Autoencoder With Neural Networks

"Autoencoding" is a data compression algorithm where the compression and decompression functions are 1) data-specific, 2) lossy, and 3) learned automatically from examples rather than engineered by a human. Additionally, in almost all contexts where the term "autoencoder" is used, the compression and decompression functions are implemented with unsupervised neural networks (i.e., no class labels or labeled data).

1) Autoencoders are data-specific, which means that they will only be able to compress data similar to what they have been trained on. This is different from, say, the MPEG-2 Audio Layer III (MP3) compression algorithm, which only holds assumptions about "sound" in general, but not about specific types of sounds.An autoencoder trained on pictures of faces would do a rather poor job of compressing pictures of trees, because the features it would learn would be face-specific.

2) Autoencoders are lossy, which means that the decompressed outputs will be degraded compared to the original inputs

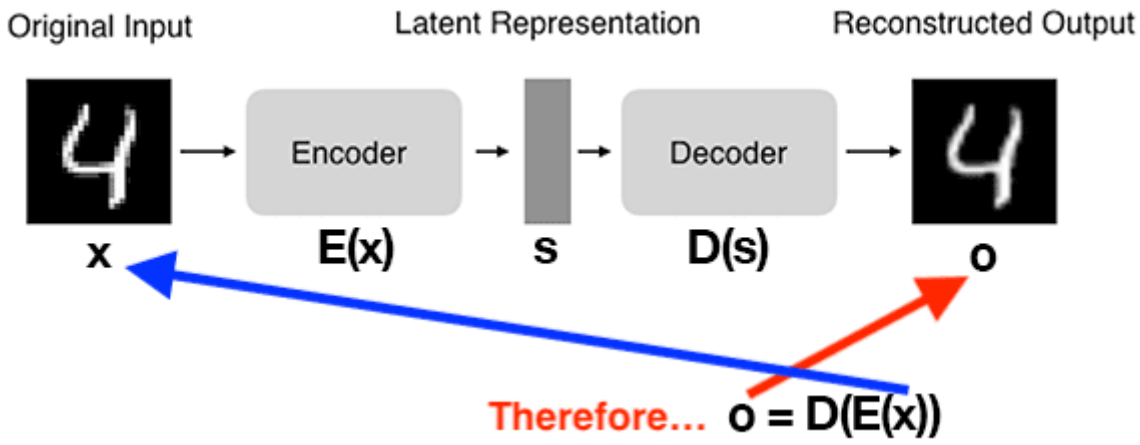
3) Autoencoders are learned automatically from data examples, which is a useful property: it means that it is easy to train specialized instances of the algorithm that will perform well on a specific type of input. It doesn't require any new engineering, just appropriate training data.

To build an autoencoder, you need three things: an encoding function, a decoding function, and a distance function between the amount of information loss between the compressed representation of your data and the decompressed representation (i.e. a "loss" function).

Autoencoders are typically used for:

- Dimensionality reduction (i.e., think PCA but more powerful/intelligent).
- Denoising (ex., removing noise and preprocessing images to improve OCR accuracy)
- Anomaly/outlier detection (ex., detecting mislabeled data points in a dataset or detecting when an input data point falls well outside our typical data distribution).

Today two interesting practical applications of autoencoders are data denoising (which we feature later in this post), and dimensionality reduction for data visualization. With appropriate dimensionality and sparsity constraints, autoencoders can learn data projections that are more interesting than PCA or other basic techniques.



We input a digit to the autoencoder. The encoder subnetwork creates a latent representation of the digit. This latent representation is substantially smaller (in terms of dimensionality) than the input. The decoder subnetwork then reconstructs the original digit from the latent representation.

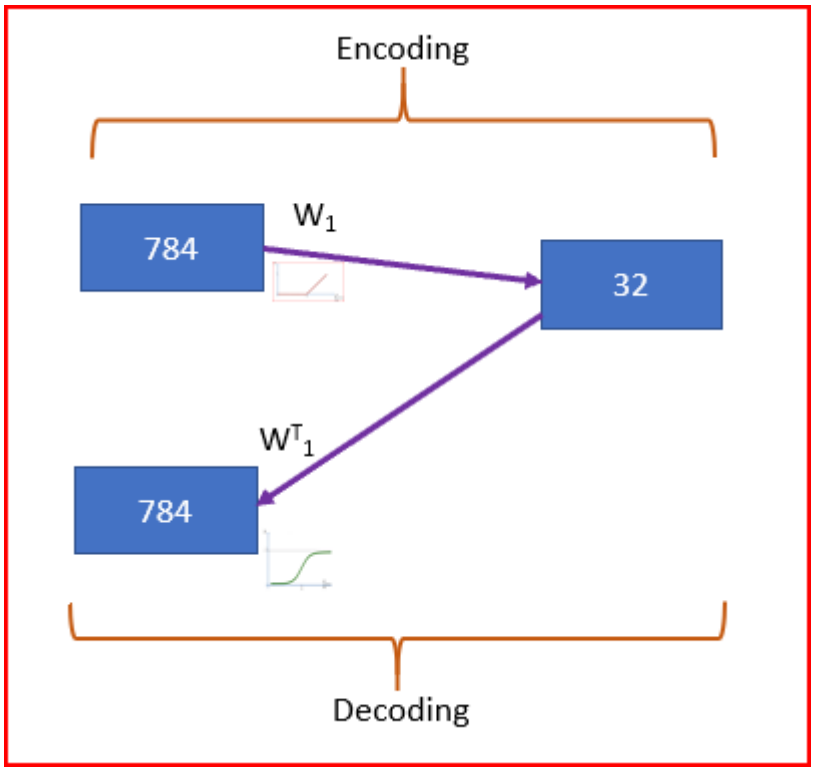
```
#Import libraries for neural network, numpy for matrix operations, and matplotlib f
#Keras library is a high level API for deep leanring and neural networks

from keras.datasets import mnist
from keras.layers import Input, Dense
from keras.models import Model
from keras import backend as b
from keras import initializers,regularizers
from tensorflow.keras.optimizers import SGD,Adam
import numpy as np
import matplotlib.pyplot as plt
```

Loading the MNIST dataset images and not their labels. We want to reconstruct the images as output of the autoencoder and hence we do not need labels. Creating a training set and test set and normalizing the data for better training.

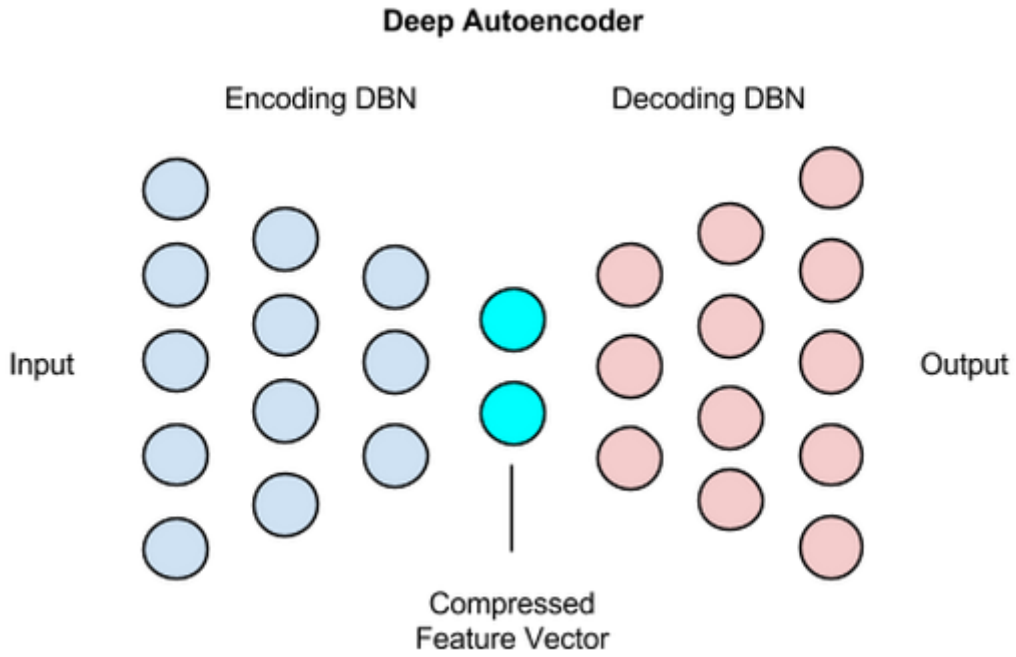
```
(X_train, _), (X_test, _) = mnist.load_data()

X_train = X_train.astype('float32')/255.
X_test = X_test.astype('float32')/255.

X_train = X_train.reshape(len(X_train), np.prod(X_train.shape[1:]))
X_test = X_test.reshape(len(X_test), np.prod(X_test.shape[1:]))
print(X_train.shape)
print(X_test.shape)
```

```
(60000, 784)
(10000, 784)
```

Our input image is a 2D matrix of size 28x28. When we represented it as a vector it has a dimension of 784. We will encode it to 32 and the decode it back to the original dimension of 784



Encoded image will have a compression rate of 784/32=24.5




```
b.clear_session()
#We need to take the input image of dimension 784 and convert it to keras tensors.
input_img= Input(shape=(784,))
#We now create the encoder and the decoder based on the figure above.
#Input image will be Encoded to 32 units and the activation will be relu and the ir
```

```
encoded = Dense(units=32, activation='relu')(input_img)
#Decoder will have 784 units as it needs to reconstruct the input image back to its
#As decoded is the output layer of the autoencoder we will use sigmoid activation f
#Sigmoid allows us to model the output as the presence of a white pixel or not.
decoded = Dense(units=784, activation='sigmoid')(encoded)
#We now create the autoencoder with input as the input image and output as the deco
autoencoder=Model(input_img, decoded)
autoencoder.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 input_1 (InputLayer)         [(None, 784)]             0

 dense (Dense)                (None, 32)                25120

 dense_1 (Dense)              (None, 784)               25872

=================================================================
Total params: 50,992
Trainable params: 50,992
Non-trainable params: 0
_________________________________________________________________
```

We train the autoencoder end to end, which has both the encoder and decoder parts.

```
#We can also extract the encoder which takes input as input images and the output i
encoder = Model(input_img, encoded)
#let see the structure of the encoded model
encoder.summary()
#We now compile the autoencoder model with adadelta optimizer.
#As pixels have a value of 0 or 1 we use binary_crossentropy as the loss function a
#i.e., the number of pixels that are correctly predicted. Alternativelly, we can th
autoencoder.compile(optimizer='adam', loss='binary_crossentropy', metrics=['binary_
```

```
Model: "model_1"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 input_1 (InputLayer)         [(None, 784)]             0

 dense (Dense)                (None, 32)                25120

=================================================================
Total params: 25,120
Trainable params: 25,120
Non-trainable params: 0
_________________________________________________________________
```

```
#We now train the autoencoder using the training data with 20 epochs and batch size
autoencoder.fit(X_train, X_train,
                epochs=20,
                batch_size=256,
```

```
            shuffle=True,
            validation_data=(X_test, X_test)
            )
```

```
Epoch 1/20
235/235 [==============================] - 5s 19ms/step - loss: 0.2779 - bina
Epoch 2/20
235/235 [==============================] - 3s 14ms/step - loss: 0.1710 - bina
Epoch 3/20
235/235 [==============================] - 3s 13ms/step - loss: 0.1443 - bina
Epoch 4/20
235/235 [==============================] - 3s 14ms/step - loss: 0.1284 - bina
Epoch 5/20
235/235 [==============================] - 3s 14ms/step - loss: 0.1179 - bina
Epoch 6/20
235/235 [==============================] - 3s 13ms/step - loss: 0.1105 - bina
Epoch 7/20
235/235 [==============================] - 3s 13ms/step - loss: 0.1054 - bina
Epoch 8/20
235/235 [==============================] - 3s 14ms/step - loss: 0.1017 - bina
Epoch 9/20
235/235 [==============================] - 3s 14ms/step - loss: 0.0991 - bina
Epoch 10/20
235/235 [==============================] - 3s 13ms/step - loss: 0.0973 - bina
Epoch 11/20
235/235 [==============================] - 3s 13ms/step - loss: 0.0961 - bina
Epoch 12/20
235/235 [==============================] - 3s 13ms/step - loss: 0.0953 - bina
Epoch 13/20
235/235 [==============================] - 3s 13ms/step - loss: 0.0947 - bina
Epoch 14/20
235/235 [==============================] - 3s 13ms/step - loss: 0.0944 - bina
Epoch 15/20
235/235 [==============================] - 3s 13ms/step - loss: 0.0941 - bina
Epoch 16/20
235/235 [==============================] - 3s 13ms/step - loss: 0.0939 - bina
Epoch 17/20
235/235 [==============================] - 3s 13ms/step - loss: 0.0937 - bina
Epoch 18/20
235/235 [==============================] - 3s 13ms/step - loss: 0.0936 - bina
Epoch 19/20
235/235 [==============================] - 3s 13ms/step - loss: 0.0935 - bina
Epoch 20/20
235/235 [==============================] - 3s 13ms/step - loss: 0.0934 - bina
<keras.callbacks.History at 0x7f401c689cd0>
```

During training it is important to look at the loss. It should decrease. Similarly we should observe that the accuracy has increased. Finally, you should also keep an eye at validation losses and accuracies that they follow the same trends to identify overfitting.

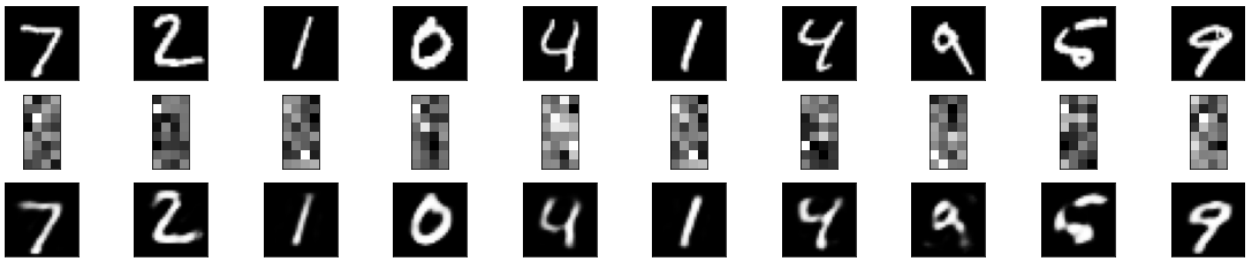
Visualizing the Encoding

Predicting the test set. We want to view the encoded images as well as the reconstructed images so we fit the test data on both autoencoder as well as encoder

```
encoded_imgs = encoder.predict(X_test)
predicted = autoencoder.predict(X_test)
```

Let's plot the original input, encoded images and the reconstructed images using matplotlib

```
plt.figure(figsize=(40, 4))
for i in range(10):
    # display original
    ax = plt.subplot(3, 20, i + 1)
    plt.imshow(X_test[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # display encoded image
    ax = plt.subplot(3, 20, i + 1 + 20)
    plt.imshow(encoded_imgs[i].reshape(8,4))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    # display reconstruction
    ax = plt.subplot(3, 20, 2*20 +i+ 1)
    plt.imshow(predicted[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

plt.show()
```



Notice that the outputs are similar with the input. Furthermore, encodings of similar numbers seem to share some patterns. The outputs are fuzzy though. We could try a deeper network or train for longer.

The representations learned by autoencoders can be used for downstream tasks such as classification or regression. Furthermore, autoencoder variants provide ways to model and learn distributions so that they can be used as generative models (e.g., Variational autoencoders (VAEs)).