ML0120EN-5.1-Review-Autoencoders

Loading...

AUTOENCODERS

Welcome to this notebook about autoencoders. **In this notebook you will find an explanation of what is an autoencoder, how it works, and see an implementation of an autoencoder in TensorFlow.**

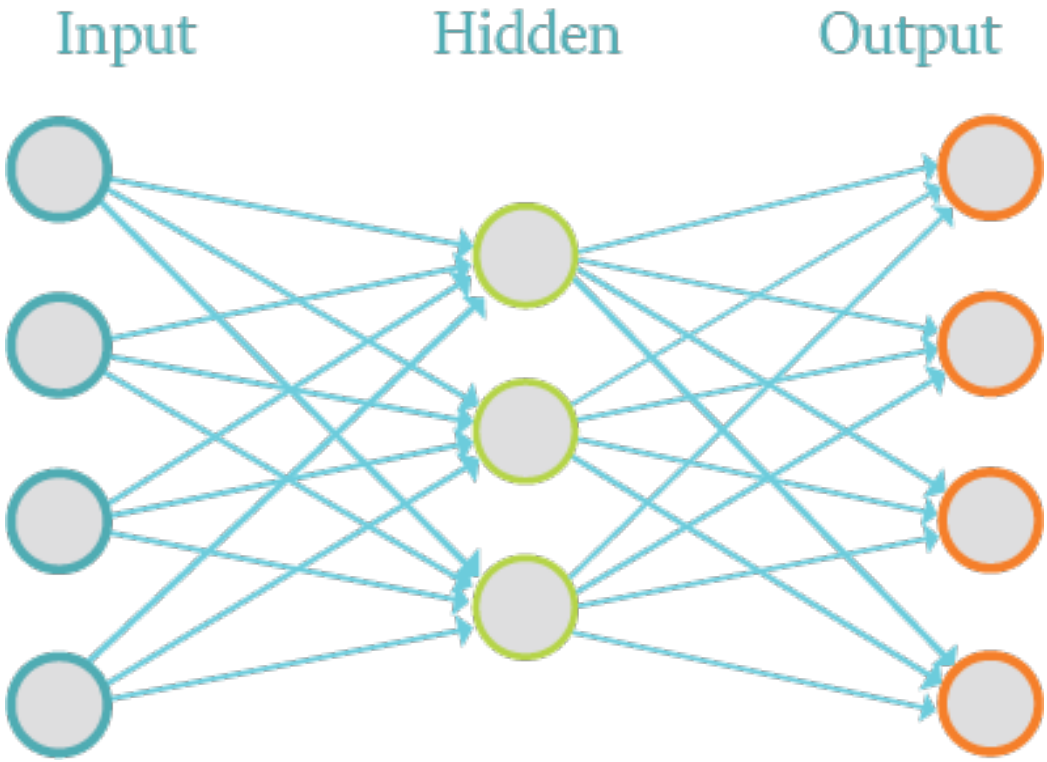
Table of Contents

By the end of this notebook, you should be able to create simple autoencoders and how to apply them to problems that involves 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).



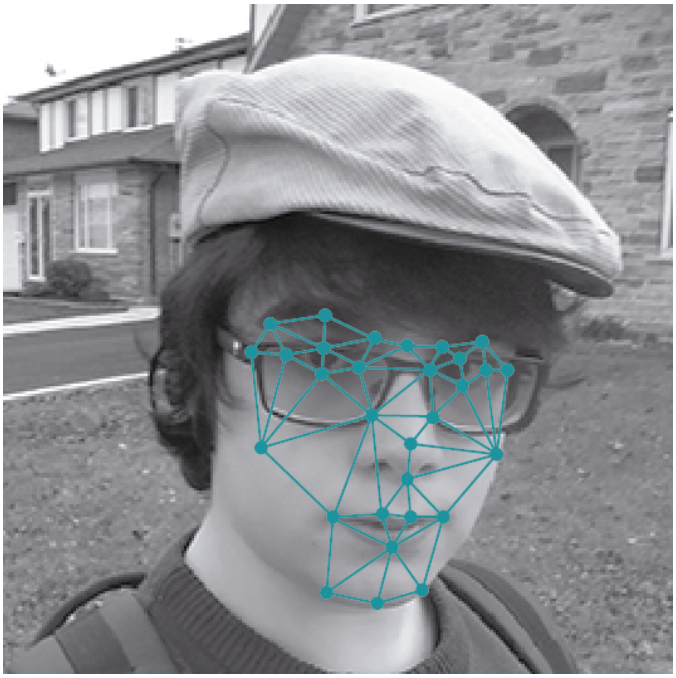
Feature Extraction and Dimensionality Reduction

An example given by Nikhil Buduma in KdNuggets (link) which gave an excellent explanation of the utility of this type of Neural Network.
Say that you want to extract what emotion the person in a photography is feeling. Using the following 256x256 pixel grayscale picture as an example:

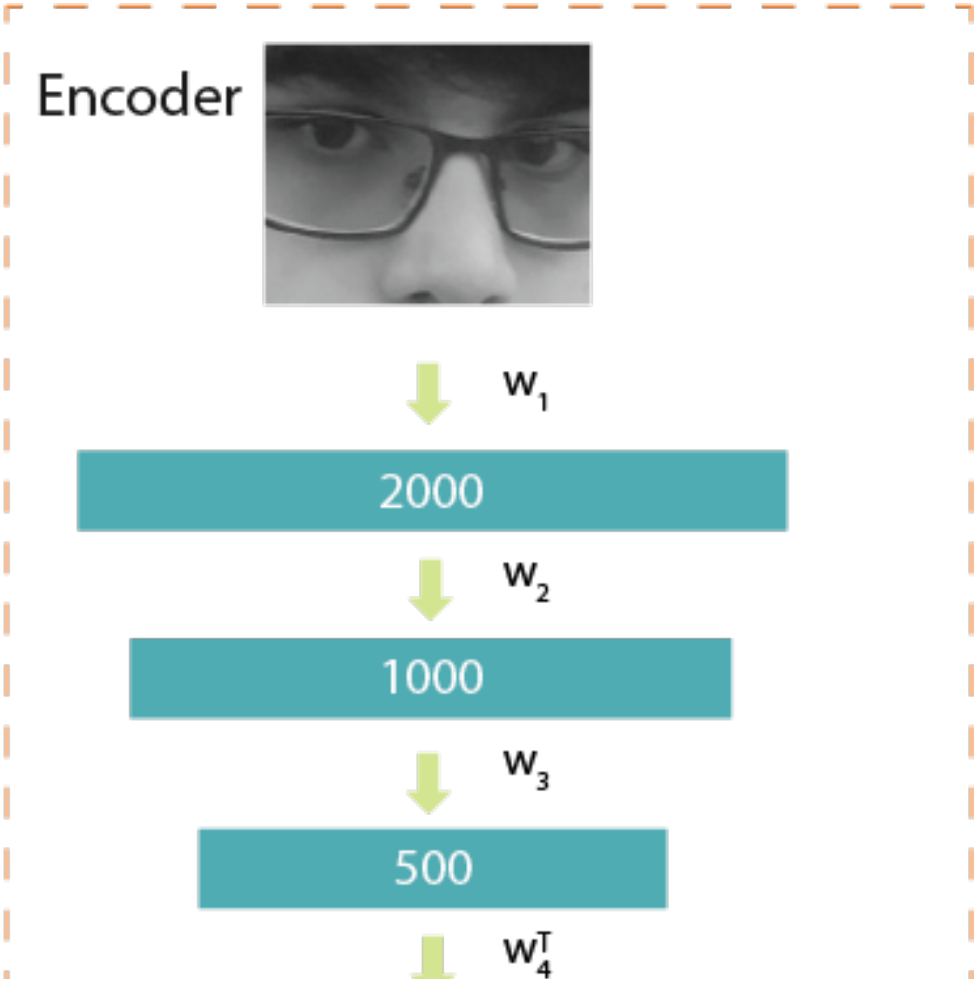
But when use this picture we start running into a bottleneck! Because this image being 256x256 pixels in size correspond with an input vector of 65536 dimensions! If we used an image produced with conventional cellphone cameras, that generates images of 4000 x 3000 pixels, we would have 12 million dimensions to analyze.
This bottleneck is further problematized as the difficulty of a machine learning problem is 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:

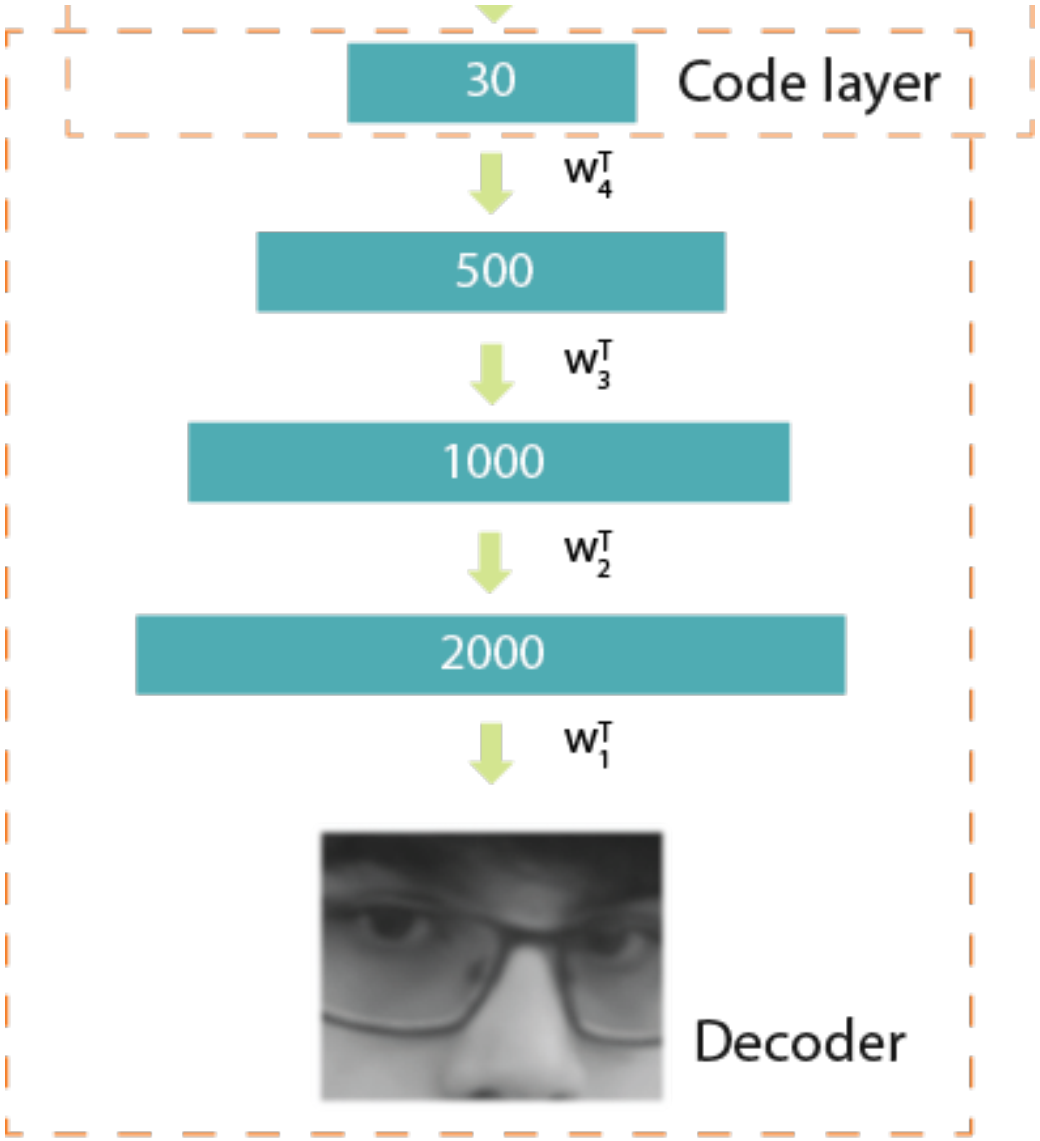
$$m^{-p/(2p+d)}$$

Where:
m: Number of data points
d: Dimensionality of the data
p: Parameter that depends on 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 identify emotions according to some specific facial expression, some **key features**, like the shape of the mouth and eyebrows.



Autoencoder Structure

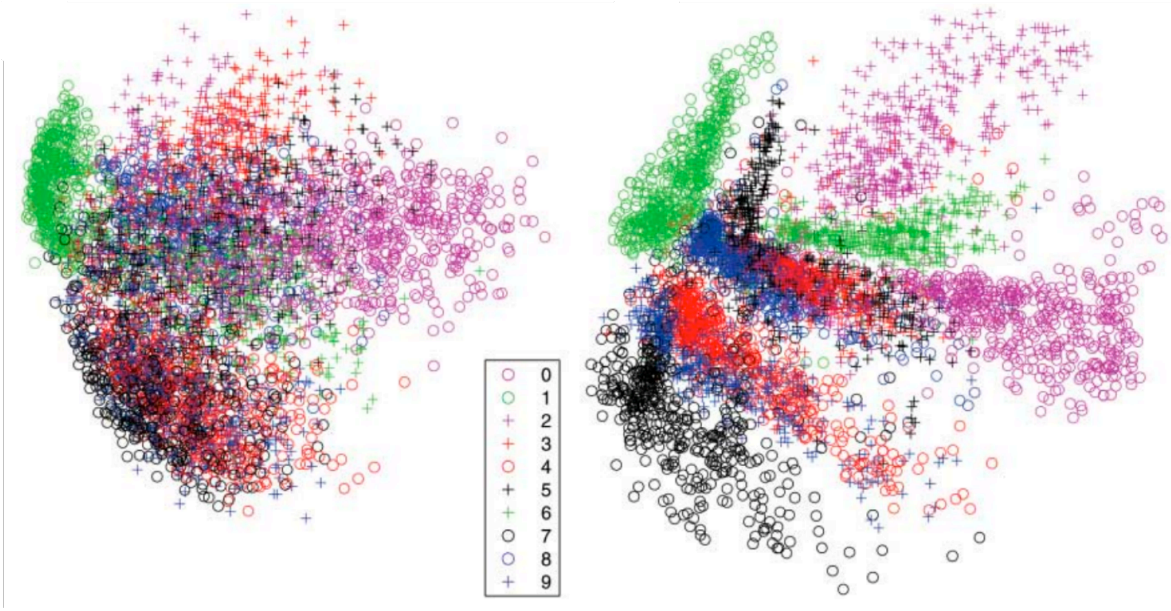




An autoencoder can be divided in two parts, the **encoder** and the **decoder**. The encoder needs to compress the representation of an input. In this case we are going to reduce the dimension the face of our actor, from 2000 dimensions to only 30 dimensions, by running the data through layers of our encoder. The decoder works like encoder network in reverse. It works to recreate the input, as closely as possible. This plays an important role during training, because it forces the autoencoder to select the most important features in the compressed representation.

Performance

After the training has been done, you can use the encoded data as a reliable dimensionally-reduced data, applying it to any 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 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 more than one type of Loss function, it depends on the type of data.

Binary Values:

$$l(f(x)) = - \sum_{k} (x_k log(\hat{x}_k) + (1 - x_k) \log (1 - \hat{x}_k) \ )$$

For binary values, we can use an equation based on the sum of Bernoulli's cross-entropy.
x_k is one of our inputs and $\hat{x}_k$ is the respective output.
We use this function so that if x_k equals to one, we want to push $\hat{x}_k$ as close as possible to one. The same if x_k equals to zero.
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 $- log(\hat{x}_k)$.
And 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:

$$l(f(x)) = - \frac{1}{2}\sum_{k} (\hat{x}_k- x_k \ )^2$$

As the above function would behave badly with inputs that are not 0 or 1, we can use the sum of squared differences for our Loss function. If you use this loss function, it's necessary that you use a linear activation function for the output layer.
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.

Loss Gradient:

$$\nabla_{\hat{a}(x^{(t)})} \ l(\ f(x^{(t)})) = \hat{x}^{(t)} - x^{(t)} $$

We use the gradient descent to reach the local minimum of our function $l(\ f(x^{(t)})$, taking steps towards the negative of the gradient of the function in the current point.
Our function about the gradient $(\nabla_{\hat{a}(x^{(t)})})$ of the loss of $l(\ f(x^{(t)})$ in the preactivation of the output layer.
It's actually a simple formula, it is done by calculating the difference between our output $\hat{x}^{(t)}$ and our input $x^{(t)}$.
Then our network backpropagates our gradient $\nabla_{\hat{a}(x^{(t)})} \ l(\ f(x^{(t)}))$ through the network using **backpropagation**.

Code

For this part, we walk through a lot of Python 2.7.11 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 for us to import the datasets to Jupyter Notebooks.

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

In [1]:

```
/home/jupyterlab/conda/envs/python/lib/python3.6/
site-packages/tensorflow/python/framework/dtypes.py:
519: FutureWarning: Passing (type, 1) or '1type' as a
synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) /
```

```
'(1,)type'.
  _np_qint8 = np.dtype([("qint8", np.int8, 1)])
/home/jupyterlab/conda/envs/python/lib/python3.6/
site-packages/tensorflow/python/framework/dtypes.py:
520: FutureWarning: Passing (type, 1) or '1type' as a
synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) /
'(1,)type'.
  _np_quint8 = np.dtype([("quint8", np.uint8, 1)])
/home/jupyterlab/conda/envs/python/lib/python3.6/
site-packages/tensorflow/python/framework/dtypes.py:
521: FutureWarning: Passing (type, 1) or '1type' as a
synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) /
'(1,)type'.
  _np_qint16 = np.dtype([("qint16", np.int16, 1)])
/home/jupyterlab/conda/envs/python/lib/python3.6/
site-packages/tensorflow/python/framework/dtypes.py:
522: FutureWarning: Passing (type, 1) or '1type' as a
synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) /
'(1,)type'.
  _np_quint16 = np.dtype([("quint16", np.uint16, 1)])
/home/jupyterlab/conda/envs/python/lib/python3.6/
site-packages/tensorflow/python/framework/dtypes.py:
523: FutureWarning: Passing (type, 1) or '1type' as a
synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) /
'(1,)type'.
  _np_qint32 = np.dtype([("qint32", np.int32, 1)])
/home/jupyterlab/conda/envs/python/lib/python3.6/
site-packages/tensorflow/python/framework/dtypes.py:
528: FutureWarning: Passing (type, 1) or '1type' as a
synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) /
'(1,)type'.
  np_resource = np.dtype([("resource", np.ubyte, 1)])

WARNING:tensorflow:From <ipython-input-1-
aeda475fcce4>:10: read_data_sets (from
tensorflow.contrib.learn.python.learn.datasets.mnist)
```

```
is deprecated and will be removed in a future
version.
Instructions for updating:
Please use alternatives such as official/mnist/
dataset.py from tensorflow/models.
WARNING:tensorflow:From /home/jupyterlab/conda/envs/
python/lib/python3.6/site-packages/tensorflow/
contrib/learn/python/learn/datasets/mnist.py:260:
maybe_download (from
tensorflow.contrib.learn.python.learn.datasets.base)
is deprecated and will be removed in a future
version.
Instructions for updating:
Please write your own downloading logic.
WARNING:tensorflow:From /home/jupyterlab/conda/envs/
python/lib/python3.6/site-packages/tensorflow/
contrib/learn/python/learn/datasets/base.py:252:
_internal_retry.<locals>.wrap.<locals>.wrapped_fn
(from
tensorflow.contrib.learn.python.learn.datasets.base)
is deprecated and will be removed in a future
version.
Instructions for updating:
Please use urllib or similar directly.
Successfully downloaded train-images-idx3-ubyte.gz
9912422 bytes.
WARNING:tensorflow:From /home/jupyterlab/conda/envs/
python/lib/python3.6/site-packages/tensorflow/
contrib/learn/python/learn/datasets/mnist.py:262:
extract_images (from
tensorflow.contrib.learn.python.learn.datasets.mnist)
is deprecated and will be removed in a future
version.
Instructions for updating:
Please use tf.data to implement this functionality.
Extracting /tmp/data/train-images-idx3-ubyte.gz
Successfully downloaded train-labels-idx1-ubyte.gz
28881 bytes.
WARNING:tensorflow:From /home/jupyterlab/conda/envs/
python/lib/python3.6/site-packages/tensorflow/
contrib/learn/python/learn/datasets/mnist.py:267:
```

```
extract_labels (from
tensorflow.contrib.learn.python.learn.datasets.mnist)
is deprecated and will be removed in a future
version.
Instructions for updating:
Please use tf.data to implement this functionality.
Extracting /tmp/data/train-labels-idx1-ubyte.gz
WARNING:tensorflow:From /home/jupyterlab/conda/envs/
python/lib/python3.6/site-packages/tensorflow/
contrib/learn/python/learn/datasets/mnist.py:110:
dense_to_one_hot (from
tensorflow.contrib.learn.python.learn.datasets.mnist)
is deprecated and will be removed in a future
version.
Instructions for updating:
Please use tf.one_hot on tensors.
Successfully downloaded t10k-images-idx3-ubyte.gz
1648877 bytes.
Extracting /tmp/data/t10k-images-idx3-ubyte.gz
Successfully downloaded t10k-labels-idx1-ubyte.gz
4542 bytes.
Extracting /tmp/data/t10k-labels-idx1-ubyte.gz
WARNING:tensorflow:From /home/jupyterlab/conda/envs/
python/lib/python3.6/site-packages/tensorflow/
contrib/learn/python/learn/datasets/mnist.py:290:
DataSet.__init__ (from
tensorflow.contrib.learn.python.learn.datasets.mnist)
is deprecated and will be removed in a future
version.
Instructions for updating:
Please use alternatives such as official/mnist/
dataset.py from tensorflow/models.
```

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

In [2]:

Now we need to create our encoder. For this, we are going to use sigmoidal 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:

In [3]:

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

In [4]:

Let's construct our model. In the variable `cost` we have the loss function and in the `optimizer` variable we have our gradient used for backpropagation.

In [5]:

For training we will run for 20 epochs.

In [6]:

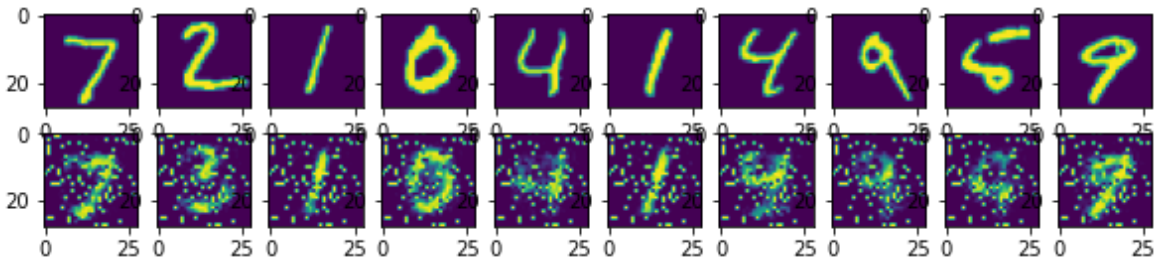
```
Epoch: 0001 cost= 0.216099694
Epoch: 0002 cost= 0.177453369
Epoch: 0003 cost= 0.161362544
Epoch: 0004 cost= 0.151179627
Epoch: 0005 cost= 0.142130136
Epoch: 0006 cost= 0.135626018
Epoch: 0007 cost= 0.134643912
Epoch: 0008 cost= 0.127500087
Optimization Finished!
```

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

In [7]:

Let's simply visualize our graphs!

In [8]:



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

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/~quoc/le/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