AUTOENCODER EXPERIMENT REPORT

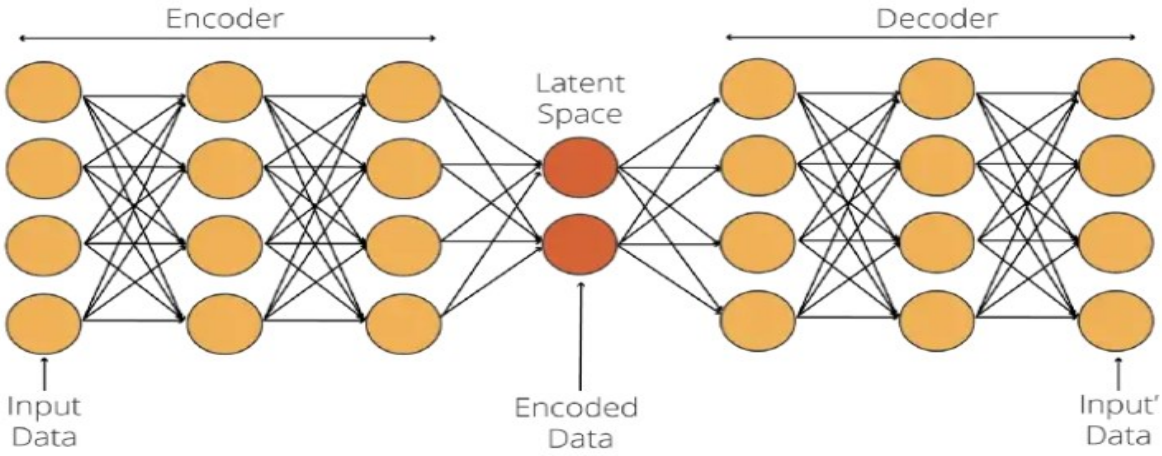
GANESHREDDY ANNAPAREDDY

50442295

ganeshre@buffalo.edu

1 AUTO ENCODER

Data compression and encoding are effectively learned by an unsupervised neural network. The algorithm then learns to recreate the data in a way that as nearly resembles the original input as possible from the reduced encoded representation. The algorithm's goal is to train the network to capture the most significant elements of the input image in order to learn lower-dimensional data, often for dimensionality reduction. A data compression approach called "autoencoding" uses compression and decompression algorithms that are: 1) data-specific, 2) lossy, and 3) automatically learnt from examples as opposed to being designed by humans. Additionally, neural networks are employed to accomplish the compression and decompression operations in practically all settings where the word "autoencoder" is used.



"The above given figure shows us an example on how the auto encoder works"

The Auto encoder consists of usually 3 main parts:

I. Encoder: A component that shrinks the input data by several orders of magnitude before converting it to an encoded representation.

II. Bottleneck: The compressed form of the input data is stored in a hidden layer. This is the network's most crucial component.

III. Decoder: This layer is in charge of decoding the data and learning how to recreate it. The result is then contrasted with the actual situation.

An autoencoder requires three components: an encoding function, a decoding function, and a distance function that measures the amount of information lost between the compressed and uncompressed versions of your material (i.e. a "loss" function). Stochastic Gradient Descent will be used to optimize the parameters of the encoder and decoder in order to reduce the reconstruction loss. The encoder and decoder will be chosen to be parametric functions (usually neural networks) and to be differentiable with respect to the distance function. It's easy! And to begin utilizing autoencoders in practice, you don't even need to comprehend any of these phrases.

Do they have skill in data compression?

Not really, usually. For example, it is exceedingly challenging to train an autoencoder for photo compression that performs better than a simple method like JPEG, and often the only way it is possible is by limiting oneself to a very specific sort of picture (e.g. one for which JPEG does not do a good job). Since they can only be used on data that is identical to the data they were trained on, autoencoders are often not viable for real-world data compression issues since doing so needs a lot of training data. But who knows how technology in the future could alter this.

What are the benefits of autoencoders?

They are seldom ever applied in real world situations. They temporarily discovered a use for greedy layer-wise pretraining for deep convolutional neural networks in 2012 , but this rapidly lost favor as researchers began to realize that improved random weight initialization strategies were enough for training deep networks from the ground up. Batch normalization began enabling increasingly deeper networks in 2014, while residual learning allowed us to train arbitrary deep networks from scratch starting in late 2015. Currently, data denoising (which we highlight later in this piece) and dimensionality reduction for data visualization are two fascinating practical uses of autoencoders. Autoencoders are able to learn data projections that are more interesting than PCA or other fundamental methods when given the right dimensionality and sparsity restrictions. The best technique currently available for 2D visualization, t-SNE (pronounced "tee-snee"), often requires low-dimensional data. So utilizing an autoencoder to compress your data into a low-dimensional space, like 32 dimensions, and then using t-SNE to map the compressed data to a 2D plane is a suitable technique for showing similarity connections in high-dimensional data. Keep in mind that Kyle McDonald created an excellent parametric t-SNE implementation in Keras, which is accessible on Github. In addition, Scikit-Learn includes a straightforward and useful implementation.

Autoencoders come in 5 different varieties:

- *Undercomplete autoencoder:* It's the most basic kind of autoencoder. It makes an effort to reduce the number of nodes in the network's hidden layer, hence reducing the amount of information that may pass through the network. The model is punished in order for it to pick up on the crucial characteristics. Reconstruction loss refers to the loss that is used to test the autoencoder's performance in reconstructing the input picture from the input. The L1 loss is used to represent the term even though the reconstruction loss might be anything based on the input and output. By applying the formula:

$$L = |x - \hat{x}|$$

Where $\hat{x}$ represents the predicted output and x represents the ground truth.

- *Sparse Autoencoders:* It provides a different way to introduce a bottleneck in the flow of information without reducing the number of nodes at our hidden levels. Instead, create a loss function that penalizes hidden layer activations in the model. There are two main techniques to include the sparsity regularize term in the loss function. KL Divergence and the loss function are provided by:

$$L = |x - \hat{x}| + \lambda \sum_i |a_i^{(h)}|$$

Where *h* represents the hidden layer, *i* represents the image in the minibatch, and *a* represents the activation.

$$L = |x - \hat{x}| + \sum_j KL(\rho||\hat{\rho}_j)$$

Where $\hat{\rho}_j = \frac{1}{m}\sum_i [a_i^{(h)}(x)]$ and *j* denotes the specific neuron for layer *h* and a collection of *m* samples is being made here, each denoted as *x*.

- *Contractive Autoencoder:* It operates under the premise that inputs with identical characteristics ought to have equivalent encodings and latent space representations. It implies that for little input alterations, the latent space shouldn't vary significantly. The following formula provides the loss function:

$$L = |x - \hat{x}| + \lambda \sum_i \|\nabla_x a_i^{(h)}(x)\|^2$$

Where *h* is the hidden layer for which the gradient is calculated and represented with respect to the input x as $\nabla_x a_i^{(h)}(x)$

- *Denoising Autoencoder:* We input a noisy version of the image—noise that has been generated by digital manipulations—to denoising autoencoders. The encoder architecture is fed the noisy picture, and the output is compared with the source image. Nonlinear dimensionality reduction is used by denoising autoencoders. These networks often employ L2 or L1 loss as the loss function.
- *Variational encoder:* This kind of auto-encoder utilizes the Stochastic Gradient Variational Bayes estimator during training and makes significant assumptions regarding the distribution of latent variables. It works to learn an approximate representation of the conditional attribute under the assumption that the data is produced by a directed graphical model.

2 EXPERIMENT

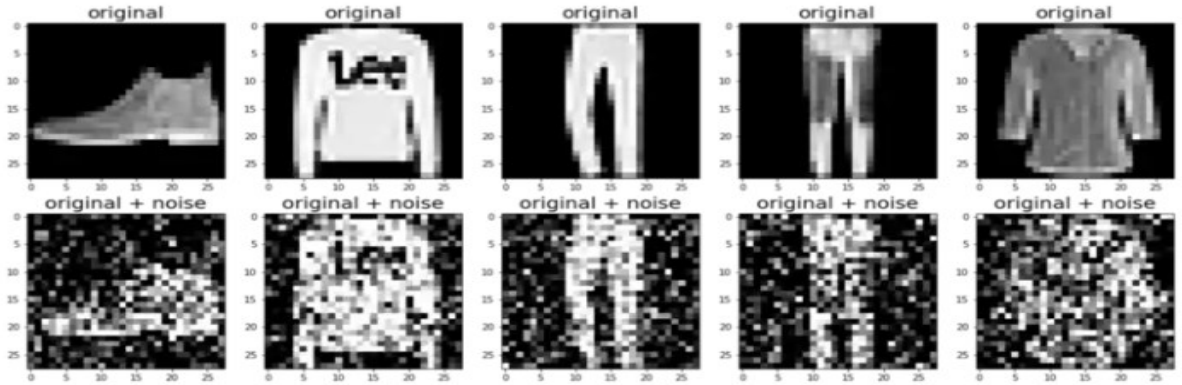
Dataset :

I have taken the dataset from Kaggle, Zalando is a German-based European e-commerce firm that created and maintains Fashion-MNIST. A training batch of 60,000 photos and a test set of 10,000 images make up the fashion MNIST. Each illustration is a 28 by 28 grayscale graphic paired with a label drawn from one of ten classes. Fashion MNIST was created as an alternative dataset to the MNIST dataset, which combines handwritten numbers plus photographs of apparel. The first output we got for the shows the sample for the first 50 sets. The picture below shows how it looks,

Let’s process the data now. Minmax normalization, which restricts the value range between 0 and 1, must be applied to our picture data for computational effectiveness and model dependability. Since our data is in RGB format, where 0 is the lowest value and 255 is the highest, I can do the Minmax normalization process.

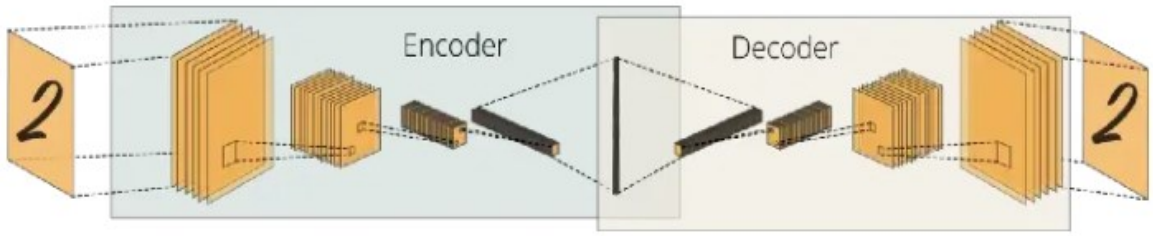
Additionally, since the datasets are now in the shape of (60000, 28, 28), I must resize our NumPy array (10000, 28, 28). It will only take a single number to add a fourth dimension (for example, from (60000, 28, 28) to (60000, 28, 28, 1)). The fact that our data is in grayscale format, with a single value representing colour information spanning from white to black, is essentially demonstrated by the fourth dimension. I would require three values in our fourth dimension if I had colourful pictures. But since I work with grayscale photos, all I really need is a fourth dimension with a single value. After looking at the shape of the NumPy arrays I got the output as (60000, 28, 28, 1) and (10000, 28, 28, 1).

Now I am going to add the noise to images. I Kept in mind that I want to create a model that can remove noise from photos. I will leverage current picture data and combine them with random noise to accomplish this. After that, I will feed the original photographs as input and the result as noisy images. Our autoencoder will learn how to clean a noisy image and the difference between a clean and a noisy image. Consequently, let's make a noisy version of the Fashion MNIST dataset. For this work, I use the tf.random.normal function to add a randomly generated value to each array item. The random value is then multiplied by a noise factor, which you may adjust. Since I have created the noisy version of the dataset I am going to take a look at it to see how it looks.



Building the model:

A third approach for building models in TensorFlow exists in addition to the Sequential API and Functional API: model subclassing. We are allowed to implement everything from scratch when using model subclassing. Model subclassing allows us to design our own unique model and is completely customisable. We can create any kind of model using this technique, making it highly powerful. However, it necessitates a fundamental understanding of object-oriented programming. The tf.keras.Model object would be subclassed by our new class. Additionally, a number of variables and functions need to be declared. It's nothing to be concerned about, though. Also keep in mind that building a convolutional autoencoder—which would resemble the following—is more effective given that we are working with picture data:



"The above given figure shows us an example on how the auto encoder works"

To build a model I will have to do these tasks :

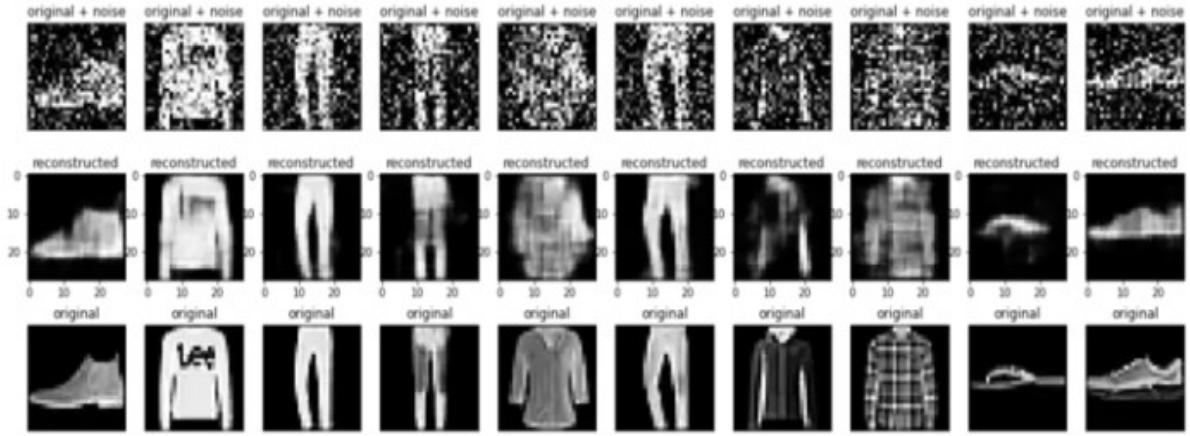
- I am creating a extending class kreas.Model object.
- Create a method called __init__ to declare two distinct Sequential API models. We must define layers inside them that would reverse one another. For the encoder model, there is one Conv2D layer, but for the decoder model, there is one Conv2DTranspose layer.
- Create a call function that instructs the model to use the initialized variables and init method to process the inputs.
- We would have to call the initialized encoder model which would take the images as the input.
- Also we will have to call the initialized decoder model which would take the output of the encoder model as the input.
- Then return the output of the decoder.

After all this I will have to configure the model. We'll use an Adam optimizer and Mean Squared Error for our model for this challenge. We can simply set up our autoencoder using the strong>compile/strong> function. Finally, after training for about a minute, we can run our model for 10 iterations by feeding it both noisy and clean images. Test datasets are also used by us for validation.

```
Epoch 1/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.0236 - val_loss: 0.0171
Epoch 2/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.0164 - val_loss: 0.0160
Epoch 3/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.0157 - val_loss: 0.0155
Epoch 4/10
1875/1875 [==============================] - 5s 2ms/step - loss: 0.0153 - val_loss: 0.0151
Epoch 5/10
```

Reducing Image Noise with our Trained Algorithm:

Our autoencoder can now begin cleaning noisy pictures because we have taught it. Because we define them under the Noise Reducer object, you should take note that we have access to both the encoder and decoder networks. Therefore, we will first encode our noisy test dataset (x test noisy) using an encoder. Then, in order to get the cleaned picture, we will use the encoded output from the encoder as input into the decoder, and let's plot the first 10 samples for a side-by-side comparison:



Noisy photos are in the first row, cleaned (reconstructed) images are in the second row, and original images are in the third row. We may observe how closely the cleaned photographs resemble the original pictures.

In conclusion I have built an autoencoder model which, for the first time, is able to clear very noisy photos (we used the test dataset). There are undoubtedly some distortions that were not restored, such as the slippers' missing bottom in the second image from the right. However, when you take into account how warped the noisy images were, we can state that our model does a good job of recovering the distorted images.