**PyTorch Notes**

A computer screen shot of a computer program

Description automatically generated

A computer screen shot of a program

Description automatically generated

A screenshot of a computer program

Description automatically generated

**To-do**

* Definition explanation/refining
* Write all sourced information in own words.
* Create own graphics for all sourced images/examples.
* Clarify super().\_\_init\_\_()
* Explain optimisers (i.e. Adam [Adam Optimizer PyTorch With Examples - Python Guides](https://pythonguides.com/adam-optimizer-pytorch/) ; [Adam — PyTorch 2.1 documentation](https://pytorch.org/docs/stable/generated/torch.optim.Adam.html#torch.optim.Adam) )
* Expand on ReLU explanation (what is the point?)
* Explain if \_\_name\_\_ == “\_\_main\_\_” ( [python - What does if \_\_name\_\_ == "\_\_main\_\_": do? - Stack Overflow](https://stackoverflow.com/questions/419163/what-does-if-name-main-do) )
* Implement a train function for training the data
* Implement a device classification if gpu is available
* Explain the implementation of the MSE ( [PyTorch MSELoss - Detailed Guide - Python Guides](https://pythonguides.com/pytorch-mseloss/) )

**A dump of links utilised**

[Building a Neural Network with PyTorch in 15 Minutes | Coding Challenge - YouTube](https://www.youtube.com/watch?v=mozBidd58VQ&t=3s)

**Definitions**

* Channels – the number of ranges for the predictions (for example, we will use 1 channel as we range from white to black, as opposed to RGB which requires 3 channels)
* Feature map
* Flatten – convert our pixel matrix into a 1D array of length “image witdh \* image height”
* Forward – a call method in PyTorch
* Kernals - shape
* Filters
* Rectified Linear Unit (ReLU) – activations
* Logits – the [0,1] predicted probabilities for each class
* Loss function (Mean Squared Error [MSE])

**Dense Layer vs Convolutional Layer**

[machine learning - Dense Layer vs convolutional layer - when to use them and how - Data Science Stack Exchange](https://datascience.stackexchange.com/questions/85582/dense-layer-vs-convolutional-layer-when-to-use-them-and-how)

(*copied from information source*)

*The* ***Dense Layer*** *uses a linear operation meaning every output is formed by the function based on every input. In other words, we "force" every input to the function and let the NN learn its relation to the output. As a result, there appear n\*m connections (or weights) where n denotes the number of inputs and m denotes the number of outputs.*

*We use Dense Layer, by giving all the inputs, we give the "full responsibility" to learn. In other words, we say to a Dense Layer that "Here are my features (pixels maybe), I don't know the true relationship between them, please find it yourself".*

*The* ***Convolutional layer*** *uses a filter to operate the convolution operation which has a small size most of the time. An output of the convolution layers is formed by just a small size of inputs which depends on the filter's size and the weights are shared for all the pixels. That is, the output is constructed by using the same coefficients for all pixels by using the neighboring pixels as an input.*

*We say to a Convolutional Layer that "Here are my features (pixels maybe), I am sure that it is enough just to look the few pixels around the center pixel" and that relationship is preserved through all the pixels.*

**Torch.nn**

[torch.nn — PyTorch master documentation](https://pytorch.org/docs/master/nn.html)

**Sequential**

[torch.nn — PyTorch master documentation](https://pytorch.org/docs/master/nn.html#torch.nn.Sequential)

[python - How do I write a PyTorch sequential model? - Stack Overflow](https://stackoverflow.com/questions/46141690/how-do-i-write-a-pytorch-sequential-model)

----------------------------

**Building our neural network**

As always, we start by importing the relevant libraries required for the entire code. These are as follows:

import torch

from torch import nn

from torch.optim import Adam

from torch.utils.data import DataLoader

from torchvision import datasets

from torchvision.transforms import ToTensor

The reason we need:

* nn – the source of the neural network classes
* Adam – this is our optimiser of choice used to refine our weights and decrease rates of error during training (also dictates the learning rate with a default parameter of 0.001)
* DataLoader – allows us to load our MNIST data into the models
* datasets – the source of our MNIST data stored in the PyTorch package
* ToTensor – convert our data into the correct variable type for PyTorch to read

We start by defining our batch size which essentially collates numerous examples in our data to process through per forward pass, thereby, reducing the workload for our computational processing and increasing the speed of running the code.

Next, rather than manually downloading the MNIST data from online, we setup both our training dataset and testing dataset using the stored data provided with PyTorch; we subsequently load them into variable list, demonstrated below:

A screenshot of a computer

Description automatically generated

A quick test of these dataloader object shapes produces the following output:

A screenshot of a computer

Description automatically generated

(N= number of images per batch, C= number of channels in the layer, H= pix\_height, W =pix\_width)

Now, we aim to build our model that classifies the images by defining our network layers and how they stack. This is the most fundamental aspect of the network since it determines precisely how we build the encoder layers themselves, in order to generate the representation desired. In this project the aim is to have 2 central neurons in the bottleneck because, in future, we aim to create 2D plots for analysis (for both the MNIST data and the terrain heightmap data). (Note – for now, I have incorporated linear layers, not convoluted layers).

We “flatten” the dimensions to 2D (conveniently, the default argument of the command) to use with the PyTorch sequential code, processing the defined layers one after another. In other words, we convert the images (represented by a matrix of the image dimension, 28x28) to one dimension (a single array of 784 pixel values).

A screenshot of a computer

Description automatically generated

(*source* [Flatten — PyTorch 2.1 documentation](https://pytorch.org/docs/stable/generated/torch.nn.Flatten.html#torch.nn.Flatten) )

**What do each of the layers do?**

**nn.linear** - applies a linear transformation to the incoming data using the currently stores weights and biases ( [Linear — PyTorch 2.1 documentation](https://pytorch.org/docs/stable/generated/torch.nn.Linear.html) ).

A screenshot of a computer

Description automatically generated

**nn.ReLU** – replaces all the negative values with ‘0’s ( [ReLU — PyTorch 2.1 documentation](https://pytorch.org/docs/stable/generated/torch.nn.ReLU.html#torch.nn.ReLU) ; [How to Apply Rectified Linear Unit Function Element-Wise in PyTorch? - GeeksforGeeks](https://www.geeksforgeeks.org/how-to-apply-rectified-linear-unit-function-element-wise-in-pytorch/)). Example:

A black screen with white text

Description automatically generated

A screenshot of a computer

Description automatically generated

A graph with a line

Description automatically generated

One thing important to note here is that the first layer takes in the “ image height multiplied by width” dimensions (from our dataloader shape, i.e. 28\*28=784) and each subsequent linear layer takes in the previous shape. During encoding, we order a decreasing number of neurons, reaching the bottleneck layer consisting of just 2 neurons. Similarly, we use those same neuron numbers but in the opposite order:

**Input** **Bottleneck** **Output**

784 > 512 > 128 > 2 > 128 > 512 > 784

We define a “forward” function in the same class, taking in the current instance and the x data. In other words, this defines the algorithmic computation performed at each call.

The model combining the layers:

A computer screen shot of a program

Description automatically generated

Obviously, to train our network, we must iteratively optimise our weights and bias. In PyTorch, this is done by defining the loss function and an optimiser for the parameters stored in the model.

[The Essential Guide to Pytorch Loss Functions (v7labs.com)](https://www.v7labs.com/blog/pytorch-loss-functions)

A computer screen shot of a program

Description automatically generated

If name = main

Loss

Optimiser

Decoder

Clarify cpu/gpu

Epochs

Data to record: