

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, hidden_size)
    model['b1'] = np.linspace(-0.3, 0.7, num=hidden_size).reshape(hidden_size, 1)
    model['W2'] = np.linspace(-0.4, 0.1, num=hidden_size*num_classes).reshape(hidden_size, num_classes)
    model['b2'] = np.linspace(-0.5, 0.9, num=num_classes).reshape(num_classes, 1)
    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()
```

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 SVM and Softmax exercises: 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:
3.84868230029e-08
```

Forward pass: compute loss

In the same function, implement the second part that computes the data and regularization 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:
4.67692551354e-12
```

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),
    print '%s max relative error: %e' % (param_name, rel_error(param_grad_num, grads[param_name]))

W1 max relative error: 4.426512e-09
W2 max relative error: 1.786138e-09
b2 max relative error: 8.190173e-11
b1 max relative error: 5.435436e-08
```

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 Desc
best_model, loss_history, _, _ = trainer.train(X, y, X, y,
                                              model, two_layer_net,
                                              reg=0.001,
                                              learning_rate=1e-1, momentum=0.0, le
                                              update='sgd', sample_batches=False,
                                              num_epochs=100,
                                              verbose=False)

print 'Final loss with vanilla SGD: %f' % (loss_history[-1], )

starting iteration  0
starting iteration 10
starting iteration 20
starting iteration 30
starting iteration 40
starting iteration 50
starting iteration 60
starting iteration 70
starting iteration 80
starting iteration 90
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
best_model, loss_history, _, _ = trainer.train(X, y, X, y,
                                              model, two_layer_net,
                                              reg=0.001,
                                              learning_rate=1e-1, momentum=0.9,
                                              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)
```

starting iteration 0
starting iteration 10
starting iteration 20
starting iteration 30
starting iteration 40
starting iteration 50
starting iteration 60
starting iteration 70
starting iteration 80
starting iteration 90
Final loss with momentum SGD: 0.494394. We get: 0.494394

Now also 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
best_model, loss_history, _, _ = trainer.train(X, y, X, y,
                                              model, two_layer_net,
                                              reg=0.001,
                                              learning_rate=1e-1, momentum=0.9,
                                              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)
```

starting iteration 0
starting iteration 10
starting iteration 20
starting iteration 30
starting iteration 40
starting iteration 50
starting iteration 60
starting iteration 70
starting iteration 80
starting iteration 90
Final loss with RMSProp: 0.429848. We get: 0.439368

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. 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

    # 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: (49000L, 3072L)
Train labels shape: (49000L,)
Validation data shape: (1000L, 3072L)
Validation labels shape: (1000L,)
Test data shape: (1000L, 3072L)
Test labels shape: (1000L,)
```

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 [18]: 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 o
trainer = ClassifierTrainer()
best_model, loss_history, train_acc, val_acc = trainer.train(X_train, y_train, X_
                                                    model, two_layer_net,
                                                    num_epochs=20, reg=0.4,
                                                    momentum=0.9, learning_rate_decay =
                                                    learning_rate=1e-4, verbose=True)

starting iteration  0
Finished epoch 0 / 20: cost 2.302588, train: 0.123000, val 0.097000, lr 1.000
000e-04
starting iteration  10
starting iteration  20
starting iteration  30
starting iteration  40
starting iteration  50
starting iteration  60
starting iteration  70
starting iteration  80
starting iteration  90
starting iteration 100
starting iteration 110
starting iteration 120
starting iteration 130
starting iteration 140
starting iteration 150
starting iteration 160
starting iteration 170
```

Debug the training

With the default parameters we provided above, you should get a validation accuracy of about 0.37 on the validation set. This isn't very good.

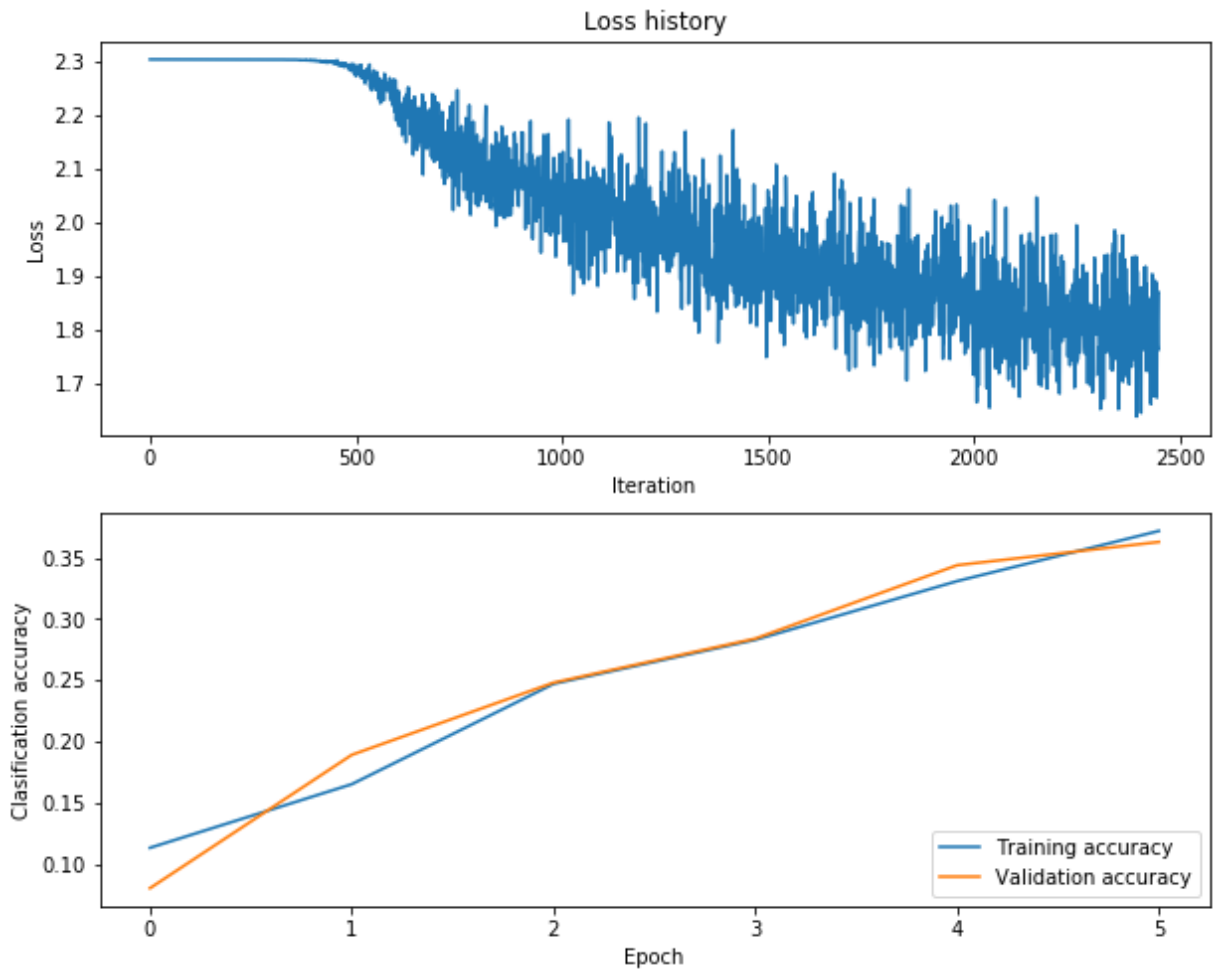
One strategy for getting insight into what's wrong is to plot the loss function and the accuracies on the training and validation sets during optimization.

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 [11]: # 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[11]: <matplotlib.text.Text at 0x7836ba8>

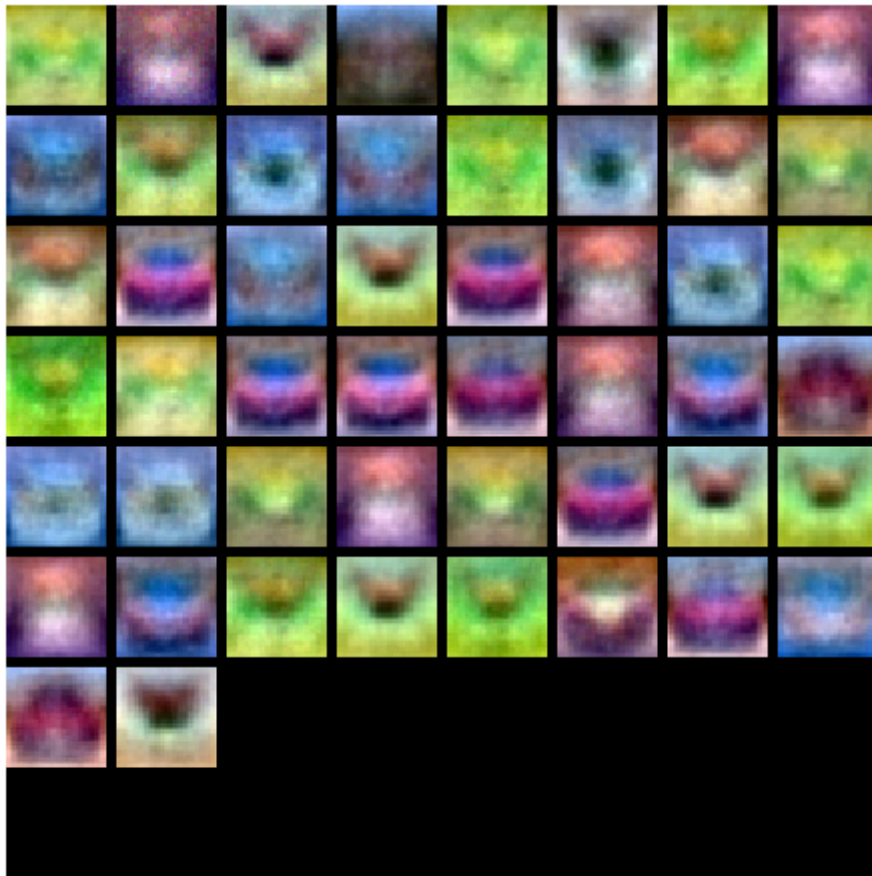


```
In [12]: from cs231n.vis_utils import visualize_grid

# 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).as
    plt.gca().axis('off')
    plt.show()

show_net_weights(model)
```



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, number 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.

Approximate results. You should be aim to achieve a classification accuracy of greater than 50% on the validation set. Our best network gets over 56% on the validation set.

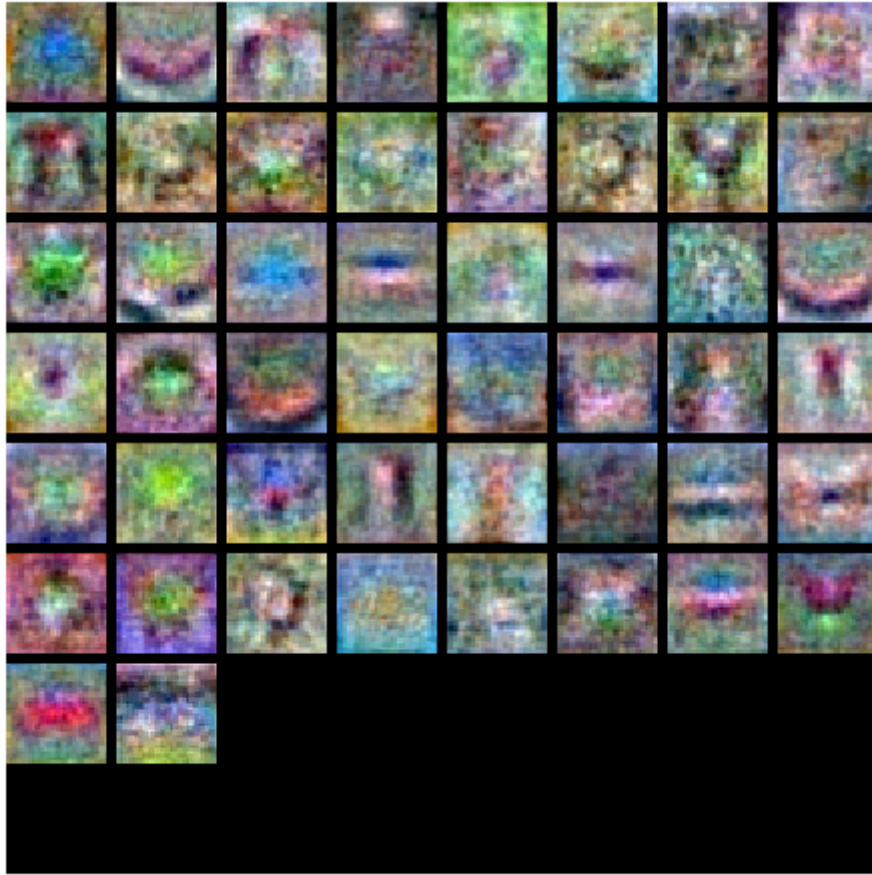
Experiment: Your goal in this exercise is to get as good of a result on CIFAR-10 as you can, with a fully-connected Neural Network. For every 1% above 56% on the Test set we will award you with one extra bonus point. Feel free implement your own techniques (e.g. PCA to reduce dimensionality, or adding dropout, or adding features to the solver, etc.).

In [16]: `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 like we did on the previous assignment. #
#####
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number o
trainer = ClassifierTrainer()
best_model, loss_history, train_acc, val_acc = trainer.train(X_train, y_train, X_
                                model, two_layer_net,
                                num_epochs=20, reg=0.4,
                                momentum=0.9, learning_rate_decay =
                                learning_rate=1e-4, verbose=True)
#####
#                                END OF YOUR CODE                                #
#####
```

```
starting iteration 0
Finished epoch 0 / 20: cost 2.302588, train: 0.120000, val 0.112000, lr 1.000
000e-04
starting iteration 10
starting iteration 20
starting iteration 30
starting iteration 40
starting iteration 50
starting iteration 60
starting iteration 70
starting iteration 80
starting iteration 90
starting iteration 100
starting iteration 110
starting iteration 120
starting iteration 130
starting iteration 140
starting iteration 150
starting iteration 160
starting iteration 170
```

```
In [14]: # visualize the weights  
show_net_weights(best_model)
```



Run on the test set

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

We will give you extra bonus point for every 1% of accuracy above 56%.

```
In [17]: scores_test = two_layer_net(X_test, best_model)  
print 'Test accuracy: ', np.mean(np.argmax(scores_test, axis=1) == y_test)  
  
Test accuracy:  0.512
```

In []:

In []: