CS 5600/6600: F20: Intelligent Systems

DL and ConvNets: Part 2

Vladimir Kulyukin
Department of Computer Science
Utah State University

Outline

ANNs vs. ConvNets

The ConvNet architectures are different from standard ANN architectures: ANNs don't have local receptive fields, shared weights and baises, convolution, and pooling.

ANNs and ConvNets are made of many simple units whose behaviors are determined by the weights and biases; the overall objective is the same: use training data to learn weights and biases; we can still use backpropagation in convolutional and pooling layers.

As far as brain science is concerned, ConvNets don't look like an improvement over ANNs.

ANNs vs. ConvNets

Generally speaking, there are several concepts that make ConvNets more resistant to overfitting than ANNs:

1. Convolutional layers reduce the number of parameters to train: vanishing/exploding gradients are less likely and training is easier;
2. Dropout and convolution tend to reduce overfitting;
3. ReLUs give slightly better performance and tend to train faster;
4. ConvNets advocates have willingness to train for weeks (sometimes months!).

How Deep is Deep?

State-of-the-art ConvNets have dozens/hundreds of hidden layers.

Try not to adopt the popular deeper-than-you attitude. Focus on the problem at hand and keep the number of hidden layers as small as necessary to achieve the required performance.

The real breakthrough in deep learning (DL) is the realization that despite the ANN Universality Theorem it's possible to go beyond the 1- and 2-hidden layer networks and achieve robust results.

Outline

Theano

Theano (`deeplearning.net/software/theano`) is a Python library for multi-dimensional array operations.

Theano makes it much easier to implement backpropagation for ConvNets.

A great feature of Theano is that it can run code either on a CPU or a GPU, when it is available.

Experiments with MNIST Networks

I ran the experiments on the following slides in Python 2 with `ann_theano.py` – this is the `ann.py` code that uses `Theano`.

Two Basic Questions:

1. Is deeper better?
2. Are ReLUs better than sigmoids?

Training a Shallow ANN with Softmax on MNIST

Here's how one can use `ann_theano.py` to train a shallow network with a single hidden layer of 100 hidden neurons followed by a softmax layer that maps 100 hidden neurons to 10 output neurons.

```
>>> from ann_theano import *
>>> from ann_theano import ConvPoolLayer, FullyConnectedLayer, \
                           SoftmaxLayer
>>> train_d, valid_d, test_d = ann_theano.load_data_shared()
>>> mini_batch_size = 10
>>> net = ann_theano([
FullyConnectedLayer(n_in=784, n_out=100),
SoftmaxLayer(n_in=100, n_out=10)],
        mini_batch_size)
>>> net.mini_batch_sgd(training_d, 100, mini_batch_size, 0.1,
                       valid_d, test_d)
```

The best validation accuracy I achieved with this net was 98.00%.

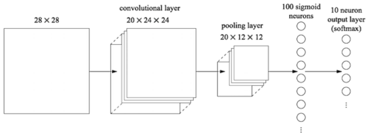
Taking Advantage of a GPU

There's a boolean variable GPU in ann_theano.py. It's set to False by default. Set it to True if you run on a GPU.

```
GPU = False
if GPU:
    print "Trying to run under a GPU.  If this is not desired, then modify "+\
        "ann_theano.py\nto set the GPU flag to False."
    try: theano.config.device = 'gpu'
    except: pass # it's already set
    theano.config.floatX = 'float32'
else:
    print "Running with a CPU.  If this is not desired, then the modify "+\
        "ann_theano.py to set\nthe GPU flag to True."
```

Training 1st ConvNet on MNIST

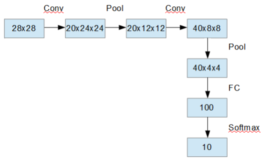
Can we do better with a deeper ConvNet? Let's insert a convolutional layer at the beginning of the network. We'll use a 5x5 local receptive field, a stride of 1, and 20 feature maps. Let's insert a max-pooling layer with a 2x2 pooling window after the convolutional layer. The best validation accuracy I achieved with this network was 98.85%.



```
>>> mini_batch_size = 10
>>> train_d, valid_d, test_d = ann_theano.load_data_shared()
>>> net = ann_theano([
        ConvPoolLayer(image_shape=(mini_batch_size, 1, 28, 28),
                      filter_shape=(20, 1, 5, 5),
                      poolsize=(2, 2)),
        FullyConnectedLayer(n_in=20*12*12, n_out=100),
        SoftmaxLayer(n_in=100, n_out=10)], mini_batch_size)
>>> net.mini_batch_sgd(train_d, 100, mini_batch_size, 0.1, valid_d, test_d)
```

Training 2nd ConvNet on MNIST: Going Deeper

Let's add another convolution/pool layer between the 1st convolution/pool layer and the FC hidden layer and train it for 100 epochs. The best validation accuracy I achieved with this network was 98.99%.



```
>>> mini_batch_size = 10
>>> train_d, valid_d, test_d = ann_theano.load_data_shared()
>>> net = ann_theano([
        ConvPoolLayer(image_shape=(mini_batch_size, 1, 28, 28),
                      filter_shape=(20, 1, 5, 5),
                      poolsize=(2, 2)),
        ConvPoolLayer(image_shape=(mini_batch_size, 20, 12, 12),
                      filter_shape=(40, 20, 5, 5),
                      poolsize=(2, 2)),
        FullyConnectedLayer(n_in=40*4*4, n_out=100),
        SoftmaxLayer(n_in=100, n_out=10)], mini_batch_size)
>>> net.mini_batch_sgd(train_d, 100, mini_batch_size, 0.1, valid_d, test_d)
```

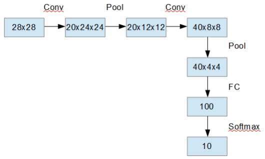
Connecting Convolutional Layers

What does it mean to connect a 20x12x12 max-pooling layer to a 40x8x8 convolutional layer?

Each neuron in the 40x8x8 convolutional layer is connected to all the neurons in the 20x12x12 max-pooling layer but only within their local receptive fields.

Training 3rd ConvNet on MNIST

Let's change the neuron type to ReLU instead of sigmoid, set the learning rate η to 0.01 and the L2 regularization parameter λ to 0.1. The best validation accuracy I achieved with this network was 99.14%.



```
>>> from ann_theano import *
>>> mini_batch_size = 10
>>> train_d, valid_d, test_d = ann_theano.load_data_shared()
>>> net = ann_theano([
        ConvPoolLayer(image_shape=(mini_batch_size, 1, 28, 28),
                      filter_shape=(20, 1, 5, 5),
                      poolsize=(2, 2),
                      activation_fn=ReLU),
        ConvPoolLayer(image_shape=(mini_batch_size, 20, 12, 12),
                      filter_shape=(40, 20, 5, 5),
                      poolsize=(2, 2),
                      activation_fn=ReLU),
        FullyConnectedLayer(n_in=40*4*4, n_out=100, activation_fn=ReLU),
        SoftmaxLayer(n_in=100, n_out=10)], mini_batch_size)
>>> net.mini_batch_sgd(train_d, 100, mini_batch_size, 0.01, valid_d, test_d, lmbda=0.1)
```

ReLU vs. Sigmoid on MNIST

In most (not all) of my experiments on MNIST, ConvNets with ReLUs do slightly better than ConvNets with sigmoids.

The mathematical understanding of why ReLUs do better than sigmoids does not exist. The reason why researchers and developers use ReLUs is empirical: several research groups tried ReLUs on MNIST, achieved better results than with sigmoids, published their findings, and the practice spread.

Training 4th ConvNet on MNIST

Let's insert another fully connected layer after the 1st fully connected layer of ConvNet 3 and train it. The best validation accuracy I achieved with this network was ≈99.27%.

```
>>> from ann_theano import *
>>> mini_batch_size = 10
>>> train_d, valid_d, test_d = ann_theano.load_data_shared()
>>> net = ann_theano([
        ConvPoolLayer(image_shape=(mini_batch_size, 1, 28, 28),
                      filter_shape=(20, 1, 5, 5),
                      poolsize=(2, 2),
                      activation_fn=ReLU),
        ConvPoolLayer(image_shape=(mini_batch_size, 20, 12, 12),
                      filter_shape=(40, 20, 5, 5),
                      poolsize=(2, 2),
                      activation_fn=ReLU),
        FullyConnectedLayer(n_in=40*4*4, n_out=100, activation_fn=ReLU),
        FullyConnectedLayer(n_in=100, n_out=100, activation_fn=ReLU),
        SoftmaxLayer(n_in=100, n_out=10)], mini_batch_size)
>>> net.mini_batch_sgd(train_d, 100, mini_batch_size, 0.01, valid_d, test_d, lmbda=0.1)
```

Training 5th ConvNet on MNIST

Let's add the dropout rate to both fully connected layers on ConvNet 4. The best validation accuracy I achieved with this network was 99.51%.

```
>>> from ann_theano import *
>>> mini_batch_size = 10
>>> train_d, valid_d, test_d = ann_theano.load_data_shared()
>>> net = ann_theano([
        ConvPoolLayer(image_shape=(mini_batch_size, 1, 28, 28),
                      filter_shape=(20, 1, 5, 5),
                      poolsize=(2, 2),
                      activation_fn=ReLU),
        ConvPoolLayer(image_shape=(mini_batch_size, 20, 12, 12),
                      filter_shape=(40, 20, 5, 5),
                      poolsize=(2, 2),
                      activation_fn=ReLU),
        FullyConnectedLayer(
            n_in=40*4*4, n_out=1000, activation_fn=ReLU, p_dropout=0.5),
        FullyConnectedLayer(
            n_in=1000, n_out=1000, activation_fn=ReLU, p_dropout=0.5),
        SoftmaxLayer(n_in=1000, n_out=10, p_dropout=0.5)],
        mini_batch_size)
>>> net.mini_batch_sgd(train_d, 200, mini_batch_size, 0.03, valid_data, test_d)
```

Which Layers Should Dropout Apply To?

We have applied dropout only to fully connected layers. Can we apply dropout to convolutional layers? Yes, we can.

Another question - should we? Probably not, because convolutional layers resist overfitting well enough: the shared weights force convolutional layers to learn from across the entire image.

TFLearn

TFLearn (`http://tflearn.org/`) is a deep learning Python library built on top of tensorflow (`https://www.tensorflow.org/`)

TFLearn is documented but not well documented; lots of technical blogs, some of dubious quality

Training ConvNet with TFLearn

TFLearn includes MNIST as one of its datasets. All we need to do is to split it into training, testing, and validation data.

```
import tflearn.datasets.mnist as mnist

# X is the training data set; Y is the labels for X.
# testX is the testing data set; testY is the labels for testX.
X, Y, testX, testY = mnist.load_data(one_hot=True)
X, Y = shuffle(X, Y)
testX, testY = shuffle(testX, testY)
trainX = X[0:50000]
trainY = Y[0:50000]
validX = X[50000:]
validY = Y[50000:]
trainX = trainX.reshape([-1, 28, 28, 1])
testX  = testX.reshape([-1, 28, 28, 1])
```

Training ConvNet with TFLearn

We need to persist the validation data for validation tests later.

```
import pickle

def save(obj, file_name):
    with open(file_name, 'wb') as fp:
        pickle.dump(obj, fp)

save(validX, '/home/vladimir/my_mnist_net/valid_x.pck')
save(validY, '/home/vladimir/my_mnist_net/valid_y.pck')
```

Training ConvNet with TFLearn

Next we can construct and train a ConvNet on the training and testing data and persist it after training. Let's construct, train, and persist the 1st ConvNet with TFLearn.

```
input_layer = input_data(shape=[None, 28, 28, 1])
conv_layer  = conv_2d(input_layer, nb_filter=20,
                      filter_size=5,
                      activation='sigmoid',
                      name='conv_layer_1')
pool_layer  = max_pool_2d(conv_layer, 2, name='pool_layer_1')
fc_layer_1  = fully_connected(pool_layer, 100,
                              activation='sigmoid',
                              name='fc_layer_1')
fc_layer_2 = fully_connected(fc_layer_1, 10,
                             activation='softmax',
                             name='fc_layer_2')
network = regression(fc_layer_2, optimizer='sgd',
                     loss='categorical_crossentropy',
                     learning_rate=0.1)

model = tflearn.DNN(network)
model.fit(trainX, trainY, n_epoch=1, shuffle=True,
          validation_set=(testX, testY),
          show_metric=True, batch_size=10,
          run_id='MNIST_ConvNet_1')
model.save('/home/vladimir/my_mnist_net/my_net.tfl')
```

Loading Trained ConvNet on MNIST with TFLearn

To load a persisted trained ConvNet, we have to construct the model of the ConvNet that was trained and persisted. Then we load it into the model.

```
input_layer = input_data(shape=[None, 28, 28, 1])
conv_layer  = conv_2d(input_layer, nb_filter=20,
                      filter_size=5,
                      activation='sigmoid',
                      name='conv_layer_1')
pool_layer  = max_pool_2d(conv_layer, 2, name='pool_layer_1')
fc_layer_1  = fully_connected(pool_layer, 100,
                              activation='sigmoid',
                              name='fc_layer_1')
fc_layer_2 = fully_connected(fc_layer_1, 10,
                             activation='softmax',
                             name='fc_layer_2')

model = tflearn.DNN(fc_layer_2)
model.load('/home/vladimir/my_mnist_net/my_net.tfl')
```

Testing Loaded ConvNet on MNIST with TFLearn

Here's how we can test a loaded ConvNet. Note the reshaping of a randomly selected data sample before it is given to the predict member function.

```
import pickle

def load(file_name):
    with open(file_name, 'rb') as fp:
        obj = pickle.load(fp)
    return obj

validX = load('/home/vladimir/my_mnist_net/valid_x.pck')
validY = load('/home/vladimir/my_mnist_net/valid_y.pck')

i = np.random.randint(0, len(validX)-1)
prediction = model.predict(validX[i].reshape([-1, 28, 28, 1]))
print(np.argmax(prediction, axis=1)[0] == np.argmax(validY[i]))
```

Outline

Project 1: Datasets

We've created a USU Box repo where you can download the datasets. The table gives the number of examples for each dataset. BUZZ1, BUZZ2, BUZZ3 are audio datasets; the others are images. The _gray datasets are for ANNs; BEE1, BEE2_1S, BEE4 are for ConvNets.

Table: CS5600/6600: F20: Project 1 Datasets

Dataset	Training	Testing	Validation
BUZZ1	7000	2110	1150
BUZZ2	7582	2332	3000
BUZZ3	9000	2746	2746
BEE1	38139	12724	3528
BEE1_gray	38139	12724	3528
BEE2_1S	35374	11863	10964
BEE2_1S_gray	35374	11863	10964
BEE4	28965	9521	16192
BEE4_gray	28965	9521	16192

Project 1: Datasets

Each dataset consists of 6 pickle files:

1. `train_X.pck`;
2. `train_Y.pck`;
3. `test_X.pck`;
4. `test_Y.pck`;
5. `valid_X.pck`;
6. `valid_Y.pck`.

The `_X` files include examples, the `_Y` files contain the corresponding targets. The `train_` and `test_` files can be used in training and testing. The `valid_` files can be used in validation.

Project 1: Loading Datasets and Creating Nets

The zip archives contain the files:

1. `project1_audio_anns.py`;
2. `project1_audio_cnns.py`;
3. `project1_image_anns.py`;
4. `project1_image_cnns.py`.

These files show you how to load the datasets and create ANNs and ConvNets with tflearn (you'll need to install this library for this project).

Project 1: Dataset Structure: BEE1_gray

Let's check the shape of the BEE1_gray training examples and targets.

```
>>> BEE1_gray_train_X.shape
(38139, 64, 64, 1)
>>> BEE1_gray_train_Y.shape
(38139, 2)
```

BEE1_gray consists of 38,139 64x64x1 numpy arrays. The last 1 means that the image has 1 channel (i.e., it is grayscale). The corresponding targets consists of 38,139 2-element numpy arrays. Let's print the first 10 targets.

Project 1: Dataset Structure: BEE1_gray

The BEE1_gray corresponding targets include of 38,139 2-element numpy arrays. Let's print the first 10 targets.

```
>>> BEE1_gray_train_Y[:10]
array([[0., 1.],
       [0., 1.],
       [0., 1.],
       [0., 1.],
       [0., 1.],
       [0., 1.],
       [1., 0.],
       [1., 0.],
       [0., 1.],
       [1., 0.]])
```

If a target is [1., 0.], the corresponding example is BEE; if a target is [0., 1.], the corresponding example is NO_BEE.

Project 1: Creating an ANN

The function `make_image_ann_model()` gives you an example of defining an ANN with `tflearn`.

```
def make_image_ann_model():
    input_layer = input_data(shape=[None, 64, 64, 1])
    fc_layer_1 = fully_connected(input_layer, 128,
                                 activation='relu',
                                 name='fc_layer_1')
    fc_layer_2 = fully_connected(fc_layer_1, 2,
                                 activation='softmax',
                                 name='fc_layer_2')
    network = regression(fc_layer_2, optimizer='sgd',
                         loss='categorical_crossentropy',
                         learning_rate=0.1)
    model = tflearn.DNN(network)
    return model

>>> ann_model = make_image_ann_model()
```

Project 1: Training an ANN

```
>>> train_tfl_image_ann_model(ann_model,
                              BEE1_gray_train_X,
                              BEE1_gray_train_Y,
                              BEE1_gray_test_X,
                              BEE1_gray_test_Y)
Training samples: 38139 Validation samples: 12724
Training Step: 3814  | total loss: 0.66422 | time: 5.367s
| SGD | epoch: 001 | loss: 0.66422 - acc: 0.5912 |
                      val_loss: 0.66742 - val_acc: 0.5386
                      -- iter: 38139/38139
Training Step: 7628  | total loss: 0.65895 | time: 5.206s
| SGD | epoch: 002 | loss: 0.65895 - acc: 0.5784 |
                     val_loss: 0.66352 - val_acc: 0.5549
                     -- iter: 38139/38139
```

Project 1: Saving, Loading, Testing Nets

We can persist it in a `tfl` file.

```
>>> ann_model.save('ann_model_bee1_gray.tfl')
```

If we want to use a persisted model or train it some more, we need to load it first and then use/train/test/validate it.

```
>>> am = load_image_ann_model('ann_model_bee1_gray.tfl')
>>> validate_tfl_image_ann_model(am,
                                 BEE2_1S_gray_valid_X,
                                 BEE2_1S_gray_valid_Y)
0.7147026632615834
```

References

1. TFLearn.org
2. T.M. Mitchell. *Machine Learning*.
3. M. Neilsen. *Neural Networks and Deep Learning*.