Berfin Kavşut
21602459
EEE 443
November 22, 2020

Report for Assignment #2

Question 1

Part A

Learning rate is chosen as 0.15. Mini-batch size is 100. Epoch number is chosen considerably high to observe convergence. At the end of each mini-batch, there is weight updates. Number of neurons in hidden layer is chosen as 10. Hyperbolic tangent activation function is used in hidden layer and output layer as given in the assignment.

Mean squared error (MSE) and mean classification error (MCE) is calculated at the end of each epoch separately. First neuron of output layer is for car and second neuron of output layer is for car classification. MSE and classification errors are shown in Fig. 1.

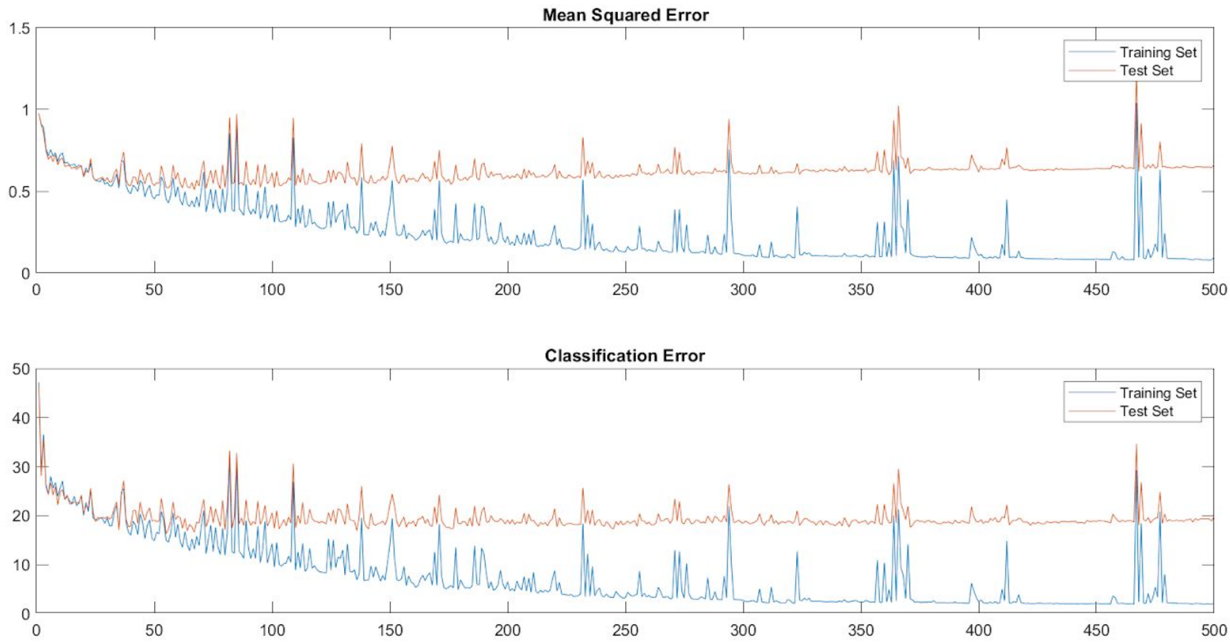


Fig. 1. Comparison of Mean Squared Error and Classification Error of Training Set and Test Set

Part B

The neural network is trained with the training set, therefore it was expected that the mean squared error and classification error was going to decrease as epoch number decreased. However, error metrics do not decrease for the test set after around 50 epochs. Mean squared error starts increasing after around 50 epochs. This shows the neural network started to overfit and could not generalize to the test set. Epoch number is too high and the neural network is forced for convergence.

Mean squared error and classification error follow the same trend in Fig. 1. Convergence of error metrics are in the same trend, but it is not possible to guess one form another numerically. We can only say mean squared error has convergence after a certain epoch number, which is around 50 epochs in our case, and classification error will converge to its best value possible as well around 50 epochs.

Part C

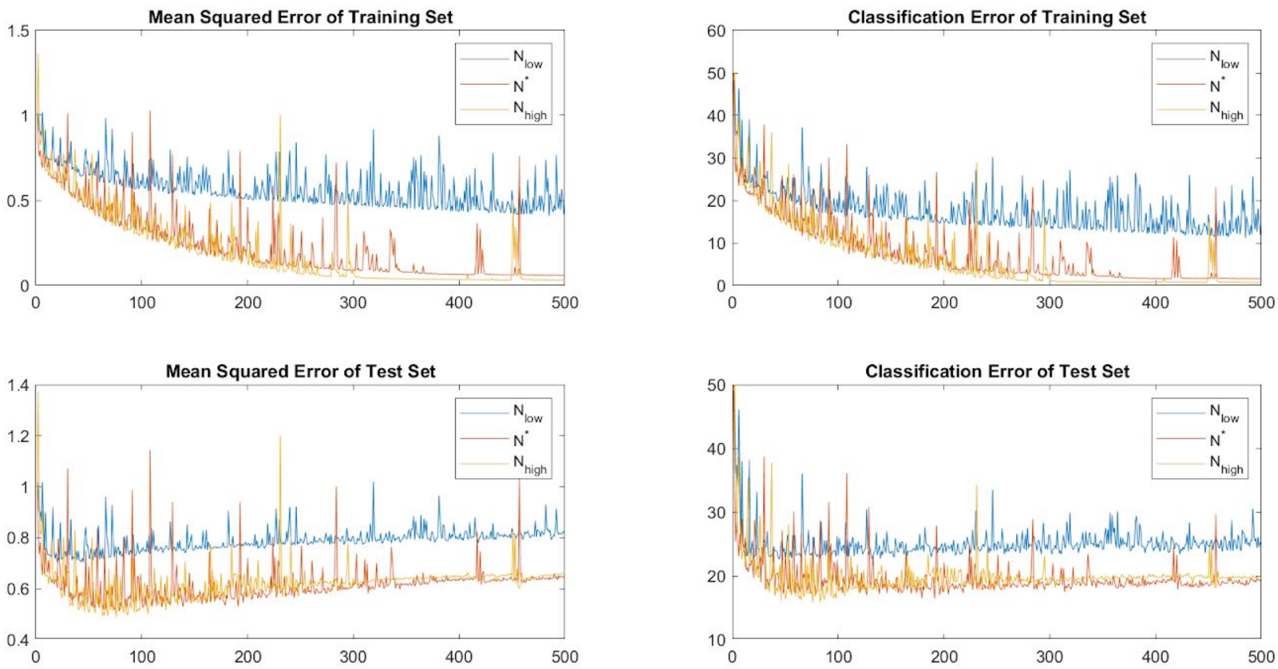


Fig. 2. MSE and Classification Error with Different Hidden Layer Neuron Numbers

The trends of error metrics for convergence are similar with N_{low}, N^* and N_{high}. N_{low} is 2, N^* is 10, and N_{high} is 50. Worst performance of MSE and classification error is with N_{low}. It converges faster at an unacceptably high value for error metrics. When the hidden neuron number is the highest, we observe the best performance for the training set but the optimal hidden neuron number gives better results for the test set. The reason is with more neurons, the neural network memorizes better for the training set and cannot generalize for the test set. Still, the difference between optimal and low case is not very large. Similar to Fig. 1, error metrics decrease until convergence for training set. For the test set, it decreases, and starts increasing until convergence.

Part D

In Part D, we add one more hidden layer and implement backpropagation similar to Part A. Learning rate is again selected as 1.15. Hidden neuron numbers are chosen to be 10 for each hidden layer.

Error plots are less noisy after convergence compared to Fig. 1. The neural network with more two hidden layer reaches to convergence for training set faster than the neural network with one hidden layer.

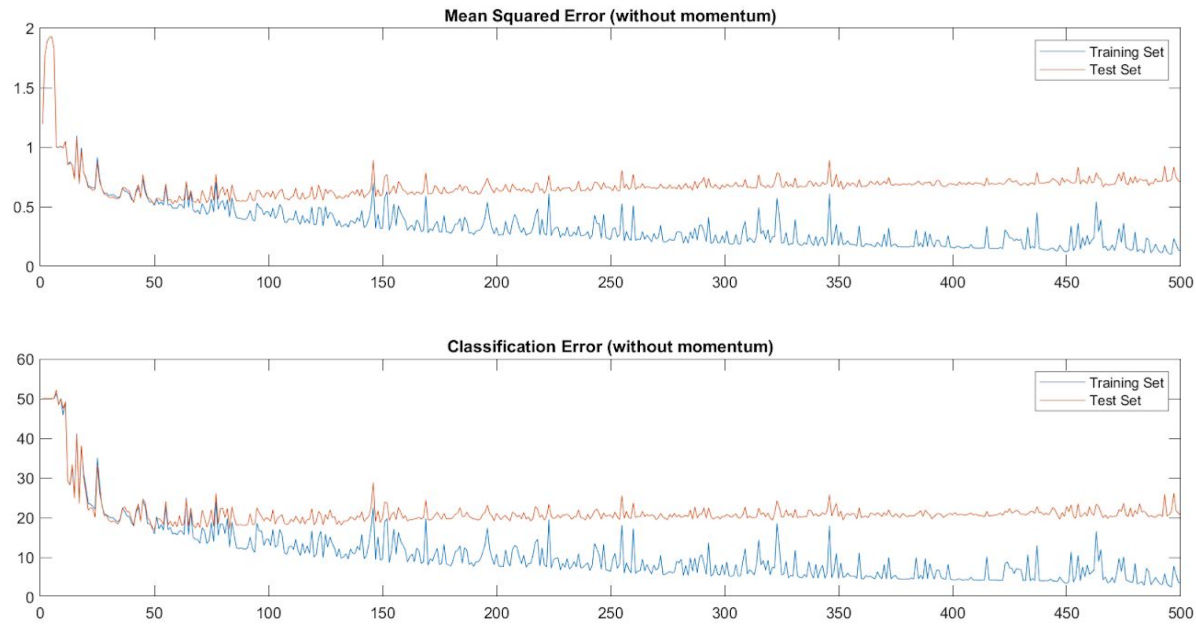


Fig. 3. Two Hidden Layer Neural Network without Momentum

Part E

Alpha is 0.5. The results in Fig. 4 are very similar to Fig. 3, with less noise. Effect of momentum terms is low pass filter in loss function, and Fig. 4. has less noisy error metrics plots compared to Fig. 3. Especially, it is seen when epoch number increases and neural network reaches to convergence.

Another effect of momentum is that the error plots decrease faster and reach to convergence faster compared to the case without momentum term.

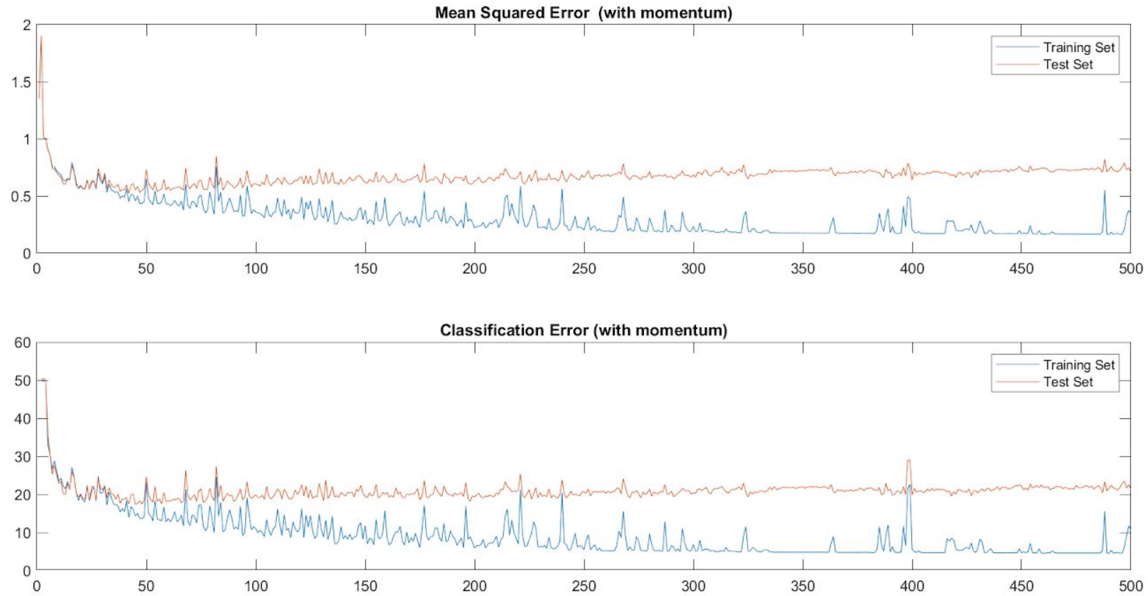


Fig. 4. Two Hidden Layer Neural Network with Momentum

Question 2

Part A

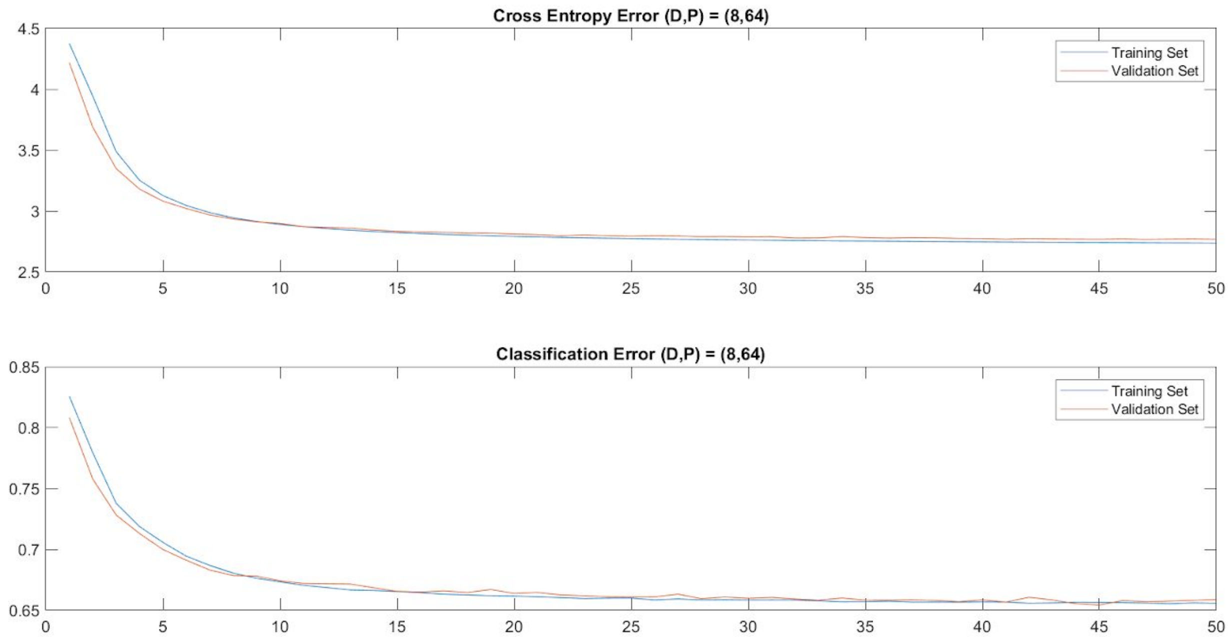


Fig. 5. Neural Network with (D,P) = (8,64)

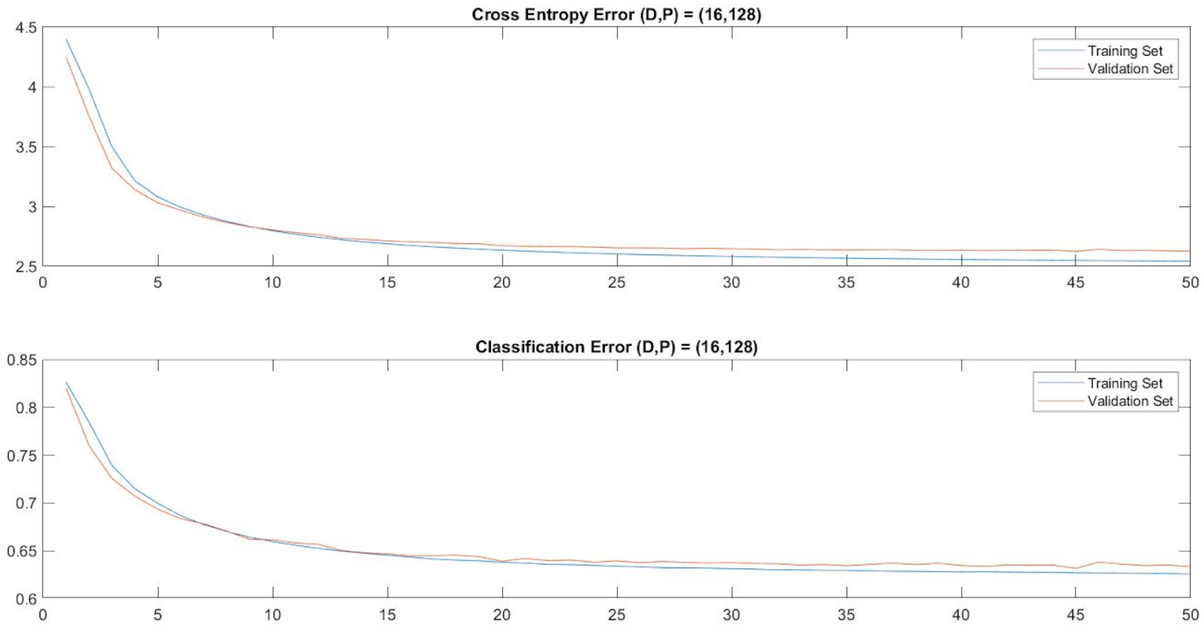


Fig. 6. Neural Network with (D,P) = (16,128)

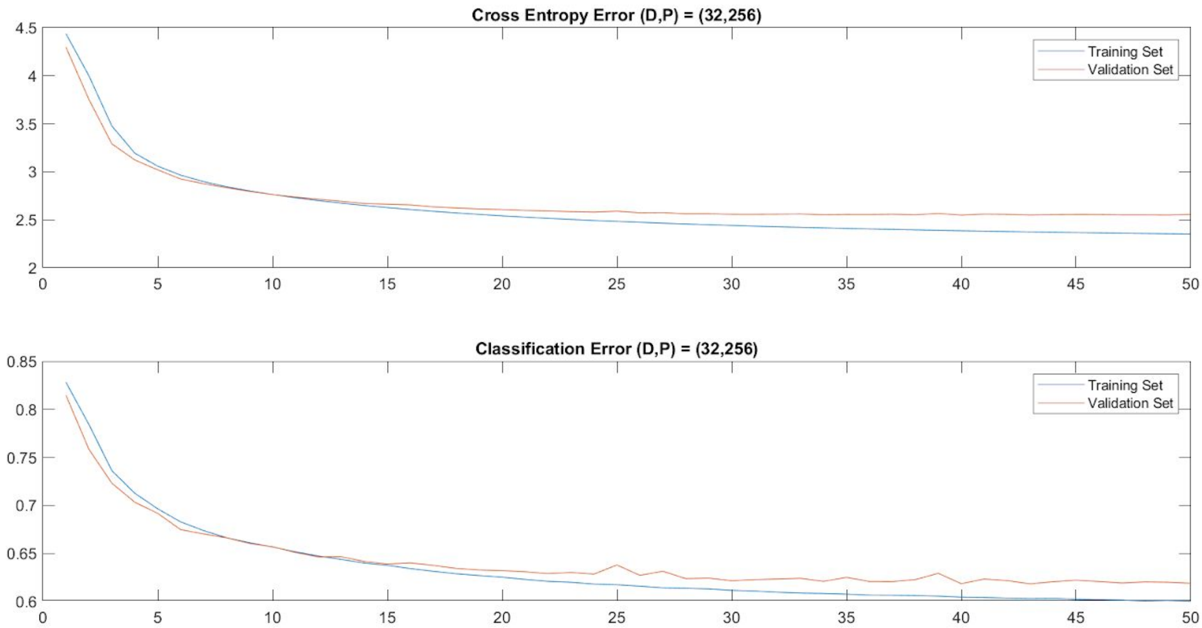


Fig. 7. Neural Network with (D,P) = (32,256)

Cross entropy error and classification error have similar trends. Around 15 epochs, decreasing of error metrics for validation set stops and continues for training set. We can state generalization stops around 15 epochs because error metrics do not improve for validation set.

In Fig. 5-7, cross entropy error and classification error in scale of (0,1) are displayed in 50 epoch numbers. Parameters are selected as asked in the assignment. According to these figures, cross entropy stopping criteria is chosen to be less than 3 since the error plots start convergence around 3 for (D,P) = (8,16), (16,128) and (32,256). Smaller values of cross entropy could be selected, however, it was going to take too much time for grading which is around 10 minutes. The chosen value 3 is good enough for the next step and less time consuming.

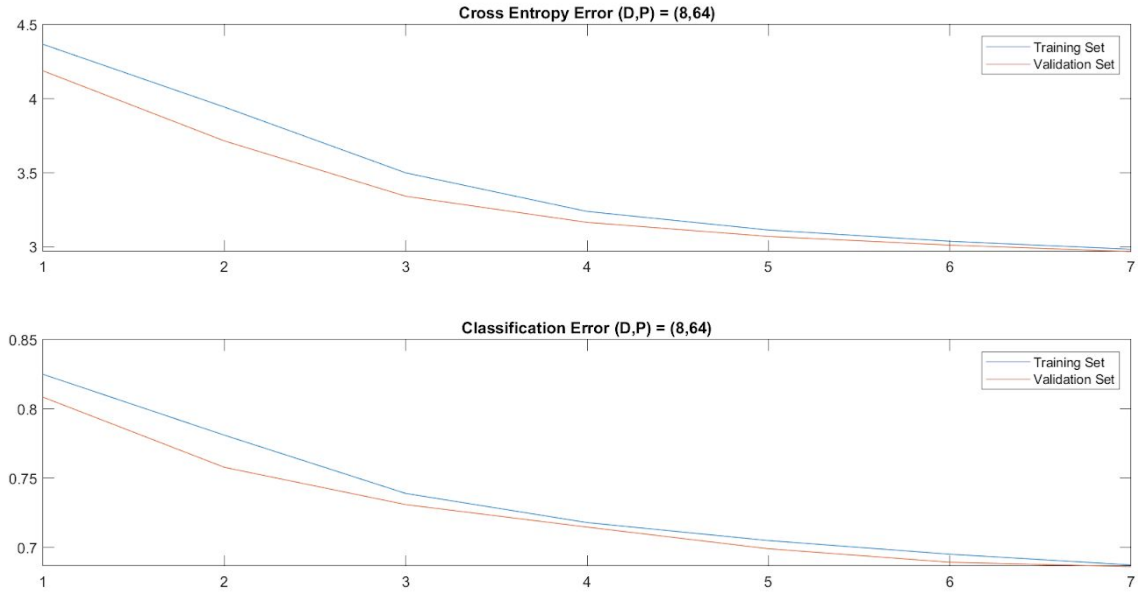


Fig. 8. Neural Network of (D,P) = (8,64) with Stopping Criteria

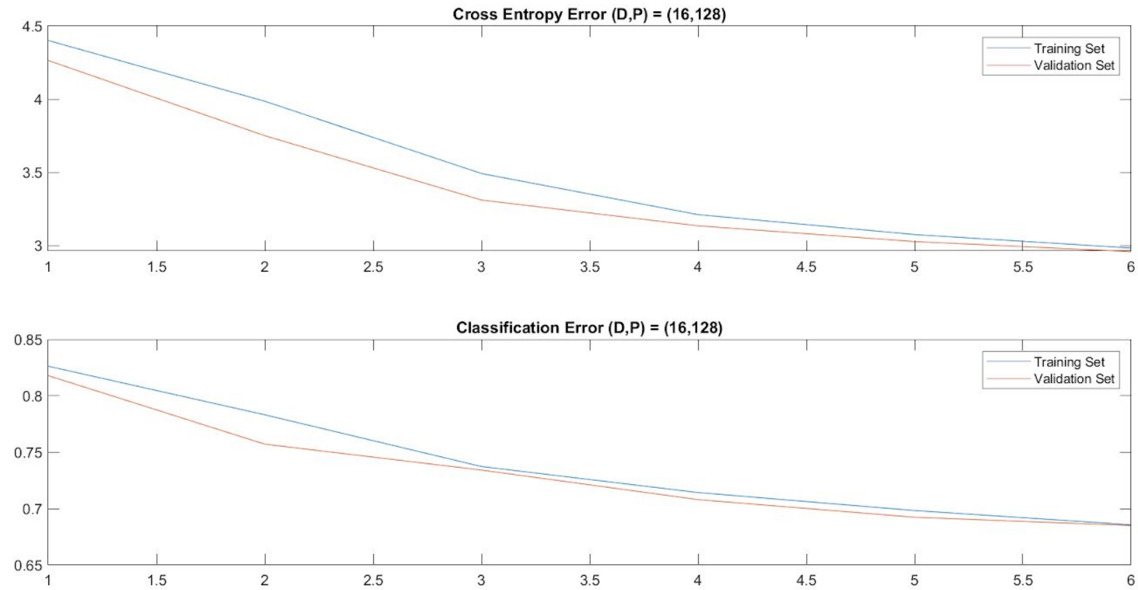


Fig. 9. Neural Network of (D,P) = (16,128) with Stopping Criteria

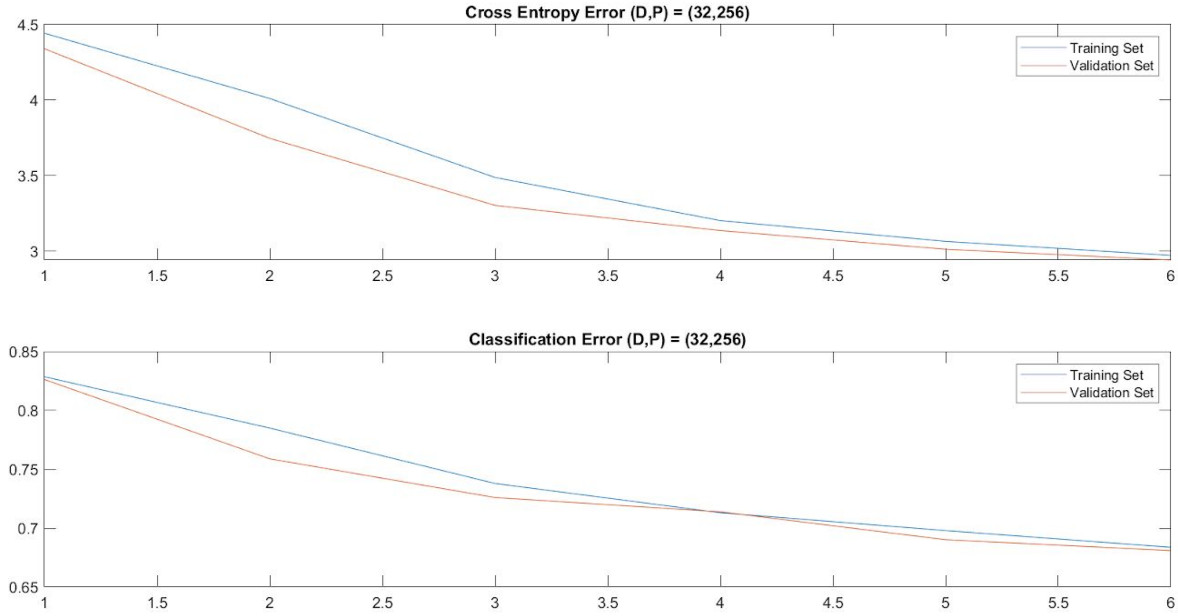


Fig. 10. Neural Network of (D,P) = (32,256) with Stopping Criteria

To have comments in Fig 5-7 is more logical rather than Fig 8-10 since we can see the whole process until convergence. My opinion is that (D,P) = (32,256) gave the best results due to having smaller cross entropy error and classification error at the end of convergence. However, there is a very slight difference between (D,P) = (16,128) and (D,P) = (32,256) and it is hard to compare. When epoch number is around 10, their performance seems to be almost the same and Fig. 9 and Fig. 10 with 6 epochs are very close to each other.

Part B

It is continued with the neural network having (D,P) = (16,128). Results for 5 random trigrams are as follows.

Trigram #1:
Desired Output: 144
Top 10 Candidates: [89, 26, 193, 23, 144, 158, 106, 151, 207, 167]

Trigram #2:
Desired Output: 23
Top 10 Candidates: [213, 149, 71, 117, 247, 196, 6, 45, 216, 206]

Trigram #3:
Desired Output: 55
Top 10 Candidates: [34, 120, 126, 149, 135, 38, 69, 224, 247, 70]

Trigram #4:
Desired Output: 75
Top 10 Candidates: [144, 89, 23, 26, 76, 188, 201, 46, 199, 228]

Trigram #5:
Desired Output: 144
Top 10 Candidates: [42, 200, 190, 116, 36, 32, 66, 50, 144, 73]

Classification error of the validation set was 0.68, therefore good classification performance was not expected. Still, the desired outputs or close words (with close index numbers) can be in top 10 candidates. In triagram #1, 144 is the 5th candidate. In triagram #4, 76 which is close to 75 is the 5th candidate. In riagram #5, 144 is the 9th candidate.

When I experimented with different trigrams, I observed that triagrams with the desired output 144 is a frequent output and mostly gives better results compared to other trigrams.

Results are not good results, but it is the best that can be reached with the available neural network because training accuracy and validation accuracy was already low.

Question 3

Inline questions are answered where they are asked in the code.

Fully-Connected Neural Nets

In the previous homework you implemented a fully-connected two-layer neural network on CIFAR-10. The implementation was simple but not very modular since the loss and gradient were computed in a single monolithic function. This is manageable for a simple two-layer network, but would become impractical as we move to bigger models. Ideally we want to build networks using a more modular design so that we can implement different layer types in isolation and then snap them together into models with different architectures.

In this exercise we will implement fully-connected networks using a more modular approach. For each layer we will implement a `forward` and a `backward` function. The `forward` function will receive inputs, weights, and other parameters and will return both an output and a `cache` object storing data needed for the backward pass, like this:

```
def layer_forward(x, w):
  """ Receive inputs x and weights w """
  # Do some computations ...
  z = # ... some intermediate value
  # Do some more computations ...
  out = # the output

  cache = (x, w, z, out) # Values we need to compute gradients

  return out, cache
```

The backward pass will receive upstream derivatives and the `cache` object, and will return gradients with respect to the inputs and weights, like this:

```
def layer_backward(dout, cache):
  """
  Receive dout (derivative of loss with respect to outputs) and cache,
  and compute derivative with respect to inputs.
  """
  # Unpack cache values
  x, w, z, out = cache

  # Use values in cache to compute derivatives
  dx = # Derivative of loss with respect to x
  dw = # Derivative of loss with respect to w

  return dx, dw
```

After implementing a bunch of layers this way, we will be able to easily combine them to build classifiers with different architectures.

In addition to implementing fully-connected networks of arbitrary depth, we will also explore different update rules for optimization, and introduce Dropout as a regularizer and Batch/Layer Normalization as a tool to more efficiently optimize deep networks.

In [2]:

```
# As usual, a bit of setup
from __future__ import print_function
import time
import numpy as np
import matplotlib.pyplot as plt
from cs231n.classifiers.fc_net import *
from cs231n.data_utils import get_CIFAR10_data
from cs231n.gradient_check import eval_numerical_gradient, eval_numerical_gradient_array
from cs231n.solver import Solver
```

```
%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2

def rel_error(x, y):
  """ returns relative error """
  return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))
```

```
run the following from the cs231n directory and try again:
python setup.py build_ext --inplace
You may also need to restart your iPython kernel
```

In [3]:

```
# Load the (preprocessed) CIFAR10 data.

data = get_CIFAR10_data()
for k, v in list(data.items()):
  print(('%s: ' % k, v.shape))
```

```
('X_train: ', (49000, 3, 32, 32))
('y_train: ', (49000,))
('X_val: ', (1000, 3, 32, 32))
('y_val: ', (1000,))
('X_test: ', (1000, 3, 32, 32))
('y_test: ', (1000,))
```

Affine layer: foward

Open the file `cs231n/layers.py` and implement the `affine_forward` function.

Once you are done you can test your implementaion by running the following:

In [4]:

```
# Test the affine_forward function

num_inputs = 2
input_shape = (4, 5, 6)
output_dim = 3

input_size = num_inputs * np.prod(input_shape)
weight_size = output_dim * np.prod(input_shape)

x = np.linspace(-0.1, 0.5, num=input_size).reshape(num_inputs, *input_shape)
w = np.linspace(-0.2, 0.3, num=weight_size).reshape(np.prod(input_shape), output_dim)
b = np.linspace(-0.3, 0.1, num=output_dim)

out, _ = affine_forward(x, w, b)
correct_out = np.array([[ 1.49834967,  1.70660132,  1.91485297],
                        [ 3.25553199,  3.5141327,   3.77273342]])

# Compare your output with ours. The error should be around e-9 or less.
print('Testing affine_forward function:')
print('difference: ', rel_error(out, correct_out))
```

```
Testing affine_forward function:
difference:  9.769849468192957e-10
```

Affine layer: backward

Now implement the `affine_backward` function and test your implementation using numeric gradient

checking.

In [5]:

```
# Test the affine_backward function
np.random.seed(231)
x = np.random.randn(10, 2, 3)
w = np.random.randn(6, 5)
b = np.random.randn(5)
dout = np.random.randn(10, 5)

dx_num = eval_numerical_gradient_array(lambda x: affine_forward(x, w, b)[0], x, dout)
dw_num = eval_numerical_gradient_array(lambda w: affine_forward(x, w, b)[0], w, dout)
db_num = eval_numerical_gradient_array(lambda b: affine_forward(x, w, b)[0], b, dout)

_, cache = affine_forward(x, w, b)
dx, dw, db = affine_backward(dout, cache)

# The error should be around e-10 or less
print('Testing affine_backward function:')
print('dx error: ', rel_error(dx_num, dx))
print('dw error: ', rel_error(dw_num, dw))
print('db error: ', rel_error(db_num, db))
```

```
Testing affine_backward function:
dx error:  5.399100368651805e-11
dw error:  9.904211865398145e-11
db error:  2.4122867568119087e-11
```

ReLU activation: forward

Implement the forward pass for the ReLU activation function in the `relu_forward` function and test your implementation using the following:

In [6]:

```
# Test the relu_forward function

x = np.linspace(-0.5, 0.5, num=12).reshape(3, 4)

out, _ = relu_forward(x)
correct_out = np.array([[ 0.,          0.,          0.,          0.,         ],
                        [ 0.,          0.,          0.04545455,  0.13636364,],
                        [ 0.22727273,  0.31818182,  0.40909091,  0.5,        ]])

# Compare your output with ours. The error should be on the order of e-8
print('Testing relu_forward function:')
print('difference: ', rel_error(out, correct_out))
```

```
Testing relu_forward function:
difference:  4.999999798022158e-08
```

ReLU activation: backward

Now implement the backward pass for the ReLU activation function in the `relu_backward` function and test your implementation using numeric gradient checking:

In [7]:

```
np.random.seed(231)
x = np.random.randn(10, 10)
dout = np.random.randn(*x.shape)

dx_num = eval_numerical_gradient_array(lambda x: relu_forward(x)[0], x, dout)

_, cache = relu_forward(x)
dx = relu_backward(dout, cache)
```

```
# The error should be on the order of e-12
print('Testing relu_backward function:')
print('dx error: ', rel_error(dx_num, dx))
```

```
Testing relu_backward function:
dx error:  3.2756349136310288e-12
```

Inline Question 1:

We've only asked you to implement ReLU, but there are a number of different activation functions that one could use in neural networks, each with its pros and cons. In particular, an issue commonly seen with activation functions is getting zero (or close to zero) gradient flow during backpropagation. Which of the following activation functions have this problem? If you consider these functions in the one dimensional case, what types of input would lead to this behaviour?

1. **Sigmoid**
2. **ReLU**
3. **Leaky ReLU**

Answer:

- **ReLU has output 0 for inputs smaller than 0 and output 1 for inputs larger than 0. When input is smaller than 0, we have the zero problem.**

Leaky ReLU is a remedy for this problem of ReLU. Leaky ReLu gives a very small constant output for inputs smaller than 0, but not 0. Sigmoids also give very small values, but not 0, when inputs are far away from 0.

"Sandwich" layers

There are some common patterns of layers that are frequently used in neural nets. For example, affine layers are frequently followed by a ReLU nonlinearity. To make these common patterns easy, we define several convenience layers in the file `cs231n/layer_utils.py`.

For now take a look at the `affine_relu_forward` and `affine_relu_backward` functions, and run the following to numerically gradient check the backward pass:

In [8]:

```
from cs231n.layer_utils import affine_relu_forward, affine_relu_backward
np.random.seed(231)
x = np.random.randn(2, 3, 4)
w = np.random.randn(12, 10)
b = np.random.randn(10)
dout = np.random.randn(2, 10)

out, cache = affine_relu_forward(x, w, b)
dx, dw, db = affine_relu_backward(dout, cache)

dx_num = eval_numerical_gradient_array(lambda x: affine_relu_forward(x, w, b)[0], x, dou
t)
dw_num = eval_numerical_gradient_array(lambda w: affine_relu_forward(x, w, b)[0], w, dou
t)
db_num = eval_numerical_gradient_array(lambda b: affine_relu_forward(x, w, b)[0], b, dou
t)

# Relative error should be around e-10 or less
print('Testing affine_relu_forward and affine_relu_backward:')
print('dx error: ', rel_error(dx_num, dx))
print('dw error: ', rel_error(dw_num, dw))
print('db error: ', rel_error(db_num, db))
```

```
Testing affine_relu_forward and affine_relu_backward:
dx error:  2.299579177309368e-11
```

```
dw error:  8.162011105764925e-11
db error:  7.826724021458994e-12
```

Loss layers: Softmax and SVM

You implemented these loss functions in the last assignment, so we'll give them to you for free here. You should still make sure you understand how they work by looking at the implementations in `cs231n/layers.py`.

You can make sure that the implementations are correct by running the following:

In [9]:

```
np.random.seed(231)
num_classes, num_inputs = 10, 50
x = 0.001 * np.random.randn(num_inputs, num_classes)
y = np.random.randint(num_classes, size=num_inputs)

dx_num = eval_numerical_gradient(lambda x: svm_loss(x, y)[0], x, verbose=False)
loss, dx = svm_loss(x, y)

# Test svm_loss function. Loss should be around 9 and dx error should be around the order
of e-9
print('Testing svm_loss:')
print('loss: ', loss)
print('dx error: ', rel_error(dx_num, dx))

dx_num = eval_numerical_gradient(lambda x: softmax_loss(x, y)[0], x, verbose=False)
loss, dx = softmax_loss(x, y)

# Test softmax_loss function. Loss should be close to 2.3 and dx error should be around e
-8
print('\nTesting softmax_loss:')
print('loss: ', loss)
print('dx error: ', rel_error(dx_num, dx))
```

```
Testing svm_loss:
loss:  8.999602749096233
dx error:  1.4021566006651672e-09

Testing softmax_loss:
loss:  2.302545844500738
dx error:  9.384673161989355e-09
```

Two-layer network

In the previous assignment you implemented a two-layer neural network in a single monolithic class. Now that you have implemented modular versions of the necessary layers, you will reimplement the two layer network using these modular implementations.

Open the file `cs231n/classifiers/fc_net.py` and complete the implementation of the `TwoLayerNet` class. This class will serve as a model for the other networks you will implement in this assignment, so read through it to make sure you understand the API. You can run the cell below to test your implementation.

In [10]:

```
np.random.seed(231)
N, D, H, C = 3, 5, 50, 7
X = np.random.randn(N, D)
y = np.random.randint(C, size=N)

std = 1e-3
model = TwoLayerNet(input_dim=D, hidden_dim=H, num_classes=C, weight_scale=std)

print('Testing initialization ... ')
W1_std = abs(model.params['W1'].std() - std)
b1 = model.params['b1']
W2_std = abs(model.params['W2'].std() - std)
```

```
b2 = model.params['b2']
assert W1_std < std / 10, 'First layer weights do not seem right'
assert np.all(b1 == 0), 'First layer biases do not seem right'
assert W2_std < std / 10, 'Second layer weights do not seem right'
assert np.all(b2 == 0), 'Second layer biases do not seem right'

print('Testing test-time forward pass ... ')
model.params['W1'] = np.linspace(-0.7, 0.3, num=D*H).reshape(D, H)
model.params['b1'] = np.linspace(-0.1, 0.9, num=H)
model.params['W2'] = np.linspace(-0.3, 0.4, num=H*C).reshape(H, C)
model.params['b2'] = np.linspace(-0.9, 0.1, num=C)
X = np.linspace(-5.5, 4.5, num=N*D).reshape(D, N).T
scores = model.loss(X)
correct_scores = np.asarray(
  [[11.53165108,  12.2917344,   13.05181771,  13.81190102,  14.57198434, 15.33206765,  1
6.09215096],
   [12.05769098,  12.74614105,  13.43459113,  14.1230412,   14.81149128, 15.49994135,  1
6.18839143],
   [12.58373087,  13.20054771,  13.81736455,  14.43418138,  15.05099822, 15.66781506,  1
6.2846319 ]])
scores_diff = np.abs(scores - correct_scores).sum()
assert scores_diff < 1e-6, 'Problem with test-time forward pass'

print('Testing training loss (no regularization)')
y = np.asarray([0, 5, 1])
loss, grads = model.loss(X, y)
correct_loss = 3.4702243556
assert abs(loss - correct_loss) < 1e-10, 'Problem with training-time loss'

model.reg = 1.0
loss, grads = model.loss(X, y)
correct_loss = 26.5948426952
assert abs(loss - correct_loss) < 1e-10, 'Problem with regularization loss'

# Errors should be around e-7 or less
for reg in [0.0, 0.7]:
  print('Running numeric gradient check with reg = ', reg)
  model.reg = reg
  loss, grads = model.loss(X, y)

  for name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    grad_num = eval_numerical_gradient(f, model.params[name], verbose=False)
    print('%s relative error: %.2e' % (name, rel_error(grad_num, grads[name])))
```

```
Testing initialization ...
Testing test-time forward pass ...
Testing training loss (no regularization)
Running numeric gradient check with reg =  0.0
W1 relative error: 1.83e-08
W2 relative error: 3.12e-10
b1 relative error: 9.83e-09
b2 relative error: 4.33e-10
Running numeric gradient check with reg =  0.7
W1 relative error: 2.53e-07
W2 relative error: 2.85e-08
b1 relative error: 1.56e-08
b2 relative error: 7.76e-10
```

Solver

In the previous assignment, the logic for training models was coupled to the models themselves. Following a more modular design, for this assignment we have split the logic for training models into a separate class.

Open the file `cs231n/solver.py` and read through it to familiarize yourself with the API. After doing so, use a `Solver` instance to train a `TwoLayerNet` that achieves at least `50%` accuracy on the validation set.

In [11]:

```
model = TwoLayerNet()
```

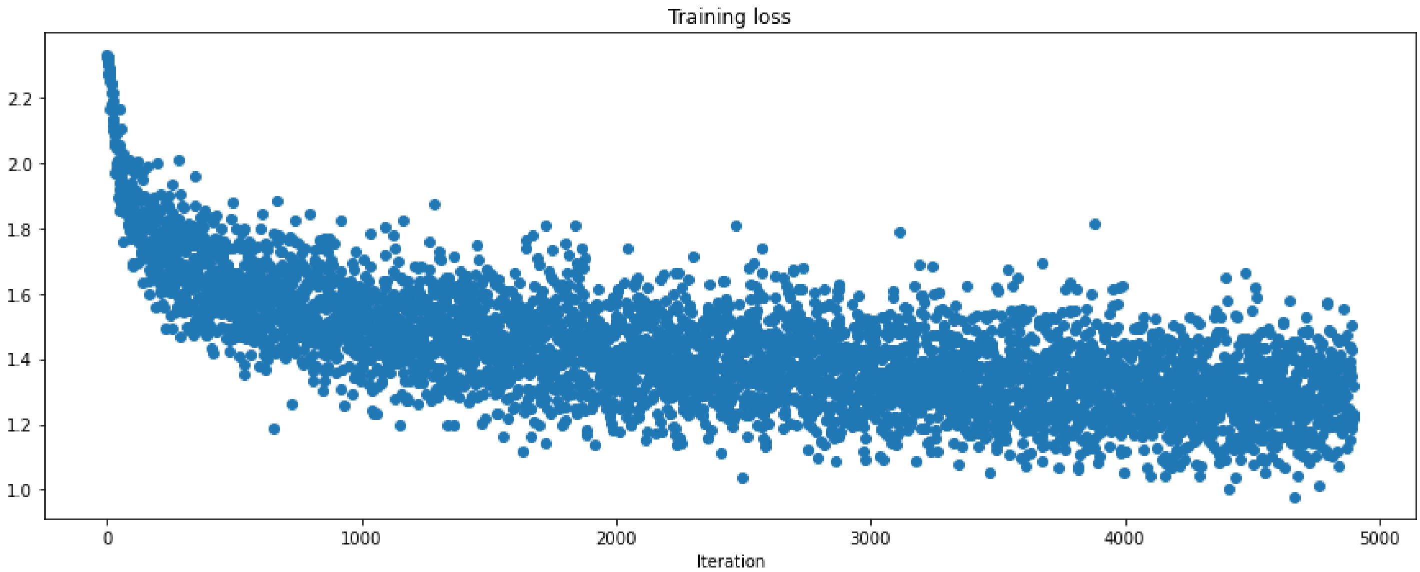
```python
model = TwoLayerNet()
solver = None

##############################################################################
# TODO: Use a Solver instance to train a TwoLayerNet that achieves at least  #
# 50% accuracy on the validation set.                                        #
##############################################################################

model = TwoLayerNet(hidden_dim=100, reg=0.2)
solver = Solver(model, data, update_rule='sgd',
                optim_config={'learning_rate': 1e-3}, lr_decay=0.95,
                num_epochs=10, batch_size=100, print_every=100)
solver.train()

##############################################################################
#                             END OF YOUR CODE                               #
##############################################################################
```

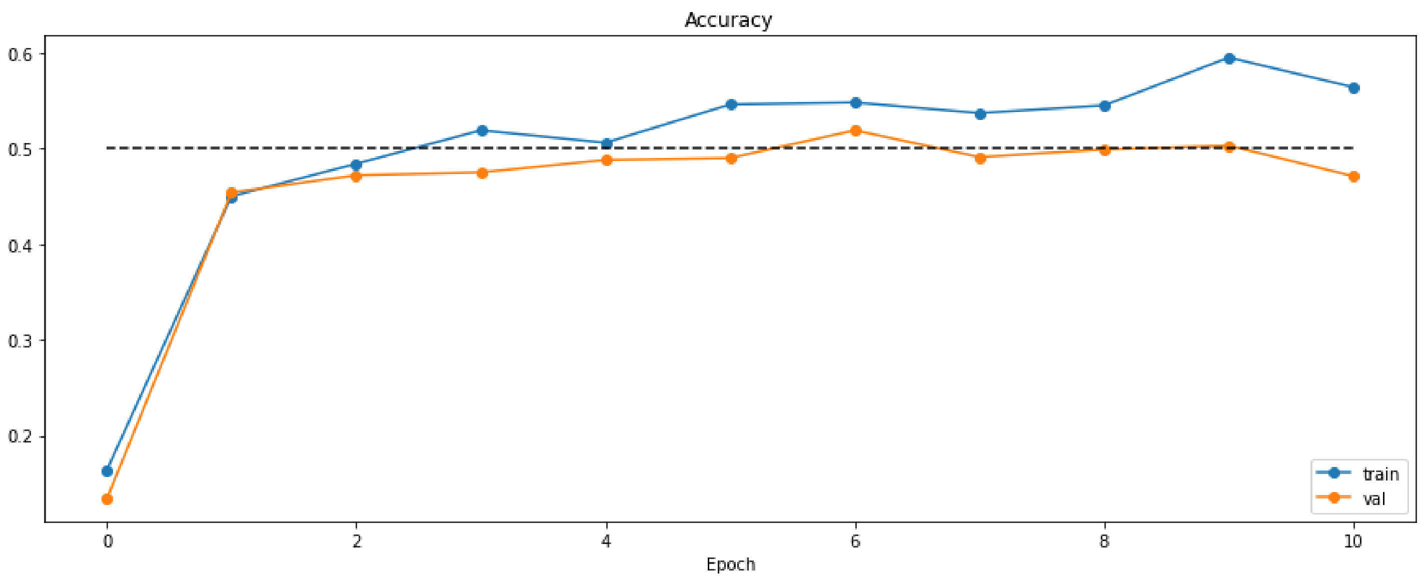
```
(Iteration 1 / 4900) loss: 2.332096
(Epoch 0 / 10) train acc: 0.164000; val_acc: 0.134000
(Iteration 101 / 4900) loss: 1.857220
(Iteration 201 / 4900) loss: 2.000576
(Iteration 301 / 4900) loss: 1.651815
(Iteration 401 / 4900) loss: 1.538214
(Epoch 1 / 10) train acc: 0.450000; val_acc: 0.454000
(Iteration 501 / 4900) loss: 1.608869
(Iteration 601 / 4900) loss: 1.501398
(Iteration 701 / 4900) loss: 1.615213
(Iteration 801 / 4900) loss: 1.656747
(Iteration 901 / 4900) loss: 1.468052
(Epoch 2 / 10) train acc: 0.484000; val_acc: 0.472000
(Iteration 1001 / 4900) loss: 1.505273
(Iteration 1101 / 4900) loss: 1.503323
(Iteration 1201 / 4900) loss: 1.418404
(Iteration 1301 / 4900) loss: 1.356568
(Iteration 1401 / 4900) loss: 1.507079
(Epoch 3 / 10) train acc: 0.519000; val_acc: 0.475000
(Iteration 1501 / 4900) loss: 1.405298
(Iteration 1601 / 4900) loss: 1.425098
(Iteration 1701 / 4900) loss: 1.388389
(Iteration 1801 / 4900) loss: 1.559448
(Iteration 1901 / 4900) loss: 1.469148
(Epoch 4 / 10) train acc: 0.506000; val_acc: 0.488000
(Iteration 2001 / 4900) loss: 1.521458
(Iteration 2101 / 4900) loss: 1.452836
(Iteration 2201 / 4900) loss: 1.515952
(Iteration 2301 / 4900) loss: 1.253438
(Iteration 2401 / 4900) loss: 1.329813
(Epoch 5 / 10) train acc: 0.546000; val_acc: 0.490000
(Iteration 2501 / 4900) loss: 1.385455
(Iteration 2601 / 4900) loss: 1.380330
(Iteration 2701 / 4900) loss: 1.344157
(Iteration 2801 / 4900) loss: 1.516297
(Iteration 2901 / 4900) loss: 1.373451
(Epoch 6 / 10) train acc: 0.548000; val_acc: 0.519000
(Iteration 3001 / 4900) loss: 1.313017
(Iteration 3101 / 4900) loss: 1.139112
(Iteration 3201 / 4900) loss: 1.596601
(Iteration 3301 / 4900) loss: 1.372248
(Iteration 3401 / 4900) loss: 1.524008
(Epoch 7 / 10) train acc: 0.537000; val_acc: 0.491000
(Iteration 3501 / 4900) loss: 1.325397
(Iteration 3601 / 4900) loss: 1.141724
(Iteration 3701 / 4900) loss: 1.368370
(Iteration 3801 / 4900) loss: 1.319290
(Iteration 3901 / 4900) loss: 1.101957
(Epoch 8 / 10) train acc: 0.545000; val_acc: 0.499000
(Iteration 4001 / 4900) loss: 1.239187
(Iteration 4101 / 4900) loss: 1.346376
(Iteration 4201 / 4900) loss: 1.154919
(Iteration 4301 / 4900) loss: 1.073516
(Iteration 4401 / 4900) loss: 1.577285
(Epoch 9 / 10) train acc: 0.595000; val_acc: 0.503000
```

```
(Iteration 4501 / 4900) loss: 1.253220
(Iteration 4601 / 4900) loss: 1.465048
(Iteration 4701 / 4900) loss: 1.484373
(Iteration 4801 / 4900) loss: 1.242994
(Epoch 10 / 10) train acc: 0.564000; val_acc: 0.471000
```

In [12]:

```python
# Run this cell to visualize training loss and train / val accuracy

plt.subplot(2, 1, 1)
plt.title('Training loss')
plt.plot(solver.loss_history, 'o')
plt.xlabel('Iteration')

plt.subplot(2, 1, 2)
plt.title('Accuracy')
plt.plot(solver.train_acc_history, '-o', label='train')
plt.plot(solver.val_acc_history, '-o', label='val')
plt.plot([0.5] * len(solver.val_acc_history), 'k--')
plt.xlabel('Epoch')
plt.legend(loc='lower right')
plt.gcf().set_size_inches(15, 12)
plt.show()
```





Multilayer network

Next you will implement a fully-connected network with an arbitrary number of hidden layers.

Read through the `FullyConnectedNet` **class in the file** `cs231n/classifiers/fc_net.py`**.**

Implement the initialization, the forward pass, and the backward pass. For the moment don't worry about implementing dropout or batch/layer normalization; we will add those features soon.

Initial loss and gradient check

As a sanity check, run the following to check the initial loss and to gradient check the network both with and without regularization. Do the initial losses seem reasonable?

For gradient checking, you should expect to see errors around 1e-7 or less.

In [13]:

```
np.random.seed(231)
N, D, H1, H2, C = 2, 15, 20, 30, 10
X = np.random.randn(N, D)
y = np.random.randint(C, size=(N,))

for reg in [0, 3.14]:
  print('Running check with reg = ', reg)
  model = FullyConnectedNet([H1, H2], input_dim=D, num_classes=C,
                            reg=reg, weight_scale=5e-2, dtype=np.float64)

  loss, grads = model.loss(X, y)
  print('Initial loss: ', loss)

  # Most of the errors should be on the order of e-7 or smaller.
  # NOTE: It is fine however to see an error for W2 on the order of e-5
  # for the check when reg = 0.0
  for name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    grad_num = eval_numerical_gradient(f, model.params[name], verbose=False, h=1e-5)
    print('%s relative error: %.2e' % (name, rel_error(grad_num, grads[name])))
```

```
Running check with reg =  0
Initial loss:  2.3004790897684924
W1 relative error: 1.48e-07
W2 relative error: 2.21e-05
W3 relative error: 3.53e-07
b1 relative error: 5.38e-09
b2 relative error: 2.09e-09
b3 relative error: 5.80e-11
Running check with reg =  3.14
Initial loss:  7.052114776533016
W1 relative error: 7.36e-09
W2 relative error: 6.87e-08
W3 relative error: 3.48e-08
b1 relative error: 1.48e-08
b2 relative error: 1.72e-09
b3 relative error: 1.80e-10
```

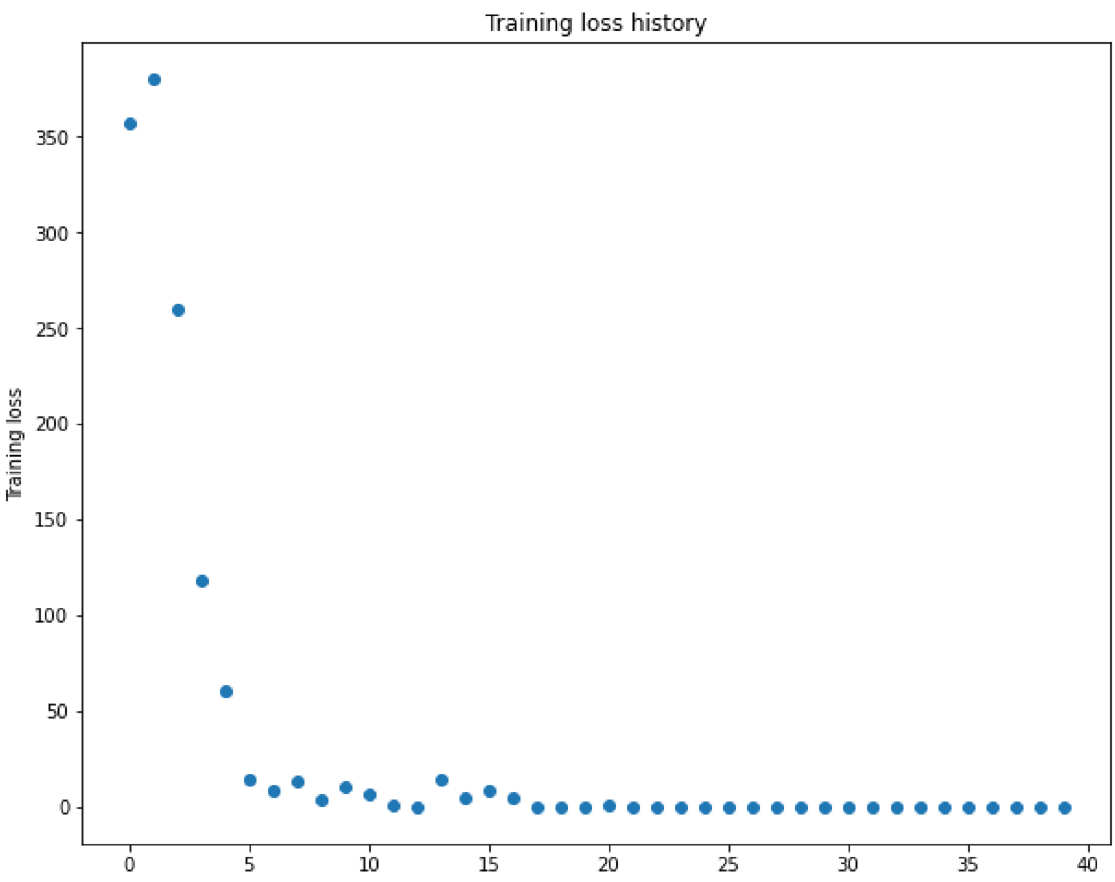
As another sanity check, make sure you can overfit a small dataset of 50 images. First we will try a three-layer network with 100 units in each hidden layer. In the following cell, tweak the learning rate and initialization scale to overfit and achieve 100% training accuracy within 20 epochs.

In [14]:

```
# TODO: Use a three-layer Net to overfit 50 training examples by
# tweaking just the learning rate and initialization scale.

num_train = 50
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

weight_scale = 1e-1
```

```
learning_rate = 1e-3
model = FullyConnectedNet([100, 100],
              weight_scale=weight_scale, dtype=np.float64)
solver = Solver(model, small_data,
                print_every=10, num_epochs=20, batch_size=25,
                update_rule='sgd',
                optim_config={
                  'learning_rate': learning_rate,
                }
         )
solver.train()

plt.plot(solver.loss_history, 'o')
plt.title('Training loss history')
plt.xlabel('Iteration')
plt.ylabel('Training loss')
plt.show()
```

```
(Iteration 1 / 40) loss: 357.428290
(Epoch 0 / 20) train acc: 0.220000; val_acc: 0.111000
(Epoch 1 / 20) train acc: 0.380000; val_acc: 0.141000
(Epoch 2 / 20) train acc: 0.520000; val_acc: 0.138000
(Epoch 3 / 20) train acc: 0.740000; val_acc: 0.130000
(Epoch 4 / 20) train acc: 0.820000; val_acc: 0.153000
(Epoch 5 / 20) train acc: 0.860000; val_acc: 0.175000
(Iteration 11 / 40) loss: 6.726589
(Epoch 6 / 20) train acc: 0.940000; val_acc: 0.163000
(Epoch 7 / 20) train acc: 0.960000; val_acc: 0.166000
(Epoch 8 / 20) train acc: 0.960000; val_acc: 0.164000
(Epoch 9 / 20) train acc: 0.980000; val_acc: 0.162000
(Epoch 10 / 20) train acc: 0.980000; val_acc: 0.162000
(Iteration 21 / 40) loss: 0.800243
(Epoch 11 / 20) train acc: 1.000000; val_acc: 0.158000
(Epoch 12 / 20) train acc: 1.000000; val_acc: 0.158000
(Epoch 13 / 20) train acc: 1.000000; val_acc: 0.158000
(Epoch 14 / 20) train acc: 1.000000; val_acc: 0.158000
(Epoch 15 / 20) train acc: 1.000000; val_acc: 0.158000
(Iteration 31 / 40) loss: 0.000000
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.158000
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.158000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.158000
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.158000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.158000
```



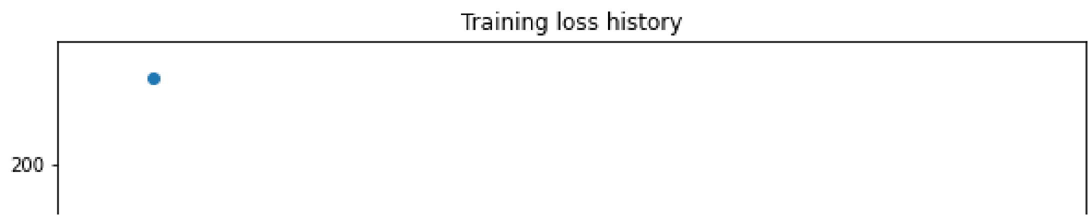
Iteration

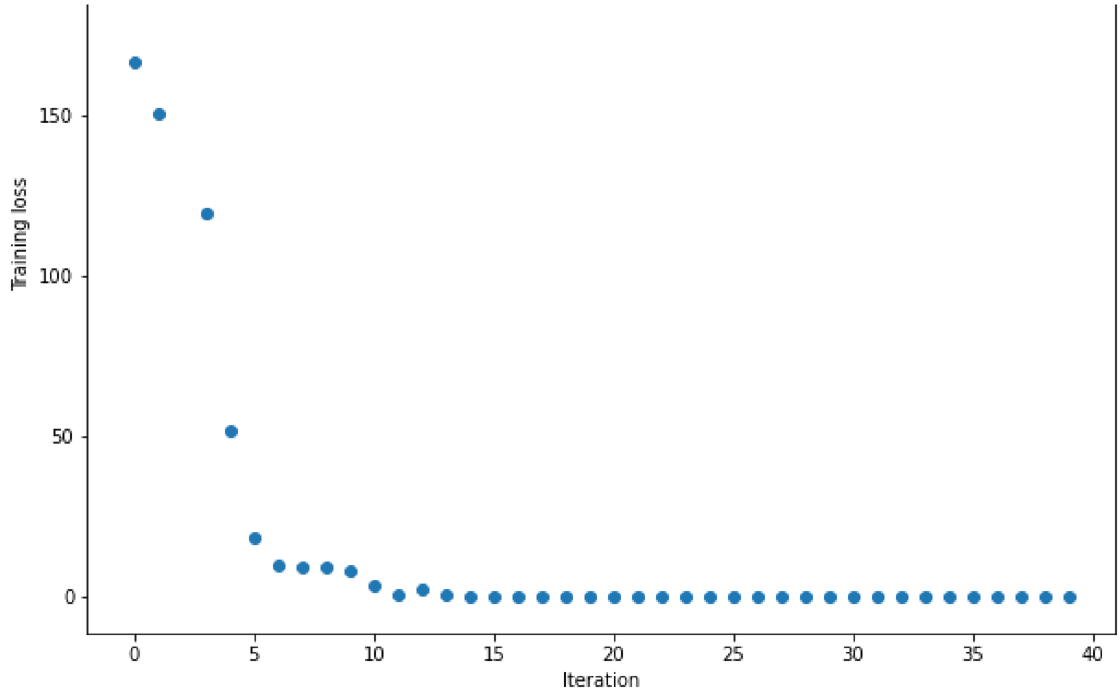
Now try to use a five-layer network with 100 units on each layer to overfit 50 training examples. Again you will have to adjust the learning rate and weight initialization, but you should be able to achieve 100% training accuracy within 20 epochs.

In [15]:

```
# TODO: Use a five-layer Net to overfit 50 training examples by
# tweaking just the learning rate and initialization scale.

num_train = 50
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

learning_rate = 2e-3
weight_scale = 1e-1
model = FullyConnectedNet([100, 100, 100, 100],
                weight_scale=weight_scale, dtype=np.float64)
solver = Solver(model, small_data,
                print_every=10, num_epochs=20, batch_size=25,
                update_rule='sgd',
                optim_config={
                  'learning_rate': learning_rate,
                }
         )
solver.train()

plt.plot(solver.loss_history, 'o')
plt.title('Training loss history')
plt.xlabel('Iteration')
plt.ylabel('Training loss')
plt.show()
```

```
(Iteration 1 / 40) loss: 166.501707
(Epoch 0 / 20) train acc: 0.100000; val_acc: 0.107000
(Epoch 1 / 20) train acc: 0.320000; val_acc: 0.101000
(Epoch 2 / 20) train acc: 0.160000; val_acc: 0.122000
(Epoch 3 / 20) train acc: 0.380000; val_acc: 0.106000
(Epoch 4 / 20) train acc: 0.520000; val_acc: 0.111000
(Epoch 5 / 20) train acc: 0.760000; val_acc: 0.113000
(Iteration 11 / 40) loss: 3.343141
(Epoch 6 / 20) train acc: 0.840000; val_acc: 0.122000
(Epoch 7 / 20) train acc: 0.920000; val_acc: 0.113000
(Epoch 8 / 20) train acc: 0.940000; val_acc: 0.125000
(Epoch 9 / 20) train acc: 0.960000; val_acc: 0.125000
(Epoch 10 / 20) train acc: 0.980000; val_acc: 0.121000
(Iteration 21 / 40) loss: 0.039138
(Epoch 11 / 20) train acc: 0.980000; val_acc: 0.123000
(Epoch 12 / 20) train acc: 1.000000; val_acc: 0.121000
(Epoch 13 / 20) train acc: 1.000000; val_acc: 0.121000
(Epoch 14 / 20) train acc: 1.000000; val_acc: 0.121000
(Epoch 15 / 20) train acc: 1.000000; val_acc: 0.121000
(Iteration 31 / 40) loss: 0.000644
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.121000
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.121000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.121000
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.121000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.121000
```





Inline Question 2:

Did you notice anything about the comparative difficulty of training the three-layer net vs training the five layer net? In particular, based on your experience, which network seemed more sensitive to the initialization scale? Why do you think that is the case?

Answer:

[FILL THIS IN]

Update rules

So far we have used vanilla stochastic gradient descent (SGD) as our update rule. More sophisticated update rules can make it easier to train deep networks. We will implement a few of the most commonly used update rules and compare them to vanilla SGD.

SGD+Momentum

Stochastic gradient descent with momentum is a widely used update rule that tends to make deep networks converge faster than vanilla stochastic gradient descent. See the Momentum Update section at http://cs231n.github.io/neural-networks-3/#sgd for more information.

Open the file `cs231n/optim.py` and read the documentation at the top of the file to make sure you understand the API. Implement the SGD+momentum update rule in the function `sgd_momentum` and run the following to check your implementation. You should see errors less than e-8.

In [16]:

```
from cs231n.optim import sgd_momentum

N, D = 4, 5
w = np.linspace(-0.4, 0.6, num=N*D).reshape(N, D)
dw = np.linspace(-0.6, 0.4, num=N*D).reshape(N, D)
v = np.linspace(0.6, 0.9, num=N*D).reshape(N, D)

config = {'learning_rate': 1e-3, 'velocity': v}
next_w, _ = sgd_momentum(w, dw, config=config)

expected_next_w = np.asarray([
```

```
  [ 0.1406,      0.20738947,  0.27417895,  0.34096842,  0.40775789],
  [ 0.47454737,  0.54133684,  0.60812632,  0.67491579,  0.74170526],
  [ 0.80849474,  0.87528421,  0.94207368,  1.00886316,  1.07565263],
  [ 1.14244211,  1.20923158,  1.27602105,  1.34281053,  1.4096    ]])
expected_velocity = np.asarray([
  [ 0.5406,      0.55475789,  0.56891579, 0.58307368,  0.59723158],
  [ 0.61138947,  0.62554737,  0.63970526,  0.65386316,  0.66802105],
  [ 0.68217895,  0.69633684,  0.71049474,  0.72465263,  0.73881053],
  [ 0.75296842,  0.76712632,  0.78128421,  0.79544211,  0.8096    ]])

# Should see relative errors around e-8 or less
print('next_w error: ', rel_error(next_w, expected_next_w))
print('velocity error: ', rel_error(expected_velocity, config['velocity']))
```

```
next_w error:  8.882347033505819e-09
velocity error:  4.269287743278663e-09
```

Once you have done so, run the following to train a six-layer network with both SGD and SGD+momentum. You should see the SGD+momentum update rule converge faster.

In [17]:

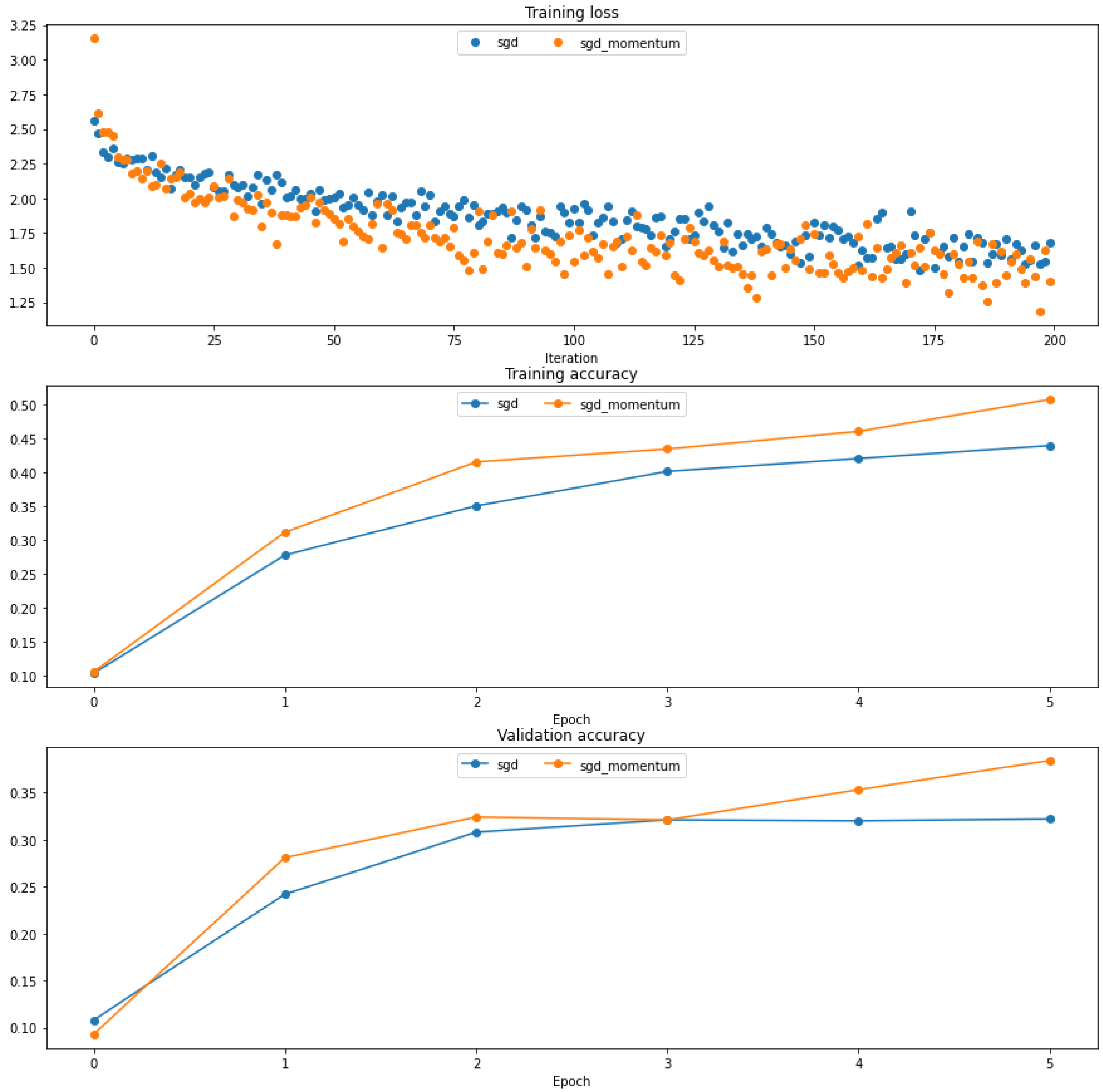
```
num_train = 4000
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

solvers = {}

for update_rule in ['sgd', 'sgd_momentum']:
  print('running with ', update_rule)
  model = FullyConnectedNet([100, 100, 100, 100, 100], weight_scale=5e-2)

  solver = Solver(model, small_data,
                  num_epochs=5, batch_size=100,
                  update_rule=update_rule,
                  optim_config={
                    'learning_rate': 1e-2,
                  },
                  verbose=True)
  solvers[update_rule] = solver
  solver.train()
  print()

plt.subplot(3, 1, 1)
plt.title('Training loss')
plt.xlabel('Iteration')

plt.subplot(3, 1, 2)
plt.title('Training accuracy')
plt.xlabel('Epoch')

plt.subplot(3, 1, 3)
plt.title('Validation accuracy')
plt.xlabel('Epoch')

for update_rule, solver in list(solvers.items()):
  plt.subplot(3, 1, 1)
  plt.plot(solver.loss_history, 'o', label=update_rule)

  plt.subplot(3, 1, 2)
  plt.plot(solver.train_acc_history, '-o', label=update_rule)

  plt.subplot(3, 1, 3)
  plt.plot(solver.val_acc_history, '-o', label=update_rule)

for i in [1, 2, 3]:
  plt.subplot(3, 1, i)
```

```
  plt.legend(loc='upper center', ncol=4)
plt.gcf().set_size_inches(15, 15)
plt.show()
```

```
running with  sgd
(Iteration 1 / 200) loss: 2.559978
(Epoch 0 / 5) train acc: 0.103000; val_acc: 0.108000
(Iteration 11 / 200) loss: 2.291086
(Iteration 21 / 200) loss: 2.153591
(Iteration 31 / 200) loss: 2.082694
(Epoch 1 / 5) train acc: 0.277000; val_acc: 0.242000
(Iteration 41 / 200) loss: 2.004171
(Iteration 51 / 200) loss: 2.010409
(Iteration 61 / 200) loss: 2.024528
(Iteration 71 / 200) loss: 2.024628
(Epoch 2 / 5) train acc: 0.350000; val_acc: 0.308000
(Iteration 81 / 200) loss: 1.804535
(Iteration 91 / 200) loss: 1.917276
(Iteration 101 / 200) loss: 1.923032
(Iteration 111 / 200) loss: 1.707939
(Epoch 3 / 5) train acc: 0.401000; val_acc: 0.321000
(Iteration 121 / 200) loss: 1.704839
(Iteration 131 / 200) loss: 1.766843
(Iteration 141 / 200) loss: 1.788663
(Iteration 151 / 200) loss: 1.828742
(Epoch 4 / 5) train acc: 0.420000; val_acc: 0.320000
(Iteration 161 / 200) loss: 1.628797
(Iteration 171 / 200) loss: 1.902930
(Iteration 181 / 200) loss: 1.542250
(Iteration 191 / 200) loss: 1.711583
(Epoch 5 / 5) train acc: 0.439000; val_acc: 0.322000

running with  sgd_momentum
(Iteration 1 / 200) loss: 3.153777
(Epoch 0 / 5) train acc: 0.105000; val_acc: 0.093000
(Iteration 11 / 200) loss: 2.145874
(Iteration 21 / 200) loss: 2.032562
(Iteration 31 / 200) loss: 1.985849
(Epoch 1 / 5) train acc: 0.311000; val_acc: 0.281000
(Iteration 41 / 200) loss: 1.882354
(Iteration 51 / 200) loss: 1.855372
(Iteration 61 / 200) loss: 1.649133
(Iteration 71 / 200) loss: 1.806432
(Epoch 2 / 5) train acc: 0.415000; val_acc: 0.324000
(Iteration 81 / 200) loss: 1.907840
(Iteration 91 / 200) loss: 1.510681
(Iteration 101 / 200) loss: 1.546872
(Iteration 111 / 200) loss: 1.512047
(Epoch 3 / 5) train acc: 0.434000; val_acc: 0.321000
(Iteration 121 / 200) loss: 1.677301
(Iteration 131 / 200) loss: 1.504686
(Iteration 141 / 200) loss: 1.633253
(Iteration 151 / 200) loss: 1.745081
(Epoch 4 / 5) train acc: 0.460000; val_acc: 0.353000
(Iteration 161 / 200) loss: 1.485411
(Iteration 171 / 200) loss: 1.610417
(Iteration 181 / 200) loss: 1.528331
(Iteration 191 / 200) loss: 1.449159
(Epoch 5 / 5) train acc: 0.507000; val_acc: 0.384000
```

```
<ipython-input-17-239314d269f9>:39: MatplotlibDeprecationWarning: Adding an axes using th
e same arguments as a previous axes currently reuses the earlier instance.  In a future v
ersion, a new instance will always be created and returned.  Meanwhile, this warning can
be suppressed, and the future behavior ensured, by passing a unique label to each axes in
stance.
  plt.subplot(3, 1, 1)
<ipython-input-17-239314d269f9>:42: MatplotlibDeprecationWarning: Adding an axes using th
e same arguments as a previous axes currently reuses the earlier instance.  In a future v
ersion, a new instance will always be created and returned.  Meanwhile, this warning can
be suppressed, and the future behavior ensured, by passing a unique label to each axes in
stance.
```

```
  plt.subplot(3, 1, 2)
<ipython-input-17-239314d269f9>:45: MatplotlibDeprecationWarning: Adding an axes using th
e same arguments as a previous axes currently reuses the earlier instance.  In a future v
ersion, a new instance will always be created and returned.  Meanwhile, this warning can
be suppressed, and the future behavior ensured, by passing a unique label to each axes in
stance.
  plt.subplot(3, 1, 3)
<ipython-input-17-239314d269f9>:49: MatplotlibDeprecationWarning: Adding an axes using th
e same arguments as a previous axes currently reuses the earlier instance.  In a future v
ersion, a new instance will always be created and returned.  Meanwhile, this warning can
be suppressed, and the future behavior ensured, by passing a unique label to each axes in
stance.
  plt.subplot(3, 1, i)
```



RMSProp and Adam

RMSProp [1] and Adam [2] are update rules that set per-parameter learning rates by using a running average of the second moments of gradients.

In the file `cs231n/optim.py`, implement the RMSProp update rule in the `rmsprop` function and implement the Adam update rule in the `adam` function, and check your implementations using the tests below.

NOTE: Please implement the *complete* Adam update rule (with the bias correction mechanism), not the first

simplified version mentioned in the course notes.

[1] Tijmen Tieleman and Geoffrey Hinton. "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude." COURSERA: Neural Networks for Machine Learning 4 (2012).

[2] Diederik Kingma and Jimmy Ba, "Adam: A Method for Stochastic Optimization", ICLR 2015.

In [18]:

```
# Test RMSProp implementation
from cs231n.optim import rmsprop

N, D = 4, 5
w = np.linspace(-0.4, 0.6, num=N*D).reshape(N, D)
dw = np.linspace(-0.6, 0.4, num=N*D).reshape(N, D)
cache = np.linspace(0.6, 0.9, num=N*D).reshape(N, D)

config = {'learning_rate': 1e-2, 'cache': cache}
next_w, _ = rmsprop(w, dw, config=config)

expected_next_w = np.asarray([
  [-0.39223849, -0.34037513, -0.28849239, -0.23659121, -0.18467247],
  [-0.132737,   -0.08078555, -0.02881884,  0.02316247,  0.07515774],
  [ 0.12716641,  0.17918792,  0.23122175,  0.28326742,  0.33532447],
  [ 0.38739248,  0.43947102,  0.49155973,  0.54365823,  0.59576619]])
expected_cache = np.asarray([
  [ 0.5976,      0.6126277,   0.6277108,   0.64284931,  0.65804321],
  [ 0.67329252,  0.68859723,  0.70395734,  0.71937285,  0.73484377],
  [ 0.75037008,  0.7659518,   0.78158892,  0.79728144,  0.81302936],
  [ 0.82883269,  0.84469141,  0.86060554,  0.87657507,  0.8926    ]])

# You should see relative errors around e-7 or less
print('next_w error: ', rel_error(expected_next_w, next_w))
print('cache error: ', rel_error(expected_cache, config['cache']))
```

```
next_w error:  9.524687511038133e-08
cache error:  2.6477955807156126e-09
```

In [19]:

```
# Test Adam implementation
from cs231n.optim import adam

N, D = 4, 5
w = np.linspace(-0.4, 0.6, num=N*D).reshape(N, D)
dw = np.linspace(-0.6, 0.4, num=N*D).reshape(N, D)
m = np.linspace(0.6, 0.9, num=N*D).reshape(N, D)
v = np.linspace(0.7, 0.5, num=N*D).reshape(N, D)

config = {'learning_rate': 1e-2, 'm': m, 'v': v, 't': 5}
next_w, _ = adam(w, dw, config=config)

expected_next_w = np.asarray([
  [-0.40094747, -0.34836187, -0.29577703, -0.24319299, -0.19060977],
  [-0.1380274,  -0.08544591, -0.03286534,  0.01971428,  0.0722929],
  [ 0.1248705,   0.17744702,  0.23002243,  0.28259667,  0.33516969],
  [ 0.38774145,  0.44031188,  0.49288093,  0.54544852,  0.59801459]])
expected_v = np.asarray([
  [ 0.69966,     0.68908382,  0.67851319,  0.66794809,  0.65738853,],
  [ 0.64683452,  0.63628604,  0.6257431,   0.61520571,  0.60467385,],
  [ 0.59414753,  0.58362676,  0.57311152,  0.56260183,  0.55209767,],
  [ 0.54159906,  0.53110598,  0.52061845,  0.51013645,  0.49966,   ]])
expected_m = np.asarray([
  [ 0.48,        0.49947368,  0.51894737,  0.53842105,  0.55789474],
  [ 0.57736842,  0.59684211,  0.61631579,  0.63578947,  0.65526316],
  [ 0.67473684,  0.69421053,  0.71368421,  0.73315789,  0.75263158],
  [ 0.77210526,  0.79157895,  0.81105263,  0.83052632,  0.85      ]])

# You should see relative errors around e-7 or less
print('next_w error: ', rel_error(expected_next_w, next_w))
print('v error: ', rel_error(expected_v, config['v']))
```

```
print('m error: ', rel_error(expected_m, config['m']))
```

```
next_w error:  1.1395691798535431e-07
v error:  4.208314038113071e-09
m error:  4.214963193114416e-09
```

Once you have debugged your RMSProp and Adam implementations, run the following to train a pair of deep networks using these new update rules:

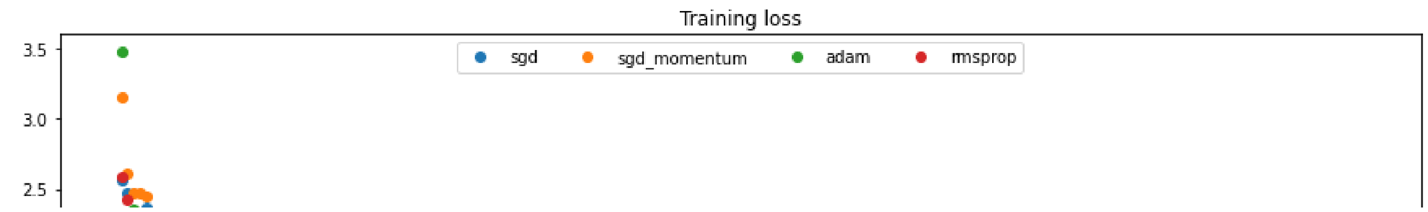
In [20]:

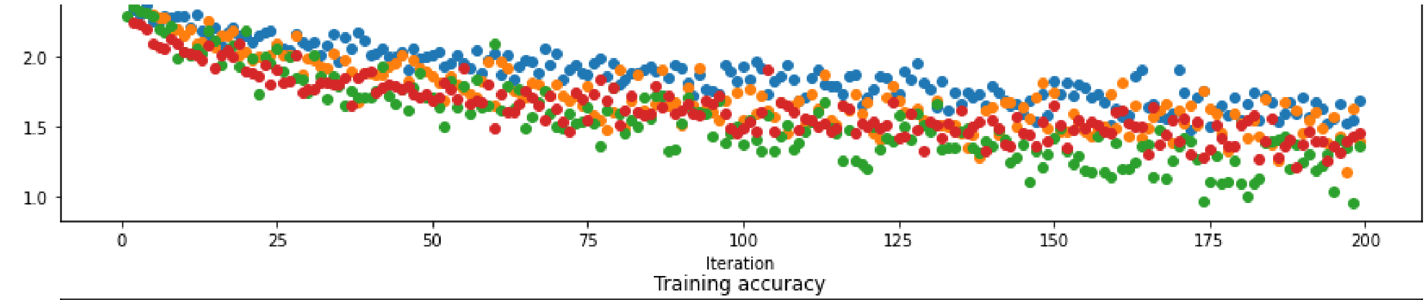
```
learning_rates = {'rmsprop': 1e-4, 'adam': 1e-3}
for update_rule in ['adam', 'rmsprop']:
  print('running with ', update_rule)
  model = FullyConnectedNet([100, 100, 100, 100, 100], weight_scale=5e-2)

  solver = Solver(model, small_data,
                  num_epochs=5, batch_size=100,
                  update_rule=update_rule,
                  optim_config={
                    'learning_rate': learning_rates[update_rule]
                  },
                  verbose=True)
  solvers[update_rule] = solver
  solver.train()
  print()

plt.subplot(3, 1, 1)
plt.title('Training loss')
plt.xlabel('Iteration')

plt.subplot(3, 1, 2)
plt.title('Training accuracy')
plt.xlabel('Epoch')

plt.subplot(3, 1, 3)
plt.title('Validation accuracy')
plt.xlabel('Epoch')

for update_rule, solver in list(solvers.items()):
  plt.subplot(3, 1, 1)
  plt.plot(solver.loss_history, 'o', label=update_rule)

  plt.subplot(3, 1, 2)
  plt.plot(solver.train_acc_history, '-o', label=update_rule)

  plt.subplot(3, 1, 3)
  plt.plot(solver.val_acc_history, '-o', label=update_rule)

for i in [1, 2, 3]:
  plt.subplot(3, 1, i)
  plt.legend(loc='upper center', ncol=4)
plt.gcf().set_size_inches(15, 15)
plt.show()
```

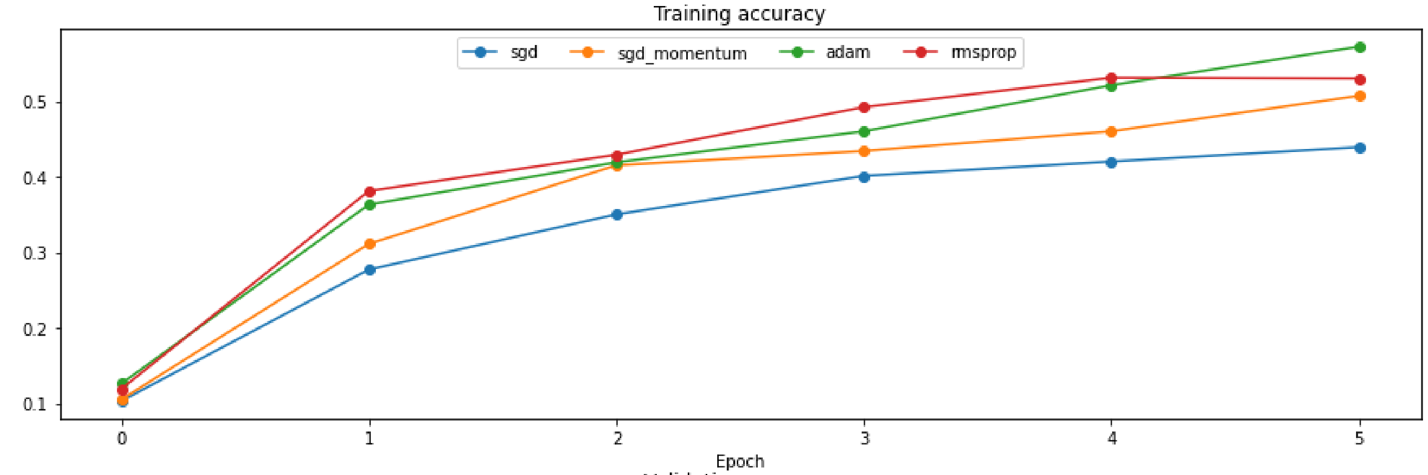
```
running with  adam
(Iteration 1 / 200) loss: 3.476928
(Epoch 0 / 5) train acc: 0.126000; val_acc: 0.110000
(Iteration 11 / 200) loss: 2.027712
(Iteration 21 / 200) loss: 2.183358
(Iteration 31 / 200) loss: 1.744257
(Epoch 1 / 5) train acc: 0.363000; val_acc: 0.330000
(Iteration 41 / 200) loss: 1.707951
(Iteration 51 / 200) loss: 1.703835
(Iteration 61 / 200) loss: 2.094758
(Iteration 71 / 200) loss: 1.505558
(Epoch 2 / 5) train acc: 0.419000; val_acc: 0.362000
(Iteration 81 / 200) loss: 1.594429
(Iteration 91 / 200) loss: 1.519017
(Iteration 101 / 200) loss: 1.368522
(Iteration 111 / 200) loss: 1.470400
```

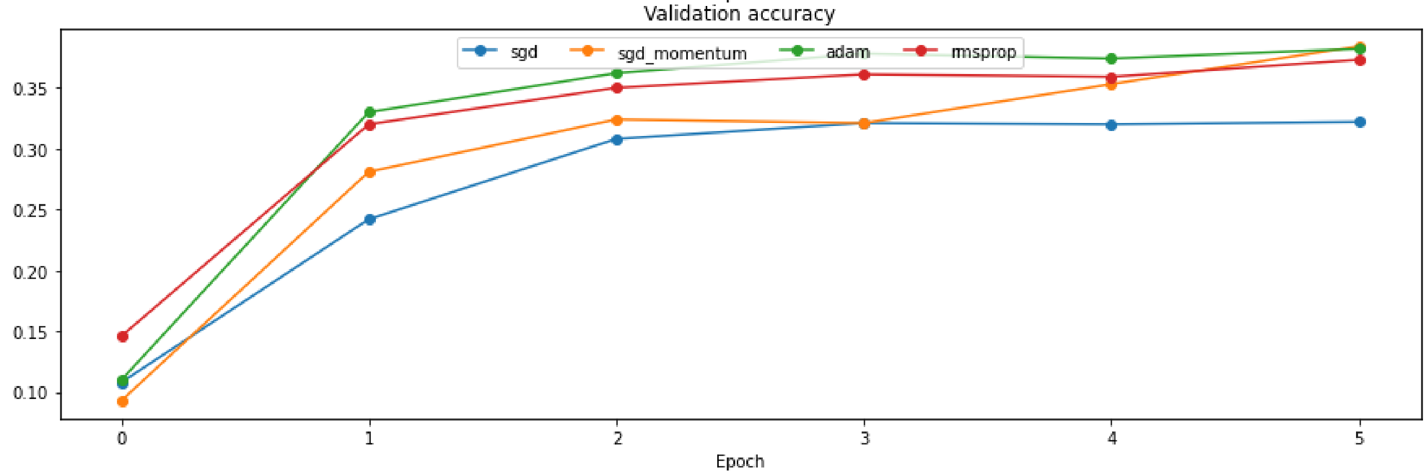
```
(Epoch 3 / 5) train acc: 0.460000; val_acc: 0.378000
(Iteration 121 / 200) loss: 1.199064
(Iteration 131 / 200) loss: 1.464705
(Iteration 141 / 200) loss: 1.359863
(Iteration 151 / 200) loss: 1.415069
(Epoch 4 / 5) train acc: 0.521000; val_acc: 0.374000
(Iteration 161 / 200) loss: 1.382818
(Iteration 171 / 200) loss: 1.359900
(Iteration 181 / 200) loss: 1.095947
(Iteration 191 / 200) loss: 1.243088
(Epoch 5 / 5) train acc: 0.572000; val_acc: 0.382000

running with  rmsprop
(Iteration 1 / 200) loss: 2.589166
(Epoch 0 / 5) train acc: 0.119000; val_acc: 0.146000
(Iteration 11 / 200) loss: 2.032921
(Iteration 21 / 200) loss: 1.897278
(Iteration 31 / 200) loss: 1.770793
(Epoch 1 / 5) train acc: 0.381000; val_acc: 0.320000
(Iteration 41 / 200) loss: 1.895731
(Iteration 51 / 200) loss: 1.681091
(Iteration 61 / 200) loss: 1.487204
(Iteration 71 / 200) loss: 1.629973
(Epoch 2 / 5) train acc: 0.429000; val_acc: 0.350000
(Iteration 81 / 200) loss: 1.506686
(Iteration 91 / 200) loss: 1.610742
(Iteration 101 / 200) loss: 1.486124
(Iteration 111 / 200) loss: 1.559454
(Epoch 3 / 5) train acc: 0.492000; val_acc: 0.361000
(Iteration 121 / 200) loss: 1.497406
(Iteration 131 / 200) loss: 1.530736
(Iteration 141 / 200) loss: 1.550958
(Iteration 151 / 200) loss: 1.652026
(Epoch 4 / 5) train acc: 0.531000; val_acc: 0.359000
(Iteration 161 / 200) loss: 1.600752
(Iteration 171 / 200) loss: 1.400347
(Iteration 181 / 200) loss: 1.509237
(Iteration 191 / 200) loss: 1.368884
(Epoch 5 / 5) train acc: 0.530000; val_acc: 0.373000
```

```
<ipython-input-20-c31f2247ce3b>:30: MatplotlibDeprecationWarning: Adding an axes using th
e same arguments as a previous axes currently reuses the earlier instance.  In a future v
ersion, a new instance will always be created and returned.  Meanwhile, this warning can
be suppressed, and the future behavior ensured, by passing a unique label to each axes in
stance.
  plt.subplot(3, 1, 1)
<ipython-input-20-c31f2247ce3b>:33: MatplotlibDeprecationWarning: Adding an axes using th
e same arguments as a previous axes currently reuses the earlier instance.  In a future v
ersion, a new instance will always be created and returned.  Meanwhile, this warning can
be suppressed, and the future behavior ensured, by passing a unique label to each axes in
stance.
  plt.subplot(3, 1, 2)
<ipython-input-20-c31f2247ce3b>:36: MatplotlibDeprecationWarning: Adding an axes using th
e same arguments as a previous axes currently reuses the earlier instance.  In a future v
ersion, a new instance will always be created and returned.  Meanwhile, this warning can
be suppressed, and the future behavior ensured, by passing a unique label to each axes in
stance.
  plt.subplot(3, 1, 3)
<ipython-input-20-c31f2247ce3b>:40: MatplotlibDeprecationWarning: Adding an axes using th
e same arguments as a previous axes currently reuses the earlier instance.  In a future v
ersion, a new instance will always be created and returned.  Meanwhile, this warning can
be suppressed, and the future behavior ensured, by passing a unique label to each axes in
stance.
  plt.subplot(3, 1, i)
```









Inline Question 3:

AdaGrad, like Adam, is a per-parameter optimization method that uses the following update rule:

```
cache += dw**2
w += - learning_rate * dw / (np.sqrt(cache) + eps)
```

John notices that when he was training a network with AdaGrad that the updates became very small, and that his network was learning slowly. Using your knowledge of the AdaGrad update rule, why do you think the updates would become very small? Would Adam have the same issue?

Answer:

We are taking squared gradients (dw) and they are therefore positive. Accumulated sum gets larger each time. This summation makes modified learning rate converge to zero and learning process slows down. This problem would not happen with Adam.

Train a good model!

Train the best fully-connected model that you can on CIFAR-10, storing your best model in the `best_model` **variable. We require you to get at least 50% accuracy on the validation set using a fully-connected net.**

If you are careful it should be possible to get accuracies above 55%, but we don't require it for this part and won't assign extra credit for doing so. Later in the assignment we will ask you to train the best convolutional network that you can on CIFAR-10, and we would prefer that you spend your effort working on convolutional

nets rather than fully-connected nets.

You might find it useful to complete the `BatchNormalization.ipynb` and `Dropout.ipynb` notebooks before completing this part, since those techniques can help you train powerful models.

In [22]:

```
best_model = None
################################################################################
# TODO: Train the best FullyConnectedNet that you can on CIFAR-10. You might   #
# find batch/layer normalization and dropout useful. Store your best model in  #
# the best_model variable.                                                     #
################################################################################

hidden_dims = [100] * 4

range_weight_scale = [1e-2, 2e-2, 5e-3]
range_lr = [1e-5, 5e-4, 1e-5]

best_val_acc = -1
best_weight_scale = 0
best_lr = 0

print("Training...")

for weight_scale in range_weight_scale:
    for lr in range_lr:
        model = FullyConnectedNet(hidden_dims=hidden_dims, reg=0.0,
                                  weight_scale=weight_scale)
        solver = Solver(model, data, update_rule='adam',
                        optim_config={'learning_rate': lr},
                        batch_size=100, num_epochs=5,
                        verbose=False)
        solver.train()
        val_acc = solver.best_val_acc

        print('Weight_scale: %f, lr: %f, val_acc: %f' % (weight_scale, lr, val_acc))

        if val_acc > best_val_acc:
            best_val_acc = val_acc
            best_weight_scale = weight_scale
            best_lr = lr
            best_model = model

print("Best val_acc: %f" % best_val_acc)
print("Best weight_scale: %f" % best_weight_scale)
print("Best lr: %f" % best_lr)

################################################################################
#                              END OF YOUR CODE                                #
################################################################################
```

```
Training...
Weight_scale: 0.010000, lr: 0.000010, val_acc: 0.338000
Weight_scale: 0.010000, lr: 0.000500, val_acc: 0.503000
Weight_scale: 0.010000, lr: 0.000010, val_acc: 0.342000
Weight_scale: 0.020000, lr: 0.000010, val_acc: 0.427000
Weight_scale: 0.020000, lr: 0.000500, val_acc: 0.523000
Weight_scale: 0.020000, lr: 0.000010, val_acc: 0.418000
Weight_scale: 0.005000, lr: 0.000010, val_acc: 0.274000
Weight_scale: 0.005000, lr: 0.000500, val_acc: 0.513000
Weight_scale: 0.005000, lr: 0.000010, val_acc: 0.257000
Best val_acc: 0.523000
Best weight_scale: 0.020000
Best lr: 0.000500
```

Test your model!

Run your best model on the validation and test sets. You should achieve above 50% accuracy on the validation set.

set.

In [23]:

```
y_test_pred = np.argmax(best_model.loss(data['X_test']), axis=1)
y_val_pred = np.argmax(best_model.loss(data['X_val']), axis=1)
print('Validation set accuracy: ', (y_val_pred == data['y_val']).mean())
print('Test set accuracy: ', (y_test_pred == data['y_test']).mean())
```

```
Validation set accuracy:  0.523
Test set accuracy:  0.51
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

Dropout

Dropout [1] is a technique for regularizing neural networks by randomly setting some features to zero during the forward pass. In this exercise you will implement a dropout layer and modify your fully-connected network to optionally use dropout.

[1] [Geoffrey E. Hinton et al, "Improving neural networks by preventing co-adaptation of feature detectors", arXiv 2012]()

In [17]:

```
# As usual, a bit of setup
from __future__ import print_function
import time
import numpy as np
import matplotlib.pyplot as plt
from cs231n.classifiers.fc_net import *
from cs231n.data_utils import get_CIFAR10_data
from cs231n.gradient_check import eval_numerical_gradient, eval_numerical_gradient_array
from cs231n.solver import Solver

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2

def rel_error(x, y):
  """ returns relative error """
  return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

In [18]:

```
# Load the (preprocessed) CIFAR10 data.

data = get_CIFAR10_data()
for k, v in data.items():
  print('%s: ' % k, v.shape)
```

```
X_train:  (49000, 3, 32, 32)
y_train:  (49000,)
X_val:  (1000, 3, 32, 32)
y_val:  (1000,)
X_test:  (1000, 3, 32, 32)
y_test:  (1000,)
```

Dropout forward pass

In the file `cs231n/layers.py`, implement the forward pass for dropout. Since dropout behaves differently during training and testing, make sure to implement the operation for both modes.

Once you have done so, run the cell below to test your implementation.

In [24]:

```
np.random.seed(231)
x = np.random.randn(500, 500) + 10
```

```
for p in [0.25, 0.4, 0.7]:
  out, _ = dropout_forward(x, {'mode': 'train', 'p': p})
  out_test, _ = dropout_forward(x, {'mode': 'test', 'p': p})

  print('Running tests with p = ', p)
  print('Mean of input: ', x.mean())
  print('Mean of train-time output: ', out.mean())
  print('Mean of test-time output: ', out_test.mean())
  print('Fraction of train-time output set to zero: ', (out == 0).mean())
  print('Fraction of test-time output set to zero: ', (out_test == 0).mean())
  print()
```

```
Running tests with p =  0.25
Mean of input:  10.000207878477502
Mean of train-time output:  10.014059116977283
Mean of test-time output:  10.000207878477502
Fraction of train-time output set to zero:  0.749784
Fraction of test-time output set to zero:  0.0

Running tests with p =  0.4
Mean of input:  10.000207878477502
Mean of train-time output:  9.977917658761159
Mean of test-time output:  10.000207878477502
Fraction of train-time output set to zero:  0.600796
Fraction of test-time output set to zero:  0.0

Running tests with p =  0.7
Mean of input:  10.000207878477502
Mean of train-time output:  9.987811912159426
Mean of test-time output:  10.000207878477502
Fraction of train-time output set to zero:  0.30074
Fraction of test-time output set to zero:  0.0
```

Dropout backward pass

In the file `cs231n/layers.py`, implement the backward pass for dropout. After doing so, run the following cell to numerically gradient-check your implementation.

In [23]:

```
np.random.seed(231)
x = np.random.randn(10, 10) + 10
dout = np.random.randn(*x.shape)

dropout_param = {'mode': 'train', 'p': 0.2, 'seed': 123}
out, cache = dropout_forward(x, dropout_param)
dx = dropout_backward(dout, cache)
dx_num = eval_numerical_gradient_array(lambda xx: dropout_forward(xx, dropout_param)[0],
x, dout)

# Error should be around e-10 or less
print('dx relative error: ', rel_error(dx, dx_num))
```

```
dx relative error:  1.892896954038074e-11
```

Inline Question 1:

What happens if we do not divide the values being passed through inverse dropout by `p` in the dropout layer? Why does that happen?

Answer:

Mean of train-time output changes without division by p in dropout_forward function.

Running tests with p = 0.25 Mean of train-time output: 10.014059116977283 Mean of train-time output (without

Running tests with p = 0.25 Mean of train-time output: 10.014059116977288 Mean of train-time output (without division): 2.5035147792443206

Running tests with p = 0.4 Mean of train-time output: 9.977917658761159 Mean of train-time output (without division): 3.991167063504464

Running tests with p = 0.7 Mean of train-time output: 9.987811912159426 Mean of train-time output (without division): 6.9914683385116

The reason is that outputs are set to zero with probability p and mean drops with proportion p. Summation of outputs drops with proportion p, therefore to keep the mean constant, division by p is required.

Fully-connected nets with Dropout

In the file `cs231n/classifiers/fc_net.py`, modify your implementation to use dropout. Specifically, if the constructor of the net receives a value that is not 1 for the `dropout` parameter, then the net should add dropout immediately after every ReLU nonlinearity. After doing so, run the following to numerically gradient-check your implementation.

In [25]:

```
np.random.seed(231)
N, D, H1, H2, C = 2, 15, 20, 30, 10
X = np.random.randn(N, D)
y = np.random.randint(C, size=(N,))

for dropout in [1, 0.75, 0.5]:
  print('Running check with dropout = ', dropout)
  model = FullyConnectedNet([H1, H2], input_dim=D, num_classes=C,
                            weight_scale=5e-2, dtype=np.float64,
                            dropout=dropout, seed=123)

  loss, grads = model.loss(X, y)
  print('Initial loss: ', loss)

  # Relative errors should be around e-6 or less; Note that it's fine
  # if for dropout=1 you have W2 error be on the order of e-5.
  for name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    grad_num = eval_numerical_gradient(f, model.params[name], verbose=False, h=1e-5)
    print('%s relative error: %.2e' % (name, rel_error(grad_num, grads[name])))
  print()
```

```
Running check with dropout =  1
Initial loss:  2.3004790897684924
W1 relative error: 1.48e-07
W2 relative error: 2.21e-05
W3 relative error: 3.53e-07
b1 relative error: 5.38e-09
b2 relative error: 2.09e-09
b3 relative error: 5.80e-11

Running check with dropout =  0.75
Initial loss:  2.302371489704412
W1 relative error: 1.90e-07
W2 relative error: 4.76e-06
W3 relative error: 2.60e-08
b1 relative error: 4.73e-09
b2 relative error: 1.82e-09
b3 relative error: 1.70e-10

Running check with dropout =  0.5
Initial loss:  2.3042759220785896
W1 relative error: 3.11e-07
W2 relative error: 1.84e-08
W3 relative error: 5.35e-08
b1 relative error: 5.37e-09
b2 relative error: 2.99e-09
b3 relative error: 1.13e-10
```

Regularization experiment

As an experiment, we will train a pair of two-layer networks on 500 training examples: one will use no dropout, and one will use a keep probability of 0.25. We will then visualize the training and validation accuracies of the two networks over time.

In [26]:

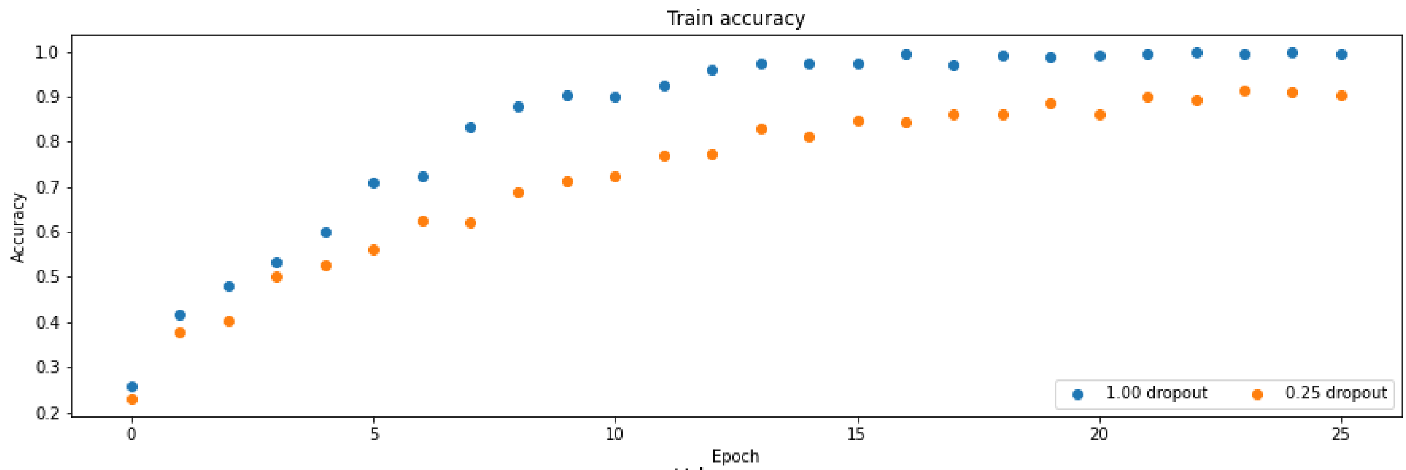
```
# Train two identical nets, one with dropout and one without
np.random.seed(231)
num_train = 500
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

solvers = {}
dropout_choices = [1, 0.25]
for dropout in dropout_choices:
  model = FullyConnectedNet([500], dropout=dropout)
  print(dropout)

  solver = Solver(model, small_data,
                  num_epochs=25, batch_size=100,
                  update_rule='adam',
                  optim_config={
                    'learning_rate': 5e-4,
                  },
                  verbose=True, print_every=100)
  solver.train()
  solvers[dropout] = solver
```

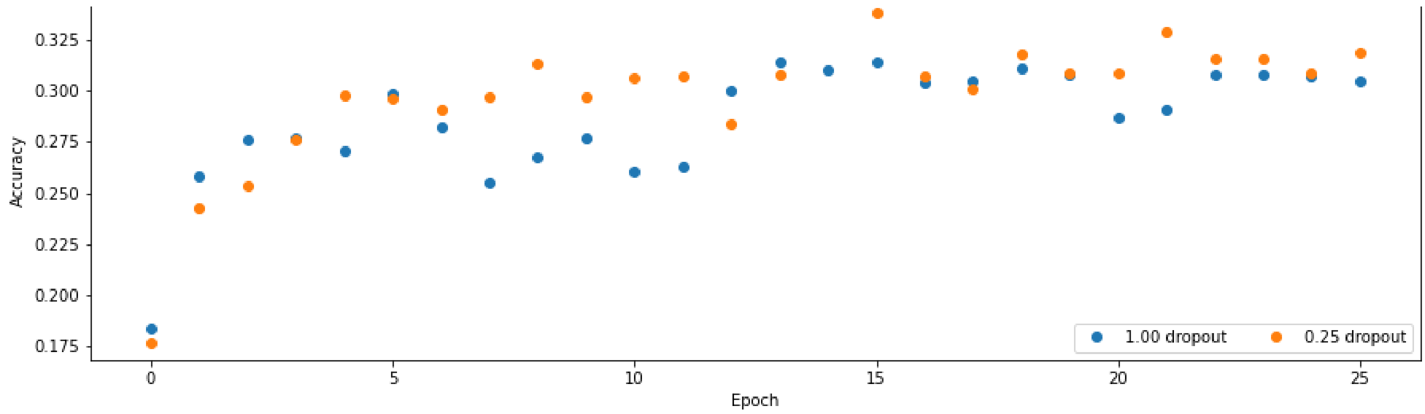
```
1
(Iteration 1 / 125) loss: 7.856643
(Epoch 0 / 25) train acc: 0.260000; val_acc: 0.184000
(Epoch 1 / 25) train acc: 0.416000; val_acc: 0.258000
(Epoch 2 / 25) train acc: 0.482000; val_acc: 0.276000
(Epoch 3 / 25) train acc: 0.532000; val_acc: 0.277000
(Epoch 4 / 25) train acc: 0.600000; val_acc: 0.271000
(Epoch 5 / 25) train acc: 0.708000; val_acc: 0.299000
(Epoch 6 / 25) train acc: 0.722000; val_acc: 0.282000
(Epoch 7 / 25) train acc: 0.832000; val_acc: 0.255000
(Epoch 8 / 25) train acc: 0.880000; val_acc: 0.268000
(Epoch 9 / 25) train acc: 0.902000; val_acc: 0.277000
(Epoch 10 / 25) train acc: 0.898000; val_acc: 0.261000
(Epoch 11 / 25) train acc: 0.924000; val_acc: 0.263000
(Epoch 12 / 25) train acc: 0.960000; val_acc: 0.300000
(Epoch 13 / 25) train acc: 0.972000; val_acc: 0.314000
(Epoch 14 / 25) train acc: 0.972000; val_acc: 0.310000
(Epoch 15 / 25) train acc: 0.974000; val_acc: 0.314000
(Epoch 16 / 25) train acc: 0.994000; val_acc: 0.304000
(Epoch 17 / 25) train acc: 0.970000; val_acc: 0.305000
(Epoch 18 / 25) train acc: 0.990000; val_acc: 0.311000
(Epoch 19 / 25) train acc: 0.988000; val_acc: 0.308000
(Epoch 20 / 25) train acc: 0.992000; val_acc: 0.287000
(Iteration 101 / 125) loss: 0.001417
(Epoch 21 / 25) train acc: 0.994000; val_acc: 0.291000
(Epoch 22 / 25) train acc: 0.998000; val_acc: 0.308000
(Epoch 23 / 25) train acc: 0.996000; val_acc: 0.308000
(Epoch 24 / 25) train acc: 0.998000; val_acc: 0.307000
(Epoch 25 / 25) train acc: 0.994000; val_acc: 0.305000
0.25
(Iteration 1 / 125) loss: 17.318478
(Epoch 0 / 25) train acc: 0.230000; val_acc: 0.177000
(Epoch 1 / 25) train acc: 0.378000; val_acc: 0.243000
(Epoch 2 / 25) train acc: 0.402000; val_acc: 0.254000
```

```
(Epoch 3 / 25) train acc: 0.502000; val_acc: 0.276000
(Epoch 4 / 25) train acc: 0.528000; val_acc: 0.298000
(Epoch 5 / 25) train acc: 0.562000; val_acc: 0.296000
(Epoch 6 / 25) train acc: 0.626000; val_acc: 0.291000
(Epoch 7 / 25) train acc: 0.622000; val_acc: 0.297000
(Epoch 8 / 25) train acc: 0.688000; val_acc: 0.313000
(Epoch 9 / 25) train acc: 0.712000; val_acc: 0.297000
(Epoch 10 / 25) train acc: 0.724000; val_acc: 0.306000
(Epoch 11 / 25) train acc: 0.768000; val_acc: 0.307000
(Epoch 12 / 25) train acc: 0.774000; val_acc: 0.284000
(Epoch 13 / 25) train acc: 0.828000; val_acc: 0.308000
(Epoch 14 / 25) train acc: 0.812000; val_acc: 0.346000
(Epoch 15 / 25) train acc: 0.848000; val_acc: 0.338000
(Epoch 16 / 25) train acc: 0.844000; val_acc: 0.307000
(Epoch 17 / 25) train acc: 0.860000; val_acc: 0.301000
(Epoch 18 / 25) train acc: 0.862000; val_acc: 0.318000
(Epoch 19 / 25) train acc: 0.886000; val_acc: 0.309000
(Epoch 20 / 25) train acc: 0.860000; val_acc: 0.309000
(Iteration 101 / 125) loss: 4.193681
(Epoch 21 / 25) train acc: 0.898000; val_acc: 0.329000
(Epoch 22 / 25) train acc: 0.892000; val_acc: 0.316000
(Epoch 23 / 25) train acc: 0.914000; val_acc: 0.316000
(Epoch 24 / 25) train acc: 0.910000; val_acc: 0.309000
(Epoch 25 / 25) train acc: 0.902000; val_acc: 0.319000
```

In [27]:

```python
# Plot train and validation accuracies of the two models

train_accs = []
val_accs = []
for dropout in dropout_choices:
  solver = solvers[dropout]
  train_accs.append(solver.train_acc_history[-1])
  val_accs.append(solver.val_acc_history[-1])

plt.subplot(3, 1, 1)
for dropout in dropout_choices:
  plt.plot(solvers[dropout].train_acc_history, 'o', label='%.2f dropout' % dropout)
plt.title('Train accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(ncol=2, loc='lower right')

plt.subplot(3, 1, 2)
for dropout in dropout_choices:
  plt.plot(solvers[dropout].val_acc_history, 'o', label='%.2f dropout' % dropout)
plt.title('Val accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(ncol=2, loc='lower right')

plt.gcf().set_size_inches(15, 15)
plt.show()
```





Inline Question 2:

Compare the validation and training accuracies with and without dropout -- what do your results suggest about dropout as a regularizer?

Answer:

Dropout operation results in sacrificing learnt weight parameters and their resulting feature extraction informations. Therefore, when there is dropout, training accuracy follows the same trend for convergence, but convergence occurs at a lower level of accuracy for training accuracy.

We cannot say directly that validation accuracy will increase or decrease after dropout. At each epoch, it gives a different result. However, if we look at the best curve for validation accuracy with and without dropout, we can see that best curve of validation accuracy with dropout is higher than the case without dropout although training accuracy with dropout was lower than the case without dropout. Here, it can be stated that dropout gives good results for generalization and helps avoiding overfitting.

Inline Question 3:

Suppose we are training a deep fully-connected network for image classification, with dropout after hidden layers (parameterized by keep probability p). How should we modify p, if at all, if we decide to decrease the size of the hidden layers (that is, the number of nodes in each layer)?

Answer:

When the number of nodes in hidden layers decrease, it means that the learnt features of training set decreases. If we apply dropout with same probability p, it means that we will lose the same amount of feature information and the learnt feature information was already lower than previous case. Hence, Keeping the dropout probability p constant will deteriorate performance of neural network. As a remedy, We should increase the probability p to keep more feature information than the previous case.

In [ ]:

```
```

In [ ]:

```
```

APPENDIX

MATLAB Code

```
function berfin_kavsut_21602459_hw2(question)
clc
close all

switch question

    case '1'
    disp('Q1')

    %read dataset
    trainims = h5read('assign2_data1.h5','/trainims');
    trainlbls = h5read('assign2_data1.h5','/trainlbls');
    testims = h5read('assign2_data1.h5','/testims');
    testlbls = h5read('assign2_data1.h5','/testlbls');

    trainlbls(trainlbls == 0) = -1;
    testlbls(testlbls == 0) = -1;

    %data type conversion
    train_images = im2double(trainims);
    test_images = im2double(testims);

    %size of images, training image number and test image number
    [m,n,train_no] = size(train_images);
    [~,~,test_no] = size(test_images);

    %training set and test set
    X_train = reshape(train_images,m*n,train_no);
    X_test = reshape(test_images,m*n,test_no);
    y_train = trainlbls;
    y_test = testlbls;

    %parameters
    lr = 0.15; % learning rate in range of [0.1,0.5]
    batch_size = 100; %mini batch size
    epoch_no = 500;

    class_no = 2; %output neuron number

    disp('Part A')
    N = 10; %hidden layer neuron number
    history = neural_net_Q1(X_train, y_train, X_test, y_test,
batch_size,lr,epoch_no,class_no,N);

    %display error metrics for Part A
    figure;set(gcf, 'WindowState', 'maximized');
    subplot(2,1,1); plot(1:epoch_no, history.train_MSE);
```

```
    hold on; plot(1:epoch_no,history.test_MSE);
    legend('Training Set','Test Set')
    title('Mean Squared Error');

    subplot(2,1,2); plot(1:epoch_no, history.train_MCE);
    hold on; plot(1:epoch_no,history.test_MCE);
    legend('Training Set','Test Set')
    title('Classification Error')
    %saveas(gcf,'Q1_Part_A.png');

    disp('Part B')
    disp('In the report.')

    disp('Part C')
    N_opt = 10;
    N_low = 2;
    N_high = 50;
    history_low = neural_net_Q1(X_train, y_train, X_test, y_test,
batch_size,lr,epoch_no,class_no,N_low);
    history_opt = neural_net_Q1(X_train, y_train, X_test, y_test,
batch_size,lr,epoch_no,class_no,N_opt);
    history_high = neural_net_Q1(X_train, y_train, X_test, y_test,
batch_size,lr,epoch_no,class_no,N_high);

    figure;set(gcf, 'WindowState', 'maximized');
    subplot(2,2,1); plot(1:epoch_no, history_low.train_MSE)
    hold on; plot(1:epoch_no, history_opt.train_MSE)
    hold on; plot(1:epoch_no, history_high.train_MSE)
    legend('N_{low}','N^*','N_{high}')
    title('Mean Squared Error of Training Set')

    subplot(2,2,2); plot(1:epoch_no, history_low.train_MCE)
    hold on; plot(1:epoch_no, history_opt.train_MCE)
    hold on; plot(1:epoch_no, history_high.train_MCE)
    legend('N_{low}','N^*','N_{high}')
    title('Classification Error of Training Set')

    subplot(2,2,3); plot(1:epoch_no, history_low.test_MSE)
    hold on; plot(1:epoch_no, history_opt.test_MSE)
    hold on; plot(1:epoch_no, history_high.test_MSE)
    legend('N_{low}','N^*','N_{high}')
    title('Mean Squared Error of Test Set')

    subplot(2,2,4); plot(1:epoch_no, history_low.test_MCE)
    hold on; plot(1:epoch_no, history_opt.test_MCE)
    hold on; plot(1:epoch_no, history_high.test_MCE)
    legend('N_{low}','N^*','N_{high}')
    title('Classification Error of Test Set')
    %saveas(gcf,'Q1_Part_C.png');

    disp('Part D')
```

```
    lr = 0.15;
    alpha = 0; %without momentum
    N1 = 10;
    N2 = 10;
    history = neural_net2_Q1(X_train, y_train, X_test, y_test,
batch_size,...
    lr,alpha,epoch_no,class_no,N1,N2);

    %error metrics for part d
    figure;set(gcf, 'WindowState', 'maximized');
    subplot(2,1,1); plot(1:epoch_no, history.train_MSE);
    hold on; plot(1:epoch_no,history.test_MSE);
    legend('Training Set','Test Set')
    title('Mean Squared Error (without momentum)');

    subplot(2,1,2); plot(1:epoch_no, history.train_MCE);
    hold on; plot(1:epoch_no,history.test_MCE);
    legend('Training Set','Test Set')
    title('Classification Error (without momentum)')
    %saveas(gcf,'Q1_Part_D.png');

    disp('Part E')

    lr = 0.15;
    alpha = 0.5;
    N1 = 10;
    N2 = 10;
    history = neural_net2_Q1(X_train, y_train, X_test, y_test,
batch_size,...
    lr,alpha,epoch_no,class_no,N1,N2);

    %error metrics for part e
    figure;set(gcf, 'WindowState', 'maximized');
    subplot(2,1,1); plot(1:epoch_no, history.train_MSE);
    hold on; plot(1:epoch_no,history.test_MSE);
    legend('Training Set','Test Set')
    title('Mean Squared Error  (with momentum)');

    subplot(2,1,2); plot(1:epoch_no, history.train_MCE);
    hold on; plot(1:epoch_no,history.test_MCE);
    legend('Training Set','Test Set')
    title('Classification Error (with momentum)')
    %saveas(gcf,'Q1_Part_E.png');

    case '2'
    disp('Q2')

    testd = h5read('assign2_data2.h5','/testd');
    testx = h5read('assign2_data2.h5','/testx');
    traind = h5read('assign2_data2.h5','/traind');
    trainx = h5read('assign2_data2.h5','/trainx');
    vald = h5read('assign2_data2.h5','/vald');
```

```
    valx = h5read('assign2_data2.h5','/valx');

    disp('Part A')
    lr = 0.15;
    alpha = 0.85;
    epoch_no = 50;
    batch_size = 200;

    %(D,P) = (32,256)
    D = 32;
    P = 256;
    [history,~] =
neural_net_Q2(trainx,traind,valx,vald,testx,testd,epoch_no,batch_size,l
r,alpha,D,P);

    figure;set(gcf, 'WindowState', 'maximized');
    subplot(2,1,1); plot(1:history.epoch,history.train_CE);
    hold on; plot(1:history.epoch,history.val_CE);
    title('Cross Entropy Error (D,P) = (32,256)')
    legend('Training Set','Validation Set')

    subplot(2,1,2); plot(1:history.epoch,history.train_MCE);
    title('Classification Error (D,P) = (32,256)')
    hold on; plot(1:history.epoch,history.val_MCE);
    legend('Training Set','Validation Set')
    %saveas(gcf,'Q2_(32,256).png')

    %(D,P) = (16,128)
    D = 16;
    P = 128;
    [history,weights] =
neural_net_Q2(trainx,traind,valx,vald,testx,testd,epoch_no,batch_size,l
r,alpha,D,P);

    figure;set(gcf, 'WindowState', 'maximized');
    subplot(2,1,1); plot(1:history.epoch,history.train_CE);
    title('Cross Entropy Error (D,P) = (16,128)')
    hold on; plot(1:history.epoch,history.val_CE);
    legend('Training Set','Validation Set')

    subplot(2,1,2); plot(1:history.epoch,history.train_MCE);
    title('Classification Error (D,P) = (16,128)')
    hold on; plot(1:history.epoch,history.val_MCE);
    legend('Training Set','Validation Set')
    %saveas(gcf,'Q2_(16,128).png')

    %(D,P) = (8,64)
    D = 8;
    P = 64;
    [history,~] =
neural_net_Q2(trainx,traind,valx,vald,testx,testd,epoch_no,batch_size,l
r,alpha,D,P);
```

```
    figure;set(gcf, 'WindowState', 'maximized');
    subplot(2,1,1); plot(1:history.epoch,history.train_CE);
    title('Cross Entropy Error (D,P) = (8,64)')
    hold on; plot(1:history.epoch,history.val_CE);
    legend('Training Set','Validation Set')

    subplot(2,1,2); plot(1:history.epoch,history.train_MCE);
    title('Classification Error (D,P) = (8,64)')
    hold on; plot(1:history.epoch,history.val_MCE);
    legend('Training Set','Validation Set')
    %saveas(gcf,'Q2_(8,64).png')

    disp('Part B')

    WE = weights.WE;
    W1 = weights.W1;
    W2 = weights.W2;

    test_no = size(testd,1);
    [X1_test,X2_test,X3_test,D_out_test] = prepare_words(testx,testd);

    %Select 5 3-words sequences and their desired outputs
    indices = 7500*[1:5];
    X1 = X1_test(:,indices);
    X2 = X2_test(:,indices);
    X3 = X3_test(:,indices);
    D_out = testd(indices);

    %TEST SET FORWARD PROPAGATION
    E1 = WE*X1;
    E2 = WE*X2;
    E3 = WE*X3;
    E = [E1;E2;E3;ones(1,5)]; %bias term
    V1 = W1*E;
    [Y1,~] = sigmoid(V1,1);
    Y1 = [Y1;ones(1,5)]; %bias term
    V2 = W2*Y1;
    Y2 = softmax(V2); %the predicted probability for each of the 250
words is Y2

    for i = 1:5

        disp(strcat('Triagram #',num2str(i),':'));

        disp('Desired Output:');
        desired_output = testd(i);
        disp(desired_output);

        disp('Top 10 Candidates for the Fourth Word:');
        [sorted,index] = sort(Y2(:,i),'descend'); %sort in descending
order
```

```
        top_10 = index(1:10,:);%show indices with 10 highest
probabilities
        disp(top_10);

    end

    case '3'
    disp('Q3')
    disp('In the report.')

end

end

function [X1,X2,X3,D] = prepare_words(x,d)
    %Create word vectors with one hot representation
    %Inputs
    %x: three words sequence
    %d: forth word of the sequence
    word_no = max(d);
    sample_no = size(d,1);
    X1 = zeros(word_no,sample_no);
    X2 = zeros(word_no,sample_no);
    X3 = zeros(word_no,sample_no);
    D = zeros(word_no,sample_no);
    for i =1:sample_no
        X1(x(1,i),i) = 1;
        X2(x(2,i),i) = 1;
        X3(x(3,i),i) = 1;
        D(d(i),i) = 1;
    end
end

function  CE = cross_entropy(y,y_hat)
    %Inputs
    %y_hat: the estimate of model
    %y: the desired output of model
    CE = -sum(y.*log(y_hat));
end

function o = softmax(x)
    %Sofmax operation
    o = exp(x)./sum(exp(x));
end

function [o,der] = sigmoid(v,lambda)
    %Unipolar sigmoid activation function
    %Results are on interval of [0,1]
    %Inputs
    %v: inputs, lambda: parameter for sigmoid, T: threshold
    if(lambda > 0)
        o = 1./(1+exp(-lambda*v));
```

```
        der = o.*(1-o);
    else
        o = nan;
        der = nan;
        disp('Lambda should be a positive value!');
    end
end


function history = neural_net_Q1(X_train, y_train, X_test, y_test,
batch_size,...
    lr,epoch_no,class_no,hidden_neuron_no)

%Output layer neuron number
M = class_no;
%Hidden layer neuron number
L = hidden_neuron_no;

%Take sizes
input_size = size(X_train,1);
train_no = size(X_train,2);

%Choose weights and biases initally, small and random to prevent
saturation
std = 0.001;
W_hidden = std*randn(L,input_size);
bias_hidden = std*randn(L,1);
W_output = std*randn(M,L);
bias_output = std*randn(M,1);

delta_We_output = zeros(M,L+1);
delta_We_hidden = zeros(L,input_size+1);

%Batch number with batch size
batch_no = floor(train_no / batch_size); % B corresponds to batch_size
in comments

%Normalize images
X_train = X_train./max(X_train);
X_test = X_test./max(X_test);

%Train set number and test set number
train_no = size(X_train,2);
test_no = size(X_test,2);

for N = 1:epoch_no

    %Shuffle training images (shuffle indices)
    indices = 1:train_no;
    indices = indices(randperm(train_no));

    for j=1:batch_no
```

```
        %Take images for each batch
        X_indices = indices( (j-1)*batch_size+1 : j*batch_size );
        X = X_train(:,X_indices);

        %FORWARD PROPAGATION
        %input matrix with bias terms
        X = [X;1*ones(1,batch_size)]; %size: 1025xB
        %linear activation potential of hidden layer
        U = [W_hidden bias_hidden]*X; %size: LxB = (Lx1025)*(1025xB)
        %output of hidden layer, hidden signal vector
        H = tanh(U); %size: LxB

        %linear activation potential of output layer
        V = [W_output bias_output]*[H;1*ones(1,batch_size)]; % size:
MxB = (M x L+1)*(L+1 x B)
        %output of output layer, output vector
        Y = tanh(V);  %size: MxB

        %BACK PROPAGATION
        %Desired output
        classes = y_train(X_indices);
        D = zeros(class_no,batch_size);
        D(:,classes == 1) = ones(class_no,sum(classes == 1)).*[1;-1];
%car neuron = 1, cat neuron = -1
        D(:,classes == -1) = ones(class_no,sum(classes == -1)).*[-1;1];
%car neuron = -1, cat neuron = 1

        %LOCAL GRADIENTS OF OUTPUT LAYER
        %gradient descent update of output weight matrix
        error_output = D-Y; %size: MxB
        %derivative of y with respect to v
        der_of_Y_with_V = 1-Y.^2; %size: MxB
        %local gradients for the output layer
        delta_output = error_output.*der_of_Y_with_V;%size: MxB

        %LOCAL GRADIENTS OF HIDDEN LAYER
        %derivative of h with respect to u
        der_of_H_with_U = 1-H.^2;  %size: LxB
        %local gradients for the hidden layer
        error_hidden = W_output'*delta_output; %size: LxB = (LxM)*(MxB)
        %local gradient for the hidden layer
        delta_hidden = error_hidden.*der_of_H_with_U; %size: LxB

        %WEIGHT UPDATES
        delta_We_output =  lr*delta_output*[H;-
1*ones(1,batch_size)]'/batch_size; %size: MxL = (MxB)*(BxL)
        delta_We_hidden=  lr*delta_hidden*X'/batch_size; %size: MxL =
(Mx1)*(1xL)

        %UPDATE OUTPUT LAYER WEIGHT MATRIX
        We_output = [W_output bias_output] + delta_We_output; %MxL
```

```
        W_output = We_output(:,1:end-1); bias_output =
We_output(:,end);

        %UPDATE HIDDEN LAYER WEIGHT MATRIX
        We_hidden = [W_hidden bias_hidden] + delta_We_hidden; %size:
Lx1025
        W_hidden = We_hidden(:,1:end-1); bias_hidden =
We_hidden(:,end);

    end

    %ERROR METRICS

    %TRAIN SET
    X = [X_train;1*ones(1,train_no)];
    U = [W_hidden bias_hidden]*X;
    H = tanh(U);
    V = [W_output bias_output]*[H;1*ones(1,train_no)];
    Y = tanh(V);

    %Desired Output
    classes = y_train;
    D = zeros(class_no,train_no);
    D(:,classes == 1) = ones(class_no,sum(classes == 1)).*[1;-1]; %car
neuron = 1, cat neuron = -1
    D(:,classes == -1) = ones(class_no,sum(classes == -1)).*[-1;1];
%car neuron = -1, cat neuron = 1

    %ERROR METRICS FOR ONE EPOCH
    %Mean Squared Error
    MSE = sum(sum(0.5*(D-Y).^2));
    MSE = MSE/train_no;
    %Mean Classification Error
    [~,real_classes] = max(D);
    [~,pred_classes] = max(Y);
    MCE = sum(real_classes ~= pred_classes);
    MCE = MCE/train_no * 100;

    %Record error
    history.train_MSE(N) = MSE;
    history.train_MCE(N) = MCE;

    %TEST SET
    X = [X_test;1*ones(1,test_no)];
    U = [W_hidden bias_hidden]*X;
    H = tanh(U);
    V = [W_output bias_output]*[H;1*ones(1,test_no)];
    Y = tanh(V);

    %Desired Output
    classes = y_test;
    D = zeros(class_no,test_no);
```

```
    D(:,classes == 1) = ones(class_no,sum(classes == 1)).*[1;-1]; %car
neuron = 1, cat neuron = -1
    D(:,classes == -1) = ones(class_no,sum(classes == -1)).*[-1;1];
%car neuron = -1, cat neuron = 1

    %Mean Squared Error
    MSE = sum(sum(0.5*(D-Y).^2))/test_no;
    %Mean Classification Error
    [~,real_classes] = max(D);
    [~,pred_classes] = max(Y);
    MCE = sum(real_classes ~= pred_classes)/test_no * 100;

    %Record error
    history.test_MSE(N) = MSE;
    history.test_MCE(N) = MCE;

end

end

function history = neural_net2_Q1(X_train, y_train, X_test, y_test,
batch_size,...
    lr,alpha,epoch_no,class_no,hidden_neuron_no,hidden_neuron_no2)
%This function is addition of one more hidden layer in neural_net_Q1

M = class_no;
L = hidden_neuron_no2;
K = hidden_neuron_no;

input_size = size(X_train,1);
train_no = size(X_train,2);

std = 0.01;
W_hidden1 = std*randn(K,input_size);
bias_hidden1 = std*randn(K,1);
W_hidden2 = std*randn(L,K);
bias_hidden2 = std*randn(L,1);
W_output = std*randn(M,L);
bias_output = std*randn(M,1);

delta_We_output = zeros(M,L+1);
delta_We_hidden1 = zeros(K,input_size+1);
delta_We_hidden2 = zeros(L,K+1);

batch_no = floor(train_no / batch_size);

X_test = X_test./max(X_test);
X_train = X_train./max(X_train);

test_no = size(X_test,2);
train_no = size(X_train,2);
```

```matlab
for N = 1:epoch_no

    %shuffle training images
    indices = 1:train_no;
    indices = indices(randperm(train_no));

    for j=1:batch_no

        X_indices = indices( (j-1)*batch_size+1 : j*batch_size );
        X = X_train(:,X_indices);

        %FORWARD PROPAGATION
        %Hidden Layer #1
        X = [X;1*ones(1,batch_size)];
        U1 = [W_hidden1 bias_hidden1]*X;
        H1 = tanh(U1);

        %Hidden Layer #2
        U2 = [W_hidden2 bias_hidden2]*[H1;1*ones(1,batch_size)];
        H2 = tanh(U2);

        %Output Layer
        V = [W_output bias_output]*[H2;1*ones(1,batch_size)];
        Y = tanh(V);

        %BACK PROPAGATION

        %Desired Output
        classes = y_train(X_indices);
        D = zeros(class_no,batch_size);
        D(:,classes == 1) = ones(class_no,sum(classes == 1)).*[1;-1];
%car neuron = 1, cat neuron = -1
        D(:,classes == -1) = ones(class_no,sum(classes == -1)).*[-1;1];
%car neuron = -1, cat neuron = 1

        %LOCAL GRADIENT OF OUTPUT LAYER
        error_output = D-Y;
        der_of_Y_with_V = 1-Y.^2;
        delta_output = error_output.*der_of_Y_with_V;

        %LOCAL GRADIENT OF HIDDEN LAYER #2
        der_of_H2_with_U2 = 1-H2.^2;
        error_hidden2 = W_output'*delta_output;
        delta_hidden2 = error_hidden2.*der_of_H2_with_U2;

        %LOCAL GRADIENT OF HIDDEN LAYER #1
        der_of_H1_with_U1 = 1-H1.^2;
        error_hidden1 = W_hidden2'*delta_hidden2;
        delta_hidden1 = error_hidden1.*der_of_H1_with_U1;

        %OUTPUT LAYER WEIGHT UPDATE
```

```
        delta_We_output =  lr*delta_output*[H2;-
1*ones(1,batch_size)]'/batch_size + alpha*delta_We_output;
        delta_We_hidden2 = lr*delta_hidden2*[H1;-
1*ones(1,batch_size)]'/batch_size + alpha*delta_We_hidden2;
        delta_We_hidden1 = lr*delta_hidden1*X'/batch_size +
alpha*delta_We_hidden1;


        %WEIGHT UPDATES
        We_output = [W_output bias_output] + delta_We_output;
        W_output = We_output(:,1:end-1); bias_output =
We_output(:,end);

        We_hidden2 = [W_hidden2 bias_hidden2] + delta_We_hidden2;
        W_hidden2 = We_hidden2(:,1:end-1); bias_hidden2 =
We_hidden2(:,end);

        We_hidden1 = [W_hidden1 bias_hidden1] + delta_We_hidden1;
        W_hidden1 = We_hidden1(:,1:end-1); bias_hidden1 =
We_hidden1(:,end);

    end

    %FORWARD PROPAGATION FOR TRAIN SET
    X = [X_train;1*ones(1,train_no)];
    U1 = [W_hidden1 bias_hidden1]*X;
    H1 = tanh(U1);
    U2 = [W_hidden2 bias_hidden2]*[H1;1*ones(1,train_no)];
    H2 = tanh(U2);
    V = [W_output bias_output]*[H2;1*ones(1,train_no)];
    Y = tanh(V);

    %ERROR METRICS
    %Desired Output
    classes = y_train;
    D = zeros(class_no,train_no);
    D(:,classes == 1) = ones(class_no,sum(classes == 1)).*[1;-1]; %car
neuron = 1, cat neuron = -1
    D(:,classes == -1) = ones(class_no,sum(classes == -1)).*[-1;1];
%car neuron = -1, cat neuron = 1

    %Mean Squared Error
    MSE = sum(sum(0.5*(D-Y).^2))/train_no;

    %Mean Classification Error
    [~,real_classes] = max(D);
    [~,pred_classes] = max(Y);
    MCE = sum(real_classes ~= pred_classes)/train_no * 100;

    history.train_MSE(N) = MSE;
    history.train_MCE(N) = MCE;
```

```
    %FORWARD PROPAGATION FOR TEST SET
    X = [X_test;1*ones(1,test_no)];
    U1 = [W_hidden1 bias_hidden1]*X;
    H1 = tanh(U1);
    U2 = [W_hidden2 bias_hidden2]*[H1;1*ones(1,test_no)];
    H2 = tanh(U2);
    V = [W_output bias_output]*[H2;1*ones(1,test_no)];
    Y = tanh(V);

    %ERROR METRICS
    %Desired Output
    classes = y_test;
    D = zeros(class_no,test_no);
    D(:,classes == 1) = ones(class_no,sum(classes == 1)).*[1;-1]; %car
neuron = 1, cat neuron = -1
    D(:,classes == -1) = ones(class_no,sum(classes == -1)).*[-1;1];
%car neuron = -1, cat neuron = 1

    [~,real_classes] = max(D);
    [~,pred_classes] = max(Y);

    MSE = sum(sum(0.5*(D-Y).^2))/test_no;
    MCE = sum(real_classes ~= pred_classes)/test_no * 100;

    history.test_MSE(N) = MSE;
    history.test_MCE(N) = MCE;

end

end

function [history,weights] = neural_net_Q2
(trainx,traind,valx,vald,testx,testd,epoch_no,batch_size,lr,alpha,D,P)

    %Take number of sets
    train_no = size(traind,1);
    val_no = size(vald,1);
    word_no = max(traind);

    %Vectorize words
    [X1_train,X2_train,X3_train,D_out_train] =
prepare_words(trainx,traind);
    [X1_val,X2_val,X3_val,D_out_val] = prepare_words(valx,vald);

    %Initialize error metrics for train, validation and test sets
    history.train_CE = [];
    history.train_MCE = [];
    history.val_CE = [];
    history.val_MCE = [];
    history.test_CE = [];
    history.test_MCE = [];
```

```matlab
batch_no = floor(train_no/batch_size);

%EMBEDDING WORDS
std  = 0.01;
WE = std*randn(D,word_no); %embedding matrix
W1 = std*randn(P,3*D + 1); %weight matrix of hidden layer #1
W2 = std*randn(word_no,P+1); %weight matrix of output layer

lambda = 1; %parameter of sigmoid activation function
delta_W2 = 0;
delta_W1 = 0;
delta_WE1 = 0;
delta_WE2 = 0;
delta_WE3 = 0;

for i=1:epoch_no

    %shuffle images
    indices = 1:train_no;
    indices = indices(randperm(train_no));

    CE = 0;
    false_pred_no = 0;

    for m = 1:batch_no

        X_indices = indices((m-1)*batch_size+1:m*batch_size);
        X1 = X1_train(:,X_indices);
        X2 = X2_train(:,X_indices);
        X3 = X3_train(:,X_indices);
        D_out = D_out_train(:,X_indices);

        %EMBEDDING WORDS
        E1 = WE*X1;
        E2 = WE*X2;
        E3 = WE*X3;

        %CONCATENATED EMBEDDED WORDS
        E = [E1;E2;E3;ones(1,batch_size)]; %with bias term

        %HIDDEN LAYER #1
        V1 = W1*E;
        [Y1,der_Y1] = sigmoid(V1,lambda);

        %HIDDEN LAYER #2
        Y1 = [Y1;ones(1,batch_size)]; %with bias term
        V2 = W2*Y1;
        Y2 = softmax(V2);

        %derivative of cross entropy
        der_CE = -(Y2-D_out);
```

```
            %BACK PROPAGATION
            %OUTPUT LAYER
            delta_output = der_CE;

            %HIDDEN LAYER
            error_hidden = W2(:,1:end-1)'*delta_output;
            delta_hidden = error_hidden.*der_Y1;

            %EMBEDDING MATRIX
            delta_embed = W1(:,1:end-1)'*delta_hidden;
            delta_embed1 = delta_embed(1:D,:);
            delta_embed2 = delta_embed(D+1:2*D,:);
            delta_embed3 = delta_embed(2*D+1:end,:);

            %UPDATE WEIGHTS
            delta_W2 = lr*delta_output*Y1'/batch_size + alpha*delta_W2;
            delta_W1 = lr*delta_hidden*E'/batch_size +  alpha*delta_W1;
            delta_WE1 = lr*delta_embed1*X1'/batch_size +
alpha*delta_WE1;
            delta_WE2 = lr*delta_embed2*X2'/batch_size +
alpha*delta_WE2;
            delta_WE3 = lr*delta_embed3*X3'/batch_size +
alpha*delta_WE3;
            delta_WE = (delta_WE1 + delta_WE2 + delta_WE3)/3;

            %ERROR METRICS
            CE = CE + cross_entropy(D_out,Y2);%y = d, y_hat = y2
            [~,real_words] = max(D_out);
            [~,pred_words] = max(Y2);
            false_pred_no = false_pred_no + sum(real_words ~=
pred_words);

            %UPDATE WEIGHTS
            W2 = W2 + delta_W2;
            W1 = W1 + delta_W1;
            WE = WE + delta_WE;

        end

        CE = sum(CE)/(batch_no * batch_size);
        MCE = false_pred_no/(batch_no * batch_size);
        history.train_CE = [history.train_CE CE];
        history.train_MCE = [history.train_MCE MCE];

        %VALIDATION SET
        E1 = WE*X1_val;
        E2 = WE*X2_val;
        E3 = WE*X3_val;
        E = [E1;E2;E3;ones(1,val_no)]; %bias term
        V1 = W1*E;
        [Y1,~] = sigmoid(V1,lambda);
        Y1 = [Y1;ones(1,val_no)]; %bias term
```

```
        V2 = W2*Y1;
        Y2 = softmax(V2);
        [~,real_words] = max(D_out_val);
        [~,pred_words] = max(Y2);

        %ERROR METRICS
        false_pred_no = sum(real_words ~= pred_words);
        CE = cross_entropy(D_out_val,Y2);%y = d, y_hat = y2
        CE = sum(CE)/val_no;
        MCE = false_pred_no/val_no;
        history.val_CE = [history.val_CE CE];
        history.val_MCE = [history.val_MCE MCE];

        if(history.train_CE(end) <= 3)
            break;
        end

    end

    history.epoch = i;

    weights.WE = WE;
    weights.W1 = W1;
    weights.W2 = W2;

end
```