf.Variable(tf.random_normal([784] ,stddev = sig1), name = 'b1')

w2 = tf.Variable(tf.random_normal([784, 10],stddev = sig2), name='w2')
b2 = tf.Variable(tf.random_normal([10], stddev = sig2),  name = 'b2')


x = tf.placeholder(tf.float32, shape=(None, 784), name='x')
y = tf.placeholder(tf.float32, shape=(None, 10), name='y')
reg = tf.placeholder(tf.float32, name='reg')

#1. Input Layer
input_layer = tf.reshape(x, shape=[-1, 28, 28, 1])

#2. Convolutional layer
conv = tf.nn.conv2d(input=input_layer, filter=wc, strides=[1,1,1,1], padding="SAME")
conv = tf.nn.bias_add(conv, bc)

#3. Relu activation
relu1 = tf.nn.relu(conv)

#4. Batch normalization layer

batch_mean, batch_var = tf.nn.moments(relu1,[0])
normal = tf.nn.batch_normalization(relu1, batch_mean, batch_var, offset = None, scale = None

#5 A 2 × 2 max pooling layer

maxpool = tf.nn.max_pool(normal, ksize=[1,2,2,1], strides=[1,2,2,1], padding="SAME")

#6 Flatten layer

flat = tf.reshape(maxpool, [-1, 14*14*32])

#7. Fully connected layer (with 784 output units, i.e. corresponding to each pixel)

full = tf.add(tf.matmul(flat, w1), b1)

#8.Dropout if needed + ReLU activation

if(dropLayer):
    drop_layer = tf.nn.dropout(full, keep_prob=keep)
    relu2 = tf.nn.relu(drop_layer)
else:
    relu2=tf.nn.relu(full)

#9. Fully connected layer (with 10 output units, i.e. corresponding to each class)

out = tf.add(tf.matmul(relu2, w2), b2)

#10. Softmax output

softmax_layer = tf.nn.softmax(out)

#11. Cross Entropy loss

if(L2_loss):
    loss_op = (tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(
    logits=out, labels=y)) +
    reg*tf.nn.l2_loss(wc) +
    reg*tf.nn.l2_loss(w1) +
    reg*tf.nn.l2_loss(w2))
```

```
else:
    loss_op = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(logits=out, labels=y

optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
train_op = optimizer.minimize(loss_op)

correct_pred = tf.equal(tf.argmax(softmax_layer, 1), tf.argmax(y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))

init = tf.global_variables_initializer()

dimw = 784
#initialize the datasets
trainData, validData, testData, trainTarget, validTarget, testTarget = loadData()
newtrain, newvalid, newtest = convertOneHot(trainTarget, validTarget, testTarget)
num_examples = trainTarget.shape[0]
num_examples_valid = validTarget.shape[0]
num_examples_test = testTarget.shape[0]
X = np.zeros((num_examples,dimw))
Xvalid = np.zeros((num_examples_valid,dimw))
Xtest = np.zeros((num_examples_test,dimw))
for i in range(0,num_examples):
    X[i]=trainData[i].flatten()
for i in range(0,num_examples_valid):
    Xvalid[i]=validData[i].flatten()
for i in range(0,num_examples_test):
    Xtest[i]=testData[i].flatten()


#prepareTables
trainLoss = np.zeros(epochs)
trainAccuracy = np.zeros(epochs)
validationLoss = np.zeros(epochs)
validationAccuracy = np.zeros(epochs)
testLoss = np.zeros(epochs)
testAccuracy = np.zeros(epochs)


with tf.Session() as sess:
    sess.run(init)
    number_of_batches = num_examples//batch_size

    for step in range(0, epochs):
        #Shuffle after each epcoh
        flat_x_shuffled,trainingLabels_shuffled = shuffle(X, newtrain)

        for minibatch_index in range(0,number_of_batches):
            #select miniatch and run optimizer
            minibatch_x = flat_x_shuffled[minibatch_index*batch_size: (minibatch_index + 1)
            minibatch_y = trainingLabels_shuffled[minibatch_index*batch_size: (minibatch_in
            sess.run(train_op, feed_dict={x: minibatch_x, y: minibatch_y, reg: regularizati

        if(step==epochs-1 or not(onlyFinal)):
            lossTrain, accTrain = sess.run([loss_op, accuracy], feed_dict={x: flat_x_shuffl
            lossValid, accValid = sess.run([loss_op, accuracy], feed_dict={x: Xvalid, y: ne
            lossTest, accTest = sess.run([loss_op, accuracy], feed_dict={x: Xtest, y: newte
            trainLoss[step]=lossTrain
            trainAccuracy[step]=accTrain
            validationLoss[step]=lossValid
            validationAccuracy[step]=accValid
            testLoss[step]=lossTest
```

```
                testAccuracy[step]=accTest
                print("Step " + str(step+1) + ", Train Loss= " + \
                              "{:.8f}".format(lossTrain) + ", Training Accuracy= " + \
                              "{:.8f}".format(accTrain))
            else:
                print("Step " + str(step+1) + " (no data, only final losses and accuracies will
        print("Optimization Finished!")
    return trainLoss, trainAccuracy, validationLoss, validationAccuracy, testLoss, testAccuracy

def getDataExercise22():
    print("Getting data for exercise 2.2...")
    trainLoss, trainAccuracy, validationLoss, validationAccuracy, testLoss, testAccuracy = cnn(
    with open('exercise22.pkl', 'wb') as f:
        pickle.dump([trainLoss, trainAccuracy, validationLoss, validationAccuracy, testLoss, te
    return

def getDataExercise231():
    print("Getting data for exercise 2.3.1...")
    _, trainAccuracy1, _, validationAccuracy1, _, testAccuracy1 = cnn(1e-4, L2_loss=True, regul
    _, trainAccuracy2, _, validationAccuracy2, _, testAccuracy2 = cnn(1e-4, L2_loss=True, regul
    _, trainAccuracy3, _, validationAccuracy3, _, testAccuracy3 = cnn(1e-4, L2_loss=True, regul
    with open('exercise231.pkl', 'wb') as f:
        pickle.dump([trainAccuracy1, trainAccuracy2, trainAccuracy3, validationAccuracy1, valid
    return

def getDataExercise232():
    print("Getting data for exercise 2.3.2...")
    _, trainAccuracy1, _, validationAccuracy1, _, testAccuracy1 = cnn(1e-4, dropLayer=True, keep
    _, trainAccuracy2, _, validationAccuracy2, _, testAccuracy2 = cnn(1e-4, dropLayer=True, keep
    _, trainAccuracy3, _, validationAccuracy3, _, testAccuracy3 = cnn(1e-4, dropLayer=True, keep
    with open('exercise232.pkl', 'wb') as f:
        pickle.dump([trainAccuracy1, trainAccuracy2, trainAccuracy3, validationAccuracy1, valid
    return

def plotExercise22():
    with open('exercise22.pkl', 'rb') as f:
        trainLoss, trainAccuracy, validationLoss, validationAccuracy, testLoss, testAccuracy = p

    startIndex = 0
    endIndex = 50
    x = range(startIndex, endIndex)
    plt.title("Losses")
    plt.plot(x,trainLoss[startIndex:endIndex], '-b', label='TrainData')
    plt.plot(x,validationLoss[startIndex:endIndex], '-r', label='ValidationData')
    plt.plot(x,testLoss[startIndex:endIndex], '-g', label='TestData')
    plt.legend(loc='best')
    plt.xlabel('Epochs')
    plt.show()

    plt.title("Accuracies")
    plt.plot(x,trainAccuracy[startIndex:endIndex], '-b', label='TrainData')
    plt.plot(x,validationAccuracy[startIndex:endIndex], '-r', label='ValidationData')
    plt.plot(x,testAccuracy[startIndex:endIndex], '-g', label='TestData')
    plt.legend(loc='best')
    plt.xlabel('Epochs')
    plt.show()
    return

def printExercise231():
    ##getting all the accuracies
    with open('exercise231.pkl', 'rb') as f:
```

```
        tr1, tr2, tr3, v1, v2, v3, te1, te2, te3 = pickle.load(f)
    print("data\treg=0.01\t\treg=0.1\t\t\treg=0.5")
    print("train\t"+str(tr1[-1])+"\t"+str(tr2[-1])+"\t"+str(tr3[-1]))
    print("valid\t"+str(v1[-1])+"\t"+str(v2[-1])+"\t"+str(v3[-1]))
    print("test\t"+str(te1[-1])+"\t"+str(te2[-1])+"\t"+str(te3[-1]))
    return

def plotExercise232():
    with open('exercise232.pkl', 'rb') as f:
        trainAccuracy1, trainAccuracy2, trainAccuracy3, validationAccuracy1, validationAccuracy2
    startIndex = 0
    endIndex = 50
    x = range(startIndex, endIndex)

    plt.title("Accuracies with p=0.9")
    plt.plot(x,trainAccuracy1[startIndex:endIndex], '-b', label='TrainData')
    plt.plot(x,validationAccuracy1[startIndex:endIndex], '-r', label='ValidationData')
    plt.plot(x,testAccuracy1[startIndex:endIndex], '-g', label='TestData')
    plt.legend(loc='best')
    plt.xlabel('Epochs')
    plt.show()

    plt.title("Accuracies with p=0.75")
    plt.plot(x,trainAccuracy2[startIndex:endIndex], '-b', label='TrainData')
    plt.plot(x,validationAccuracy2[startIndex:endIndex], '-r', label='ValidationData')
    plt.plot(x,testAccuracy2[startIndex:endIndex], '-g', label='TestData')
    plt.legend(loc='best')
    plt.xlabel('Epochs')
    plt.show()

    plt.title("Accuracies with p=0.5")
    plt.plot(x,trainAccuracy3[startIndex:endIndex], '-b', label='TrainData')
    plt.plot(x,validationAccuracy3[startIndex:endIndex], '-r', label='ValidationData')
    plt.plot(x,testAccuracy3[startIndex:endIndex], '-g', label='TestData')
    plt.legend(loc='best')
    plt.xlabel('Epochs')
    plt.show()
    return

getDataExercise22()
getDataExercise231()
getDataExercise232()

plotExercise22()
printExercise231()
plotExercise232()
```