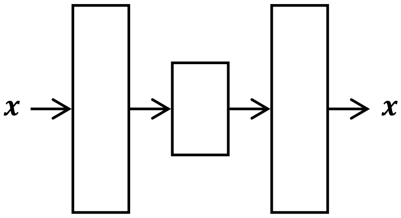
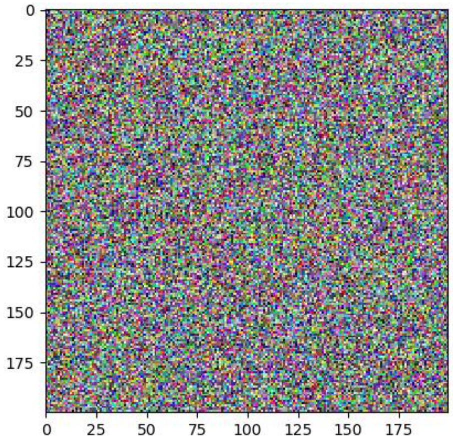
Topic 5: Autoencoders

A very interesting kind of hidden layer occurs when you train a neural network to output the exact same thing it is given as input. This is called an autoencoder. By making its hidden layer smaller than the input, the neural network has to learn to create a compressed representation of the input without losing any crucial information about it. This means that it ignores anything that can be inferred from other parts of the input in order to reduce redundancy, thus resulting in a representation consisting of just the essential features that describe the input.



So is there a lot of redundancy in typical input data that can be compressed by an autoencoder? We can get an intuition of the amount of redundancy in the input by comparing it to randomly generated inputs. If the two sets of data look similar then there is little redundancy, otherwise there is a lot of redundancy. Take photos for example. A photo is a structure consisting of pixels where each pixel is a vector of three colours: the intensity of red, the intensity of green, and the intensity of blue. These intensities are numbers between 0 and 255. If we randomly generate these pixel values to form a 200 by 200 image, the generated image would look something like this:



If you imagine a list of all possible images that can be made in a 200 by 200 grid of coloured pixels (there are $200 \times 200 \times 256^3 = 671{,}088{,}640{,}000$ different possibilities), the vast majority will look nothing like a photo, but will instead look more like static. An image autoencoder would take the tiny percentage of photo-like pixel combinations and represent them using a small vector.

1. Simple autoencoders

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/05_-_Hidden_representations/01_-_Simple_autoencoders.py

The part of the neural network leading to the middle layer is called the encoder, as it compresses the input, whilst the latter part is called the decoder, as it decompresses the compressed input. The compressed vector generated by the encoder is called a thought vector and has many interesting uses. For example, if the encoder part of the neural network is ignored, then we can make up our own thought vector to be fed to the decoder directly. Since the thought vector is a compressed dense representation of the input, then a random vector is likely to be a valid thought vector which will generate a valid output.

We can also do things like finding the average of several thought vectors in order to generate a thought vector that, when decoded, results in an average output of all the inputs that produced the original thought vectors. You can find examples of what to do with thought vectors of images of faces as well as a hypothesis of what is being encoded in the thought vector here.

In the code, we're going to make an autoencoder that performs binary encoding and decoding. Given a one-hot vector consisting of four bits (so there are four possible different inputs), the autoencoder needs to compress it into a two bit representation (using sigmoids to force the representation to be bits). What will the two bit thought vectors of the four bit one-hot vectors look like?

2. Variational autoencoders

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/05_-_Hidden_representations/02_-_Variational_autoencoders.py

The previous autoencoder is great but if we really wanted to generate valid outputs from random thought vectors then we need to consider the distribution of the thought vectors. Are random standard normal vectors likely to cover all possible thought vectors? Probably not. But we can force the thought vector to be close to the normal distribution. In fact, we can force the autoencoder to work on actual random standard normal thought vectors.

This kind of autoencoder is called a variational autoencoder. It works by having the encoder generate a mean and standard deviation for each element in the thought vector. These are then used to set a random normal generator which will generator a thought vector. Left to its own devices, such an autoencoder will learn to set the standard deviations to be close to zero so that the thought vectors will be deterministic as if it is a simple autoencoder. But we want the standard deviation to be close to 1 and the mean to be close to 0 in order to be able to use random standard normal vectors as thought vectors when generating outputs after training.

To do this we modify the loss function so that the mean and standard deviation are optimised to be close to that of a standard normal distribution. This is done using KL-divergence which measures the similarity between two probability distributions, or in this case, the similarity between the distributions being generated by the encoder and the standard normal distribution. You can find a nice explanation of variational autoencoders in this blog post.