COGS 260: Assignment 3

Kriti Aggarwal (PID: A53214465)
by Department of Computer Science and Engineering
University of California, San Diego
San Diego, CA 92093
kriti@eng.ucsd.edu

Abstract

In this assignment we explore the three generations of neural networks: Perceptron learning, Feed-forward Networks and Convolutional Networks. We experiment with single layer perceptron which is the simplest feedforward neural network, a 2 layered feed forward network and moving to a 10 layered complex convolution neural network. For each of the networks, we experiment with a different dataset: Iris flower dataset, MNIST dataset, CIFAR10 dataset.

1 Description of the methods and Experimental Results

1.1 Perceptron Learning

Single layer perceptron is the simplest feedforward neural network, which aims at learning a binary classifier. Given a set of points assigned binary labels, the perceptron algorithm aims at finding a hyperplane which can separate the points by class. To do this, the algorithm uses gradient descent by updating the weights per training sample in the direction of reducing error.

1.1.1 Plotting Pairwise attributes for Setosa and Versicolor

In this section, we took all the 16 combinations of the 4 features (sepal width, sepal length, petal length and petal width), represented them in 2D space and plotted for each of the flower. This helped us to identify that the 2 flower species are linearly separable in each of the plots.

1.1.2 Training a Perceptron

For this experiment, we wrote a program in python for training a Perceptron for classification. We used learning rate as 0.1, and initial small random weights. We were able to achieve 100% accuracy on test set in just 3 epochs. The weights and test accuracy has been shown for 6 epochs in figure 2. This can be attributed to the fact that the data is linearly separable as can be seen by the plots above.

1.1.3 Z scoring the data

We performed Z-scoring of the train and test data. We found that the convergence happened much faster in just 1 epoch with the learning rate of 0.4 as shown in figure 3. This can be explained by the fact that Z-scoring normalizes each of the features which reduces the spread of the features to a more concentrated space. In case of unnormalized data, the error surface is elongated while normalizing makes the error surface more circular thus helping Gradient Descent to reach convergence faster.

1.1.4 Result

We found that the Iris flower dataset was easily linearly separable. We were able to train Perceptron classifier with 100% test accuracy in just 3 epochs to separate the Setosa and Versicolor classes.

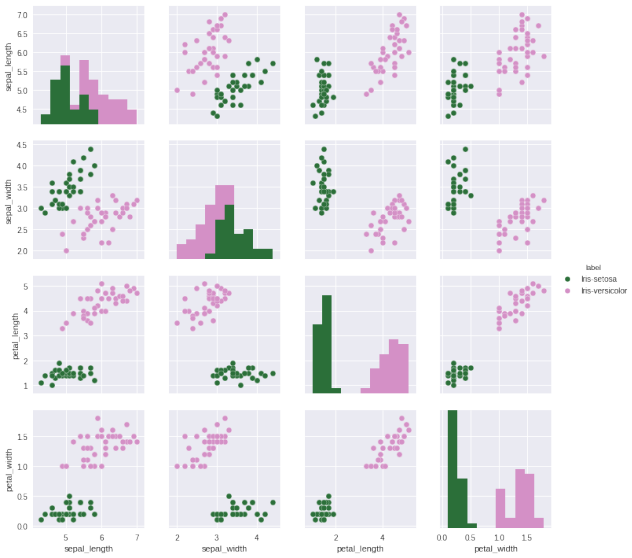

Figure 1: Pairwise feature plots for Setosa and Versicolor

```
Epoch:0
weights: [0.5719898042840076 -0.6281475363442173 0.5693249557574183
 0.010644785236553739 -0.059264395025021941
Test Error Percentage:50.0%

Epoch:1
weights: [0.7619898042840076 -0.6581475363442173 0.8993249557574183
 0.13064478523655537 -0.059264395025021941
Test Error Percentage:50.0%

Epoch:2
weights: [0.25198980428400763 -1.0081475363442174 0.7593249557574183
 0.11064478523655537 -0.15926439502502194]
Test Error Percentage:0.0%

Epoch:3
weights: [0.25198980428400763 -1.0081475363442174 0.7593249557574183
 0.11064478523655537 -0.15926439502502194]
Test Error Percentage:0.0%

Epoch:4
weights: [0.25198980428400763 -1.0081475363442174 0.7593249557574183
 0.11064478523655537 -0.15926439502502194]
Test Error Percentage:0.0%
```

Figure 2: Perceptron test accuracy after every epoch

1.2 Feed Forward Neural Network

Perceptron classifiers perform well when the data is linearly separable. A very simple example where it does not work is XOR which is not linearly separable. For this, neural networks come to the rescue, where backpropagation learns the internal representation of the system.

The feedforward propagation basically means that the information (activation) moves in only one direction, forward, from the input nodes, through the hidden nodes and to the output nodes.

After the activations are propagated forward in the network, we perform backpropagation. In this algorithm, the output of the network is compared to the desired output using cross-entropy cost function. After that an error value is calculated for each of the neurons in the output layer. The error values are then propagated backwards, starting from the output, until each neuron has an associated error value which roughly represents its contribution to the original output.

Backpropagation then uses these error values to calculate the gradient of the loss function with respect to the weights in the network. Finally, this gradient is fed to the "Gradient Descent" learning algorithm which in turn update the weights so as to minimize the cross entropy cost function.

1.2.1 Update equations in vectorized form

In vectorized form, we could represent our inputs as a matrix of dimension N x I, where N are the number of examples and I are the number of units input layer. The weights for input to hidden layer can be represented with a matrix W_{ij} of dimension I x J, where I are the number of units in the input layer and J are the number of units in the hidden layer. Similarly, the weights for hidden to output

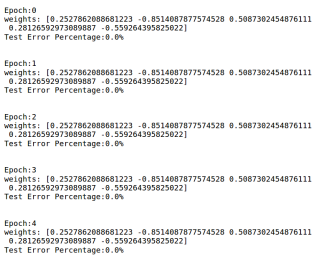

Figure 3: Perceptron test accuracy after every epoch on normalized train and test data

layer can be represented with a matrix W_{jk} of dimension J x C, where C is the number of classes. We can take a list of matrices W that can store all the weights parameters.

For each layer, the output could be represented with a vector Z^l. Then, the weighted sum $W^l Z^l$ can be stored in A^l and Z^{l+1} can be calculated as f(A^l) where f(.) is an appropriate activation function. Using the above formulations, we write our update rules as:

$W^l = W^l + \eta(\delta^l(Z^l)T)$, where η is the learning rate.

For hidden layer, delta or derivative of the Error with respect to the weight is defined as: $\delta^l = (W^l)^T(\delta^{l+1}) . * \sigma'(Z^l)$

where .* is an element wise product operator,and σ' is the derivative of the activation function in later l.

For output layer: $\delta^l = T - Y$

where T is the true output label and Y is the predicted output.

1.2.2 Tricks of trade

There are a number of tricks that can be used to optimize vanilla SGD. We experimented with the following optimization tricks.

- *Shuffling the examples*: Since the network learns the fastest when presented with the most unexpected sample. Hence, it makes sense to shuffle the input samples before performing SGD. We found that for mini batch gradient descent, the best results were obtained if each mini batch contained samples from all the class labels.

  Hence, we shuffled all the examples before performing SGD.
- *Stochastic mini batch gradient descent*: In case the data set is redundant and huge, stochastic gradient descent makes more sense. Since, as opposed to batch learning, stochastic gradient descent changes the weights after computing gradient for every sample. Instead of changing the weights for every sample, we use mini batches after which we perform the weight updates. Stochastic gradient learning is the method of choice, since it is usually much faster than batch learning, leads to better solutions and is advantageous in tracking changes.

  Considering that the size of MNIST is 60,000 and the above advantages of mini batch gradient descent, we used mini batch gradient descent with batch size of 128.
- *Normalization and shifting the inputs:* When using steepest descent, it usually helps to transform each input component so that it has zero mean over the entire training set. Scaling the inputs, so that each component of the input vector has unit variance over the whole training set. This helps because if the inputs vary a lot in their magnitude the error surface in which there is high curvature in the direction where input component is low and vice versa. Hence, if scale the input, the error surface is more circular or the curvature is almost the same in all the directions. Hence, we normalized all the images before performing SGD.
- *Comparison of various activation functions* Softmax regression (or multinomial logistic regression) is a generalization of logistic regression to the case where we want to handle

multiple classes. The tanh(z) function is a rescaled version of the sigmoid, and its output range is [1,1][1,1] instead of [0,1][0,1]. Hence, using tanh(z) helps in preserving all the outputs in the range [-1,1] that is the standard deviation of the outputs is still preserved to be unit and the mean is roughly zero. That is when the inputs have mean zero and unit standard deviation.

Hence, we used funny tanh as our activation function in our hidden layer (funny tanh performs better than tanh empirically as mentioned in the paper Efficient BackProp by Yan Lecun).

- *Initialization of the input weights* If two hidden units have the same bias and exactly the same incoming and outgoing weights, then they will always get the same gradient. Hence, they would never learn different features. To break this symmetry, we can initialize the weights with small random values. If a hidden unit has a big fan-in, small changes on many of its incoming weights can cause the learning to overshoot. We generally want smaller incoming weights when the fan-in is big, so initialize the weights to be proportional to square root of the fan in. This helps as if the fan in large, the weights are smaller and vice versa. We can also scale the learning rates according to the fan in.

  Hence, we initialized the input weights by small random values and dividing these values by square root of the fan in.

- *Learning rate* Learning rate plays an important role while training a neural network. Learning rate needs to be neither too small (to prevent slow training) and neither too large(to prevent oscillations).

  Annealing is a method of decreasing the learning rate as gradient descent approaches minima in the error surface. This helps in the descent to take bigger steps when its away from minima and smaller when its near the trough.

  We used annealing with learning rate decay as 1e-6.

- *Momentum* If we imagine a ball on the error surface, the location of the ball in the horizontal plane represents the weight vector. We dont use gradient to change the position of the weight as done in the standard method. We use gradient to change the acceleration of the weights and use velocity to change the position of the weight. The reason that it is different is because the weights have momentum that is they remember the last change in their weights.

  What momentum does is that it damps the oscillations in the direction of high curvature. Hence, we used momentum to accelerate our SGD.

1.3 Experiments

We implemented a one layer feed forward network in python (the implementation can be found in appendix) with 80 hidden units, funny tanh as the activation function, and optimization tricks as mentioned in detail in the above section. We performed mini batch SGD with a batch size of 50 samples. We used learning rate as 0.01/(batch size) We were able to achieve an accuracy of 96.76% with 80 hidden units with vanilla SGD. The figure 4 shows the train and test accuracy plot for 80 hidden units.

1.3.1 Number of hidden units

We also experimented with different hidden units and found the following results:

1.3.2 Number of hidden layers

No. of hidden units	Test accuracy	Train accuracy
5	76%	78%
80	96.71%	98.5%
150	96%	98.4%
300	97.3%	98.2%
600	96.58%	99.58%

We found that reducing the units to just 5 units degraded the performance to just 78% which can be attributed to underfitting. While 600 hidden units led to overfitting with 99.58% training accuracy

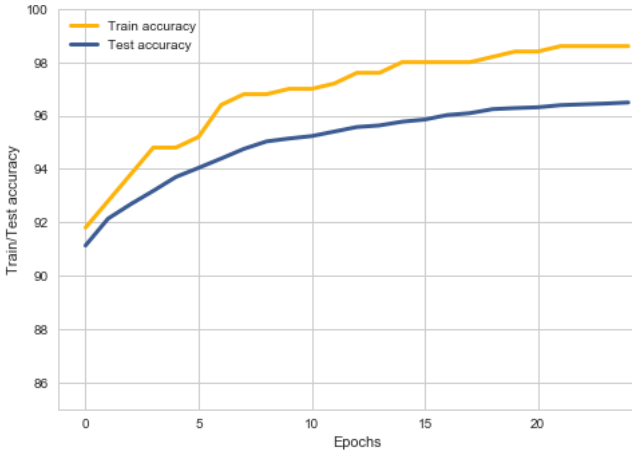

Figure 4: Training and test accuracy with 80 hidden units

and 96.58% test accuracy. The units 80 performed well, but we got the best performance at 300 hidden units. Though there is not a considerable difference in the performance with 80 and 300 hidden units. We conclude that 80 units could capture the internal representation in MNIST data well.

1.3.3 Number of hidden layers

Hidden layers and units	Test accuracy	Train accuracy
1 hidden layer 80 units	96.71%	98.5%
2 hidden layers 80 and 50 units	97.4%	99.6%

We found that increasing the number of hidden layers slightly improved the performance as shown in the figure 5 with training accuracy as 99.6% and test accuracy as 97.4%. Though we can see slight overfitting at the end. Increasing the number of layers, increases the number of parameters and helps in fitting the data well which helped in increasing the accuracy a bit but also led to overfitting.

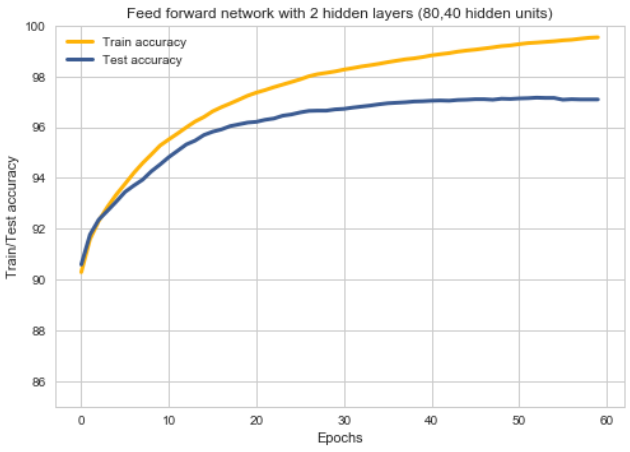

Figure 5: Training and test accuracy with 2 layers

1.3.4 Momentum

We have described momentum and nesterov momentum We used Nesterov momentum with α 0.9 after experimenting with several values of α including 0.85, 0.9,0.95.

The major advantage of Nesterov momentum was the convergence speed, which was achieved within 15 epochs, which took close to 20 epochs in vanilla SGD. Also, the test accuracy jumps to 92% in the first epoch with Nesterov momentum which was not the case with vanilla SGD. We performed mini batch SGD and the change in training and test accuracy is as shown in figure 6. We achieved an accuracy of 97.5% on the test set using this method.

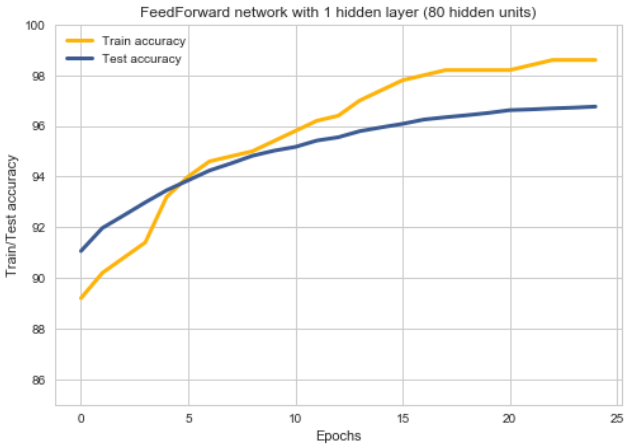

Figure 6: Training and test accuracy with Nesterov momentum SGD

1.3.5 Regularization

Since, with earlier techniques using 2 hidden layers, we achieved accuracy of 97.4% on test data but we got 99.6% accuracy on the training data showing overfitting.

Regularization is the way to reduce the overfitting of the model by penalizing the weights. Hence, we experimented with L2 regularization with various lambda values. We found that setting a high value of the regularization parameter degraded the test accuracy and also train accuracy causing underfitting. Hence, we used a very small regularization parameter 1e-4 with L2 regularization and we were able to get 97.8% accuracy on test set and 99.01% accuracy on training set using batch GD. The figure 7 shows batch Gradient descent with regularization and momentum and other optimization techniques. We performed 1000 iterations and as can be seen the model is still not overfitting because of the regularization parameter used.

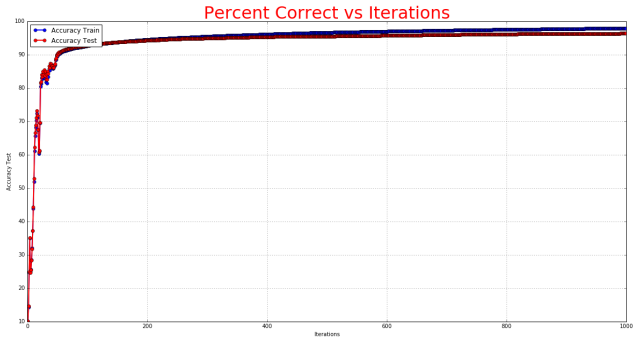

Figure 7: Batch GD with regularization and other tricks of trade

1.3.6 Result:

We explore the use of multi-layer neural networks for the task of handwritten digit recognition. MNIST is a dataset of handwritten digits, originally comprising of 60000 training examples and 10000 test examples. However, for the experiment we conducted, we sampled 50,000 training examples and used the rest 10,000 examples as the validation set. Working with the feedforward propagation and backpropagation algorithm, we built a classifier that maps each input data to one of the labels t0,1,2,3,4,5,6,7,8,9. We used both batch gradient and stochastic gradient descent techniques for optimizing on the Cost Function along with early stopping. We also applied various "Tricks of the trade" techniques in order to speed up the optimization. Without using 'tricks of the trade' we achieved an accuracy of 96.7% on the test set, and 97.868% on the training set. Subsequently, after applying 'tricks of the trade' techniques, we achieved an accuracy of 97.8% on test set and 99.108% on training set.

1.4 Convolutional Neural Network

ConvNets derive their name from the convolution operator. The primary purpose of Convolution in case of a ConvNet is to extract features from the input image. Convolution preserves the spatial relationships between pixels by learning image features using small squares of input data.

Here are the basic layers that comprise a convolution neural network:

- *Convolution layer*: Convolutional layers consist of a rectangular grid of neurons. It requires that the previous layer also be a rectangular grid of neurons. Each neuron takes inputs from a rectangular section of the previous layer; the weights for this rectangular section are the same for each neuron in the convolutional layer.
- *Pooling layer*: Pooling layer is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting. In case of Max Pooling, we define a spatial neighborhood (for example, a 2x2 window) and take the largest element/average of all elements from the rectified feature map within that window. Pooling layer also makes the network invariant to small transformations, distortions and translations in the input image by representing a window of pixel by only a single value.
- *Fully connected Layer*: A fully connected layer takes all neurons in the previous layer and connects it to every single neuron it has.

1.4.1 The structure of the Convolutional Neural Network adopted in this assignment

- Convolutional layer with 32 feature maps of size 3x3.
- Convolutional layer with 32 feature maps of size 3x3.
- Pooling layer taking the max over 2x2 patches.
- Dropout layer with a probability of 20%.
- Convolutional layer with 64 feature maps of size 3x3.
- Convolutional layer with 64 feature maps of size 3x3.
- Pooling layer taking the max over 2x2 patches.
- Flatten layer.
- Fully connected layer with 512 neurons and rectifier activation.
- Dropout layer with a probability of 50%.
- Softmax Output layer.

The network architecture is inspired by VGGNet in which every 2 convolutional layers are followed by a max pooling layer with 2 fully connected layers at the end and same filter sizes for covolutions and pooling in the entire network. Though we made various modifications in our architecture by using additional dropout layers, different number of covolutional layers,etc.. Also, we experimented with using 2 convolution layers before a max pool layer since that helps in a better receptive field before the pooling layer.

For all the experiments on convolution neural layer, we used Keras and implemented our own code which can be found in the appendix section. Because of resources and time constraints we have performed all our experiments on a smaller dataset with train:test split of 4000:1000 with 30 epochs. We then use the results of these experiments and use them on the complete dataset.

1.4.2 SGD

In Stochastic Gradient Descent we perform weight update after every sample. In this experiment, we used vanilla SGD and got accuracy as 40%. The parameters used:

Parameter	Value
learning rate	0.0001
learning rate decay	1e-6
batch size	32
momentum	0
optimizer	Plain SGD

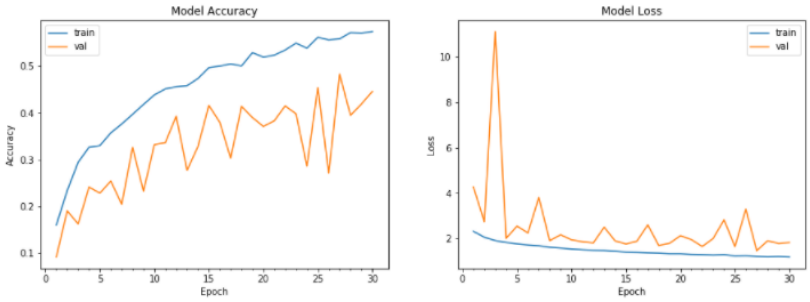

Figure 8: Accuracy and Loss plots for SGD

We plotted the test and training accuracy and loss on the dataset and got the results as shown in figure 8. The loss on test set is highly inconsistent where it decreases and increases with each epoch with a overall decrease in the test loss. While the train loss smoothly decreases. Also, the loss is still decreasing at the end of 30 epochs for both train and test set showing that the model can learn more with more epochs.

We get a test and train accuracy of 41.2% and 61.3% where the accuracy of the test plot is very erratic increasing and decreasing between epochs while for the training set, the accuracy consistently increases. The sporadicity can be attributed to a number of factors including : mis-configured learning rate, unnormalized inputs in the middle layers of the network, etc.

We try to improve on the accuracy, convergence and performance using the methods/experiments below.

1.4.3 Batch Normalization(BN)

The importance of normalization has been discussed in detail in the previous section. Though, normalizing the inputs in the beginning helps in improving the performance, one of the problems of bad performance of deep networks, is the loss of normalization of the deeper layers in the network. One way to deal with this is to use normalization layers so that the input is 0 centered and in linear section of activation functions like sigmoid. Though, there are cases, when the system's performance degrades with such normalization. Hence, we experimented with batch normalization which learns the normalization parameters through back propagation.

Considering the properties of BN, it makes sense to use it before the input of non-linear functions. We found that the use of 3 batch normalization layers gave good results, one after every convolution layer and one before the fully connected layer. This could be because the inputs to these layers were getting unnormalized while training was done.

We were able to achieve an improvement of 11% test accuracy with 52% test accuracy as shown in figure9. Also, the sporadicity in the train and test loss is lesser as compared to the SGD method because of the normalization of the inputs to the Fully connected layer. Also, batch normalization helps in giving a higher learning rate and we gave 0.1 learning rate and were still able to achieve good results.

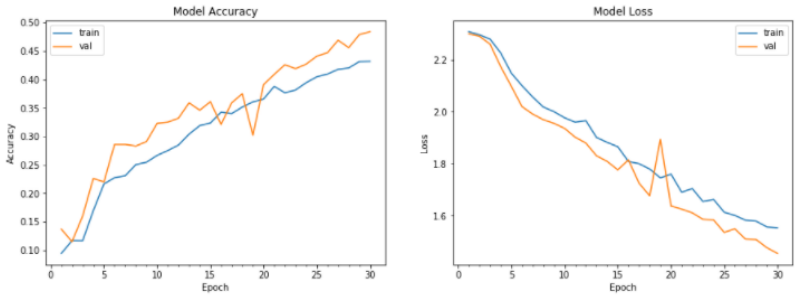

Figure 9: Accuracy and Loss plots for SGD with Batch Normalization

1.4.4 Replacing last layer by global average pooling layer

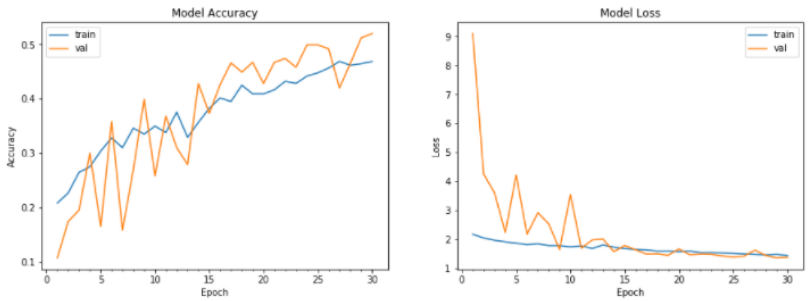

Figure 10: Accuracy and Loss plots for SGD with global average pooling layer

In this experiment, instead of adopting the traditional fully connected layers for classification in CNN, we directly output the spatial average of the feature maps from the last convolution layer as the confidence of categories via a global average pooling layer, and then the resulting vector is fed into the softmax layer. In traditional CNN, it is difficult to interpret how the category level information from the objective cost layer is passed back to the previous convolution layer due to the fully connected layers which act as a black box in between. In contrast, global average pooling is more meaningful and interpretable as it enforces correspondence between feature maps and categories, which is made possible by a stronger local modeling using the micro network. Furthermore, the fully connected layers are prone to overfitting and heavily depend on dropout regularization, while global average pooling is itself a structural regularizer, which natively prevents overfitting for the overall structure.

For this experiment, we removed the last fully connected layer and added a convolution layer with 10 nodes with softmax activation and used a global average pooling layer on top of it. We achieved test accuracy of 52%, which was 10% more than the SGD. The test accuracy was very sporadic as compared to the training accuracy. The major point to note here is the training and test loss, which decreases exponentially as compared to other methods where there is a small decay of loss.

1.4.5 Adaptive Gradient

Adagrad is an algoithm for gradient-based optimization that adapts the learning rate to the parameters, performing larger updates for infrequent and smaller updates for frequent parameters. But

Adagrad's main weakness is its accumulation of the squared gradients in the denominator, Since every added term is positive, the accumulated sum keeps growing during training. This in turn causes the learning rate to shrink and eventually become infinitesimally small.

We used the following parameters for this experiment:

Parameter	Value
learning rate	0.01
learning rate decay	exponential, 1e-8
batch size	32
momentum	0
optimizer	AdaGrad

We were able to get an accuracy of approximately 61% using this technique on this section of the dataset. Both the testing and training accuracy increases with epochs and the model has not reached convergence with 30 epochs.

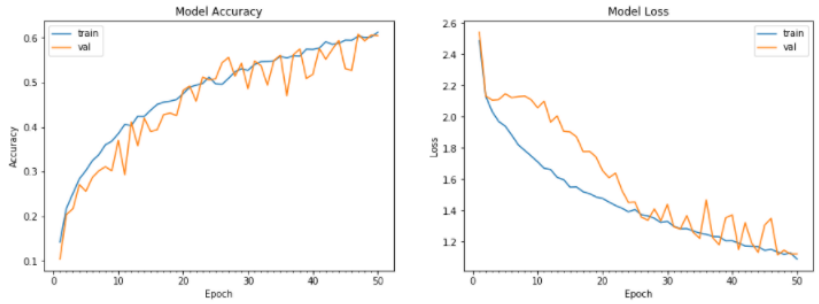

Figure 11: Accuracy and Loss plots for SGD with Adaptive Gradient

1.4.6 Nesterov Accelerated Gradient

Momentum is a method that helps accelerate SGD in the relevant direction and dampens oscillations. Nesterov Momentum is a slightly different version of the momentum update in which we maintain a running average of the past momentums. This has been discussed in detail in the previous section.

Parameter	Value
learning rate	0.001
learning rate decay	exponential, 1e-4
batch size	32
momentum	0.9, Nesterov
optimizer	None

We were able to achieve an improvement of approximately 20% on the test accuracy more than vanilla SGD with this approach, getting a total accuracy of 55%. This can be attributed to how nesterov momentum helps in moving the weights in the direction of consistent gradients.

1.4.7 RMS Prop

AdaDelta improves upon AdaGrad by restricting the past accumulated gradients to a fixed sized window. RMSProp is quite similar to AdaDelta except that it also divides the learning rate by a decaying average of squared gradients.

We used the following parameters:

Parameter	Value
learning rate	0.0001
learning rate decay	exponential, 1e-6
batch size	32
momentum	0
optimizer	RMSProp

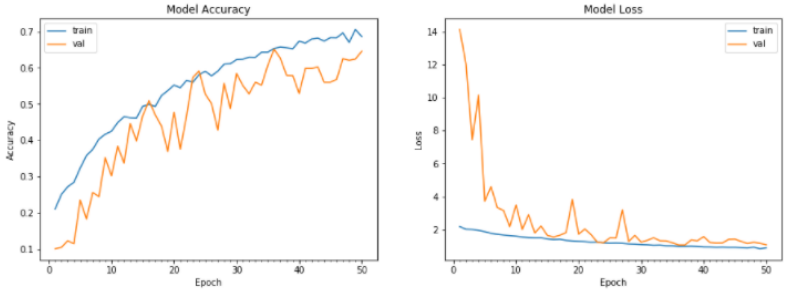

Figure 12: Accuracy and Loss plots for SGD with rmsprop optimizer

Hence, the performance of RMS Prop is slightly better to that of AdaGrad on this subsection of dataset with 63% accuracy on the test set. Here too, the fall in loss is considerable in the beginning epochs as compared to other methods.

1.4.8 Result

We performed various experiments on top of vanilla SGD and various optimizers which lead to interesting results. We found that the evil of learning rate and decay can be eliminated by using optimizers such as AdaGrad, RMSProp etc. RMS PRop gave the best accuracy on this dataset with 63% test accuracy, closely followed by AdaGrad at 61% test accuracy and Nesterov Momentum layer with 55% accuracy.

2 Conclusion

We analyzed 3 datasets and used different models on each of them. The complexity and sophistication of both the images and the networks increased as we advanced from one dataset to the next. Perceptron classifier is a simple classifier which can classify linearly separable classes easily. Neural networks come to rescue to capture complex non linearities in the data. While the convolution neural networks are good at capturing complex features in image data.

3 References

$Network in Network by Min Lin, Qiang Chen, Shuicheng Yan \quad https://www.cs.nyu.edu/fergus/papers/zeilerECCV2014.pdf \quad http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf$

4 Appendix

```
## Perceptron classifier

# coding: utf-8

# In[3]:

get_ipython().magic('matplotlib inline')
```

```
# In[270]:

import pandas as pd
from numba import jit
import seaborn as seaborn
from collections import defaultdict
import numpy as np


# ## Getting the train and test data

# In[274]:

train = pd.read_csv("iris/iris_train.data",
                        names = ["sepal_length","sepal_width","petal_length",
                                 "petal_width","label"])
test = pd.read_csv("iris/iris_test.data",
                       names = ["sepal_length","sepal_width",
                                "petal_length","petal_width","label"])


# ## 2D feature wise scatter plots

# In[272]:

seaborn.pairplot(train, hue="label",palette="cubehelix")


# In[341]:

pd_train = train
pd_test = test


# In[357]:

class Perceptron(object):
    # constructor for perceptron has learning rate and no. of iterations
    def __init__(self, eta=0.01, n_iter=5):
        self.eta = eta
        self.n_iter = n_iter

    def predict(self, X):
        return np.where(self.dot_product(X) >= 0.0, 1, 0)

    def dot_product(self, X):
        return np.dot(X, self.weights[0:-1]) + self.weights[-1]

    def train(self, trainX, testX):
        trainX = np.array(trainX)
        # separate into features and target
        Y = trainX[:,-1]
        Y = [0 if x=='Iris-setosa' else 1 for x in Y]
        X = trainX[:,0:-1]

        #initialize weights and errors.
        # size of weights is no. of features + 1(bias)p
        np.random.seed(2)
```

```
        self.weights = np.random.rand(1 + X.shape[1])*2 - 1
        self.errors = []
        #print(self.weights)
        # Perform weight updates for n_iter
        it = 0
        for i in range(self.n_iter):
            num_errors = 0
            # Iterate over all training examples
            for sample, target in zip(X, Y):
                sampleN = np.append(np.array(sample), 1)
                #print(sampleN, target)
                prediction = self.predict(sample)
                #print(prediction, target)
                #print(sampleN.shape)
                #print(self.eta)
                update = np.array(self.eta * (target - prediction)*sampleN)
                #print(self.weights)
                self.weights = self.weights + update
               # print('iteration:'+str(it))
                it = it + 1
                #self.error(test)
            print('Epoch:'+str(i))
            print('weights:',str(self.weights))
            self.error(test)
        return self

    def error(self,test):
        testX = np.array(test)
        # separate into features and target
        Y = testX[:,-1]
        Y = np.array([0 if x=='Iris-setosa' else 1 for x in Y])
        X = testX[:,0:-1]
        predictions = []
        for row in X:
            input = np.append(np.array(row[:4]), 1)
            predictions.append(self.predict(row[:4]))
        predictions = np.array(predictions)
        error = sum(predictions != Y)
        error = error * 100.0 / len(test)
        print("Test Error Percentage:" + str(error)+"%")
        print("\n")
        return error


# In[358]:

p = Perceptron(0.1)


# In[359]:

p.train(train, test)


# ## Z-score the data

# In[327]:

mean = defaultdict(float)
```

```
sd = defaultdict(float)

def transform(df_orig):
    df = df_orig.copy(deep = True)
    cols = list(df_orig.columns)
    cols.remove('label')
    for col in cols:
        df[col] = (df_orig[col] - mean[col]) / sd[col]
    return df


def computeStatsFeatures(train_data):
    cols = list(train_data.columns)
    cols.remove('label')
    for col in cols:
        mean[col] = train_data[col].mean()
        sd[col] = train_data[col].std(ddof = 0)



# In[328]:

computeStatsFeatures(train)
for m in sd:
    print(str(m) + " " + str(sd[m]))
normalizedTrain = transform(train)
normalizedTest = transform(test)


# In[ ]:




# In[339]:

p = Perceptron(0.4)


# In[340]:

p.train(normalizedTrain, normalizedTest)

#print(normalizedTrain)
# In[317]:
normalizedTest
# In[ ]:


## Keras CNN

# coding: utf-8

# In[409]:

'''Train a simple deep CNN on the CIFAR10 small images dataset.
GPU run command with Theano backend (with TensorFlow, the GPU is automatically used):
    THEANO_FLAGS=mode=FAST_RUN,device=gpu,floatx=float32 python cifar10_cnn.py
It gets down to 0.65 test logloss in 25 epochs, and down to 0.55 after 50 epochs.
```

```
(it's still underfitting at that point, though).
'''

from __future__ import print_function
import keras
from keras.datasets import cifar10
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Conv2D, MaxPooling2D,BatchNormalization
from keras.callbacks import History
from keras import backend as K
from keras.constraints import maxnorm
import numpy as np
from keras.optimizers import SGD
K.set_image_dim_ordering('th')
history = History()


# In[410]:

batch_size = 50
num_classes = 10
data_augmentation = True
epochs = 25
lrate = 0.01
decay = lrate/epochs


# In[411]:

# fix random seed for reproducibility
seed = 7
np.random.seed(seed)


# In[412]:

# The data, shuffled and split between train and test sets:
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

# Convert class vectors to binary class matrices.
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)


# ## Pick subset of sample

# In[413]:

x_train = x_train[:2000]
y_train = y_train[:2000]
x_test = x_test[:500]
y_test = y_test[:500]
```

```
# In[414]:

x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255


# ## Plotting the input images

# In[336]:

from matplotlib import pyplot
from scipy.misc import toimage
# create a grid of 3x3 images
for i in range(0, 9):
        pyplot.subplot(330 + 1 + i)
        pyplot.imshow(toimage(x_train[i]))
# show the plot
pyplot.show()


# ## Fiddling with various architectures
# ### 2 Conv, 2 Dropout, 1 maxpool

# In[83]:

# Create the model
model = Sequential()
model.add(Conv2D(32, (3, 3), input_shape=x_train.shape[1:], padding='same', activatio
model.add(Dropout(0.2))
model.add(Conv2D(32, (3, 3), activation='relu', padding='same', kernel_constraint=max
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(512, activation='relu', kernel_constraint=maxnorm(3)))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
# Compile model
epochs = 25
lrate = 0.01
decay = lrate/epochs
sgd = SGD(lr=lrate, momentum=0.9, decay=decay, nesterov=False)
model.compile(loss='categorical_crossentropy', optimizer=sgd, metrics=['accuracy'])
print(model.summary())


# In[40]:

x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255

model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=epochs, batch_si
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

```
# ## old Model

# In[13]:

model = Sequential()

model.add(Conv2D(32, (3, 3), padding='same',
                 input_shape=x_train.shape[1:]))
model.add(Activation('relu'))
model.add(Conv2D(32, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Conv2D(64, (3, 3), padding='same'))
model.add(Activation('relu'))
# model.add(Conv2D(64, (3, 3)))
# model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes))
model.add(Activation('softmax'))
print(model.summary())


# In[7]:

# initiate RMSprop optimizer
opt = keras.optimizers.rmsprop(lr=0.0001, decay=1e-6)

# Let's train the model using RMSprop
model.compile(loss='categorical_crossentropy',
              optimizer=opt,
              metrics=['accuracy'])

if not data_augmentation:
    print('Not using data augmentation.')
    model.fit(x_train, y_train,
              batch_size=batch_size,
              epochs=epochs,
              validation_data=(x_test, y_test),
              shuffle=True,callbacks = [history])
else:
    print('Using real-time data augmentation.')
    # This will do preprocessing and realtime data augmentation:
    datagen = ImageDataGenerator(
        featurewise_center=False,  # set input mean to 0 over the dataset
        samplewise_center=False,  # set each sample mean to 0
        featurewise_std_normalization=False,  # divide inputs by std of the dataset
        samplewise_std_normalization=False,  # divide each input by its std
        zca_whitening=False,  # apply ZCA whitening
        rotation_range=0,  # randomly rotate images in the range (degrees, 0 to 180)
        width_shift_range=0.1,  # randomly shift images horizontally (fraction of to
        height_shift_range=0.1,  # randomly shift images vertically (fraction of tota
```

```
        horizontal_flip=True,  # randomly flip images
        vertical_flip=False)  # randomly flip images

    # Compute quantities required for feature-wise normalization
    # (std, mean, and principal components if ZCA whitening is applied).
    datagen.fit(x_train)

    # Fit the model on the batches generated by datagen.flow().
    model.fit_generator(datagen.flow(x_train, y_train,
                                     batch_size=batch_size),
                        steps_per_epoch=x_train.shape[0] // batch_size,
epochs=epochs,
                        validation_data=(x_test, y_test),callbacks = [history])


# ## Best model
# # RMS prop

# In[415]:

def train(opt,x_train,y_train,x_test,y_test,history,epochs,batch_size,model):
    history = History()
#     if(func):
#         model = Sequential()

#         model.add(Conv2D(32, (3, 3), input_shape=x_train.shape[1:], padding='same',
#         model.add(Dropout(0.2))
#         model.add(Conv2D(32, (3, 3), activation='relu', padding='same', kernel_con
#         model.add(MaxPooling2D(pool_size=(2, 2)))
#         model.add(Flatten())
#         model.add(Dense(512, activation='relu', kernel_constraint=maxnorm(3)))
#         model.add(Dropout(0.5))
#         model.add(Dense(num_classes, activation='softmax'))
#         print(model.summary())

    # Let's train the model using RMSprop
    model.compile(loss='categorical_crossentropy',
                  optimizer=opt,
                  metrics=['accuracy'])

    if not data_augmentation:
        print('Not using data augmentation.')
        model.fit(x_train, y_train,
                  batch_size=batch_size,
                  epochs=epochs,
                  validation_data=(x_test, y_test),
                  shuffle=True,callbacks = [history])
    else:
        print('Using real-time data augmentation.')
        # This will do preprocessing and realtime data augmentation:
        datagen = ImageDataGenerator(
            featurewise_center=False,  # set input mean to 0 over the dataset
            samplewise_center=False,  # set each sample mean to 0
            featurewise_std_normalization=False,  # divide inputs by std of the data
            samplewise_std_normalization=False,  # divide each input by its std
            zca_whitening=False,  # apply ZCA whitening
            rotation_range=0,  # randomly rotate images in the range (degrees, 0 to
            width_shift_range=0.1,  # randomly shift images horizontally (fraction o
            height_shift_range=0.1,  # randomly shift images vertically (fraction of
```

```
            horizontal_flip=True,  # randomly flip images
            vertical_flip=False)  # randomly flip images

        # Compute quantities required for feature-wise normalization
        # (std, mean, and principal components if ZCA whitening is applied).
        datagen.fit(x_train)

        # Fit the model on the batches generated by datagen.flow().
        model.fit_generator(datagen.flow(x_train, y_train,
                                         batch_size=batch_size),
                            steps_per_epoch=x_train.shape[0] // batch_size,
                            epochs=epochs,
                            validation_data=(x_test, y_test),callbacks = [history])
    return store_hist(history)


# In[416]:

def store_hist(history):
    loss = history.history['loss']
    val_loss = history.history['val_loss']
    acc = history.history['acc']
    val_acc = history.history['val_acc']
    return loss,val_loss,acc,val_acc


# In[417]:

def plot(epochs,acc,val_acc,ylabel,label1,label2,marker,line,color1,color2):
    X_axis = range(1,epochs+1)
    import matplotlib.pyplot as plt
    #plt.plot(X_axis, new1_acc, marker='x', linestyle='-', color='c',label="test acc

    # plt.plot(X_axis, layer1acc, marker='o', linestyle='--', color='r',label="1 con

    plt.plot(X_axis, acc, marker=marker, linestyle=line, color=color1,label=label1)
    plt.plot(X_axis, val_acc, marker=marker, linestyle=line, color=color2,label=label
    # plt.plot(X_axis, rms, marker='o', linestyle='-', color='g',label="rmsprop")
    # plt.plot(X_axis, nesterov, marker='o', linestyle='--', color='navy',label="nes
    # plt.plot(X_axis, val_acc[:10], marker='o', linestyle='--', color='r',label="ad
    # plt.plot(X_axis, denseless, marker='o', linestyle='--', color='c',label="1 con
    plt.xlabel('No. of epochs')
    plt.ylabel(ylabel)
    legend = plt.legend(loc='lower right', shadow=True)
    plt.show()


# In[418]:

epochs = 40


# In[427]:

model = Sequential()

model.add(Conv2D(32, (3, 3), input_shape=x_train.shape[1:], padding='same', activatio
model.add(Dropout(0.2))
model.add(Conv2D(32, (3, 3), activation='relu', padding='same', kernel_constraint=max
```

```
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(200, activation='relu', kernel_constraint=maxnorm(3)))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
print(model.summary())


# In[428]:

# initiate RMSprop optimizer
opt = keras.optimizers.rmsprop(lr=0.001, decay=1e-6)
batch_size=128
# Let's train the model using RMSprop
model.compile(loss='categorical_crossentropy',
              optimizer=opt,
              metrics=['accuracy'])

if not data_augmentation:
    print('Not using data augmentation.')
    model.fit(x_train, y_train,
              batch_size=batch_size,
              epochs=epochs,
              validation_data=(x_test, y_test),
              shuffle=True,callbacks = [history])
else:
    print('Using real-time data augmentation.')
    # This will do preprocessing and realtime data augmentation:
    datagen = ImageDataGenerator(
        featurewise_center=False,  # set input mean to 0 over the dataset
        samplewise_center=False,  # set each sample mean to 0
        featurewise_std_normalization=False,  # divide inputs by std of the dataset
        samplewise_std_normalization=False,  # divide each input by its std
        zca_whitening=False,  # apply ZCA whitening
        rotation_range=0,  # randomly rotate images in the range (degrees, 0 to 180)
        width_shift_range=0.1,  # randomly shift images horizontally (fraction of to
        height_shift_range=0.1,  # randomly shift images vertically (fraction of tota
        horizontal_flip=True,  # randomly flip images
        vertical_flip=False)  # randomly flip images

    # Compute quantities required for feature-wise normalization
    # (std, mean, and principal components if ZCA whitening is applied).
    datagen.fit(x_train)

    # Fit the model on the batches generated by datagen.flow().
    model.fit_generator(datagen.flow(x_train, y_train,
                                     batch_size=batch_size),
                        steps_per_epoch=x_train.shape[0] // batch_size,
epochs=epochs,
                        validation_data=(x_test, y_test),callbacks = [history])


# In[112]:

rms_loss = history.history['loss']
rms_val_loss = history.history['val_loss']
rms_acc = history.history['acc']
rms_val_acc = history.history['val_acc']
```

```
# In[113]:

X_axis = range(1,epochs+1)
import matplotlib.pyplot as plt
#plt.plot(X_axis, new1_acc, marker='x', linestyle='-', color='c',label="test accuracy

# plt.plot(X_axis, layer1acc, marker='o', linestyle='--', color='r',label="1 conv lay

plt.plot(X_axis, rms_acc, marker='o', linestyle='--', color='b',label="train accuracy
plt.plot(X_axis, rms_val_acc, marker='o', linestyle='--', color='r',label="test accur
# plt.plot(X_axis, rms, marker='o', linestyle='-', color='g',label="rmsprop")
# plt.plot(X_axis, nesterov, marker='o', linestyle='--', color='navy',label="nesterov
# plt.plot(X_axis, val_acc[:10], marker='o', linestyle='--', color='r',label="adam")
# plt.plot(X_axis, denseless, marker='o', linestyle='--', color='c',label="1 conv lay
plt.xlabel('No. of epochs')
plt.ylabel('Accuracy')
legend = plt.legend(loc='lower right', shadow=True)
plt.show()


# In[123]:

X_axis = range(1,epochs+1)
import matplotlib.pyplot as plt
#plt.plot(X_axis, new1_acc, marker='x', linestyle='-', color='c',label="test accuracy

# plt.plot(X_axis, layer1acc, marker='o', linestyle='--', color='r',label="1 conv lay

plt.plot(X_axis, rms_acc, marker='x', linestyle='-', color='g',label="train acc")
plt.plot(X_axis, rms_val_acc, marker='x', linestyle='-', color='y',label="test acc")
# plt.plot(X_axis, rms, marker='o', linestyle='-', color='g',label="rmsprop")
# plt.plot(X_axis, nesterov, marker='o', linestyle='--', color='navy',label="nesterov
# plt.plot(X_axis, val_acc[:10], marker='o', linestyle='--', color='r',label="adam")
# plt.plot(X_axis, denseless, marker='o', linestyle='--', color='c',label="1 conv lay
plt.xlabel('No. of epochs')
plt.ylabel('Accuracy')
legend = plt.legend(loc='lower right', shadow=True)
plt.show()


# In[114]:

time_rms = 25


# ## SGD

# In[120]:

# initiate SGD optimizer
opt = SGD(lr=lrate, momentum=0.0, decay=decay, nesterov=False)
train(opt,x_train,y_train,x_test,y_test,history,epochs,batch_size)


# In[121]:

sgd_loss = history.history['loss']
sgd_val_loss = history.history['val_loss']
```

```
sgd_acc = history.history['acc']
sgd_val_acc = history.history['val_acc']


# In[122]:

X_axis = range(1,epochs+1)
import matplotlib.pyplot as plt
#plt.plot(X_axis, new1_acc, marker='x', linestyle='-', color='c',label="test accuracy

# plt.plot(X_axis, layer1acc, marker='o', linestyle='--', color='r',label="1 conv lay

plt.plot(X_axis, sgd_acc, marker='o', linestyle='--', color='b',label="train accuracy
plt.plot(X_axis, sgd_val_acc, marker='o', linestyle='--', color='r',label="test accur
# plt.plot(X_axis, rms, marker='o', linestyle='-', color='g',label="rmsprop")
# plt.plot(X_axis, nesterov, marker='o', linestyle='--', color='navy',label="nesterov
# plt.plot(X_axis, val_acc[:10], marker='o', linestyle='--', color='r',label="adam")
# plt.plot(X_axis, denseless, marker='o', linestyle='--', color='c',label="1 conv lay
plt.xlabel('No. of epochs')
plt.ylabel('Accuracy')
legend = plt.legend(loc='lower right', shadow=True)
plt.show()


# In[124]:

X_axis = range(1,epochs+1)
import matplotlib.pyplot as plt
#plt.plot(X_axis, new1_acc, marker='x', linestyle='-', color='c',label="test accuracy

# plt.plot(X_axis, layer1acc, marker='o', linestyle='--', color='r',label="1 conv lay

plt.plot(X_axis, sgd_acc, marker='x', linestyle='-', color='g',label="train acc")
plt.plot(X_axis, sgd_val_acc, marker='x', linestyle='-', color='y',label="test acc")
# plt.plot(X_axis, rms, marker='o', linestyle='-', color='g',label="rmsprop")
# plt.plot(X_axis, nesterov, marker='o', linestyle='--', color='navy',label="nesterov
# plt.plot(X_axis, val_acc[:10], marker='o', linestyle='--', color='r',label="adam")
# plt.plot(X_axis, denseless, marker='o', linestyle='--', color='c',label="1 conv lay
plt.xlabel('No. of epochs')
plt.ylabel('Accuracy')
legend = plt.legend(loc='lower right', shadow=True)
plt.show()


# ## Adaptive gradient

# In[126]:

# initiate Adagrad optimizer
opt = keras.optimizers.Adagrad(lr=0.01, epsilon=1e-08, decay=0.0)
train(opt,x_train,y_train,x_test,y_test,history,epochs,batch_size,model)


# In[144]:

ada_loss,ada_val_loss,ada_acc,ada_val_acc = store_hist(history)


# In[147]:
```

```
plot(epochs,acc,val_acc,'SGD with Adagrad',"Train Accuracy","Test Accuracy","o","--",


# In[149]:

plot(epochs,acc,val_acc,'SGD with Adagrad',"Train Accuracy","Test Accuracy","x","-",


# ## Nesterov Accelerated gradient

# In[303]:

epochs = 30
# initiate Nesterov optimizer
opt = keras.optimizers.SGD(lr=lrate, decay=decay, momentum=0.9, nesterov=True)
nest_loss,nest_val_loss,nest_acc,nest_val_acc = train(opt,x_train,y_train,x_test,y_te


# In[304]:

plot(epochs,nest_acc,nest_val_acc,'SGD with Nesterov',"Train Accuracy","Test Accuracy
plot(epochs,nest_acc,nest_val_acc,'SGD with Nesterov',"Train Accuracy","Test Accuracy


# ## Batch normalization

# In[305]:


model = Sequential()

model.add(Conv2D(32, (3, 3), input_shape=x_train.shape[1:], padding='same', activatio
model.add(BatchNormalization())
model.add(Dropout(0.2))
model.add(Conv2D(32, (3, 3), activation='relu', padding='same', kernel_constraint=max
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(512, activation='relu', kernel_constraint=maxnorm(3)))
#model.add(BatchNormalization())
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))

print(model.summary())


# In[306]:

epochs = 30
# initiate RMSprop optimizer
opt = keras.optimizers.rmsprop(lr=0.0001, decay=1e-6)
batch_loss,batch_val_loss,batch_acc,batch_val_acc = train(opt,x_train,y_train,x_test,


# In[307]:

plot(epochs,batch_acc,batch_val_acc,'SGD with Batch Normalization',"Train Accuracy","
plot(epochs,batch_acc,batch_val_acc,'SGD with Batch Normalization',"Train Accuracy","
```

```
# ## Batch norm:2

# In[359]:

model = Sequential()

model.add(Conv2D(32, (3, 3), input_shape=x_train.shape[1:], padding='same', activatio
model.add(Dropout(0.3))
model.add(Conv2D(32, (3, 3), kernel_initializer='uniform', padding='same', kernel_co
# model.add(BatchNormalization())
model.add(Activation('relu'))

model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(512, kernel_initializer='uniform', kernel_constraint=maxnorm(3)))
model.add(BatchNormalization())
model.add(Activation('relu'))

model.add(Dropout(0.5))
model.add(Dense(num_classes, kernel_initializer='uniform'))
#model.add(BatchNormalization())
model.add(Activation('softmax'))

print(model.summary())


# In[360]:

epochs = 30
history = History()
# initiate RMSprop optimizer
opt = keras.optimizers.rmsprop(lr=0.001, decay=1e-6)
batch_loss1,batch_val_loss1,batch_acc1,batch_val_acc1 = train(opt,x_train,y_train,x_t


# In[310]:

plot(epochs,batch_acc1,batch_val_acc1,'SGD with Batch Normalization',"Train Accuracy"
plot(epochs,batch_acc1,batch_val_acc1,'SGD with Batch Normalization',"Train Accuracy"


# ## Global average pooling layer

# In[357]:

from keras.layers.pooling import GlobalAveragePooling2D
model = Sequential()

model.add(Conv2D(32, (3, 3), input_shape=x_train.shape[1:], padding='same', activatio
# model.add(Dropout(0.2))
model.add(Conv2D(32, (3, 3), activation='relu', padding='same', kernel_constraint=max
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(10, 1, 1, border_mode='valid'))
model.add(Activation('softmax'))
model.add(GlobalAveragePooling2D())

# model.add(Flatten())
```

```
# model.add(Dropout(0.5))
#model.add(Dense(num_classes, activation='softmax'))
# model.add(Activation('softmax'))
print(model.summary())


# In[358]:

epochs = 30
history = History()
# initiate RMSprop optimizer
opt = keras.optimizers.rmsprop(lr=0.0001, decay=1e-6)
glob_loss,glob_val_loss,glob_acc,glob_val_acc = train(opt,x_train,y_train,x_test,y_te


# In[313]:

plot(epochs,glob_acc,glob_val_acc ,'SGD with Global avg layer',"Train Accuracy","Test
plot(epochs,glob_acc,glob_val_acc ,'SGD with Global avg layer',"Train Accuracy","Test


# In[ ]:



## Feed forward network implementation without momentum
import numpy as np
import load_mnist as mnist
import mlp as mlp
import activation_functions as func
import plot as pl

def loadData(num_train_samples, num_test_samples, normalization_constant):
    b = mnist.MNIST()

    TrainingI, TrainingL = b.load_training()
    TestingI, TestingL = b.load_testing()

    # load training and testing data
    X = TrainingI[:num_train_samples]
    Y = TrainingL[:num_train_samples]
    X_test = TestingI[:num_test_samples]
    Y_test = TestingL[:num_test_samples]

    #convert to np arrays
    X= np.array(X)
    Y = np.array(Y)
    X_test = np.array(X_test)
    Y_test = np.array(Y_test)

    #Shuffling the array
    X = np.column_stack((X,Y))
    np.random.shuffle(X)
    Y = X[:,784:785]
    X = X[:,:784]
    Y = Y.reshape((60000,))
    #Adding a bias to X
    X = np.column_stack((np.ones(len(X)), X))
```

```
    #Adding a bias to X_test
    X_test = np.column_stack((np.ones(len(X_test)), X_test))
    #normalization
    X= X/normalization_constant
    X_test = X_test/normalization_constant

    #Subtract mean
    mean_train = np.mean(X,axis =1)
    X = X - mean_train[:, np.newaxis]
    # Mean test
    mean_test = np.mean(X_test,axis =1)
    X_test = X_test - mean_test[:, np.newaxis]

    # Validation set
    X_H = X[50000:60000]
    X = X[:50000]
    Y_H = Y[50000:60000]
    Y = Y[:50000]
    # Hot ones
    Y = np.eye(10)[Y]
    Y_H = np.eye(10)[Y_H]
    Y_test = np.eye(10)[Y_test]
    return X, Y, X_test, Y_test, X_H, Y_H

def calcAccuracy(Y, T, Iset):
    Y_Max = np.argmax(Y, axis=1)
    T_Max = np.argmax(T, axis=1)
    temp_Max = T_Max-Y_Max
    accuracy = (temp_Max == 0).sum()
    accuracy = 100*accuracy/len(temp_Max)
    #print("Accuracy " + str(Iset) + ":"+str(accuracy))
    return accuracy

X, T, X_test, T_test, X_H, T_H = loadData(60000,10000, 255.0)
eta = 0.001
mu = 0.9
iterations_arr = []
accuracy_train_arr = []
accuracy_hold_arr = []
accuracy_test_arr = []
early_stopping = []
T_complete = T

#Training
hiddenLayer = mlp.Layer("hidden",func.funnytanh, 785, 80)
outputLayer = mlp.Layer("output", func.softmax, 80, 10)
batch_size = 500
for epoch in range(0,25):
    count = 0
    #anealing
    l_r = eta/(1+ epoch/300)
    while count < (50000 - batch_size):
        #Feed Forward
        # layerType="hidden", n_in=None, n_out=None, w_in=None,w_out = None, random_
        Z = hiddenLayer.output(X[count:(count+batch_size)])  # 50000X300
        Y = outputLayer.output(Z) # 50000X10
        T = T_complete[count:(count+batch_size)]
        #Test_Testing
        Z_test = hiddenLayer.output(X_test)  # 10000X300
```

```
            Y_test = outputLayer.output(Z_test) # 10000X10
            #Hold-out_Testing
            Z_H = hiddenLayer.output(X_H)  # 10000X300
            Y_H = outputLayer.output(Z_H) # 10000X10

            #Accuracy
            accuracy_train = calcAccuracy(Y,T, "Train")
            accuracy_test = calcAccuracy(Y_test,T_test, "Test")
            accuracy_hold = calcAccuracy(Y_H,T_H, "Hold")

            #backprop
            delta_output = T - Y # 50000X10
            # temp
            d_temp = np.dot(delta_output, np.transpose(outputLayer.W)) #50000X300
            delta_hidden = np.multiply(func.dfunnytanh(Z), d_temp) #50000X300
            outputLayer.setW(outputLayer.W + l_r*np.dot(np.transpose(Z), delta_output))
            hiddenLayer.setW(hiddenLayer.W + l_r*np.dot(np.transpose(X[count:(count+batch
            #print(" Iteration: " + str(epoch) + " Count:" + str(count))
            # Error
            cross_err = (-1/len(T))*np.sum(np.sum((np.multiply(T,np.log(Y)))))
            cross_err_hold = (-1/len(T_H))*np.sum(np.sum((np.multiply(T_H,np.log(Y_H)))))


            # Early Stopping
            if len(early_stopping) == 0 or ( cross_err_hold > early_stopping[-1]):
                 early_stopping.append(cross_err_hold)
            else:
                early_stopping[:] = []

            if(len(early_stopping) == 6):
                print(early_stopping)
                print("Early stopped at iteration:" + str(epoch))
                break
            count = count+batch_size
                #print("cross_err:" + str(cross_err_hold))
        accuracy_train = calcAccuracy(Y,T, "Train")
        accuracy_test = calcAccuracy(Y_test,T_test, "Test")
        accuracy_train_arr.append(accuracy_train)
        accuracy_test_arr.append(accuracy_test)
        accuracy_hold_arr.append(accuracy_hold)
        print('Epoch '+ str(epoch)+ ": Train/Test Accuracy:[" + str(accuracy_train) +" "
        iterations_arr.append(epoch)


import seaborn as sns
x = iterations_arr
sigmoid_train = accuracy_train_arr
sigmoid_test = accuracy_test_arr
sns.plt.ylim([85,100])
sns.plt.xlabel("Epochs")
sns.plt.ylabel("Train/Test accuracy")
sns.plt.plot(x, sigmoid_train, sns.xkcd_rgb["amber"], lw=3, label = "Train accuracy"
sns.plt.plot(x, sigmoid_test,  sns.xkcd_rgb["denim blue"], lw=3, label = "Test accura
sns.set_style("whitegrid")
sns.plt.title("FeedForward network with 1 hidden layer (80 hidden units)")
sns.plt.legend()

# pl.plotPoints(1,20,10,iterations_arr,accuracy_train_arr,32,'blue',"Percent Correct
# pl.plotPoints(1,20,10,iterations_arr,accuracy_test_arr,32,'red',"Percent Correct v
```

```
## Feed forward network with momentum
import numpy as np
import load_mnist as mnist
import mlp as mlp
import activation_functions as func
import plot as pl

def loadData(num_train_samples, num_test_samples, normalization_constant):
    b = mnist.MNIST()

    TrainingI, TrainingL = b.load_training()
    TestingI, TestingL = b.load_testing()

    # load training and testing data
    X = TrainingI[:num_train_samples]
    Y = TrainingL[:num_train_samples]
    X_test = TestingI[:num_test_samples]
    Y_test = TestingL[:num_test_samples]

    #convert to np arrays
    X= np.array(X)
    Y = np.array(Y)
    X_test = np.array(X_test)
    Y_test = np.array(Y_test)

    #Shuffling the array
    X = np.column_stack((X,Y))
    np.random.shuffle(X)
    Y = X[:,784:785]
    X = X[:,:784]
    Y = Y.reshape((60000,))
    #Adding a bias to X
    X = np.column_stack((np.ones(len(X)), X))
    #Adding a bias to X_test
    X_test = np.column_stack((np.ones(len(X_test)), X_test))
    #normalization
    X= X/normalization_constant
    X_test = X_test/normalization_constant

    #Subtract mean
    mean_train = np.mean(X,axis =1)
    X = X - mean_train[:, np.newaxis]
    # Mean test
    mean_test = np.mean(X_test,axis =1)
    X_test = X_test - mean_test[:, np.newaxis]

    # Validation set
    X_H = X[50000:60000]
    X = X[:50000]
    Y_H = Y[50000:60000]
    Y = Y[:50000]
    # Hot ones
    Y = np.eye(10)[Y]
    Y_H = np.eye(10)[Y_H]
    Y_test = np.eye(10)[Y_test]
    return X, Y, X_test, Y_test, X_H, Y_H
```

```
def calcAccuracy(Y, T, Iset):
    Y_Max = np.argmax(Y, axis=1)
    T_Max = np.argmax(T, axis=1)
    temp_Max = T_Max-Y_Max
    accuracy = (temp_Max == 0).sum()
    accuracy = 100*accuracy/len(temp_Max)
#    print("Accuracy " + str(Iset) + ":"+str(accuracy))
    return accuracy

X, T, X_test, T_test, X_H, T_H = loadData(60000,10000, 255.0)
eta = 1.0/50
mu = 0.9
iterations_arr = []
accuracy_train_arr = []
accuracy_hold_arr = []
accuracy_test_arr = []
early_stopping = []
T_complete = T

#Training
hiddenLayer = mlp.Layer("hidden",func.funnytanh, 785, 80)
outputLayer = mlp.Layer("output", func.softmax,80, 10)

W_prev_ij = np.zeros(hiddenLayer.W.shape)
W_prev_jk = np.zeros(outputLayer.W.shape)
batch_size = 50
max_acc_test = 0.0
max_acc_hold = 0.0
for epoch in range(0,40):
    count = 0
    #anealing
    l_r = eta/(1+ epoch/300)
    while count < (50000 - batch_size):
#        print("Epoch: " + str(epoch) + " Count:" + str(count))
        #Feed Forward
        # layerType="hidden", n_in=None, n_out=None, w_in=None,w_out = None, random_
#        Z = hiddenLayer.output(X[count:(count+batch_size)])  # 50000X300
#        Y = outputLayer.output(Z) # 50000X10

#        T = T_complete[count:(count+batch_size)]
#        #Test_Testing
#        Z_test = hiddenLayer.output(X_test)  # 10000X300
#        Y_test = outputLayer.output(Z_test) # 10000X10
#

        #Hold-out_Testing
#        Z_H = hiddenLayer.output(X_H)  # 10000X300
#        Y_H = outputLayer.output(Z_H) # 10000X10

        #Accuracy
#        accuracy_train = calcAccuracy(Y,T, "Train")
#        accuracy_test = calcAccuracy(Y_test,T_test, "Test")
#        accuracy_hold = calcAccuracy(Y_H,T_H, "Hold")
#        if max_acc_test < accuracy_test :
#            max_acc_test = accuracy_test
#        if max_acc_hold < accuracy_hold :
#            max_acc_hold = accuracy_hold
        # Error
#        cross_err_test = (-1/len(T_test))*np.sum(np.sum((np.multiply(T_test,np.log(
```

```
#         cross_err_hold = (-1/len(T_H))*np.sum(np.sum((np.multiply(T_H,np.log(Y_H)))
        #print("cross entropy error Test:" + str(cross_err_test))
        #print("cross entropy error Hold:" + str(cross_err_hold))
        # momentum

        temp_weight_ij = hiddenLayer.W
        temp_weight_jk = outputLayer.W

        new_weight_ij = hiddenLayer.W + mu*(hiddenLayer.W - W_prev_ij)
        new_weight_jk = outputLayer.W + mu*(outputLayer.W - W_prev_jk)

        outputLayer.setW(new_weight_jk)
        hiddenLayer.setW(new_weight_ij)

        #Feed forward
        Z = hiddenLayer.output(X[count:(count+batch_size)])  # 50000X300
        Y = outputLayer.output(Z) # 50000X10

        T = T_complete[count:(count+batch_size)]
        #Test_Testing
#         Z_test = hiddenLayer.output(X_test)  # 10000X300
#         Y_test = outputLayer.output(Z_test) # 10000X10


        #backprop
        delta_output = T - Y # 50000X10

        d_temp = np.dot(delta_output, np.transpose(outputLayer.W)) #500X50
        delta_hidden = np.multiply(func.dfunnytanh(Z), d_temp) #500X50

        grad_jk = np.dot(np.transpose(Z), delta_output)
        new_weight_jk = new_weight_jk + l_r*(1/batch_size)*grad_jk

        grad_ij = np.dot(np.transpose(X[count:(count+batch_size)]), delta_hidden)
        new_weight_ij = new_weight_ij + l_r*(1/batch_size)*grad_ij
        # temp

        outputLayer.setW(new_weight_jk) #300X10
        hiddenLayer.setW(new_weight_ij) #785X300

        W_prev_ij = temp_weight_ij
        W_prev_jk = temp_weight_jk
        count = count+batch_size

    iterations_arr.append(epoch)
    Z = hiddenLayer.output(X)  # 50000X300
    Y = outputLayer.output(Z) # 50000X10
    Z_test = hiddenLayer.output(X_test)  # 10000X300
    Y_test = outputLayer.output(Z_test) # 10000X10
    accuracy_train = calcAccuracy(Y,T_complete, "TrainComplete:")
    accuracy_test = calcAccuracy(Y_test,T_test, "Test:")
    accuracy_train_arr.append(accuracy_train)
    accuracy_test_arr.append(accuracy_test)
#     accuracy_hold_arr.append(accuracy_hold)
    print('Epoch '+ str(epoch)+ ": Train/Test Accuracy:[" + str(accuracy_train) +" "
#print("Accuracy test:" + str(max_acc_test))
#print("Accuracy hold:" + str(max_acc_hold))
```

```
import seaborn as sns
x = iterations_arr
sigmoid_train = accuracy_train_arr
sigmoid_test = accuracy_test_arr
sns.plt.ylim([85,100])
sns.plt.xlabel("Epochs")
sns.plt.ylabel("Train/Test accuracy")
sns.plt.plot(x, sigmoid_train, sns.xkcd_rgb["amber"], lw=3, label = "Train accuracy"
sns.plt.plot(x, sigmoid_test,  sns.xkcd_rgb["denim blue"], lw=3, label = "Test accura
sns.set_style("whitegrid")
sns.plt.title("Sigmoid performance with 1 hidden layer (80 hidden units)")
sns.plt.legend()


#pl.plotPoints(1,20,10,iterations_arr,accuracy_train_arr,32,'blue',"Percent Correct
#pl.plotPoints(1,20,10,iterations_arr,accuracy_test_arr,32,'red',"Percent Correct vs


## Feed forward network with 2 layers and its experiments
# -*- coding: utf-8 -*-
"""
Created on Wed Feb  1 16:37:21 2017

@author: kriti
"""

import numpy as np
import load_mnist as mnist
import mlp as mlp
import activation_functions as func
import plot as pl

def loadData(num_train_samples, num_test_samples, normalization_constant):
    b = mnist.MNIST()

    TrainingI, TrainingL = b.load_training()
    TestingI, TestingL = b.load_testing()

    # load training and testing data
    X = TrainingI[:num_train_samples]
    Y = TrainingL[:num_train_samples]
    X_test = TestingI[:num_test_samples]
    Y_test = TestingL[:num_test_samples]

    #convert to np arrays
    X= np.array(X)
    Y = np.array(Y)
    X_test = np.array(X_test)
    Y_test = np.array(Y_test)

    #Shuffling the array
    X = np.column_stack((X,Y))
    np.random.shuffle(X)
    Y = X[:,784:785]
    X = X[:,:784]
    Y = Y.reshape((60000,))
    #Adding a bias to X
    X = np.column_stack((np.ones(len(X)), X))
```

```
    #Adding a bias to X_test
    X_test = np.column_stack((np.ones(len(X_test)), X_test))
    #normalization
    X= X/normalization_constant
    X_test = X_test/normalization_constant

    #Subtract mean
    mean_train = np.mean(X,axis =1)
    X = X - mean_train[:, np.newaxis]
    # Mean test
    mean_test = np.mean(X_test,axis =1)
    X_test = X_test - mean_test[:, np.newaxis]

    # Validation set
    X_H = X[50000:60000]
    X = X[:50000]
    Y_H = Y[50000:60000]
    Y = Y[:50000]
    # Hot ones
    Y = np.eye(10)[Y]
    Y_H = np.eye(10)[Y_H]
    Y_test = np.eye(10)[Y_test]
    return X, Y, X_test, Y_test, X_H, Y_H

def calcAccuracy(Y, T, Iset):
    Y_Max = np.argmax(Y, axis=1)
    T_Max = np.argmax(T, axis=1)
    temp_Max = T_Max-Y_Max
    accuracy = (temp_Max == 0).sum()
    accuracy = 100*accuracy/len(temp_Max)
    #print("Accuracy " + str(Iset) + ":"+str(accuracy))
    return accuracy

X, T, X_test, T_test, X_H, T_H = loadData(60000,10000, 255.0)
eta = (1/50.0)
mu = 0.9
iterations_arr = []
accuracy_train_arr = []
accuracy_hold_arr = []
accuracy_test_arr = []
early_stopping = []
T_complete = T

#Training
hiddenLayer1 = mlp.Layer("hidden",func.funnytanh, 785, 300)
hiddenLayer2 = mlp.Layer("hidden",func.funnytanh, 300, 100)
outputLayer = mlp.Layer("output", func.softmax, 100, 10)

W_prev_ij = np.zeros(hiddenLayer1.W.shape) #hidden1
W_prev_jk = np.zeros(hiddenLayer2.W.shape) #hidden2
W_prev_kl = np.zeros(outputLayer.W.shape) #output
#W_prev_ij = [np.random.randn(784,80)/np.sqrt(2.0/80) for x in hiddenLayer1.W.shape]
#W_prev_jk = [np.random.randn(80,30)/np.sqrt(2.0/30) for x in hiddenLayer2.W.shape]
#W_prev_kl = [np.random.randn(30,10)/np.sqrt(2.0/10) for x in outputLayer.W.shape] #

batch_size = 10
ctr=0
for epoch in range(0,200):
    count = 0
```

```
    #anealing
    l_r = eta/50.0#/(1+ epoch/300)
    if(epoch>25):
        l_r = l_r*0.75
    while count < (50000 - batch_size):
        ctr = ctr+1
        #print("Iteration: " + str(epoch) + " Count:" + str(count))
        #Feed Forward
        # layerType="hidden", n_in=None, n_out=None, w_in=None,w_out = None, random_

        Z1 = hiddenLayer1.output(X[count:(count+batch_size)])  # 50000X300
        Z2 = hiddenLayer2.output(Z1)
        Y = outputLayer.output(Z2) # 50000X10
        T = T_complete[count:(count+batch_size)]
        #Test_Testing
#         Z_test1 = hiddenLayer1.output(X_test)  # 10000X300
#         Z_test2 = hiddenLayer2.output(Z_test1)
#         Y_test = outputLayer.output(Z_test2) # 10000X10
        #change
        #Hold-out_Testing
#         Z_H1 = hiddenLayer1.output(X_H)  # 10000X300
#         Z_H2 = hiddenLayer2.output(Z_H1)
#         Y_H = outputLayer.output(Z_H2) # 10000X10
        #change
        #Accuracy
#         accuracy_train = calcAccuracy(Y,T, "Train")
#         accuracy_test = calcAccuracy(Y_test,T_test, "Test")
#         accuracy_hold = calcAccuracy(Y_H,T_H, "Hold")
#
#         iterations_arr.append(ctr)
#         accuracy_train_arr.append(accuracy_train)
#         accuracy_test_arr.append(accuracy_test)
#         accuracy_hold_arr.append(accuracy_hold)
        # Error
        # cross_err = (-1/len(T))*np.sum(np.sum((np.multiply(T,np.log(Y)))))
#         cross_err_hold = (-1/len(T_H))*np.sum(np.sum((np.multiply(T_H,np.log(Y_H))),
#         print("cross_err:" + str(cross_err_hold))
        #change
#         temp_weight_ij = hiddenLayer1.W
#         temp_weight_jk = hiddenLayer2.W
#         temp_weight_kl = outputLayer.W
#
#         new_weight_ij = hiddenLayer1.W + mu*(hiddenLayer1.W - W_prev_ij)
#         new_weight_jk = hiddenLayer2.W + mu*(hiddenLayer2.W - W_prev_jk)
#         new_weight_kl = outputLayer.W + mu*(outputLayer.W - W_prev_kl)
##
#
#         hiddenLayer1.setW(new_weight_ij)
#         hiddenLayer2.setW(new_weight_jk)
#         outputLayer.setW(new_weight_kl)
#
        #Feed forward
#         Z1 = hiddenLayer1.output(X[count:(count+batch_size)])  # 50000X300
#         Z2 = hiddenLayer2.output(Z1)
#         Y = outputLayer.output(Z2) # 50000X10
#         #change
        #backprop
        delta_output = T - Y # 50000X10
```

```
            d_temp2 = np.dot(delta_output, np.transpose(outputLayer.W)) #500X50
            delta_hidden2 = np.multiply(func.dfunnytanh(Z2), d_temp2) #500X50

            d_temp1 = np.dot(delta_hidden2, np.transpose(hiddenLayer2.W)) #500X50
            delta_hidden1 = np.multiply(func.dfunnytanh(Z1), d_temp1) #500X50

            #output
            grad_kl = np.dot(np.transpose(Z2), delta_output)
            new_weight_kl = outputLayer.W + l_r*grad_kl
            #new_weight_kl = new_weight_kl + l_r*grad_kl
            #hidden2
            grad_jk = np.dot(np.transpose(Z1), delta_hidden2)
            new_weight_jk = hiddenLayer2.W  + l_r*grad_jk

            #new_weight_jk = new_weight_jk + l_r*grad_jk
            #hidden1
            grad_ij = np.dot(np.transpose(X[count:(count+batch_size)]), delta_hidden1)
            new_weight_ij = hiddenLayer1.W + l_r*grad_ij
           # new_weight_ij = new_weight_ij + l_r*grad_ij
            # temp

            outputLayer.setW(new_weight_kl) #300X10
            hiddenLayer2.setW(new_weight_jk) #785X300
            hiddenLayer1.setW(new_weight_ij)

#           W_prev_ij = temp_weight_ij
#           W_prev_jk = temp_weight_jk
#           W_prev_kl = temp_weight_kl

            count = count+batch_size
        Z1 = hiddenLayer1.output(X)  # 50000X300
        Z2 = hiddenLayer2.output(Z1)
        Y = outputLayer.output(Z2) # 50000X10
        Z_test1 = hiddenLayer1.output(X_test)  # 10000X300
        Z_test2 = hiddenLayer2.output(Z_test1)
        Y_test = outputLayer.output(Z_test2) # 10000X10
        accuracy_train = calcAccuracy(Y,T_complete, "Train")
        accuracy_test = calcAccuracy(Y_test,T_test, "Test")
        print('Epoch '+ str(epoch)+ ": Train/Test Accuracy:[" + str(accuracy_train) +" "
        accuracy_train_arr.append(accuracy_train)
        accuracy_test_arr.append(accuracy_test)
        iterations_arr.append(epoch)
#        accuracy_hold_arr.append(accuracy_hold)


import seaborn as sns
x = iterations_arr
sigmoid_train = accuracy_train_arr
sigmoid_test = accuracy_test_arr
sns.plt.ylim([85,100])
sns.plt.xlabel("Epochs")
sns.plt.ylabel("Train/Test accuracy")
sns.plt.plot(x, sigmoid_train, sns.xkcd_rgb["amber"], lw=3, label = "Train accuracy"
sns.plt.plot(x, sigmoid_test,  sns.xkcd_rgb["denim blue"], lw=3, label = "Test accura
sns.set_style("whitegrid")
sns.plt.title("Feed forward network with 2 hidden layers (80,40 hidden units)")
sns.plt.legend()

#pl.plotPoints(1,20,10,iterations_arr,accuracy_train_arr,32,'blue',"Percent Correct
```

```
#pl.plotPoints(1,20,10,iterations_arr,accuracy_test_arr,32,'red',"Percent Correct vs

## activation functions
import numpy as np
# transfer functions
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# derivative of sigmoid
def dsigmoid(y):
    return np.multiply(y , (1.0 - y))

# using softmax as output layer is recommended for classification where outputs are
def softmax(w):
    w = np.transpose(w)
    return np.transpose(np.exp(w) / np.sum(np.exp(w), axis=0))

# using tanh over logistic sigmoid for the hidden layer is recommended
def tanh(x):
    return np.tanh(x)

# derivative for tanh sigmoid
def dtanh(y):
    return 1 - np.multiply(y,y)

# with linear term
def funnytanh(x):
    return 1.7159*np.tanh((2/3)*x)

def dfunnytanh(x):
    return 1.7159*(2/3)*(1-(np.tanh((2/3)*x)*np.tanh((2/3)*x)))

## Layer class

import numpy as np

# W: weights coming in Layer l
class Layer(object):
    def __init__(self,layerType="hidden", activation=None, n_in=None, n_out=None):
        self.layerType = layerType
        # Random state to seed the randomization of weights
        rnd = np.random.RandomState()
        # Initialize weights according to the number of inputs and outputs of the la
        # Can be changed later to try different values
        self.W = rnd.normal(loc=0.0,scale=1/np.sqrt(n_in),size=(n_in,n_out))
        # print(n_in,n_out,self.W)
        self.activation = activation

    def setW(self,W):
        self.W = W

    def output(self, input):
        self.y = self.activation(np.dot(input, self.W))
        return self.y

    def computeDelta(self, last_delta):
        delta = last_delta
        return delta
```

##