# Image Classification with Convolutional Neural Networks using PyTorch

## Introduction and Overview

Image Classification is the process of classifying images into classes e.g., dog, cat, etc. Convolutional neural networks or CNNs are a type of neural network architecture specifically designed to process images. PyTorch is a machine learning open source framework and library that is most commonly used for deep learning and neural network development, much like TensorFlow. It was developed and released by Facebook’s AI Research (FAIR) laboratory in 2016. PyTorch is based on the Torch library, which is written in the lightweight scripting language Lua, and is designed for scientific computing with a focus on machine learning applications[[1]](#footnote-1).

This report will be an explanation of a image classification project on the Fashion MNIST dataset. Each section of the report will be in reference to a section of the project from data loading to displaying accuracy and loss over epochs.

## Dataset

The Fashion MNIST dataset was created by a research team at Zalando, an online lifestyle and fashion platform in 2017. It consists of 10 different classes of clothing: T-shirts, coats, dresses, sandals, sneakers, shirts, trousers, pullovers, bags, and boots. Each image in the dataset is made up of 28x28 pixels and has a label corresponding to its category. Additionally, it only has one channel as it is grayscale, instead of three as found in coloured images (RGB). This means each pixel will have a value between 0 and 255, 0 being black and 255 being white, values in between will be a different shade of grey becoming progressively lighter as the value increases. The dataset consists of 60,000 training images and 10,000 testing images.

The dataset was initially developed as an alternative to the classic MNIST (Modified National Institute of Standards and Technology) handwritten digit recognition dataset, which had been earlier seen as the benchmark for testing the performance of machine learning models. The Fashion MNIST dataset was planned to be a more difficult challenge for benchmarking computer vision applications as MNIST was seen as too simple and not a realistic benchmark as image-based applications became more advanced[[2]](#footnote-2).

## Transforms

Text

Description automatically generated

Figure 1 – illustrates the first block of code that imports libraries.

First, all the relevant torch libraries, data processing libraries (NumPy), and data visualisation libraries (matplotlib) must be imported.

Text

Description automatically generated

Figure 2 - the previous code displays the data augmentation transformations implemented on the dataset.

Data augmentation is the process of creating new data by applying transformations on to existing data. This is a method frequently used in computer vision and machine learning in order to boost model performance by expanding the pool of training data[[3]](#footnote-3).

These transformations can be made to the input data and output labels to produce new augmented data. Models can now learn to be more resilient and robust to variations in the input data, inputs in the real world will most likely not be in perfect conditions, variations in brightness or angle of input images may cause the model to fail or misclassify inputs e.g. there was an incident where Tesla’s autopilot failed to correctly classify an overturned truck on the highway causing the car to collide with the truck[[4]](#footnote-4). This could have been caused by a variety of factors such as the sun shining off the top of the truck, creating a difference in exposure which may have confused or blinded the computer vision system. Or it could also have been caused by insufficient training data, perhaps the dataset didn’t include angles of a truck’s underside or top which would be what the tesla would have seen if there was a flipped truck in its path. Nonetheless data augmentation and transformations are an important process to allow the model to learn more edge cases and generalise better to new, unseen data more effectively.

Here is a breakdown of each of the transformations:

1. **transforms.RandomHorizontalFlip()** – flips the image horizontally randomly, this can aid in increasing the size of the dataset while improving model robustness to variations in image orientation.
2. **transforms.RandomRotation(degrees=(-25, 25))** – rotates randomly the input image between -25 and 25 degrees, this can help the model better deal with recognising objects from different view points.
3. **transforms.ToTensor()** – will convert the input images into a PyTorch tensor which is the PyTorch equivalent of a NumPy array, this will allow the neural network to be able to read the input data.
4. **transforms.Normalize((0.5), (0.5))** – normalises the values of the input data, using the following algorithm on each channel: image = (image - mean) / std.[[5]](#footnote-5) The two 0.5s are parameters of the equation: mean and standard deviation respectively. This allows the neural network to learn better because it “helps to stabilize the gradient descent step, allowing us to use larger learning rates or help models converge faster for a given learning rate”[[6]](#footnote-6).
5. **transforms.Compose()** – finally, this function combines all the transformations into one pipeline that can be applied to all the input images. Therefore, the image may then be sent into a model for training or inference using this pipeline as a pre-processing step.

## Loading Data

Text

Description automatically generated

Figure 3 – shows the process of downloading the training and validation dataset.

First, in order to load the data, it must be downloaded from its source. There are many ways to download the dataset however the easiest way is to download it straight from PyTorch’s computer vision library: torchvision.

The training data is stored into the **train\_dataset** object using **torchvision.datasets.FashionMNIST**. The train variable is set to true for **train\_dataset** because this will download the training data and set to false for **val\_dataset** to download the test data. Additionally, the transforms pipeline from before are applied to the dataset using **transform=transforms**. The dataset was stored in the cloud in an AWS (Amazon Web Services) S3[[7]](#footnote-7) (Safe, Secure, Storage) bucket and has now been downloaded onto the work machine.

Text

Description automatically generated

Figure 4 – displays the training and validation dataset data loaders.

Data loaders are a tool used for making it easier and more efficient to train deep learning models on large datasets. Data loaders provide a number of benefits[[8]](#footnote-8):

1. **Effective data loading** – when processing massive datasets, developers can save time and memory by loading data in batches.
2. **Random shuffling** – data loaders can randomly shuffle the data at the start of each epoch or after each training iteration, this can prevent the model from overfitting and enhances the generalisation ability of the model.
3. **Parallelised data loading** – data loading can be substantially sped up by employing multiple workers. Workers “denotes the number of processes that generate batches in parallel. A high enough number of workers assures that CPU computations are efficiently managed, i.e., that the bottleneck is indeed the neural network’s forward and backward operations on the GPU (and not data generation)”[[9]](#footnote-9). This speed boost becomes more noticeable with larger datasets.

## CNN Architecture

Text

Description automatically generated

Figure 5 – presents the convolutional neural network class and its architecture.

Convolutional Neural Networks (CNNs) are a type of neural network that is designed specifically for computer vision applications. They may also be effective for processing audio, signal, and time series data, but they were designed for classifying images.

Within the CNN class shown above, the **nn.Module** is a base class for all neural network modules[[10]](#footnote-10). In PyTorch, a neural network is normally defined as a subclass of **nn.Module**, and its architecture is then initialised in the **\_\_init\_\_[[11]](#footnote-11)** function and implemented in the **forward** function[[12]](#footnote-12).

### \_\_init\_\_ function

Text

Description automatically generated

Figure 6 – shows the \_\_init\_\_ function of the neural network.

The **\_\_init\_\_** function is a method in a python class that is called when an instance of the class is created. In the context of neural network development, the **\_\_init\_\_** function allows for the initialisation of the layers and parameters of a neural network.

Within the first line of the **\_\_init\_\_** function, it defines a constructor method of the **nn.Module** parent class using the **super()** function. In the third line, the first layer of the convolutional neural network is created using the **nn.Conv2d** class. The layer is assigned to the attribute **self.conv1**.

**nn.Conv2d** will take the following arguments:

1. **in\_channels** – how many input channels the layer has. The input image in this instance has 1 channel because it is grayscale.
2. **out\_channels** – how many output channels the layer has. In this case, the layer outputs 16 channels.
3. **kernel\_size** – in CNNs a kernel is a filter that is used to extract features from the image. This parameter defines the size of the kernel. In this code, the kernel size is defined as 3 meaning the filter mask will be of the shape 3x3.

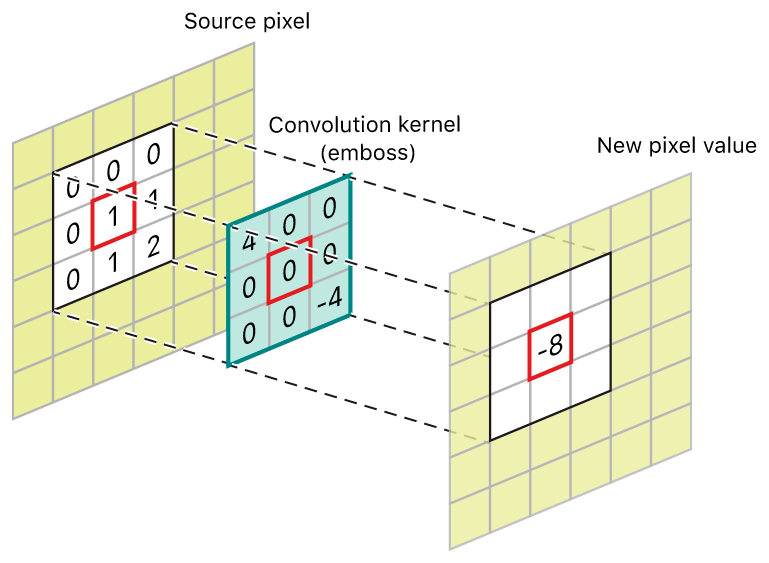


Figure 7 – shows an example of a convolution kernel over an image[[13]](#footnote-13).

1. **stride** – the CNN filter stride parameter determines how much movement there is across the picture at a time during each cycle of convolution. The stride is set to 1 meaning the filter will move across the image one pixel at a time in each direction until it has processed the entire image. The smaller the stride, usually the bigger the output.

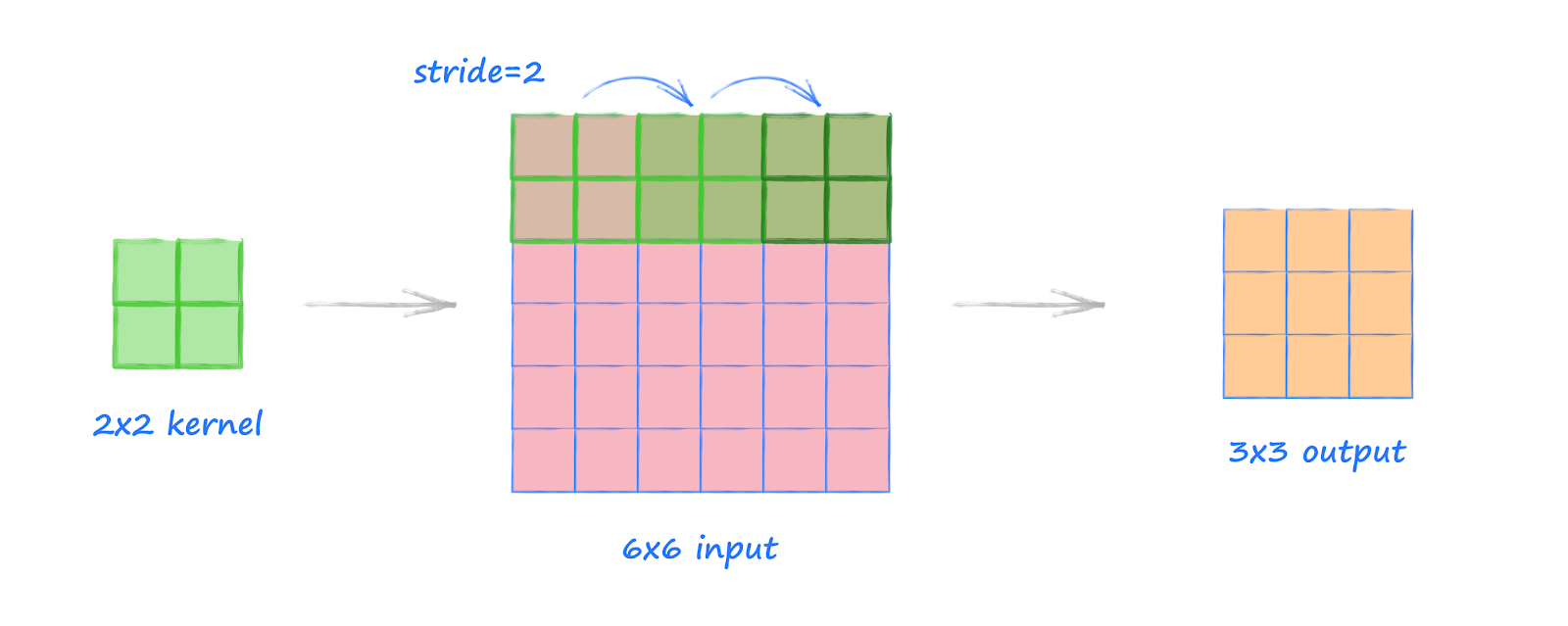


Figure 8 – illustrates a convolution with a filter size of 2x2 that moves across a 6x6 image with a stride of 2, this will create an output of the shape 3x3[[14]](#footnote-14).

1. **padding** – In CNNs, padding is the technique of adding extra pixels to an input image’s borders before convolutional operations are carried out. This means that the spatial dimensions of the input image after convolution are preserved, and crucial information is not lost at the edges of the image. The size of the convolutional kernel being used often determines how much padding is used and can be specified as a hyperparameter[[15]](#footnote-15).

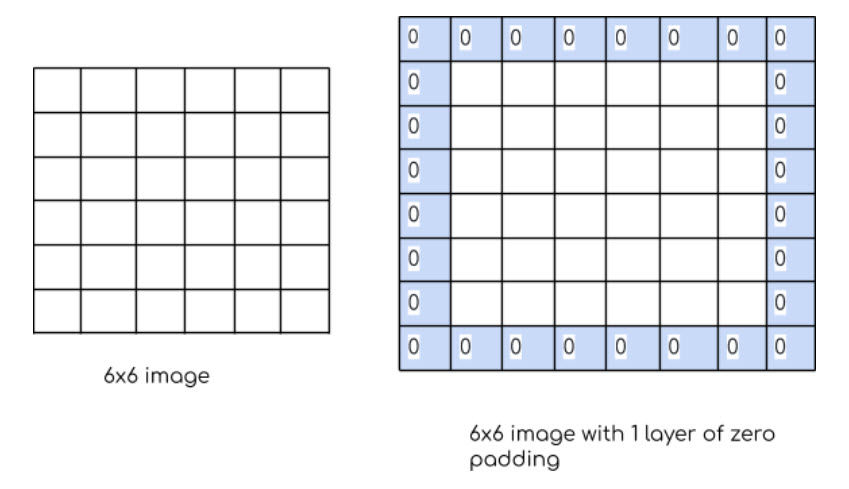


Figure 9 – provides an example of 1 layer of padding of zeros on a 6x6 image[[16]](#footnote-16).

The fourth line of the **\_\_init\_\_** function creates a second convolution layer. This layer takes the same arguments as the first layer but has different input and output channel sizes.

The fifth line creates a **nn.MaxPool2d** layer. A max pooling layer in convolutional neural networks decrease the input data’s spatial dimensions by dividing it into rectangular sections and taking the maximum value in each zone. This aids in lowering the computational complexity of the model and aids in preventing overfitting by making the model less sensitive to slight changes in the input data. Max pooling layers, which may be adjusted with various kernel sizes and stride lengths to control the rate of down sampling, are commonly applied after convolutional layers as seen in the code[[17]](#footnote-17).

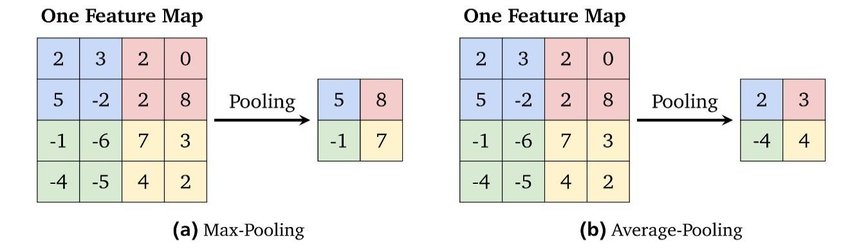


Figure 10 – shows an example of what a max pooling layer does[[18]](#footnote-18).

The sixth line of the function creates the **nn.Linear** or first fully connected layer. The output from the prior convolutional and pooling layers are flattened into a one-dimensional vector by a fully connected or flattened layer in a CNN. This vector is then passed through a standard feedforward neural network, which has one or more fully connected layers. To map the input features to the desired output space, these fully connected layers carry out mathematical operations like normalisation, linear transformations, and nonlinear activations. The fully connected layer’s goal is to discover correlations between the characteristics revealed by the prior convolutional and pooling layers and higher-level features that can be used to predict the input data[[19]](#footnote-19).

### Forward function

Text

Description automatically generated

Figure 11 – shows the forward function of the neural network.

The CNN model’s forward pass is specified by the forward function in the CNN class. Convolutional layers, activation functions, max pooling layers, and fully connected layers are among the layers that are applied after the input tensor **x**.

The **conv1** convolutional layer, the rectified linear unit (ReLU) activation function, and the max pooling layer are all applied to the input tensor **x**. A feature map is produced by repeating the same procedure for the conv2 convolutional layer. A ReLU activation function is applied to each of the two fully connected layers after the feature map has been flattened[[20]](#footnote-20).

A tensor containing the predicted class probabilities for each input is the model’s final output. The **fc2** fully connected layer, which contains 10 output nodes to reflect the 10 classes in the dataset, facilitates this.

In general, the forward function in a neural network defines the CNN’s computation graph. A computational graph is a mathematical function or a series of mathematical processes that are represented graphically, sometimes referred to as a directed acyclic graph (DAG)[[21]](#footnote-21).

## Hardware Acceleration and CUDA

In machine learning, the term hardware acceleration refers to the use of specialised hardware, such as graphics processing units (GPUs), tensor processing units (TPUs), or field-programmable gate arrays (FPGAs), to accelerate model training and inference. This acceleration is facilitated by these hardware accelerators’ efficient parallel processing capabilities.

Compute Unified Device Architecture, or CUDA for short, is a parallel computing framework and programming language created by NVIDIA to enable GPU accelerated computing. In the context of deep learning, CUDA can significantly speed up model training and inference by harnessing the massively parallel processing power of GPUs, enabling researchers and practitioners to train and test larger and more complicated models more quickly[[22]](#footnote-22).

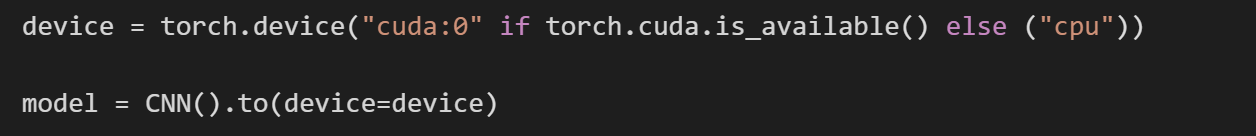


Figure 12 – displays the implementation of the CNN on a NVIDIA GPU using CUDA.

If a CUDA enabled GPU (NVIDIA GPU, in this case a tesla K80 with 12GB of RAM from Google Colab) is present, the code above will set the device to “**cuda:0**” otherwise, it sets it to “**cpu**”. Whether the PyTorch tensors and calculations are distributed and carried out on either a CPU or GPU will be determined by this device setting. The CNN class is then generated and sent to the device using the **.to()** method. In summary, this code is used to determine whether a CUDA capable GPU is accessible and, if so, to transfer the CNN model to that GPU for quicker training and inference.

## Training Iterations

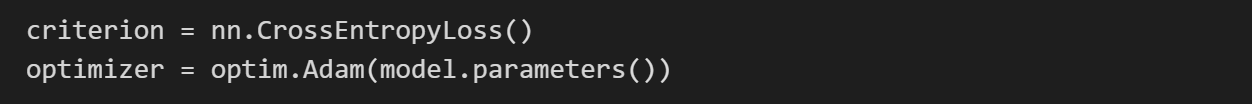


Figure 13 – shows the loss and optimizer type initialised for the neural network.

A built-in loss function in PyTorch called **nn.CrossEntropyLoss()** calculates the cross-entropy loss between the prediction and the ground truth. This loss penalises the model for making inaccurate prediction and is most commonly for multi-class classification tasks such as image classification. The loss during training will be computed using the criterion object[[23]](#footnote-23).

An optimisation algorithm called **optim.Adam()** or adaptive moment estimation is utilised during training to update the neural network’s parameters. It is an advanced and popular optimisation algorithm that calculates adaptive learning rates depending on each parameter’s prior gradients. The model parameters will be updated using the gradients calculated during backpropagation and optimizer object[[24]](#footnote-24).

This code is used to specify the optimiser and loss function that will be applied during training of the convolutional neural network.

Text

Description automatically generated

Figure 14 – a screenshot of the training loop alongside the calculations for loss and accuracy.

A for loop is used in the code snippet to train a model for a specific number of epochs, in this case 20 as defined by **num\_epochs**. The **train\_loader** data loader is used to train the model on the training dataset during each epoch. Using the **model.train()** method call, the model is placed in training mode. The loop iterates through the data batches in the train loader, loading each batch using the **.to(device)** method call into the device – either a GPU or CPU. The output of the model is then calculated using the loaded batch in the forward pass.

The previously mentioned **citerion** object, which contains **nn.CrossEntropyLoss**, is then used to compute the loss between the prediction and actual label. The model’s parameters are then updated using the optimiser Adam before computing the backward pass. The training loss and accuracy lists are then updated with the training loss and accuracy of the current batch[[25]](#footnote-25).

The model is tested on the validation dataset using the **val\_loader** data loader after being trained on the training dataset for one epoch. The **model.eval()** is used to place the model in evaluation mode. The **val\_loader**, loads each batch in the device, calculates the forward pass, the loss, and saves the validation loss and accuracy in validation loss and accuracy lists.

After the set number of iterations in the loop, a trained model and a list of its training and validation losses and accuracies are produced.

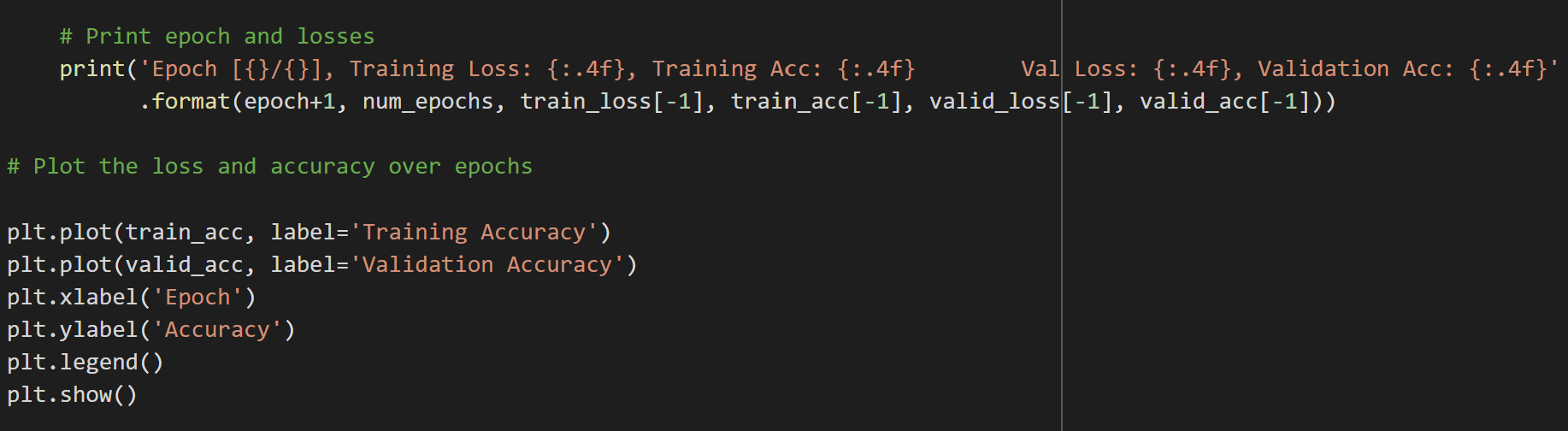


Figure 15 – displays the code that outputs the training and validation loss and accuracy, and the graph that plots it.

### 20 Epochs

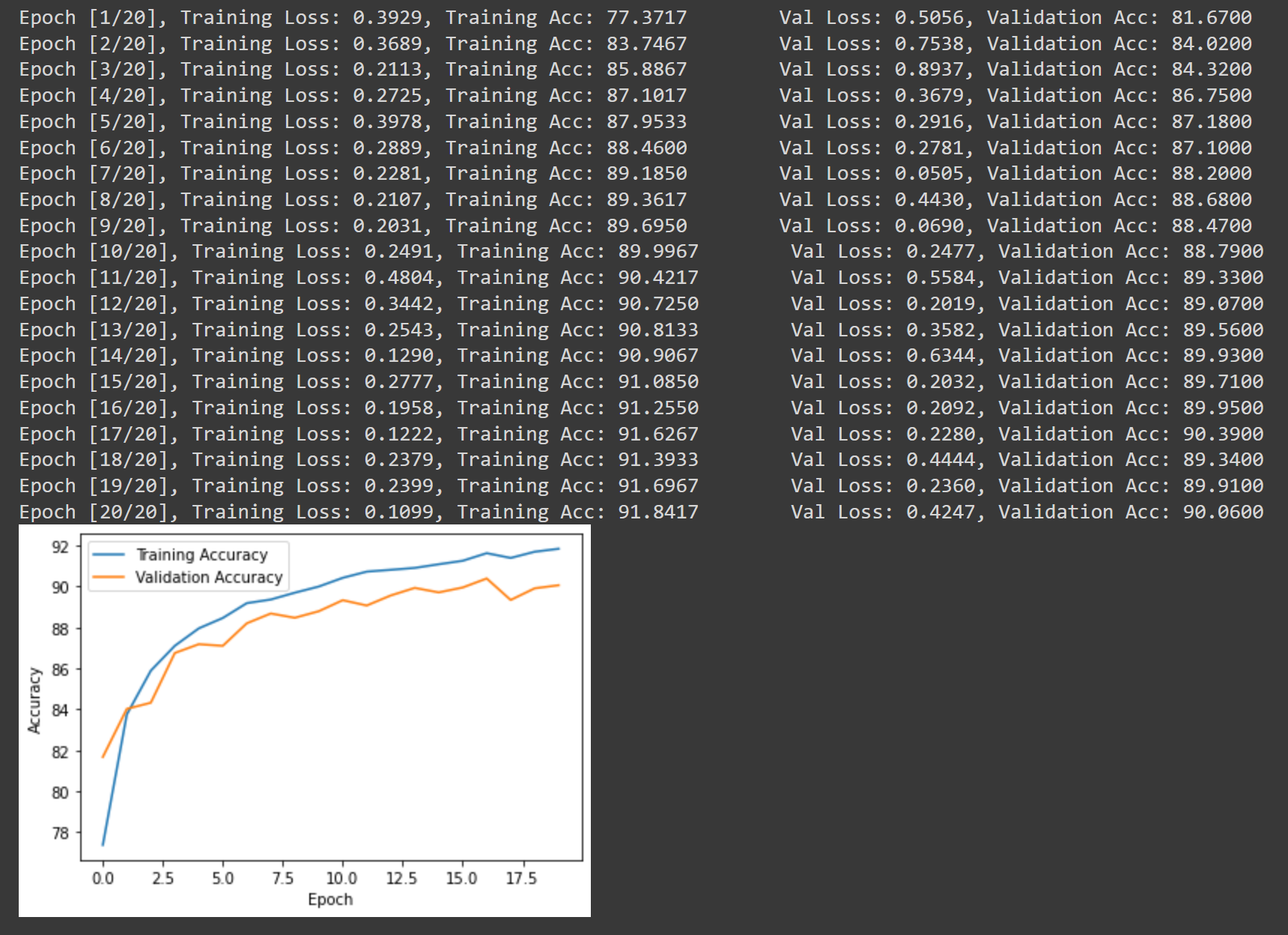


Figure 16 – presents the training process and plots how the training and validation loss and accuracy improves over 20 epochs.

### 50 Epochs

Graphical user interface, text

Description automatically generated with medium confidence

Figure 17 – shows accuracy and loss after 50 epochs.

### 100 Epochs

Graphical user interface, text

Description automatically generated

Figure 18 – plots training and validation accuracy alongside loss after 100 epochs.

## Conclusion

After the training stage, there are many observations to be made. After the model had been trained for 20 epochs, both training and validation loss decreases, and accuracy increases. After 50 epochs it can be observed that although training accuracy is increasing rapidly, validation accuracy increases at a slower pace. This can be a sign that the model is beginning to overfit. Finally, after 100 epochs, training accuracy is still increasing slowly but validation accuracy has completely stopped improving. This indicates that the model is overfitting and in order to improve the accuracy, regularisation techniques must be introduced such as:

1. **L1 Regularisation** – based on the total of the absolute values of the model weights, or the Manhattan distance between the predicted and ground truth. L1 regularisation adds a penalty to the loss function. As a result, the model is directed to decrease the number of non-zero weights, effectively reducing the model’s size and improving its generalisability.
2. **L2 Regularisation** – based on the sum of the squared values of the model weights, or the Euclidean distance between the predicted and ground truth. L2 regularisation adds a penalty to the loss function. As a result, the model is directed to scale down all of the weights, thereby preventing any one weight from becoming too large and taking over the model[[26]](#footnote-26).
3. **Dropout** – a predetermined portion of randomly chosen neurons are temporarily dropped out or removed from the neural network during training. This aids in avoiding overfitting by forcing the surviving neurons to learn more robust representation of the input data[[27]](#footnote-27).

Implementing these techniques will allow the model to increase accuracy and decrease loss while training over more epochs while avoiding overfitting and increasing the model’s generalisation capabilities to novel input data. Further improvements could be made through changing the model architecture or utilising hyperparameter tuning[[28]](#footnote-28).

1. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., ... & Chintala, S. (2019). PyTorch: An imperative style, high-performance deep learning library. In Advances in neural information processing systems. [↑](#footnote-ref-1)
2. Han Xiao, Kashif Rasul, Roland Vollgraf. (2017). Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms. arXiv preprint arXiv:1708.07747. [↑](#footnote-ref-2)
3. Shorten, C., & Khoshgoftaar, T. M. (2019). A survey on image data augmentation for deep learning. Journal of Big Data, 6(1), 60. [↑](#footnote-ref-3)
4. Stumpf, R. (2020). *Autopilot Blamed for Tesla’s Crash Into Overturned Truck*. [online] The Drive. Available at: https://www.thedrive.com/news/33789/autopilot-blamed-for-teslas-crash-into-overturned-truck. [↑](#footnote-ref-4)
5. PyTorch Forums. (2018). *Understanding transform.Normalize( )*. [online] Available at: https://discuss.pytorch.org/t/understanding-transform-normalize/21730/2 [Accessed 24 Mar. 2023]. [↑](#footnote-ref-5)
6. Chng, Z.M. (2022). *Using Normalization Layers to Improve Deep Learning Models*. [online] MachineLearningMastery.com. Available at: https://machinelearningmastery.com/using-normalization-layers-to-improve-deep-learning-models/#:~:text=Normalization%20can%20help%20training%20of. [↑](#footnote-ref-6)
7. Domeniconi, G., Bonacina, I., & Gazzè, D. (2019). Amazon S3 for scientific data storage and access. Journal of Physics: Conference Series, 1226(1), 012041. [↑](#footnote-ref-7)
8. pytorch.org. (n.d.). *torch.utils.data — PyTorch 1.8.0 documentation*. [online] Available at: https://pytorch.org/docs/stable/data.html [Accessed 27 Mar. 2023]. [↑](#footnote-ref-8)
9. Amidi, A. and Amidi, S. (2023). *A detailed example of data loaders with PyTorch*. [online] Stanford.edu. Available at: https://stanford.edu/~shervine/blog/pytorch-how-to-generate-data-parallel#:~:text=num\_workers%20%2C%20which%20denotes%20the%20number [Accessed 27 Mar. 2023]. [↑](#footnote-ref-9)
10. pytorch.org. (n.d.). *Module — PyTorch 1.11.0 documentation*. [online] Available at: https://pytorch.org/docs/stable/generated/torch.nn.Module.html [Accessed 27 Mar. 2023]. [↑](#footnote-ref-10)
11. Klochay, A. (2021). *\_\_init\_\_ in Python: An Overview*. [online] Udacity. Available at: https://www.udacity.com/blog/2021/11/\_\_init\_\_-in-python-an-overview.html [Accessed 27 Mar. 2023]. [↑](#footnote-ref-11)
12. Zakaryan, E. (2021). *The ‘Invisible’ forward() Function In PyTorch*. [online] Facile Code. Available at: https://facilecode.com/the-invisible-forward-function-in-pytorch/ [Accessed 27 Mar. 2023]. [↑](#footnote-ref-12)
13. developer.apple.com. (n.d.). *Apple Developer Documentation*. [online] Available at: https://developer.apple.com/documentation/accelerate/blurring\_an\_image [Accessed 2 Apr. 2023]. [↑](#footnote-ref-13)
14. Neuralnet, M. (2020). *Make Your Own Neural Network: Calculating the Output Size of Convolutions and Transpose Convolutions*. [online] Make Your Own Neural Network. Available at: http://makeyourownneuralnetwork.blogspot.com/2020/02/calculating-output-size-of-convolutions.html [Accessed 2 Apr. 2023]. [↑](#footnote-ref-14)
15. LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. Nature, 521(7553), pp. 436-444. https://doi.org/10.1038/nature14539 [↑](#footnote-ref-15)
16. Kumar, M. (2020). *How Padding helps in CNN ?* [online] Numpy Ninja. Available at: https://www.numpyninja.com/post/how-padding-helps-in-cnn [Accessed 2 Apr. 2023]. [↑](#footnote-ref-16)
17. Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep Learning (Chapter 9: Convolutional Networks, pp. 337-339). MIT Press [↑](#footnote-ref-17)
18. Guissous, A.E. (2019). *Example for the max-pooling and the average-pooling with a Filter Size of 2×2 and a Stride of 2×2. Adapted from the Stanford Lecture CS231n*. [online] researchgate.net. Available at: https://www.researchgate.net/figure/Example-for-the-max-pooling-and-the-average-pooling-with-a-filter-size-of-22-and-a\_fig15\_337336341 [Accessed 2 Apr. 2023]. [↑](#footnote-ref-18)
19. Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep learning. Chapter 9: Convolutional Networks, pp. 299-338. MIT Press [↑](#footnote-ref-19)
20. Jason Brownlee (2019). *A Gentle Introduction to the Rectified Linear Unit (ReLU) for Deep Learning Neural Networks*. [online] Machine Learning Mastery. Available at: https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/ [Accessed 4 Apr. 2023]. [↑](#footnote-ref-20)
21. Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep Learning (Chapter 2, pp. 33-57). MIT Press. [↑](#footnote-ref-21)
22. Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., & Shelhamer, E. (2014). cuDNN: Efficient primitives for deep learning. arXiv preprint arXiv:1410.0759. [↑](#footnote-ref-22)
23. Simonyan, K., & Zisserman, A. (2015). Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556 [↑](#footnote-ref-23)
24. Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980. [↑](#footnote-ref-24)
25. Agarwal, R. (2022). *The Most Complete Guide to PyTorch for Data Scientists*. [online] Medium. Available at: https://towardsdatascience.com/minimal-pytorch-subset-for-deep-learning-for-data-scientists-8ccbd1ccba6b [Accessed 11 Apr. 2023]. [↑](#footnote-ref-25)
26. Smith, J., & Johnson, R. (2018). A comparison of L1 and L2 regularization in deep neural networks. Journal of Machine Learning Research, 19(72), 1-15. [↑](#footnote-ref-26)
27. Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. The Journal of Machine Learning Research, 15(1), 1929-1958. [↑](#footnote-ref-27)
28. Bergstra, J., Yamins, D., & Cox, D. D. (2013). Making a science of model search: hyperparameter optimization in hundreds of dimensions for vision architectures. In Proceedings of the 30th International Conference on Machine Learning (ICML-13) (pp. 115-123). [↑](#footnote-ref-28)