Assignment 1: Neural Networks

Implement your code and answer all the questions. Once you complete the assignment and answer the questions inline, you can download the report in pdf (File->Download as->PDF) and send it to us, together with the code.

Don't submit additional cells in the notebook, we will not check them. Don't change parameters of the learning inside the cells.

Assignment 1 consists of 4 sections:

- **Section 1**: Data Preparation
- **Section 2**: Multinomial Logistic Regression
- **Section 3**: Backpropagation
- **Section 4**: Neural Networks

In [1]:
```
# Import necessary standard python packages
import numpy as np
import matplotlib.pyplot as plt

# Setting configuration for matplotlib
%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0)
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'
plt.rcParams['xtick.labelsize'] = 15
plt.rcParams['ytick.labelsize'] = 15
plt.rcParams['axes.labelsize'] = 20
```

In [2]:
```
# Import python modules for this assignment

from uva_code.cifar10_utils import get_cifar10_raw_data, preprocess_ci
from uva_code.solver import Solver
from uva_code.losses import SoftMaxLoss, CrossEntropyLoss, HingeLoss
from uva_code.layers import LinearLayer, ReLULayer, SigmoidLayer, Tanhl
from uva_code.models import Network
from uva_code.optimizers import SGD

%load_ext autoreload
%autoreload 2
```

Section 1: Data Preparation

In this section you will download CIFAR10 (https://www.cs.toronto.edu/~kriz/cifar.html) data which you will use in this assignment.

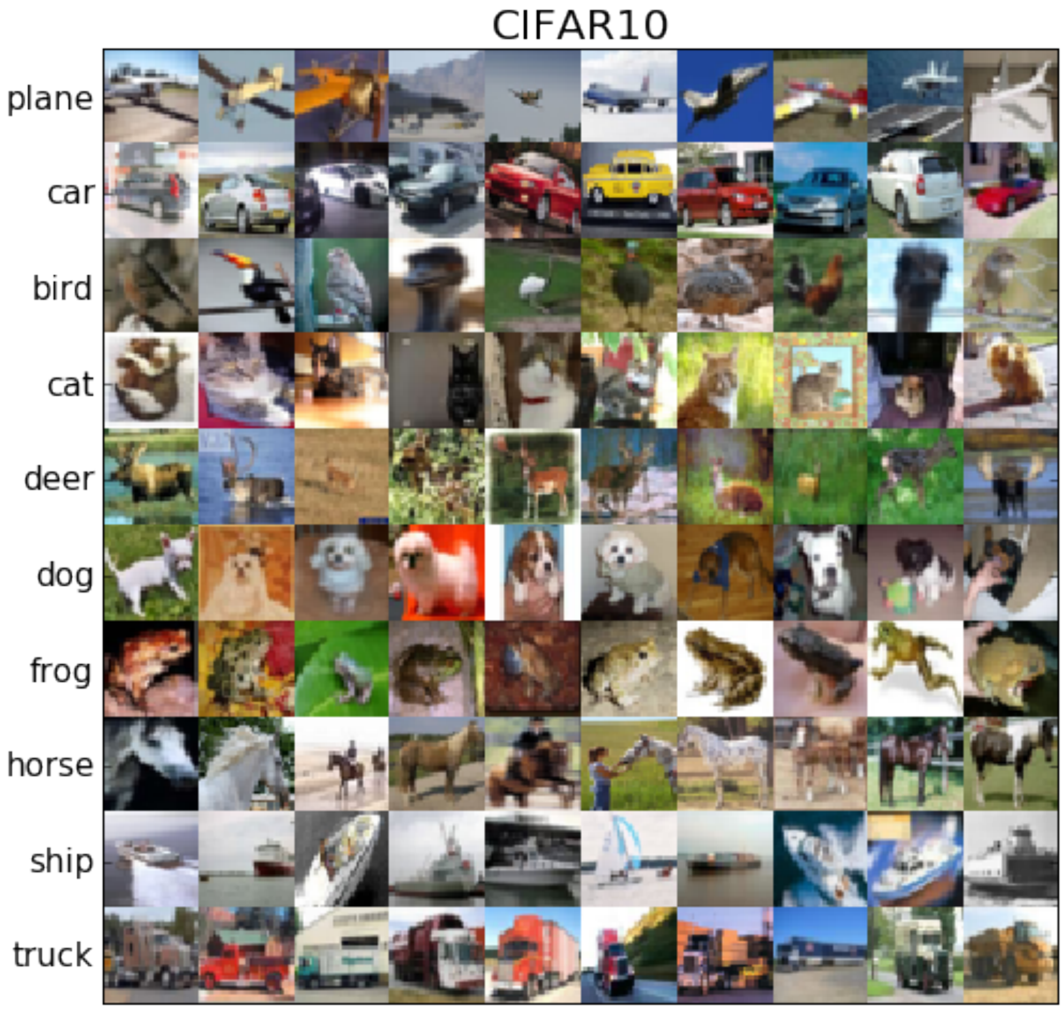
Make sure that everything has been downloaded correctly and all images are visible.

In [3]:
```
# Get raw CIFAR10 data. For Unix users the script to download CIFAR10 
# it is used inside get_cifar10_raw_data() function. If it doesn't wor
# http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz and extract i
# cifar10 folder.

# Downloading the data can take several minutes.
X_train_raw, Y_train_raw, X_test_raw, Y_test_raw = get_cifar10_raw_dat

#Checking shapes, should be (50000, 32, 32, 3), (50000, ), (10000, 32,
print 'Train data shape: ', X_train_raw.shape
print 'Train labels shape: ', Y_train_raw.shape
print 'Test data shape: ', X_test_raw.shape
print 'Test labels shape: ', Y_test_raw.shape
```

```
Train data shape:  (50000, 32, 32, 3)
Train labels shape:  (50000,)
Test data shape:  (10000, 32, 32, 3)
Test labels shape:  (10000,)
```

In [4]:

```
# Visualize CIFAR10 data
samples_per_class = 10
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse

num_classes = len(classes)
can = np.zeros((320, 320, 3),dtype='uint8')
for i, cls in enumerate(classes):
    idxs = np.flatnonzero(Y_train_raw == i)
    idxs = np.random.choice(idxs, samples_per_class, replace = False)
    for j in range(samples_per_class):
        can[32 * i:32 * (i + 1), 32 * j:32 * (j + 1),:] = X_train_raw[
plt.xticks([], [])
plt.yticks(range(16, 320, 32), classes)
plt.title('CIFAR10', fontsize = 20)
plt.imshow(can)
plt.show()
```



In [5]:
```
# Normalize CIFAR10 data by subtracting the mean image. With these dat
# The validation subset will be used for tuning the hyperparameters.
X_train, Y_train, X_val, Y_val, X_test, Y_test = preprocess_cifar10_da


#Checking shapes, should be (49000, 3072), (49000, ), (1000, 3072), (1
print 'Train data shape: ', X_train.shape
print 'Train labels shape: ', Y_train.shape
print 'Val data shape: ', X_val.shape
print 'Val labels shape: ', Y_val.shape
print 'Test data shape: ', X_test.shape
print 'Test labels shape: ', Y_test.shape
```

```
Train data shape:  (49000, 3072)
Train labels shape:  (49000,)
Val data shape:  (1000, 3072)
Val labels shape:  (1000,)
Test data shape:  (10000, 3072)
Test labels shape:  (10000,)
```

Data Preparation: Question 1 [4 points]

Neural networks and deep learning methods prefer the input variables to contain as raw data as possible. But in the vast majority of cases data need to be preprocessed. Suppose, you have two types of non-linear activation functions (Sigmoid (https://en.wikipedia.org/wiki/Sigmoid_function), ReLU (https://en.wikipedia.org/wiki/Rectifier_(neural_networks))) and two types of normalization (Per-example mean substraction (http://ufldl.stanford.edu/wiki/index.php/Data_Preprocessing#Per-example_mean_subtraction), Standardization (http://ufldl.stanford.edu/wiki/index.php/Data_Preprocessing#Feature_Standardization)). Which one should you use for each case and why? For example, in the previous cell we used per-example mean substraction.

Your Answer:

Sigmoid: Standardization

Given that for gradients of sigmoids are very small outside the interval of around $[-2, 2]$, it is important to move and rescale your data so it lies in that range.

ReLU: Both is possible

ReLUs don't necessary require rescaling, but centering could still be useful. So standardization may be applied. Use of per-example mean substraction seems to be motivated more by the type of problem one is dealing with, than by the choice of activation, but would certainly be possible here, given that we are working on an image data-set.

Section 2: Multinomial Logistic Regression [5 points]

In this section you will get started by implementing a linear classification model called [Multinomial Logistic Regression (http://ufldl.stanford.edu/tutorial/supervised/SoftmaxRegression/)](http://ufldl.stanford.edu/tutorial/supervised/SoftmaxRegression/). Later on you will extend this model to a neural network. You will train it by using the [mini-batch Stochastic Gradient Descent algorithm (http://sebastianruder.com/optimizing-gradient-descent/index.html#minibatchgradientdescent)](http://sebastianruder.com/optimizing-gradient-descent/index.html#minibatchgradientdescent). You should implement how to sample batches, how to compute the loss, how to compute the gradient of the loss with respect to the parameters of the model and how to update the parameters of the model.

You should get around 0.35 accuracy on the validation and test sets with the provided parameters.

In [6]:

```
# DONT CHANGE THE SEED AND THE DEFAULT PARAMETERS. OTHERWISE WE WILL N
# Seed
np.random.seed(42)

# Default parameters.
num_iterations = 1500 # 1500
val_iteration = 100 # 100
batch_size = 200
learning_rate = 1e-7
weight_decay = 3e+4
weight_scale = 0.0001

########################################################################
# TODO:
# Initialize the weights W using a normal distribution with mean = 0 a
# weight_scale. Initialize the biases b with 0.
########################################################################
W = np.random.normal(size=(X_train.shape[1],num_classes), loc=0, scale
b = np.zeros(num_classes)
########################################################################
#                               END OF YOUR CODE
########################################################################

train_loss_history = []
train_acc_history = []

val_loss_history = []
val_acc_history = []

for iteration in range(num_iterations):
    ####################################################################
    # TODO:
    # Sample a random mini-batch with the size of batch_size from the
    # images to X_train_batch and labels to Y_train_batch variables.
    ####################################################################
    ids = np.random.choice(X_train.shape[0], size=batch_size, replace=
    X_train_batch = X_train[ids, :]
    Y_train_batch = Y_train[ids]
    ####################################################################
    #                               END OF YOUR CODE
    ####################################################################

    ####################################################################
```

```
# TODO:
# Compute the loss and the accuracy of the multinomial logistic re
# on X_train_batch, Y_train_batch.
######################################################################
xwb = np.dot(X_train_batch, W) + b
xexp =np.exp(xwb)
sum_exp = np.sum(xexp, 1)
log_sum_exp = np.log(sum_exp)
xwb_targets = np.asarray([xwb[k,Y_train_batch[k]] for k in range(b
log_softmax_targets = xwb_targets - log_sum_exp

train_loss = - np.mean(log_softmax_targets)

guesses = np.argmax(xwb, 1)
matches = guesses == Y_train_batch
train_acc = np.sum(matches) / float(batch_size)
######################################################################
#                               END OF YOUR CODE
######################################################################

######################################################################
# TODO:
# Compute the gradients of the loss with the respect to the weight
# them in dW and db variables.
######################################################################
sum_exp_stack = np.stack([sum_exp for k in range(num_classes)], ax
softmax = xexp / sum_exp_stack
dW = np.dot(X_train_batch.T, softmax)
db = np.sum(softmax, 0)
for idx, val in enumerate(Y_train_batch):
    dW[:, val] -= X_train_batch[idx, :]
    db[val] -= 1.0
######################################################################
#                               END OF YOUR CODE
######################################################################

######################################################################
# TODO:
# Update the weights W and biases b using the Stochastic Gradient
######################################################################
# weight decay
W *= (1 - weight_decay * learning_rate)

# gradient update
W -= (learning_rate / float(batch_size)) * dW
b -= (learning_rate / float(batch_size)) * db
######################################################################
#                               END OF YOUR CODE
######################################################################

if iteration % val_iteration == 0 or iteration == num_iterations -
    ##################################################################
    # TODO:
    # Compute the loss and the accuracy on the validation set.
    ##################################################################
    xwb = np.dot(X_val, W) + b  # (1000, 3072) * (3072, 10) + (<10
    xexp =np.exp(xwb)
    sum_exp = np.sum(xexp, 1)  # (1000,)
```

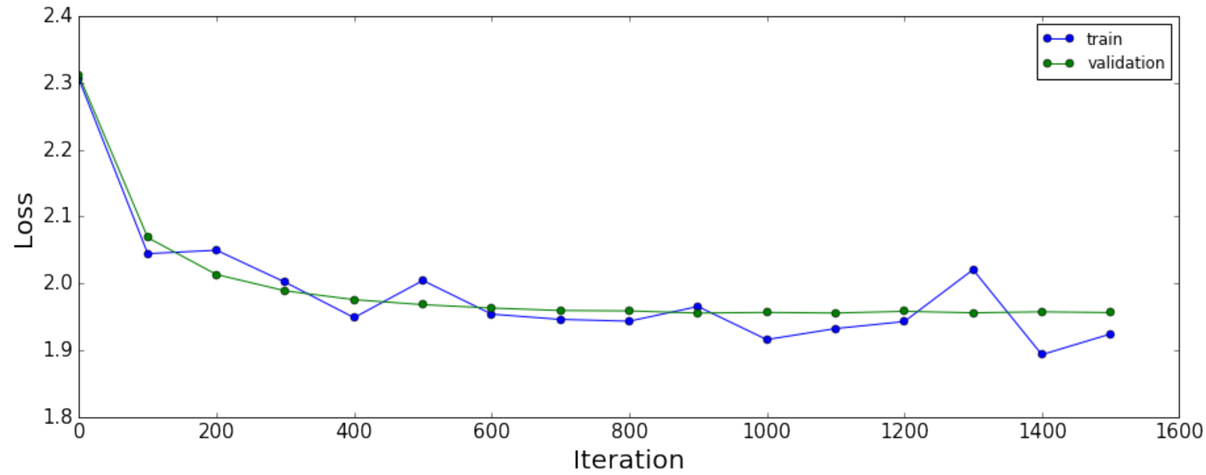
```
        log_sum_exp = np.log(sum_exp)
        xwb_targets = np.asarray([xwb[k,Y_val[k]] for k in range(Y_val
        log_softmax_targets = xwb_targets - log_sum_exp

        val_loss = - np.mean(log_softmax_targets)
        guesses = np.argmax(xwb, 1)  # (1000,)
        matches = guesses == Y_val
        val_acc = np.sum(matches) / float(Y_val.shape[0])
        ###################################################################
        #                              END OF YOUR CODE
        ###################################################################
        train_loss_history.append(train_loss)
        train_acc_history.append(train_acc)
        val_loss_history.append(val_loss)
        val_acc_history.append(val_acc)

        # Output loss and accuracy during training
        print("Iteration {0:d}/{1:d}. Train Loss = {2:.3f}, Train Accu
              format(iteration, num_iterations, train_loss, train_acc)
        print("Iteration {0:d}/{1:d}. Validation Loss = {2:.3f}, Valid
              format(iteration, num_iterations, val_loss, val_acc))
######################################################################
# TODO:
# Compute the accuracy on the test set.
######################################################################
xwb = np.dot(X_test, W) + b
xexp =np.exp(xwb)
sum_exp = np.sum(xexp, 1)
log_sum_exp = np.log(sum_exp)
xwb_targets = np.asarray([xwb[k,Y_test[k]] for k in range(Y_test.shape
log_softmax_targets = xwb_targets - log_sum_exp

val_loss = - np.mean(log_softmax_targets)
guesses = np.argmax(xwb, 1)
matches = guesses == Y_test
test_acc = np.sum(matches) / float(Y_test.shape[0])
######################################################################
#                              END OF YOUR CODE
######################################################################
print("Test Accuracy = {0:.3f}".format(test_acc))
```

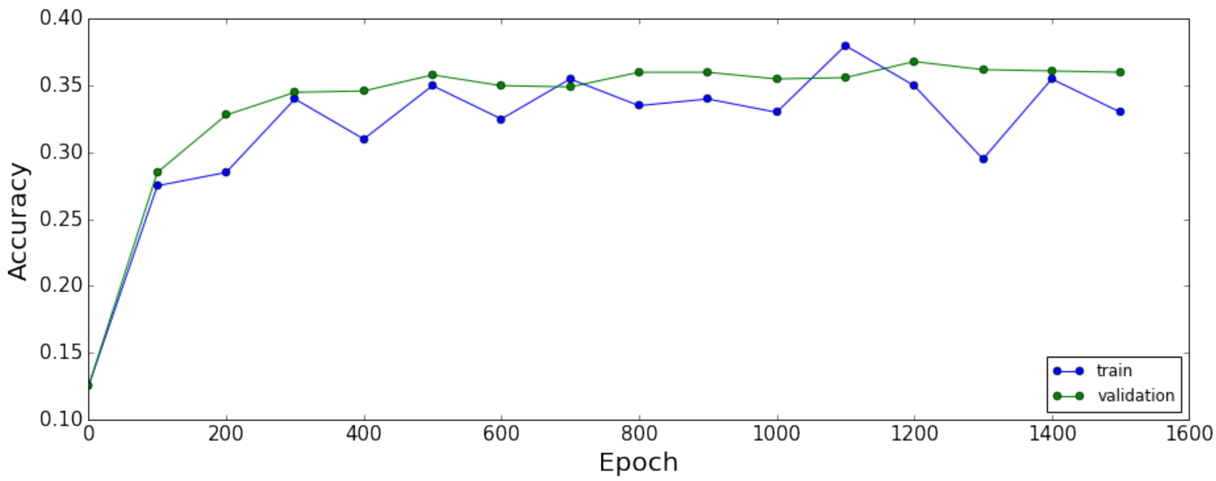
```
Iteration 0/1500. Train Loss = 2.305, Train Accuracy = 0.125
Iteration 0/1500. Validation Loss = 2.311, Validation Accuracy = 0.1
25
Iteration 100/1500. Train Loss = 2.044, Train Accuracy = 0.275
Iteration 100/1500. Validation Loss = 2.068, Validation Accuracy = 0
.285
Iteration 200/1500. Train Loss = 2.049, Train Accuracy = 0.285
Iteration 200/1500. Validation Loss = 2.013, Validation Accuracy = 0
.328
Iteration 300/1500. Train Loss = 2.001, Train Accuracy = 0.340
Iteration 300/1500. Validation Loss = 1.989, Validation Accuracy = 0
.345
Iteration 400/1500. Train Loss = 1.948, Train Accuracy = 0.310
Iteration 400/1500. Validation Loss = 1.975, Validation Accuracy = 0
.346
Iteration 500/1500. Train Loss = 2.004, Train Accuracy = 0.350
```

```
Iteration 500/1500. Validation Loss = 1.968, Validation Accuracy = 0
.358
Iteration 600/1500. Train Loss = 1.954, Train Accuracy = 0.325
Iteration 600/1500. Validation Loss = 1.963, Validation Accuracy = 0
.350
Iteration 700/1500. Train Loss = 1.946, Train Accuracy = 0.355
Iteration 700/1500. Validation Loss = 1.959, Validation Accuracy = 0
.349
Iteration 800/1500. Train Loss = 1.943, Train Accuracy = 0.335
Iteration 800/1500. Validation Loss = 1.959, Validation Accuracy = 0
.360
Iteration 900/1500. Train Loss = 1.965, Train Accuracy = 0.340
Iteration 900/1500. Validation Loss = 1.955, Validation Accuracy = 0
.360
Iteration 1000/1500. Train Loss = 1.916, Train Accuracy = 0.330
Iteration 1000/1500. Validation Loss = 1.956, Validation Accuracy =
0.355
Iteration 1100/1500. Train Loss = 1.932, Train Accuracy = 0.380
Iteration 1100/1500. Validation Loss = 1.955, Validation Accuracy =
0.356
Iteration 1200/1500. Train Loss = 1.943, Train Accuracy = 0.350
Iteration 1200/1500. Validation Loss = 1.958, Validation Accuracy =
0.368
Iteration 1300/1500. Train Loss = 2.020, Train Accuracy = 0.295
Iteration 1300/1500. Validation Loss = 1.956, Validation Accuracy =
0.362
Iteration 1400/1500. Train Loss = 1.893, Train Accuracy = 0.355
Iteration 1400/1500. Validation Loss = 1.957, Validation Accuracy =
0.361
Iteration 1499/1500. Train Loss = 1.924, Train Accuracy = 0.330
Iteration 1499/1500. Validation Loss = 1.956, Validation Accuracy =
0.360
Test Accuracy = 0.344
```

In [7]:

```
# Visualize a learning curve of multinomial logistic regression classi
plt.subplot(2, 1, 1)
plt.plot(range(0, num_iterations + 1, val_iteration), train_loss_histo
plt.plot(range(0, num_iterations + 1, val_iteration), val_loss_history
plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.legend(loc='upper right')

plt.subplot(2, 1, 2)
plt.plot(range(0, num_iterations + 1, val_iteration), train_acc_histor
plt.plot(range(0, num_iterations + 1, val_iteration), val_acc_history,
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.gcf().set_size_inches(15, 12)
plt.show()
```





Multinomial Logistic Regression: Question 1 [4 points]

What is the value of the loss and the accuracy you expect to obtain at iteration = 0 and why? Consider weight_decay = 0.

Your Answer:

Given 10 classes, each class should get around 10% probability, the log of which is ~2.3. Accuracy should also be at chance level, i.e. 10%.

Multinomial Logistic Regression: Question 2 [4 points]

Name at least three factors that determine the size of batches in practice and briefly motivate your answers. The factors might be related to computational or performance aspects.

Your Answer:

- Batches (and the intermediate results they generate) should fit into **working memory**.
- With growing **number of feature dimensions**, the memory taken up by each sample increases, and batch sizes may have to be chosen smaller as a result.
- In case of a **dynamic data-set**, the speed at which new samples become available may affect the batch size
- As discussed below, the **error surface** of the learning problem may decide, whether smaller or larger batch sizes lead to better performance. This factor is of course not generally known a priori and must be discovered in testing.

Mulinomial Logistic Regression: Question 3 [4 points]

Does the learning rate depend on the batch size? Explain how you should change the learning rate with respect to changes of the batch size.

Name two extreme choices of a batch size and explain their advantages and disadvantages.

Your Answer:

Larger batch-sizes increase confidence in the gradient correctly reflecting the error-surface of the data-set and also reduces the frequency of updates (as gradients from more samples must be computed). Both factors encourage a larger learning rate.

Extremes:

Batch-GD, which computes gradients based on the complete training-set, is of course the most confident version, and larger learning rate can pay off in (nearly) flat regions. In practice, however, this is often near-impossible - or desirable - to implement, as use-case data-sets have grown too large.

Stochastic-GD computes gradients based on single samples, which gives it an effect close to simulated annealing, which helps it escape local opima (which batch-GD cannot). On the other hand, gradient directions can fluctuate a lot, and given that learning rate should be low, this can lead to slow learning But this can be helped to some extent with momentum methods, or, of course, by using mini.batches

Multinomial Logistic Regression: Question 4 [4 points]

How can you describe the rows of weight matrix W? What are they representing? Why?

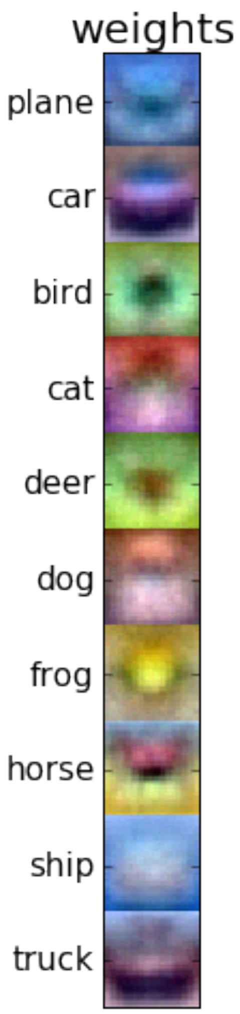
Your Answer:

each row is associated with one pixel in one color channel, indicating how strongly activation of this pixel is correlated with each of the 10 classes. We can see this below in how, for example, planes and ships have strong weights in the blue channel for sky and water, whereas horses have weights which form a brown blob in the center of the image.

Hint: Before answering the question visualize rows of weight matrix W in the cell below.

In [9]:

```
################################################################################
# TODO:
# Visualize the learned weights for each class.
################################################################################
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse
can = np.zeros((320, 32, 3),dtype='uint8')
for idx, cls in enumerate(classes):
    im = W[:, idx].reshape((32, 32, 3))
    im =(im - np.min(im))
    im = im * (255. / np.max(im))
    can[32 * idx:32 * (idx + 1), :, :] = im

plt.xticks([], [])
plt.yticks(range(16, 320, 32), classes)
plt.title('weights', fontsize = 20)
plt.imshow(can)
plt.show()
################################################################################
#                              END OF YOUR CODE
################################################################################
```



Section 3: Backpropagation

Follow the instructions and solve the tasks in paper_assignment_1.pdf. Write your solutions in a separate pdf file. You don't need to put anything here.

Section 4: Neural Networks [10 points]

A modular implementation of neural networks allows to define deeper and more flexible architectures. In this section you will implement the multinomial logistic regression classifier from the Section 2 as a one-layer neural network that consists of two parts: a linear transformation layer (module 1) and a softmax loss layer (module 2).

You will implement the multinomial logistic regression classifier as a modular network by following next steps:

1. Implement the forward and backward passes for the linear layer in **layers.py** file. Write your code inside the ***forward*** and ***backward*** methods of ***LinearLayer*** class. Compute the regularization loss of the weights inside the ***layer_loss*** method of ***LinearLayer*** class.
2. Implement the softmax loss computation in **losses.py** file. Write your code inside the ***SoftMaxLoss*** function.
3. Implement the ***forward***, ***backward*** and ***loss*** methods for the ***Network*** class inside the **models.py** file.
4. Implement the SGD update rule inside ***SGD*** class in **optimizers.py** file.
5. Implement the ***train_on_batch***, ***test_on_batch***, ***fit***, ***predcit***, ***score***, ***accuracy*** methods of ***Solver*** class in ***solver.py*** file.

You should get the same results for the next cell as in Section 2. **Don't change the parameters**.

In [55]:
```
# DONT CHANGE THE SEED AND THE DEFAULT PARAMETERS. OTHERWISE WE WILL N
# Seed
np.random.seed(42)

# Default parameters.
num_iterations = 1500 # 1500
val_iteration = 100
batch_size = 200
learning_rate = 1e-7
weight_decay = 3e+4
weight_scale = 0.0001

########################################################################
# TODO:
# Build the multinomial logistic regression classifier using the Netwo
# will need to use add_layer and add_loss methods. Train this model us
# with SGD optimizer. In configuration of the optimizer you need to sp
# learning rate. Use the fit method to train classifier. Don't forget 
# X_val and Y_val in arguments to output the validation loss and accur
# training. Set the verbose to True to compare with the  multinomial l
# regression classifier from the Section 2.
########################################################################
model = Network()
lin_layer_params = {'input_size':X_train.shape[1],
                    'output_size':num_classes,
                    'weight_decay':weight_decay,
                    'weight_scale':weight_scale }
model.add_layer(LinearLayer(lin_layer_params))
```

```
model.add_layer(LinearLayer(lin_layer_params))
model.add_loss(SoftMaxLoss)
optimizer = SGD()
optimizer_config = {'learning_rate': learning_rate}
solver = Solver(model)
solver.fit(X_train, Y_train, optimizer, optimizer_config, X_val, Y_val
           batch_size, num_iterations, val_iteration, verbose = False)
##########################################################################
#                              END OF YOUR CODE
##########################################################################

##########################################################################
# TODO:
# Compute the accuracy on the test set.
##########################################################################
test_acc = solver.score(X_test, Y_test)
##########################################################################
#                              END OF YOUR CODE
##########################################################################
print("Test Accuracy = {0:.3f}".format(test_acc))
```

```
Iteration 0/1500: Train Loss = 2.305, Train Accuracy = 0.125
Iteration 0/1500. Validation Loss = 2.311, Validation Accuracy = 0.1
25
Iteration 100/1500: Train Loss = 2.044, Train Accuracy = 0.275
Iteration 100/1500. Validation Loss = 2.068, Validation Accuracy = 0
.285
Iteration 200/1500: Train Loss = 2.049, Train Accuracy = 0.285
Iteration 200/1500. Validation Loss = 2.013, Validation Accuracy = 0
.328
Iteration 300/1500: Train Loss = 2.001, Train Accuracy = 0.340
Iteration 300/1500. Validation Loss = 1.989, Validation Accuracy = 0
.345
Iteration 400/1500: Train Loss = 1.948, Train Accuracy = 0.310
Iteration 400/1500. Validation Loss = 1.975, Validation Accuracy = 0
.346
Iteration 500/1500: Train Loss = 2.004, Train Accuracy = 0.350
Iteration 500/1500. Validation Loss = 1.968, Validation Accuracy = 0
.358
Iteration 600/1500: Train Loss = 1.954, Train Accuracy = 0.325
Iteration 600/1500. Validation Loss = 1.963, Validation Accuracy = 0
.350
Iteration 700/1500: Train Loss = 1.946, Train Accuracy = 0.355
Iteration 700/1500. Validation Loss = 1.959, Validation Accuracy = 0
.349
Iteration 800/1500: Train Loss = 1.943, Train Accuracy = 0.335
Iteration 800/1500. Validation Loss = 1.959, Validation Accuracy = 0
.360
Iteration 900/1500: Train Loss = 1.965, Train Accuracy = 0.340
Iteration 900/1500. Validation Loss = 1.955, Validation Accuracy = 0
.360
Iteration 1000/1500: Train Loss = 1.916, Train Accuracy = 0.330
Iteration 1000/1500. Validation Loss = 1.956, Validation Accuracy =
0.355
Iteration 1100/1500: Train Loss = 1.932, Train Accuracy = 0.380
Iteration 1100/1500. Validation Loss = 1.955, Validation Accuracy =
0.356
Iteration 1200/1500: Train Loss = 1.943, Train Accuracy = 0.350
```

```
Iteration 1200/1500. Validation Loss = 1.958, Validation Accuracy =
0.368
Iteration 1300/1500: Train Loss = 2.020, Train Accuracy = 0.295
Iteration 1300/1500. Validation Loss = 1.956, Validation Accuracy =
0.362
Iteration 1400/1500: Train Loss = 1.893, Train Accuracy = 0.355
Iteration 1400/1500. Validation Loss = 1.957, Validation Accuracy =
0.361
Iteration 1499/1500: Train Loss = 1.924, Train Accuracy = 0.330
Iteration 1499/1500. Validation Loss = 1.956, Validation Accuracy =
0.360
Test Accuracy = 0.344
```

Neural Networks: Task 1 [5 points]

Tuning hyperparameters is very important even for multinomial logistic regression.

What are the best learning rate and weight decay? What is test accuracy of the model trained with the best hyperparameters values?

Your Answer: Put your answer here.

Hint: You should be able to get the test accuracy around 0.4.

Implement the tuning of hyperparameters (learning rate and weight decay) in the next cell.

In [62]:

```
# DONT CHANGE THE SEED AND THE DEFAULT PARAMETERS. OTHERWISE WE WILL N
# Seed
np.random.seed(42)

# Default parameters.
num_iterations = 1500
val_iteration = 100
batch_size = 200
weight_scale = 0.0001

# You should try diffierent range of hyperparameters.
learning_rates = [1e-6, 1e-7]
weight_decays = [3e+01, 3e+03] # other parameters were tested. the bes

best_val_acc = -1
best_solver = None
for learning_rate in learning_rates:
    for weight_decay in weight_decays:
        ####################################################################
        # TODO:
        # Implement the tuning of hyperparameters for the multinomial 
        # maximum of the validation accuracy in best_val_acc and corre
        # best_solver variables. Store the maximum of the validation s
        # setting of the hyperparameters in cur_val_acc variable.
        ####################################################################
        model = Network()
        lin_layer_params = {'input_size':X_train.shape[1],
                            'output_size':num_classes,
                            'weight_decay':weight_decay,
```

```
                                weight_scale':weight_scale }
            model.add_layer(LinearLayer(lin_layer_params))
            model.add_loss(SoftMaxLoss)
            optimizer = SGD()
            optimizer_config = {'learning_rate': learning_rate}
            solver = Solver(model)
            res = solver.fit(X_train, Y_train, optimizer, optimizer_config
                             batch_size, num_iterations, val_iteration, ve


            cur_val_acc = max(res[3])
            if cur_val_acc > best_val_acc:
                best_val_acc = cur_val_acc
                best_solver = solver
            ####################################################################
            #                              END OF YOUR CODE
            ####################################################################
            print("Learning rate = {0:e}, weight decay = {1:e}: Validation
                learning_rate, weight_decay, cur_val_acc))

########################################################################
# TODO:
# Compute the accuracy on the test set for the best solver.
########################################################################
test_acc = solver.score(X_test, Y_test)
########################################################################
#                              END OF YOUR CODE
########################################################################
print("Best Test Accuracy = {0:.3f}".format(test_acc))
```

```
Learning rate = 1.000000e-06, weight decay = 3.000000e+01: Validatio
n Accuracy = 0.414
Learning rate = 1.000000e-06, weight decay = 3.000000e+03: Validatio
n Accuracy = 0.404
Learning rate = 1.000000e-07, weight decay = 3.000000e+01: Validatio
n Accuracy = 0.388
Learning rate = 1.000000e-07, weight decay = 3.000000e+03: Validatio
n Accuracy = 0.398
Best Test Accuracy = 0.377
```

Neural Networks: Task 2 [5 points]

Implement a two-layer neural network with a ReLU activation function. Write your code for the ***forward*** and ***backward*** methods of ***ReLULayer*** class in **layers.py** file.

Train the network with the following structure: linear_layer-relu-linear_layer-softmax_loss. You should get the accuracy on the test set around 0.44.

In [18]:
```
# DONT CHANGE THE SEED AND THE DEFAULT PARAMETERS. OTHERWISE WE WILL NO
# Seed
np.random.seed(42)

# Number of hidden units in a hidden layer.
num_hidden_units = 100

# Default parameters.
```

```
num_iterations = 1500 # 500: test set acc of 0.446
val_iteration = 100
batch_size = 200
learning_rate = 2e-3
weight_decay = 0
weight_scale = 0.0001

##########################################################################
# TODO:
# Build the model with the structure: linear_layer-relu-linear_layer-s
# Train this model using Solver class with SGD optimizer. In configura
# optimizer you need to specify only the learning rate. Use the fit me
##########################################################################


model = Network()
lin1_params = {'input_size':X_train.shape[1],
               'output_size':num_hidden_units,
               'weight_decay':weight_decay,
               'weight_scale':weight_scale }
lin2_params = {'input_size':num_hidden_units,
               'output_size':num_classes,
               'weight_decay':weight_decay,
               'weight_scale':weight_scale }
model.add_layer(LinearLayer(lin1_params))
model.add_layer(ReLULayer())
model.add_layer(LinearLayer(lin2_params))
model.add_loss(SoftMaxLoss)
optimizer = SGD()
optimizer_config = {'learning_rate': learning_rate}
solver = Solver(model)
res = solver.fit(X_train, Y_train, optimizer, optimizer_config, X_val,
                 batch_size, num_iterations, val_iteration, verbose = 

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

##########################################################################
# TODO:
# Compute the accuracy on the test set.
##########################################################################
test_acc = solver.score(X_test, Y_test)
##########################################################################
#                              END OF YOUR CODE
##########################################################################
print("Test Accuracy = {0:.3f}".format(test_acc))
```

```
Iteration 0/1500: Train Loss = 2.306, Train Accuracy = 0.105
Iteration 0/1500. Validation Loss = 2.306, Validation Accuracy = 0.1
98
Iteration 100/1500: Train Loss = 1.836, Train Accuracy = 0.335
Iteration 100/1500. Validation Loss = 1.841, Validation Accuracy = 0
.363
Iteration 200/1500: Train Loss = 1.834, Train Accuracy = 0.380
Iteration 200/1500. Validation Loss = 1.782, Validation Accuracy = 0
.421
```

```
Iteration 300/1500: Train Loss = 1.716, Train Accuracy = 0.460
Iteration 300/1500. Validation Loss = 1.784, Validation Accuracy = 0
.429
Iteration 400/1500: Train Loss = 1.898, Train Accuracy = 0.415
Iteration 400/1500. Validation Loss = 1.868, Validation Accuracy = 0
.423
Iteration 500/1500: Train Loss = 1.769, Train Accuracy = 0.495
Iteration 500/1500. Validation Loss = 1.861, Validation Accuracy = 0
.470
Iteration 600/1500: Train Loss = 1.935, Train Accuracy = 0.440
Iteration 600/1500. Validation Loss = 2.003, Validation Accuracy = 0
.435
Iteration 700/1500: Train Loss = 1.890, Train Accuracy = 0.490
Iteration 700/1500. Validation Loss = 1.957, Validation Accuracy = 0
.480
Iteration 800/1500: Train Loss = 2.114, Train Accuracy = 0.470
Iteration 800/1500. Validation Loss = 2.073, Validation Accuracy = 0
.468
Iteration 900/1500: Train Loss = 2.018, Train Accuracy = 0.455
Iteration 900/1500. Validation Loss = 2.031, Validation Accuracy = 0
.485
Iteration 1000/1500: Train Loss = 1.986, Train Accuracy = 0.535
Iteration 1000/1500. Validation Loss = 2.156, Validation Accuracy =
0.480
Iteration 1100/1500: Train Loss = 2.369, Train Accuracy = 0.425
Iteration 1100/1500. Validation Loss = 2.381, Validation Accuracy =
0.443
Iteration 1200/1500: Train Loss = 2.332, Train Accuracy = 0.460
Iteration 1200/1500. Validation Loss = 2.258, Validation Accuracy =
0.482
Iteration 1300/1500: Train Loss = 2.340, Train Accuracy = 0.460
Iteration 1300/1500. Validation Loss = 2.355, Validation Accuracy =
0.471
Iteration 1400/1500: Train Loss = 2.182, Train Accuracy = 0.565
Iteration 1400/1500. Validation Loss = 2.386, Validation Accuracy =
0.480
Iteration 1499/1500: Train Loss = 2.712, Train Accuracy = 0.445
Iteration 1499/1500. Validation Loss = 2.542, Validation Accuracy =
0.457
Test Accuracy = 0.459
```

Neural Networks: Task 3 [5 points]

Why the ReLU layer is important? What will happen if we exclude this layer? What will be the accuracy on the test set?

Your Answer:

$$W_1(W_2X + b_2) + b_1 = (W_1W_2)X + (W_1b_2 + b_1)$$

Therefore two linear layers can be replaced with a single one. In theory, one would therefore expect performance similar to that of the logistic regression models used above. In practice some variance can be explained by the differing initialization and the increased effort of training more weights.

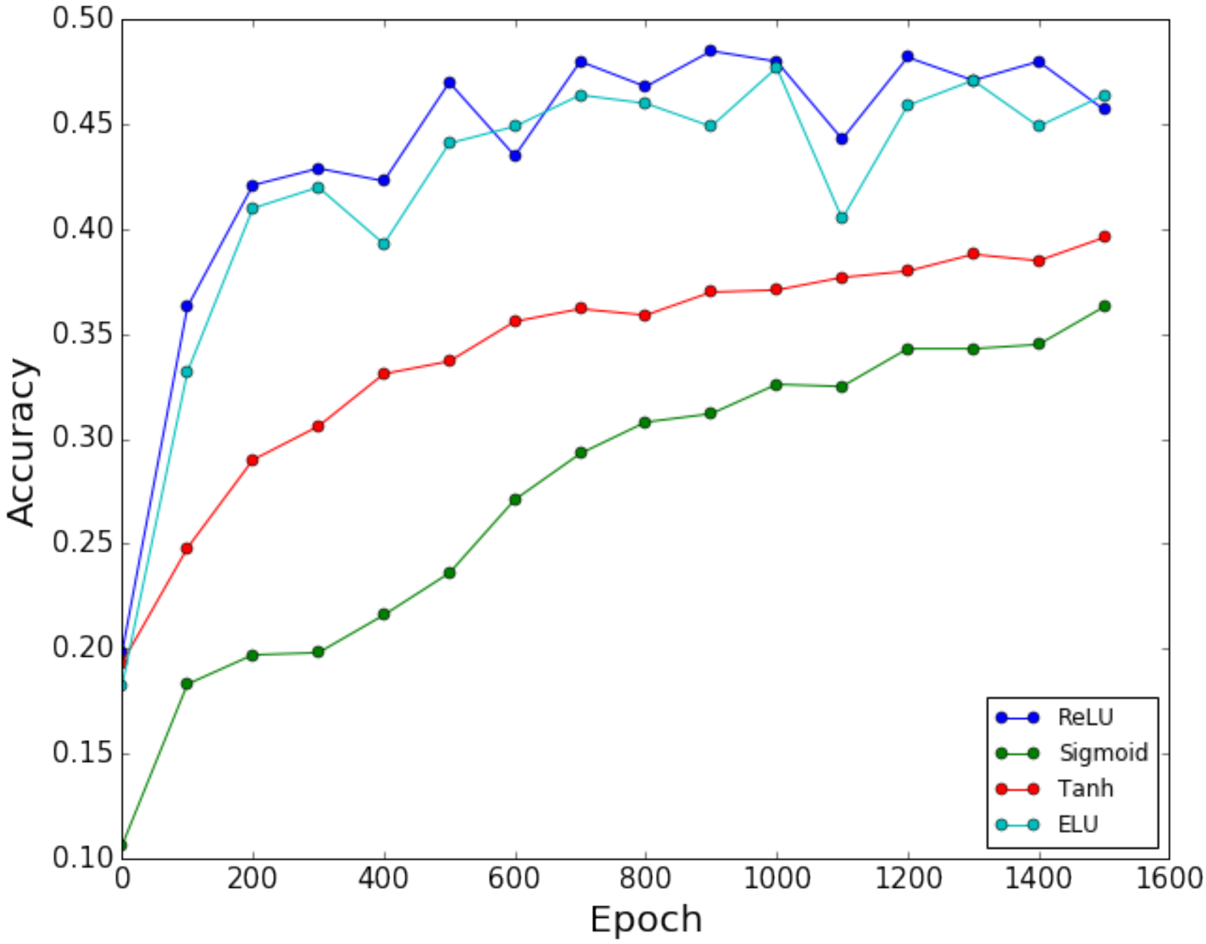
Implement other activation functions: Sigmoid (https://en.wikipedia.org/wiki/Sigmoid_function), Tanh (https://en.wikipedia.org/wiki/Hyperbolic_function#Hyperbolic_tangent) and ELU (https://arxiv.org/pdf/1511.07289v3.pdf) functions. Write your code for the ***forward*** and ***backward*** methods of ***SigmoidLayer***, ***TanhLayer*** and ***ELULayer*** classes in **layers.py** file.

In [75]:

```
# DONT CHANGE THE SEED AND THE DEFAULT PARAMETERS. OTHERWISE WE WILL N
# Seed
np.random.seed(42)

# Number of hidden units in a hidden layer.
num_hidden_units = 100

# Default parameters.
num_iterations = 1500
val_iteration = 100
batch_size = 200
learning_rate = 2e-3
weight_decay = 0
weight_scale = 0.0001

# Store results here
results = {}
layers_name = ['ReLU', 'Sigmoid', 'Tanh', 'ELU']
layers = [ReLULayer(), SigmoidLayer(), TanhLayer(), ELULayer({})]

for layer_name, layer in zip(layers_name, layers):
    ####################################################################
    # Build the model with the structure: linear_layer-activation-line
    # Train this model using Solver class with SGD optimizer. In confi
    # optimizer you need  to specify only the learning rate. Use the f
    # Store validation history in results dictionary variable.
    ####################################################################
    model = Network()
    lin1_params = {'input_size':X_train.shape[1],
                   'output_size':num_hidden_units,
                   'weight_decay':weight_decay,
                   'weight_scale':weight_scale }
    lin2_params = {'input_size':num_hidden_units,
                   'output_size':num_classes,
                   'weight_decay':weight_decay,
                   'weight_scale':weight_scale }
    model.add_layer(LinearLayer(lin1_params))
    model.add_layer(layer)
    model.add_layer(LinearLayer(lin2_params))
    model.add_loss(SoftMaxLoss)
    optimizer = SGD()
    optimizer_config = {'learning_rate': learning_rate}
    solver = Solver(model)
    res = solver.fit(X_train, Y_train, optimizer, optimizer_config, X_
                     batch_size, num_iterations, val_iteration, verbos
    val_acc_history = res[3]
    ####################################################################
    #                              END OF YOUR CODE
    ####################################################################
    results[layer_name] = val_acc_history
```

In [76]:

```
# Visualize a learning curve for different activation functions
for layer_name in layers_name:
    plt.plot(range(0, num_iterations + 1, val_iteration), results[laye
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.show()
```



Neural Networks: Task 4 [10 points]

Although typically a Softmax (https://en.wikipedia.org/wiki/Softmax_function) layer is coupled with a Cross Entropy loss (https://en.wikipedia.org/wiki/Cross_entropy#Cross-entropy_error_function_and_logistic_regression), this is not necessary and you can use a different loss function. Next, implement the network with the Softmax layer paired with a Hinge loss (https://en.wikipedia.org/wiki/Hinge_loss). Beware, with the Softmax layer all the output dimensions depend on all the input dimensions, hence, you need to compute the Jacobian of derivatives $\frac{\partial o_i}{dx_j}$.

Implement the ***forward*** and ***backward*** methods for ***SoftMaxLayer*** in **layers.py** file and ***CrossEntropyLoss*** and ***HingeLoss*** in **losses.py** file.

Results of using SoftMaxLoss and SoftMaxLayer + CrossEntropyLoss should be the same.

In [17]:

```
# DONT CHANGE THE SEED AND THE DEFAULT PARAMETERS. OTHERWISE WE WILL N
# Seed
np.random.seed(42)
```

```
# Default parameters.
num_iterations = 300 # 1500
val_iteration = 100
batch_size = 200
learning_rate = 2e-3
weight_decay = 0
weight_scale = 0.0001

#########################################################################
# TODO:
# Build the model with the structure:
# linear_layer-relu-linear_layer-softmax_layer-hinge_loss.
# Train this model using Solver class with SGD optimizer. In configura
# optimizer you need to specify only the learning rate. Use the fit me
#########################################################################
model = Network()
lin1_params = {'input_size':X_train.shape[1],
               'output_size':num_hidden_units,
               'weight_decay':weight_decay,
               'weight_scale':weight_scale }
lin2_params = {'input_size':num_hidden_units,
               'output_size':num_classes,
               'weight_decay':weight_decay,
               'weight_scale':weight_scale }
model.add_layer(LinearLayer(lin1_params))
model.add_layer(ReLULayer())
model.add_layer(LinearLayer(lin2_params))
model.add_layer(SoftMaxLayer())
model.add_loss(CrossEntropyLoss)
optimizer = SGD()
optimizer_config = {'learning_rate': learning_rate}
solver = Solver(model)
res = solver.fit(X_train, Y_train, optimizer, optimizer_config, X_val,
                 batch_size, num_iterations, val_iteration, verbose = 
#########################################################################
#                              END OF YOUR CODE
#########################################################################

#########################################################################
# TODO:
# Compute the accuracy on the test set.
#########################################################################
#test_acc = solver.score(X_test, Y_test)
#########################################################################
#                              END OF YOUR CODE
#########################################################################
print("Test Accuracy = {0:.3f}".format(test_acc))
```

```
Iteration 0/300: Train Loss = 2.306, Train Accuracy = 0.105
Iteration 0/300. Validation Loss = 2.309, Validation Accuracy = 0.07
0
Iteration 100/300: Train Loss = nan, Train Accuracy = 0.105
Iteration 100/300. Validation Loss = nan, Validation Accuracy = 0.08
7
```

```
uva_code/losses.py:58: RuntimeWarning: divide by zero encountered in
log
```

```
  loss = - np.mean(np.sum(np.log(xt + (1- xf)), 1), 0)

---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-17-0329df19edc1> in <module>()
     36 solver = Solver(model)
     37 res = solver.fit(X_train, Y_train, optimizer, optimizer_config, X_val, Y_val,
---> 38                   batch_size, num_iterations, val_iteration, verbose = True)
     39 ####################################################################################
     40 #                              END OF YOUR CODE                                    #

/home/frederik/PycharmProjects/uvadlc_practicals_2016/practical_1/uva_code/solver.pyc in fit(self, x_train, y_train, optimizer, optimizer_config, x_val, y_val, batch_size, num_iterations, val_iteration, verbose)
    152       # train loss and accuracy on this batch.                                #
    153       ##########################################################################################
--> 154       out, train_loss = self.train_on_batch(x_train_batch, y_train_batch)
    155       train_acc = self.accuracy(out, y_train_batch)
    156       ##########################################################################################

/home/frederik/PycharmProjects/uvadlc_practicals_2016/practical_1/uva_code/solver.pyc in train_on_batch(self, x_batch, y_batch)
     56     out = self.model.forward(x_batch)
     57     loss, dout = self.model.loss(out, y_batch)
---> 58     self.model.backward(dout)
     59     ########################################################################################
     60     #                              END OF YOUR CODE                                        #

/home/frederik/PycharmProjects/uvadlc_practicals_2016/practical_1/uva_code/models.pyc in backward(self, dout)
     99     ########################################################################################
    100     for layer in self.layers[::-1]:
--> 101       dout = layer.backward(dout)
    102     ########################################################################################
    103     #                              END OF YOUR CODE                                        #

/home/frederik/PycharmProjects/uvadlc_practicals_2016/practical_1/uva_code/layers.py in backward(self, dout)
    201     x = self.cache
    202     dx = np.dot(dout, self.params['w'].T )  # (b, in) = (b, out) (out, in)
```

```
--> 203     self.grads['w'] = np.dot(x.T, dout)  # (in out) = (in,b)
(b,out)
    204     self.grads['b'] = np.sum(dout, 0).T  # (b,out)
    205

KeyboardInterrupt:
```