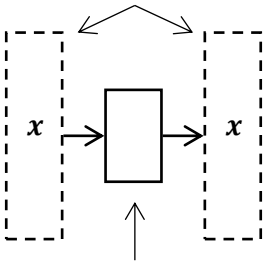
Topic 5: Autoencoders

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. Since the input values have to be reproduced as-is in the output activations, the neural network must learn to preserve all information about the input throughout all of its hidden layers (otherwise it wouldn't be able to reconstruct it). If one of the hidden layers is smaller than the input layer, the neural network has to learn to create a compressed representation of the input at that layer and then decompress it back at the output.

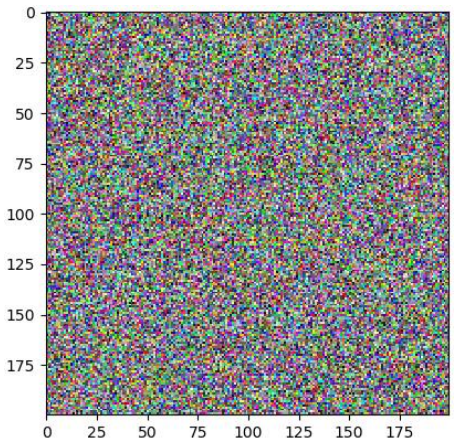
Output must be exactly the same as input



Hidden layer that is smaller (in terms of number of values) than the input

The compressed vector generated in the middle is called a thought vector. The part of the neural network generating the thought vector is called the encoder, as it compresses the input, whilst the part that generates the output from the thought vector is called the decoder, as it decompresses the compressed input. In order to be able to compress the thought vector in such a way that it can be decompressed back into the original input vector (lossless compression), it must leave out any information that can be correctly inferred (guessed) about the input in order to reduce redundancy. This results in a representation consisting of just the essential features that are necessary to describe the input. The more specific the training set of inputs is, such as a dataset consisting of only photos of human faces, the more can be assumed about the input without needing to explicitly encode it in the thought vector. For example, since the neural network was trained on only faces, it does not need to encode information about whether there is a nose in the photo or not as generally every input includes a nose. And furthermore, given that there is already information about the position of the nose, the position of the mouth and eyes can generally be inferred from there rather than also be encoded separately.

How much redundant information can you expect to find in a typical photo? We can get an intuition of how much wasted space there is in a type of input structure, such as English sentences in literature or photos of human faces, by generating random data of that structure. 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:



In the space of all possible pixel combinations 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, let alone a photo of a face, but will instead look like noise. This means that most of the pixel combinations that are possible in an image are never used. We say that images are a sparse representation. An autoencoder would take the tiny percentage of photo-like pixel combinations and represent them using a small thought vector such that the thought vector has a compact representation rather than a sparse one. Given a small enough thought vector, the representation becomes so compact that even if you make up a random thought vector, there exists a valid photo that can be encoded into it. In fact, if the encoder part of the neural network is ignored, then we can create our own 'artificial' random thought vector to be fed to the decoder directly in order to decode them into random, but sensible-looking, images that reflect the kind of images found in the training set used to train the autoencoder. You can learn more about all the interesting things you can do with thought vectors, as well as a hypothesis of what is being encoded in the thought vector, in this blog post.

Apart from their use in data generation, autoencoders are also useful for anomaly detection, that is, for recognising if a new image is like the images in the training set. If it breaks the assumptions made by the autoencoder about the inputs it has seen in the training set, it will not be able to reconstruct it correctly as it would not be trained to encode the unexpected information found in the new image. For this reason we can use the autoencoder in order to see if an image belongs to a set of images or not by measuring its reconstruction error. This is useful for single class classification tasks such as determining if a sentence is in English or if an image shows anything unexpected from the usual daily activities.

1. Simple autoencoders

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

Rather than work on images right away, we will be seeing examples of how to make an autoencoder that works on binary vectors. In the given script, we're going to make an autoencoder that performs binary encoding and decoding, that is, 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 using sigmoids for the thought vector might not be great as it might restrict the compression too much. We can allow the encoder to generate unbounded values but that would make it hard to generate valid thought vectors randomly as it would not be sparse. There is also the issue of how valid our assumption is that the random number generator we are using will cover all the thought vector space equally. Are normally distributed random vectors likely to cover all possible thought vectors? One solution for both these problems is to force the thought vector elements to be generated in a way that is similar to what you get with a random number generator. In fact, we can force the autoencoder to work on actual randomly generated thought vectors. This kind of autoencoder is called a variational autoencoder.

A normally distributed random variable requires two parameters in order to generate numbers: the mean and standard deviation of the generated numbers. The variational autoencoder has an encoder and a decoder but the encoder does not produce the thought vector directly. Instead it produces a mean and standard deviation for each element in the thought vector. These are then used to set a normally distributed random number generator which will generate a thought vector. You might be concerned about how a random number generator is differentiable, but this is solved by using this nice identity:

$$N(\mu, \sigma) = \sigma N(0,1) + \mu$$

where $N(\mu, \sigma)$ is the normally distributed random number generator with the mean μ and standard deviation σ and $N(0,1)$ is the standard normal distribution random number generator with mean 0 and standard deviation 1. The mean and standard deviation are produced by the encoder of the autoencoder whilst the standard normal distribution is produced by a normal random number generator. This removes the need to differentiate the random number generator as instead it can be treated as a constant whilst the mean and standard deviation (which do need to be differentiated) are simply added or multiplied to it. Since the standard deviation must be strictly positive (not negative or zero), we use the exponential function as an activation function (unusual but it works in this case).

Left to its own devices, such an autoencoder will learn to degenerate into a normal deterministic autoencoder that always produces the same thought vector given the same input. This can be achieved by setting the standard deviations to be close to zero (thus cancelling out the random part) whilst setting the mean to be the required value. But we want the encoder to produce random numbers which are distributed in a way that is similar to what we will be using when we're generating random thought vectors directly. In general we will be using a standard normal distribution to do so with a standard deviation of 1 and a mean of 0.

To force the encoder to generate means and standard deviations that are close to 1 and 0 we modify the loss function so that, apart from optimising the error of the output, we also optimise the means and standard deviations 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. The KL divergence between a normal distribution with mean μ and standard deviation σ and the standard normal distribution is measured as follows:

$$KL(\mu, \sigma) = \frac{1}{2}(\mu^2 + \sigma^2 - \ln(\sigma^2) + 1)$$

You can find a nice explanation of variational autoencoders in this blog post.