IBM Developer
SKILLS NETWORK

AUTOENCODERS

Welcome to this notebook about autoencoders. **In this notebook you will learn the definition of an autoencoder, how it works, and see an implementation in TensorFlow.**

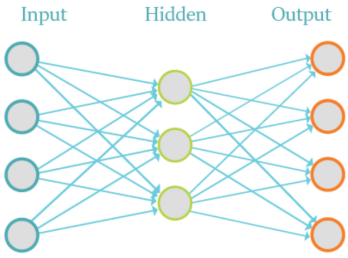
Table of Contents

By the end of this notebook, you should be able to create simple autoencoders apply them to problems in the field of unsupervised learning.

Introduction

An autoencoder, also known as autoassociator or Diabolo networks, is an artificial neural network employed to recreate the given input. It takes a set of unlabeled inputs, encodes them and then tries to extract the most valuable information from them. They are used for feature extraction, learning generative models of data, dimensionality reduction and can be used for compression.
A 2006 paper named Reducing the Dimensionality of Data with Neural Networks, done by G. E. Hinton and R. R. Salakhutdinov, showed better results than years of refining other types of network, and was a breakthrough in the field of Neural Networks, a field that was "stagnant" for 10 years.

Now, autoencoders, based on Restricted Boltzmann Machines, are employed in some of the largest deep learning applications. They are the building blocks of Deep Belief Networks (DBN).



Feature Extraction and Dimensionality Reduction

An example given by Nikhil Buduma in KdNuggets (link) gives an excellent explanation of the utility of this type of Neural Network.

Say that you want to extract the emotion that a person in a photograph is feeling. Take the following 256x256 pixel grayscale picture as an example:

If we just use the raw image, we have too many dimensions to analyze. This image is 256x256 pixels, which corresponds to an input vector of 65536 dimensions! Conventional cell phones can produce images in the 4000 x 3000 pixels range, which gives us 12 million dimensions to analyze.

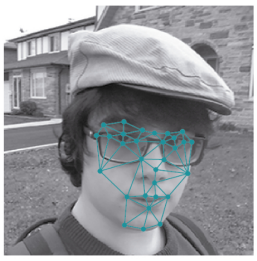
This is particularly problematic, since the difficulty of a machine learning problem is vastly increased as more dimensions are involved. According to a 1982 study by C.J. Stone (link), the time to fit a model, is optimal if:

$$m^{-p/(2p+d)}$$

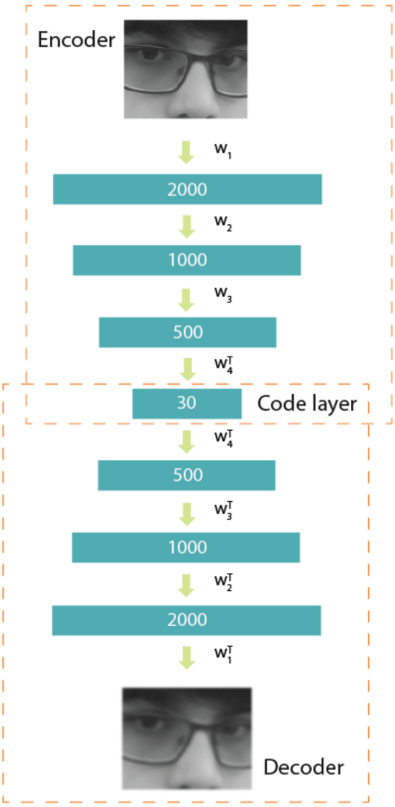
Where:
m: Number of data points
d: Dimensionality of the data
p: Number of Parameters in the model

As you can see, it increases exponentially!

Returning to our example, we don't need to use all of the 65,536 dimensions to classify an emotion. A human identifies emotions according to specific facial expressions, and some key features, like the shape of the mouth and eyebrows.



Autoencoder Structure



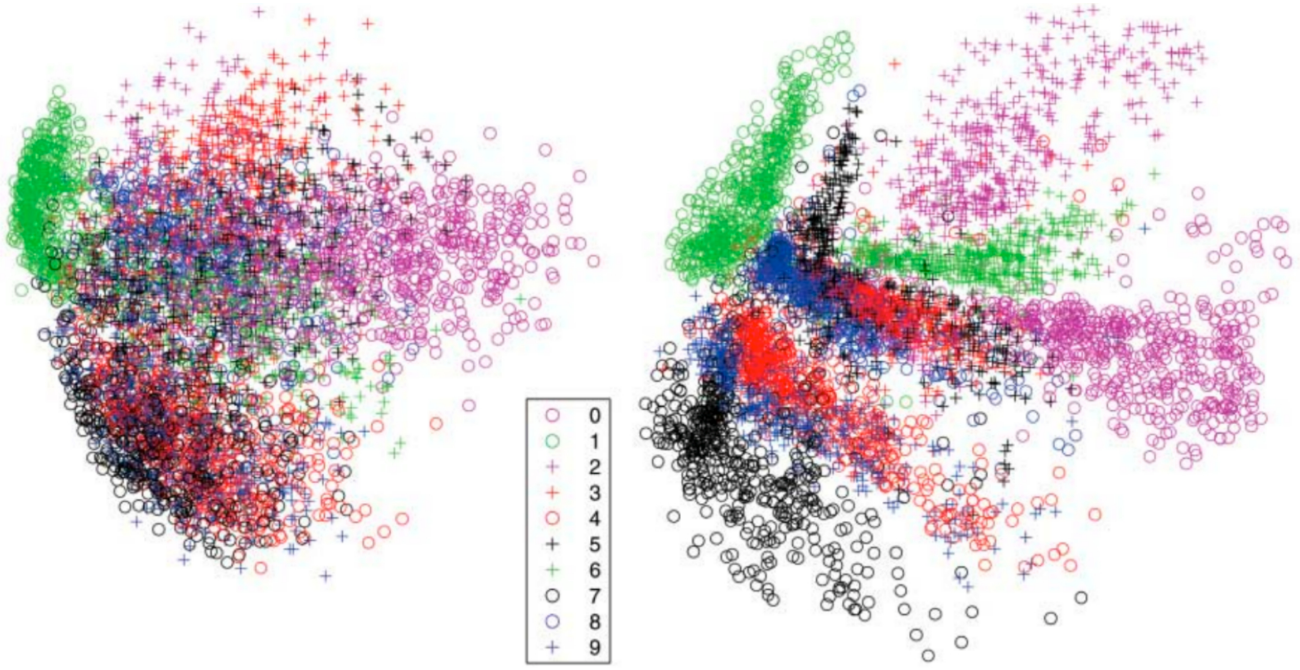
An autoencoder can be divided in two parts, the encoder and the decoder.

The encoder needs to compress the representation of an input. In this case, we are going to reduce the dimensions of the image of the example face from 2000 dimensions to only 30 dimensions. We will acomplish this by running the data through the layers of our encoder.

The decoder works like encoder network in reverse. It works to recreate the input as closely as possible. The training procedure produces at the center of the network a compressed, low dimensional representation that can be decoded to obtain the higher dimensional representation with minimal loss of information between the input and the output.

Performance

After training has been completed, you can use the encoded data as a reliable low dimensional representation of the data. This can be applied to many problems where dimensionality reduction seems appropriate.



This image was extracted from the G. E. Hinton and R. R. Salakhutdinovcomparing's paper, on the two-dimensional reduction for 500 digits of the MNIST, with PCA (Principal Component Analysis) on the left and autoencoder on the right. We can see that the autoencoder provided us with a better separation of data.

Training: Loss function

An autoencoder uses the Loss function to properly train the network. The Loss function will calculate the differences between our output and the expected results. After that, we can minimize this error with gradient descent. There are many types of Loss functions, and it is important to consider the type of problem (classification, regression, etc.) when choosing this funtion.

Binary Values:

$$L(W) = -\sum_k (x_k log(\hat{x}_k) + (1 - x_k) \log(1 - \hat{x}_k))$$

For binary values, we can use an equation based on the sum of Bernoulli's cross-entropy. This loss function is best for binary classification problems.

x_k is one of our inputs and
$\hat{x}\_k$
is the respective output. Note that:

$$\hat{x} = f(x, W)$$

where W is the full parameter set of the neural network.

We use this function so that when $x_k = 1$, we want the calculated value of
$\hat{x}\_k$
to be very close to one, and likewise if $x_k = 0$.

If the value is one, we just need to calculate the first part of the formula, that is,
$-x_k log($
$\hat{x}\_k)$
. Which, turns out to just calculate $-log(\hat{x}\_k)$. We explicitly exclude the second term to avoid numerical difficulties when computing the logarithm of very small numbers.

Likewise, if the value is zero, we need to calculate just the second part,
$(1 - x_k)$
$\log(1 - \hat{x}\_k))$
- which turns out to be $log(1 - \hat{x}\_k)$.

Real values:

$$L(W) = -\frac{1}{2}\sum_k (\hat{x}_k - x_k)^2$$

For data where the value (not category) is important to reproduce, we can use the sum of squared errors (SSE) for our Loss function. This function is usually used in regressions.

As it was with the above example, x_k is one of our inputs and
$\hat{x}\_k$
is the respective output, and we want to make our output as similar as possible to our input.

Computing Gradient

The gradient of the loss function is an important and complex function. It is defined as:

$$\nabla\_W L(W)\_j = \frac{\partial f(x, W)}{\partial W_j}$$

Fortunately for us, TensorFlow computes these complex functions automatically when we define our functions that are used to compute loss! They automatically manage the backpropagation algorithm, which is an efficient way of computing the gradients in complex neural networks.

Code

We are going to use the MNIST dataset for our example. The following code was created by Aymeric Damien. You can find some of his code in here. We made some modifications which allow us to import the datasets to Jupyter Notebooks.

Let's call our imports and make the MNIST data available to use.

[ ]:
```
!pip install tensorflow==2.2.0rc0
```

[ ]:
```
#from __future__ import division, print_function, absolute_import

import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

if not tf.__version__ == '2.2.0-rc0':
    print(tf.__version__)
    raise ValueError('please upgrade to TensorFlow 2.2.0-rc0, or restart your Kernel (Kernel->Restart & Clear Output)')
```

[ ]:
```
# Import MINST data
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

[ ]:
```
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
```

[ ]:
```
y_train = y_train.astype('float32') / 255.
y_test = y_test.astype('float32') / 255.
```

[ ]:
```
x_image_train = tf.reshape(x_train, [-1,28,28,1])
x_image_train = tf.cast(x_image_train, 'float32')

x_image_test = tf.reshape(x_test, [-1,28,28,1])
x_image_test = tf.cast(x_image_test, 'float32')
```

We use the tf.keras.layers.Flatten() function to prepare the training data to be compatible with the encoding and decoding layer

[ ]:
```
print(x_train.shape)
```

[ ]:
```
flatten_layer = tf.keras.layers.Flatten()
x_train = flatten_layer(x_train)
```

Notice how the `x_train.shape` changes from (60000,28,28) to (60000, 784)

[ ]:
```
print(x_train.shape)
```

Now, let's give the parameters that are going to be used by our NN.

[ ]:
```
learning_rate = 0.01
training_epochs = 20
batch_size = 256
display_step = 1
examples_to_show = 10
global_step = tf.Variable(0)
total_batch = int(len(x_train) / batch_size)

# Network Parameters
n_hidden_1 = 256 # 1st layer num features
n_hidden_2 = 128 # 2nd layer num features
encoding_layer = 32 # final encoding bottleneck features
n_input = 784 # MNIST data input (img shape: 28*28)
```

encoder

Now we need to create our encoder. For this, we are going to use tf.keras.layers.Dense with sigmoidal activation functions. Sigmoidal functions delivers great results with this type of network. This is due to having a good derivative that is well-suited to backpropagation. We can create our encoder using the sigmoidal function like this:

[ ]:
```
enocoding_1 = tf.keras.layers.Dense(n_hidden_1, activation=tf.nn.sigmoid)
encoding_2 = tf.keras.layers.Dense(n_hidden_2, activation=tf.nn.sigmoid)
encoding_final = tf.keras.layers.Dense(encoding_layer, activation=tf.nn.relu)

# Building the encoder
def encoder(x):
    x_reshaped = flatten_layer(x)
    # Encoder first layer with sigmoid activation #1
    layer_1 = enocoding_1(x_reshaped)
    # Encoder second layer with sigmoid activation #2
    layer_2 = encoding_2(layer_1)
    code = encoding_final(layer_2)
    return code
```

decoder

You can see that the layer_1 in the encoder is the layer_2 in the decoder and vice-versa.

[ ]:
```
decoding_1 = tf.keras.layers.Dense(n_hidden_2, activation=tf.nn.sigmoid)
decoding_2 = tf.keras.layers.Dense(n_hidden_1, activation=tf.nn.sigmoid)
decoding_final = tf.keras.layers.Dense(n_input)
# Building the decoder
def decoder(x):
    # Decoder first layer with sigmoid activation #1
    layer_1 = decoding_1(x)
    # Decoder second layer with sigmoid activation #2
    layer_2 = decoding_2(layer_1)
    decode = self.decoding_final(layer_2)
    return decode
```

Let's construct our model. We define a `cost` function to calculate the loss and a `grad` function to calculate gradients that will be used in backpropagation.

[ ]:
```
class AutoEncoder(tf.keras.Model):
    def __init__(self):
        super(AutoEncoder, self).__init__()
```

```
        self.n_hidden_1 = n_hidden_1 # 1st layer num features
        self.n_hidden_2 = n_hidden_2 # 2nd layer num features
        self.encoding_layer = encoding_layer
        self.n_input = n_input # MNIST data input (img shape: 28*28)

        self.flatten_layer = tf.keras.layers.Flatten()
        self.enocoding_1 = tf.keras.layers.Dense(self.n_hidden_1, activation=tf.nn.sigmoid)
        self.encoding_2 = tf.keras.layers.Dense(self.n_hidden_2, activation=tf.nn.sigmoid)
        self.encoding_final = tf.keras.layers.Dense(self.encoding_layer, activation=tf.nn.relu)
        self.decoding_1 = tf.keras.layers.Dense(self.n_hidden_2, activation=tf.nn.sigmoid)
        self.decoding_2 = tf.keras.layers.Dense(self.n_hidden_1, activation=tf.nn.sigmoid)
        self.decoding_final = tf.keras.layers.Dense(self.n_input)


    # Building the encoder
    def encoder(self,x):
        #x = self.flatten_layer(x)
        layer_1 = self.enocoding_1(x)
        layer_2 = self.encoding_2(layer_1)
        code = self.encoding_final(layer_2)
        return code
----------
    # Building the decoder
    def decoder(self, x):
        layer_1 = self.decoding_1(x)
        layer_2 = self.decoding_2(layer_1)
        decode = self.decoding_final(layer_2)
        return decode

----------
    def call(self, x):
        encoder_op  = self.encoder(x)
        # Reconstructed Images
        y_pred = self.decoder(encoder_op)
        return y_pred
----------
def cost(y_true, y_pred):
    loss = tf.losses.mean_squared_error(y_true, y_pred)
    cost = tf.reduce_mean(loss)
    return cost

def grad(model, inputs, targets):
    #print('shape of inputs : ',inputs.shape)
    #targets = flatten_layer(targets)
    with tf.GradientTape() as tape:------
        reconstruction = model(inputs)
        loss_value = cost(targets, reconstruction)
    return loss_value, tape.gradient(loss_value, model.trainable_variables),reconstruction
```

For training we will run for 20 epochs.

[ ]:
```
model = AutoEncoder()
optimizer = tf.keras.optimizers.RMSprop(learning_rate)

for epoch in range(training_epochs):
    for i in range(total_batch):
        x_inp = x_train[i : i + batch_size]
        loss_value, grads, reconstruction = grad(model, x_inp, x_inp)
        optimizer.apply_gradients(zip(grads, model.trainable_variables))
    # Display logs per epoch step
    if epoch % display_step == 0:
        print("Epoch:", '%04d' % (epoch+1),
              "cost=", "{:.9f}".format(loss_value))

print("Optimization Finished!")
```

Now, let's apply encoder and decoder for our tests.

[ ]:
```
# Applying encode and decode over test set
encode_decode = model(flatten_layer(x_image_test[:examples_to_show]))
```

Let's simply visualize our graphs!

[ ]:
```
# Compare original images with their reconstructions
f, a = plt.subplots(2, 10, figsize=(10, 2))
for i in range(examples_to_show):
    a[0][i].imshow(np.reshape(x_image_test[i], (28, 28)))
    a[1][i].imshow(np.reshape(encode_decode[i], (28, 28)))
```

As you can see, the reconstructions were successful. It can be seen that some noise were added to the image.

Want to learn more?

Running deep learning programs usually needs a high performance platform. **PowerAI** speeds up deep learning and AI. Built on IBM's Power Systems, **PowerAI** is a scalable software platform that accelerates deep learning and AI with blazing performance for individual users or enterprises. The **PowerAI** platform supports popular machine learning libraries and dependencies including TensorFlow, Caffe, Torch, and Theano. You can use PowerAI on IMB Cloud.

Also, you can use **Watson Studio** to run these notebooks faster with bigger datasets.**Watson Studio** is IBM's leading cloud solution for data scientists, built by data scientists. With Jupyter notebooks, RStudio, Apache Spark and popular libraries pre-packaged in the cloud, **Watson Studio** enables data scientists to collaborate on their projects without having to install anything. Join the fast-growing community of **Watson Studio** users today with a free account at Watson Studio.This is the end of this lesson. Thank you for reading this notebook, and good luck on your studies.

Thanks for completing this lesson!

Created by Francisco Magioli, Erich Natsubori Sato, Saeed Aghabozorgi

Updated to TF 2.X by Samaya Madhavan

References:

- https://en.wikipedia.org/wiki/Autoencoder
- http://ufldl.stanford.edu/tutorial/unsupervised/Autoencoders/
- http://www.slideshare.net/billlangjun/simple-introduction-to-autoencoder
- http://www.slideshare.net/danieljohnlewis/piotr-mirowski-review-autoencoders-deep-learning-ciuuk14
- https://cs.stanford.edu/~quocle/tutorial2.pdf
- https://gist.github.com/hussius/1534135a419bb0b957b9
- http://www.deeplearningbook.org/contents/autoencoders.html
- http://www.kdnuggets.com/2015/03/deep-learning-curse-dimensionality-autoencoders.html/

- https://www.youtube.com/watch?v=xTU79Zs4XKY
- http://www-personal.umich.edu/~jizhu/jizhu/wuke/Stone-AoS82.pdf

---