toy_nn

November 11, 2020

1 CS145 Howework 3, Part 2: Neural Networks

Important Note: HW3 is due on **11:59 PM PT, Nov 9 (Monday, Week 6)**. Please submit through GradeScope.

Note that, Howework #3 has two jupyter notebooks to complete (Part 1: kNN and Part 2: Neural Network).

1.1 Print Out Your Name and UID

Name: Devyan Biswas, UID: 804988161

1.2 Before You Start

You need to first create HW3 conda environment by the given `cs145hw3.yml` file, which provides the name and necessary packages for this tasks. If you have `conda` properly installed, you may create, activate or deactivate by the following commands:

```
conda env create -f cs145hw3.yml
conda activate hw3
conda deactivate
```

OR

```
conda env create --name NAMEOFYOURCHOICE -f cs145hw3.yml
conda activate NAMEOFYOURCHOICE
conda deactivate
```

To view the list of your environments, use the following command:

```
conda env list
```

More useful information about managing environments can be found here.

You may also quickly review the usage of basic Python and Numpy package, if needed in coding for matrix operations.

In this notebook, you must not delete any code cells in this notebook. If you change any code outside the blocks (such as hyperparameters) that you are allowed to edit (between `STRART/END YOUR CODE HERE`), you need to highlight these changes. You may add some additional cells to help explain your results and observations.

1.3 Section 1: Backprop in a neural network

Note: Section 1 is "question-answer" style problem. You do not need to code anything and you are required to calculate by hand (with a scientific calculator), which helps you understand the back propagation in neural networks.

In this question, let's consider a simple two-layer neural network and manually do the forward and backward pass. For simplicity, we assume our input data is two dimension. Then the model architecture looks like the following. Notice that in the example we saw in class, the bias term `b` was not explicit listed in the architecture diagram. Here we include the term `b` explicitly for each layer in the diagram. Recall the formula for computing $\mathbf{x}^{(l)}$ in the l-th layer from $\mathbf{x}^{(l-1)}$ in the $(l-1)$-th layer is $\mathbf{x}^{(l)} = \mathbf{f}^{(l)}(\mathbf{W}^{(l)}\mathbf{x}^{(l-1)} + \mathbf{b}^{(l)})$. The activation function $\mathbf{f}^{(l)}$ we choose is the `sigmoid` function for all layers, i.e. $\mathbf{f}^{(l)}(z) = \frac{1}{1+\exp(-z)}$. The final loss function is $\frac{1}{2}$ of the `mean squared error` loss, i.e. $l(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{2}||\mathbf{y} - \hat{\mathbf{y}}||^2$.

We initialize our weights as

$$\mathbf{W}^{(1)} = \begin{bmatrix} 0.15 & 0.2 \\ 0.25 & 0.3 \end{bmatrix}, \quad \mathbf{W}^{(2)} = [0.4, 0.45], \quad \mathbf{b}^{(1)} = [0.35, 0.35], \quad \mathbf{b}^{(2)} = 0.6$$

1.3.1 Forward pass

Questions

1. When the input $\mathbf{x}^{(0)} = [0.05, 0.1]$, what will be the value of $\mathbf{x}^{(1)}$ in the hidden layer? (Show your work).
2. Based on the value $\mathbf{x}^{(1)}$ you computed, what will be the value of $\mathbf{x}^{(2)}$ in the output layer? (Show your work).
3. When the target value of this input is $y = 0.01$, based on the value $\mathbf{x}^{(2)}$ you computed, what will be the loss? (Show your work).

Answers:

1. $\mathbf{x}^{(1)} = \mathbf{f}^{(1)}(\begin{bmatrix} 0.15 & 0.2 \\ 0.25 & 0.3 \end{bmatrix} * [0.05, 0.1] + [0.35, 0.35]) \approx [0.593, 0.597]$
2. $\mathbf{x}^{(2)} = \mathbf{f}^{(2)}(\begin{bmatrix} 0.4 & 0.45 \end{bmatrix} * [0.593, 0.597] + 0.6) \approx 0.751$
3. $\frac{(0.01-0.751)^2}{2} \approx 0.275$

1.3.2 Backward pass

With the loss computed below, we are ready for a backward pass to update the weights in the neural network. Kindly remind that the gradients of a variable should have the same shape with the variable.

Questions

1. Consider the loss l of the same input $\mathbf{x}^{(0)} = [0.05, 0.1]$, what will be the update of $\mathbf{W}^{(2)}$ and $\mathbf{b}^{(2)}$ when we backprop, i.e. $\frac{\partial l}{\partial \mathbf{W}^{(2)}}$, $\frac{\partial l}{\partial \mathbf{b}^{(2)}}$ (Show your work in detailed calculation steps. Answers without justification will not be credited.).
2. Based on the result you computed in part 1, when we keep backproping, what will be the update of $\mathbf{W}^{(1)}$ and $\mathbf{b}^{(1)}$, i.e. $\frac{\partial l}{\partial \mathbf{W}^{(1)}}$, $\frac{\partial l}{\partial \mathbf{b}^{(1)}}$ (Show your work in details calculation steps. Answers without justification will not be credited.).

Answers:

1. $\frac{\partial l}{\partial \mathbf{W}^{(2)}} => -(y - x^{(2)}) * f^{'}(z^{(2)}) * x^{(1)}$

$z^{(2)} = 0.665$
$f^{'}(z) = \sigma(z)(1 - \sigma(z))$; plugging and chugging:
$-(0.01 - 0.75) * (\sigma(0.665) * (1 - \sigma(0.665)) = \delta^{(2)} = \frac{\partial l}{\partial \mathbf{b}^{(2)}} \approx \boxed{0.1662}$
$\frac{\partial l}{\partial \mathbf{W}^{(2)}} = \delta^{(2)} * x^{(1)} = \boxed{[0.0986\ ,\ 0.0992]}$
2 (Doing one calc for all, since same nodes propogate backward and only one output going back through both)
$\frac{\partial l}{\partial \mathbf{W}_{\mathbf{k}}^{(1)}} => \delta^{(2)} \mathbf{W}_{\mathbf{j}}^{(\mathbf{1})} * f^{'}(z^{(1)}) * x_k^{(0)}$
$z^{(1)} = [0.3775, 0.3925]$
$0.1662 * [0.4, 0.45] * (\sigma(z^{(1)}) * (1 - \sigma(z^{(1)})) = \delta^{(1)} = \frac{\partial l}{\partial \mathbf{b}^{(1)}} \approx \boxed{[0.0321\ ,\ 0.0361]}$
$\frac{\partial l}{\partial \mathbf{W}_{\mathbf{k}}^{(1)}} = \delta^{(1)} * x^{(0)} = \begin{bmatrix} 0.00522 & 0.00522 \\ 0.00522 & 0.00522 \end{bmatrix}$

1.4 Section 2: Coding a two-layer neural network

Import libraries and define relative error function, which is used to check results later.

```
In [23]: import random
         import numpy as np
         from data.data_utils import load_CIFAR10
         import matplotlib.pyplot as plt

         %matplotlib inline
         plt.rcParams['figure.figsize'] = (10.0, 8.0)
         %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
```

1.5 Toy example

Before loading CIFAR-10, there will be a toy example to test your implementation of the forward and backward pass.

```
In [24]: from hw3code.neural_net import TwoLayerNet
```

```
In [25]: # Create a small net and some toy data to check your implementations.
         # Note that we set the random seed for repeatable experiments.
```

```
input_size = 4
hidden_size = 10
num_classes = 3
num_inputs = 5

def init_toy_model():
    np.random.seed(0)
    return TwoLayerNet(input_size, hidden_size, num_classes, std=1e-1)

def init_toy_data():
    np.random.seed(1)
    X = 10 * np.random.randn(num_inputs, input_size)
    y = np.array([0, 1, 2, 2, 1])
    return X, y

net = init_toy_model()
X, y = init_toy_data()
```

1.5.1 Compute forward pass scores

In [26]:
```
## Implement the forward pass of the neural network.

# Note, there is a statement if y is None: return scores, which is why
# the following call will calculate the scores.
scores = net.loss(X)
print('Your scores:')
print(scores)
print()
print('correct scores:')
correct_scores = np.asarray([
    [-1.07260209,  0.05083871, -0.87253915],
    [-2.02778743, -0.10832494, -1.52641362],
    [-0.74225908,  0.15259725, -0.39578548],
    [-0.38172726,  0.10835902, -0.17328274],
    [-0.64417314, -0.18886813, -0.41106892]])
print(correct_scores)
print()

# The difference should be very small. We get < 1e-7
print('Difference between your scores and correct scores:')
print(np.sum(np.abs(scores - correct_scores)))
```

```
Your scores:
[[-1.07260209  0.05083871 -0.87253915]
 [-2.02778743 -0.10832494 -1.52641362]
 [-0.74225908  0.15259725 -0.39578548]
 [-0.38172726  0.10835902 -0.17328274]
```

```
 [-0.64417314 -0.18886813 -0.41106892]]

correct scores:
[[-1.07260209  0.05083871 -0.87253915]
 [-2.02778743 -0.10832494 -1.52641362]
 [-0.74225908  0.15259725 -0.39578548]
 [-0.38172726  0.10835902 -0.17328274]
 [-0.64417314 -0.18886813 -0.41106892]]

Difference between your scores and correct scores:
3.381231204052648e-08
```

1.5.2 Forward pass loss

The total loss includes data loss (MSE) and regularization loss, which is,

$$L = L_{data} + L_{reg} = \frac{1}{2N}\sum_{i=1}^{N}\left(\boldsymbol{y}_{\text{pred}} - \boldsymbol{y}_{\text{target}}\right)^2 + \frac{\lambda}{2}\left(||W_1||^2 + ||W_2||^2\right)$$

More specifically in multi-class situation, if the output of neural nets from one sample is $y_{\text{pred}} = (0.1, 0.1, 0.8)$ and $y_{\text{target}} = (0, 0, 1)$ from the given label, then the MSE error will be $Error = (0.1 - 0)^2 + (0.1 - 0)^2 + (0.8 - 1)^2 = 0.06$

Implement data loss and regularization loss. In the MSE function, you also need to return the gradients which need to be passed backward. This is similar to batch gradient in linear regression. Test your implementation of loss functions. The Difference should be less than 1e-12.

```
In [27]: loss, _ = net.loss(X, y, reg=0.05)
         correct_loss_MSE = 1.8973332763705641

         # should be very small, we get < 1e-12
         print('Difference between your loss and correct loss:')
         print(np.sum(np.abs(loss - correct_loss_MSE)))
```

```
Difference between your loss and correct loss:
0.0
```

1.5.3 Backward pass (You do not need to implemented this part)

We have already implemented the backwards pass of the neural network for you. Run the block of code to check your gradients with the gradient check utilities provided. The results should be automatically correct (tiny relative error).

If there is a gradient error larger than 1e-8, the training for neural networks later will be negatively affected.

```
In [28]: from data.gradient_check import eval_numerical_gradient

         # Use numeric gradient checking to check your implementation of the backward pass.
```

```
        # If your implementation is correct, the difference between the numeric and
        # analytic gradients should be less than 1e-8 for each of W1, W2, b1, and b2.

        loss, grads = net.loss(X, y, reg=0.05)

        # these should all be less than 1e-8 or so
        for param_name in grads:
            f = lambda W: net.loss(X, y, reg=0.05)[0]
            param_grad_num = eval_numerical_gradient(f, net.params[param_name], verbose=False)
            print('{} max relative error: {}'.format(param_name, rel_error(param_grad_num, grad
```

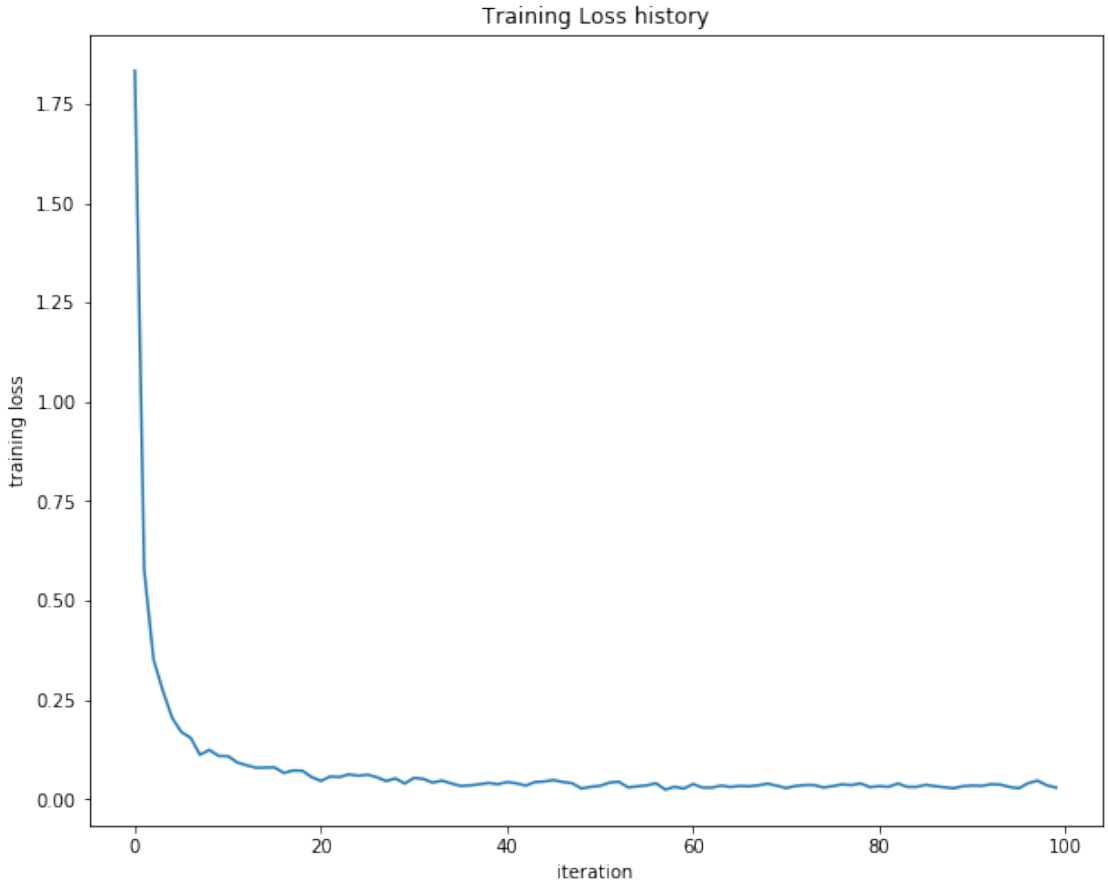
```
W2 max relative error: 8.80091875172355e-11
b2 max relative error: 2.4554844805570154e-11
W1 max relative error: 1.7476665046687833e-09
b1 max relative error: 7.382451041178829e-10
```

1.5.4 Training the network

Implement neural_net.train() to train the network via stochastic gradient descent, much like the linear regression.

```
In [29]: net = init_toy_model()
         stats = net.train(X, y, X, y,
                     learning_rate=1e-1, reg=5e-6,
                     num_iters=100, verbose=False)

         print('Final training loss: ', stats['loss_history'][-1])

         # plot the loss history
         plt.plot(stats['loss_history'])
         plt.xlabel('iteration')
         plt.ylabel('training loss')
         plt.title('Training Loss history')
         plt.show()
```

```
Final training loss:  0.02950555626206818
```



1.6 Classify CIFAR-10

Do classification on the CIFAR-10 dataset.

```
In [30]: from data.data_utils import load_CIFAR10

         def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000):
             """
             Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
             it for the two-layer neural net classifier. These are the same steps as
             we used for the SVM, but condensed to a single function.
             """
             # Load the raw CIFAR-10 data
             cifar10_dir = './data/datasets/cifar-10-batches-py'
             X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

             # Subsample the data
             mask = list(range(num_training, num_training + num_validation))
             X_val = X_train[mask]
```

```
    y_val = y_train[mask]
    mask = list(range(num_training))
    X_train = X_train[mask]
    y_train = y_train[mask]
    mask = list(range(num_test))
    X_test = X_test[mask]
    y_test = y_test[mask]

    # Normalize the data: subtract the mean image
    mean_image = np.mean(X_train, axis=0)
    X_train -= mean_image
    X_val -= mean_image
    X_test -= mean_image

    # Reshape data to rows
    X_train = X_train.reshape(num_training, -1)
    X_val = X_val.reshape(num_validation, -1)
    X_test = X_test.reshape(num_test, -1)

    return X_train, y_train, X_val, y_val, X_test, y_test


# Invoke the above function to get our data.
X_train, y_train, X_val, y_val, X_test, y_test = get_CIFAR10_data()
print('Train data shape: ', X_train.shape)
print('Train labels shape: ', y_train.shape)
print('Validation data shape: ', X_val.shape)
print('Validation 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,)
Validation data shape:  (1000, 3072)
Validation labels shape:  (1000,)
Test data shape:  (1000, 3072)
Test labels shape:  (1000,)
```

1.6.1 Running SGD

If your implementation is correct, you should see a validation accuracy of around 15-18%.

```
In [40]: input_size = 32 * 32 * 3
         hidden_size = 50
         num_classes = 10
         net = TwoLayerNet(input_size, hidden_size, num_classes)
```

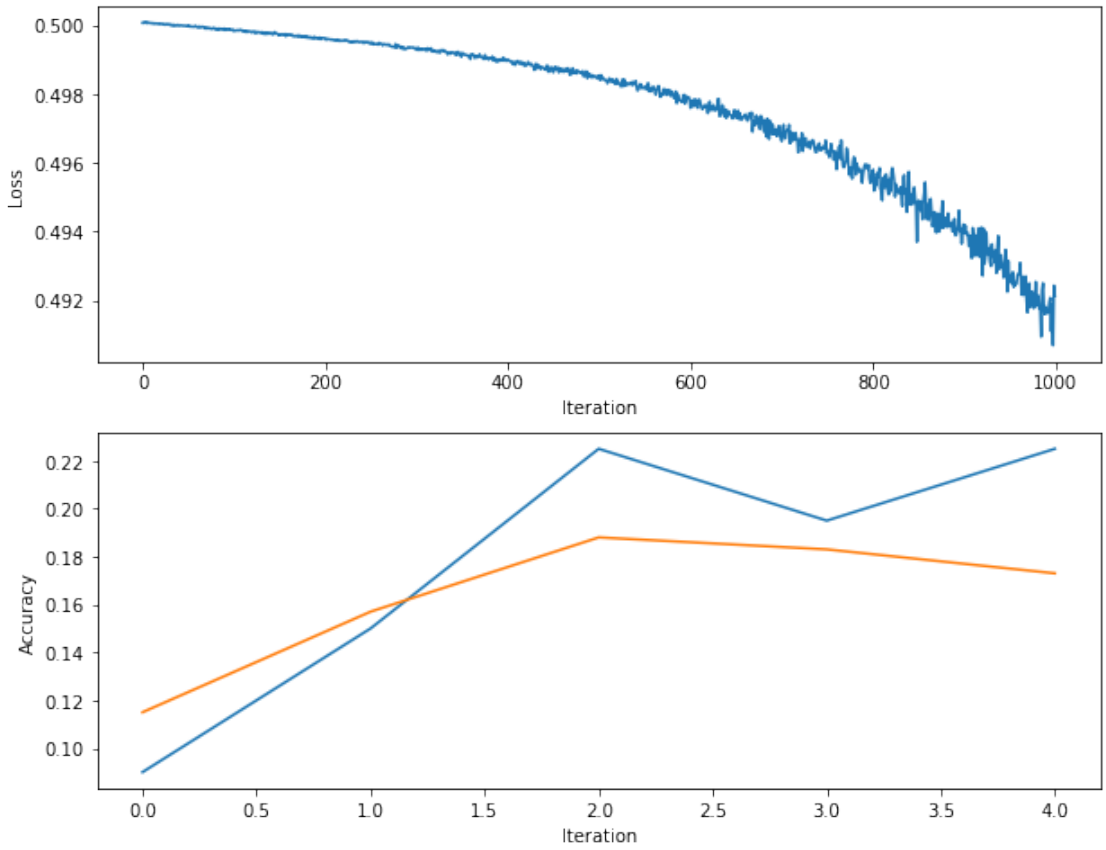
```
        # Train the network
        stats = net.train(X_train, y_train, X_val, y_val,
                    num_iters=1000, batch_size=200,
                    learning_rate=1e-5, learning_rate_decay=0.95,
                    reg=0.1, verbose=True)

        # Predict on the validation set
        val_acc = (net.predict(X_val) == y_val).mean()
        print('Validation accuracy: ', val_acc)

        # Save this net as the variable subopt_net for later comparison.
        subopt_net = net
        test_acc = (subopt_net.predict(X_test) == y_test).mean()
        print('Test accuracy (subopt_net): ', test_acc)
```

```
iteration 0 / 1000: loss 0.5000653841132241
iteration 100 / 1000: loss 0.4998690749422438
iteration 200 / 1000: loss 0.4996798178691218
iteration 300 / 1000: loss 0.4994397950516031
iteration 400 / 1000: loss 0.4991026153222729
iteration 500 / 1000: loss 0.49884677571338243
iteration 600 / 1000: loss 0.4981137879077918
iteration 700 / 1000: loss 0.49755140486802873
iteration 800 / 1000: loss 0.4965314440354824
iteration 900 / 1000: loss 0.49479052775342364
Validation accuracy:  0.173
Test accuracy (subopt_net):  0.186
```

```
In [32]: stats['train_acc_history']
```

```
Out[32]: [0.09, 0.15, 0.225, 0.195, 0.225]
```

```
In [33]: # Plot the loss function and train / validation accuracies
         plt.subplot(2, 1, 1)
         plt.plot(stats['loss_history'])
         plt.xlabel('Iteration')
         plt.ylabel('Loss')

         plt.subplot(2, 1, 2)
         plt.plot(stats['train_acc_history'], label='train')
         plt.plot(stats['val_acc_history'], label='val')
         plt.xlabel('Iteration')
         plt.ylabel('Accuracy')

         plt.show()
```



Questions:

The training accuracy isn't great. It seems even worse than simple KNN model, which is not as good as expected.

(1) What are some of the reasons why this is the case? Based on previous observations, please provide at least two possible reasons with justification.

(2) How should you fix the problems you identified in (1)?

Answers:

1. The first thing that comes to mind is that this is only a 2-Layer network, so low accuracy is to be expected since our dataset likely cannot be accurately handled with a low-powered neural network. Another reason could be that our hyperparams are not accurate; given that we only tested with a small set of possible hyperparameters, our neural network is limited.
2. Of course, with tuning, our issue of hyperparameter inaccuracy will go down. Additionally, we can increase the number of hidden layers, allowing for more complex data processing.

1.7 Optimize the neural network

Use the following part of the Jupyter notebook to optimize your hyperparameters on the validation set. Store your nets as best_net. To get the full credit of the neural nets, you should get at least **45%** accuracy on validation set.

Reminder: Think about whether you should retrain a new model from scratch every time your try a new set of hyperparameters.

In [34]:
```
best_net = None # store the best model into this

# ================================================================ #
# START YOUR CODE HERE:
# ================================================================ #
#   Optimize over your hyperparameters to arrive at the best neural
#   network.  You should be able to get over 45% validation accuracy.
#   For this part of the notebook, we will give credit based on the
#   accuracy you get.  Your score on this question will be multiplied by:
#      min(floor((X - 23%)) / %22, 1)
#   where if you get 50% or higher validation accuracy, you get full
#   points.
#
#   Note, you need to use the same network structure (keep hidden_size = 50)!
# ================================================================ #

# todo: optimal parameter search (you may use grid search by for-loops )
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
best_valid_acc = 0

# net = TwoLayerNet(input_size, hidden_size, num_classes)
# net = init_toy_model()
# Train the network and find best parameter:

opt_settings_str = ''
lrs = [1e-4, 1e-3]
decays = [0.8, 0.75, 0.6]
regularizations = [1, 2, 4]

for lr in lrs:
    for d in decays:
        for regz in regularizations:
            stats = net.train(X_train, y_train, X_val, y_val, num_iters=1000, batch_siz
            valid_acc = (net.predict(X_val) == y_val).mean()
            print('Validation accuracy: ', valid_acc)
            print('learning_rate: {}, iterations: {}, batch_sizes: {}, lrdecay: {}, reg
            if valid_acc > best_valid_acc:
                best_valid_acc = valid_acc
                best_net = net
                opt_settings_str = 'learning_rate: {}, iterations: {}, batch_sizes: {},


# ================================================================ #
```

```
            # END YOUR CODE HERE
            # ================================================================ #
            # Output your results
            print("== Best parameter settings ==")
            print(opt_settings_str)
            print("Best accuracy on validation set: {}".format(best_valid_acc))
```

```
iteration 0 / 1000: loss 0.491828519168669
iteration 100 / 1000: loss 0.4593743449019668
iteration 200 / 1000: loss 0.4480651160670165
iteration 300 / 1000: loss 0.44335528959711595
iteration 400 / 1000: loss 0.4452274893069379
iteration 500 / 1000: loss 0.43910652028271485
iteration 600 / 1000: loss 0.4344220339248548
iteration 700 / 1000: loss 0.4354469254224735
iteration 800 / 1000: loss 0.4376013062916228
iteration 900 / 1000: loss 0.428242941065242
Validation accuracy:  0.264
learning_rate: 0.0001, iterations: 1000, batch_sizes: 500, lrdecay: 0.8, regularization: 1
iteration 0 / 1000: loss 0.4340652324416934
iteration 100 / 1000: loss 0.4306596476220053
iteration 200 / 1000: loss 0.42239392452453467
iteration 300 / 1000: loss 0.4154876727613203
iteration 400 / 1000: loss 0.41857104571277093
iteration 500 / 1000: loss 0.4154808619073914
iteration 600 / 1000: loss 0.41937266564593945
iteration 700 / 1000: loss 0.4196167011193107
iteration 800 / 1000: loss 0.4126504399414269
iteration 900 / 1000: loss 0.4172734956739185
Validation accuracy:  0.307
learning_rate: 0.0001, iterations: 1000, batch_sizes: 500, lrdecay: 0.8, regularization: 2
iteration 0 / 1000: loss 0.42541767522510265
iteration 100 / 1000: loss 0.41454760918284805
iteration 200 / 1000: loss 0.40817473528668957
iteration 300 / 1000: loss 0.4121884993652861
iteration 400 / 1000: loss 0.41326292821382576
iteration 500 / 1000: loss 0.4085735662435541
iteration 600 / 1000: loss 0.41198020737559043
iteration 700 / 1000: loss 0.40466962529437694
iteration 800 / 1000: loss 0.40684672049226916
iteration 900 / 1000: loss 0.40830316077315343
Validation accuracy:  0.337
learning_rate: 0.0001, iterations: 1000, batch_sizes: 500, lrdecay: 0.8, regularization: 4
iteration 0 / 1000: loss 0.40385631596372557
iteration 100 / 1000: loss 0.39553689152016397
iteration 200 / 1000: loss 0.40809878914139663
iteration 300 / 1000: loss 0.39564180747654276
iteration 400 / 1000: loss 0.4040007266188983
```

```
iteration 500 / 1000: loss 4.0020189655775834
iteration 600 / 1000: loss 3.9857880112706795
iteration 700 / 1000: loss 4.0049433828780739
iteration 800 / 1000: loss 3.9986725078900686
iteration 900 / 1000: loss 4.0429988779609255
Validation accuracy:  0.359
learning_rate: 0.0001, iterations: 1000, batch_sizes: 500, lrdecay: 0.75, regularization: 1
iteration 0 / 1000: loss 4.0026028110560171
iteration 100 / 1000: loss 4.0058554045957627
iteration 200 / 1000: loss 3.9806453983719103
iteration 300 / 1000: loss 3.9582881424568783
iteration 400 / 1000: loss 3.9993962018531759
iteration 500 / 1000: loss 4.0010262521148996
iteration 600 / 1000: loss 3.9958181890379916
iteration 700 / 1000: loss 3.9947632080731116
iteration 800 / 1000: loss 3.9129350845641403
iteration 900 / 1000: loss 3.9922696777303684
Validation accuracy:  0.375
learning_rate: 0.0001, iterations: 1000, batch_sizes: 500, lrdecay: 0.75, regularization: 2
iteration 0 / 1000: loss 4.0090515871947538
iteration 100 / 1000: loss 4.0055587388220986
iteration 200 / 1000: loss 3.9312697122879464
iteration 300 / 1000: loss 3.9977079524053877
iteration 400 / 1000: loss 4.0079274296376304
iteration 500 / 1000: loss 3.9943891282957726
iteration 600 / 1000: loss 4.0528261145629546
iteration 700 / 1000: loss 4.0779912107005195
iteration 800 / 1000: loss 4.0015813000859951
iteration 900 / 1000: loss 4.0269052270689804
Validation accuracy:  0.389
learning_rate: 0.0001, iterations: 1000, batch_sizes: 500, lrdecay: 0.75, regularization: 4
iteration 0 / 1000: loss 3.9152767988323434
iteration 100 / 1000: loss 3.8850913545049704
iteration 200 / 1000: loss 3.93466663692234
iteration 300 / 1000: loss 3.8843112784972863
iteration 400 / 1000: loss 3.9002898305056159
iteration 500 / 1000: loss 3.8994587314742592
iteration 600 / 1000: loss 3.8825700638736951
iteration 700 / 1000: loss 3.8860121914990725
iteration 800 / 1000: loss 3.9074373916264726
iteration 900 / 1000: loss 3.8850524936569003
Validation accuracy:  0.405
learning_rate: 0.0001, iterations: 1000, batch_sizes: 500, lrdecay: 0.6, regularization: 1
iteration 0 / 1000: loss 3.9028015185169188
iteration 100 / 1000: loss 3.8992813120306964 4
iteration 200 / 1000: loss 3.9038844873086486
iteration 300 / 1000: loss 3.8476176183982563
iteration 400 / 1000: loss 3.8488054412851075
```

```
iteration 500 / 1000: loss 0.3843954747814342
iteration 600 / 1000: loss 0.3916274156343006
iteration 700 / 1000: loss 0.3832640840674542
iteration 800 / 1000: loss 0.395536014188009
iteration 900 / 1000: loss 0.3899887218715062
Validation accuracy:  0.416
learning_rate: 0.0001, iterations: 1000, batch_sizes: 500, lrdecay: 0.6, regularization: 2
iteration 0 / 1000: loss 0.39655607477874205
iteration 100 / 1000: loss 0.40032458804960136
iteration 200 / 1000: loss 0.4005033239302925
iteration 300 / 1000: loss 0.3957050390554905
iteration 400 / 1000: loss 0.401247187267484
iteration 500 / 1000: loss 0.4040651440203802
iteration 600 / 1000: loss 0.3961496951348371
iteration 700 / 1000: loss 0.3961830514682346
iteration 800 / 1000: loss 0.39390753937566975
iteration 900 / 1000: loss 0.400503938138193
Validation accuracy:  0.424
learning_rate: 0.0001, iterations: 1000, batch_sizes: 500, lrdecay: 0.6, regularization: 4
iteration 0 / 1000: loss 0.39509424094899426
iteration 100 / 1000: loss 0.38422287282679674
iteration 200 / 1000: loss 0.39042892398391316
iteration 300 / 1000: loss 0.36675894740367404
iteration 400 / 1000: loss 0.37080681040512925
iteration 500 / 1000: loss 0.37788172958659516
iteration 600 / 1000: loss 0.3724291575982607
iteration 700 / 1000: loss 0.3633699006934623
iteration 800 / 1000: loss 0.36048350399392637
iteration 900 / 1000: loss 0.3707601036243043
Validation accuracy:  0.476
learning_rate: 0.001, iterations: 1000, batch_sizes: 500, lrdecay: 0.8, regularization: 1
iteration 0 / 1000: loss 0.3688431877691851
iteration 100 / 1000: loss 0.3899867335338655
iteration 200 / 1000: loss 0.38068339277648333
iteration 300 / 1000: loss 0.37525616397667666
iteration 400 / 1000: loss 0.3706662612130924
iteration 500 / 1000: loss 0.3742766362521676
iteration 600 / 1000: loss 0.38262336044069034
iteration 700 / 1000: loss 0.3800716725313875
iteration 800 / 1000: loss 0.37358215692614993
iteration 900 / 1000: loss 0.37120669073102025
Validation accuracy:  0.474
learning_rate: 0.001, iterations: 1000, batch_sizes: 500, lrdecay: 0.8, regularization: 2
iteration 0 / 1000: loss 0.39575525869352207
iteration 100 / 1000: loss 0.3928015609552376
iteration 200 / 1000: loss 0.38916966122415503
iteration 300 / 1000: loss 0.3797780923172795
iteration 400 / 1000: loss 0.3903785101935017
```

```
iteration 500 / 1000: loss 0.38711647185236386
iteration 600 / 1000: loss 0.3958934468662587
iteration 700 / 1000: loss 0.38876964215132825
iteration 800 / 1000: loss 0.3836799607901521
iteration 900 / 1000: loss 0.39241119383675616
Validation accuracy:  0.458
learning_rate: 0.001, iterations: 1000, batch_sizes: 500, lrdecay: 0.8, regularization: 4
iteration 0 / 1000: loss 0.36340771337618044
iteration 100 / 1000: loss 0.37142853822435845
iteration 200 / 1000: loss 0.37187505056103887
iteration 300 / 1000: loss 0.3646539629783366
iteration 400 / 1000: loss 0.3673818393854741
iteration 500 / 1000: loss 0.3601854811160642
iteration 600 / 1000: loss 0.35799604936668106
iteration 700 / 1000: loss 0.36772029079358937
iteration 800 / 1000: loss 0.36599811158486495
iteration 900 / 1000: loss 0.3643512630912791
Validation accuracy:  0.48
learning_rate: 0.001, iterations: 1000, batch_sizes: 500, lrdecay: 0.75, regularization: 1
iteration 0 / 1000: loss 0.3771556355523204
iteration 100 / 1000: loss 0.36430739663007317
iteration 200 / 1000: loss 0.37719774454599436
iteration 300 / 1000: loss 0.36174202905625413
iteration 400 / 1000: loss 0.3771642404024748
iteration 500 / 1000: loss 0.37721489889045107
iteration 600 / 1000: loss 0.3774499236730192
iteration 700 / 1000: loss 0.3703718348821798
iteration 800 / 1000: loss 0.36762177220887654
iteration 900 / 1000: loss 0.3830401480792569
Validation accuracy:  0.483
learning_rate: 0.001, iterations: 1000, batch_sizes: 500, lrdecay: 0.75, regularization: 2
iteration 0 / 1000: loss 0.3867937184721809
iteration 100 / 1000: loss 0.38203997973860415
iteration 200 / 1000: loss 0.390956099625095
iteration 300 / 1000: loss 0.3956644182966086
iteration 400 / 1000: loss 0.3874746849416148
iteration 500 / 1000: loss 0.38488345004615176
iteration 600 / 1000: loss 0.39746644989955304
iteration 700 / 1000: loss 0.390113385197794
iteration 800 / 1000: loss 0.3909561111766971
iteration 900 / 1000: loss 0.39243666230239665
Validation accuracy:  0.475
learning_rate: 0.001, iterations: 1000, batch_sizes: 500, lrdecay: 0.75, regularization: 4
iteration 0 / 1000: loss 0.3658066954949327
iteration 100 / 1000: loss 0.3626337811809807
iteration 200 / 1000: loss 0.35084285352458483
iteration 300 / 1000: loss 0.355869653546313
iteration 400 / 1000: loss 0.36383910950187764
```

```
iteration 500 / 1000: loss 0.35283204614912134
iteration 600 / 1000: loss 0.35836518538838724
iteration 700 / 1000: loss 0.3638619927574204
iteration 800 / 1000: loss 0.3542773530403485
iteration 900 / 1000: loss 0.35525510994313925
Validation accuracy:  0.489
learning_rate: 0.001, iterations: 1000, batch_sizes: 500, lrdecay: 0.6, regularization: 1
iteration 0 / 1000: loss 0.3599081502883241
iteration 100 / 1000: loss 0.3693910462572361
iteration 200 / 1000: loss 0.36494867355108446
iteration 300 / 1000: loss 0.3733823385804047
iteration 400 / 1000: loss 0.36671532119959327
iteration 500 / 1000: loss 0.3718087493182603
iteration 600 / 1000: loss 0.37440713756780397
iteration 700 / 1000: loss 0.37778253025779024
iteration 800 / 1000: loss 0.3756751103797165
iteration 900 / 1000: loss 0.37839581396212163
Validation accuracy:  0.483
learning_rate: 0.001, iterations: 1000, batch_sizes: 500, lrdecay: 0.6, regularization: 2
iteration 0 / 1000: loss 0.38745729727112105
iteration 100 / 1000: loss 0.38286401747762755
iteration 200 / 1000: loss 0.39013155795851434
iteration 300 / 1000: loss 0.38659207567764176
iteration 400 / 1000: loss 0.3945722207389524
iteration 500 / 1000: loss 0.3903150274614551
iteration 600 / 1000: loss 0.3892858420266587
iteration 700 / 1000: loss 0.39455983404110007
iteration 800 / 1000: loss 0.39312300155899255
iteration 900 / 1000: loss 0.3868618911507201
Validation accuracy:  0.48
learning_rate: 0.001, iterations: 1000, batch_sizes: 500, lrdecay: 0.6, regularization: 4
== Best parameter settings ==
learning_rate: 0.001, iterations: 1000, batch_sizes: 500, lrdecay: 0.6, regularization: 1
Best accuracy on validation set: 0.489
```

Quesions

(1) What is your best parameter settings? (Output from the previous cell)

(2) What parameters did you tune? How are they changing the performance of nerural network? You can discuss any observations from the optimization.

Answers

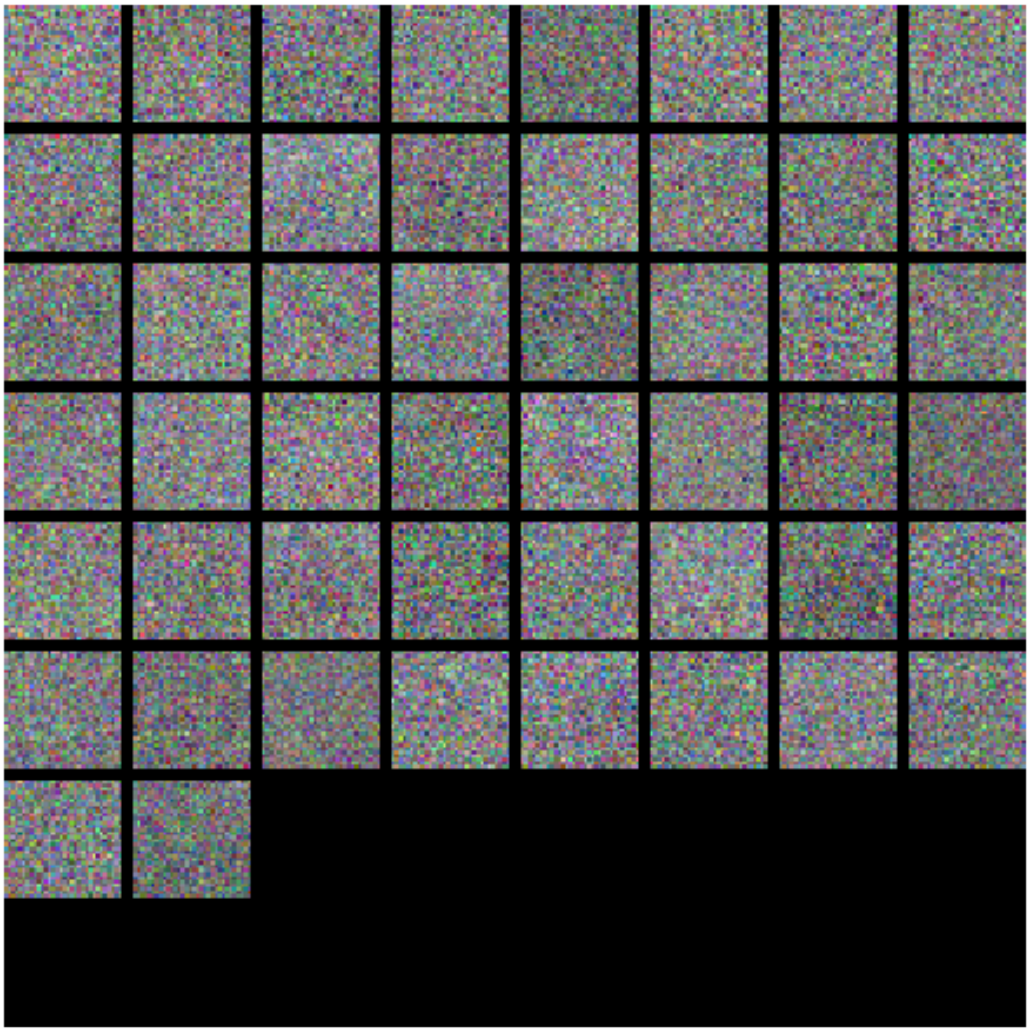
1. Based on the above, the ideal parameters for this subset are: `learning_rate: 0.001, iterations: 1000, batch_sizes: 500, lrdecay: 0.6, regularization: 1`
2. Given that the params that result in the greatest change in response to tuning were the learning rate, learning rate decay, and regularization, I chose to optimize on those parameters, using a few subsets of values for each.

1.8 Visualize the weights of your neural networks

In [42]:
```
from data.vis_utils import visualize_grid

# Visualize the weights of the network

def show_net_weights(net):
    W1 = net.params['W1']
    W1 = W1.T.reshape(32, 32, 3, -1).transpose(3, 0, 1, 2)
    plt.imshow(visualize_grid(W1, padding=3).astype('uint8'))
    plt.gca().axis('off')
    plt.show()

#Had to re-run the subopt one to get the right params, since I used the net obj kinda r
print("Subpopt")
print(subopt_net.params['W1'])
print("Opt")
print(best_net.params['W1'])
show_net_weights(subopt_net)
show_net_weights(best_net)
```

```
Subpopt
[[-7.40695078e-05 -3.49500350e-05 -4.19172611e-05 ...  7.22647179e-06
  -1.15812816e-06 -3.25142389e-05]
 [ 5.35857596e-07 -2.15522067e-06  6.72090349e-05 ...  2.62624771e-06
   6.89106405e-05  2.66278742e-05]
 [ 3.56652442e-05  1.78354344e-04  1.20388108e-05 ...  7.67448887e-05
  -1.49414583e-04  7.05453706e-05]
 ...
 [ 9.33278981e-06  9.96739146e-05 -6.16069902e-05 ... -2.63987006e-05
   7.74001900e-05  1.59241077e-04]
 [-1.03445726e-04  5.77482886e-05  1.20718286e-04 ...  2.07328612e-05
   2.96483766e-05 -6.06061271e-05]
 [-1.41636813e-04  1.29133814e-04  7.17963843e-05 ...  2.49462398e-06
  -7.40524730e-05  1.36491449e-05]]
Opt
[[-3.80722349e-04 -4.88864580e-04 -2.54289992e-04 ...  1.32588075e-04
   1.80212506e-04  4.14105981e-05]
 [-6.76622138e-04 -5.30150255e-04 -8.52004578e-05 ...  1.95883085e-04
   7.29646147e-05  4.45374032e-04]
 [-1.94392771e-04 -1.38243373e-04 -2.72851410e-04 ...  2.93340172e-04
   1.20403374e-04  1.43483987e-04]
 ...
 [-9.88778318e-05 -5.11932411e-05  1.07321192e-04 ...  3.91029391e-05
  -3.77051506e-04 -6.95368976e-04]
 [-5.50675183e-05  7.72192394e-05  1.74916413e-04 ... -6.17585636e-04
```

```
 -5.44980610e-04 -3.38554647e-04]
[ 2.69372922e-04  1.74520235e-04 -5.83476635e-04 ...  1.82538235e-04
  3.01571298e-05 -7.28183748e-05]]
```



Questions:

What differences do you see in the weights between the suboptimal net and the best net you arrived at? What do the weights in neural networks probably learn after training?

Answer:

The weights are far less noisy, and the features are more distinct in our optimized neural network. The weights likely "learned" to distinguish the different attributes in the CIFAR dataset.

1.9 Evaluate on test set

```
In [43]: test_acc = (best_net.predict(X_test) == y_test).mean()
         print('Test accuracy (best_net): ', test_acc)

Test accuracy (best_net):  0.465
```

Questions:

(1) What is your test accuracy by using the best NN you have got? How much does the performance increase compared with kNN? Why can neural networks perform better than kNN?

(2) Do you have any other ideas or suggestions to further improve the performance of neural networks other than the parameters you have tried in the homework?

Answers:

1. The test accuracy is 46.5%. It's a lot better than out kNN approach. That had an error rate of 71.8%, so an accuracy of 28.2%. kNN, while a quick training algo, loses out overall. kNN is limited to the weights and data in a direct way: where it is measuring the distance between features of datapoints. Neural nets, on the other hand, have multiple hidden layers whose outputs cascade into others. This allows for a finer granularity in making distinctions between features and, therefore, increase the accuracy of classifying.
2. Only thing I can think of is what I said before, which is more layers to leverage what I said in 1.

1.10 Bonus Question: Change MSE Loss to Cross Entropy Loss

This is a bonus question. If you finish this (cross entropy loss) correctly, you will get **up to 10 points** (add up to your HW3 score).

Note: From grading policy of this course, your maximum points from homework are still 25 out of 100, but you can use the bonus question to make up other deduction of other assignments.

Pass output scores in networks from forward pass into softmax function. The softmax function is defined as,

$$p_j = \sigma(z_j) = \frac{e^{z_j}}{\sum_{c=1}^{C} e^{z_c}}$$

After softmax, the scores can be considered as probability of j-th class.

The cross entropy loss is defined as,

$$L = L_{\text{CE}} + L_{reg} = \frac{1}{N}\sum_{i=1}^{N} \log\left(p_{i,j}\right) + \frac{\lambda}{2}\left(||W_1||^2 + ||W_2||^2\right)$$

To take derivative of this loss, you will get the gradient as,

$$\frac{\partial L_{\text{CE}}}{\partial o_i} = p_i - y_i$$

More details about multi-class cross entropy loss, please check http://cs231n.github.io/linear-classify/ and more explanation about the derivative of cross entropy.

Change the loss from MSE to cross entropy, you only need to change you `MSE_loss(x,y)` in `TwoLayerNet.loss()` function to `softmax_loss(x,y)`.

Now you are free to use any code to show your results of the two-layer networks with newly-implemented cross entropy loss. You can use code from previous cells.

```
In [ ]: # Start training your networks and show your results
        # ================================================================ #
        # START YOUR CODE HERE:
        # ================================================================ #
        pass
        # ================================================================ #
        # END YOUR CODE HERE
        # ================================================================ #
```

1.11 End of Homework 3, Part 2 :)

After you've finished both parts the homework, please print out the both of the entire `ipynb` notebooks and `py` files into one PDF file. Make sure you include the output of code cells and answers for questions. Prepare submit it to GradeScope. Do not include any dataset in your submission.