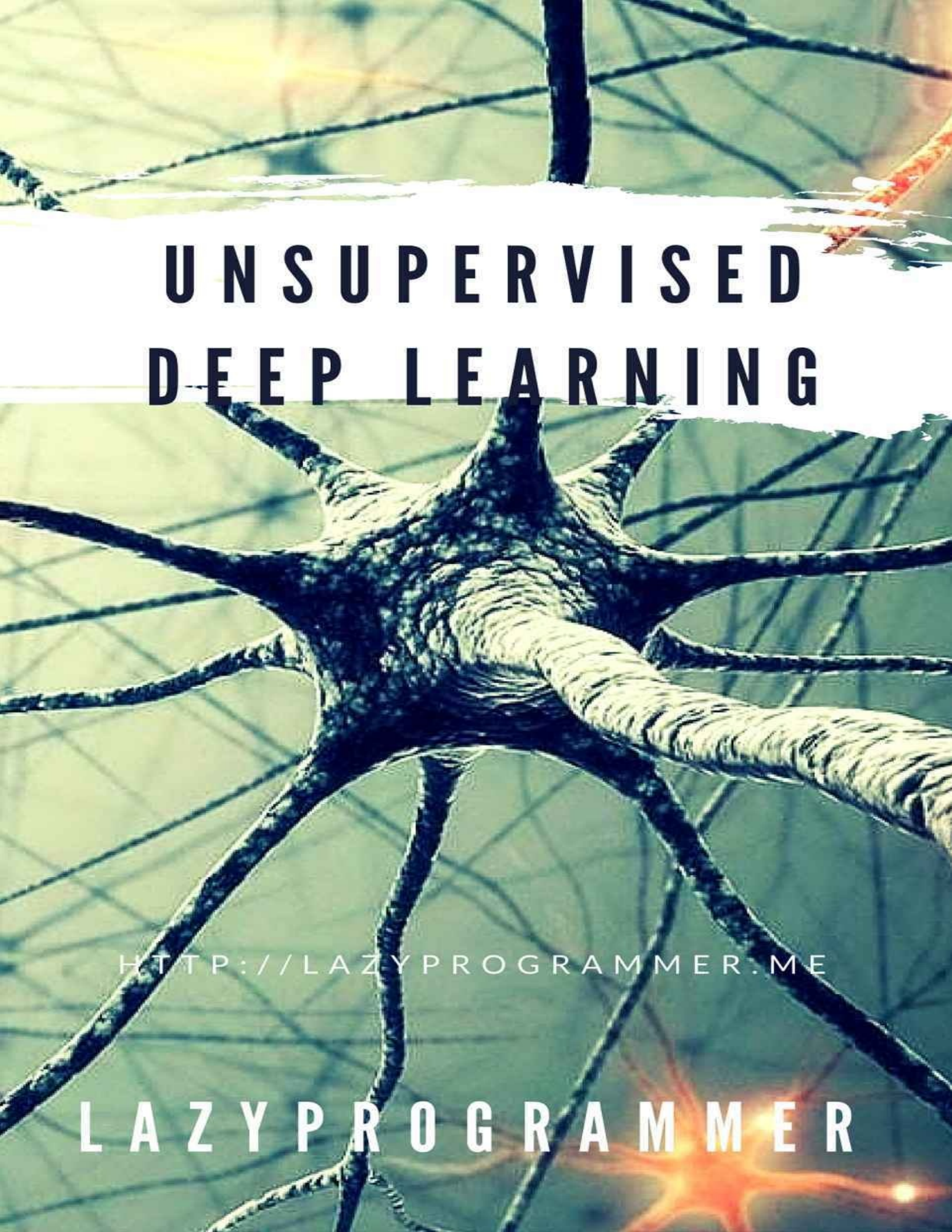

UNSUPERVISED
DEEP LEARNING
HTTP://LAZYPROGRAMMER.ME
LAZYPROGRAMMER

Unsupervised Deep Learning in Python

Master Data Science and Machine Learning with Modern Neural Networks written in Python and Theano

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

Introduction

Chapter 1: Principal Components Analysis

Chapter 2: t-SNE

Chapter 3: Autoencoders and Stacked Denoising Autoencoders

Chapter 4: Restricted Boltzmann Machines and Deep Belief Networks

Chapter 5: Feature Visualization

Chapter 6: Tricking a Neural Network

Conclusion

Introduction

When we talk about modern deep learning, we are often not talking about vanilla neural networks - but newer developments, like using Autoencoders and Restricted Boltzmann Machines to do unsupervised pretraining.

Deep neural networks suffer from the vanishing gradient problem, and for many years researchers couldn't get around it - that is, until new unsupervised deep learning methods were invented.

That is what this book aims to teach you.

Aside from that, we are also going to look at Principal Components Analysis (PCA) and t-Distributed Stochastic Neighbor Embedding (t-SNE), which are not only related to deep learning mathematically, but often are part of a deep learning or machine learning pipeline.

Mostly I am just ultra frustrated with the way PCA is usually taught! So I'm using this platform to teach you Principal Components Analysis in a clear, logical, and intuitive way without you having to imagine rotating globes and spinning vectors and all that nonsense.

One major component of unsupervised learning is **visualization**. We are going to do a lot of that in this book. PCA and t-SNE both help you visualize data from high dimensional spaces on a flat plane.

Autoencoders and Restricted Boltzmann Machines help you visualize what each hidden node in a neural network has learned. One interesting feature researchers have discovered is that neural networks learn hierarchically. Take images of faces for example. The first layer of a neural network will learn some basic strokes. The next layer will combine the strokes into combinations of strokes. The next layer might form the pieces of a face, like the eyes, nose, ears, and mouth. It truly is amazing!

Perhaps this might provide insight into how our own brains take simple electrical signals and combine them to perform complex reactions.

We will also see in this book how you can "trick" a neural network after training it! You may think it has learned to recognize all the images in your dataset, but add some intelligently designed noise, and the neural network will think it's seeing something else, even when the picture looks exactly the same to you!

So if the machines ever end up taking over the world, you'll at least have some tools to combat them.

Finally, in this book I will show you exactly how to train a deep neural network so that you avoid the vanishing gradient problem - a method called “greedy layer-wise pretraining”.

“Hold up... what’s deep learning and all this other crazy stuff you’re talking about?”

If you are completely new to deep learning, you might want to check out my earlier books and courses on the subject:

Deep Learning in Python

Deep Learning in Python Prerequisities

Much like how IBM’s Deep Blue beat world champion chess player Garry Kasparov in 1996, Google’s AlphaGo recently made headlines when it beat world champion Lee Sedol in March 2016.

What was amazing about this win was that experts in the field didn’t think it would happen for another 10 years. The search space of Go is much larger than that of chess, meaning that existing techniques for playing games with artificial

intelligence were infeasible. Deep learning was the technique that enabled AlphaGo to correctly predict the outcome of its moves and defeat the world champion.

Deep learning progress has accelerated in recent years due to more processing power (see: Tensor Processing Unit or TPU), larger datasets, and new algorithms like the ones discussed in this book.

Formatting

I know that the e-book format can be quite limited on many platforms. If you find the formatting in this book lacking, particularly for the code or diagrams, please shoot me an email at info@lazyprogrammer.me along with a proof-of-purchase, and I will send you the original ePub from which this book was created.

Chapter 1: Principal Components Analysis

In this chapter we are going to talk about PCA or principal components analysis. There are 2 components to their description:

1) describing what PCA does and how it is used

2) the math behind PCA.

So what does PCA do? Firstly, it is a linear transformation.

If you think about what happens when you multiply a vector by a scalar, you'll see that it never changes direction. It only becomes a vector of different length. Of course, you could multiply it by -1 and it would face the opposite direction, but it can't be rotated arbitrarily.

Ex. $2(1, 2) = (2, 4)$

If you multiply a vector by a matrix - it CAN change direction and be rotated

arbitrarily.

Ex. $[[1, 1], [1, 0]] [1, 2]^T = [3, 1]^T$

So what does PCA do? Simply put, a linear transformation on your data matrix:

$Z = XQ$

It takes an input data matrix X, which is NxD, multiplies it by a transformation matrix Q, which is DxD, and outputs the transformed data Z which is also NxD.

If you want to transform an individual vector x to a corresponding individual vector z, that would be $z = Qx$. It's in a different order because when x is in a data matrix it's a 1xD row vector, but when we talk about individual vectors they are Dx1 column vectors. It's just convention.

What makes PCA an interesting algorithm is how it chooses the Q matrix.

Notice that because this is unsupervised learning, we have an X but no Y, or no targets.

On Rotation

Another view of what happens when you multiply by a matrix is not that you are are rotating the vectors, but you instead are rotating the coordinate system in which the vectors live.

Which view we take will depend on which problem we are trying to solve.

Dimensionality Reduction

One use of PCA is dimensionality reduction. When you're looking at the MNIST dataset, which is 28x28 images, or vectors of size 784, that's a lot of dimensions, and it's definitely not something you can visualize.

28x28 is a very tiny image, and most images these days are much larger than that - so we either need to have the resources to handle data that can have millions of dimensions, or we could reduce the data dimensionality using techniques like PCA.

We of course can't just take arbitrary dimensions from X - we want to reduce the data size but at the same time capture as much information as possible.

So if we want to go from 784 to 2 dimensions - so that we can visualize it - we want those 2 dimensions to have as much information from X as possible.

Ex. info(1st col of Z) > info(2nd col of Z) > ...

How do we measure this information? In traditional PCA and many other traditional statistical methods we use variance. If something varies more, it carries more information.

You can imagine the opposite situation, where a variable is completely deterministic, *i.e.* it has no variance. Then measuring this variable would not give us any new information, because we already knew what it was going to be.

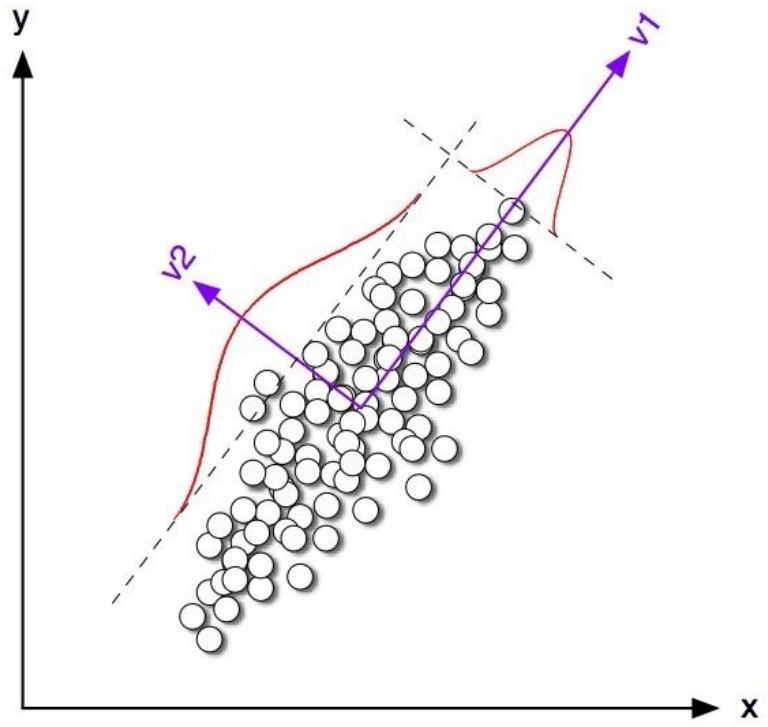
So when we get our transformed data Z, what we want is for the first column to have the most information, the second column to have the second most information, and so on.

Thus, when we take the 2 columns with the most information, that would mean taking the first 2 columns.

Decorrelation

Another thing PCA does is decorrelation.

If we find correlations, that means some of the data is redundant, because we can predict one column from another column.



So now if you take the coordinate system rotation view, you can imagine that if we rotated our coordinate system to be aligned with the spread of these points, then the data would become uncorrelated.

Again the question is - how do we find Q so that we rotate the data in exactly this way?

Visualization

Once we have the information of our data sorted in descending order in each dimension, we can just take the top 2 and make a scatter plot. This will then give us an idea of the separation of the data points, and it allows us to create a visual representation of high-dimensional data.

Pre-processing and Overfitting

The last application of PCA we'll talk about is pre-processing and overfitting.

You can imagine that our data is often noisy. Hopefully, the noise is small compared to the true pattern. In that case, the variance of the noise, which should be small, would go into the last columns of our transformed data Z, at which point we could just discard it.

We could then feed this new data into a supervised machine learning model such as logistic regression - in fact we did this in my previous courses.

So by getting rid of the noise we are preventing overfitting by making sure we are not fitting to the noise.

To make this clear, here's an imaginary data pipeline:

Input: X (data), Y (targets)

1) Convert X to Z --> Z = XQ
2) Take the first K columns of Z, call that Z_K
3) Train your model on (Z_K, Y), *i.e.* model.fit(Z_K, Y)
4) Any further predictions can use the same model and pipeline:
    1) z_K = Qx
    2) prediction = model.predict(z_K)

You'll see that this idea of unsupervised pre-training will come into play again when we study autoencoders and RBMs.

Latent Variables

Another view of PCA is that the transformed variables Z are the “latent variables”, as in they are some sort of underlying cause of the data X.

Then it makes sense that they should be uncorrelated, because they are just independent hidden causes.

It also makes sense that some of the data in X is correlated because they are just measurements you’re taking of some data that is produced by a combination of those hidden causes.

What we are assuming when we do PCA is that the data is a linear combination of those hidden causes.

In fact with PCA the linearity goes both ways - the latent variable Z is a linear combination of the observed variable X, but if you were to do a reverse transformation, the observed data X is also a linear combination of the latent variable Z.

$Z = XQ$

$X = ZQ^{-1}$

You'll recall that we first encountered the idea of latent variables in my first unsupervised learning course on clustering and Gaussian mixture models - because those models assumed that the identities of the clusters were the latent variable.

The Math Behind PCA

Now let us turn to the second part of the PCA description - how to actually find the transformation matrix Q.

To recap, I've just told you about all the magical things this Q matrix can do:

1) Make Z uncorrelated even though X is correlated

2) Order each column if Z by its information content (variance)

A lot of the steps here may seem arbitrary at first, but you'll see how it all fits together in the end and results in all the properties that I talked about in the previously.

The first step is to calculate the covariance of X.

$C(x(i), x(j)) = E[(x(i) - m(i))(x(j) - m(j))] = sum(n=1..N){ (x_n(i) - m(i))(x_n(j) - m(j)) } / N$

Where m(i) is the mean of all x(i).

If i = j then it's just the regular variance. In other words, the diagonals of the C matrix are the variances of each dimension of X. This will be important later.

In matrix form:

$C = (X - m)^T(X - m) / N$

This gives us a DxD matrix. Remember, D is the dimensionality and N is the number of data points.

Technically you can't subtract the mean like that because it has a different shape than X, but we will assume that broadcasting is being used.

Eigenvalues and Eigenvectors

Recall that a DxD matrix has D eigenvalues and D eigenvectors. If you don't know what eigenvalues and eigenvectors are, I'm going to give you a short introduction.

Remember that matrices in general change the direction of, or rotate, a vector. Eigenvectors of a matrix are special vectors which are NOT rotated by the matrix, but just change in length. The change in length is called the eigenvalue.

So we can relate the covariance matrix, its eigenvector, and its eigenvalue, using this equation, where e is the eigenvalue and v is the eigenvector.

$Cv = ev$

There are some theorems which we won't prove, but basically there will be D eigenvectors and D corresponding eigenvalues, and the eigenvalues will be greater than or equal to 0.

Finding eigenvectors and eigenvalues is itself not a trivial task, and there are many algorithms that can do this, including gradient descent. Since numpy already has a function to do this, we're not going to worry about it - just the theory is important to give you the right perspective.

Now we have these D eigenvalues, what do we do with them?

Again an arbitrary step, but let's sort the eigenvalues in descending order. This means that the corresponding eigenvectors have to be sorted in the same way.

$e_1 > e_2 > e_3 > ... > e_D$

Once we've done this, we can put them into matrices of size DxD. The eigenvalues will go in a diagonal matrix of size DxD we'll call E, and the eigenvectors will be lined up beside each other in a matrix we'll call V.

Ex. In 2 dimensions:

E = [e1 0]

[0 e2]

V = [v11 v12]

[v21 v22]

Where v(i,j) is the ith component of the jth eigenvector.

It's easy to prove that:

CV = VE

Try it yourself on paper with the scalar components.

One last ingredient is that the matrix V is orthonormal. This means that any eigenvector dotted with itself is 1, and any eigenvector dotted with another different eigenvector is 0.

Ex.

$v_i^T v_j = 0$ if i != j

$v_i^T v_i = 1$

In matrix form:

$V^T V = I$ (the identity matrix)

Finally, we'll now look at the transformed data, Z. Remember that we still don't know what Q is. But let's solve for its covariance. Notice how we can express

the covariance of Z in terms of the covariance of X.

$C_Z = (Z - m_Z)^T(Z - m_Z) / N$

$C_Z = (XQ - mQ)^T(XQ - mQ) / N$

$C_Z = Q^T(X - m)^T(X - m)Q / N$

$C_Z = Q^T [(X - m)^T(X - m) / N] Q$

$C_Z = Q^T C Q$

We can express the covariance of Z in terms of the covariance of X and Q!

Next, look what happens if we choose Q = V.

$C_Z = V^TCV$

But remember that CV = VE.

$C_Z = V^T V E$

And remember that V is orthonormal so $V^T V = I$ (the identity matrix). Thus:

$C_Z = E$

We get that the covariance of Z is just equal to E, which is the diagonal matrix of eigenvalues.

So what does this all mean?

Since all the off-diagonal elements of E are 0, that means any dimension i is not correlated with any other dimension j, which means there are no correlations in the transformed data.

So by choosing Q = V, we've decorrelated Z.

Next, because we sorted E by the eigenvalues in descending order, that means the first dimension of Z has the most variance, the second dimension of Z has the second most variance, and so on.

The PCA Objective

One thing that is not immediately obvious with the PCA derivation is that it actually minimizes an objective function.

The objective function is what you would naturally expect - it's the squared reconstruction error of the data.

$$J = || X - XQQ^{-1} ||^2$$

Since Q is orthonormal, we know that $Q^T = Q^{-1}$.

In other words, the transpose of Q is equal to the inverse of Q.

We can thus reconstruct X using the transpose instead of the inverse.

We'll encounter this again when we look at autoencoders.

Q_k is just the first k eigenvectors of the covariance of X - meaning that it's a Dxk matrix.

By multiplying X by Q_k, we get Z_k, the first k columns of Z. Remember, this gives us Z "without the noise".

Since we are now not using the full Q, there will be a non-zero reconstruction error.

And we can get back the reconstruction by multiplying by Q_k^T.

$X_{hat} = Z_k Q_k^T$

$$J = || X - XQ_kQ_k^T ||^2$$

You'll want to keep this idea of the PCA objective function in your memory because we are going to encounter it again later.

Exercises

Try PCA on the MNIST dataset. (Use the sci-kit learn library or try writing PCA yourself)

Chapter 2: t-SNE

In this lecture we are going to talk about another dimensionality reduction and visualization method called t-SNE. The t means we are going to incorporate the t-distribution and SNE stands for “stochastic neighbor embedding”.

One big advantage of t-SNE is that it’s a nonlinear method, so it is more expressive than PCA.

Why study t-SNE?

2 reasons:

1) t-SNE was jointly developed by Geoffrey Hinton, who as you know, is one of the main figures of deep learning.

2) by doing t-SNE, we are seeing how the limitations of PCA can be overcome by using a more complex mathematical model.

One key difference between PCA and t-SNE, other than the fact that t-SNE is nonlinear, is that there is no transformation model with t-SNE.

Instead, t-SNE just modifies the outputs directly in order to minimize the cost function.

What this means is that, you won't have any train and test sets, and you can't transform data after fitting on some other data.

The way that t-SNE works is essentially it tries to preserve the distances between each input vector.

We will start with symmetric SNE since it intuitively makes more sense.

On the original data X, we define a joint probability distribution p(i,j) which is equal to:

$$p(i,j) = \exp(-||xi - xj||^2 / (2s^2)) / \text{sum}[k!=m]\{ \exp(-||xm - xk||^2 / (2s^2)) \}$$

Note that i and j here are not the index for the dimensions like we did with PCA, but rather the ith and jth data point in the dataset, *i.e.* i and j are the ith and jth sample.

Notice it kind of looks like a Gaussian distribution. You can think of “s” as a hyperparameter. It’s controlling the “spread” of the distribution.

Next, we have our low-dimensional mapping Y, which we define in the same way, but there is no s term.

$$q(i,j) = \exp(-||yi - yj||^2) / \text{sum}[k!=m]\{ \exp(-||ym - yk||^2) \}$$

Note that we just set $p(i,i) = q(i,i) = 0$.

So we’ve defined 2 probability distributions, one between every pair of points in X, and one between every pair of points in Y.

We usually just initialize every data point in Y randomly.

i.e. If we're looking for a 2-D representation, we'll create Y as:

Y = np.random.randn(N, 2)

Once we've done that, we can try to find a better Y by optimizing some objective function that relates p(i,j) and q(i,j).

If you are not familiar with how we compare 2 probability distributions, we usually use the Kullback-Leibler divergence or KL divergence for short.

$C = D_{KL}(P \parallel Q) = \text{sum}[i,j]\{ p(i,j)\log(p(i,j) / q(i,j)) \}$

If P and Q are exactly the same, this would be 0.

How we solve this problem is the same as how we solve all other problems of this type - we take the derivative of the objective and do gradient descent.

Notice how with this type of model we don’t have weights - we just have Y. So are taking the gradient with respect to Y, which is the output mapping itself.

i.e.

Y <- Y - learning_rate*dC/dY

One problem with symmetric SNE and the SNE that came before it is known as the “crowding problem”, which prevents gaps from forming around the natural clusters.

What t-SNE does is it uses slightly different distributions for Q and P, which helps space out the clusters better.

The new P and Q are defined as follows:

$$p(i,j) = [\ p(i \mid j) + p(j \mid i)\ ] / 2N$$

Where:

$$p(j \mid i) = \exp(-||x_i - x_j||^2 / (2s_i^2)) / \sum_{k \neq i} \exp(-||x_i - x_k||^2 / (2s_i^2))$$

Notice how here each sample has its own “s”.

$$q(i,j) = (1 + ||y_i - y_j||^2)^{-1} / \sum_{k \neq m} (1 + ||y_k - y_m||^2)^{-1}$$

The Q distribution uses the t-distribution, hence the name.

The cost function remains the same as before.

Note that we won’t actually implement t-SNE - even though it should be relatively simple given the definitions above.

It’s easy to define the cost and find its derivative, which is more than we can say for many of the deep learning models we’ve worked with.

The problem is that it’s slow and has huge RAM requirements - in fact t-SNE will probably crash on your computer with the full MNIST dataset.

Why? Because we need to calculate q(i,j) and p(i,j) for i=1..N and j=1..N, this is naturally an $O(N^2)$ algorithm.

There is a variant of it, which is called Barnes-Hut, that is O(NlogN) run time, but still has huge RAM requirements. This is the default method used in Sci-Kit Learn.

One solution is to take a sample of just a few hundred data points and do t-SNE on that, but of course you can increase this amount if you’re willing to wait longer and you have enough RAM.

Exercises

Try t-SNE (from the sci-kit learn library) on the MNIST dataset.

Try writing your own naive implementation given the definitions above, and take advantage of Theano’s automatic differentiation.

Chapter 3: Autoencoders and Stacked Denoising Autoencoders

In this chapter we are going to talk about autoencoders. Autoencoders are actually nothing really new, just a small twist on something you already know.

I always say that a supervised machine learning model has 2 main functions as its API, train or fit, and predict.

Usually when we call a neural network, we call model.fit(X, Y), and then to make predictions we call model.predict(X).

But what if we just make a neural network try to predict itself? So we call model.fit(X, X) instead.

o---o---o

x z x'

That’s exactly what an autoencoder is.

In most of my previous courses, we did classification, but remember that our X can be any real value.

So if you are trying to predict real values in general, you can use the squared error, and it becomes more like a regression.

You can alternatively still use the cross-entropy error, and consider your inputs and outputs to be binary variables, even for variables that are not exactly binary.

You’ll see that we do this with both autoencoders and RBMs, and you’ll see that both error functions can work.

You can think of images like MNIST as having pixel intensities. 0 would be no intensity at all, and 1 would be maximum intensity, since we always scale by 255.

To make the outputs go between 0 and 1, we are going to use the sigmoid function at both the hidden layer and output layer.

One slight modification we sometimes use for both autoencoders and RBMs is the idea of shared weights.

So instead of using another weight at the output layer, we just use the transpose of the first weight.

Ex.

$Z = \text{sigmoid}(XW + b)$

$X_{hat} = \text{sigmoid}(ZW^T + c)$

We first encountered the idea of shared weights when we looked at convolutional neural networks, since it was the same filter getting passed along each part of the image.

Remember that having shared weights is a kind of regularization because we are reducing the number of parameters, thereby reducing the chance of overfitting.

Next, let's return to the squared error objective.

If we write it out in terms of the weights and the inputs (ignoring the biases), we get:

$$J = || X - s(s(XW)W^T) ||^2$$

Remember back when we were doing PCA, I asked you to remember the objective function?

$$J = || X - XQQ^T ||^2$$

Well consider what would happen if we did NOT use the sigmoid here.

$$J = || X - XWW^T ||^2$$

We would in fact, just get back PCA!

You can in fact think of autoencoders as a sort of nonlinear PCA.

This is doing a nonlinear mapping, just like t-SNE.

Note that this isn't really the full picture because we are not considering biases, and we also don't have any requirement that each column of the weight has length 1, or that each column is orthogonal, or that they are ordered in any particular way.

So while the functional form looks "like" PCA, it's not doing everything PCA is doing.

Denoising Autoencoders

In this section we are going to talk about yet another method of regularization.

Remember that when we're given a dataset, we don't have to only train on that dataset. We first saw this when we talked about images. An upside-down cat is

still a cat, so we should add that to our dataset. Similarly, a sideways cat is still a cat. A cat on the top right of a photo is still the same cat as that cat shifted to the bottom left.

Therefore, there are ways we can modify our data to improve our neural network's generalization capabilities.

Another way to modify data other than moving things around explicitly is to add noise.

One way is to add Gaussian noise to the input, another way is to simply set some of the values to 0.

This is almost trivial to do since it's just adding a bitmask to your input, or in other words, you generate a random vector of 0s and 1s, and do an element by element multiplication on X, to get your new X.

Ex.

from theano.tensor.shared_randomstreams import RandomStreams rng = RandomStreams()

bitmask = rng.binomial(n=1, p=p, size=X.shape) X = bitmask * X

Typically, we set the probability of generating a 0 to something less than 50%, so perhaps maybe something like 30%, but researchers have found that up to 50% still provides good results.

Stacked Autoencoders

In this section we are going to explore what happens when we take a bunch of autoencoders and put them together in layers.

The basic algorithm is this.

You train one autoencoder, and then fix its weights. Then you throw out the last layer, keeping only the input to hidden part.

Next, you take the output of the hidden layer of the first autoencoder, and you

make that the input to another autoencoder.

And you just repeat this process for multiple autoencoders.

Ex:

Train:

o--o--o

x z x’

Keep:

o--o

x z

Train another:

o--o--o

z z2 z’

Keep another:

o--o

z z2

Train another:

o--o--o

z2 z3 z2’

Keep another:

o--o

z2 z3

Finally, you end up with a stack of autoencoders, each designed simply to learn a more compact version of the previous layer.

What you end up with is a deep unsupervised network:

o--o--o--o

x z z2 z3

Since each time we use an autoencoder, we are trying to get a more compact representation than what we had in the previous layer, we are going to want each layer to be smaller in size than the previous layer.

Note that this doesn't necessarily have to be the case. We could in fact have a huge number of hidden units in the middle layer. Theoretically, if we had N training samples, and N hidden units, then we could get a perfect reconstruction because each of the N columns of W could be responsible for reproducing a different sample.

You would think that this would lead to severe overfitting, however researchers have found it does not overfit, and even a number of hidden units greater than N can work.

In our code, we will use classes and objects to represent the ANN and Autoencoder.

This will help us compartmentalize our code much better, given the fact that we can have an arbitrary number of autoencoders.

We can then simply make the hidden output of each autoencoder the input into the next autoencoder.

Greedy layer-wise pretraining

The process that we've just talked about is called "greedy layer-wise pretraining".

The reason it is called greedy you should understand if you have ever studied algorithms. Greedy intuitively means you make the best short-sighted decision, which in our case is simply training one layer of the autoencoder.

While it is theoretically not optimal, it actually helps a lot with supervised training, which is the next step, and that is why we call it pretraining.

The last step in the process is to add a logistic regression layer to the end of the stacked autoencoders.

Once you've done that, you've made what is called a deep neural network.

Since the last layer with logistic regression will have randomized weights, we will still need to do some backpropagation.

But what we find is that doing backpropagation at this point doesn't take as long, because the autoencoder weights already put us in the neighborhood of the correct answer.

So we just need to do a few epochs of backpropagation to "fine-tune" the network.

Notice that there are zero new things here in terms of architecture and functional forms, just a new concept using old ideas.

You already know how to do all this stuff - build a neural network, build a neural network with multiple layers, do gradient descent in Theano, *etc.*

In the code, note that we assume as usual that we have a function getData() that returns some NxD matrix for the input data and an Nx1 vector for the output targets.

We will load both training data and test data, so we'll have Xtrain of size N_{train}xD, Ytrain of size N_{train}x1, Xtest of size N_{test}xD, and Ytest of size N_{test}x1.

Note that there is code for both the squared error objective and cross-entropy objective. Typically we use cross-entropy for 0/1 targets, but you'll see that it works even better than squared error. We will explain why later in the book.

Let's see the code:

```
def relu(x):

return x * (x > 0)
```

```
def error_rate(p, t):

return np.mean(p != t)


def init_weights(shape):

return np.random.randn(*shape) / np.sqrt(sum(shape))


import numpy as np

import theano

import theano.tensor as T

import matplotlib.pyplot as plt


from sklearn.utils import shuffle


class AutoEncoder(object):
```

```
def __init__(self, M):

self.M = M



def fit(self, X, learning_rate=0.5, mu=0.99, epochs=1, batch_sz=100,
show_fig=False): N, D = X.shape

n_batches = N / batch_sz



W0 = init_weights((D, self.M))

self.W = theano.shared(W0)

self.bh = theano.shared(np.zeros(self.M)) self.bo = theano.shared(np.zeros(D))

self.params = [self.W, self.bh, self.bo]

self.forward_params = [self.W, self.bh]



self.dW = theano.shared(np.zeros(W0.shape)) self.dbh =
theano.shared(np.zeros(self.M)) self.dbo = theano.shared(np.zeros(D))

self.dparams = [self.dW, self.dbh, self.dbo]

self.forward_dparams = [self.dW, self.dbh]
```

```
X_in = T.matrix('X_%s' % self.id)

X_hat = self.forward_output(X_in)



# attach it to the object so it can be used later # must be sigmoidal because the output is also a sigmoid H = T.nnet.sigmoid(X_in.dot(self.W) + self.bh) self.hidden_op = theano.function(

inputs=[X_in],

outputs=H,

)



# cost = ((X_in - X_hat) * (X_in - X_hat)).sum() / N

cost = -(X_in * T.log(X_hat) + (1 - X_in) * T.log(1 - X_hat)).sum() / N

cost_op = theano.function(

inputs=[X_in],

outputs=cost,

)
```

```
updates = [

(p, p + mu*dp - learning_rate*T.grad(cost, p)) for p, dp in zip(self.params,
self.dparams) ] + [

(dp, mu*dp - learning_rate*T.grad(cost, p)) for p, dp in zip(self.params,
self.dparams) ]

train_op = theano.function(

inputs=[X_in],

updates=updates,

)



costs = []

for i in xrange(epochs):

print "epoch:", i

X = shuffle(X)

for j in xrange(n_batches):

batch = X[j*batch_sz:(j*batch_sz + batch_sz)]
```

```
train_op(batch)

the_cost = cost_op(X) # technically we could also get the cost for Xtest here
print "j  n_batches:", j, "", n_batches, "cost:", the_cost costs.append(the_cost)

if show_fig:

plt.plot(costs)

plt.show()



def forward_hidden(self, X):

Z = T.nnet.sigmoid(X.dot(self.W) + self.bh) return Z



def forward_output(self, X):

Z = self.forward_hidden(X)

Y = T.nnet.sigmoid(Z.dot(self.W.T) + self.bo) return Y



@staticmethod

def createFromArrays(W, bh, bo):
```

```
ae = AutoEncoder(W.shape[1])

ae.W = theano.shared(W)

ae.bh = theano.shared(bh)

ae.bo = theano.shared(bo)

ae.params = [ae.W, ae.bh, ae.bo]

ae.forward_params = [ae.W, ae.bh]

return ae




class DNN(object):

def __init__(self, hidden_layer_sizes, UnsupervisedModel=AutoEncoder):
self.hidden_layers = []

for M in hidden_layer_sizes:

ae = UnsupervisedModel(M)

self.hidden_layers.append(ae)



def fit(self, X, Y, Xtest, Ytest, pretrain=True, learning_rate=0.01, mu=0.99,
```

reg=0.1, epochs=1, batch_sz=100): # greedy layer-wise training of autoencoders
pretrain_epochs = 1

if not pretrain:

pretrain_epochs = 0

current_input = X

for ae in self.hidden_layers:

ae.fit(current_input, epochs=pretrain_epochs)

# create current_input for the next layer current_input =
ae.hidden_op(current_input)

initialize logistic regression layer

N = len(Y)

K = len(set(Y))

W0 = init_weights((self.hidden_layers[-1].M, K)) self.W = theano.shared(W0)

self.b = theano.shared(np.zeros(K))

self.params = [self.W, self.b]

```
for ae in self.hidden_layers:

self.params += ae.forward_params



# for momentum

self.dW = theano.shared(np.zeros(W0.shape)) self.db =
theano.shared(np.zeros(K))

self.dparams = [self.dW, self.db]

for ae in self.hidden_layers:

self.dparams += ae.forward_dparams



X_in = T.matrix('X_in')

targets = T.ivector('Targets')

pY = self.forward(X_in)



# squared_magnitude = [(p*p).sum() for p in self.params]

# reg_cost = T.sum(squared_magnitude)

cost = -T.mean( T.log(pY[T.arange(pY.shape[0]), targets]) ) #+ reg*reg_cost
```

```
cost = T.mean(-T.log(p_y_1[T.arange(p_y_1.shape[0]), targets])) + reg*reg_cost
prediction = self.predict(X_in)

cost_predict_op = theano.function(

inputs=[X_in, targets],

outputs=[cost, prediction],

)



updates = [

(p, p + mu*dp - learning_rate*T.grad(cost, p)) for p, dp in zip(self.params,
self.dparams) ] + [

(dp, mu*dp - learning_rate*T.grad(cost, p)) for p, dp in zip(self.params,
self.dparams) ]



train_op = theano.function(

inputs=[X_in, targets],

updates=updates,

)
```

```
n_batches = N / batch_sz

costs = []

print "supervised training..."

for i in xrange(epochs):

print "epoch:", i

X, Y = shuffle(X, Y)

for j in xrange(n_batches):

Xbatch = X[j*batch_sz:(j*batch_sz + batch_sz)]

Ybatch = Y[j*batch_sz:(j*batch_sz + batch_sz)]

train_op(Xbatch, Ybatch)

the_cost, the_prediction = cost_predict_op(Xtest, Ytest) error =
error_rate(the_prediction, Ytest) print "j  n_batches:", j, "", n_batches, "cost:",
the_cost, "error:", error costs.append(the_cost)

plt.plot(costs)

plt.show()



def predict(self, X):
```

```
return T.argmax(self.forward(X), axis=1)

def forward(self, X):

current_input = X

for ae in self.hidden_layers:

Z = ae.forward_hidden(current_input)

current_input = Z



# logistic layer

Y = T.nnet.softmax(T.dot(current_input, self.W) + self.b) return Y



def main():

Xtrain, Ytrain, Xtest, Ytest = getKaggleMNIST() # TRY BOTH!

# dnn = DNN([1000, 750, 500])

# dnn.fit(Xtrain, Ytrain, Xtest, Ytest, epochs=3) # vs

dnn = DNN([1000, 750, 500])
```

```
dnn.fit(Xtrain, Ytrain, Xtest, Ytest, pretrain=False, epochs=10)


if __name__ == '__main__':

main()
```

Exercises

Add denoising to the autoencoder.

Cross-Entropy vs. KL Divergence

You have already seen that KL Divergence can be used to compare the similarity of 2 probability distributions, as we did with t-SNE.

One identity to remember is that cross-entropy is equivalent to KL divergence up to an additive constant.

i.e.

H(P, Q) = cross-entropy between P and Q

H(P) = entropy of P

D_{KL}(P || Q) = KL divergence between P and Q

H(P,Q) = -sum[i]{ P(i) logQ(i) }

H(P) = -sum[i]{ P(i) logP(i) }

D_{KL}(P || Q) = sum[i]{ P(i)log(P(i) / Q(i) }

We can show that:

H(P,Q) = H(P) + D_{KL}(P || Q)

Why is this significant?

The derivative of a constant is 0. Therefore, the gradient, and thus gradient descent, is the same for both cross-entropy and KL divergence.

So when we use cross-entropy on the autoencoder, what we're really saying is the distribution of the input X is the target distribution, and the distribution of the reconstruction of X is the output distribution, and we would like them to be the same.

Summary

The MOST important concept to get out of this is that for the single autoencoder, the "Z" is a "low-dimensional representation" of the input data X. We saw this same concept with PCA. We will see it again with RBMs.

Chapter 4: Restricted Boltzmann Machines and Deep Belief Networks

In this chapter we are going to talk about Restricted Boltzmann Machines and deep belief networks, which is just stacked Restricted Boltzmann Machines - used in exactly the same way we stacked autoencoders.

It’s interesting to look back at what we’ve learned at this point.

When we did linear regression - we were able to calculate the derivative of the cost, set it to 0, and solve for the weights to get the best weights.

With logistic regression and neural networks, you know that we can’t directly solve for the weights, so we take the derivative of the cost with respect to the weights, and take small steps in that direction.

With RBMs, we enter an even more difficult situation. We can’t even calculate the derivative, so instead we try to estimate it approximately using Gibbs sampling, which is a special case of Markov Chain Monte Carlo.

To understand RBMs, we first have to talk a little bit about Markov random fields.

A Markov random field is just a graph of states, where each node is a state, which is a random variable.

P(A, B, C, D)

A Markov random field represents a joint probability distribution, *i.e.* the probability of all those random variables.

2 things to note about the MRF:

First: that the Markov property holds, i.e.:

$P(s(t) \mid s(t-1), s(t-2), ...) = P(s(t) \mid s(t-1))$

The next state only depends on the current state, and not on any states before that.

Second: the graph is undirected, so you can move along any edge in any direction.

This brings us to the concept of Boltzmann machines.

Boltzmann machines are Markov random fields where everything is connected to everything.

The idea is you define an energy on the network, and your goal is to find the minimum energy state, much like how physical systems do.

In fact, the Boltzmann distribution, which is the distribution we use when we talk about Boltzmann machines and restricted Boltzmann machines, is from statistical mechanics.

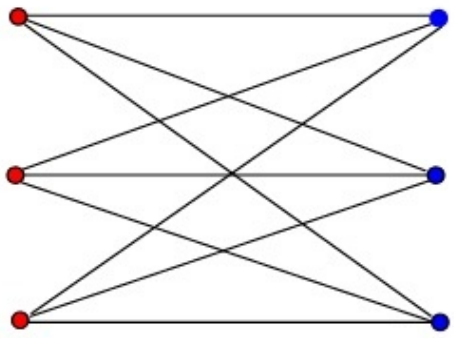
The energy is defined as:

$$E(s) = -(\sum_{i,j}{ w(i,j)s(i)s(j) } + \sum_{i}{ b(i)s(i) }$$

It’s not super important since we won’t be using this exact form, but recognize the “structure” here. There’s a 2-D weight that gets multiplied by “interaction terms” and a 1-D linear weight.

Now we can go from Boltzmann machines to restricted Boltzmann machines.

Here instead of everything being connected to everything, we instead have a “bipartite graph”.



This means that everything on the left is connected to everything on the right, and vice versa.

We usually label the nodes on the left the visible units (v), and the nodes on the right the hidden units (h). This gives us a latent variable model, just like with PCA and autoencoders.

Note that the Markov property still holds. What this means in this graph is that

we have conditional independence. So the probability p(h(i) | v) is independent of the probability p(h(j) | v).

Similarly, if you go in the other direction (remember, this graph is undirected), the probability p(v(i) | h) is independent of the probability p(v(j) | h).

With RBMs we technically assume both the visible units and hidden units are binary units.

This is why we used the sigmoid for autoencoders also, and why we only use the sigmoid uniformly throughout this book.

We know that this works just fine for MNIST, and in fact a lot of data is binary. All categorical variables, since we use one-hot encoding, are binary, as are bag of words models, so basically anything to do with NLP will work here too.

There is some material out there that goes over at a high level how you could have Gaussian units, but these other formulations have been reported to have stability issues, so we won't cover them here.

We are ready to talk about the math behind RBMs.

Note that this is very similar to the regular Boltzmann machine, there is one term that depends on the interaction of 2 different nodes, and then a bias-like term that depends only on one node.

The total energy is $E(v,h) = -b^T v - c^T h - v^T W h$

So if v is Dx1 and h is Mx1, then W must be DxM, which is what we usually have in a neural network.

b is Dx1 and c is Mx1 to give us a valid dot product.

Note that in terms of these variables, the conditional probabilities we talked about earlier are:

$p(h \mid v) = \text{sigmoid}(W^T v + c)$ (forward direction)

$p(v \mid h) = \text{sigmoid}(Wh + b)$ (backward direction)

Like autoencoders, we again use shared weights.

Now that we have an expression for our energy function, we can now define the joint probability of v and h.

$$P(v,h) = \exp(-E(v,h)) / Z$$

Where:

$$Z = \text{sum}\{ \text{sum}\{ \exp(-E(v,h) \}\}$$

We call Z the partition function, which just ensures that all the probabilities add up to 1.

Note that this itself is intractable to calculate, because the summations mean summing over all possible values of v and h. Since they are both binary variables, that is $O(2^N)$ for each of them.

Ex. if M=3, then the possible values for h are:

000

001

010

011

100

101

110

111

And we must sum over all of them.

What we want then, given this model, is to maximize P(v), which is P(v,h) marginalized over h. This is because we don’t actually care about the values of h, we just want to maximize the probability of what we have seen, which is v.

i.e.

$$P(v) = \text{sum}[h]\{ P(v, h) \}$$

As usual, we don't maximize P(v) directly, but rather the log of P(v). So let's begin by trying to take its derivative with respect to some arbitrary parameter w.

$$d(-\log(P(v))/dw$$

$$= d(-\log(\text{sum}[h] \{P(v,h) \})/dw$$

$$= d(-\log(\text{sum}[h] \{ \exp(-E(v,h)) \, Z \, \})dw$$

$$= \text{sum}[h]\{ P(h \mid v)E(v,h) \} - \text{sum}[v,h]\{ P(v,h)E(v,h) \}$$

$$= E_{h \mid v}\{ E(v,h) \} - E_{v,h}\{ E(v,h) \}$$

This is read as "the expected value of E(v,h) over the conditional distribution P(h | v)" - "the expected value of E(v,h) over the joint distributoon P(v, h)".

So what we end up with is the difference between 2 expectations. The first

expectation is easy but the second expectation is hard because it requires an infinite number of Gibbs samples. We usually call the first term the clamped term and the second term the unclamped term.

Let's take the previous expression and expand it further. Let's do the input to hidden weights explicitly.

$$J = \text{sum}[h]\{ P(h \mid v)E(v,h) \} - \text{sum}[v,h]\{ P(v,h)E(v,h) \}$$

$$dJ/dW(i, j) = - \text{sum}[h] \{ P(h \mid v)\, h(j)v(i) \} + \text{sum}[v]\{ P(v)hP(h \mid v)h(j)v(i) \}$$

$$dJ/dW(i, j) = - p(h(j)=1 \mid v)v(i) + \text{sum}[v]\{ P(v)p(h(j)=1 \mid v)v(i) \}$$

$$dJ/dW(i, j) = - \text{sigmoid}(c(j) + v^T W(:,j))v(i) + \text{sum}[v]\{ P(v)\text{sigmoid}(c(j) + v^T W(:,j))v(i) \}$$

We can do similar derivatives for the biases and that gives us these 3 update equations:

$$dJ/dW(i, j) = - \text{sigmoid}(c(j) + v^T W(:,j))v(i) + \text{sum}[v]\{ P(v)\text{sigmoid}(c(j) + v^T W(:,j))v(i) \}$$

$$dJ/b(i) = v(i) - \text{sum}[v]\{ P(v)v(i) \}$$

$dJ/dc(j) = \text{sigmoid}(c(j) + v^T W(:,j)) - \text{sum}[v]\{ P(v)\text{sigmoid}(c(j) + v^T W(:,j)) \}$

So now you see what we mean by the clamped term and unclamped term. We are going to get the first term simply by direct calculation, but we are going to get the second term - which requires a sum over all v - which is infeasible - by sampling.

Note that if we were to code the RBM using numpy, we would use the actual gradient expressions directly as I previously defined them, but since Theano automatically finds gradients we don't have to.

Another method, which is the method we are going to use, makes use of a function called the free energy.

We define the free energy as the negative log of the sum over all h of the top part of the joint probability.

$F(v) = -\log(\text{sum}[h]\{ \exp(-E(v,h)) \})$

It's hard to prove because you can't just use direct algebraic notation, but if you are interested in learning more this is a result of all the units being conditionally

independent, and it is related to something called the sum-product algorithm. In any case, the free energy reduces to:

$$F(v) = -b^T v - \sum_{j=1..M}\{ \log(1+\exp(c(j) + v^T W(:,j))) \}$$

So that it now no longer depends on h at all.

When you use this in the cost function and take the derivative, what you get is a simpler expression with only one expected value.

$$dJ(v)/dw = dF(v)/dw - \sum_{v'}\{ P(v')dF(v')/dw \}$$

Notice how this actually gives us the same update rules as before.

We call the first term the positive phase and the second term the negative phase.

Since the first term goes in the direction of the free energy at the sample v, it's basically trying to reduce the energy for that particular v, which corresponds to a

higher probability for that sample v.

The other term, the negative phase, tries to lower the probability for ALL possible values of v.

One last note about RBMs is that we can stack them just like how we stack autoencoders and do “greedy layer-wise pretraining”.

Except when we stack RBMs they get a special name - they are called deep belief networks.

One small note about the proper terminology here - “deep belief network” refers to the unsupervised model, whereas if you train it in a supervised way it is called a “deep neural network”.

Sometimes “deep belief network” can also refer to the stacked autoencoder configuration from the last chapter.

Contrastive Divergence

In this section we are going to talk about the sampling algorithm, called contrastive divergence, that we use to estimate the gradient we derived in the last section.

If you found the material in the last lecture hard, don't worry too much about it, because following the steps of the CD algorithm is actually much easier.

Contrastive divergence is usually called CD-k, meaning you do k steps of Gibbs sampling.

What that means is, given a training sample v0, we are going to calculate p(h0|v0) which is just the usual forward pass on the network. Next, we are going to use those probabilities to draw a sample of h - call that h0.

So if p(h_j | v) is 0.3, then h_j will be 1 with probability 0.3.

Next, we do a backwards pass, so we calculate p(v1|h0). Now we use these probabilities to draw a sample of visibles v1.

Next, we calculate p(h1|v1), and take a sample of h again. We keep repeating this process to infinity.

o---o---o---o-...

v0 h0 v1 h1...

By going to infinity, we get an exact answer, but we obviously cannot train the neural network for that long.

What researchers have found is that k=1 works just fine. We will use k=1 also, which will allow us to avoid having to do any loops.

In pseudocode, the process might look something like this:

```
def derivative(v0):

p_h0 = self.forward(v0)

h0 = sample(p_h0)
```

```
p_v1 = self.backward(h0)

v1 = sample(p_v1)

p_h1 = self.forward(v1)

return v0.dot(p_h0.T) - v1.dot(p_h1.T)
```

Of course, that would work fine if we were using numpy, but we are going to use Theano, which will find the gradients automatically.

What we will do instead, is after we calculate v1 from v0, we will set the objective to the free_energy(v0) - free_energy(v1), and let Theano calculate the gradient for us.

One interesting thing is, because we don't actually want to calculate P(v), we still want to plot some cost function as training progresses.

We are going to continue to use the cross-entropy error like we did with autoencoders, and if you run the code you'll see that despite using a completely different training algorithm and theoretical framework, this error function still decreases as we train.

Let’s get to the code. Assume relu, error_rate, getData, and init_weights are defined the same as the last example.

```
import numpy as np

import theano

import theano.tensor as T

import matplotlib.pyplot as plt


from sklearn.utils import shuffle

from theano.tensor.shared_randomstreams import RandomStreams

from autoencoder import DNN


class RBM(object):

def __init__(self, M):

self.M = M
```

```
self.rng = RandomStreams()


def fit(self, X, learning_rate=0.1, epochs=1, batch_sz=100, show_fig=False):

N, D = X.shape

n_batches = N / batch_sz


W0 = init_weights((D, self.M))

self.W = theano.shared(W0)

self.c = theano.shared(np.zeros(self.M))

self.b = theano.shared(np.zeros(D))

self.params = [self.W, self.c, self.b]

self.forward_params = [self.W, self.c]


# we won't use this to fit the RBM but we will use these for backpropagation later

self.dW = theano.shared(np.zeros(W0.shape))
```

```
self.dc = theano.shared(np.zeros(self.M))

self.db = theano.shared(np.zeros(D))

self.dparams = [self.dW, self.dc, self.db]

self.forward_dparams = [self.dW, self.dc]



X_in = T.matrix('X')



# attach it to the object so it can be used later

# must be sigmoidal because the output is also a sigmoid

H = T.nnet.sigmoid(X_in.dot(self.W) + self.c)

self.hidden_op = theano.function(

inputs=[X_in],

outputs=H,

)



# we won't use this cost to do any updates
```

```
# but we would like to see how this cost function changes

# as we do contrastive divergence

X_hat = self.forward_output(X_in)

cost = -(X_in  T.log(X_hat) + (1 - X_in)  T.log(1 - X_hat)).sum() / N

cost_op = theano.function(

inputs=[X_in],

outputs=cost,

)



# do one round of Gibbs sampling to obtain X_sample

H = self.sample_h_given_v(X_in)

X_sample = self.sample_v_given_h(H)



# define the objective, updates, and train function

objective = T.mean(self.free_energy(X_in)) -
T.mean(self.free_energy(X_sample))
```

```
# need to consider X_sample constant because you can't take the gradient of random numbers in Theano

updates = [(p, p - learning_rate*T.grad(objective, p, consider_constant=[X_sample])) for p in self.params]

train_op = theano.function(

inputs=[X_in],

updates=updates,

)



costs = []

for i in xrange(epochs):

print "epoch:", i

X = shuffle(X)

for j in xrange(n_batches):

batch = X[j*batch_sz:(j*batch_sz + batch_sz)]

train_op(batch)
```

```
the_cost = cost_op(X)  # technically we could also get the cost for Xtest here

print "j  n_batches:", j, "", n_batches, "cost:", the_cost

costs.append(the_cost)

if show_fig:

plt.plot(costs)

plt.show()



def free_energy(self, V):

return  -V.dot(self.b) - T.sum(T.log(1 + T.exp(V.dot(self.W) + self.c)), axis=1)



def sample_h_given_v(self, V):

p_h_given_v = T.nnet.sigmoid(V.dot(self.W) + self.c)

h_sample = self.rng.binomial(size=p_h_given_v.shape, n=1, p=p_h_given_v)

return h_sample



def sample_v_given_h(self, H):
```

```
p_v_given_h = T.nnet.sigmoid(H.dot(self.W.T) + self.b)

v_sample = self.rng.binomial(size=p_v_given_h.shape, n=1, p=p_v_given_h)

return v_sample


def forward_hidden(self, X):

return T.nnet.sigmoid(X.dot(self.W) + self.c)


def forward_output(self, X):

Z = self.forward_hidden(X)

Y = T.nnet.sigmoid(Z.dot(self.W.T) + self.b)

return Y


@staticmethod

def createFromArrays(W, c, b, an_id):

rbm = AutoEncoder(W.shape[1], an_id)

rbm.W = theano.shared(W, 'W_%s' % rbm.id)
```

```
rbm.c = theano.shared(c, 'c_%s' % rbm.id)

rbm.b = theano.shared(b, 'b_%s' % rbm.id)

rbm.params = [rbm.W, rbm.c, rbm.b]

rbm.forward_params = [rbm.W, rbm.c]

return rbm



def main():

Xtrain, Ytrain, Xtest, Ytest = getKaggleMNIST()

dnn = DNN([1000, 750, 500], UnsupervisedModel=RBM)

dnn.fit(Xtrain, Ytrain, Xtest, Ytest, epochs=3)



# we compare with no pretraining in autoencoder.py



if __name__ == '__main__':
```

if __name__ == "__main__":

main()

Chapter 5: Feature Visualization

One good exercise is to look at what each hidden node at each hidden layer has actually learned once training is complete.

This is one way to interpret what a neural network has learned.

How can we do this?

Assume we want h(i,j) - the jth hidden node in the ith hidden layer - to be maximal, and all other hidden nodes in that layer to be minimal.

This is just a calculus problem!

Maximize h(i,j) with respect to X.

Unfortunately, it’s not as easy as simply finding dh(i,j)/dX, setting it to 0, and solving for X.

You can solve for the optimal X in closed-form for the first input-to-hidden layer.

You will need to use the method of Lagrange multipliers.

What you should find is that to optimize the jth hidden node, X should be proportional to W(:,j) - the jth column of the input-to-hidden weight matrix W.

Recall that W is a DxM matrix, so taking 1 column will give us a D-dimensional vector.

To plot the image, we simply reshape the jth column of W to the original image size (28x28 for MNIST) and use plt.imshow() from matplotlib.

Use:

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

To ensure the image is grayscale rather than a heatmap.

For the layers beyond the input-to-hidden, we can't solve in closed-form, so we return to our old friend gradient descent.

This should be simple since Theano can calculate gradients for us. The only caveat is that you now need to take the derivative with respect to the input, not the weights, like we usually do during training.

Chapter 6: Tricking a Neural Network

Now that you know how to take the derivative of a variable with respect to the inputs instead of the weights, we are ready to talk about how to trick a neural network.

The key is the cost function, J.

We want to increase the cost function as much as possible, while inducing minimal changes in X.

To do that, we simply ask, “in what direction should I change X, such that it results in the maximum change in J?”

This is again just a calculus problem - we want to find dJ/dX.

And again, we use Theano to automatically calculate the gradient.

Once you've done that, we simply change X by a small amount in that direction.

$X = X + eps*dJ/dX$

Where eps is a small number.

What you should find that is the image itself doesn't look any different (remember, matplotlib will scale your values to be from 0..255), but the neural network has a good chance of classifying the image wrong, even when it classifies the original image right.

What implications does this have in terms of training a neural network?

You surely don't want your neural network to be so easily fooled, and the key here is something we've already talked about before:

Data augmentation.

The example I like to give: an upside-down cat is still a cat. A cat shifted to the left is still a cat. A rotated cat is still a cat.

So if you only have one cat, you should create these invariant transformations and include them as training data.

Similarly, a cat with some noise is still a cat.

This is like what we did with autoencoders - set some of the inputs to 0. A cat with some 0s is still a cat, so add that as training data.

And like we've seen in this chapter, a cat + some "intelligently designed" noise is still a cat, so include that as training data too.

By including all these forms of data augmentation, you are making your neural network more robust.

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

I do 1:1 coaching and consulting as well.

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.

My first course in deep learning is a lot like the book, but you get to see me derive the formulas and write the code live:

Data Science: Deep Learning in Python

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

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 this 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

When you've got the basics of deep learning down, you're ready to explore alternative architectures. One very popular alternative is the convolutional neural network, created specifically for image classification. These have promising applications in medical imaging, self-driving vehicles, and more. In this course, I show you how to build convolutional nets in Theano and TensorFlow.

Deep Learning: Convolutional Neural Networks in Python

https://www.udemy.com/deep-learning-convolutional-neural-networks-theano-tensorflow

In part 4 of my deep learning series, I take you through unsupervised deep learning methods (that's this book!). 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

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

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

Are you interested in stock prediction, time series, and sequences in general? My Hidden Markov Models course is where you want to be. I teach you not only all the classical theory of HMMs, but I also show you how to write them in Theano using gradient descent! This is great practice for writing deep learning models and it will prepare you well for its sequel, Deep Learning Part 5: Recurrent Neural Networks in Python. You can get the HMM course here:

Unsupervised Machine Learning: Hidden Markov Models in Python

https://udemy.com/unsupervised-machine-learning-hidden-markov-models-in-python

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"!)