COGS 260: Image Processing, Assignment 3

Ankur Jain, A53097130, anj022@ucsd.edu

May 9, 2016

Abstract

In this project we present different methods of classification (multi-class) on MNIST, CIFAR-10 and UCI-iris dataset. The MNIST database consists of a total of 70000 including 60000 training samples and 10000 test samples, CIFAR-10 has 50,000 RGB images as train set and 10000 images as test set. Both MNIST and CIFAR-10 has 10 different classes We compare the performance of various classification models and do a comparative study.

Keywords: Convolutional Neural Networks, Feed Forward Neural Nets, Perceptron.

1 Perceptron

1.1 Scatter Plot

Fig 1 (a-f) shows the scatter plot of the iris training set between all combinations of features. Blue dots represent 1 class and yellow represents the other. In all the plots we can see that the training data is linearly separable. Code for this section can be found in listing 1

1.2 Training Perceptron

- Initial Weight = [0.01, 0.01, 0.01, 0.01]
- theta = 1
- Learning Rate = 1

  Code for this section can be found in listing 2

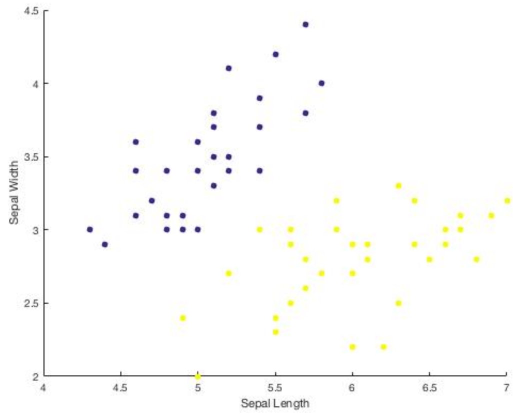
Pre − processing	*Accuracy(%)*		*Iterations*
	Train	*Test*	
No	100	100	7

Table 1: Perceptron Results on Iris Database

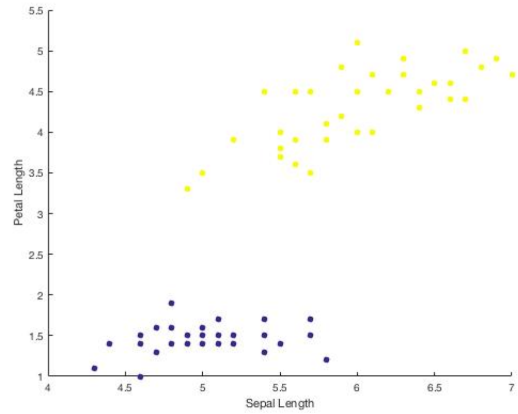
1.3 Training Perceptron with z-scoring

- Initial Weight = [0.01, 0.01, 0.01, 0.01]
- theta = 1
- Learning Rate = 1

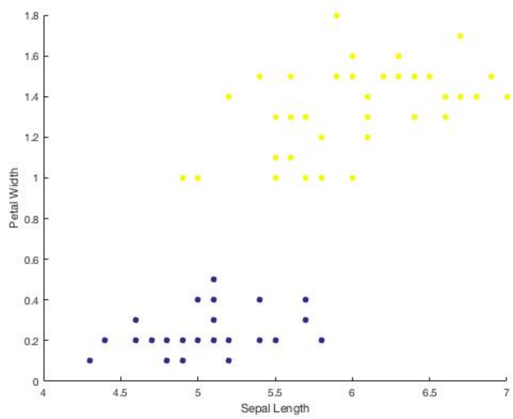
  Code for this section can be found in listing 2



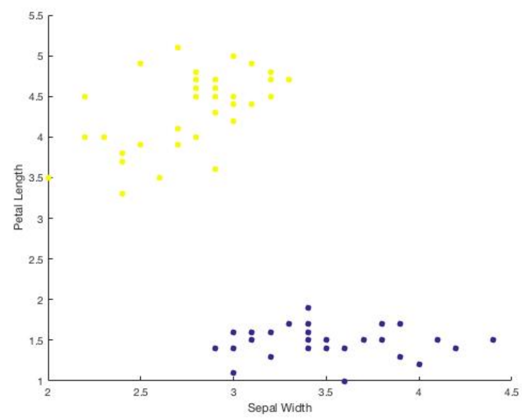
(a)



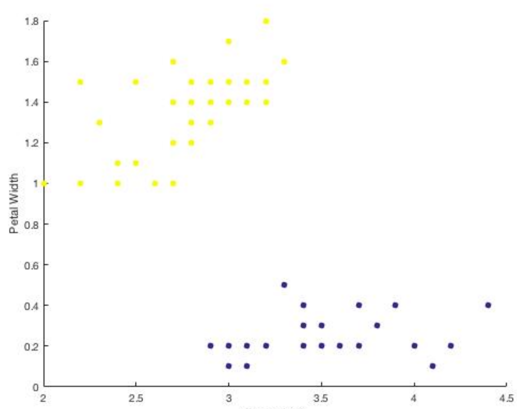
(b)



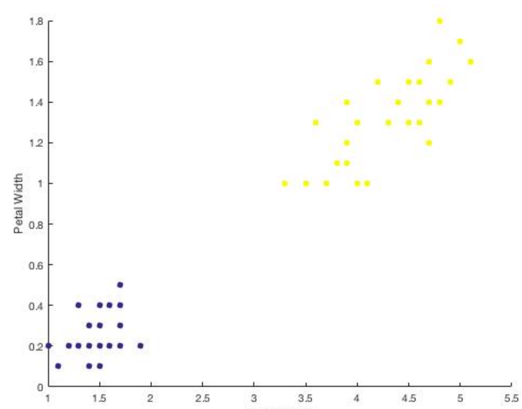
(c)



(d)



(e)



(f)

Figure 1: a-f Plot of between all combination of features

$Pre-processing$	$Accuracy(\%)$		$Iterations$
	$Train$	$Test$	
$Z-scoring$	100	100	1

Table 2: Perceptron Results on Iris Database

After using z-scoring the convergence was very fast and completed in just 1 iteration as compared to the non-processed data which took 7 iterations.

2 Feed Forward Neural Network

2.1 1 Hidden Layer

- No Pre-processing
- Number of hidden layers = 1
- Other layers = Input layer (Size = 784) and Output Layer (Size = 10)
- Initial Weight W = np.random.uniform(low=-np.sqrt(6.0/(D+h)), high=np.sqrt(6.0/(D+h)), size=(D, h))
- Initial Weight W2 = np.random.uniform(low=-np.sqrt(6.0/(k+h)), high=np.sqrt(6.0/(k+h)), size=(h, k))
- b = np.zeros((1,h)) #Bias1
- Initial Bias b2 = np.zeros((1,k)) #Bias2
- Weight Update Rules
  - W += -step_size * dW
  - b += -step_size * db
  - W2 += -step_size * dW2
  - b2 += -step_size * db2
- Step_size = 1e-1
- D = 784 #feature Dimension
- h = 100 # Size of hidden layer
- Activation function for output layer = softmax
- Activation function used for other layers = ReLu
- k=10; #Number of classes
- Iteration vs Loss is tabulated in table 3 and test/train accuracy in table 4

  Code for this section can be found in listing 3

Iteration	*Loss*
0	2.5940543069617954
50	0.747715206574926
100	0.5687253541322465
150	0.5102307359885822
200	0.4798862678931431
250	0.4603233746841243
300	0.44609411247272157
350	0.4348931330999727
400	0.4255995357224782
450	0.41765853956633214
500	0.4105747171711387
550	0.4041691236355091
600	0.39830549449054325
650	0.3928600317589662
700	0.3877499429289797
750	0.38291032761117877
790	0.3792199839536094

Table 3: Feed Forward Network 1 hidden layer Iteration vs Loss

Set	*Accuracy%*
train	0.9289
test	0.9275167

Table 4: Feed Forward Network 1 hidden layer Results

2.2 2 Hidden Layers with back propagation

- No Pre-processing
- Number of hidden layers = 2
- Other layers = Input layer (Size = 784) and Output Layer (Size = 10)
- Initial Weights
  - W = np.random.uniform(low=-np.sqrt(6.0/(D+h)), high=np.sqrt(6.0/(D+h)), size=(D, h))
  - W1 = np.random.uniform(low=-np.sqrt(6.0/(h+h)), high=np.sqrt(6.0/(h+h)), size=(h, h))
  - W2 = np.random.uniform(low=-np.sqrt(6.0/(k+h)), high=np.sqrt(6.0/(k+h)), size=(h, k))
- b = np.zeros((1,h)) #Bias1
- Initial Bias b2 = np.zeros((1,k)) #Bias2
- Weight Update Rules
  - W += -step_size * dW
  - b += -step_size * db
  - W1 += -step_size * dW1
  - b1 += -step_size * db1
  - W2 += -step_size * dW2
  - b2 += -step_size * db2
- Step_size = 1e-1
- D = 784 #feature Dimension
- h = 100 # Size of both hidden layer
- Activation function for output layer = softmax
- Activation function used for other layers = ReLu
- k=10; #Number of classes
- Iteration vs Loss is tabulated in table 5 and test/train accuracy in table 6
  Code for this section can be found in listing 4

Iteration	*Loss*
0	2.462944
50	1.108595
100	0.636004
150	0.487001
200	0.449254
250	0.432056
300	0.407610
350	0.393729
400	0.380419
450	0.370034
500	0.360481
550	0.351053
600	0.342832
650	0.335231
700	0.328106
750	0.321428
800	0.315152
850	0.309406
900	0.304001
950	0.298656
1000	0.293429
1050	0.288491
1100	0.283986

Table 5: Feed Forward Network 2 hidden layer Iteration vs Loss

Set	*Accuracy%*
train	97.67
test	96.19

Table 6: Feed Forward Network 2 hidden layer Results

We notice that after adding another layer each iteration takes longer time and convergence was slower but the accuracy was greater than while using only 1 hidden layer.

2.3 1 Hidden Layer with Regularization and Momentum

- No Pre-processing
- Number of hidden layers = 1
- Other layers = Input layer (Size = 784) and Output Layer (Size = 10)
- Step_size = 1e-1
- D = 784 #feature Dimension
- h = 100 # Size of hidden layer
- k=10; #Number of classes
- Activation function used = ReLu
- Initial Weight W = np.random.uniform(low=-np.sqrt(6.0/(D+h)), high=np.sqrt(6.0/(D+h)), size=(D, h))
- Initial Weight W2 = np.random.uniform(low=-np.sqrt(6.0/(k+h)), high=np.sqrt(6.0/(k+h)), size=(h, k))
- Initial bias b = np.zeros((1,h)) #Bias1
- Initial Bias b2 = np.zeros((1,k)) #Bias2
- Weight update rules:
  - dW2 += reg * W2
  - dW += reg * W
  - W += 0.9 * vW - step_size * dW
  - b += 0.9 * vB - step_size * db
  - W2 += 0.9 * vW2 - step_size * dW2
  - b2 += 0.9 * vB2 - step_size * db2
- Regularization strength = 1e-3
- Momentum = 0.9
- Iteration vs Loss is tabulated in table 7 and test/train accuracy in table 8

  Code for this section can be found in listing 3

Iteration	*Loss*
0	2.5617856828048344
50	0.44050862192365337
100	0.38326191771064244
150	0.3462090066700228
200	0.3156635527124032
250	0.29112958484072776
300	0.27164157543487233
350	0.2558534246926922
400	0.24278012560295464
450	0.2319293009798825
500	0.22274727625029428
550	0.21491594644338755
600	0.20820578323365566
650	0.20240326850698986
700	0.19736016076443563
720	0.19553153191873873

Table 7: Feed Forward Network 1 hidden layer and regularization and momentum Iteration vs Loss

Set	*Accuracy*%
train	97.33
test	96.68

Table 8: Feed Forward Network 1 hidden layer and regularisation and momentum Results

With only regularization enabled, the convergence as well as results improved only by a small factor but when we enabled both regularization and momentum, the convergence was fast and accuracy improved by around 4%. There was no effect on the run time for each iteration when compared to the section 2.1

3 Convolutional Neural Network

Convolutional Neural Networks or CNNs[2] exploit spatially-local correlation by enforcing a local connectivity pattern between neurons of adjacent layers. In other words, the inputs of hidden units in layer m are from a subset of units in layer m-1, units that have spatially contiguous receptive fields.

A feature map is obtained by repeated application of a function across sub-regions of the entire image, in other words, by convolution of the input image with a linear filter, adding a bias term and then applying a non-linear function.

3.1 Stochastic Gradient Descent

Model

- model.add(Convolution2D(32, 3, 3, border_mode='same', input_shape=(img_channels, img_rows, img_cols)))
- model.add(Activation('relu'))
- model.add(Convolution2D(32, 3, 3))
- model.add(Activation('relu'))
- model.add(MaxPooling2D(pool_size=(2, 2)))
- model.add(Dropout(0.25))
- model.add(Convolution2D(64, 3, 3, border_mode='same'))
- model.add(Activation('relu'))
- model.add(Convolution2D(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(nb_classes))
- model.add(Activation('softmax'))
- sgd = SGD(lr=0.01, decay=0, momentum=0.0, nesterov=False)
- model.compile(loss='categorical_crossentropy', optimizer=sgd, metrics=['accuracy'])

$S.No$	$Layers$	$Input\,Size\,to\,Layer$	$Activation\,Type$	$Dropout$	$PoolSize$	$Stride$	$No\,of\,Filters$
1	$Input$	$(3, 32, 32)$	$-$				
2	$Conv2D$	$(3, 32, 32)$	$-$			1	32
3	$Activation$	$(32, 32, 32)$	$RELU$				
4	$Conv2D$	$(32, 32, 32)$	$-$			1	32
5	$Activation$	$(32, 30, 30)$	$RELU$				
6	$MaxPool2D$	$(32, 30, 30)$	$-$		$(2, 2)$		
7	$Dropout$	$(32, 15, 15)$	$-$	0.25			
8	$Conv2D$	$(32, 15, 15)$	$-$			1	64
9	$Activation$	$(64, 15, 15)$	$RELU$				
10	$Conv2D$	$(64, 15, 15)$	$-$			1	64
11	$Activation$	$(64, 13, 13)$	$RELU$				
12	$MaxPool2D$	$(64, 13, 13)$	$-$		$(2, 2)$		
13	$Dropout$	$(64, 6, 6)$	$-$	0.25			
14	$Flatten$	$(64, 6, 6)$	$-$				
15	$Dense$	2304	$-$				
16	$Activation$	512	$RELU$				
17	$Dropout$	512	$-$	0.5			
18	$Dense$	512	$-$				
19	$Activation$	10	$Softmax$				

(a)Layer Details

- batch_size = 32
- nb_classes = 10
- nb_epoch = 20
- img_rows, img_cols = 32, 32, img_channels = 3
- Learning Rate = 0.01

(b)Other Details

Table 9: Network Architecture

Figure 2 shows the loss and accuracy in each epoch. 20th epoch gets an Train accuracy of 80.214% and Test accuracy 74.84%. Code for this section can be found in listing 5.

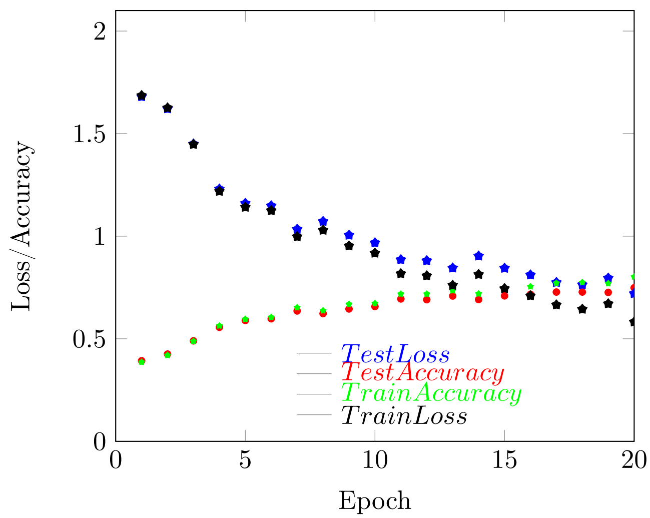


Figure 2: Stochastic Gradient CNN Results

3.2 Batch Normalization

Model

- model.add(Convolution2D(32, 3, 3, border_mode='same', input_shape=(img_channels, img_rows, img_cols)))
- model.add(Activation('relu'))
- model.add(Convolution2D(32, 3, 3))
- model.add(Activation('relu'))
- model.add(MaxPooling2D(pool_size=(2, 2)))
- model.add(Dropout(0.25))
- model.add(Convolution2D(64, 3, 3, border_mode='same'))
- model.add(BatchNormalization())
- model.add(Activation('relu'))
- model.add(Convolution2D(64, 3, 3))
- model.add(BatchNormalization())
- 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(nb_classes))
- model.add(Activation('softmax'))
- sgd = SGD(lr=0.01, decay=0, momentum=0.0, nesterov=False)
- model.compile(loss='categorical_crossentropy', optimizer=sgd, metrics=['accuracy'])

$S.No$	$Layers$	$Input\,Size\,to\,Layer$	$Activation\,Type$	$Dropout$	$PoolSize$	$Stride$	$No\,of\,Filters$
1	$Input$	$(3, 32, 32)$	$-$				
2	$Conv2D$	$(3, 32, 32)$	$-$			1	32
3	$Activation$	$(32, 32, 32)$	$RELU$				
4	$Conv2D$	$(32, 32, 32)$	$-$			1	32
5	$Activation$	$(32, 30, 30)$	$RELU$				
6	$MaxPool2D$	$(32, 30, 30)$	$-$		$(2, 2)$		
7	$Dropout$	$(32, 15, 15)$	$-$	0.25			
8	$Conv2D$	$(32, 15, 15)$	$-$			1	64
9	$Activation$	$(64, 15, 15)$	$RELU$				
10	$Batch\,Normalization$	$(64, 15, 15)$	$-$				
11	$Conv2D$	$(64, 15, 15)$	$-$			1	64
12	$Activation$	$(64, 13, 13)$	$RELU$				
13	$Batch\,Normalization$	$(64, 13, 13)$	$-$				
14	$MaxPool2D$	$(64, 13, 13)$	$-$		$(2, 2)$		
15	$Dropout$	$(64, 6, 6)$	$-$	0.25			
16	$Flatten$	$(64, 6, 6)$	$-$				
17	$Dense$	2304	$-$				
18	$Activation$	512	$RELU$				
19	$Dropout$	512	$-$	0.5			
20	$Dense$	512	$-$				
21	$Activation$	10	$Softmax$				

(a)Layer Details

- batch_size = 32
- nb_classes = 10
- Learning Rate = 0.01
- nb_epoch = 20
- img_rows, img_cols = 32, 32, img_channels = 3

(b)Other Details

Table 10: Network Architecture

Figure 3below shows the loss and accuracy in each epoch. 20th epoch gets an Train accuracy of 82.608% and Test accuracy 76.08%. Code for this section can be found in listing 6.

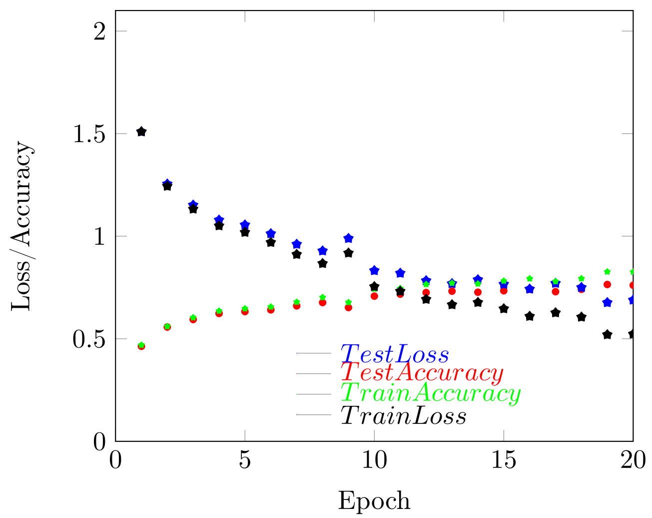


Figure 3: Batch Normalization CNN Results

Adding two batch normalization layers slowed down the run time for each epoch by a factor of 5, The convergence happened in about same number of epoch and accuracy improved by around 1.5%.

3.3 Replace the fully connected layer by average pooling layer

Model

- model.add(Convolution2D(32, 3, 3, border_mode='same', input_shape=(img_channels, img_rows, img_cols)))
- model.add(Activation('relu'))
- model.add(BatchNormalization())
- model.add(Convolution2D(32, 3, 3))
- model.add(Activation('relu'))
- model.add(BatchNormalization())
- model.add(MaxPooling2D(pool_size=(2, 2)))
- model.add(Dropout(0.25))
- model.add(Convolution2D(64, 3, 3, border_mode='same'))
- model.add(Activation('relu')) model.add(BatchNormalization())
- model.add(Convolution2D(64, 3, 3))
- model.add(Activation('relu'))
- model.add(BatchNormalization())
- model.add(MaxPooling2D(pool_size=(2, 2)))
- model.add(Dropout(0.25))
- model.add(AveragePooling2D(pool_size=(2, 2), strides=None, border_mode='valid', dim_ordering='th'))
- model.add(Activation('relu'))
- model.add(Dropout(0.5))
- model.add(Flatten())
- model.add(Dense(nb_classes))
- model.add(Activation('softmax'))

S.No	*Layers*	*Input Size to Layer*	*Activation Type*	*Dropout*	*PoolSize*	*Stride*	*No of Filters*
1	*Input*	$(3, 32, 32)$	–				
2	*Conv2D*	$(3, 32, 32)$	–			1	32
3	*Activation*	$(32, 32, 32)$	*RELU*				
4	*Conv2D*	$(32, 32, 32)$	–			1	32
5	*Activation*	$(32, 30, 30)$	*RELU*				
6	*MaxPool2D*	$(32, 30, 30)$	–		$(2, 2)$		
7	*Dropout*	$(32, 15, 15)$	–	0.25			
8	*Conv2D*	$(32, 15, 15)$	–			1	64
9	*Activation*	$(64, 15, 15)$	*RELU*				
11	*Conv2D*	$(64, 15, 15)$	–			1	64
12	*Activation*	$(64, 13, 13)$	*RELU*				
14	*MaxPool2D*	$(64, 13, 13)$	–		$(2, 2)$		
15	*Dropout*	$(64, 6, 6)$		0.25			
16	*Average Pool*	$(64, 6, 6)$					
17	*Activation*	$(64, 3, 3)$					
18	*Dropout*	$(64, 3, 3)$	–	0.5			
19	*Flatten*	$(64, 3, 3)$	–				
20	*Dense*	576	–				
21	*Activation*	10	*RELU*				

(a)Layer Details

-
    - batch_size = 32
    - nb_classes = 10
    - nb_epoch = 20
    - Learning Rate = 0.01
    - img_rows, img_cols = 32, 32, img_channels = 3

(b)Other Details

Table 11: Network Architecture

Figure 4below shows the loss and accuracy in each epoch. 20th epoch gets an Train accuracy of 70.798% and Test accuracy 69.74%. Code for this section can be found in listing 7.

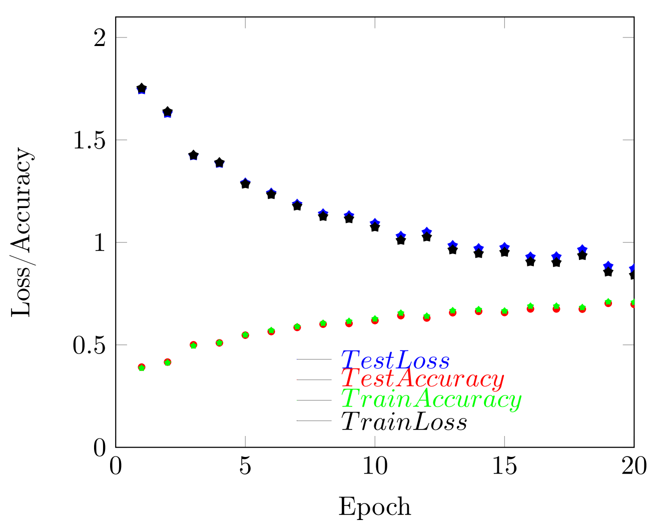


Figure 4: Average Pool Layer CNN Result

Replacing the fully connected layer by an average pooling layer degraded the performance by around 5%.

3.4 Adaptive Gradient

- adg = Adagrad(lr=0.01, epsilon=1e-06)
- model.compile(loss='categorical_crossentropy', optimizer=adg, metrics=['accuracy'])

Architecture for this is same as Table 9

Apart from that the following parameters were used for adaptive gradient

- Learning Rate = 0.01
- epsilon=1e-06

Figure 5below shows the loss and accuracy in each epoch. 20th epoch gets an Train accuracy of 88.952% and Test accuracy 78.32%. Code for this section can be found in listing 8.

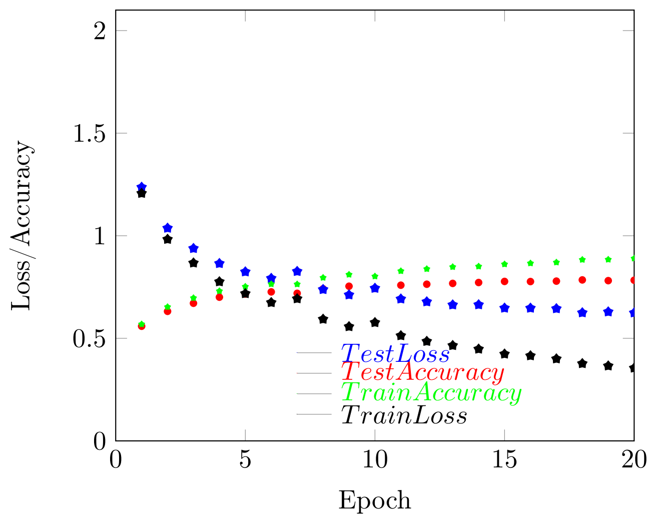


Figure 5: Adaptive Gradient CNN Results

This was the best optimization in terms of performance and speed. Convergence was faster and the run time of each epoch remained un-changed as compared to section 3.1

3.5 Nesterovs Accelerated Gradient

- sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
- model.compile(loss='categorical_crossentropy', optimizer=sgd, metrics=['accuracy'])

Architecture for this is same as Table 9

Apart from that the following parameters were used for Nesterovs Accelerated Gradient

- Learning Rate = 0.01
- Epsilon=1e-06
- Decay=1e-6
- Momentum=0.9

Figure 6below shows the loss and accuracy in each epoch. 20th epoch gets an Train accuracy of 87.698% and Test accuracy 77.06%. Code for this section can be found in listing 9.

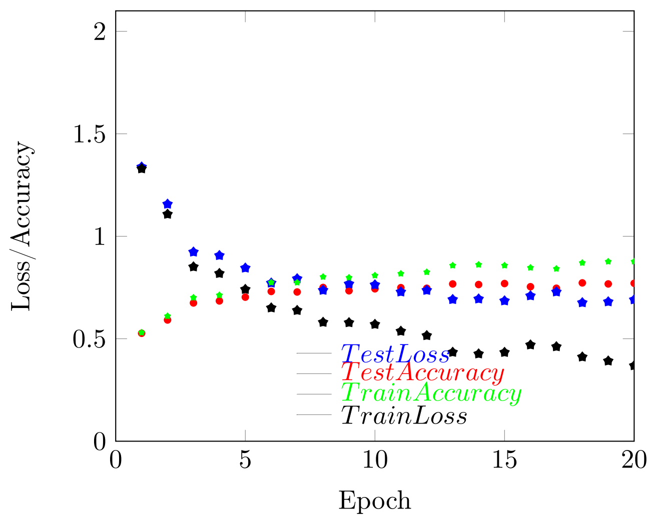


Figure 6: Nesterovs accelerated Gradient CNN Results

Nesterovs Accelerated Gradient optimization was similar in terms of performance and speed to adaptive gradient. Convergence was faster and the run time of each epoch remained un-changed as compared to section 3.1

3.6 RMSprop

- rms = RMSprop(lr=0.001, rho=0.9, epsilon=1e-06)
- model.compile(loss='categorical_crossentropy', optimizer=rms, metrics=['accuracy'])

Architecture for this is same as Table 9

Apart from that the following parameters were used for Nesterovs Accelerated Gradient

- Learning Rate = 0.01
- rho=0.9
- epsilon=1e-06

Figure 7below shows the loss and accuracy in each epoch. 20th epoch gets an Train accuracy of 82.344% and Test accuracy 75.06%. Code for this section can be found in listing 10.

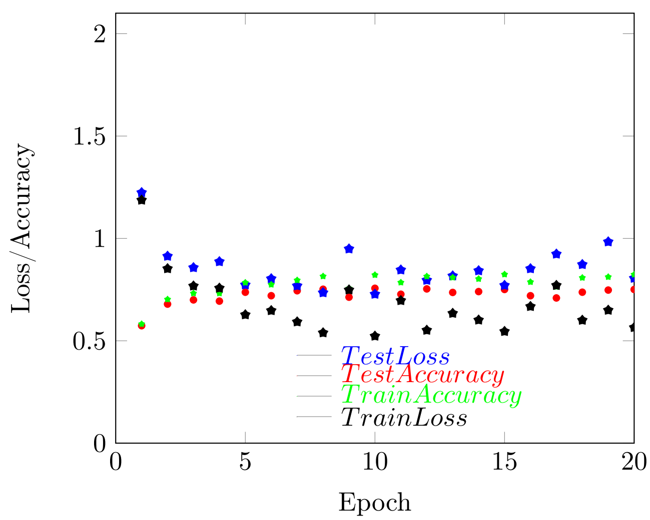


Figure 7: RMSProp CNN results

RMSprop optimization was slightly better than stochastic gradient. Convergence was faster but the accuracy improved by a small factor. Run time for each epoch was similar to when not using this optimization.

4 References

1. https://github.com/fchollet/keras

5 Appendix A:: Code

```
% train_raw = importdata('iris/train');
% test_raw = importdata('iris/test');
%
% for i=1:length(train_raw)
%     string = strsplit(char(train_raw(i)), ',');
%     for j=1:length(string)
%         cells = strsplit(char(string(j)));
%         train(i,:)= cells;
%         for k=1:length(cells)
%         end
%     end
% end
%
% for i=1:length(test_raw)
%     string = strsplit(char(test_raw(i)), ',');
%     for j=1:length(string)
%         cells = strsplit(char(string(j)));
%         test(i,:)= cells;
%         for k=1:length(cells)
%         end
%     end
% end

train_feat = zeros (70, 4);
train_label = ones (70,1);

test_feat = zeros (30,4);
test_label = ones (30,1);

for i=1:70
    for j=1:4
        train_feat(i,j) = str2double(train(i,j));
    end
    if (strcmp(train(i,5), 'Iris-setosa'))
        train_label(i,1) = -1;
    end
end

for i=1:30
    for j=1:4
        test_feat(i,j) = str2double(test(i,j));
    end
    if (strcmp(test(i,5), 'Iris-setosa'))
        test_label(i,1) = -1;
    end
end

%%
figure(1);
scatter(train_feat(:,1), train_feat(:,2), [], train_label(:,1),'filled');
xlabel('Sepal Length');
```

```
ylabel('Sepal Width');
figure(2);
scatter(train_feat(:,1), train_feat(:,3), [], train_label(:,1),'filled');
xlabel('Sepal Length');
ylabel('Petal Length');
figure(3);
scatter(train_feat(:,1), train_feat(:,4), [], train_label(:,1),'filled');
xlabel('Sepal Length');
ylabel('Petal Width');
figure(4);
scatter(train_feat(:,2), train_feat(:,3), [], train_label(:,1),'filled');
xlabel('Sepal Width');
ylabel('Petal Length');
figure(5);
scatter(train_feat(:,2), train_feat(:,4), [], train_label(:,1),'filled');
xlabel('Sepal Width');
ylabel('Petal Width');
figure(6);
scatter(train_feat(:,3), train_feat(:,4), [], train_label(:,1),'filled');
xlabel('Petal Length');
ylabel('Petal Width');
```

Listing 1: Answer1

```
import numpy as np

def dot_product(a, b):
    return np.array(np.dot(np.asarray(a), np.asarray(b)))


def decision( x, w, theta ):
    return (dot_product(x, w) > theta)


def perceptron( training_data ):
    theta = 1
    iteration = 0
    weights = [0.01, 0.01, 0.01, 0.01]
    converged = False

    while not converged:
        correct_count = 0;
        for key, val in training_data.iteritems():
            d = decision(key, weights, theta)
            if d == val:
                correct_count +=1
                continue
            elif d == False and val == True:
                theta -= 1
                iteration += 1
                for i in range(len(key)):
                    weights[i] += key[i]

            elif d == True  and val == False:
                theta += 1
                iteration += 1
                for i in range(len(key)):
                    weights[i] -= key[i]
```

```
        if (correct_count == len(training_data)):
            break;

    print ("Converged in Iterations {}".format(iteration))
    return weights, theta

lines = [line.rstrip('\n') for line in open('iris/iris_train.data')]

train = {}
for line in lines:
    words = line.split(',')
    tup   = (float(words[0]),float(words[1]),float(words[2]),float(words[3]),)
    if (words[4] == 'Iris-setosa'):
        train[tup] = True
    else:
        train[tup] = False

lines1 = [line.rstrip('\n') for line in open('iris/iris_test.data')]

test = {}
for line in lines1:
    words = line.split(',')
    tup   = (float(words[0]),float(words[1]),float(words[2]),float(words[3]),)
    if (words[4] == 'Iris-setosa'):
        test[tup] = True
    else:
        test[tup] = False

#print (train)
#print (test)


train_feat = np.asarray(train.keys())
test_feat = np.asarray(test.keys())

train_mean = np.mean(train_feat, axis=0)
test_mean = np.mean(test_feat, axis=0)
train_std = np.std(train_feat, axis=0)
test_std = np.std(test_feat, axis=0)

train_z = {}
for key, val in train.iteritems():
        key_new = tuple(np.array((key-train_mean)/train_std))
        train_z[key_new] = val

test_z = {}
for key, val in test.iteritems():
        key_new = tuple(np.array((key-test_mean)/test_std))
        test_z[key_new] = val



weights, theta = perceptron( train )
total_correct = 0
for key, val in test.iteritems():
    d = decision(key, weights, theta)
    if d == val:
        total_correct += 1
```

```
print ("No Z scoring\n")
print ("Total Correct = {}, out of {}".format(total_correct, len(test)))

print ("\n\n")


weights1, theta1 = perceptron( train_z )
total_correct = 0
for key, val in test_z.iteritems():
    d = decision(key, weights1, theta1)
    if d == val:
        total_correct += 1

print ("With Z scoring\n")
print ("Total Correct = {}, out of {}".format(total_correct, len(test_z)))
```

Listing 2: Perceptron

```
import numpy as np
from keras.datasets import mnist

# initialize parameters randomly
D = 784
h = 100 # size of hidden layer
W = np.random.uniform(low=-np.sqrt(6.0/(D+h)), high=np.sqrt(6.0/(D+h)), size=(D, h
    ))
b = np.zeros((1,h))
k=10;
W2 = np.random.uniform(low=-np.sqrt(6.0/(k+h)), high=np.sqrt(6.0/(k+h)), size=(h,
    k))
b2 = np.zeros((1,k))

# some hyperparameters
step_size = 1e-1
reg = 1e-3 # regularization strength

###################
#READ MNIST data
(X, y), (test_feat, test_label) = mnist.load_data()

X = np.array(X, np.float)
X = X.reshape(X.shape[0], 784)

test_feat = np.array(test_feat, np.float)
test_feat = test_feat.reshape(test_feat.shape[0], 784)

X = (X - 128)/255.0
test_feat = (test_feat-128)/255.0
###################

vW = 0
vB = 0
vW2 = 0
vB2 = 0
# gradient descent loop
num_examples = X.shape[0]

prev_loss = 0.0;
enable_regularization = 1
```

```
enable_momentum = 1

#%%
for i in range(10000):

  # evaluate class scores, [N x K]
  hidden_layer = np.maximum(0, np.dot(X, W) + b) # note, ReLU activation
  scores = np.dot(hidden_layer, W2) + b2

  # compute the class probabilities
  exp_scores = np.exp(scores)
  probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True) # [N x K]

  # compute the loss: average cross-entropy loss and regularization
  corect_logprobs = -np.log(probs[range(num_examples),y])
  data_loss = np.sum(corect_logprobs)/num_examples
  reg_loss = 0.5*reg*np.sum(W*W) + 0.5*reg*np.sum(W2*W2)
  loss = data_loss + reg_loss
  if i % 10 == 0:
      print('iteration {} :: loss {}'.format(i, loss))

  if (loss - prev_loss < 0.001):
      print('iteration {} :: loss {}'.format(i, loss))
      prev_loss = loss
      break

  # compute the gradient on scores
  dscores = probs
  dscores[range(num_examples),y] -= 1
  dscores /= num_examples

  # backpropate the gradient to the parameters
  # first backprop into parameters W2 and b2
  dW2 = np.dot(hidden_layer.T, dscores)
  db2 = np.sum(dscores, axis=0, keepdims=True)
  # next backprop into hidden layer
  dhidden = np.dot(dscores, W2.T)
  # backprop the ReLU non-linearity
  dhidden[hidden_layer <= 0] = 0
  # finally into W,b
  dW = np.dot(X.T, dhidden)
  db = np.sum(dhidden, axis=0, keepdims=True)

#  # add regularization gradient contribution
  if enable_regularization == 1:
      dW2 += reg * W2
      dW += reg * W


###Add momentum
  vW = 0.9 * vW - step_size * dW
  vW2 = 0.9 * vW2 - step_size * dW2
  vB = 0.9 * vB - step_size * db
  vB2 = 0.9 * vB2 - step_size * db2

  # perform a parameter update
  if enable_momentum == 1:
      W += vW
      b += vB
```

```
        W2 += vW2
        b2 += vB2
    else:
        W += -step_size * dW
        b += -step_size * db
        W2 += -step_size * dW2
        b2 += -step_size * db2

##############
#%%
hidden_layer = np.maximum(0, np.dot(test_feat, W) + b)
scores = np.dot(hidden_layer, W2) + b2
predicted_class = np.argmax(scores, axis=1)
print ('test accuracy: {}'.format(np.mean(predicted_class == test_label)))

hidden_layer = np.maximum(0, np.dot(X, W) + b)
scores = np.dot(hidden_layer, W2) + b2
predicted_class = np.argmax(scores, axis=1)
print ('train accuracy: {}'.format(np.mean(predicted_class == y)))
```

Listing 3: Answer2(part 1 and 3)

```
import numpy as np
from keras.datasets import mnist

# initialize parameters randomly
D = 784
h = 100 # size of hidden layer
W = np.random.uniform(low=-np.sqrt(6.0/(D+h)), high=np.sqrt(6.0/(D+h)), size=(D, h
    ))
b = np.zeros((1,h))
k=10;

b1 = np.zeros((1,h))
W1 = np.random.uniform(low=-np.sqrt(6.0/(h+h)), high=np.sqrt(6.0/(h+h)), size=(h,
    h))
W2 = np.random.uniform(low=-np.sqrt(6.0/(k+h)), high=np.sqrt(6.0/(k+h)), size=(h,
    k))
b2 = np.zeros((1,k))

# some hyperparameters
step_size = 1e-1
reg = 1e-3 # regularization strength

###################
#READ MNIST data
(X, y), (test_feat, test_label) = mnist.load_data()

X = np.array(X, np.float)
X = X.reshape(X.shape[0], 784)

test_feat = np.array(test_feat, np.float)
test_feat = test_feat.reshape(test_feat.shape[0], 784)

X = (X - 128)/255.0
test_feat = (test_feat-128)/255.0
###################

vW = 0
```

```
vB = 0
vW2 = 0
vB2 = 0
# gradient descent loop
num_examples = X.shape[0]

W_v = 0
W1_v = 0
W2_v = 0
b1_v = 0
b2_v = 0
mu = 0.9

i = 0

loss_prev = 10

for i in range(600):
    ## FeedForward Code
    hidden_layer1 = np.maximum(0, np.dot(X, W) + b) # note, ReLU activation
    hidden_layer2 = np.maximum(0, np.dot(hidden_layer1, W1) + b1)
    scores = np.dot(hidden_layer2, W2) + b2

    # compute the class probabilities
    exp_scores = np.exp(scores)
    probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True) # [N x K]
    corect_logprobs = -np.log(probs[range(num_examples),y])
    data_loss = np.sum(corect_logprobs)/num_examples
    reg_loss = 0.5*reg*np.sum(W*W) + 0.5*reg*np.sum(W2*W2)
    loss = data_loss + reg_loss

    loss_prev = loss
    if i % 10 == 0:
        print ("iteration {}: loss {}".format(i, loss))

    loss_prev=loss
    # compute the gradient on scores
    dscores = probs
    dscores[range(num_examples),y] -= 1
    dscores /= num_examples

    # backpropate the gradient to the parameters
    # first backprop into parameters W2 and b2
    dW2 = np.dot(hidden_layer2.T, dscores)
    db2 = np.sum(dscores, axis=0, keepdims=True)
    # next backprop into hidden layer 2
    dhidden2 = np.dot(dscores, W2.T)
    # backprop the ReLU non-linearity
    dhidden2[hidden_layer2 <= 0] = 0

    dW1 = np.dot(hidden_layer1.T,dhidden2)
    db1 = np.sum(dhidden2,axis=0,keepdims=True)
    # next backprop into hidden layer 1
    dhidden1 = np.dot(dhidden2, W1.T)
    # backprop the ReLU non-linearity
    dhidden1[hidden_layer1 <= 0] = 0


    # finally into W,b
```

```
    dW = np.dot(X.T, dhidden1)
    db = np.sum(dhidden1, axis=0, keepdims=True)

    # add regularization gradient contribution
    dW2 += reg * W2
    dW1 += reg*W1
    dW += reg * W

    # perform a parameter update
    W += -step_size * dW
    b += -step_size * db
    W1 += -step_size * dW1
    b1 += -step_size * db1

    W2 += -step_size * dW2
    b2 += -step_size * db2
    i+=1
#%%
X_test =(test_feat-128)/255.0

y_2 = test_label
y_1 = y

hidden_layer1 = np.maximum(0, np.dot(X_test, W) + b)
hidden_layer2 = np.maximum(0, np.dot(hidden_layer1, W1) + b1)

scores = np.dot(hidden_layer2, W2) + b2
predicted_class = np.argmax(scores, axis=1)
print ('testing accuracy: {}'.format(np.mean(predicted_class == y_2)) )


hidden_layer1 = np.maximum(0, np.dot(X, W) + b)
hidden_layer2 = np.maximum(0, np.dot(hidden_layer1, W1) + b1)

scores = np.dot(hidden_layer2, W2) + b2
predicted_class = np.argmax(scores, axis=1)
print ('train accuracy: {}'.format(np.mean(predicted_class == y_1)) )
```

Listing 4: Answer2 (part 2)

```
'''Train a simple deep CNN on the CIFAR10 small images dataset.

GPU run command:
    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).

Note: the data was pickled with Python 2, and some encoding issues might prevent
    you
from loading it in Python 3. You might have to load it in Python 2,
save it in a different format, load it in Python 3 and repickle it.
'''

from __future__ import print_function
from keras.datasets import cifar10
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Activation, Flatten
```

```
from keras.layers.convolutional import Convolution2D, MaxPooling2D
from keras.optimizers import SGD
from keras.utils import np_utils
from keras.callbacks import Callback

batch_size = 32
nb_classes = 10
nb_epoch = 200
data_augmentation = True

# input image dimensions
img_rows, img_cols = 32, 32
# the CIFAR10 images are RGB
img_channels = 3

# the data, shuffled and split between train and test sets
(X_train, y_train), (X_test, y_test) = cifar10.load_data()

class TrainCallback(Callback):
    def __init__(self, test_data):
        self.test_data = test_data

    def on_epoch_end(self, epoch, logs={}):
        x, y = self.test_data
        loss, acc = self.model.evaluate(x, y, verbose=0)
        print('\nTrain loss: {}, Train acc: {}'.format(loss, acc))



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 = np_utils.to_categorical(y_train, nb_classes)
Y_test = np_utils.to_categorical(y_test, nb_classes)

model = Sequential()


#Input layer
model.add(Convolution2D(32, 3, 3, border_mode='same',
                        input_shape=(img_channels, img_rows, img_cols)))
model.add(Activation('relu'))

#Conv layer 1
model.add(Convolution2D(32, 3, 3))
model.add(Activation('relu'))

#Pool Layer 1
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

#Conv Layer 2
model.add(Convolution2D(64, 3, 3, border_mode='same'))
model.add(Activation('relu'))

#Conv Layer 3
model.add(Convolution2D(64, 3, 3))
```

```
model.add(Activation('relu'))

#Pool Layer 2
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())

#Dense Layer 1
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dropout(0.5))

#Output layer
model.add(Dense(nb_classes))
model.add(Activation('softmax'))

# let's train the model using SGD + momentum (how original).
sgd = SGD(lr=0.01, decay=0, momentum=0.0, nesterov=False)
model.compile(loss='categorical_crossentropy',
              optimizer=sgd,
              metrics=['accuracy'])

X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train /= 255
X_test /= 255

if not data_augmentation:
    print('Not using data augmentation.')
    model.fit(X_train, Y_train,
              batch_size=batch_size,
              nb_epoch=nb_epoch,
              validation_data=(X_test, Y_test),
              shuffle=True)
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
            total width)
        height_shift_range=0.1,  # randomly shift images vertically (fraction of
            total height)
        horizontal_flip=True,  # randomly flip images
        vertical_flip=False)  # randomly flip images

    # compute quantities required for featurewise 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),
                        samples_per_epoch=X_train.shape[0],
                        nb_epoch=nb_epoch,
                        validation_data=(X_test, Y_test),
                        callbacks=[TrainCallback((X_train, Y_train))])
```

Listing 5: Answer3(a)

```
'''Train a simple deep CNN on the CIFAR10 small images dataset.

GPU run command:
    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).

Note: the data was pickled with Python 2, and some encoding issues might prevent
    you
from loading it in Python 3. You might have to load it in Python 2,
save it in a different format, load it in Python 3 and repickle it.
'''

from __future__ import print_function
from keras.datasets import cifar10
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Activation, Flatten
from keras.layers.convolutional import Convolution2D, MaxPooling2D
from keras.optimizers import SGD
from keras.utils import np_utils
from keras.callbacks import Callback
from keras.layers.normalization import BatchNormalization

batch_size = 32
nb_classes = 10
nb_epoch = 200
data_augmentation = True

# input image dimensions
img_rows, img_cols = 32, 32
# the CIFAR10 images are RGB
img_channels = 3

# the data, shuffled and split between train and test sets
(X_train, y_train), (X_test, y_test) = cifar10.load_data()

class TrainCallback(Callback):
    def __init__(self, test_data):
        self.test_data = test_data

    def on_epoch_end(self, epoch, logs={}):
        x, y = self.test_data
        loss, acc = self.model.evaluate(x, y, verbose=0)
        print('\nTrain loss: {}, Train acc: {}'.format(loss, acc))



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 = np_utils.to_categorical(y_train, nb_classes)
Y_test = np_utils.to_categorical(y_test, nb_classes)

model = Sequential()


#Input layer
model.add(Convolution2D(32, 3, 3, border_mode='same',
                        input_shape=(img_channels, img_rows, img_cols)))
model.add(Activation('relu'))
model.add(BatchNormalization())

#Conv layer 1
model.add(Convolution2D(32, 3, 3))
model.add(Activation('relu'))
model.add(BatchNormalization())

#Pool Layer 1
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

#Conv Layer 2
model.add(Convolution2D(64, 3, 3, border_mode='same'))
model.add(Activation('relu'))
model.add(BatchNormalization())

#Conv Layer 3
model.add(Convolution2D(64, 3, 3))
model.add(Activation('relu'))
model.add(BatchNormalization())

#Pool Layer 2
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())

#Dense Layer 1
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dropout(0.5))

#Output layer
model.add(Dense(nb_classes))
model.add(Activation('softmax'))

# let's train the model using SGD + momentum (how original).
sgd = SGD(lr=0.01, decay=0, momentum=0.0, nesterov=False)
model.compile(loss='categorical_crossentropy',
              optimizer=sgd,
              metrics=['accuracy'])

X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train /= 255
X_test /= 255
```

```
if not data_augmentation:
    print('Not using data augmentation.')
    model.fit(X_train, Y_train,
              batch_size=batch_size,
              nb_epoch=nb_epoch,
              validation_data=(X_test, Y_test),
              shuffle=True)
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
            total width)
        height_shift_range=0.1,  # randomly shift images vertically (fraction of
            total height)
        horizontal_flip=True,  # randomly flip images
        vertical_flip=False)  # randomly flip images

    # compute quantities required for featurewise 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),
                        samples_per_epoch=X_train.shape[0],
                        nb_epoch=nb_epoch,
                        validation_data=(X_test, Y_test),
                        callbacks=[TrainCallback((X_train, Y_train))])
```

Listing 6: Answer3(b)

```
'''Train a simple deep CNN on the CIFAR10 small images dataset.

GPU run command:
    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).

Note: the data was pickled with Python 2, and some encoding issues might prevent
   you
from loading it in Python 3. You might have to load it in Python 2,
save it in a different format, load it in Python 3 and repickle it.
'''

from __future__ import print_function
from keras.datasets import cifar10
from keras.preprocessing.image import ImageDataGenerator
```

```
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Activation, Flatten
from keras.layers.convolutional import Convolution2D, MaxPooling2D,
    AveragePooling2D
from keras.optimizers import SGD
from keras.utils import np_utils
from keras.callbacks import Callback
from keras.layers.normalization import BatchNormalization

batch_size = 32
nb_classes = 10
nb_epoch = 200
data_augmentation = True

# input image dimensions
img_rows, img_cols = 32, 32
# the CIFAR10 images are RGB
img_channels = 3

# the data, shuffled and split between train and test sets
(X_train, y_train), (X_test, y_test) = cifar10.load_data()

class TrainCallback(Callback):
    def __init__(self, test_data):
        self.test_data = test_data

    def on_epoch_end(self, epoch, logs={}):
        x, y = self.test_data
        loss, acc = self.model.evaluate(x, y, verbose=0)
        print('\nTrain loss: {}, Train acc: {}'.format(loss, acc))



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 = np_utils.to_categorical(y_train, nb_classes)
Y_test = np_utils.to_categorical(y_test, nb_classes)

model = Sequential()


#Input layer
model.add(Convolution2D(32, 3, 3, border_mode='same',
                        input_shape=(img_channels, img_rows, img_cols)))
model.add(Activation('relu'))
model.add(BatchNormalization())

#Conv layer 1
model.add(Convolution2D(32, 3, 3))
model.add(Activation('relu'))
model.add(BatchNormalization())

#Pool Layer 1
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
```

```
#Conv Layer 2
model.add(Convolution2D(64, 3, 3, border_mode='same'))
model.add(Activation('relu'))
model.add(BatchNormalization())

#Conv Layer 3
model.add(Convolution2D(64, 3, 3))
model.add(Activation('relu'))
model.add(BatchNormalization())

#Pool Layer 2
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

#Average Pool Layer
model.add(AveragePooling2D(pool_size=(2, 2), strides=None, border_mode='valid',
    dim_ordering='th'))


model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Flatten())

#Output layer
model.add(Dense(nb_classes))
model.add(Activation('softmax'))

# let's train the model using SGD + momentum (how original).
sgd = SGD(lr=0.01, decay=0, momentum=0.0, nesterov=False)
model.compile(loss='categorical_crossentropy',
              optimizer=sgd,
              metrics=['accuracy'])

X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train /= 255
X_test /= 255

if not data_augmentation:
    print('Not using data augmentation.')
    model.fit(X_train, Y_train,
              batch_size=batch_size,
              nb_epoch=nb_epoch,
              validation_data=(X_test, Y_test),
              shuffle=True)
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
```

```
            total width)
        height_shift_range=0.1,  # randomly shift images vertically (fraction of
            total height)
        horizontal_flip=True,  # randomly flip images
        vertical_flip=False)  # randomly flip images

    # compute quantities required for featurewise 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),
                        samples_per_epoch=X_train.shape[0],
                        nb_epoch=nb_epoch,
                        validation_data=(X_test, Y_test),
                        callbacks=[TrainCallback((X_train, Y_train))])
```

Listing 7: Answer3(c)

```
'''Train a simple deep CNN on the CIFAR10 small images dataset.

GPU run command:
    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).

Note: the data was pickled with Python 2, and some encoding issues might prevent
    you
from loading it in Python 3. You might have to load it in Python 2,
save it in a different format, load it in Python 3 and repickle it.
'''

from __future__ import print_function
from keras.datasets import cifar10
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Activation, Flatten
from keras.layers.convolutional import Convolution2D, MaxPooling2D
from keras.optimizers import SGD
from keras.optimizers import Adagrad
from keras.utils import np_utils
from keras.callbacks import Callback

batch_size = 32
nb_classes = 10
nb_epoch = 200
data_augmentation = True

# input image dimensions
img_rows, img_cols = 32, 32
# the CIFAR10 images are RGB
img_channels = 3

# the data, shuffled and split between train and test sets
(X_train, y_train), (X_test, y_test) = cifar10.load_data()

class TrainCallback(Callback):
```

```
    def __init__(self, test_data):
        self.test_data = test_data

    def on_epoch_end(self, epoch, logs={}):
        x, y = self.test_data
        loss, acc = self.model.evaluate(x, y, verbose=0)
        print('\nTrain loss: {}, Train acc: {}'.format(loss, acc))


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 = np_utils.to_categorical(y_train, nb_classes)
Y_test = np_utils.to_categorical(y_test, nb_classes)

model = Sequential()


#Input layer
model.add(Convolution2D(32, 3, 3, border_mode='same',
                        input_shape=(img_channels, img_rows, img_cols)))
model.add(Activation('relu'))

#Conv layer 1
model.add(Convolution2D(32, 3, 3))
model.add(Activation('relu'))

#Pool Layer 1
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

#Conv Layer 2
model.add(Convolution2D(64, 3, 3, border_mode='same'))
model.add(Activation('relu'))

#Conv Layer 3
model.add(Convolution2D(64, 3, 3))
model.add(Activation('relu'))

#Pool Layer 2
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())

#Dense Layer 1
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dropout(0.5))

#Output layer
model.add(Dense(nb_classes))
model.add(Activation('softmax'))

adg = Adagrad(lr=0.01, epsilon=1e-06)

model.compile(loss='categorical_crossentropy',
```

```
              optimizer=adg,
              metrics=['accuracy'])

X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train /= 255
X_test /= 255

if not data_augmentation:
    print('Not using data augmentation.')
    model.fit(X_train, Y_train,
              batch_size=batch_size,
              nb_epoch=nb_epoch,
              validation_data=(X_test, Y_test),
              shuffle=True)
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
            total width)
        height_shift_range=0.1,  # randomly shift images vertically (fraction of
            total height)
        horizontal_flip=True,  # randomly flip images
        vertical_flip=False)  # randomly flip images

    # compute quantities required for featurewise 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),
                        samples_per_epoch=X_train.shape[0],
                        nb_epoch=nb_epoch,
                        validation_data=(X_test, Y_test),
                        callbacks=[TrainCallback((X_train, Y_train))])
```

Listing 8: Answer3(d)

```
'''Train a simple deep CNN on the CIFAR10 small images dataset.

GPU run command:
    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).

Note: the data was pickled with Python 2, and some encoding issues might prevent
    you
```

```
from loading it in Python 3. You might have to load it in Python 2,
save it in a different format, load it in Python 3 and repickle it.
'''

from __future__ import print_function
from keras.datasets import cifar10
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Activation, Flatten
from keras.layers.convolutional import Convolution2D, MaxPooling2D
from keras.optimizers import SGD
from keras.utils import np_utils
from keras.callbacks import Callback

batch_size = 32
nb_classes = 10
nb_epoch = 200
data_augmentation = True

# input image dimensions
img_rows, img_cols = 32, 32
# the CIFAR10 images are RGB
img_channels = 3

# the data, shuffled and split between train and test sets
(X_train, y_train), (X_test, y_test) = cifar10.load_data()

class TrainCallback(Callback):
    def __init__(self, test_data):
        self.test_data = test_data

    def on_epoch_end(self, epoch, logs={}):
        x, y = self.test_data
        loss, acc = self.model.evaluate(x, y, verbose=0)
        print('\nTrain loss: {}, Train acc: {}'.format(loss, acc))



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 = np_utils.to_categorical(y_train, nb_classes)
Y_test = np_utils.to_categorical(y_test, nb_classes)

model = Sequential()


#Input layer
model.add(Convolution2D(32, 3, 3, border_mode='same',
                        input_shape=(img_channels, img_rows, img_cols)))
model.add(Activation('relu'))

#Conv layer 1
model.add(Convolution2D(32, 3, 3))
model.add(Activation('relu'))

#Pool Layer 1
```

```
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

#Conv Layer 2
model.add(Convolution2D(64, 3, 3, border_mode='same'))
model.add(Activation('relu'))

#Conv Layer 3
model.add(Convolution2D(64, 3, 3))
model.add(Activation('relu'))

#Pool Layer 2
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())

#Dense Layer 1
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dropout(0.5))

#Output layer
model.add(Dense(nb_classes))
model.add(Activation('softmax'))

sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy',
              optimizer=sgd,
              metrics=['accuracy'])

X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train /= 255
X_test /= 255

if not data_augmentation:
    print('Not using data augmentation.')
    model.fit(X_train, Y_train,
              batch_size=batch_size,
              nb_epoch=nb_epoch,
              validation_data=(X_test, Y_test),
              shuffle=True)
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
            total width)
        height_shift_range=0.1,  # randomly shift images vertically (fraction of
            total height)
```

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

    # compute quantities required for featurewise 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),
                        samples_per_epoch=X_train.shape[0],
                        nb_epoch=nb_epoch,
                        validation_data=(X_test, Y_test),
                        callbacks=[TrainCallback((X_train, Y_train))])
```

Listing 9: Answer3(e)

```
'''Train a simple deep CNN on the CIFAR10 small images dataset.

GPU run command:
    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).

Note: the data was pickled with Python 2, and some encoding issues might prevent
    you
from loading it in Python 3. You might have to load it in Python 2,
save it in a different format, load it in Python 3 and repickle it.
'''

from __future__ import print_function
from keras.datasets import cifar10
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Activation, Flatten
from keras.layers.convolutional import Convolution2D, MaxPooling2D
from keras.optimizers import SGD
from keras.optimizers import Adagrad
from keras.optimizers import RMSprop
from keras.utils import np_utils
from keras.callbacks import Callback

batch_size = 32
nb_classes = 10
nb_epoch = 200
data_augmentation = True

# input image dimensions
img_rows, img_cols = 32, 32
# the CIFAR10 images are RGB
img_channels = 3

# the data, shuffled and split between train and test sets
(X_train, y_train), (X_test, y_test) = cifar10.load_data()

class TrainCallback(Callback):
    def __init__(self, test_data):
        self.test_data = test_data
```

```
    def on_epoch_end(self, epoch, logs={}):
        x, y = self.test_data
        loss, acc = self.model.evaluate(x, y, verbose=0)
        print('\nTrain loss: {}, Train acc: {}'.format(loss, acc))



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 = np_utils.to_categorical(y_train, nb_classes)
Y_test = np_utils.to_categorical(y_test, nb_classes)

model = Sequential()


#Input layer
model.add(Convolution2D(32, 3, 3, border_mode='same',
                        input_shape=(img_channels, img_rows, img_cols)))
model.add(Activation('relu'))

#Conv layer 1
model.add(Convolution2D(32, 3, 3))
model.add(Activation('relu'))

#Pool Layer 1
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

#Conv Layer 2
model.add(Convolution2D(64, 3, 3, border_mode='same'))
model.add(Activation('relu'))

#Conv Layer 3
model.add(Convolution2D(64, 3, 3))
model.add(Activation('relu'))

#Pool Layer 2
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())

#Dense Layer 1
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dropout(0.5))

#Output layer
model.add(Dense(nb_classes))
model.add(Activation('softmax'))

rms = RMSprop(lr=0.001, rho=0.9, epsilon=1e-06)

model.compile(loss='categorical_crossentropy',
              optimizer=rms,
              metrics=['accuracy'])
```

```
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train /= 255
X_test /= 255

if not data_augmentation:
    print('Not using data augmentation.')
    model.fit(X_train, Y_train,
              batch_size=batch_size,
              nb_epoch=nb_epoch,
              validation_data=(X_test, Y_test),
              shuffle=True)
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
            total width)
        height_shift_range=0.1,  # randomly shift images vertically (fraction of
            total height)
        horizontal_flip=True,  # randomly flip images
        vertical_flip=False)  # randomly flip images

    # compute quantities required for featurewise 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),
                        samples_per_epoch=X_train.shape[0],
                        nb_epoch=nb_epoch,
                        validation_data=(X_test, Y_test),
                        callbacks=[TrainCallback((X_train, Y_train))])
```

Listing 10: Answer3(f)