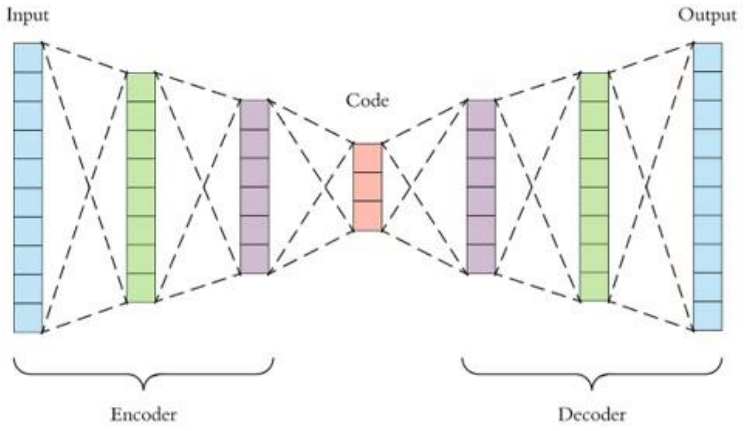
***Theory in overleaf***




**Autoencoder - MNIST**

In essence, autoencoders are mirrored neural networks.

The key idea here is to understand the process of mapping of a representation of something to its conceptual value. For example, when a human views a '9', the brain converts the light input signals and interprets that particular composition of information as a 9. This goes for anything, including the very letters you are reading here. But what if you were to draw a '9' on a piece of paper, use a camera to capture a photograph and ask a computer: what is this? A question we so fortunately deem easy is more complex, when considered on a deeper level. Every individual human has their own unique handwriting, and there exists some variation with each instance; yet we are all able to distinguish the '9' from literally anything else in the entire universe that is not a nine.

*"What is this?" one asks a computer.*

*"Ah ha! That is a 1 followed by a 0 and a 1 and a 1 and a 0 and a 1 and a 0!" the computer proudly pronounces. "I'm glad I can help identifying what you're looking at!"*

My point here is that, in order to digitise the conceptual value of the image, we must write a script that actually reads what it represents, explicitly not a bunch of 1s and 0s. How do we go about this? We train the computer by building an autoencoder model by gathering a large collection of images with one underlying identifiable theme of which we know what they actually are.

One such example of this is the MNIST dataset, consisting of 70000 hand-drawn digit images of 28 pixels by 28 pixels and their values are assigned by a human. These are grouped into 'batches', say 64 randomly selected images from the dataset per batch, to all be fed into the autoencoder once each time the script is ran. Next, stack the batches on top of one another in a Tensor, using the PyTorch library. We create 2 tensors: one for a 'training set' (60,000 of the 70,000) and another for the 'testing set' (10,000 of the 70,000). If the number of batches doesn't exactly divide the dataset, we divide it and then roof it for the tensor size – round it up to the next integer. Obviously, this means that last batch has less images than the others, so we **asjfbnasdfljg**.

To make it crystal clear, what we have for the <u>training set</u> is:

- 1 tensor
- 948 batches in the tensor
- 64 objects per batch

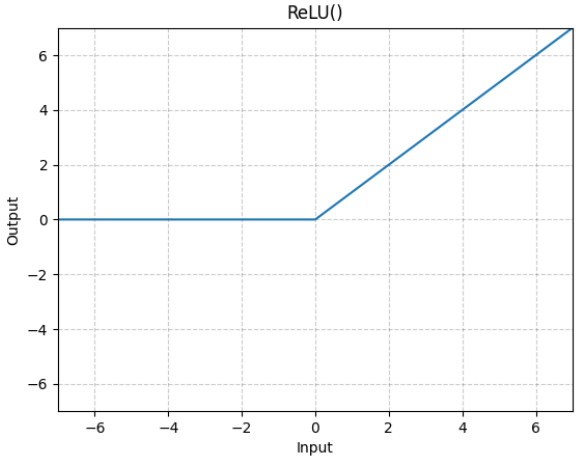
- (28*28) heightmap/gradient values per object
- 1 channel per set of values

*Note: The number of channels is just the number of different 'themes' we are looking for. Here, the MNIST dataset (and later the Perlin Noise generations) are on a scale of white to black, so only one channel is required. Compare that with a coloured image of red, green and blue, which would require 3 channels for the 3 different gradients.*

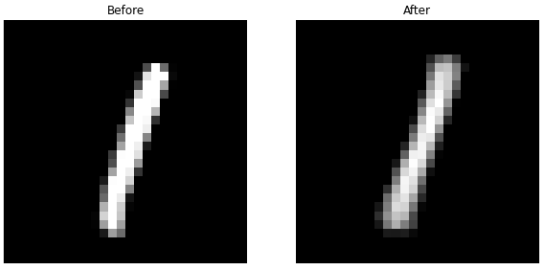
| Name | Type | Size | Value |
|---|---|---|---|
| batch_size | int | 1 | 64 |
| test_data | datasets.mnist.MNIST | 10000 | MNIST object of torchvision.datasets.mnist module |
| test_dataloader | utils.data.dataloader.DataLoader | 157 | DataLoader object of torch.utils.data.dataloader module |
| train_dataloader | utils.data.dataloader.DataLoader | 938 | DataLoader object of torch.utils.data.dataloader module |
| training_data | datasets.mnist.MNIST | 60000 | MNIST object of torchvision.datasets.mnist module |

```
Shape of X [N, C, H, W]: torch.Size([64, 1, 28, 28])
Shape of y: torch.Size([64]) torch.int64
```

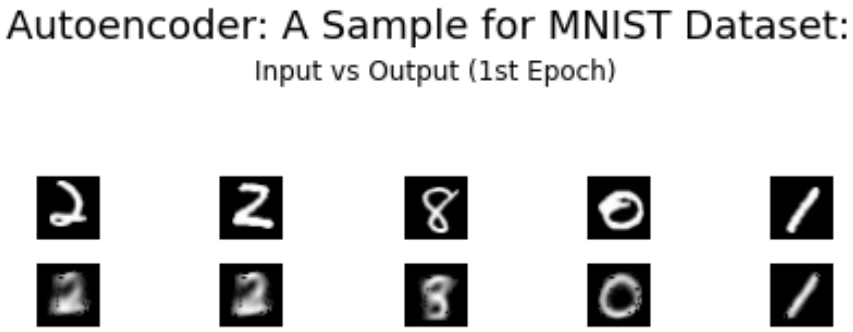
To create our model autoencoder on the MNIST dataset, we train it using the specified training data, for a number of epochs – the amount of times the batches are randomly selected and input into the autoencoder for training. But what exactly happens during training? We must define our model layers explicitly. In our project, we have chosen to use dense layers. These dense layers are sequenced as linear functions between one another, each looking at every single dataset and changing the number of neurons (datapoints) that represent each individual image. With each subsequent layer, additional time is taken but more data is preserved through the process. One alternative type of layer is a convolutional layer, whereby a filter is used (imagine a small radius of pixels neighbouring a specific centre pixel, changing the centre pixel as appropriate) and the weights are shared for all the pixels. Each layer is fully connected to another, i.e. every neuron has some weighted connection to each neuron in the next layer. It's is important to note that the input of a layer must match the size of the output from the previous. Between the layers, we have used a 'ReLU' (rectified linear unit) which is a non-linear function as shown in the figure, increasing the model's predictive capabilities; it basically just zeros the negative values shown below.



As explained previously, the model completes a forward pass per batch, calculating the loss (MSE) compared to original batch data. Each image is first flattened, simply converting the 28px by 28px into a single 784 string. We use the 'Adam' optimizer from the PyTorch library which adjusts the layer weights (and bias, as applicable) during back-propagation and set the 'learning rate' parameter to be 0.001; this essentially changes how much we update the weights with each batch passed through. Upon periodically printing the current loss at sequenced times in the console, we observe that the mean loss decreases as the number of batches and epochs are used in training, as expected. Once trained, the test dataset is used to see how good our model is. You can then show the model a new hand-written digit and it can predict what it represents very accurately.



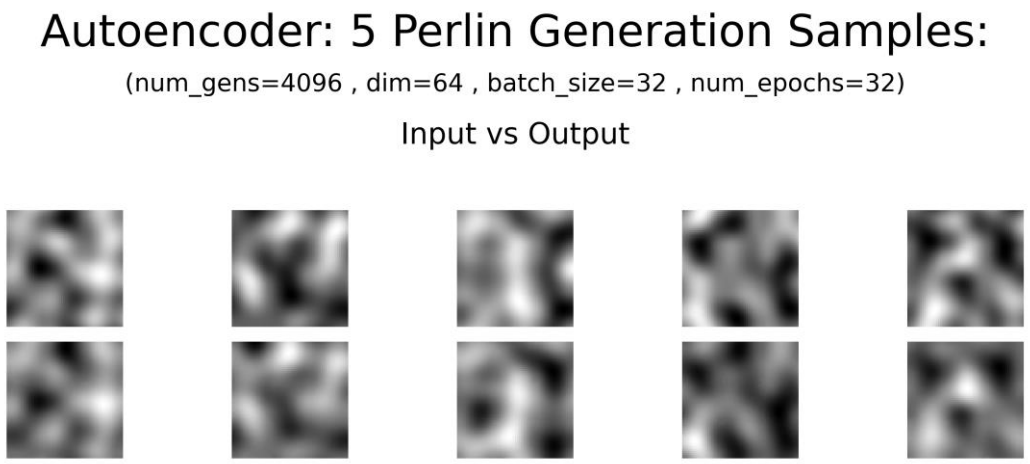

To analyse the inputs vs the outputs, we plot a figure of various corresponding subplots due to time and restricted processing power capabilities towards the end of the project, we have only done this for the first epoch, shown below. We have modified our code to display our passed images, rather than emit the corresponding numerical values. As you can see, even after one epoch, our weights have been adjusted to compress the data to just 2 neurons and extrapolate that back out to resemble the original image. Given more time and less restrictions, this would be repeated for many epochs and the results will significantly increase, as reported in the next section in which we train a model on Perlin Noise heightmaps.




*Note: Our exact script for the MNIST dataset is redacted since the Perlin Noise code that we have included is of superior quality.*

**Autoencoder - Perlin Noise**

Referring to our objectives set out at the beginning of the project, our intention is to be able to generate new realistic terrain from a small (2D) latent space. Since we have had issues with correctly accumulating the LiDAR scans after the PCA dimensionality reduction, we decided to utilise our Perlin Noise code to prove the effectiveness of the model. Attached is a figure proving just how efficient our autoencoder is.

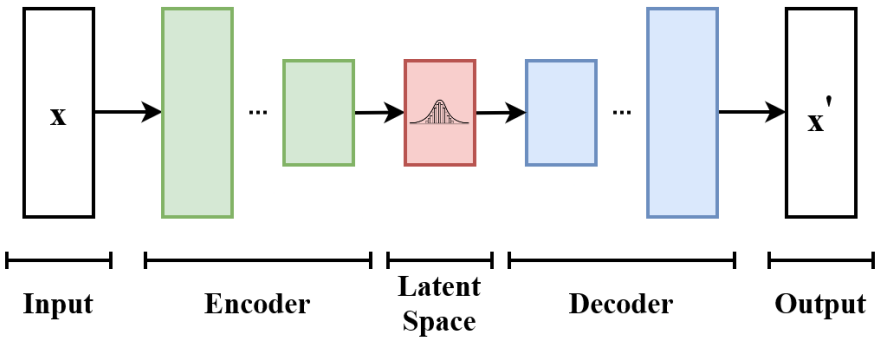



In our prime example above, we generated 4096 Perlin Noise images, each of dimension 64 pixels by 64 pixels; these were batched into 32s. The model was modified with an additional layer, since we have more pixels, and the bottleneck was changed to 8 neurons, rather than the 2 previously. Run this training for 32 epochs and the outputs tend increasingly towards the inputs with impeccable precision. With more time and processing power, we could increase these parameters for even better effect. In addition, colour could be returned to the images, as in the Perlin Noise section of the report, for a more aesthetically pleasing colourmap. We unfortunately didn't have time to use 2 central neurons to generate new plots, but with even a slight increase in time we could have modified the code for this purpose.

Attached is the exact Perlin Noise autoencoder script that produced the last figure.

**Autoencoder - Variational Autoencoders**

Before concluding the report, it's worth briefly discussing variational autoencoders (VAEs). This is the name for the type of network referred to when writing about mapping from the latent space, used in unsupervised learning, semi-supervised learning and supervised learning. We would make this modification to our model to be able to generate those new images, given extra time.




Basically, the model receives the input (i.e. our LiDAR scan/s) and compresses it into the latent space. It is here that a Gaussian distribution represents datapoints and we can sample for new generative terrain. The point is to maximise the likelihood of the data given a mixture of Normal distributions for it.

etc