CS7643: Deep Learning
Spring 2019
Homework 1, Solutions

Alexis DUROCHER (903336265 adurocher3)

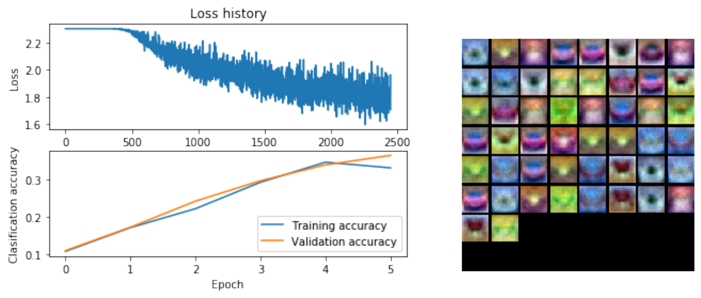
February 17, 2019

1 Part I

In this first part, I implemented the 'vanilla' version of a two-layer NNet and a CNN. Including the forward and backpropagation passes. Classifiers will be trained on the CIFAR10 dataset. I will now discuss their relative performance, with respect to the hyperparameters.

	vanilla SGD	momentum SGD	RMSprop
final loss	0.941	0.494	0.439

Table 1: Optimizer loss impact, after 1 epoch



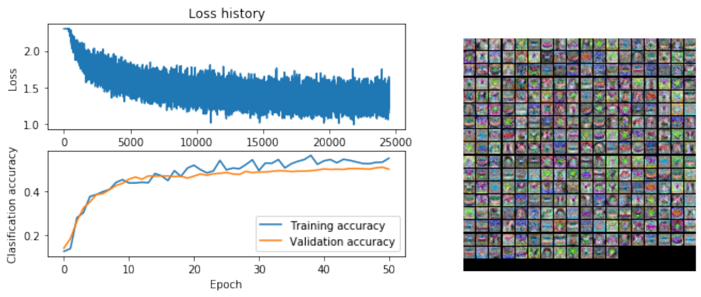


Figure 1: Two layer NN performance and weight visualization pre fine-tuning

Figure 2: Two layer NN performance and weight visualization post fine-tuning

1. The two-layer NNet includes a non-linear activation function (ReLU, here). To compare the different optimizers, we will keep the default other hyperparameters ($reg = 0.001$ and $lr = 1e^{-1}$, $batchsize = 100$) and compare the final loss on the evaluation dataset, after 1 epoch. From Table 1, we can see that RMSprop optimizer is the best at minimizing the loss here. .

   A first attempt at training our network with default hyperparameters ($H = 50$, $numepoch = 5$, $reg = 1$) resulted in a clear under-fitting (see Figure1). The evaluation learning curve is constantly increasing, indicating that the loss didn't converge yet: The model is either too simple, or the regularization is too strong; a fine-tuning is required.

   My solution was: $H = 300$, $numepoch = 50$, $reg = 0.001$, $lr = 1e^{-5}$. Explanations: Increase the complexity of the model (higher H). Incerase training iterations (higher $numepoch$) to give more SGD iterations for the network to minimize the loss. Decrease the regularization (lower reg) to reduce weights constraints. Note that I kept a low learning rate ($lr = 1e^{-5}$) to avoid big steps in SGD updates, thus converging slowly but avoiding misleading steps. **This solutions resulted in a better final training, evaluation (see Figure 2) and test accuracy** (0.513). *Note also the relatively great interpretability of the weight visualization (recognize CIFAR class pattern).*

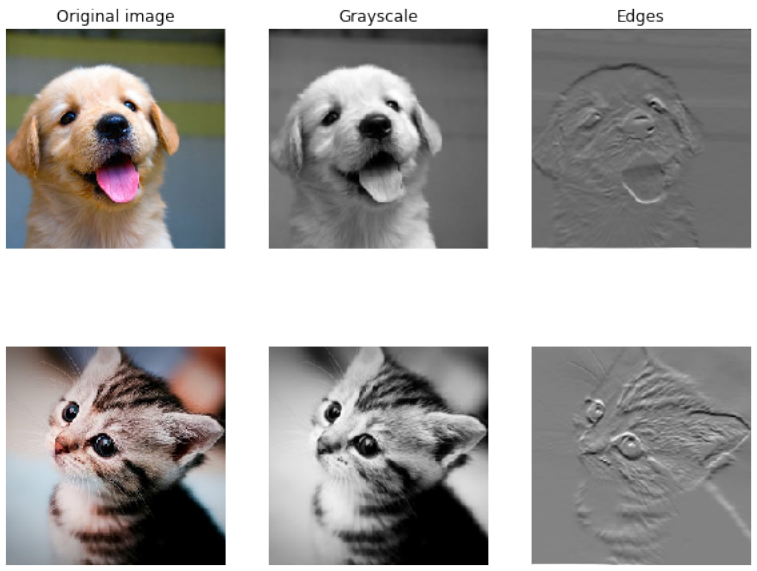


Figure 3: Different convolution kernel applications

2. In this subsection, I implemented a modular version for fully-connected layers and convolution layers. Including "sandwich layers" which group either convolution, relu and pool operations or affine and ReLU operations. An example of vanilla convolution operation can be seen in Figure 3.

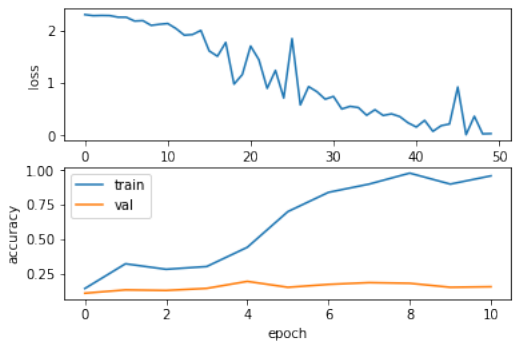


Figure 4: Convnet performance after training on 50 samples

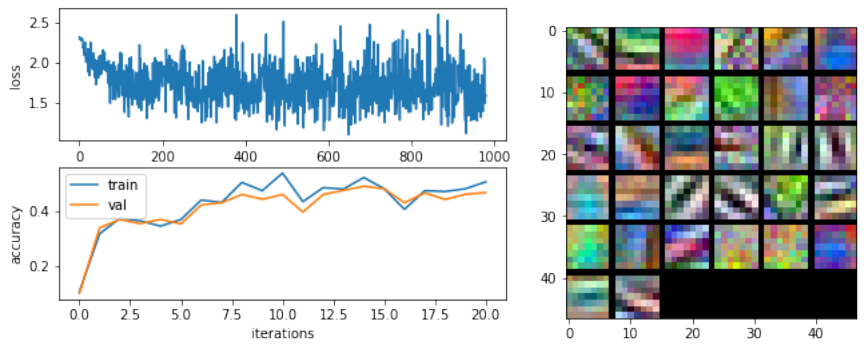


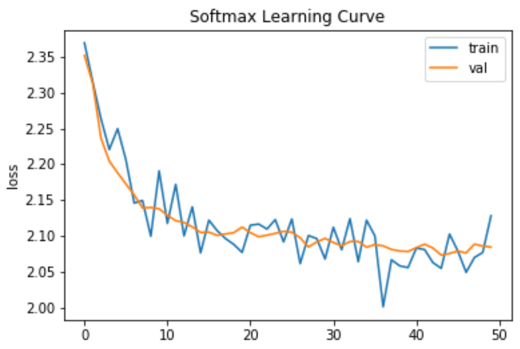
Figure 5: Convnet performance and weight visualization using complete training dataset

3. In this subsection, I used the modular version of convolution layers and fully connected layers implemented previously to train a complete a convnet on CIFAR 10 dataset. A first attempt with default parameters ($reg = 0.001$, $lr = 0.0001$, $batchsize = 10$, $numepochs = 10$) led to serious over-fitting on a very small dataset (50 samples): the validation learning curve keep being constant while the training curve increase. This is a sanity check to ensure our model can over-fit on very small datasets. (see Figure 4).

   Then, we consider the entire training and evaluation datasets, to avoid over-fitting (see Figure 5) with appropriate regularization ($reg = 0.001$) and learning rate here ($lr = 0.0001$) (see Figure 5). The best evaluation accuracy obtained here is 0.489. *Note, the weight visualization are, again, interpretable. The visible patterns correspond to low-level features, extracted by convolution operation*

2 Part II

In this second part, we will use the Pytorch framework to implement a Softmax calssifier, the two-layer NNet and the Convolutional NNet.



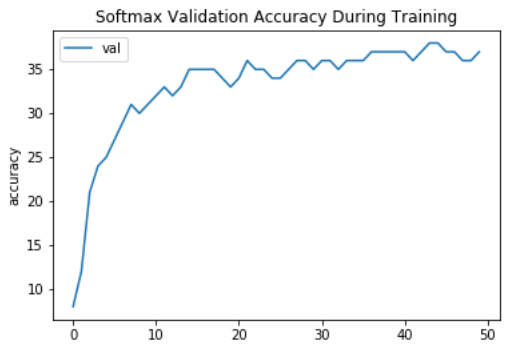


Figure 6: Softmax performance after fine-tuning

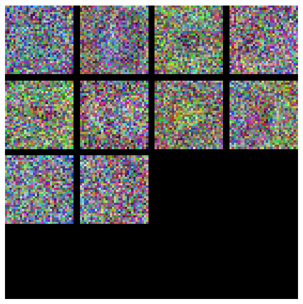


Figure 7: Softmax weight visualization

1. Softmax classifier with Pytorch. To reach such a validation accuracy (38%, see Figure 6) and testing accuracy(37%), I used the following parameters: $epoch = 2$, $weight - decay = 0.001$, $momentum = 0.9$ $batch - size = 200$ and $lr = 0.0001$.

   Explanations: Added a weight decay (regularization factor) to reduce overfitting by penalizing high weight values in the loss. Reduced the batch-size so each udpate is more representative of the model's performance on the actual batch samples. Finally I reduced the learning rate to avoid misleading steps while adding a momentum to use previous gradient value in the SGD. *Note that the weight visualization is not interpretable here (see Figure 7). This points out the fact that interpretability is not always linked to more robust models. Only learning curves give a trustful numerical interpretation of our network performance.*

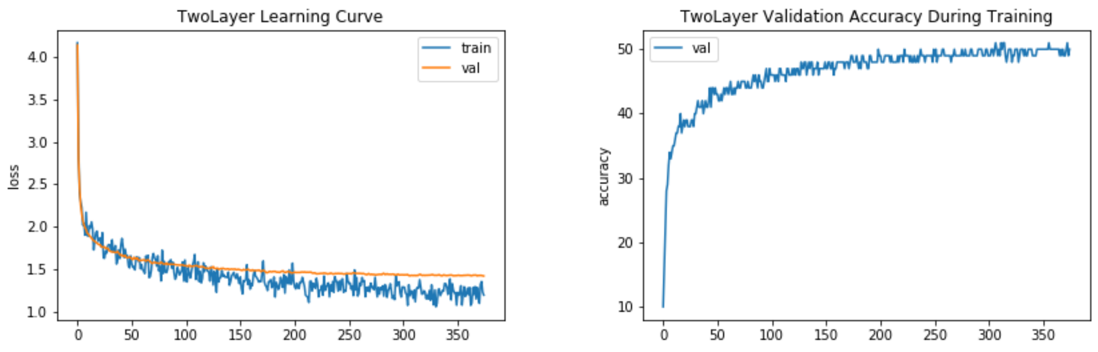


Figure 8: Two-layer NN performance after fine-tuning

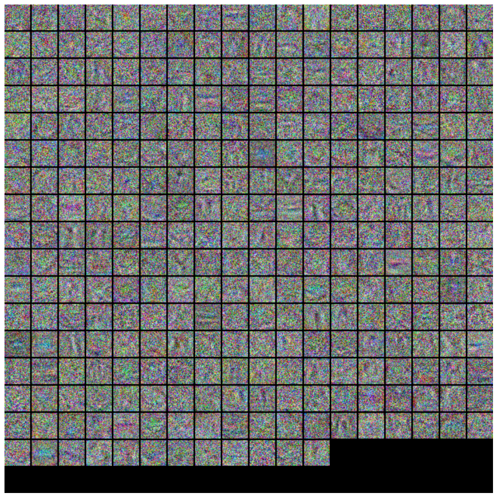


Figure 9: Two-layer NN weight visualization

2. Two-layer classifier with Pytorch. To reach such a validation accuracy (54%, see Figure 8) and testing accuracy(50%), I used the following parameters: $hidden-dim = 300$, $epoch = 15$, $weight-decay = 0.001$, $momentum = 0.9$ $batch-size = 200$ and $lr = 0.0001$.

   Explanations: Higher hidden dim to increase the complexity of the model. Added a weight decay (regularization factor) to reduce over-fitting by penalizing high weight values in the loss. Reduced the batch-size so each update is more representative of the model's performance on the actual batch samples. I reduced the learning rate to avoid misleading steps while adding a momentum to use previous gradient value in the SGD. I added a learning rate decay ($lrdecay = 0.95$) using Pytorch *Scheduler* to dynamically reduce update steps after each epoch. *Note that the weight visualization is not interpretable here (see Figure 9).*

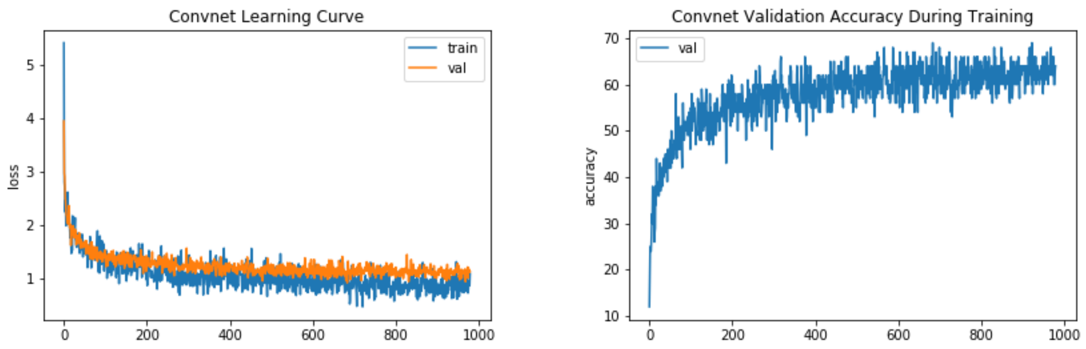


Figure 10: Two-layer NN performance after fine-tuning

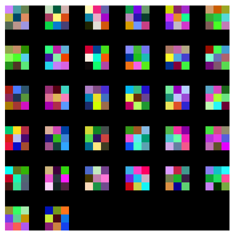


Figure 11: Two-layer NN weight visualization

3. Convnet classifier with Pytorch. To reach such a validation accuracy (69%, see Figure 10) and testing accuracy(62%), I used the following parameters: $hidden-dim = 32$, $kernel-size = 3$, $epoch = 10$, $weight - decay = 0.0001$, $momentum = 0.9$ $batch - size = 50$ and $lr = 0.0001$.

   Explanations: Higher hidden dim to increase the complexity of the model. *Note that this H is smaller than H for the two-layers NN since each convolution kernel extract informative features for pattern recognition in image data, thus reducing the need for a high hidden dimension. However, we will see later (c.f mymodel) that increasing the hidden dimension and adding other layers will result in a better accuracy, but it was not required in this first convnet.* Added a weight decay (regularization factor) to reduce over-fitting by penalizing high weight values in the loss that I reduced progressively since the model didn't over-fit. Smaller batch-size so each update is more representative of the model's performance on the actual batch samples. Smaller learning rate to avoid misleading steps while adding a momentum to use previous gradient value in the SGD. Added a learning rate decay ($lrdecay = 0.95$) using Pytorch *Scheduler* to dynamically reduce update steps after each epoch. Increased the kernel size (3x3) to extract mid-level informative features given that input dimension is relatively small (32x32) *Note that the weight visualization is not very interpretable here due to the very small kernel size (see Figure 11).*

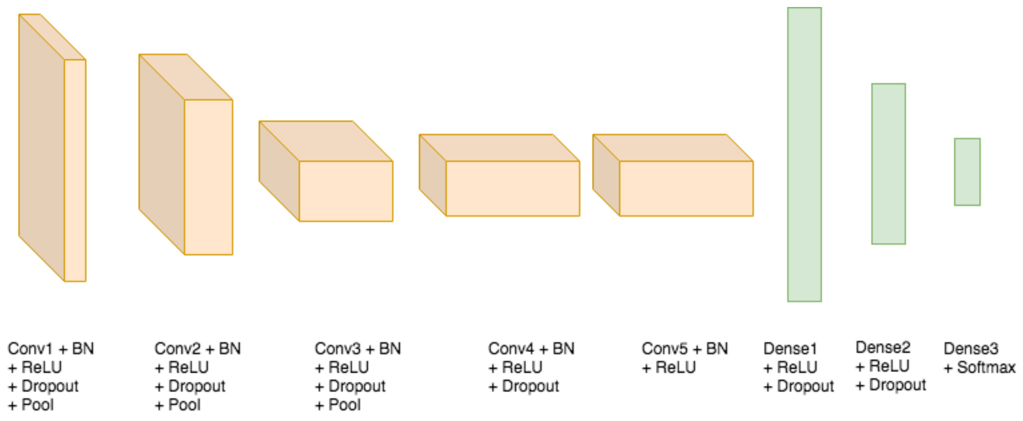


Figure 12: MyModel Convnet architecture

4. Mymodel Convnet classifier with Pytorch. Now, the objective is to design a Convnet architecture which will have the highest performance on CIFAR 10 dataset. **On the test dataset for the submission, my network showed 77% accuracy**.

   I will use the same hyperparameters as in the previous convnet. However, I used a completely new - more complex - architecture with a different optimizer: ADAM to maintain a per-parameter learning rate that improves performance on problems with sparse gradients - image recognition, here - also based on the average of recent magnitudes of the gradients.

   The new architecture can be divided in two parts. First, it includes different levels of convolution layers to extract image features (from low-level to high-level). Each convolution step includes a batch-norm and a non-linear activation function. Then three fully connected layers and a softmax operation for the classification. *Architecture: 5 convolution layers (5x5x32), (5x5x64), (3x3x128), (3x3x256), (3x3x256), 3 dense layers (2304x2304) (2304x1152) (1152x10)* (see Figure 12).

   Note that I used a very low learning-rate, added Dropout, and used different kernel size in this architecture. To illustrate the relative impact of this hyper paramater we will train the MyModel NNetwork on 3 epochs with different learning-rate; with or without Dropout and with different kernel sizes.

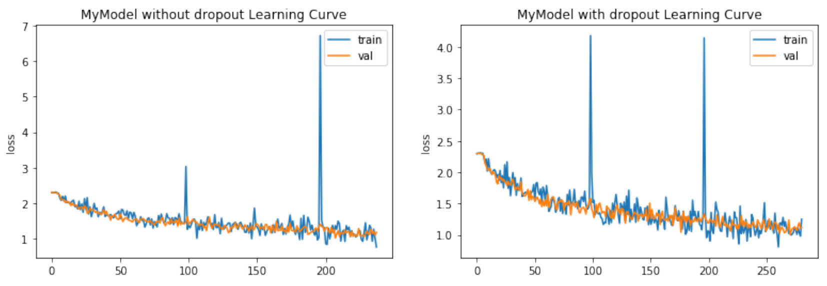


Figure 13: Dropout impact on loss minimization

Interpretation Figure 13: Adding Dropout helps the NNetwork to generalize better since it sets some neurons (with a given probabilities) to be null thus preventing complex co-adaptations on the training data (over-fitting). Here we can see that the validation curve without dropout already reached a "plateau" at epoch 3: indicating further over-fitting. Note that this effect could be emphasized with more epochs (>10).

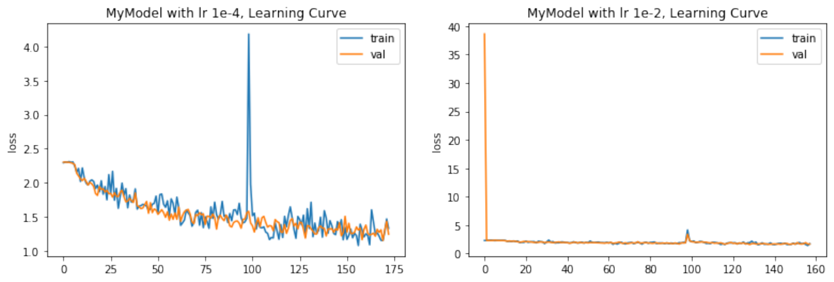


Figure 14: Learning rate impact on loss minimization

Interpretation Figure 14: The learning-rate corresponds to the "size of the step" taken at each parameters' update during the optimization. Since the function to approximate is complex and I used small batch sizes, it is important to take small steps to give the Network enough freedom of gradient directions to converge towards the global minima. A higher learning rate would result in a faster training phase but a lower final accuracy (the loss is stuck in a local minima). (see Figure 14).

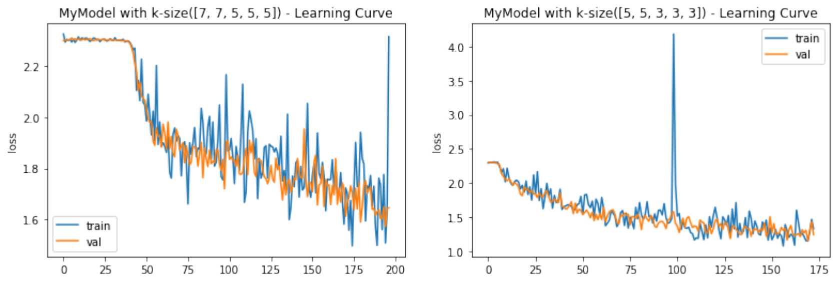


Figure 15: Kernel size impact on loss minimization

Interpretation Figure 15: Finally the kernel size will impact the size of the features the Network will learn at each layer. Here, given the small inputs (32x32) it is preferred to use small kernel sizes (5x5) or (3x3) instead of bigger ones (7x7). Also smaller kernel sizes tend to be easier to learn (less parameters) and provide more informative features - when combined - than bigger ones in convolutional NNets.

two_layer_net

February 10, 2019

1 Implementing a Neural Network

In this exercise we will develop a neural network with fully-connected layers to perform classification, and test it out on the CIFAR-10 dataset.

```
In [1]: # A bit of setup

        import numpy as np
        import matplotlib.pyplot as plt

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

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

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

The neural network parameters will be stored in a dictionary (`model` below), where the keys are the parameter names and the values are numpy arrays. Below, we initialize toy data and a toy model that we will use to verify your implementations.

```
In [2]: # Create some toy data to check your implementations
        input_size = 4
        hidden_size = 10
        num_classes = 3
        num_inputs = 5

        def init_toy_model():
            model = {}
            model['W1'] = np.linspace(-0.2, 0.6, num=input_size*hidden_size).reshape(input_size,
            model['b1'] = np.linspace(-0.3, 0.7, num=hidden_size)
            model['W2'] = np.linspace(-0.4, 0.1, num=hidden_size*num_classes).reshape(hidden_siz
            model['b2'] = np.linspace(-0.5, 0.9, num=num_classes)
            return model

        def init_toy_data():
            X = np.linspace(-0.2, 0.5, num=num_inputs*input_size).reshape(num_inputs, input_size
            y = np.array([0, 1, 2, 2, 1])
            return X, y

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

2 Forward pass: compute scores

Open the file `cs231n/classifiers/neural_net.py` and look at the function `two_layer_net`. This function is very similar to the loss functions you have written for the Softmax exercise in HW0: It takes the data and weights and computes the class scores, the loss, and the gradients on the parameters.

Implement the first part of the forward pass which uses the weights and biases to compute the scores for all inputs.

```
In [3]: from cs231n.classifiers.neural_net import two_layer_net

        scores = two_layer_net(X, model)
        print(scores)
        correct_scores = [[-0.5328368, 0.20031504, 0.93346689],
         [-0.59412164, 0.15498488, 0.9040914 ],
         [-0.67658362, 0.08978957, 0.85616275],
         [-0.77092643, 0.01339997, 0.79772637],
         [-0.89110401, -0.08754544, 0.71601312]]

        # the difference should be very small. We get 3e-8
        print('Difference between your scores and correct scores:')
        print(np.sum(np.abs(scores - correct_scores)))
```

```
[[-0.5328368   0.20031504  0.93346689]
 [-0.59412164  0.15498488  0.9040914 ]
 [-0.67658362  0.08978957  0.85616275]
 [-0.77092643  0.01339997  0.79772637]
 [-0.89110401 -0.08754544  0.71601312]]
Difference between your scores and correct scores:
3.848682278081994e-08
```

3 Forward pass: compute loss

In the same function, implement the second part that computes the data and regularizaion loss.

```
In [4]: reg = 0.1
        loss, _ = two_layer_net(X, model, y, reg)
        correct_loss = 1.38191946092

        # should be very small, we get 5e-12
        print('Difference between your loss and correct loss:')
        print(np.sum(np.abs(loss - correct_loss)))
```

```
Difference between your loss and correct loss:
4.6769255135359344e-12
```

4 Backward pass

Implement the rest of the function. This will compute the gradient of the loss with respect to the variables W1, b1, W2, and b2. Now that you (hopefully!) have a correctly implemented forward pass, you can debug your backward pass using a numeric gradient check:

```
In [5]: from cs231n.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 = two_layer_net(X, model, y, reg)

        # these should all be less than 1e-8 or so
        for param_name in grads:
            param_grad_num = eval_numerical_gradient(lambda W: two_layer_net(X, model, y, reg)[0
            print('%s max relative error: %e' % (param_name, rel_error(param_grad_num, grads[par
```

```
W2 max relative error: 9.913913e-10
b2 max relative error: 8.190173e-11
W1 max relative error: 4.426512e-09
b1 max relative error: 5.435430e-08
```

5 Train the network

To train the network we will use SGD with Momentum. Last assignment you implemented vanilla SGD. You will now implement the momentum update and the RMSProp update. Open the file classifier_trainer.py and familiarize yourself with the ClassifierTrainer class. It performs optimization given an arbitrary cost function data, and model. By default it uses vanilla SGD, which we have already implemented for you. First, run the optimization below using Vanilla SGD:

```
In [6]: from cs231n.classifier_trainer import ClassifierTrainer

        model = init_toy_model()
        trainer = ClassifierTrainer()
        # call the trainer to optimize the loss
        # Notice that we're using sample_batches=False, so we're performing Gradient Descent (no
        best_model, loss_history, _, _ = trainer.train(X, y, X, y,
                                                       model, two_layer_net,
                                                       reg=0.001,
                                                       learning_rate=1e-1, momentum=0.0, learning_
                                                       update='sgd', sample_batches=False,
                                                       num_epochs=100,
                                                       verbose=False)
        print('Final loss with vanilla SGD: %f' % (loss_history[-1], ))
```

```
Final loss with vanilla SGD: 0.940686
```

Now fill in the **momentum update** in the first missing code block inside the `train` function, and run the same optimization as above but with the momentum update. You should see a much better result in the final obtained loss:

```
In [7]: model = init_toy_model()
        trainer = ClassifierTrainer()
        # call the trainer to optimize the loss
        # Notice that we're using sample_batches=False, so we're performing Gradient Descent (no
        best_model, loss_history, _, _ = trainer.train(X, y, X, y,
                                                       model, two_layer_net,
                                                       reg=0.001,
                                                       learning_rate=1e-1, momentum=0.9, learning_
                                                       update='momentum', sample_batches=False,
                                                       num_epochs=100,
                                                       verbose=False)
        correct_loss = 0.494394
        print('Final loss with momentum SGD: %f. We get: %f' % (loss_history[-1], correct_loss))
```

```
Final loss with momentum SGD: 0.494394. We get: 0.494394
```

The **RMSProp** update step is given as follows:

```
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += - learning_rate * dx / np.sqrt(cache + 1e-8)
```

Here, `decay_rate` is a hyperparameter and typical values are [0.9, 0.99, 0.999].
Implement the **RMSProp** update rule inside the `train` function and rerun the optimization:

```
In [8]: model = init_toy_model()
        trainer = ClassifierTrainer()
        # call the trainer to optimize the loss
        # Notice that we're using sample_batches=False, so we're performing Gradient Descent (no
        best_model, loss_history, _, _ = trainer.train(X, y, X, y,
                                                       model, two_layer_net,
                                                       reg=0.001,
                                                       learning_rate=1e-1, momentum=0.9, learning_
                                                       update='rmsprop', sample_batches=False,
                                                       num_epochs=100,
                                                       verbose=False)
        correct_loss = 0.439368
        print('Final loss with RMSProp: %f. We get: %f' % (loss_history[-1], correct_loss))
```

```
Final loss with RMSProp: 0.439368. We get: 0.439368
```

6 Load the data

Now that you have implemented a two-layer network that passes gradient checks, it's time to load up our favorite CIFAR-10 data so we can use it to train a classifier.

```
In [9]: from cs231n.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.
            """
            # Load the raw CIFAR-10 data
            cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'
            X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

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

7 Train a network

To train our network we will use SGD with momentum. In addition, we will adjust the learning rate with an exponential learning rate schedule as optimization proceeds; after each epoch, we will reduce the learning rate by multiplying it by a decay rate.

```
In [14]: from cs231n.classifiers.neural_net import init_two_layer_model

         model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of clas
         trainer = ClassifierTrainer()
         best_model, loss_history, train_acc, val_acc = trainer.train(X_train, y_train, X_val, y
                                                      model, two_layer_net,
                                                      num_epochs=5, reg=1.0,
                                                      momentum=0.9, learning_rate_decay = 0.95,
                                                      learning_rate=1e-5, verbose=False)
```

8 Debug the training

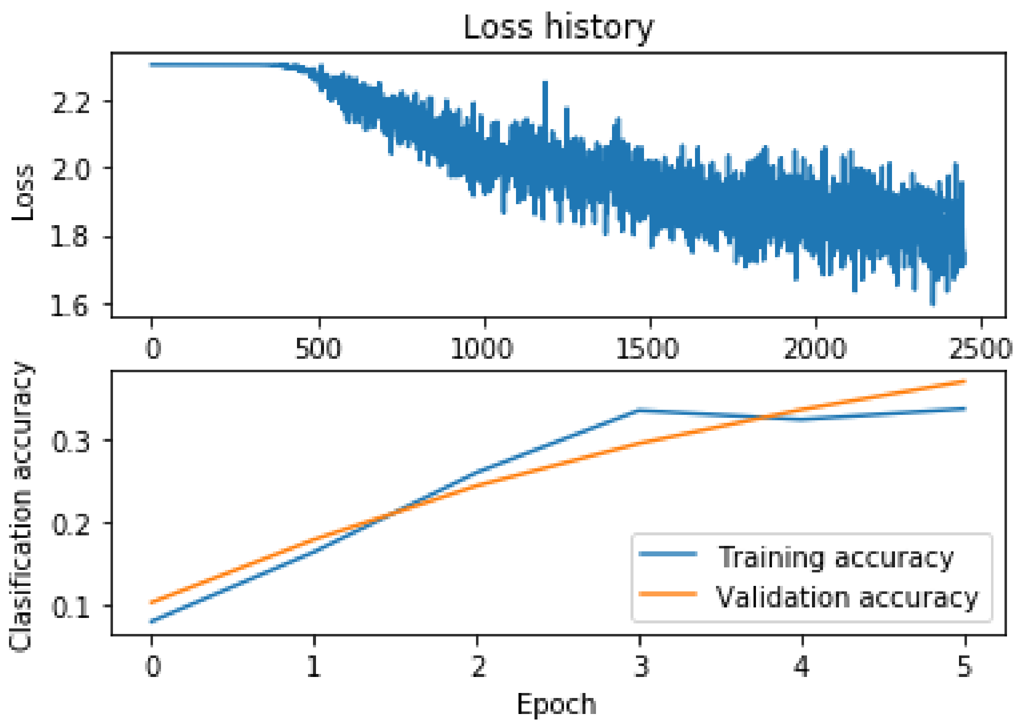
With the default parameters we provided above, you should get a validation accuracy of about 35% on the validation set. This isn't very good.

One strategy for getting insight into what's wrong is to plot the loss function and the accuracies on the training and validation sets during optimization.

Another strategy is to visualize the weights that were learned in the first layer of the network. In most neural networks trained on visual data, the first layer weights typically show some visible structure when visualized.

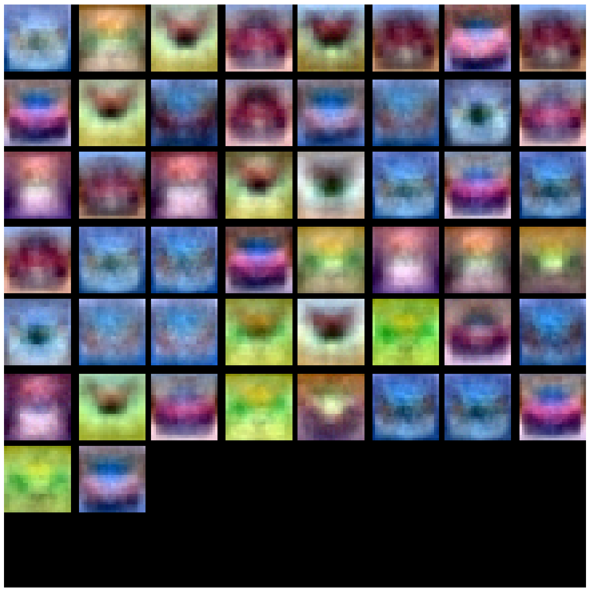
```
In [15]: # Plot the loss function and train / validation accuracies
         plt.subplot(2, 1, 1)
         plt.plot(loss_history)
         plt.title('Loss history')
         plt.xlabel('Iteration')
         plt.ylabel('Loss')

         plt.subplot(2, 1, 2)
         plt.plot(train_acc)
         plt.plot(val_acc)
         plt.legend(['Training accuracy', 'Validation accuracy'], loc='lower right')
         plt.xlabel('Epoch')
         plt.ylabel('Clasification accuracy')
```

```
Out[15]: Text(0, 0.5, 'Clasification accuracy')
```



```
In [16]: from cs231n.vis_utils import visualize_grid

         # Visualize the weights of the network
```

```
def show_net_weights(model):
    plt.imshow(visualize_grid(model['W1'].T.reshape(-1, 32, 32, 3), padding=3).astype('
    plt.gca().axis('off')
    plt.show()

show_net_weights(model)
```



9 Tune your hyperparameters

What's wrong?. Looking at the visualizations above, we see that the loss is decreasing more or less linearly, which seems to suggest that the learning rate may be too low. Moreover, there is no gap between the training and validation accuracy, suggesting that the model we used has low capacity, and that we should increase its size. On the other hand, with a very large model we would expect to see more overfitting, which would manifest itself as a very large gap between the training and validation accuracy.

Tuning. Tuning the hyperparameters and developing intuition for how they affect the final performance is a large part of using Neural Networks, so we want you to get a lot of practice. Below, you should experiment with different values of the various hyperparameters, including hidden layer size, learning rate, numer of training epochs, and regularization strength. You might also consider tuning the momentum and learning rate decay parameters, but you should be able to get good performance using the default values.

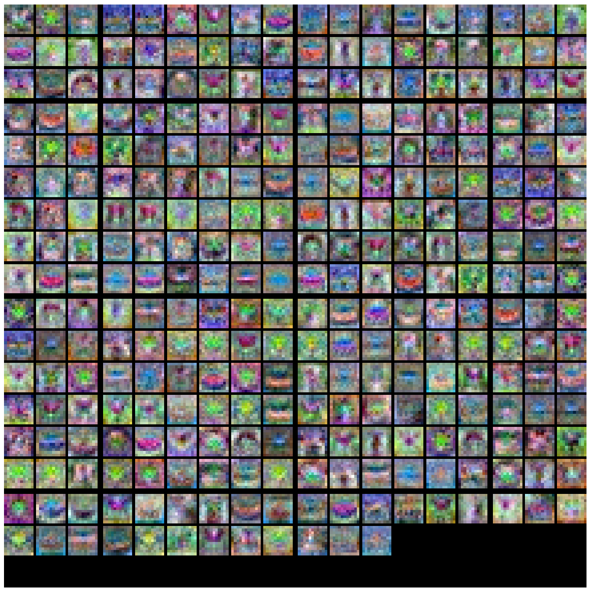
Results. To receive full credit, you should obtain at least 45% validation and testing accuracy.

```
In [17]: best_model = None # store the best model into this
```

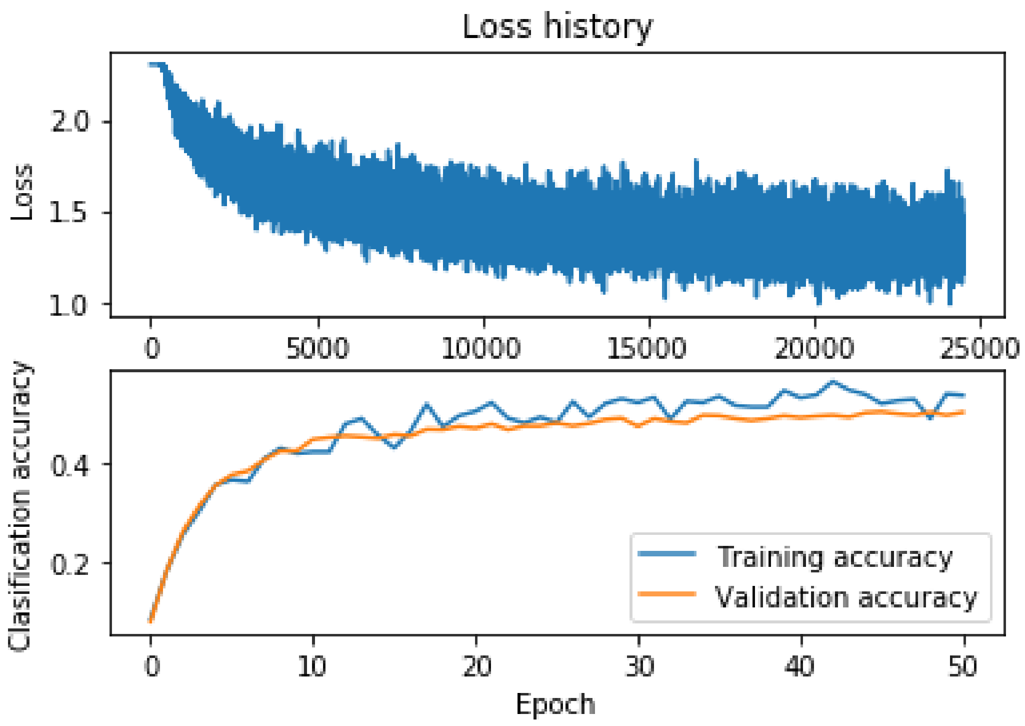
```
#################################################################################
# TODO: Tune hyperparameters using the validation set. Store your best trained  #
# model in best_model.                                                          #
#                                                                               #
# To help debug your network, it may help to use visualizations similar to the  #
# ones we used above; these visualizations will have significant qualitative    #
# differences from the ones we saw above for the poorly tuned network.          #
#                                                                               #
# Tweaking hyperparameters by hand can be fun, but you might find it useful to  #
# write code to sweep through possible combinations of hyperparameters          #
# automatically.                                                                #
#################################################################################
# input size, hidden size, number of classes
model = init_two_layer_model(32*32*3, 300, 10)
trainer = ClassifierTrainer()
best_model, loss_history, train_acc, val_acc = trainer.train(X_train, y_train,
                                             X_val, y_val,
                                             model, two_layer_net,
                                             num_epochs=50, reg=0.001,
                                             momentum=0.9,
                                             learning_rate_decay=0.95,
                                             learning_rate=1e-5, verbose=False)
#################################################################################
#                                 END OF YOUR CODE                              #
#################################################################################
```

In [ ]:

In [18]:
```
# visualize the weights
show_net_weights(best_model)
```



```
In [19]: # Plot the loss function and train / validation accuracies
         plt.subplot(2, 1, 1)
         plt.plot(loss_history)
         plt.title('Loss history')
         plt.xlabel('Iteration')
         plt.ylabel('Loss')

         plt.subplot(2, 1, 2)
         plt.plot(train_acc)
         plt.plot(val_acc)
         plt.legend(['Training accuracy', 'Validation accuracy'], loc='lower right')
         plt.xlabel('Epoch')
         plt.ylabel('Clasification accuracy')

Out[19]: Text(0, 0.5, 'Clasification accuracy')
```



10 Run on the test set

When you are done experimenting, you should evaluate your final trained network on the test set.

```
In [20]: scores_test = two_layer_net(X_test, best_model)
         print('Test accuracy: ', np.mean(np.argmax(scores_test, axis=1) == y_test))
```

```
Test accuracy:  0.507
```

layers

February 10, 2019

1 Modular neural nets

In the previous exercise, we computed the loss and gradient for a two-layer neural network in a single monolithic function. This isn't very difficult for a small two-layer network, but would be tedious and error-prone for larger networks. Ideally we want to build networks using a more modular design so that we can snap together different types of layers and loss functions in order to quickly experiment with different architectures.

In this exercise we will implement this approach, and develop a number of different layer types in isolation that can then be easily plugged together. For each layer we will implement `forward` and `backward` functions. The `forward` function will receive data, weights, and other parameters, and will return both an output and a `cache` object that stores data needed for the backward pass. The `backward` function will recieve upstream derivatives and the cache object, and will return gradients with respect to the data and all of the weights. This will allow us to write code that looks like this:

```
def two_layer_net(X, W1, b1, W2, b2, reg):
    # Forward pass; compute scores
    s1, fc1_cache = affine_forward(X, W1, b1)
    a1, relu_cache = relu_forward(s1)
    scores, fc2_cache = affine_forward(a1, W2, b2)

    # Loss functions return data loss and gradients on scores
    data_loss, dscores = svm_loss(scores, y)

    # Compute backward pass
    da1, dW2, db2 = affine_backward(dscores, fc2_cache)
    ds1 = relu_backward(da1, relu_cache)
    dX, dW1, db1 = affine_backward(ds1, fc1_cache)

    # A real network would add regularization here

    # Return loss and gradients
    return loss, dW1, db1, dW2, db2
```

```
In [1]: # As usual, a bit of setup

        import numpy as np
        import matplotlib.pyplot as plt
        from cs231n.gradient_check import eval_numerical_gradient_array, eval_numerical_gradient
        from cs231n.layers import *

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

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

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

2 Affine layer: forward

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

Once you are done we will test your can test your implementation by running the following:

```
In [2]: # Test the affine_forward function
        num_inputs = 2
        input_shape = (4, 5, 6)
        output_dim = 3

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

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

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

        # Compare your output with ours. The error should be around 1e-9.
        print('Testing affine_forward function:')
        print('difference: ', rel_error(out, correct_out))
```

```
Testing affine_forward function:
difference:  9.769847728806635e-10
```

3 Affine layer: backward

Now implement the `affine_backward` function. You can test your implementation using numeric gradient checking.

```
In [3]: # Test the affine_backward function

        x = np.random.randn(10, 2, 3)
        w = np.random.randn(6, 5)
        b = np.random.randn(5)
        dout = np.random.randn(10, 5)

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

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

        # The error should be less than 1e-10
        print('Testing affine_backward function:')
        print('dx error: ', rel_error(dx_num, dx))
        print('dw error: ', rel_error(dw_num, dw))
        print('db error: ', rel_error(db_num, db))
```

```
Testing affine_backward function:
dx error:  9.49814151320581e-10
dw error:  7.953860621293711e-11
db error:  1.9684874918918535e-11
```

4 ReLU layer: forward

Implement the `relu_forward` function and test your implementation by running the following:

```
In [4]: # Test the relu_forward function

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

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

        # Compare your output with ours. The error should be around 1e-8
        print('Testing relu_forward function:')
        print('difference: ', rel_error(out, correct_out))
```

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

5 ReLU layer: backward

Implement the `relu_backward` function and test your implementation using numeric gradient checking:

```
In [ ]: x = np.random.randn(10, 10)
        dout = np.random.randn(*x.shape)

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

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

        # The error should be around 1e-12
        print('Testing relu_backward function:')
        print('dx error: ', rel_error(dx_num, dx))
```

```
Testing relu_backward function:
dx error:  3.2756099632174382e-12
```

6 Loss layers: Softmax and SVM

You implemented these loss functions in the last assignment, so we'll give them to you for free here. It's still a good idea to test them to make sure they work correctly.

```
In [ ]: num_classes, num_inputs = 10, 50
        x = 0.001 * np.random.randn(num_inputs, num_classes)
        y = np.random.randint(num_classes, size=num_inputs)

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

        # Test svm_loss function. Loss should be around 9 and dx error should be 1e-9
        print('Testing svm_loss:')
        print('loss: ', loss)
        print('dx error: ', rel_error(dx_num, dx))

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

        # Test softmax_loss function. Loss should be 2.3 and dx error should be 1e-8
        print('\nTesting softmax_loss:')
        print('loss: ', loss)
        print('dx error: ', rel_error(dx_num, dx))
```

```
Testing svm_loss:
loss:  9.00094085562819
dx error:  1.4021566006651672e-09

Testing softmax_loss:
loss:  2.3026796217657965
dx error:  8.665832340076076e-09
```

7 Convolution layer: forward naive

We are now ready to implement the forward pass for a convolutional layer. Implement the function `conv_forward_naive` in the file `cs231n/layers.py`.

You don't have to worry too much about efficiency at this point; just write the code in whatever way you find most clear.

You can test your implementation by running the following:

```
In [ ]: x_shape = (2, 3, 4, 4)
        w_shape = (3, 3, 4, 4)
        x = np.linspace(-0.1, 0.5, num=np.prod(x_shape)).reshape(x_shape)
        w = np.linspace(-0.2, 0.3, num=np.prod(w_shape)).reshape(w_shape)
        b = np.linspace(-0.1, 0.2, num=3)

        conv_param = {'stride': 2, 'pad': 1}
        out, _ = conv_forward_naive(x, w, b, conv_param)

        correct_out = np.array([[[[-0.08759809, -0.10987781],
                                   [-0.18387192, -0.2109216 ]],
                                  [[ 0.21027089,  0.21661097],
                                   [ 0.22847626,  0.23004637]],
                                  [[ 0.50813986,  0.54309974],
                                   [ 0.64082444,  0.67101435]]],
                                 [[[-0.98053589, -1.03143541],
                                   [-1.19128892, -1.24695841]],
                                  [[ 0.69108355,  0.66880383],
                                   [ 0.59480972,  0.56776003]],
                                  [[ 2.36270298,  2.36904306],
                                   [ 2.38090835,  2.38247847]]]])

        # Compare your output to ours; difference should be around 1e-8
        print('Testing conv_forward_naive')
        print('difference: ', rel_error(out, correct_out))
```

```
Testing conv_forward_naive
difference:  2.2121476417505994e-08
```

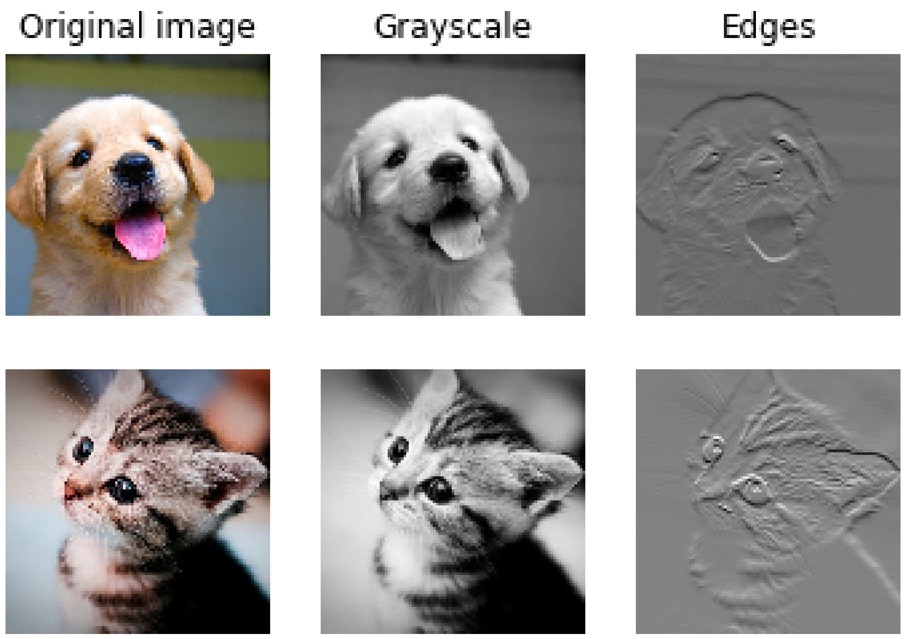
8 Aside: Image processing via convolutions

As fun way to both check your implementation and gain a better understanding of the type of operation that convolutional layers can perform, we will set up an input containing two images and manually set up filters that perform common image processing operations (grayscale conversion and edge detection). The convolution forward pass will apply these operations to each of the input images. We can then visualize the results as a sanity check.

```
In [ ]: from scipy.misc import imread, imresize

        kitten, puppy = imread('kitten.jpg'), imread('puppy.jpg')
        # kitten is wide, and puppy is already square
        d = kitten.shape[1] - kitten.shape[0]
        kitten_cropped = kitten[:, d//2:-d//2, :]

        img_size = 200   # Make this smaller if it runs too slow
        x = np.zeros((2, 3, img_size, img_size))
        x[0, :, :, :] = imresize(puppy, (img_size, img_size)).transpose((2, 0, 1))
        x[1, :, :, :] = imresize(kitten_cropped, (img_size, img_size)).transpose((2, 0, 1))

        # Set up a convolutional weights holding 2 filters, each 3x3
        w = np.zeros((2, 3, 3, 3))

        # The first filter converts the image to grayscale.
        # Set up the red, green, and blue channels of the filter.
        w[0, 0, :, :] = [[0, 0, 0], [0, 0.3, 0], [0, 0, 0]]
        w[0, 1, :, :] = [[0, 0, 0], [0, 0.6, 0], [0, 0, 0]]
        w[0, 2, :, :] = [[0, 0, 0], [0, 0.1, 0], [0, 0, 0]]

        # Second filter detects horizontal edges in the blue channel.
        w[1, 2, :, :] = [[1, 2, 1], [0, 0, 0], [-1, -2, -1]]

        # Vector of biases. We don't need any bias for the grayscale
        # filter, but for the edge detection filter we want to add 128
        # to each output so that nothing is negative.
        b = np.array([0, 128])

        # Compute the result of convolving each input in x with each filter in w,
        # offsetting by b, and storing the results in out.
        out, _ = conv_forward_naive(x, w, b, {'stride': 1, 'pad': 1})

        def imshow_noax(img, normalize=True):
            """ Tiny helper to show images as uint8 and remove axis labels """
            if normalize:
                img_max, img_min = np.max(img), np.min(img)
                img = 255.0 * (img - img_min) / (img_max - img_min)
            plt.imshow(img.astype('uint8'))
            plt.gca().axis('off')
```

```
# Show the original images and the results of the conv operation
plt.subplot(2, 3, 1)
imshow_noax(puppy, normalize=False)
plt.title('Original image')
plt.subplot(2, 3, 2)
imshow_noax(out[0, 0])
plt.title('Grayscale')
plt.subplot(2, 3, 3)
imshow_noax(out[0, 1])
plt.title('Edges')
plt.subplot(2, 3, 4)
imshow_noax(kitten_cropped, normalize=False)
plt.subplot(2, 3, 5)
imshow_noax(out[1, 0])
plt.subplot(2, 3, 6)
imshow_noax(out[1, 1])
plt.show()
```

```
/Users/alexisdurocher/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:3: Deprecation
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.
  This is separate from the ipykernel package so we can avoid doing imports until
/Users/alexisdurocher/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:10: Deprecatio
`imresize` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``skimage.transform.resize`` instead.
  # Remove the CWD from sys.path while we load stuff.
/Users/alexisdurocher/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:11: Deprecatio
`imresize` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``skimage.transform.resize`` instead.
  # This is added back by InteractiveShellApp.init_path()
```



9 Convolution layer: backward naive

Next you need to implement the function `conv_backward_naive` in the file `cs231n/layers.py`. As usual, we will check your implementation with numeric gradient checking.

```
In [ ]: x = np.random.randn(4, 3, 5, 5)
        w = np.random.randn(2, 3, 3, 3)
        b = np.random.randn(2,)
        dout = np.random.randn(4, 2, 5, 5)
        conv_param = {'stride': 1, 'pad': 1}

        dx_num = eval_numerical_gradient_array(lambda x: conv_forward_naive(x, w, b, conv_param)
        dw_num = eval_numerical_gradient_array(lambda w: conv_forward_naive(x, w, b, conv_param)
        db_num = eval_numerical_gradient_array(lambda b: conv_forward_naive(x, w, b, conv_param)

        out, cache = conv_forward_naive(x, w, b, conv_param)
        dx, dw, db = conv_backward_naive(dout, cache)

        # Your errors should be around 1e-9'
        print('Testing conv_backward_naive function')
        print('dx error: ', rel_error(dx, dx_num))
        print('dw error: ', rel_error(dw, dw_num))
        print('db error: ', rel_error(db, db_num))
```

```
Testing conv_backward_naive function
dx error:  4.150387569305753e-09
dw error:  2.1719502642537122e-09
db error:  9.417065769808873e-12
```

10 Max pooling layer: forward naive

The last layer we need for a basic convolutional neural network is the max pooling layer. First implement the forward pass in the function `max_pool_forward_naive` in the file `cs231n/layers.py`.

```
In [ ]: x_shape = (2, 3, 4, 4)
        x = np.linspace(-0.3, 0.4, num=np.prod(x_shape)).reshape(x_shape)
        pool_param = {'pool_width': 2, 'pool_height': 2, 'stride': 2}

        out, _ = max_pool_forward_naive(x, pool_param)

        correct_out = np.array([[[[-0.26315789, -0.24842105],
                                  [-0.20421053, -0.18947368]],
                                 [[-0.14526316, -0.13052632],
                                  [-0.08631579, -0.07157895]],
                                 [[-0.02736842, -0.01263158],
                                  [ 0.03157895,  0.04631579]]],
                                [[[ 0.09052632,  0.10526316],
                                  [ 0.14947368,  0.16421053]],
                                 [[ 0.20842105,  0.22315789],
                                  [ 0.26736842,  0.28210526]],
                                 [[ 0.32631579,  0.34105263],
                                  [ 0.38526316,  0.4       ]]]])

        # Compare your output with ours. Difference should be around 1e-8.
        print('Testing max_pool_forward_naive function:')
        print('difference: ', rel_error(out, correct_out))
```

```
Testing max_pool_forward_naive function:
difference:  4.1666665157267834e-08
```

11 Max pooling layer: backward naive

Implement the backward pass for a max pooling layer in the function `max_pool_backward_naive` in the file `cs231n/layers.py`. As always we check the correctness of the backward pass using numerical gradient checking.

```
In [ ]: x = np.random.randn(3, 2, 8, 8)
        dout = np.random.randn(3, 2, 4, 4)
        pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}
        dx_num = eval_numerical_gradient_array(lambda x: max_pool_forward_naive(x, pool_param)[0

        out, cache = max_pool_forward_naive(x, pool_param)
        dx = max_pool_backward_naive(dout, cache)

        # Your error should be around 1e-12
        print('Testing max_pool_backward_naive function:')
        print('dx error: ', rel_error(dx, dx_num))
```

```
Testing max_pool_backward_naive function:
dx error:  3.275644045130772e-12
```

12 Fast layers

Making convolution and pooling layers fast can be challenging. To spare you the pain, we've provided fast implementations of the forward and backward passes for convolution and pooling layers in the file `cs231n/fast_layers.py`.

The fast convolution implementation depends on a Cython extension; to compile it you need to run the following from the `cs231n` directory:

```
python setup.py build_ext --inplace
```

The API for the fast versions of the convolution and pooling layers is exactly the same as the naive versions that you implemented above: the forward pass receives data, weights, and parameters and produces outputs and a cache object; the backward pass recieves upstream derivatives and the cache object and produces gradients with respect to the data and weights.

NOTE: The fast implementation for pooling will only perform optimally if the pooling regions are non-overlapping and tile the input. If these conditions are not met then the fast pooling implementation will not be much faster than the naive implementation.

You can compare the performance of the naive and fast versions of these layers by running the following:

```
In [ ]: from cs231n.fast_layers import conv_forward_fast, conv_backward_fast
        from time import time

        x = np.random.randn(100, 3, 31, 31)
        w = np.random.randn(25, 3, 3, 3)
        b = np.random.randn(25,)
        dout = np.random.randn(100, 25, 16, 16)
        conv_param = {'stride': 2, 'pad': 1}

        t0 = time()
        out_naive, cache_naive = conv_forward_naive(x, w, b, conv_param)
        t1 = time()
        out_fast, cache_fast = conv_forward_fast(x, w, b, conv_param)
        t2 = time()
        print('Testing conv_forward_fast:')
        print('Naive: %fs' % (t1 - t0))
        print('Fast: %fs' % (t2 - t1))
        print('Speedup: %fx' % ((t1 - t0) / (t2 - t1)))
        print('Difference: ', rel_error(out_naive, out_fast))

        t0 = time()
        dx_naive, dw_naive, db_naive = conv_backward_naive(dout, cache_naive)
        t1 = time()
        dx_fast, dw_fast, db_fast = conv_backward_fast(dout, cache_fast)
        t2 = time()

        print('\nTesting conv_backward_fast:')
        print('Naive: %fs' % (t1 - t0))
        print('Fast: %fs' % (t2 - t1))
        print('Speedup: %fx' % ((t1 - t0) / (t2 - t1)))
        print('dx difference: ', rel_error(dx_naive, dx_fast))
        print('dw difference: ', rel_error(dw_naive, dw_fast))
        print('db difference: ', rel_error(db_naive, db_fast))
```

```
In [ ]: from cs231n.fast_layers import max_pool_forward_fast, max_pool_backward_fast

        x = np.random.randn(100, 3, 32, 32)
        dout = np.random.randn(100, 3, 16, 16)
        pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

        t0 = time()
        out_naive, cache_naive = max_pool_forward_naive(x, pool_param)
        t1 = time()
        out_fast, cache_fast = max_pool_forward_fast(x, pool_param)
        t2 = time()

        print('Testing pool_forward_fast:')
        print('Naive: %fs' % (t1 - t0))
        print('fast: %fs' % (t2 - t1))
        print('speedup: %fx' % ((t1 - t0) / (t2 - t1)))
        print('difference: ', rel_error(out_naive, out_fast))

        t0 = time()
        dx_naive = max_pool_backward_naive(dout, cache_naive)
        t1 = time()
        dx_fast = max_pool_backward_fast(dout, cache_fast)
        t2 = time()

        print('\nTesting pool_backward_fast:')
        print('Naive: %fs' % (t1 - t0))
        print('speedup: %fx' % ((t1 - t0) / (t2 - t1)))
        print('dx difference: ', rel_error(dx_naive, dx_fast))
```

13 Sandwich layers

There are a couple common layer "sandwiches" that frequently appear in ConvNets. For example convolutional layers are frequently followed by ReLU and pooling, and affine layers are frequently followed by ReLU. To make it more convenient to use these common patterns, we have defined several convenience layers in the file `cs231n/layer_utils.py`. Lets grad-check them to make sure that they work correctly:

```
In [ ]: from cs231n.layer_utils import conv_relu_pool_forward, conv_relu_pool_backward

        x = np.random.randn(2, 3, 16, 16)
        w = np.random.randn(3, 3, 3, 3)
        b = np.random.randn(3,)
        dout = np.random.randn(2, 3, 8, 8)
        conv_param = {'stride': 1, 'pad': 1}
        pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

        out, cache = conv_relu_pool_forward(x, w, b, conv_param, pool_param)
        dx, dw, db = conv_relu_pool_backward(dout, cache)

        dx_num = eval_numerical_gradient_array(lambda x: conv_relu_pool_forward(x, w, b, conv_pa
        dw_num = eval_numerical_gradient_array(lambda w: conv_relu_pool_forward(x, w, b, conv_pa
        db_num = eval_numerical_gradient_array(lambda b: conv_relu_pool_forward(x, w, b, conv_pa

        print('Testing conv_relu_pool_forward:')
        print('dx error: ', rel_error(dx_num, dx))
        print('dw error: ', rel_error(dw_num, dw))
        print('db error: ', rel_error(db_num, db))

In [ ]: from cs231n.layer_utils import conv_relu_forward, conv_relu_backward

        x = np.random.randn(2, 3, 8, 8)
        w = np.random.randn(3, 3, 3, 3)
        b = np.random.randn(3,)
        dout = np.random.randn(2, 3, 8, 8)
        conv_param = {'stride': 1, 'pad': 1}

        out, cache = conv_relu_forward(x, w, b, conv_param)
        dx, dw, db = conv_relu_backward(dout, cache)

        dx_num = eval_numerical_gradient_array(lambda x: conv_relu_forward(x, w, b, conv_param)[
        dw_num = eval_numerical_gradient_array(lambda w: conv_relu_forward(x, w, b, conv_param)[
        db_num = eval_numerical_gradient_array(lambda b: conv_relu_forward(x, w, b, conv_param)[

        print('Testing conv_relu_forward:')
        print('dx error: ', rel_error(dx_num, dx))
        print('dw error: ', rel_error(dw_num, dw))
        print('db error: ', rel_error(db_num, db))
```

```
In [ ]: from cs231n.layer_utils import affine_relu_forward, affine_relu_backward

        x = np.random.randn(2, 3, 4)
        w = np.random.randn(12, 10)
        b = np.random.randn(10)
        dout = np.random.randn(2, 10)

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

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

        print('Testing affine_relu_forward:')
        print('dx error: ', rel_error(dx_num, dx))
        print('dw error: ', rel_error(dw_num, dw))
        print('db error: ', rel_error(db_num, db))
```

convnet

February 10, 2019

1 Train a ConvNet!

We now have a generic solver and a bunch of modularized layers. It's time to put it all together, and train a ConvNet to recognize the classes in CIFAR-10. In this notebook we will walk you through training a simple two-layer ConvNet and then set you free to build the best net that you can to perform well on CIFAR-10.

Open up the file `cs231n/classifiers/convnet.py`; you will see that the `two_layer_convnet` function computes the loss and gradients for a two-layer ConvNet. Note that this function uses the "sandwich" layers defined in `cs231n/layer_utils.py`.

```
In [1]: # As usual, a bit of setup

        import numpy as np
        import matplotlib.pyplot as plt
        from cs231n.classifier_trainer import ClassifierTrainer
        from cs231n.gradient_check import eval_numerical_gradient
        from cs231n.classifiers.convnet import *

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

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

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

In [2]: from cs231n.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 = 'cs231n/datasets/cifar-10-batches-py'
            X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

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

            # Transpose so that channels come first
            X_train = X_train.transpose(0, 3, 1, 2).copy()
            X_val = X_val.transpose(0, 3, 1, 2).copy()
            x_test = X_test.transpose(0, 3, 1, 2).copy()

            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, 3, 32, 32)
Train labels shape:  (49000,)
Validation data shape:  (1000, 3, 32, 32)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
```

2 Sanity check loss

After you build a new network, one of the first things you should do is sanity check the loss. When we use the softmax loss, we expect the loss for random weights (and no regularization) to be about `log(C)` for `C` classes. When we add regularization this should go up.

```
In [3]: model = init_two_layer_convnet()

        X = np.random.randn(100, 3, 32, 32)
        y = np.random.randint(10, size=100)

        loss, _ = two_layer_convnet(X, model, y, reg=0)

        # Sanity check: Loss should be about log(10) = 2.3026
        print('Sanity check loss (no regularization): ', loss)

        # Sanity check: Loss should go up when you add regularization
        loss, _ = two_layer_convnet(X, model, y, reg=1)
        print('Sanity check loss (with regularization): ', loss)
```

```
Sanity check loss (no regularization):  2.302589986326018
Sanity check loss (with regularization):  2.344903490002597
```

3 Gradient check

After the loss looks reasonable, you should always use numeric gradient checking to make sure that your backward pass is correct. When you use numeric gradient checking you should use a small amount of artifical data and a small number of neurons at each layer.

```
In [4]: num_inputs = 2
        input_shape = (3, 16, 16)
        reg = 0.0
        num_classes = 10
        X = np.random.randn(num_inputs, *input_shape)
        y = np.random.randint(num_classes, size=num_inputs)

        model = init_two_layer_convnet(num_filters=3, filter_size=3, input_shape=input_shape)
        loss, grads = two_layer_convnet(X, model, y)
        for param_name in sorted(grads):
            f = lambda _: two_layer_convnet(X, model, y)[0]
            param_grad_num = eval_numerical_gradient(f, model[param_name], verbose=False, h=1e-6
            e = rel_error(param_grad_num, grads[param_name])
            print('%s max relative error: %e' % (param_name, rel_error(param_grad_num, grads[par
```

```
W1 max relative error: 2.159361e-07
W2 max relative error: 1.545029e-05
b1 max relative error: 2.058688e-08
```

```
b2 max relative error: 1.502365e-09
```

4 Overfit small data

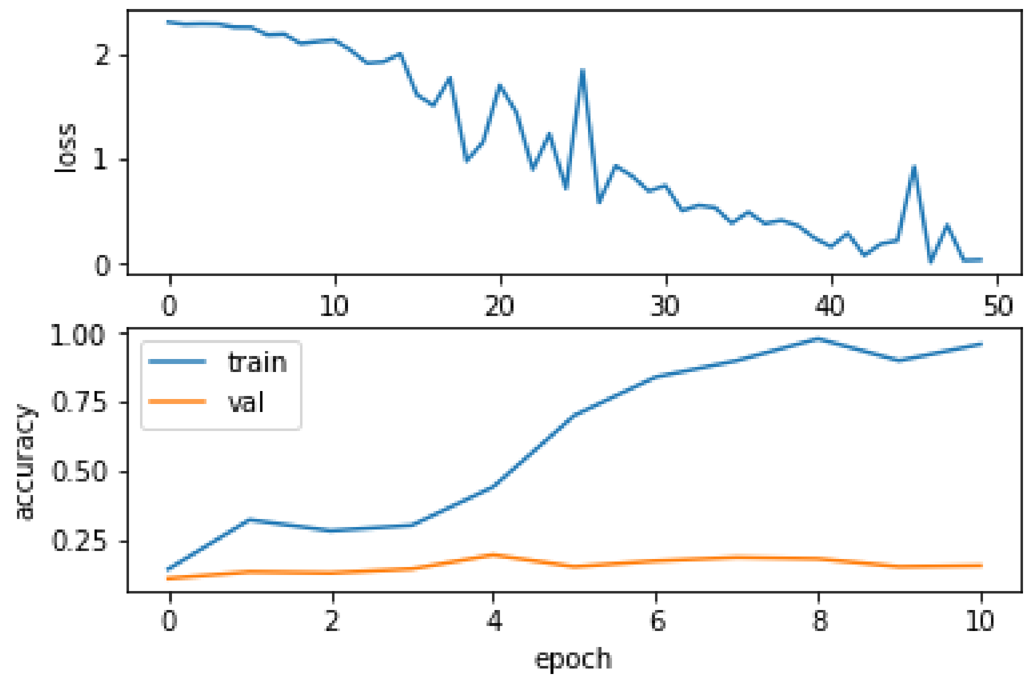
A nice trick is to train your model with just a few training samples. You should be able to overfit small datasets, which will result in very high training accuracy and comparatively low validation accuracy.

```
In [5]: # Use a two-layer ConvNet to overfit 50 training examples.

        model = init_two_layer_convnet()
        trainer = ClassifierTrainer()
        best_model, loss_history, train_acc_history, val_acc_history = trainer.train(
                  X_train[:50], y_train[:50], X_val, y_val, model, two_layer_convnet,
                  reg=0.001, momentum=0.9, learning_rate=0.0001, batch_size=10, num_epochs=10,
                  verbose=True)
```

```
starting iteration  0
Finished epoch 0 / 10: cost 2.308428, train: 0.140000, val 0.106000, lr 1.000000e-04
Finished epoch 1 / 10: cost 2.259305, train: 0.320000, val 0.130000, lr 9.500000e-05
Finished epoch 2 / 10: cost 2.123196, train: 0.280000, val 0.126000, lr 9.025000e-05
starting iteration  10
Finished epoch 3 / 10: cost 2.006251, train: 0.300000, val 0.141000, lr 8.573750e-05
Finished epoch 4 / 10: cost 1.162897, train: 0.440000, val 0.192000, lr 8.145062e-05
starting iteration  20
Finished epoch 5 / 10: cost 0.716435, train: 0.700000, val 0.149000, lr 7.737809e-05
Finished epoch 6 / 10: cost 0.692818, train: 0.840000, val 0.170000, lr 7.350919e-05
starting iteration  30
Finished epoch 7 / 10: cost 0.386743, train: 0.900000, val 0.183000, lr 6.983373e-05
Finished epoch 8 / 10: cost 0.242038, train: 0.980000, val 0.178000, lr 6.634204e-05
starting iteration  40
Finished epoch 9 / 10: cost 0.218968, train: 0.900000, val 0.149000, lr 6.302494e-05
Finished epoch 10 / 10: cost 0.036106, train: 0.960000, val 0.153000, lr 5.987369e-05
finished optimization. best validation accuracy: 0.192000
```

Plotting the loss, training accuracy, and validation accuracy should show clear overfitting:

```
In [6]: plt.subplot(2, 1, 1)
        plt.plot(loss_history)
        plt.xlabel('iteration')
        plt.ylabel('loss')

        plt.subplot(2, 1, 2)
        plt.plot(train_acc_history)
        plt.plot(val_acc_history)
        plt.legend(['train', 'val'], loc='upper left')
        plt.xlabel('epoch')
        plt.ylabel('accuracy')
        plt.show()
```



5 Train the net

Once the above works, training the net is the next thing to try. You can set the `acc_frequency` parameter to change the frequency at which the training and validation set accuracies are tested. If your parameters are set properly, you should see the training and validation accuracy start to improve within a hundred iterations, and you should be able to train a reasonable model with just one epoch.

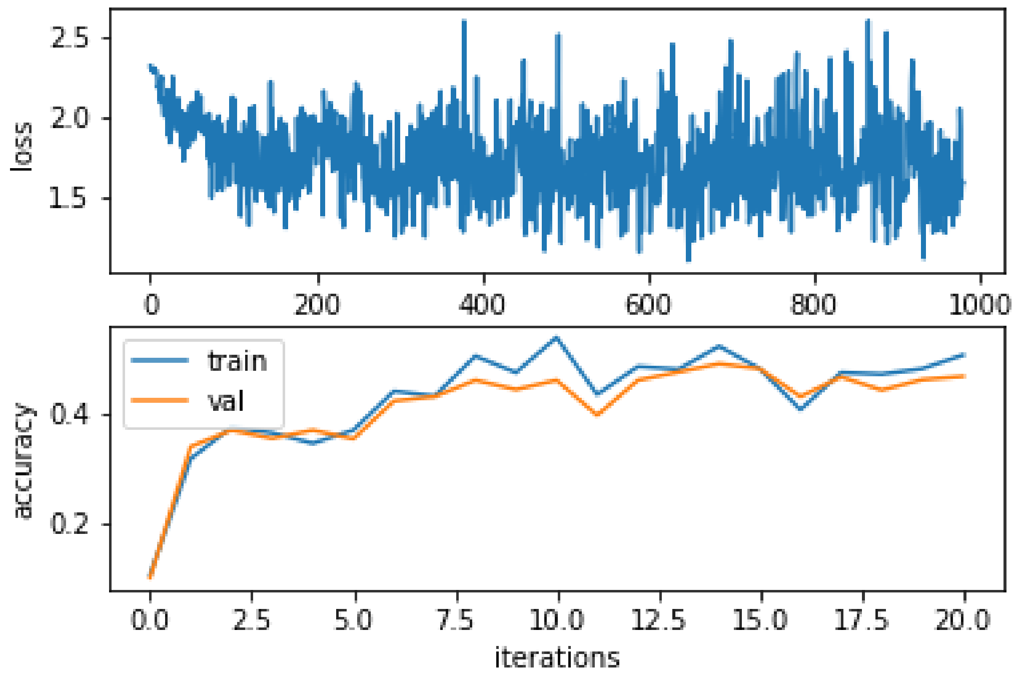
Using the parameters below you should be able to get at least 48% accuracy on the validation set.

```
In [7]: model = init_two_layer_convnet(filter_size=7)
        trainer = ClassifierTrainer()
        best_model, loss_history, train_acc_history, val_acc_history = trainer.train(
                  X_train, y_train, X_val, y_val, model, two_layer_convnet,
                  reg=0.001, momentum=0.9, learning_rate=0.0001, batch_size=50, num_epochs=1,
                  acc_frequency=50, verbose=False)
```

```
In [13]: print(f'Best validation accuracy is {max(val_acc_history)}')
```

```
Best validation accuracy is 0.489
```

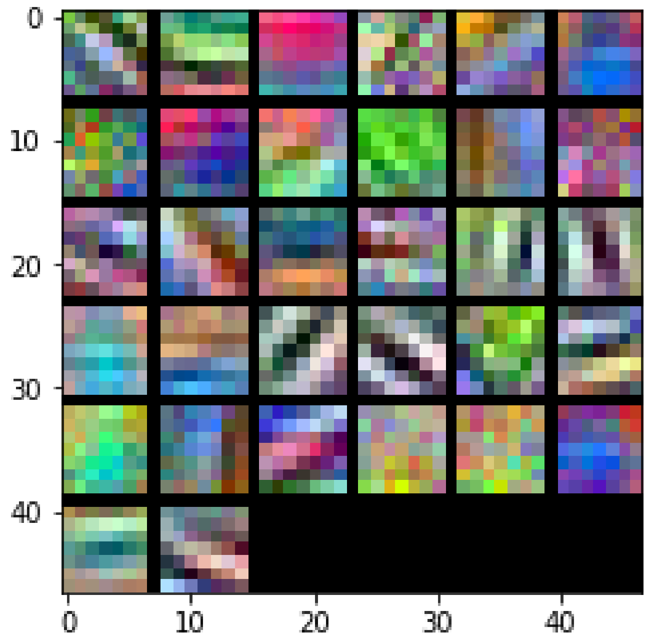
```
In [14]: plt.subplot(2, 1, 1)
         plt.plot(loss_history)
         plt.xlabel('iteration')
         plt.ylabel('loss')

         plt.subplot(2, 1, 2)
         plt.plot(train_acc_history)
         plt.plot(val_acc_history)
         plt.legend(['train', 'val'], loc='upper left')
         plt.xlabel('iterations')
         plt.ylabel('accuracy')
         plt.show()
```



6 Visualize weights

We can visualize the convolutional weights from the first layer. If everything worked properly, these will usually be edges and blobs of various colors and orientations.

```
In [8]: from cs231n.vis_utils import visualize_grid

        grid = visualize_grid(best_model['W1'].transpose(0, 2, 3, 1))
        plt.imshow(grid.astype('uint8'))

Out[8]: <matplotlib.image.AxesImage at 0x120a73940>
```


0
10
20
30
40
0
10
20
30
40

twolayer

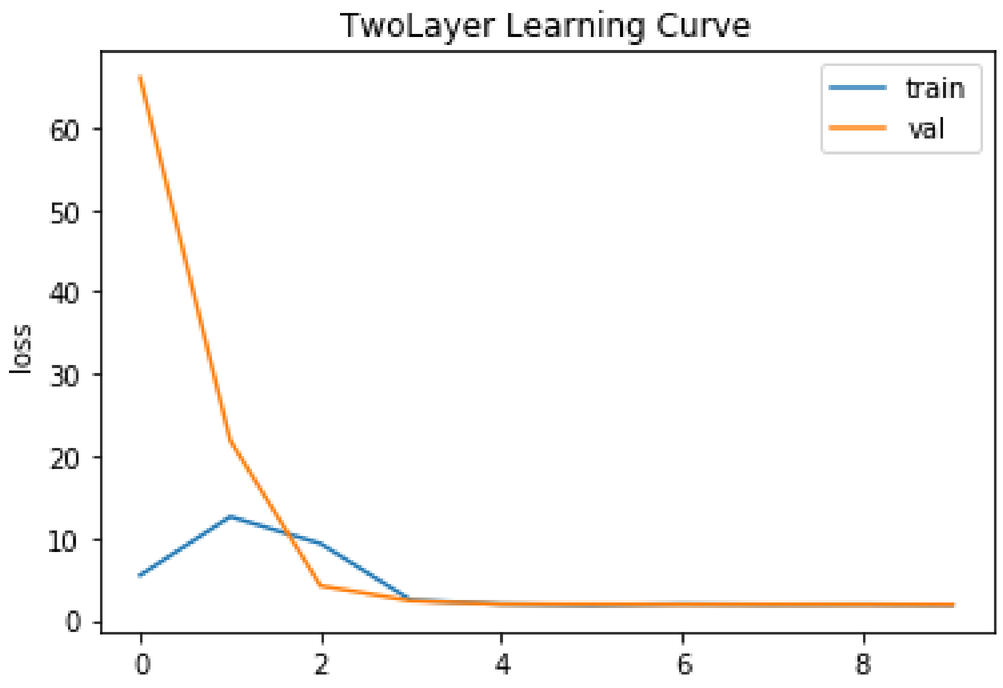
February 10, 2019

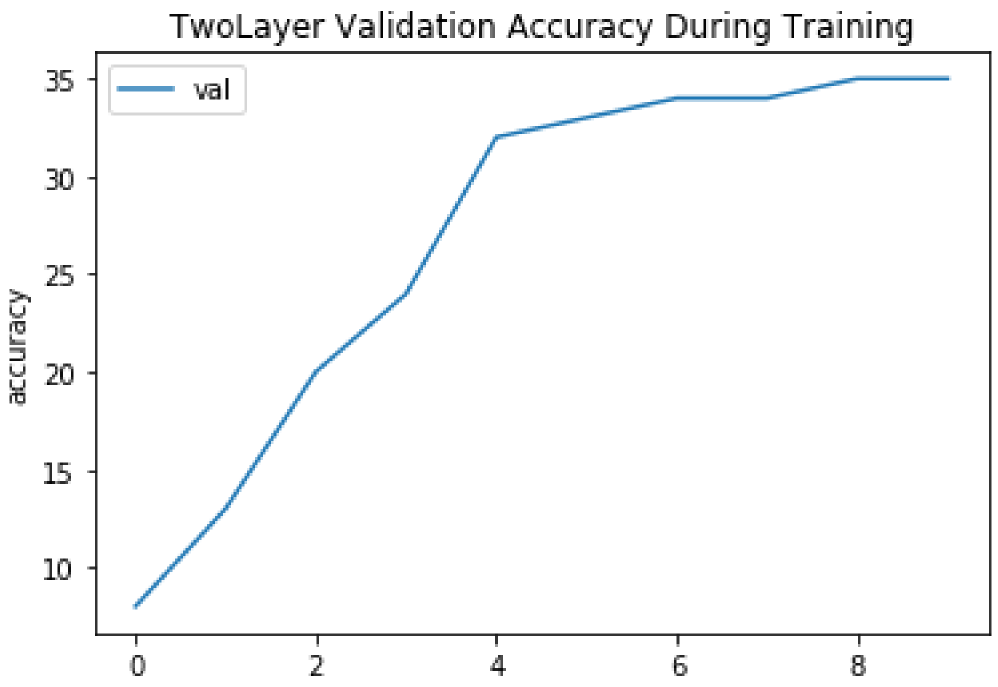
0.1 Visualizing the PyTorch model

```
In [8]: # Assuming that you have completed training the classifer, let us plot the training loss
        # example to show a simple way to log and plot data from PyTorch.

        # we neeed matplotlib to plot the graphs for us!
        import matplotlib
        # This is needed to save images
        matplotlib.use('Agg')
        import matplotlib.pyplot as plt

        %matplotlib inline
```

```
In [10]: # Parse the train and val losses one line at a time.
         import re
         # regexes to find train and val losses on a line
         float_regex = r'[-+]?(\d+(\.\d*)?|\.\d+)([eE][-+]?\d+)?'
         train_loss_re = re.compile('.*Train Loss: ({})'.format(float_regex))
         val_loss_re = re.compile('.*Val Loss: ({})'.format(float_regex))
         val_acc_re = re.compile('.*Val Acc: ({})'.format(float_regex))
         # extract one loss for each logged iteration
         train_losses = []
         val_losses = []
         val_accs = []
         # NOTE: You may need to change this file name.
         with open('twolayernn.log', 'r') as f:
             for line in f:
                 train_match = train_loss_re.match(line)
                 val_match = val_loss_re.match(line)
                 val_acc_match = val_acc_re.match(line)
                 if train_match:
                     train_losses.append(float(train_match.group(1)))
                 if val_match:
                     val_losses.append(float(val_match.group(1)))
                 if val_acc_match:
                     val_accs.append(float(val_acc_match.group(1)))
```

```
In [12]: fig = plt.figure()
         plt.plot(train_losses, label='train')
plt.plot(val_losses, label='val')
plt.title('TwoLayer Learning Curve')
plt.ylabel('loss')
plt.legend()
fig.savefig('twolayernn_lossvstrain.png')

fig = plt.figure()
plt.plot(val_accs, label='val')
plt.title('TwoLayer Validation Accuracy During Training')
plt.ylabel('accuracy')
plt.legend()
fig.savefig('twolayernn_valaccuracy.png')
```




TwoLayer Validation Accuracy During Training
val
accuracy
35
30
25
20
15
10
0
2
4
6
8

twolayer-filtervis

February 10, 2019

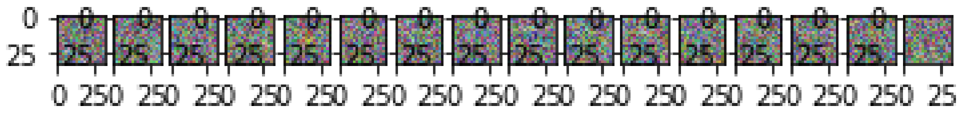
0.1 Visualizing the trained filters

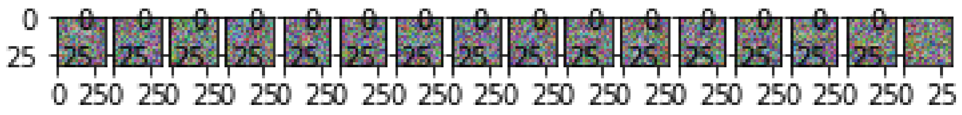
```
In [1]: # some startup!
        import numpy as np
        import matplotlib
        # This is needed to save images
        matplotlib.use('Agg')
        import matplotlib.pyplot as plt
        %matplotlib inline
        plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
        plt.rcParams['image.interpolation'] = 'nearest'
        plt.rcParams['image.cmap'] = 'gray'
        import torch
```

```
In [2]: # load the model saved by train.py
        # This will be an instance of models.softmax.Softmax.
        # NOTE: You may need to change this file name.
        twolayer_model = torch.load('twolayernn.pt')
```

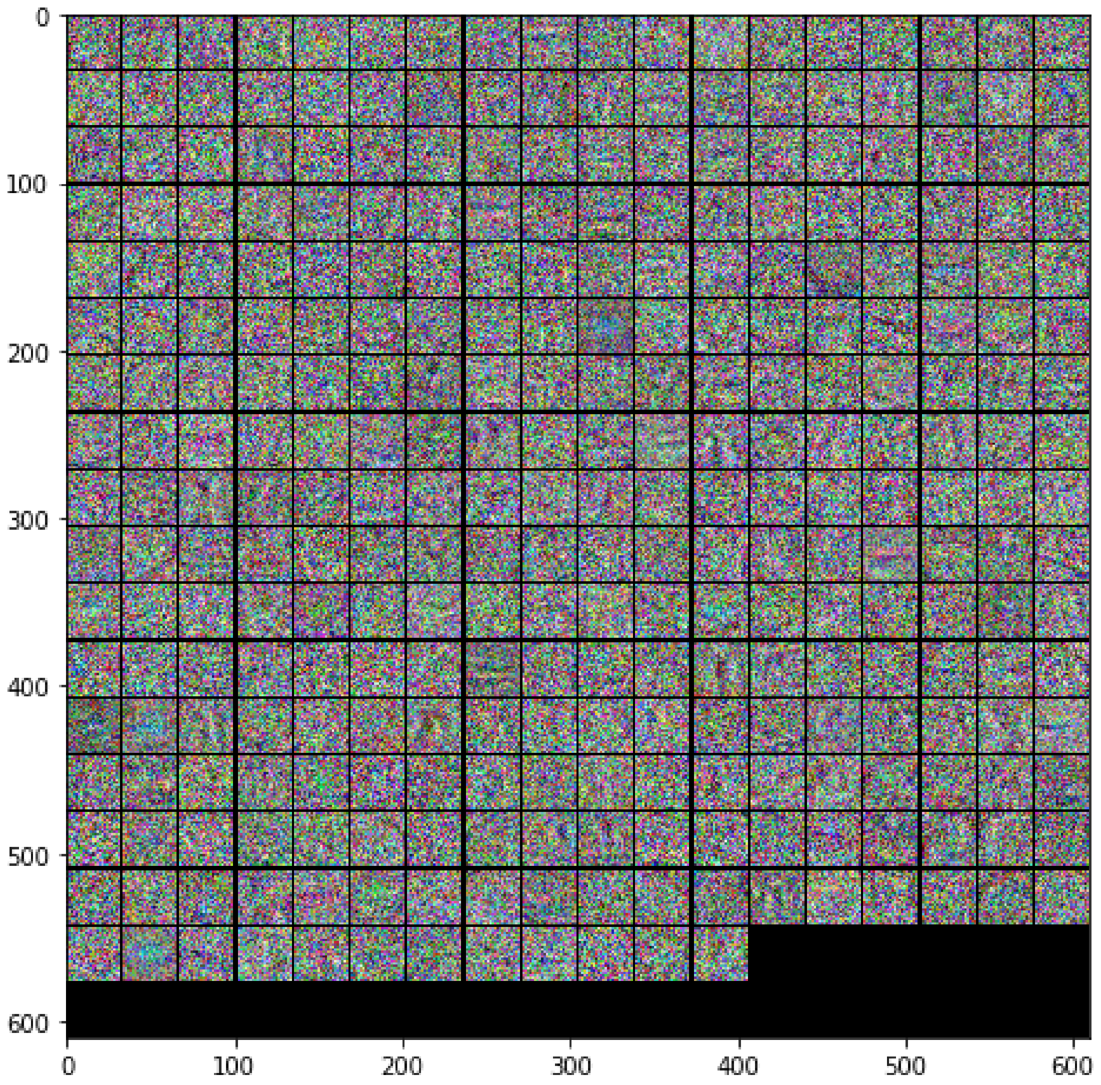
```
In [8]: # collect all the weights
        w1,b1,w2,b2 = [param.data for param in twolayer_model.parameters()]
        print(w1.shape)
        w = w1.view(300, 3, 32, 32) #(N, C, H, W)
        w = w.numpy().transpose(0,2,3,1) #(N, H, W, C)
        # obtain min,max to normalize
        w_min, w_max = np.min(w), np.max(w)
        # classes
        classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck
        # init figure
        fig = plt.figure(figsize=(6,6))
        for i in range(32):
            wimg = 255.0*(w[i].squeeze() - w_min) / (w_max - w_min)
            # subplot is (2,5) as ten filters are to be visualized
            fig.add_subplot(2,16,i+1).imshow(wimg.astype('uint8'))
        # save fig!
        #fig.show()
        fig.savefig('twolayernn_gridfilt.png')
        print('figure saved')
```

```
torch.Size([300, 3072])
figure saved
```





```
In [9]: # vis_utils.py has helper code to view multiple filters in single image. Use this to vis
        # neural network adn convnets.
        # import vis_utils
        from vis_utils import visualize_grid
        # saving the weights is now as simple as:
        grid = visualize_grid(w, padding = 2).astype('uint8')
        plt.imshow(grid)
        plt.show()
        plt.imsave('twolayernn_gridfilt.png',grid)
        # padding is the space between images. Make sure that w is of shape: (N,H,W,C)
        print('figure saved as a grid!')
```



```
figure saved as a grid!
```

softmax-classifier

February 10, 2019

0.1 PyTorch data

PyTorch comes with a nice paradigm for dealing with data which we'll use here. A PyTorch `Dataset` knows where to find data in its raw form (files on disk) and how to load individual examples into Python datastructures. A PyTorch `DataLoader` takes a dataset and offers a variety of ways to sample batches from that dataset.

Take a moment to browse through the `CIFAR10 Dataset` in `2_pytorch/cifar10.py`, read the `DataLoader` documentation linked above, and see how these are used in the section of `train.py` that loads data. Note that in the first part of the homework we subtracted a mean CIFAR10 image from every image before feeding it in to our models. Here we subtract a constant color instead. Both methods are seen in practice and work equally well.

PyTorch provides lots of vision datasets which can be imported directly from `torchvision.datasets`. Also see `torchtext` for natural language datasets.

0.2 Softmax Classifier in PyTorch

In PyTorch Deep Learning building blocks are implemented in the neural network module `torch.nn` (usually imported as `nn`). A PyTorch model is typically a subclass of `nn.Module` and thereby gains a multitude of features. Because your logistic regressor is an `nn.Module` all of its parameters and sub-modules are accessible through the `.parameters()` and `.modules()` methods.

Now implement a softmax classifier by filling in the marked sections of `models/softmax.py`.

The main driver for this question is `train.py`. It reads arguments and model hyperparameter from the command line, loads CIFAR10 data and the specified model (in this case, softmax). Using the optimizer initialized with appropriate hyperparameters, it trains the model and reports performance on test data.

Complete the following couple of sections in `train.py`: 1. Initialize an optimizer from the torch.optim package 2. Update the parameters in model using the optimizer initialized above

At this point all of the components required to train the softmax classifer are complete for the softmax classifier. Now run

```
$ run_softmax.sh
```

to train a model and save it to `softmax.pt`. This will also produce a `softmax.log` file which contains training details which we will visualize below.

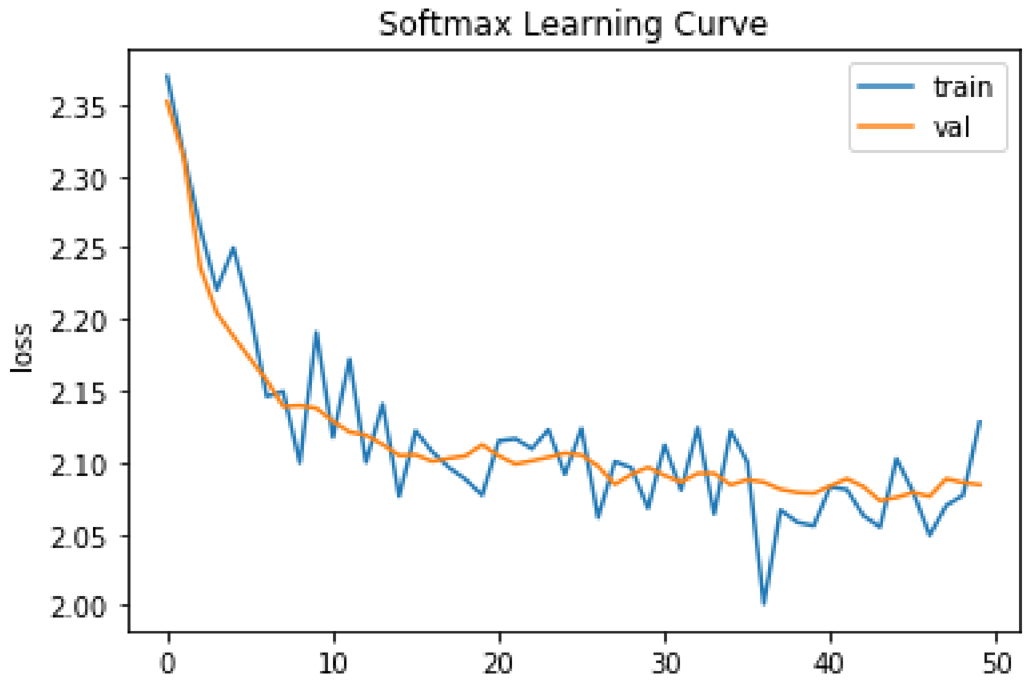
Note: You may want to adjust the hyperparameters specified in `run_softmax.sh` to get reasonable performance.

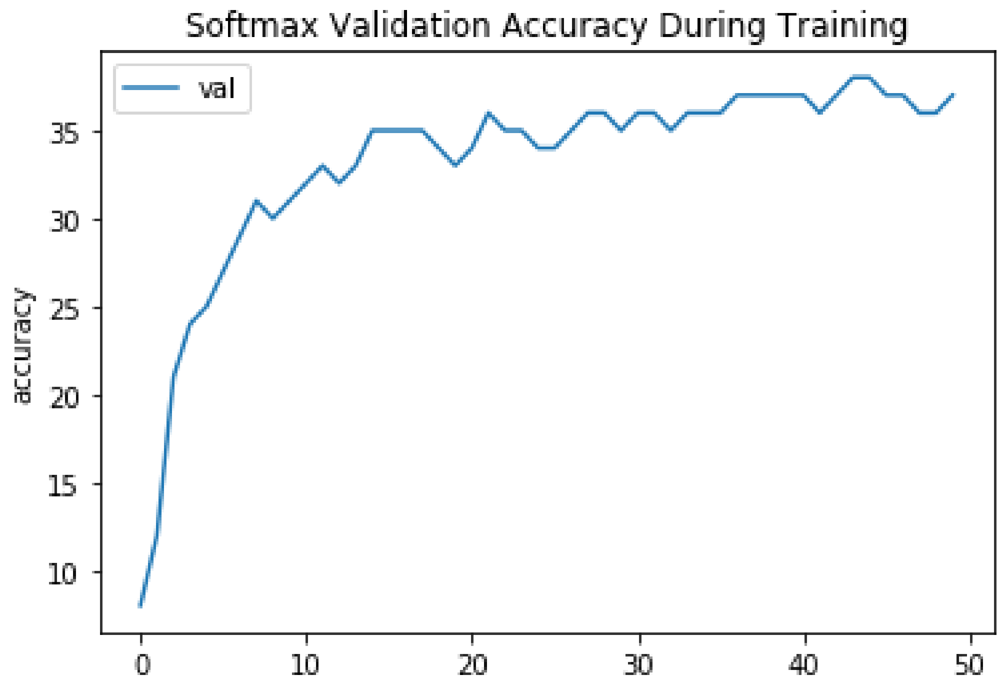
0.3 Visualizing the PyTorch model

```
In [7]: # Assuming that you have completed training the classifer, let us plot the training loss
        # example to show a simple way to log and plot data from PyTorch.

        # we neeed matplotlib to plot the graphs for us!
        import matplotlib
        # This is needed to save images
        matplotlib.use('Agg')
        import matplotlib.pyplot as plt
        %matplotlib inline
```

```
In [8]: # Parse the train and val losses one line at a time.
        import re
        # regexes to find train and val losses on a line
        float_regex = r'[-+]?(\d+(\.\d*)?|\.\d+)([eE][-+]?\d+)?'
        train_loss_re = re.compile('.*Train Loss: ({})'.format(float_regex))
        val_loss_re = re.compile('.*Val Loss: ({})'.format(float_regex))
        val_acc_re = re.compile('.*Val Acc: ({})'.format(float_regex))
        # extract one loss for each logged iteration
        train_losses = []
        val_losses = []
        val_accs = []
        # NOTE: You may need to change this file name.
        with open('softmax.log', 'r') as f:
            for line in f:
                train_match = train_loss_re.match(line)
                val_match = val_loss_re.match(line)
                val_acc_match = val_acc_re.match(line)
                if train_match:
                    train_losses.append(float(train_match.group(1)))
                if val_match:
                    val_losses.append(float(val_match.group(1)))
                if val_acc_match:
                    val_accs.append(float(val_acc_match.group(1)))
```

```
In [9]: fig = plt.figure()
        plt.plot(train_losses, label='train')
        plt.plot(val_losses, label='val')
        plt.title('Softmax Learning Curve')
        plt.ylabel('loss')
        plt.legend()
        fig.savefig('softmax_lossvstrain.png')

        fig = plt.figure()
        plt.plot(val_accs, label='val')
        plt.title('Softmax Validation Accuracy During Training')
        plt.ylabel('accuracy')
        plt.legend()
        fig.savefig('softmax_valaccuracy.png')
```





filter-viz

February 10, 2019

0.1 Visualizing the trained filters

```
In [1]: # some startup!
        import numpy as np
        import matplotlib
        # This is needed to save images
        matplotlib.use('Agg')
        import matplotlib.pyplot as plt
        import torch

In [2]: # load the model saved by train.py
        # This will be an instance of models.softmax.Softmax.
        # NOTE: You may need to change this file name.
        softmax_model = torch.load('softmax.pt')

In [3]: # collect all the weights
        w,b = [param.data for param in softmax_model.parameters()]
        w = w.view(10, 3, 32, 32) #(N, C, H, W)
        w = torch.transpose(w, 1, 3).numpy() #(N, H, W, C)
        #############################################################################
        # TODO: Extract the weight matrix (without bias) from softmax_model, convert
        # it to a numpy array with shape (10, 32, 32, 3), and assign this array to w.
        # The first dimension should be for channels, then height, width, and color.
        # This step depends on how you implemented models.softmax.Softmax.
        #############################################################################

        #############################################################################
        #                             END OF YOUR CODE                              #
        #############################################################################
        # obtain min,max to normalize
        w_min, w_max = np.min(w), np.max(w)
        # classes
        classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck
        # init figure
        fig = plt.figure(figsize=(6,6))
        for i in range(10):
            wimg = 255.0*(w[i].squeeze() - w_min) / (w_max - w_min)
            # subplot is (2,5) as ten filters are to be visualized
            fig.add_subplot(2,5,i+1).imshow(wimg.astype('uint8'))
        # save fig!
        fig.show()
        fig.savefig('softmax_filt.png')
        print('figure saved')
```

```
figure saved
```

```
/Users/alexisdurocher/anaconda3/lib/python3.6/site-packages/matplotlib/figure.py:445: UserWarnin
  % get_backend())
```

```
In [4]: # vis_utils.py has helper code to view multiple filters in single image. Use this to vis
        # neural network adn convnets.
        # import vis_utils
        from vis_utils import visualize_grid
        # saving the weights is now as simple as:
        plt.imsave('softmax_gridfilt.png',visualize_grid(w, padding=3).astype('uint8'))
        # padding is the space between images. Make sure that w is of shape: (N,H,W,C)
        print('figure saved as a grid!')
```

```
figure saved as a grid!
```

convnet

February 10, 2019

1 Train a ConvNet!

We now have a generic solver and a bunch of modularized layers. It's time to put it all together, and train a ConvNet to recognize the classes in CIFAR-10. In this notebook we will walk you through training a simple two-layer ConvNet and then set you free to build the best net that you can to perform well on CIFAR-10.

Open up the file `cs231n/classifiers/convnet.py`; you will see that the `two_layer_convnet` function computes the loss and gradients for a two-layer ConvNet. Note that this function uses the "sandwich" layers defined in `cs231n/layer_utils.py`.

```
In [1]: # As usual, a bit of setup

        import numpy as np
        import matplotlib.pyplot as plt
        from cs231n.classifier_trainer import ClassifierTrainer
        from cs231n.gradient_check import eval_numerical_gradient
        from cs231n.classifiers.convnet import *

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

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

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

```
In [2]: from cs231n.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 = 'cs231n/datasets/cifar-10-batches-py'
            X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

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

            # Transpose so that channels come first
            X_train = X_train.transpose(0, 3, 1, 2).copy()
            X_val = X_val.transpose(0, 3, 1, 2).copy()
            x_test = X_test.transpose(0, 3, 1, 2).copy()

            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, 3, 32, 32)
Train labels shape:  (49000,)
Validation data shape:  (1000, 3, 32, 32)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
```

2 Sanity check loss

After you build a new network, one of the first things you should do is sanity check the loss. When we use the softmax loss, we expect the loss for random weights (and no regularization) to be about `log(C)` for `C` classes. When we add regularization this should go up.

```
In [3]: model = init_two_layer_convnet()

        X = np.random.randn(100, 3, 32, 32)
        y = np.random.randint(10, size=100)

        loss, _ = two_layer_convnet(X, model, y, reg=0)

        # Sanity check: Loss should be about log(10) = 2.3026
        print('Sanity check loss (no regularization): ', loss)

        # Sanity check: Loss should go up when you add regularization
        loss, _ = two_layer_convnet(X, model, y, reg=1)
        print('Sanity check loss (with regularization): ', loss)
```

```
Sanity check loss (no regularization):  2.302589986326018
Sanity check loss (with regularization):  2.344903490002597
```

3 Gradient check

After the loss looks reasonable, you should always use numeric gradient checking to make sure that your backward pass is correct. When you use numeric gradient checking you should use a small amount of artifical data and a small number of neurons at each layer.

```
In [4]: num_inputs = 2
        input_shape = (3, 16, 16)
        reg = 0.0
        num_classes = 10
        X = np.random.randn(num_inputs, *input_shape)
        y = np.random.randint(num_classes, size=num_inputs)

        model = init_two_layer_convnet(num_filters=3, filter_size=3, input_shape=input_shape)
        loss, grads = two_layer_convnet(X, model, y)
        for param_name in sorted(grads):
            f = lambda _: two_layer_convnet(X, model, y)[0]
            param_grad_num = eval_numerical_gradient(f, model[param_name], verbose=False, h=1e-6
            e = rel_error(param_grad_num, grads[param_name])
            print('%s max relative error: %e' % (param_name, rel_error(param_grad_num, grads[par
```

```
W1 max relative error: 2.159361e-07
W2 max relative error: 1.545029e-05
b1 max relative error: 2.058688e-08
```

```
b2 max relative error: 1.502365e-09
```

4 Overfit small data

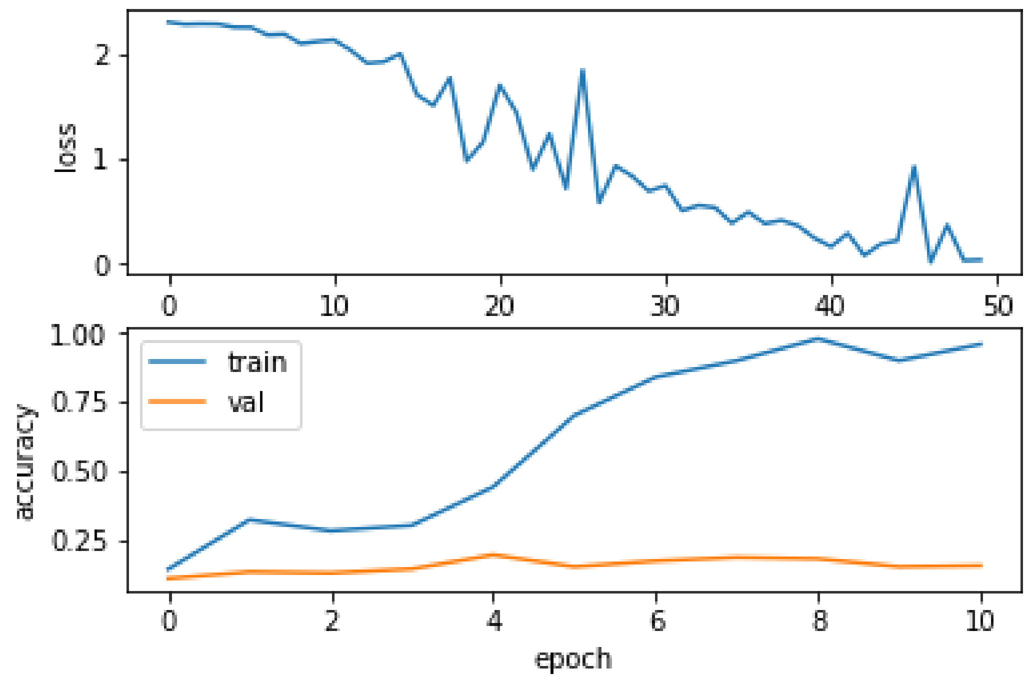
A nice trick is to train your model with just a few training samples. You should be able to overfit small datasets, which will result in very high training accuracy and comparatively low validation accuracy.

```
In [5]: # Use a two-layer ConvNet to overfit 50 training examples.

        model = init_two_layer_convnet()
        trainer = ClassifierTrainer()
        best_model, loss_history, train_acc_history, val_acc_history = trainer.train(
                  X_train[:50], y_train[:50], X_val, y_val, model, two_layer_convnet,
                  reg=0.001, momentum=0.9, learning_rate=0.0001, batch_size=10, num_epochs=10,
                  verbose=True)
```

```
starting iteration  0
Finished epoch 0 / 10: cost 2.308428, train: 0.140000, val 0.106000, lr 1.000000e-04
Finished epoch 1 / 10: cost 2.259305, train: 0.320000, val 0.130000, lr 9.500000e-05
Finished epoch 2 / 10: cost 2.123196, train: 0.280000, val 0.126000, lr 9.025000e-05
starting iteration  10
Finished epoch 3 / 10: cost 2.006251, train: 0.300000, val 0.141000, lr 8.573750e-05
Finished epoch 4 / 10: cost 1.162897, train: 0.440000, val 0.192000, lr 8.145062e-05
starting iteration  20
Finished epoch 5 / 10: cost 0.716435, train: 0.700000, val 0.149000, lr 7.737809e-05
Finished epoch 6 / 10: cost 0.692818, train: 0.840000, val 0.170000, lr 7.350919e-05
starting iteration  30
Finished epoch 7 / 10: cost 0.386743, train: 0.900000, val 0.183000, lr 6.983373e-05
Finished epoch 8 / 10: cost 0.242038, train: 0.980000, val 0.178000, lr 6.634204e-05
starting iteration  40
Finished epoch 9 / 10: cost 0.218968, train: 0.900000, val 0.149000, lr 6.302494e-05
Finished epoch 10 / 10: cost 0.036106, train: 0.960000, val 0.153000, lr 5.987369e-05
finished optimization. best validation accuracy: 0.192000
```

Plotting the loss, training accuracy, and validation accuracy should show clear overfitting:

```
In [6]: plt.subplot(2, 1, 1)
        plt.plot(loss_history)
        plt.xlabel('iteration')
        plt.ylabel('loss')

        plt.subplot(2, 1, 2)
        plt.plot(train_acc_history)
        plt.plot(val_acc_history)
        plt.legend(['train', 'val'], loc='upper left')
        plt.xlabel('epoch')
        plt.ylabel('accuracy')
        plt.show()
```



5 Train the net

Once the above works, training the net is the next thing to try. You can set the `acc_frequency` parameter to change the frequency at which the training and validation set accuracies are tested. If your parameters are set properly, you should see the training and validation accuracy start to improve within a hundred iterations, and you should be able to train a reasonable model with just one epoch.

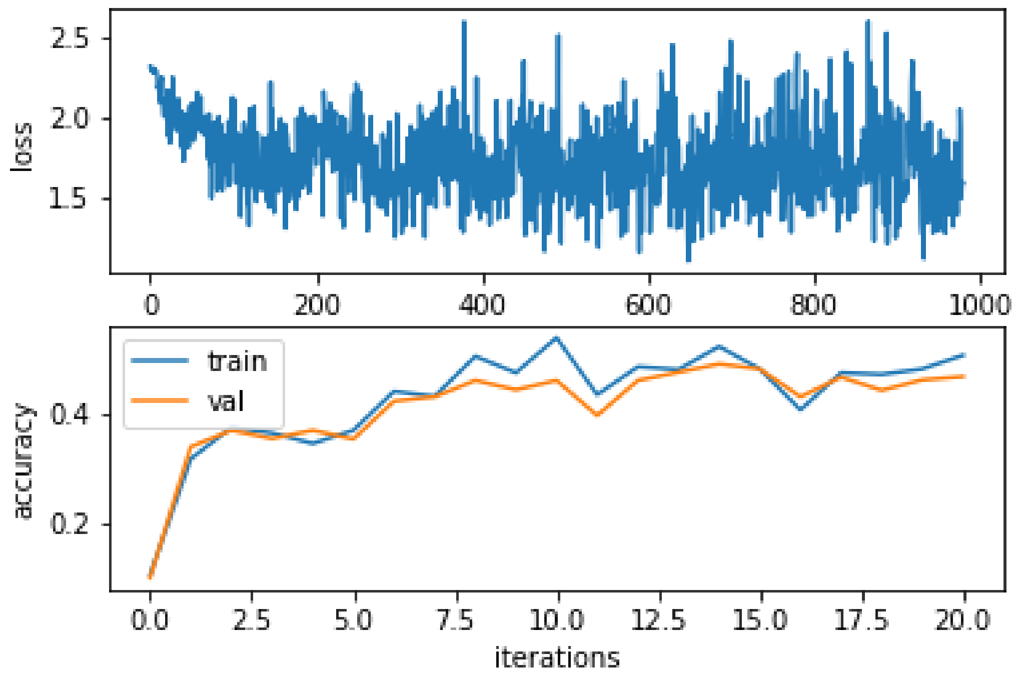
Using the parameters below you should be able to get at least 48% accuracy on the validation set.

```
In [7]: model = init_two_layer_convnet(filter_size=7)
        trainer = ClassifierTrainer()
        best_model, loss_history, train_acc_history, val_acc_history = trainer.train(
                  X_train, y_train, X_val, y_val, model, two_layer_convnet,
                  reg=0.001, momentum=0.9, learning_rate=0.0001, batch_size=50, num_epochs=1,
                  acc_frequency=50, verbose=False)
```

```
In [13]: print(f'Best validation accuracy is {max(val_acc_history)}')
```

```
Best validation accuracy is 0.489
```

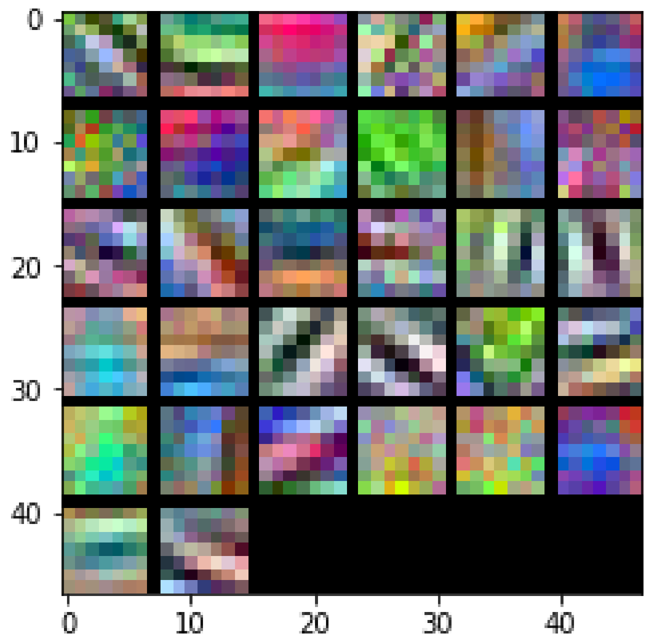
```
In [14]: plt.subplot(2, 1, 1)
         plt.plot(loss_history)
         plt.xlabel('iteration')
         plt.ylabel('loss')

         plt.subplot(2, 1, 2)
         plt.plot(train_acc_history)
         plt.plot(val_acc_history)
         plt.legend(['train', 'val'], loc='upper left')
         plt.xlabel('iterations')
         plt.ylabel('accuracy')
         plt.show()
```



6 Visualize weights

We can visualize the convolutional weights from the first layer. If everything worked properly, these will usually be edges and blobs of various colors and orientations.

```
In [8]: from cs231n.vis_utils import visualize_grid

        grid = visualize_grid(best_model['W1'].transpose(0, 2, 3, 1))
        plt.imshow(grid.astype('uint8'))

Out[8]: <matplotlib.image.AxesImage at 0x120a73940>
```


0
10
20
30
40
0
10
20
30
40

convnet-filtervis

February 10, 2019

0.1 Visualizing the trained filters

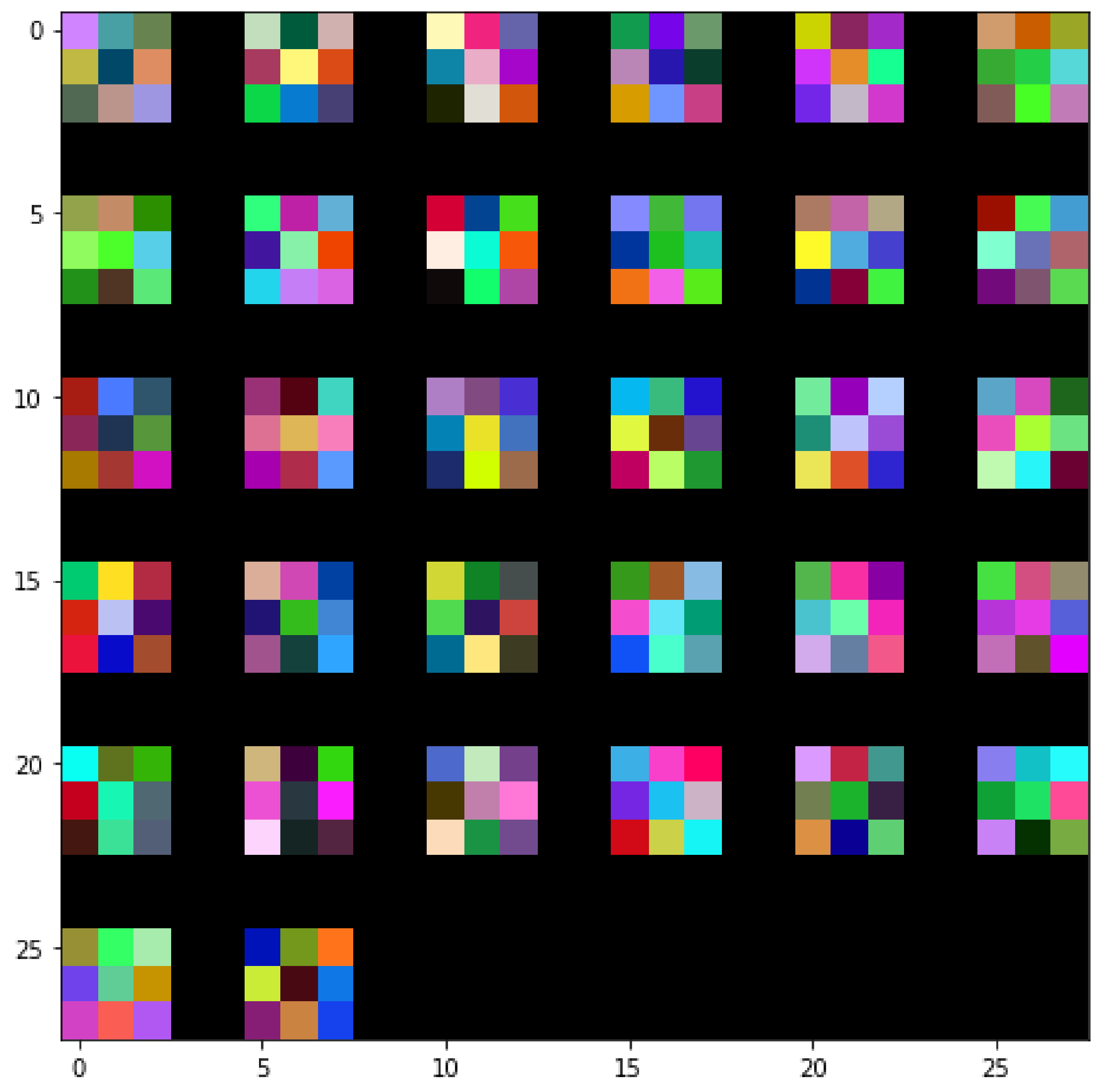
```
In [1]: # some startup!
        import numpy as np
        import matplotlib
        # This is needed to save images
        matplotlib.use('Agg')
        import matplotlib.pyplot as plt
        %matplotlib inline
        plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
        plt.rcParams['image.interpolation'] = 'nearest'
        plt.rcParams['image.cmap'] = 'gray'

        import torch

In [2]: # load the model saved by train.py
        # This will be an instance of models.softmax.Softmax.
        # NOTE: You may need to change this file name.
        convnet_model = torch.load('convnet.pt')

In [5]: # collect all the weights
        w1,b1,w2,b2 = [param.data for param in convnet_model.parameters()]
        print(w1.shape)
        w = w1.view(32, 3, 3, 3) #(N, C, H, W)
        w = w.numpy().transpose(0,2,3,1) #(N, H, W, C)
        # obtain min,max to normalize
        w_min, w_max = np.min(w), np.max(w)
        # classes
        classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck
        # init figure
        fig = plt.figure(figsize=(6,6))
        for i in range(32):
            wimg = 255.0*(w[i].squeeze() - w_min) / (w_max - w_min)
            # subplot is (2,5) as ten filters are to be visualized
            fig.add_subplot(2,16,i+1).imshow(wimg.astype('uint8'))
        # save fig!
        fig.show()
        fig.savefig('convnet_gridfilt.png')
        print('figure saved')
```

```
torch.Size([32, 3, 3, 3])
```

```
/Users/alexisdurocher/anaconda3/lib/python3.6/site-packages/matplotlib/figure.py:445: UserWarnin
  % get_backend())
```

```
figure saved
```

```
In [6]: # vis_utils.py has helper code to view multiple filters in single image. Use this to vis
        # neural network adn convnets.
        # import vis_utils
        from vis_utils import visualize_grid
        # saving the weights is now as simple as:
        grid = visualize_grid(w, padding = 2).astype('uint8')
        plt.imshow(grid)
        plt.show()
        plt.imsave('convnet_gridfilt.png',grid)
        # padding is the space between images. Make sure that w is of shape: (N,H,W,C)
        print('figure saved as a grid!')
```



```
figure saved as a grid!
```