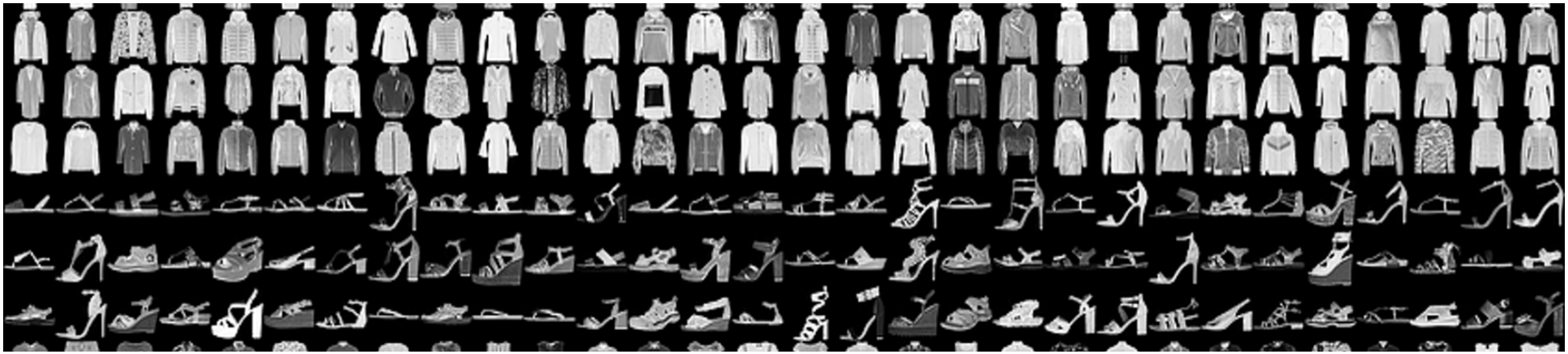
Simple Autoencoder example using Tensorflow in Python on the Fashion MNIST dataset

Soumya Ghosh Follow
Mar 11, 2018 · 6 min read



Autoencoders can be used to solve a lot of problems. The one I'll try to solve here is that of dimensionality reduction. This is a pretty common problem in data science. I've seen it pop up in a lot of projects that I've worked on. If you have structured data, usually how you'd deal with it is use PCA, SVD etc. But here I'll use an autoencoder to get latent features for every image.

In this tutorial, I'll focus more on building a simple tensorflow model. You can build it using keras too. Their website has some really helpful examples of doing just that. If you're interested in a detailed explanation of what autoencoders are and how they work, a quick google search points to some great resources.

I'll be using the MNIST fashion dataset for this demonstration. It contains images of shirts, dresses and shoes and whatnot. Although I'll be working with images, the idea can be extended to any other type of data—text, music etc.

Lets get started! First I import some stuff I'll need to build our model.

```
# Importing tensorflow

import tensorflow as tf

# Importing some more libraries

import numpy as np
import matplotlib.pyplot as plt
```

I downloaded the data from this Kaggle webpage. You can find this dataset at a ton of other placs too. I'll just use the file that has training images.

```
#loading the images
```

```
all_images = np.loadtxt('fashion-mnist_train.csv',\
                  delimiter=',', skiprows=1)[:,1:]


#looking at the shape of the file


print(all_images.shape)
```

I'm reading the file as a numpy array. Feel free to use pandas or something to read the file. But I feel it'll be convenient to deal with a numpy array later on. Its interesting how this data is arranged. Every images is given as an array of 784 features. Actually its just (28*28) pixels flattened out. I use the argument skiprows=1 to ignore the header. The first column is the label of that image—whether it is a shoe or a shirt or whatever. I dont need that information either. That why the [:,1:].

all_images has 60,000 images. You can print out the first image in the training set with print(all_images[0]). You'll notice its an array with values ranging from 0 to 255. Higher the value, darker the pixel. You can reshape the 784 features into 28*28 pixels and print out the image.

```
# printing the array representation of the first image


print("the array of the first image looks like",
all_images[0])


# printing something that actually looks like an image


print("and the actual image looks like")
plt.imshow(all_images[0].reshape(28,28),  cmap='Greys')
```

```
plt.show()
```

Now lets come to building the autoencoder itself. Now you know what both the shape of the input and output are supposed to be. An image goes in, same image comes out. Here, an image is an array of length 784. So, the input and the output layer should have 784 nodes. Now you have the choice of deciding how many hidden layers there should be and what they'd look like. I've decided to keep it simple and put just two hidden layers which have 32 nodes each. You had probably read that autoencoders are made up of an encoder and a decoder. Here the first two layer make up the encoder and the last two are the decoder.

```
# Deciding how many nodes wach layer should have

n_nodes_inpl = 784  #encoder
n_nodes_hl1  = 32  #encoder

n_nodes_hl2  = 32  #decoder
n_nodes_outl = 784  #decoder
```

Next I assign random values to the weights and biases associated to the different layers of our model. For example, *tf.random_normal([n_nodes_inpl,n_nodes_hl1])* is a 784*32 weight matrix and *tf.random_normal([n_nodes_hl1])* is a bias matrix. *A*ll weights and biases are *variables* here—they'll be updated continuosly as we train the model. I like to save them in dictionaries. Remember that in a fully connected neural net, if first layer has *m* nodes and second layer has *n* nodes, then their connection has *m*n* weights and *n* biases.

```
# first hidden layer has 784*32 weights and 32 biases

hidden_1_layer_vals = {
'weights':tf.Variable(tf.random_normal([n_nodes_inpl,n_nodes
_hl1])),
'biases':tf.Variable(tf.random_normal([n_nodes_hl1]))  }

# second hidden layer has 32*32 weights and 32 biases

hidden_2_layer_vals = {
'weights':tf.Variable(tf.random_normal([n_nodes_hl1,
n_nodes_hl2])),
'biases':tf.Variable(tf.random_normal([n_nodes_hl2]))  }

# second hidden layer has 32*784 weights and 784 biases

output_layer_vals = {
'weights':tf.Variable(tf.random_normal([n_nodes_hl2,n_nodes_
outl])),
'biases':tf.Variable(tf.random_normal([n_nodes_outl])) }
```

Now I'll define the neural net.

```
# image with shape 784 goes in
input_layer = tf.placeholder('float', [None, 784])

# multiply output of input_layer wth a weight matrix and add
biases

layer_1 = tf.nn.sigmoid(

tf.add(tf.matmul(input_layer,hidden_1_layer_vals['weights'])
```

```
,
       hidden_1_layer_vals['biases']))


# multiply output of layer_1 wth a weight matrix and add
biases


layer_2 = tf.nn.sigmoid(

tf.add(tf.matmul(layer_1,hidden_2_layer_vals['weights']),
       hidden_2_layer_vals['biases']))


# multiply output of layer_2 wth a weight matrix and add
biases


output_layer =
tf.matmul(layer_2,output_layer_vals['weights']) +
               output_layer_vals['biases']


# output_true shall have the original image for error
calculations


output_true = tf.placeholder('float', [None, 784])


# define our cost function
meansq =    tf.reduce_mean(tf.square(output_layer -
output_true))


# define our optimizer
learn_rate = 0.1   # how fast the model should learn
optimizer =
tf.train.AdagradOptimizer(learn_rate).minimize(meansq)
```

What's happening here is—we are iteratively multiplying the output of each layer with the corresponding weight matrix , adding the bias to it and then taking it through the activation function— for example,

tf.matmul(input_layer, hidden_1_layer_vals['weights']) multiplies the *input_layer* with a 784*32 weight matrix, adds the 32*1 bias matrix to it, and then takes it through the sigmoid function.

input_layer and *output_true* are just placeholders here. It means I can tell the model what these should hold when I run it. And these are supposed to hold the images that we put into the model. So an image goes into the *input_layer* and comes out of the *output_layer*. Then I will compare this to *output_true* which will have the original image, see how close I get. I'll use MSE as a cost function and sigmoid as an activation function for both the hidden layers.

Now lets run this bad boy !

```
# initialising stuff and starting the session

init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)

# defining batch size, number of epochs and learning rate

batch_size = 100  # how many images to use together for
training
hm_epochs =1000    # how many times to go through the entire
dataset
tot_images = 60000 # total number of images

# running the model for a 1000 epochs taking 100 images in
batches
# total improvement is printed out after each epoch

for epoch in range(hm_epochs):
```

```
    epoch_loss = 0    # initializing error as 0

    for i in range(int(tot_images/batch_size)):

        epoch_x = all_images[ i*batch_size :
(i+1)*batch_size ]

        _, c = sess.run([optimizer, meansq],\
               feed_dict={input_layer: epoch_x, \
               output_true: epoch_x})

        epoch_loss += c

print('Epoch', epoch, '/', hm_epochs, 'loss:',epoch_loss)
```

And we're done! You'll notice there are two loops in the code. The outer one is for the epoch i.e. one loop means you've gone through the whole dataset once. And the inner loop is for batching, where you pass 100 images in batches to train the network, until you've gone through the whole dataset once. Every time you run the optimizer, all the weights and variables get updated in your network. I also print out the progress I'm making after every epoch.

A 1000 epochs takes a lot of time on my old macbook. You can try with a bit more. Now to finish things off, lets run one image thorugh the autoencoder and see what the encoded and decoded ouput looks like.

```
# pick any image
```

```
any_image = all_images[999]

# run it though the autoencoder

output_any_image = sess.run(output_layer,\
                   feed_dict={input_layer:[any_image]})

# run it though just the encoder

encoded_any_image = sess.run(layer_1,\
                   feed_dict={input_layer:[any_image]})

# print the original image

plt.imshow(any_image(28,28),  cmap='Greys')
plt.show()

# print the encoding

print(encoded_any_image)
```

Now if you want, you can have every image mapped to their encodings. Couple more things beofre I wrap up, you might have noticed I didnt validate my model against a test set here, which I should. Also, using something like a CNN instead would also make more sense here, something for me to explore next.

Going from 784 features to 32 isn't that big of a deal but this is just an example. Imagine if you have coloured images of 720*720*3 (3 because of RGB values) = 1.5 million features and you needed to get it down to 64 or 32.

Thank you for making it to the end! If this was helpful to you, please leave a comment or a like :)