convnet

September 27, 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):  2.3024743101644125
Sanity check loss (with regularization):  2.344555140008056
```

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: 6.949192e-07
W2 max relative error: 5.067411e-04
b1 max relative error: 1.643673e-08
```

```
b2 max relative error: 2.099409e-09
```

4 Overfit small data

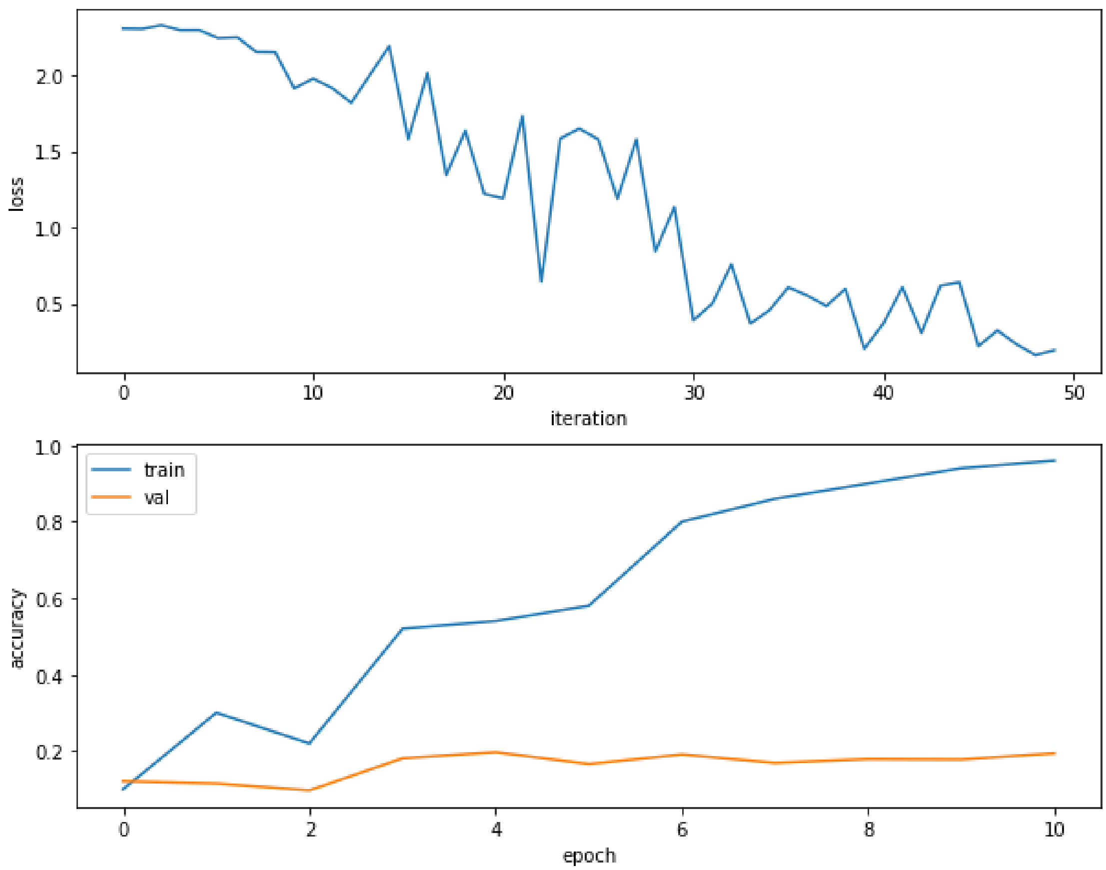
A nice trick is to train your model with just a few training samples. You should be able to overfit small datasets, which will result in very high training accuracy and comparatively low validation accuracy.

```
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 2.303622, train: 0.100000, val 0.121000, lr 1.000000e-04
Finished epoch 1 / 10: cost 2.293518, train: 0.300000, val 0.115000, lr 9.500000e-05
Finished epoch 2 / 10: cost 1.911979, train: 0.220000, val 0.097000, lr 9.025000e-05
Finished epoch 3 / 10: cost 2.189026, train: 0.520000, val 0.181000, lr 8.573750e-05
Finished epoch 4 / 10: cost 1.220391, train: 0.540000, val 0.197000, lr 8.145062e-05
Finished epoch 5 / 10: cost 1.647710, train: 0.580000, val 0.166000, lr 7.737809e-05
Finished epoch 6 / 10: cost 1.133029, train: 0.800000, val 0.191000, lr 7.350919e-05
Finished epoch 7 / 10: cost 0.456323, train: 0.860000, val 0.169000, lr 6.983373e-05
Finished epoch 8 / 10: cost 0.204952, train: 0.900000, val 0.179000, lr 6.634204e-05
Finished epoch 9 / 10: cost 0.641365, train: 0.940000, val 0.178000, lr 6.302494e-05
Finished epoch 10 / 10: cost 0.195545, train: 0.960000, val 0.194000, lr 5.987369e-05
finished optimization. best validation accuracy: 0.197000
```

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 around 50% 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=2,
                  acc_frequency=50, verbose=True)
```

```
starting iteration  0
Finished epoch 0 / 2: cost 2.290782, train: 0.129000, val 0.132000, lr 1.000000e-04
```

```
Finished epoch 0 / 2: cost 1.819116, train: 0.310000, val 0.320000, lr 1.000000e-04
starting iteration  100
Finished epoch 0 / 2: cost 1.775723, train: 0.374000, val 0.363000, lr 1.000000e-04
Finished epoch 0 / 2: cost 2.232364, train: 0.403000, val 0.401000, lr 1.000000e-04
starting iteration  200
Finished epoch 0 / 2: cost 1.838677, train: 0.407000, val 0.420000, lr 1.000000e-04
Finished epoch 0 / 2: cost 1.896676, train: 0.426000, val 0.461000, lr 1.000000e-04
starting iteration  300
Finished epoch 0 / 2: cost 1.658628, train: 0.402000, val 0.418000, lr 1.000000e-04
Finished epoch 0 / 2: cost 1.510652, train: 0.441000, val 0.455000, lr 1.000000e-04
starting iteration  400
Finished epoch 0 / 2: cost 1.576475, train: 0.444000, val 0.425000, lr 1.000000e-04
Finished epoch 0 / 2: cost 1.617816, train: 0.489000, val 0.483000, lr 1.000000e-04
starting iteration  500
Finished epoch 0 / 2: cost 1.671812, train: 0.438000, val 0.469000, lr 1.000000e-04
Finished epoch 0 / 2: cost 1.645641, train: 0.487000, val 0.458000, lr 1.000000e-04
starting iteration  600
Finished epoch 0 / 2: cost 1.359532, train: 0.456000, val 0.488000, lr 1.000000e-04
Finished epoch 0 / 2: cost 1.652872, train: 0.493000, val 0.478000, lr 1.000000e-04
starting iteration  700
Finished epoch 0 / 2: cost 1.283509, train: 0.496000, val 0.463000, lr 1.000000e-04
Finished epoch 0 / 2: cost 1.468094, train: 0.488000, val 0.468000, lr 1.000000e-04
starting iteration  800
Finished epoch 0 / 2: cost 1.964609, train: 0.487000, val 0.501000, lr 1.000000e-04
Finished epoch 0 / 2: cost 1.861586, train: 0.486000, val 0.483000, lr 1.000000e-04
starting iteration  900
Finished epoch 0 / 2: cost 1.716851, train: 0.478000, val 0.484000, lr 1.000000e-04
Finished epoch 0 / 2: cost 1.555007, train: 0.477000, val 0.457000, lr 1.000000e-04
Finished epoch 1 / 2: cost 1.532074, train: 0.485000, val 0.513000, lr 9.500000e-05
starting iteration  1000
Finished epoch 1 / 2: cost 1.545110, train: 0.511000, val 0.463000, lr 9.500000e-05
Finished epoch 1 / 2: cost 1.567729, train: 0.506000, val 0.505000, lr 9.500000e-05
starting iteration  1100
Finished epoch 1 / 2: cost 1.924106, train: 0.545000, val 0.495000, lr 9.500000e-05
Finished epoch 1 / 2: cost 1.402845, train: 0.574000, val 0.529000, lr 9.500000e-05
starting iteration  1200
Finished epoch 1 / 2: cost 1.308199, train: 0.524000, val 0.489000, lr 9.500000e-05
Finished epoch 1 / 2: cost 1.493040, train: 0.506000, val 0.500000, lr 9.500000e-05
starting iteration  1300
Finished epoch 1 / 2: cost 1.167977, train: 0.521000, val 0.507000, lr 9.500000e-05
Finished epoch 1 / 2: cost 1.769514, train: 0.542000, val 0.501000, lr 9.500000e-05
starting iteration  1400
Finished epoch 1 / 2: cost 1.530140, train: 0.472000, val 0.454000, lr 9.500000e-05
Finished epoch 1 / 2: cost 1.531063, train: 0.505000, val 0.464000, lr 9.500000e-05
starting iteration  1500
Finished epoch 1 / 2: cost 1.756947, train: 0.504000, val 0.498000, lr 9.500000e-05
Finished epoch 1 / 2: cost 1.217859, train: 0.527000, val 0.520000, lr 9.500000e-05
starting iteration  1600
```

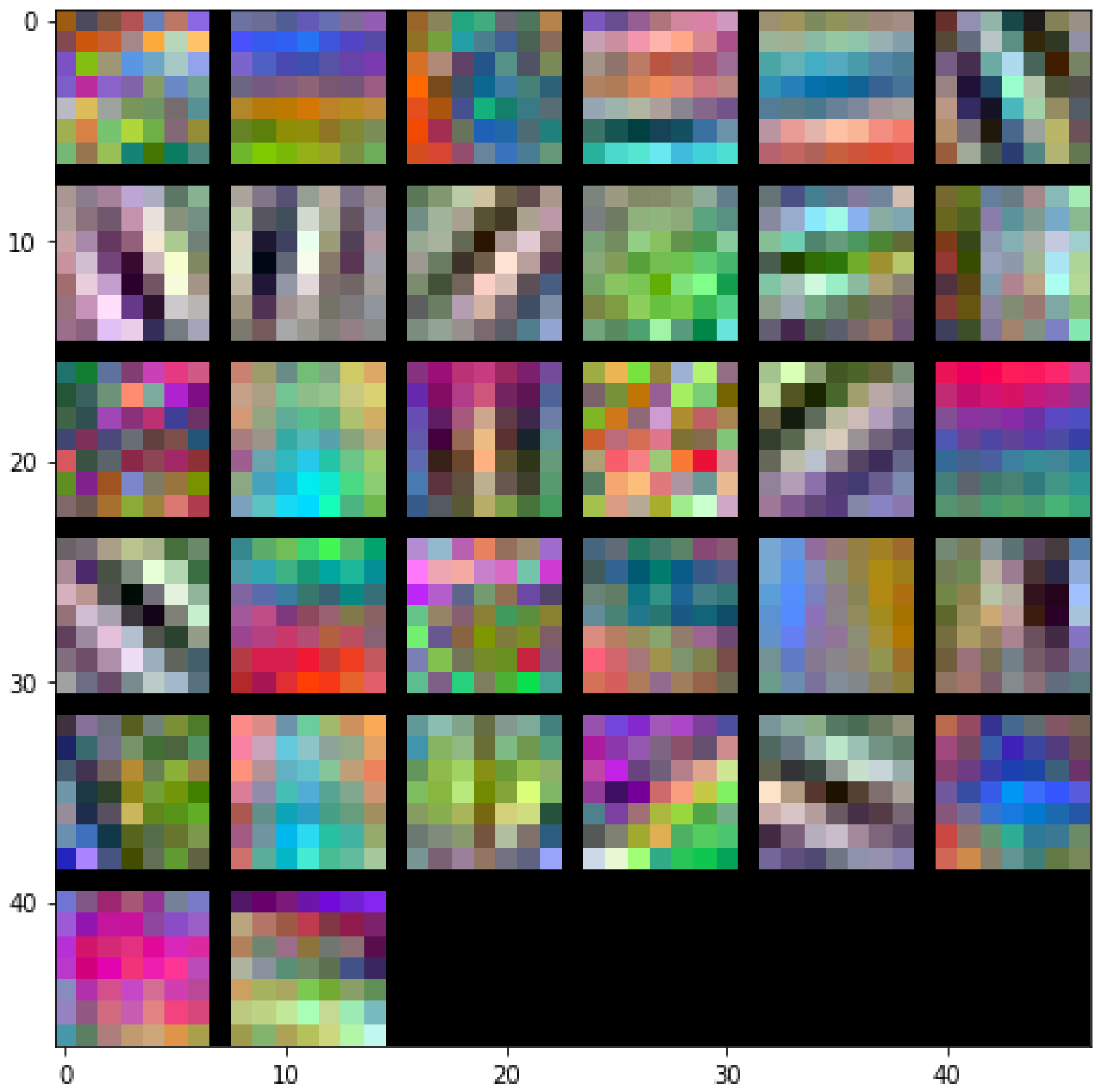
```
Finished epoch 1 / 2: cost 1.315626, train: 0.518000, val 0.522000, lr 9.500000e-05
Finished epoch 1 / 2: cost 1.767651, train: 0.551000, val 0.506000, lr 9.500000e-05
starting iteration  1700
Finished epoch 1 / 2: cost 1.279082, train: 0.509000, val 0.516000, lr 9.500000e-05
Finished epoch 1 / 2: cost 1.497780, train: 0.519000, val 0.479000, lr 9.500000e-05
starting iteration  1800
Finished epoch 1 / 2: cost 1.598472, train: 0.497000, val 0.507000, lr 9.500000e-05
Finished epoch 1 / 2: cost 1.650268, train: 0.562000, val 0.516000, lr 9.500000e-05
starting iteration  1900
Finished epoch 1 / 2: cost 0.884921, train: 0.561000, val 0.526000, lr 9.500000e-05
Finished epoch 1 / 2: cost 1.344762, train: 0.496000, val 0.443000, lr 9.500000e-05
Finished epoch 2 / 2: cost 1.360149, train: 0.463000, val 0.437000, lr 9.025000e-05
finished optimization. best validation accuracy: 0.529000
```

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 0x7f6c4731d1d0>
```


0
10
20
30
40
0
10
20
30
40