python_assignment_1

November 9, 2016

1 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_cifar10
from uva_code.solver import Solver
from uva_code.losses import SoftMaxLoss, CrossEntropyLoss, HingeLoss
from uva_code.layers import LinearLayer, ReLULayer, SigmoidLayer, TanhLayer
from uva_code.models import Network
from uva_code.optimizers import SGD

%load_ext autoreload
%autoreload 2
```

1.1 Section 1: Data Preparation

In this section you will download CIFAR10 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 datas
# Try to run script to download the data. It should download tar archive, u
# If it is doesn't work for some reasons (like Permission denied) then manu
# http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz and extract it to
# cifar10 folder.

X_train_raw, Y_train_raw, X_test_raw, Y_test_raw = get_cifar10_raw_data()

#Checking shapes, should be (50000, 32, 32, 3), (50000, ), (10000, 32, 32,
print("Train data shape: {0}").format(str(X_train_raw.shape))
print("Train labels shape: {0}").format(str(Y_train_raw.shape))
print("Test data shape: {0}").format(str(X_test_raw.shape))
print("Test labels shape: {0}").format(str(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]:
```
# Normalize CIFAR10 data by subtracting the mean image. With these data you
# The validation subset will be used for tuning the hyperparameters.
X_train, Y_train, X_val, Y_val, X_test, Y_test = preprocess_cifar10_data(X_
                                                                         X_

#Checking shapes, should be (49000, 3072), (49000, ), (1000, 3072), (1000,
print "Train data shape: {0}".format(str(X_train.shape))
print "Train labels shape: {0}".format(str(Y_train.shape))
print "Val data shape: {0}".format(str(X_val.shape))
print "Val labels shape: {0}".format(str(Y_val.shape))
print "Test data shape: {0}".format(str(X_test.shape))
print "Test labels shape: {0}".format(str(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,)
```

In [5]:
```
# 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[idxs[
    plt.xticks([], [])
    plt.yticks(range(16, 320, 32), classes)
    plt.title('CIFAR10', fontsize = 20)
    plt.imshow(can)
    plt.show()
```

1.1.1 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, ReLU and two types of normalization (Per-example mean substraction, Standardization). What type of preprocessing you would prefer to use for each activation function and why? For example, in the previous cell we used per-example mean substraction.

Your Answer: Using a Sigmoid along with standardization makes more sense, since the standardization leads to a zero mean and unit variance. This fits best with the sigmoid since the sigmoid works well with values that are withon one standard deviation from the mean. The ReLU and the per-example mean substraction are better together than the Sigmoid and the per-example mean substraction, since there is a risk of vanishing gradients when using the Sigmoid function on input that is not centered at zero and between one standard deviation from the mean.

1.2 Section 2: Multinomial Logistic Regression [5 points]

In this section you will get started by implementing a linear classification model called Multinomial Logistic Regression. Later on you will extend this model to a neural network. You will train it by using the mini-batch Stochastic Gradient Descent algorithm. 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 NOT BE
        # Seed
        np.random.seed(42)

        # Default parameters.
        num_iterations = 1500
        val_iteration = 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 and st
        # weight_scale. Initialize the biases b with 0.
        ################################################################################
        W = np.random.normal(0.0, weight_scale,(num_classes,X_train.shape[1]))
        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 train
    # images to X_train_batch and labels to Y_train_batch variables.
    ###########################################################################
    X_train_batch = []
    Y_train_batch = []
    indices = np.random.choice(X_train.shape[0], batch_size, False)
    for i in range(0,batch_size):
        X_train_batch.append(X_train[indices[i]])
        Y_train_batch.append(Y_train[indices[i]])
    X_train_batch = np.matrix(X_train_batch)
    ###########################################################################
    #                                 END OF YOUR CODE
    ###########################################################################

    ###########################################################################
    # TODO:
    # Compute the loss and the accuracy of the multinomial logistic regress
    # on X_train_batch, Y_train_batch. The loss should be an average of the
    # samples in the mini-batch. Include to the loss L2-regularization over
    # matrix W with regularization parameter equals to weight_decay.
    ###########################################################################
    predictions = np.dot(X_train_batch,W.T) + b
    predictions = np.exp(predictions)
    summed = np.sum(predictions, 1)[:,None]
    norm_predictions = predictions/np.sum(predictions, 1)

    train_loss = 0
    train_acc = 0
    correct = 0
    yTrs = []
    for i in range(0, batch_size):
        yTr = np.zeros(num_classes)
        yTr[Y_train_batch[i]] = 1
        yTrs.append(yTr)
        if (np.argmax(norm_predictions[i,:]) == Y_train_batch[i]):
            correct += 1
    train_acc = float(correct) / batch_size
    vl = -np.multiply(np.matrix(yTrs), np.log(np.matrix(norm_predictions)))
    train_loss = np.sum(vl)/batch_size # + (weight_decay*np.sum(np.sum(np.p
    ###########################################################################
    #                                 END OF YOUR CODE
    ###########################################################################

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

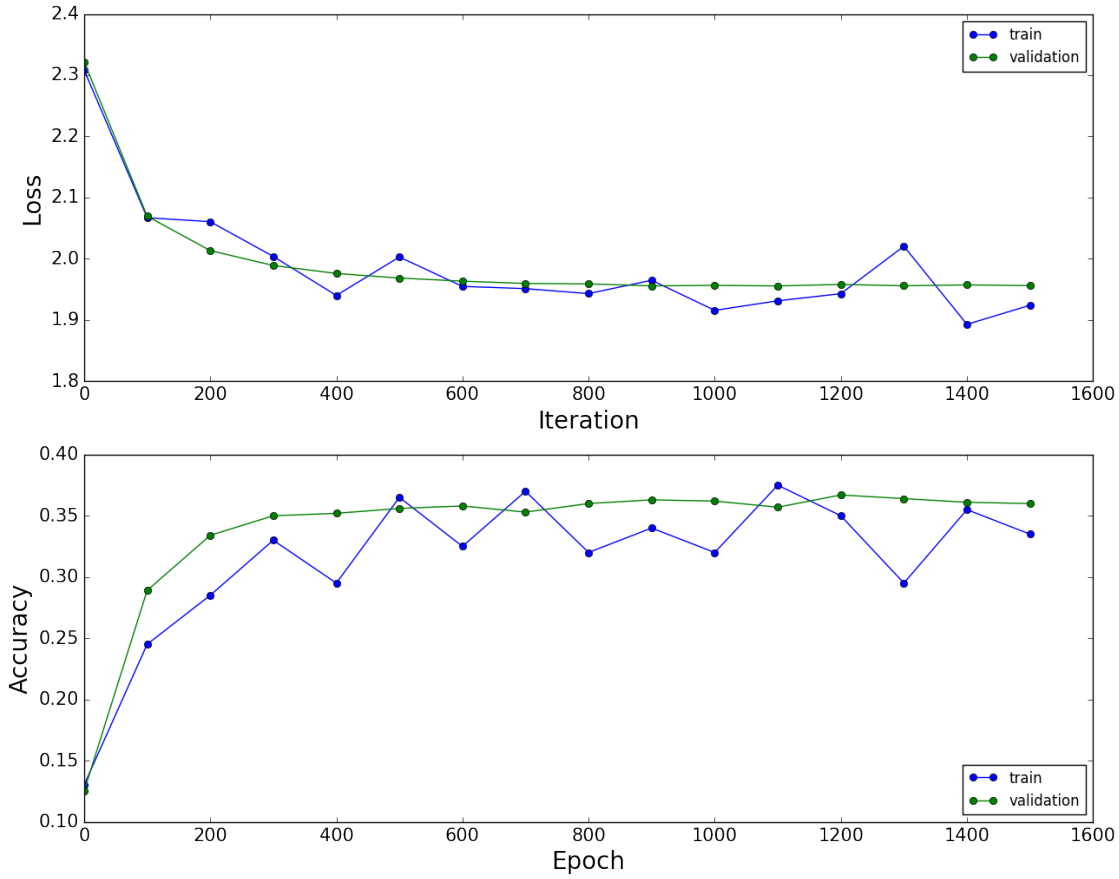
```
# TODO:
# Compute the gradients of the loss with the respect to the weights and
# them in dW and db variables.
#############################################################################
dW = np.dot(-np.matrix(X_train_batch).T, (yTrs - norm_predictions)) / b
db = np.sum(-(yTrs - norm_predictions), 0) / batch_size
#############################################################################
#                                 END OF YOUR CODE
#############################################################################

#############################################################################
# TODO:
# Update the weights W and biases b using the Stochastic Gradient Desce
#############################################################################
W = W - (learning_rate*dW.T) - weight_decay*learning_rate*W
b = b - learning_rate*db - weight_decay*learning_rate*b
#############################################################################
#                                 END OF YOUR CODE
#############################################################################

if iteration % val_iteration == 0 or iteration == num_iterations - 1:
    #########################################################################
    # TODO:
    # Compute the loss and the accuracy on the validation set.
    #########################################################################
    predictions = np.dot(X_val,W.T) + b
    predictions = np.exp(predictions)
    summed = np.sum(predictions, 1)[:,None]
    norm_predictions = predictions/np.sum(predictions, 1)

    val_loss = 0
    val_acc = 0
    correct = 0
    yTrs = []
    for i in range(0, len(X_val)):
        yTr = np.zeros(num_classes)
        yTr[Y_val[i]] = 1
        yTrs.append(yTr)
        if (np.argmax(norm_predictions[i,:]) == Y_val[i]):
            correct += 1

    vl = -np.multiply(np.matrix(yTrs), np.log(np.matrix(norm_prediction
    val_loss = np.sum(vl)/X_val.shape[0] #+ (weight_decay*np.sum(np.sum
    val_acc = float(correct) / len(X_val)
    #########################################################################
    #                                 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 Accuracy
                  format(iteration, num_iterations, train_loss, train_acc))
            print("Iteration {0:d}/{1:d}. Validation Loss = {2:.3f}, Validation
                  format(iteration, num_iterations, val_loss, val_acc))

    ################################################################################
    # TODO:
    # Compute the accuracy on the test set.
    ################################################################################
    predictions = np.dot(X_test,W.T) + b
    predictions = np.exp(predictions)
    summed = np.sum(predictions, 1)[:,None]
    norm_predictions = predictions/np.sum(predictions, 1)

    correct = 0
    yTrs = []
    for i in range(0, len(X_test)):
        yTr = np.zeros(num_classes)
        yTr[Y_test[i]] = 1
        yTrs.append(yTr)
        if (np.argmax(norm_predictions[i,:]) == Y_test[i]):
            correct += 1
    test_acc = float(correct) / len(X_test)
    ################################################################################
    #                              END OF YOUR CODE
    ################################################################################
    print("Test Accuracy = {0:.3f}".format(test_acc))
```

```
Iteration 0/1500. Train Loss = 2.309, Train Accuracy = 0.130
Iteration 0/1500. Validation Loss = 2.322, Validation Accuracy = 0.125
Iteration 100/1500. Train Loss = 2.067, Train Accuracy = 0.245
Iteration 100/1500. Validation Loss = 2.070, Validation Accuracy = 0.289
Iteration 200/1500. Train Loss = 2.061, Train Accuracy = 0.285
Iteration 200/1500. Validation Loss = 2.013, Validation Accuracy = 0.334
Iteration 300/1500. Train Loss = 2.004, Train Accuracy = 0.330
Iteration 300/1500. Validation Loss = 1.989, Validation Accuracy = 0.350
Iteration 400/1500. Train Loss = 1.940, Train Accuracy = 0.295
Iteration 400/1500. Validation Loss = 1.976, Validation Accuracy = 0.352
Iteration 500/1500. Train Loss = 2.003, Train Accuracy = 0.365
Iteration 500/1500. Validation Loss = 1.968, Validation Accuracy = 0.356
Iteration 600/1500. Train Loss = 1.955, Train Accuracy = 0.325
Iteration 600/1500. Validation Loss = 1.963, Validation Accuracy = 0.358
Iteration 700/1500. Train Loss = 1.951, Train Accuracy = 0.370
```

```
Iteration 700/1500. Validation Loss = 1.960, Validation Accuracy = 0.353
Iteration 800/1500. Train Loss = 1.943, Train Accuracy = 0.320
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 = 1.956, Validation Accuracy = 0.363
Iteration 1000/1500. Train Loss = 1.916, Train Accuracy = 0.320
Iteration 1000/1500. Validation Loss = 1.956, Validation Accuracy = 0.362
Iteration 1100/1500. Train Loss = 1.931, Train Accuracy = 0.375
Iteration 1100/1500. Validation Loss = 1.956, Validation Accuracy = 0.357
Iteration 1200/1500. Train Loss = 1.943, Train Accuracy = 0.350
Iteration 1200/1500. Validation Loss = 1.958, Validation Accuracy = 0.367
Iteration 1300/1500. Train Loss = 2.020, Train Accuracy = 0.295
Iteration 1300/1500. Validation Loss = 1.956, Validation Accuracy = 0.364
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 = 0.335
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 classifier
plt.subplot(2, 1, 1)
plt.plot(range(0, num_iterations + 1, val_iteration), train_loss_history, '
plt.plot(range(0, num_iterations + 1, val_iteration), val_loss_history, '-o
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_history, '-
plt.plot(range(0, num_iterations + 1, val_iteration), val_acc_history, '-o'
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.gcf().set_size_inches(15, 12)
plt.show()
```



1.2.1 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: One would expect an accuracy of approximately 0.1 (10%) during the first iteration, since the weights are initialized randomly and there are ten classes thus 10% chance of guessing the right class. So if the probability of guessing the right class is 0.1 and we take the negative log in the loss function, we expect a loss of about 2.3 (-ln(0.1) = 2.3)

1.2.2 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: Three factors that determine the size of batches in practice are the size of the dataset, the computational cost of the backpropagation and the size of the network. The size of the dataset influences the choice of the batch size since a large batch size on a small dataset does not make much sense. The computational cost and size of the network influence the batch size because having a very expensive backpropagation computation or a very large network would encourage the use of larger batch sizes to minimize the time the algorithm needs to learn.

1.2.3 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: The learning rate depends on the batch size, since a larger batch size ensures that there is less variance leading to being able to use a larger learning rate.

If, for instance, the batch size were set to 1 the sample that is examined can have a very different gradient than another sample. If the sample for some reason is not representative for the entire data then a large learning rate will cause a large step in the "wrong" direction. The advantage of this batch size is that the iterations are very fast.

If the batch size were to be set to the size of the data set than the error that will be backpropagated is an average error over all samples in the dataset, so we know that the step will be taken in the right direction. The disadvantage is that having such a large batch size is very slow.

1.2.4 Multinomial Logistic Regression: Question 4 [4 points]

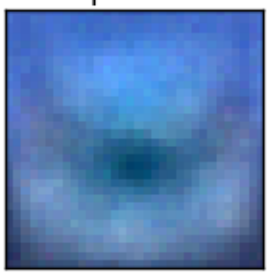
Suppose that the weight matrix W has the shape (num_features, num_classes). How can you describe the columns of the weight matrix W? What are they representing? Why?

Your Answer: When the columns of the weight matrix W are reshaped to the size of the input images (32, 32, 3) they represent basic forms of the object that belongs to the class. For instance, the weights of the car look like a very coarse representation of a car.

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

In [8]:
```
################################################################################
# TODO:
# Visualize the learned weights for each class.
################################################################################
for i in range(0,10):
    w = np.array(W[i,:])
    w = w.reshape(32,32,3)
    w = (w - np.min(w))
    w = w / np.max(w)
    plt.subplot(2,5,i+1)
    plt.xticks([], [])
    plt.yticks([],[])
    plt.title(classes[i])
    plt.imshow(w)
    plt.show()
################################################################################
#                                END OF YOUR CODE
################################################################################
```

plane



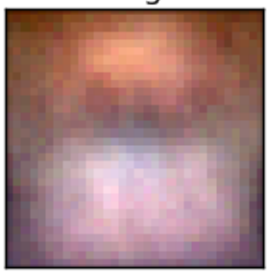
car

bird

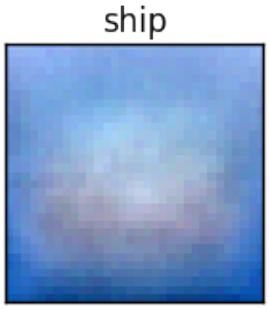
cat

deer

dog



frog



1.3 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.

1.4 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.

All computations should be implemented in vectorized. Don't loop over samples in the mini-batch.

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

```
In [9]: # DONT CHANGE THE SEED AND THE DEFAULT PARAMETERS. OTHERWISE WE WILL NOT BE
        # Seed
        np.random.seed(42)

        # Default parameters.
        num_iterations = 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 Network mo
        # will need to use add_layer and add_loss methods. Train this model using S
        # with SGD optimizer. In configuration of the optimizer you need to specify
        # learning rate. Use the fit method to train classifier. Don't forget to in
        # X_val and Y_val in arguments to output the validation loss and accuracy d
        # training. Set the verbose to True to compare with the  multinomial logist
        # regression classifier from the Section 2.
        ################################################################################

        optimizer_config = {'learning_rate': learning_rate}

        params = {'input_size': X_train.shape[1], 'output_size': num_classes, 'weig

        model = Network()
```

```
        model.add_layer(LinearLayer(params))
        model.add_loss(SoftMaxLoss)
        optimizer = SGD()
        solver = Solver(model)
        solver.fit(X_train, Y_train, optimizer, optimizer_config, X_val, Y_val, bat
        ################################################################################
        #                                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 = 0.125
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
```

1.4.1 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 which produces the highest accuracy on the validation set? What is test accuracy of the model trained with the found best hyperparameters values?

Your Answer: The best learning rate and weight decay that were found were 1.000000e-06 and 3.000000e+02 respectively. These lead to a validation accuracy of 0.413 and a test accuracy of 0.402.

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

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

```
In [10]: # DONT CHANGE THE SEED AND THE DEFAULT PARAMETERS. OTHERWISE WE WILL NOT B
         # 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+02]

         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 logi
                 # maximum of the validation accuracy in best_val_acc and correspon
                 # best_solver variables. Store the maximum of the validation score
                 # setting of the hyperparameters in cur_val_acc variable.
                 ################################################################
                 optimizer_config = {'learning_rate': learning_rate}

                 params = {'input_size': X_train.shape[1], 'output_size': num_class

                 model = Network()
```

```
                model.add_layer(LinearLayer(params))
                model.add_loss(SoftMaxLoss)
                optimizer = SGD()
                solver = Solver(model)
                [tl, ta, vl, va] = solver.fit(X_train, Y_train, optimizer, optimiz
                                              X_val, Y_val, batch_size, num_iterations, v
                cur_val_acc = va[-1]
                if cur_val_acc > best_val_acc:
                    best_solver = solver
                    best_val_acc = cur_val_acc


                #cur_val_acc = None
                ################################################################
                #                                   END OF YOUR CODE
                ################################################################
                print("Learning rate = {0:e}, weight decay = {1:e}: Validation Acc
                    learning_rate, weight_decay, cur_val_acc))

        ################################################################################
        # TODO:
        # Compute the accuracy on the test set for the best solver.
        ################################################################################
        test_acc = best_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: Validation Accuracy = 0.
Learning rate = 1.000000e-06, weight decay = 3.000000e+02: Validation Accuracy = 0.
Learning rate = 1.000000e-07, weight decay = 3.000000e+01: Validation Accuracy = 0.
Learning rate = 1.000000e-07, weight decay = 3.000000e+02: Validation Accuracy = 0.
Best Test Accuracy = 0.402
```

1.4.2 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 [11]: # DONT CHANGE THE SEED AND THE DEFAULT PARAMETERS. OTHERWISE WE WILL NOT B
         # 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

################################################################################
# TODO:
# Build the model with the structure: linear_layer-relu-linear_layer-softm
# Train this model using Solver class with SGD optimizer. In configuration
# optimizer you need to specify only the learning rate. Use the fit method
################################################################################
optimizer_config = {'learning_rate': learning_rate}

LL1_params = {'input_size': X_train.shape[1], 'output_size': num_hidden_un
LL2_params = {'input_size': num_hidden_units, 'output_size': num_classes,

model = Network()
model.add_layer(LinearLayer(LL1_params))
model.add_layer(ReLULayer())
model.add_layer(LinearLayer(LL2_params))
model.add_loss(SoftMaxLoss)
optimizer = SGD()
solver = Solver(model)
solver.fit(X_train, Y_train, optimizer, optimizer_config, X_val, Y_val, ba
################################################################################
#                                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 = 2.303, Train Accuracy = 0.105
Iteration 0/1500. Validation Loss = 2.302, Validation Accuracy = 0.198
Iteration 100/1500: Train Loss = 1.776, Train Accuracy = 0.335
Iteration 100/1500. Validation Loss = 1.780, Validation Accuracy = 0.363
Iteration 200/1500: Train Loss = 1.710, Train Accuracy = 0.380
```

```
Iteration 200/1500. Validation Loss = 1.657, Validation Accuracy = 0.421
Iteration 300/1500: Train Loss = 1.530, Train Accuracy = 0.460
Iteration 300/1500. Validation Loss = 1.597, Validation Accuracy = 0.429
Iteration 400/1500: Train Loss = 1.650, Train Accuracy = 0.415
Iteration 400/1500. Validation Loss = 1.620, Validation Accuracy = 0.423
Iteration 500/1500: Train Loss = 1.453, Train Accuracy = 0.495
Iteration 500/1500. Validation Loss = 1.543, Validation Accuracy = 0.470
Iteration 600/1500: Train Loss = 1.549, Train Accuracy = 0.440
Iteration 600/1500. Validation Loss = 1.617, Validation Accuracy = 0.435
Iteration 700/1500: Train Loss = 1.444, Train Accuracy = 0.490
Iteration 700/1500. Validation Loss = 1.509, Validation Accuracy = 0.480
Iteration 800/1500: Train Loss = 1.607, Train Accuracy = 0.470
Iteration 800/1500. Validation Loss = 1.566, Validation Accuracy = 0.468
Iteration 900/1500: Train Loss = 1.439, Train Accuracy = 0.455
Iteration 900/1500. Validation Loss = 1.451, Validation Accuracy = 0.485
Iteration 1000/1500: Train Loss = 1.340, Train Accuracy = 0.535
Iteration 1000/1500. Validation Loss = 1.509, Validation Accuracy = 0.480
Iteration 1100/1500: Train Loss = 1.654, Train Accuracy = 0.425
Iteration 1100/1500. Validation Loss = 1.667, Validation Accuracy = 0.443
Iteration 1200/1500: Train Loss = 1.551, Train Accuracy = 0.460
Iteration 1200/1500. Validation Loss = 1.476, Validation Accuracy = 0.482
Iteration 1300/1500: Train Loss = 1.497, Train Accuracy = 0.460
Iteration 1300/1500. Validation Loss = 1.512, Validation Accuracy = 0.471
Iteration 1400/1500: Train Loss = 1.274, Train Accuracy = 0.565
Iteration 1400/1500. Validation Loss = 1.477, Validation Accuracy = 0.480
Iteration 1499/1500: Train Loss = 1.744, Train Accuracy = 0.445
Iteration 1499/1500. Validation Loss = 1.575, Validation Accuracy = 0.457
Test Accuracy = 0.459
```

1.4.3 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?

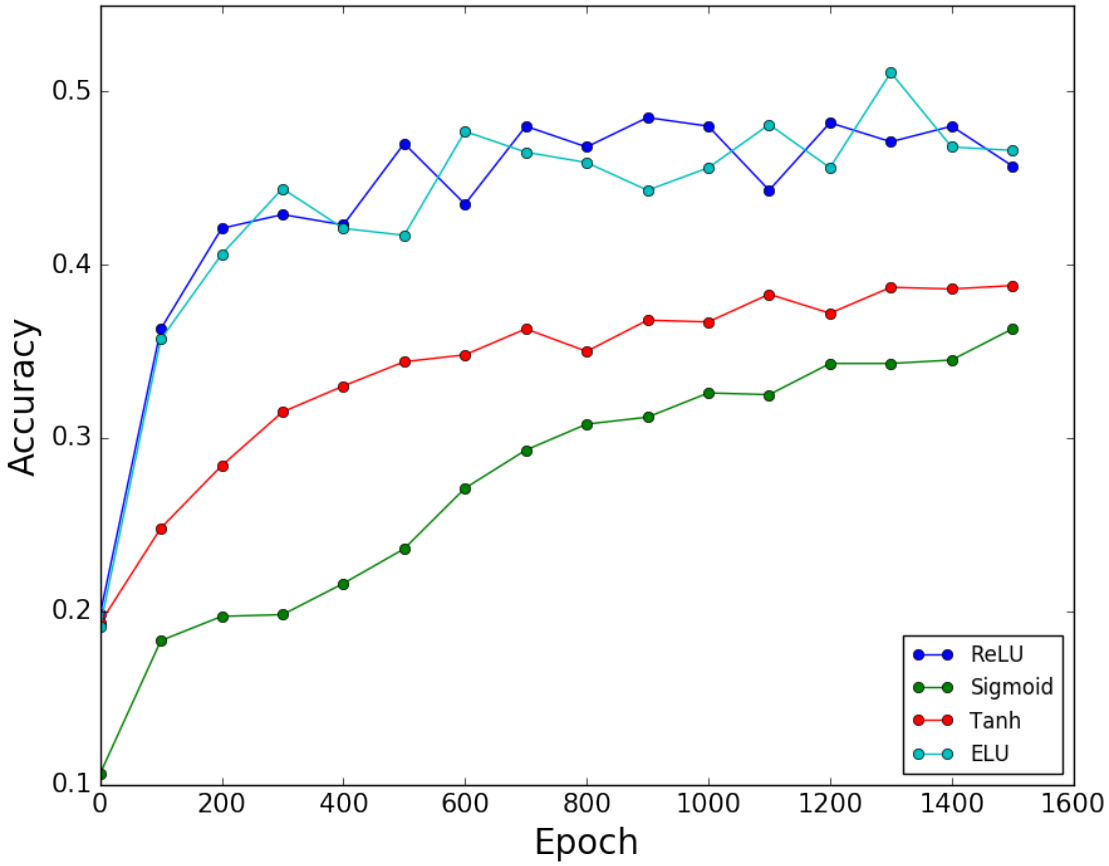
Your Answer: the ReLU layer is important for taking care of the problem of vanishing gradients, if we exclude this layer the learning will go in a much slower rate. The accuracy on the test set when excluding the ReLU layer is 0.389 while it is 0.455 when using a ReLU layer.

Implement other activation functions: Sigmoid, Tanh and ELU functions. Write your code for the ***forward*** and ***backward*** methods of ***SigmoidLayer***, ***TanhLayer*** and ***ELULayer*** classes in **layers.py** file.

```
In [12]: # DONT CHANGE THE SEED AND THE DEFAULT PARAMETERS. OTHERWISE WE WILL NOT B
         # 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-linear_l
    # Train this model using Solver class with SGD optimizer. In configura
    # optimizer you need  to specify only the learning rate. Use the fit m
    # Store validation history in results dictionary variable.
    ################################################################
    optimizer_config = {'learning_rate': learning_rate}

    LL1_params = {'input_size': X_train.shape[1], 'output_size': num_hidde
    LL2_params = {'input_size': num_hidden_units, 'output_size': num_class

    model = Network()
    model.add_layer(LinearLayer(LL1_params))
    model.add_layer(layer)

    model.add_layer(LinearLayer(LL2_params))
    model.add_loss(SoftMaxLoss)
    optimizer = SGD()
    solver = Solver(model)
    [tl, ta, vl, va] = solver.fit(X_train, Y_train, optimizer, optimizer_c
                                  X_val, Y_val, batch_size, num_iterations, v
    val_acc_history = va
    #                                END OF YOUR CODE
    ################################################################
    results[layer_name] = val_acc_history
```

In [13]:
```
# Visualize a learning curve for different activation functions
for layer_name in layers_name:
    plt.plot(range(0, num_iterations + 1, val_iteration), results[layer_na
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.show()
```



1.4.4 Neural Networks: Task 4 [10 points]

Although typically a Softmax layer is coupled with a Cross Entropy loss, 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. 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. You should implement multi-class cross-entropy and hinge losses.

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

```
In [14]: # DONT CHANGE THE SEED AND THE DEFAULT PARAMETERS. OTHERWISE WE WILL NOT B
         # Seed
         np.random.seed(42)

         # Default parameters.
         num_iterations = 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 configuration
# optimizer you need to specify only the learning rate. Use the fit method
################################################################################

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

################################################################################
# TODO:
# Compute the accuracy on the test set.
################################################################################
test_acc = None
################################################################################
#                                END OF YOUR CODE
################################################################################
print("Test Accuracy = {0:.3f}".format(test_acc))
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-14-c20757d07343> in <module>()
     31 #                                END OF YOUR CODE
     32 ################################################################################
---> 33 print("Test Accuracy = {0:.3f}".format(test_acc))


ValueError: Unknown format code 'f' for object of type 'str'
```

In [ ]:

In [ ]: