# AutoEncoder on Dimension Reduction

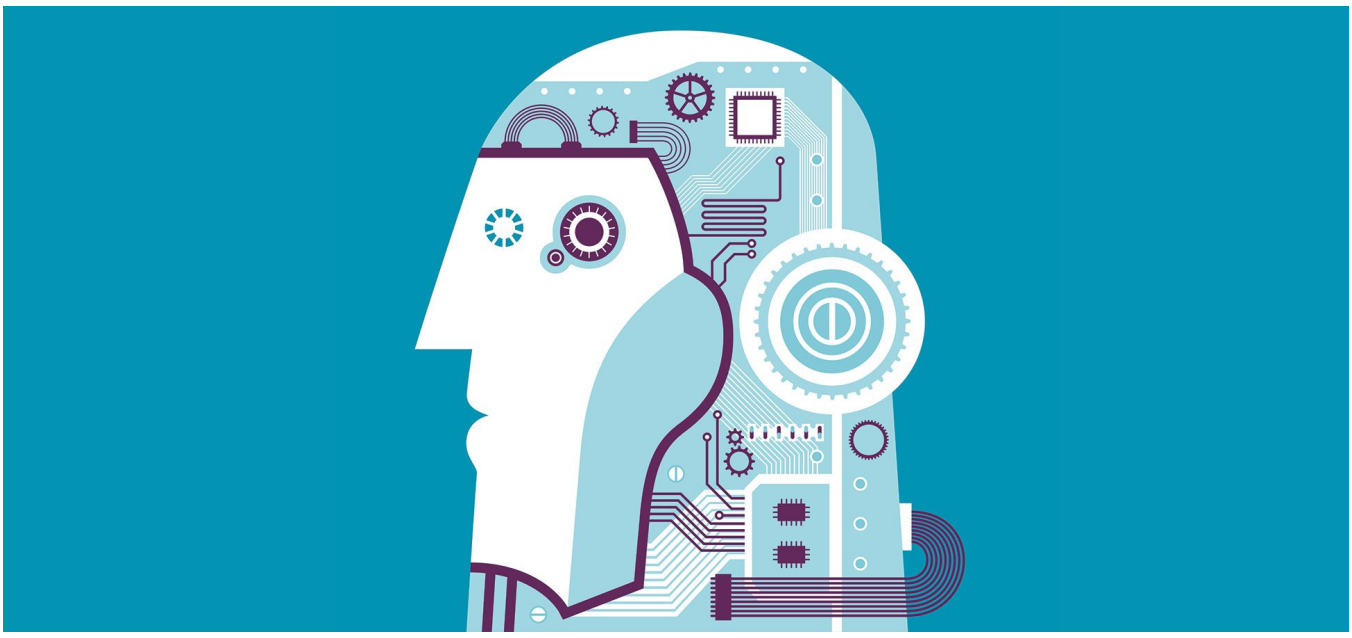
## An Example of Applying AutoEncoder on Tabular Data

Jeremy Zhang
Oct 11 · 4 min read ★

A general situation happens during feature engineering, especially in some competitions, is that one tries exhaustively all sorts of combinations of features and ends up with too many features that is hard to select from. In order to avoid overfitting, one can either select a subset of features with highest importance or apply some dimension reduction techniques.

I vaguely remember that there was one Kaggle competition in which the first prize solution was using autoencoder in dimension reduction. So, in this post, let's talk a bit on autoencoder and how to apply it on general tabular data. The structure follows:

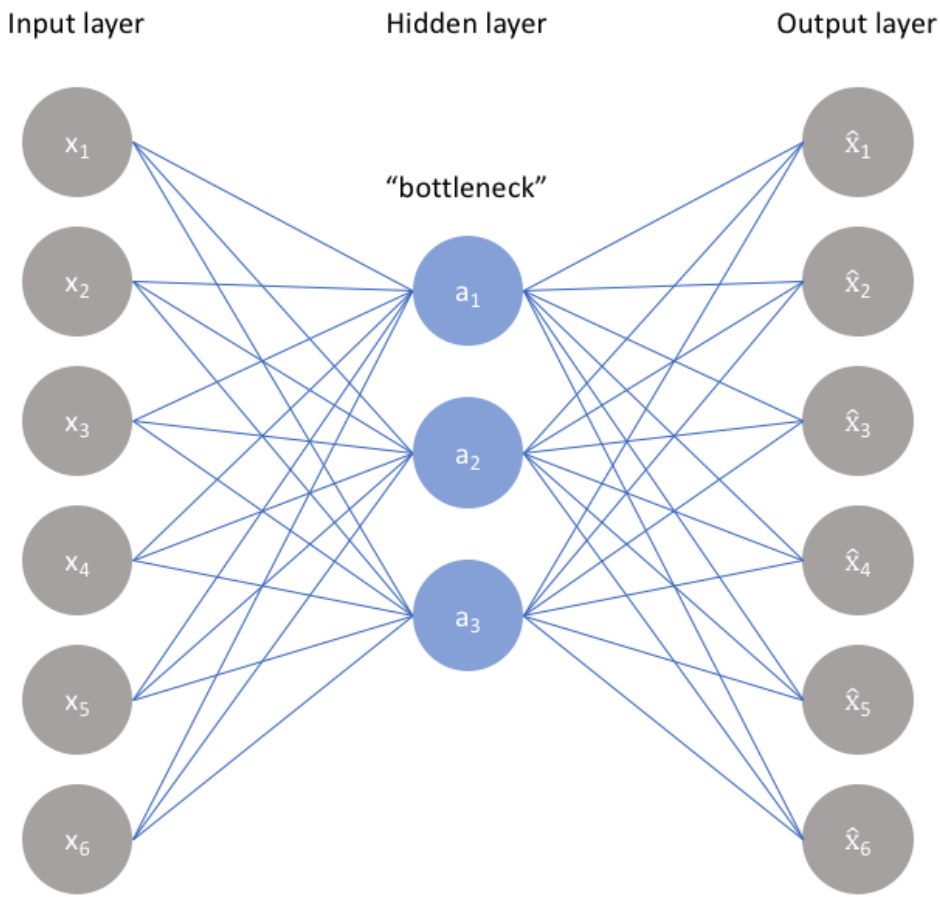
1. Walk through a quick example to understand the concept of autoencoder
2. Apply autoencoder on competition data



## A Great Example of Autoencoder

There is a great explanation of autoencoder here. Let's start with the most basic example from there as an illustration of how autoencoder works and then apply it to a general use case in competition data.

The most fundamental autoencoder follow the structure:



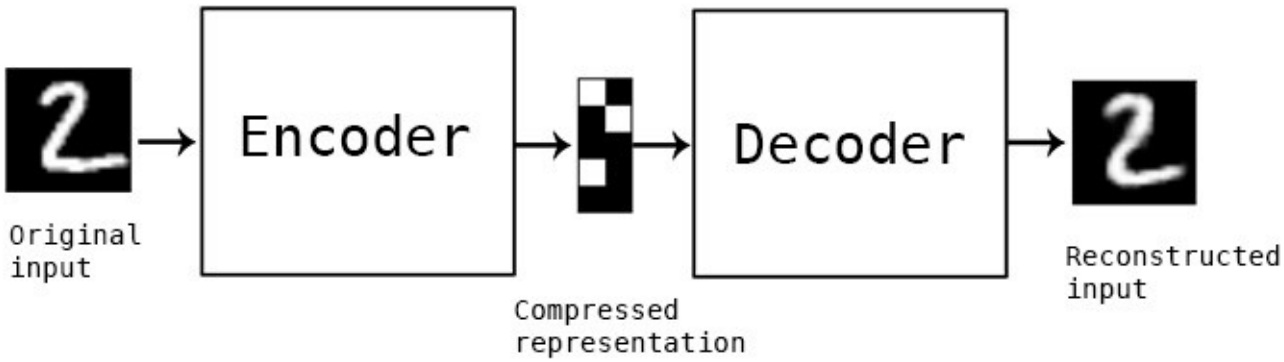

Notice that the input and output has same number of dimensions(in fact, the input is used as 'label' for the output), and the hidden layer has less dimensions, thus it contains compressed informations of input layer, which is why it acts as a dimension reduction for the original input. From the hidden layer, the neural network is able to decode the information to it original dimensions. From `input_layer -> hidden_layer` is called encoding, and `hidden_layer -> output_layer` is called decoding.

Autoencoder, in a sense, is unsupervised learning, as it does not require external labels. The encoding and decoding process all happen within the data set.

> *Autoencoders are learned automatically from data examples, which is a useful property: it means that it is easy to train specialised 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.*

Let's get through an example to understand the mechanism of autoencoder.




We will be using the famous MNIST data to see how the images are able to be compressed and recovered.

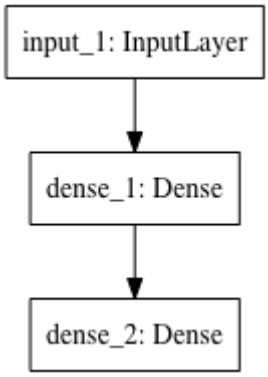
## Build the Network

The example is from Keras Blog.

```
# this is the size of our encoded representations
encoding_dim = 32  # 32 floats -> compression of factor 24.5, assuming the input is 784

# this is our input placeholder
input_img = Input(shape=(784,))

# "encoded" is the encoded representation of the input
encoded = Dense(encoding_dim, activation='relu')(input_img)

# "decoded" is the lossy reconstruction of the input
decoded = Dense(784, activation='sigmoid')(encoded)

# this model maps an input to its reconstruction
autoencoder = Model(input_img, decoded)

# intermediate result
# this model maps an input to its encoded representation
encoder = Model(input_img, encoded)

autoencoder.compile(optimizer='adadelta', loss='binary_crossentropy')
```

acder.py hosted with ❤ by GitHub view raw

Our network is straight forward: the input image with size of 784 will go through a dense layer and be encoded into size of 32, from which the decode layer will recover to the original dimension with size of 784. The structure is as simple as:




We also record the `encoded` in order to get the intermediate result of our autoencoder.

## Prepare Dataset and 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:])))

autoencoder.fit(x_train, x_train,
                epochs=50,
                batch_size=256,
                shuffle=True,
                validation_data=(x_test, x_test))
```

acder2.py hosted with ❤ by GitHub view raw

The data is normalised in to 0 and 1, and passed into our autoencoder. **Notice that both the input and output is `x_train` , the idea is that we hope our encoded layer to be juicy enough to recover as much information as possible.**

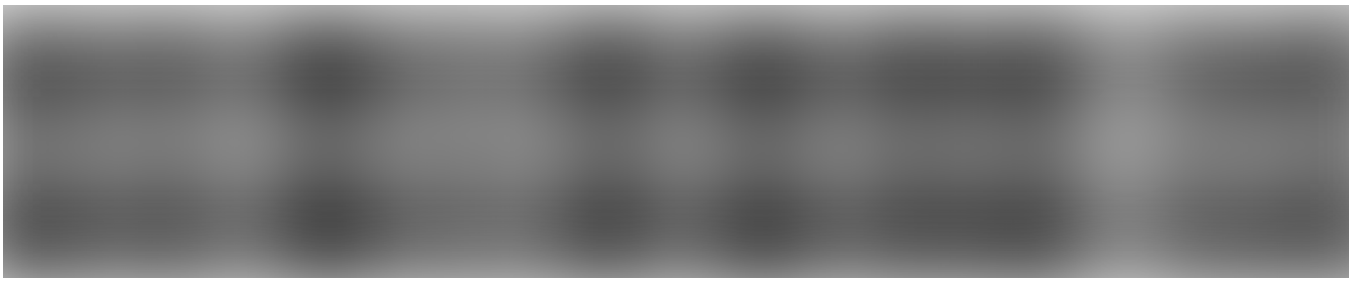
## Visualise the Result

```
reconst_test = autoencoder.predict(x_test)
encode_test = encoder.predict(x_test)

n = 10
row = 2
```

```
6
7  plt.figure(figsize=(20, 4))
8  for i in range(n):
9      # display original
10     ax = plt.subplot(row, n, i + 1)
11     plt.imshow(x_test[i].reshape(28, 28))
12     plt.gray()
13     ax.get_xaxis().set_visible(False)
14     ax.get_yaxis().set_visible(False)
15
16     # display reconstruction
17     ax = plt.subplot(row, n, i + 1 + n)
18     plt.imshow(reconst_test[i].reshape(28, 28))
19     plt.gray()
20     ax.get_xaxis().set_visible(False)
21     ax.get_yaxis().set_visible(False)
22
23 plt.show()
```

acder3.py hosted with ❤️ by GitHub view raw

The original image is compared to the image recovered from our encoded layer.



See that our hidden layer of dimension 32 is able to recover an image of dimension 784 and able to capture the information quite well.

Now let's apply this dimension reduction technique on a competition data set.

## Autoencoder on Tabular Data

In the post here we applied some general feature engineering technique and generate more than 170 features on the data set. Let's try to reduce its dimension.

We split the data in training and validating, and the training data looks like this with 171 columns:

Now we use the same technique and reduce the dimension to 40,

```
# reduce to 40 features
encoding_dim = 40

input_df = Input(shape=(171,))
encoded = Dense(encoding_dim, activation='relu')(input_df)
decoded = Dense(171, activation='sigmoid')(encoded)

# encoder
autoencoder = Model(input_df, decoded)

# intermediate result
encoder = Model(input_df, encoded)

autoencoder.compile(optimizer='adadelta', loss='mean_squared_error')

autoencoder.fit(X_train, X_train,
                epochs=150,
                batch_size=256,
                shuffle=True,
                validation_data=(X_val, X_val))
```

acder4.py hosted with ❤ by GitHub view raw

The `encoder` will be used later for dimension reduction.

Now let's apply prediction on reduced dimensions,

```
encoded_X_train = encoder.predict(X_train)
encoded_X_val = encoder.predict(X_val)

lgb_train = lgb.Dataset(encoded_X_train, y_train, free_raw_data=False)
lgb_test = lgb.Dataset(encoded_X_val, y_val, reference=lgb_train,  free_raw_data=False)

params = {"objective": "binary",
          "metric": {"binary_logloss"},
          "boosting_type": "gbdt",
          "learning_rate": 0.01
```

```
            "learning_rate" : 0.01,
            "max_depth": 4,
            "num_leaves": 16,
            "min_data_in_leaf": 30,
            "min_child_samples": 10,
            }

print('start training...')

model = lgb.train(params,
                  lgb_train,
                  num_boost_round=200,
                  valid_sets=lgb_test,
                  early_stopping_rounds=50,
                  learning_rates=lambda iter: 0.7 * (0.999 ** iter))
```

acder5.py hosted with ❤️ by GitHub view raw

For comparison, we still applied lightgbm for prediction and got a result of 0.595 with only 40 features comparing previously 0.57 with 171 features. Although the reduced dimension model does not outperform the previous one, I believe we see the advantages of autoencoder. (code)

## Conclusion

The autoencoder introduced here is the most basic one, based on which, one can extend to deep autoencoder and denoising autoencoder, etc.

Another advantage of autoencoder in competition is that one can build the autoencoder based on both training and testing data, which means the encoded layer would contain information from testing data as well! The post I read of the first prize solution on Kaggle competition used denoising autoencoder by adding some noise to the original features in order to make his network more robust, meanwhile, he also used testing data into his model, which, I believe, also contribute to his winning.

**Reference**:

[1] https://blog.keras.io/building-autoencoders-in-keras.html

Machine Learning Autoencoder Neural Networks Towards Data Science