Data Programming with Python

By Ayushree Gangal (18200623)

Dataset A

Section 1: Background

Libraries used: Pytorch (for deep learning) and Plotly (for data visualisation)

Aim: The main objective of this project is to use Pytorch to perform deep learning on an image dataset to recognise images. The problem of image recognition can be formulated as a classification problem where the model is required to recognise the image by identifying which of the given categories it belongs to.

We will first build and train our classification model on the given training data and then evaluate the quality of the model on the given testing data. Further, we will analyse the results using Plotly for data visualisation and suggest improvements to the model, if applicable.

Dataset: American Sign Language (ASL) dataset

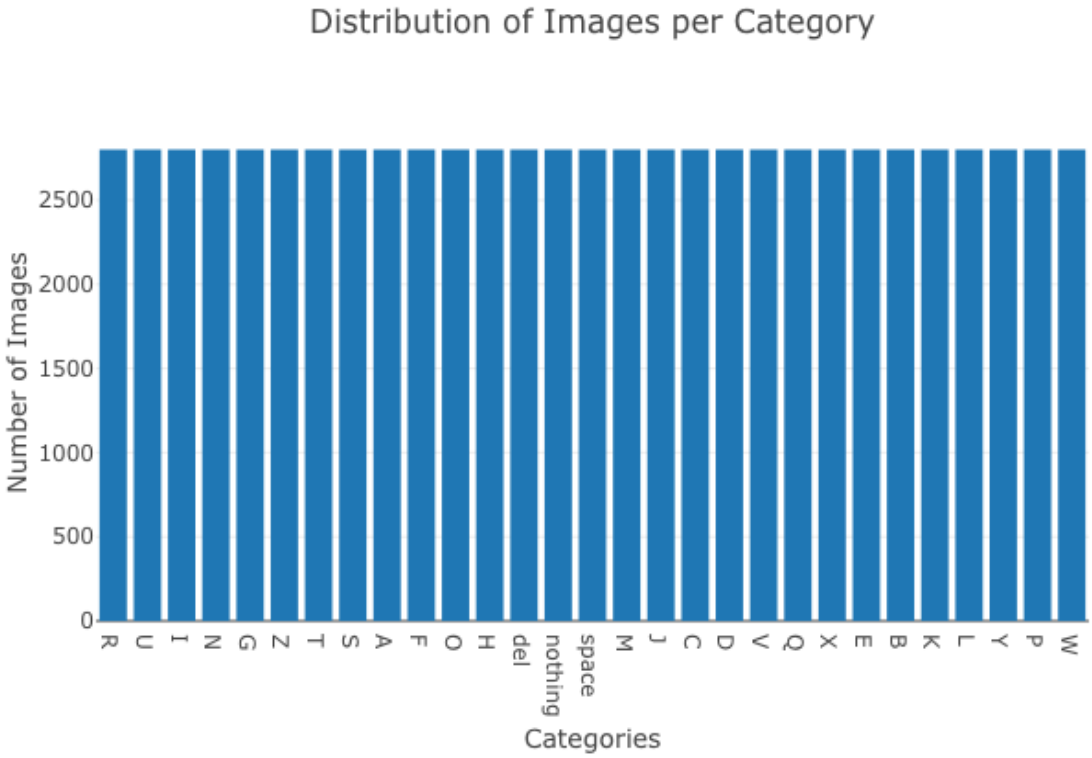
Source: Kaggle (https://www.kaggle.com/grassknoted/asl-alphabet)

About the dataset: The dataset consists of 87,000 images of alphabets of the sign language distributed over 29 categories. It contains only 29 test images (one per category) which is why we will split our training data into train, validation and test data. All images are 200x200 in size and coloured. To get the size of the dataset to below 100MB, we resized the images to 28x28 and deleted 200 pictures per category. This did not affect the performance of our model.

Motivation: The development of sign language recognition technologies is vital to creating a more accessible environment for the deaf and mute to communicate. As presented by Quan et.al. in their paper titled 'Application of improved sign language recognition and synthesis technology in IB', a system that recognises sign language can be used as a bridge that helps the healthy and the deaf and mute to communicate with each other naturally[1].

In a more common scenario, many universities still face problems while trying to provide support to the differently abled. A sign language recognition system can facilitate student involvement without requiring the healthy to learn sign language and yet giving the differently abled the opportunity to communicate with their peer group. This project is just a first step in this endeavour.

Descriptive Analysis: The dataset contains (after amending it) over 80,000 images of alphabets of the American sign language. After the necessary splitting, we end up with about 56000 training images, 16000 testing images and around 8000 validation images. Following chart represents the distribution of images for each category:



As we can see from the graph, all signs are equally represented which may result in our model performing in a uniform fashion for all categories.

Section 2: Introduction to concepts behind deep learning

1. Neural Networks: an Overview

Neural networks, as the name suggests, have been inspired by the neurons in the human brain. In terms of computer science, A neural network has an input layer, an output layer and a bunch of hidden layers in between the two and all these layers are connected to each other in some manner. At a very high level, we want to give an image as an input to the network, which should ideally identify patterns in the image and subsequently, the output layer should confidently classify the image as belonging to one of the categories.

Each image in our dataset is represented in terms of an array of pixels. As each image is 32x32 in size, we can view it as a 32x32 matrix of pixels. Thus, we can think of a neuron as an entity that contains a number that pertains to a corresponding pixel in the image. Thus, we can look at our 32x32 matrix of pixels as a 32x32 matrix of neurons containing information about its corresponding pixel. These 32x32=1024 neurons make up the first layer of our network. Our output layer on the other hand will have 10 neurons, each representing one of the categories in our dataset. The numbers contained in the neurons in the output layer loosely correspond to the probability that the image belongs to a particular category. The numbers contained in the neurons (also known as activations) in a layer (excluding the input layer) depend upon the activations of the previous layer and so on. Thus, each image ideally results in certain activations in the input layer which result in certain other activations in the next layer and so on till these activations affect the activations in the output layer which finally spits out the category of the image according to the model (the neuron with the highest activation).

With a connected layered structure, we are essentially breaking down the task of recognising an image in its entirety to each layer being given the task of recognising some part of the image. Now, in order for any layer to recognise a particular part of an image or a pattern, the connections between that layer and its previous layer are given some weights which result in a high activation number for the pixels corresponding to the pattern and low activation for all other pixels. Starting from the first layer, each neuron in the second layer is connected to all the neurons in the first layer by connections that have some weights. Thus, the activation of any neuron in the second layer is a form of a weighted sum of the activation neurons in the first layer. This is true for all neurons in the subsequent layers as well. This weighted sum is then normalised by using an activation function. Thus, the behaviour of the network can be controlled by changing all these weights, which essentially defines the concept of learning as finding the optimal weights in order to classify the given images correctly. In order to see how well the network has “learnt” based on the training data, we then test the model on data it has not seen before and use certain metrics to see how it classifies images given in that dataset. For the network to learn based on the training data, we need some method of identifying when to change certain weights. Thus, a cost function is defined based on the difference in the activations of the output layer and what those activations should have been based on our knowledge of the category of the image. Thus, for the model to improve, the cost function needs to be minimised. The cost function can be defined as the sum of the squares of the differences between the activations in the output layer and the ‘should be’ activations. In the end, we will take the average of all the cost functions for each image in the training data so while minimising the average, we will be improving the performance on all images. Due to the high number of parameters, we look for a local minima for the cost function using the concept of **gradient descent.**

Backpropagation: during learning, in order to change the activations in the output layer so as to minimise the cost function for any particular image, the network needs to figure out which activations to increase and which ones to decrease. Furthermore, it also needs to identify which of these proposed changes will have the maximum impact on the cost function and change the weights connecting the output layer to the previous layer accordingly. The same line of reasoning can be followed by the activations in the penultimate layer to figure out which of the weights connecting the penultimate layer and its predecessor to change. This process needs to then be repeated for each image. Thus, the change to all the weights become a sum total of all the

changes proposed by all the activations in the output layer by all images. This average change to the parameters is essentially the negative of the gradient function which strives to minimise the overall cost function to improve the model's performance.

2. Convolutional Neural Networks for Image Recognition

Convolutional neural networks or CNNs are typically used for image recognition/classification. CNNs follow the layered structure, as described above, where each layer identifies a part of the image and by a collective effort, the network identifies the entire image. Rather than trying to recognise the image as a whole, CNNs use features to match parts of the image. Each feature essentially represents a pattern which is compared to different parts of the image to see if a match can be obtained. A **convolution layer** uses these features to filter the input image into a stack of filtered images using a mathematical formula. Thus, each filtered image represents a part or a pattern in the given image.

A **pooling layer** shrinks the filtered images obtained as a result of the operations of the convolution layer. A maxpooling layer uses a window to run in a defined stride length over each of the filtered images and pick the maximum value in that window. Because we are choosing the maximum value in each window, the pattern recognised in our original image by the convolution layer is still preserved after pooling. Thus, we have increased network efficiency and decreased time by shrinking our image size while still preserving the pattern that the layer is supposed to recognise.

Another layer is the **activation layer** which tweaks the values of the images a bit. In the case of ReLU (rectified linear unit), the function takes the max of zero and the value of the image, i.e., it converts negative values to 0 and leaves non-negative values unchanged. Activation functions are used to introduce non-linearity in the network so as to make them complex enough to be able to represent any possible function.

At the end of all these layers, a single image has been split into a stack of filtered images which have been reduced in size while keeping the patterns they represent intact, and containing only non-negative values. These three layers can be stacked any number of times in the model in a manner where the output of one layer becomes the input for the next layer.

Once we have stacked the aforementioned layers multiple times, the output is a much reduced array as compared to the original pixel array of any image. This output array is then fed to a fully connected layer as a one dimensional array where each element in the array gets a vote depending on what the original image was. A particular image will ensure that certain values in the output array are high and have a stronger say in classifying the image.

Following the concept of backpropagation, while designing any model, there are certain hyperparameters whose values are decided by the designer. Some of these are: the number and size of features in a convolution layer, the window size and stride in a pooling layer, the number of neurons in the fully connected layer and the overall architecture of the model.

Section 3: Our Model's Architecture

```
ASLNet(
  (layer1): Sequential(
    (0): Conv2d(3, 16, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (layer2): Sequential(
    (0): Conv2d(16, 32, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (fc): Linear(in_features=3200, out_features=29, bias=True)
)
```

Section 4: Our Model

```
training_dataset = TrainingDataASL(
    '/Users/ayushree/Desktop/data prog (python)/Project_2/final_code/iambrill/asl_alphabet_train_resized/')
training_set, testing_set, validation_set = random_split(training_dataset, [56840, 16240, 8120])

batch_size = 32
train_loader = torch.utils.data.DataLoader(dataset=training_set, batch_size=batch_size, shuffle=True)
test_loader = torch.utils.data.DataLoader(dataset=testing_set,
                                          batch_size=batch_size,
                                          shuffle=True)
validation_loader = torch.utils.data.DataLoader(dataset=validation_set, batch_size=batch_size, shuffle=True)
```

Pytorch makes use of an efficient way to generate and store data by using DataLoader. Due to the large size of datasets required for deep learning, storing them in memory may not be the best way to efficiently perform deep learning on them. DataLoader takes the following arguments:

1. Data: training or testing data
2. batch_size: DataLoader generates batches which contain the same number of elements as batch_size. Thus, in our case for batch_size = 32, each batch of images will contain 32 images.
3. Shuffle: the truth value of this argument determines whether the order of images remains the same between epochs or becomes different each time. The latter ensures that the model is more robust.
4. num_workers: DataLoader uses parallel programming and thus in our case uses 0 workers by default (main process) to generate batches of images in parallel.

Now, we define a class called ASLNet where the 'init' method contains all the different layer definitions in our model (as shown in the model architecture). We will also define how the forward pass of our model will work by defining the forward method of this class in the above manner. Thus, our layer 1 is a combination os a convolution layer, a ReLU activation layer and a MaxPool2d layer. We used ReLU (Rectified Linear Units) over Tanh and Sigmoid as both these functions present problems that ReLU has overcome. This is followed by layer 2 which has the same structure as layer 1. Finally, we reshape our input to flatten it and feed it to a linear layer called fc. The linear layer serves as an output layer which applies a linear transformation to the given input and gives out 29 values as the output. We finally return the output.
Once our ASLNet class has been defined, we define an instance of this class called model. We then define our loss function as 'CrossEntropyLoss' and choose the 'Adam' optimiser (a version of the gradient descent algorithm and fairly efficient).

```
for epoch in range(num_epochs):
    num_correct_class = 0
    num_classes = 0
    total_accuracy = 0
    for i, (images, labels) in enumerate(train_loader):
        images = Variable(images.float())  # .to(device)
        labels = Variable(labels.long())
        images = images.float()
        optimizer.zero_grad()
        outputs = cnn(images)
        predicted_class = torch.max(outputs.data, 1)[1]
        num_classes += len(labels)
        for j in range(len(predicted_class)):
            if predicted_class[j] == labels[j]:
                num_correct_class += 1
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        losses.append(loss.item())
```

Next, we train our model. For a particular epoch, we extract the image information (variable: inputs) and the classes that they belong to (variable: labels). We run our model on the inputs and store the result in a variable called outputs. Next, we compute the accuracy by finding out the number of correctly classified images. We then compute the loss function. loss.backward() is the backpropagation step. Pytorch's autograd class requires the user to define a forward pass but it takes care of the backpropagation by learning and updating the weights. This is the reason why we do optimiser.zero_grad() in the beginning as the .backward() step is a cumulative command that adds up all the gradient in each back pass. So as to compute the gradients correctly for each pass, we initialise them to zero at the beginning of each pass. We then compute the loss, both total and mini-batch loss.

The next step is the validation step. We use the previously defined validation_dataloader() to compute loss and accuracy for the validation step.

Once we have the total accuracy per epoch, the total loss per epoch and the validation loss and accuracy per epoch, we test our model on the test data. The steps are similar to those followed in the testing data but instead of running our model on the training_loader we use the testing_loader.

Section 5: Results

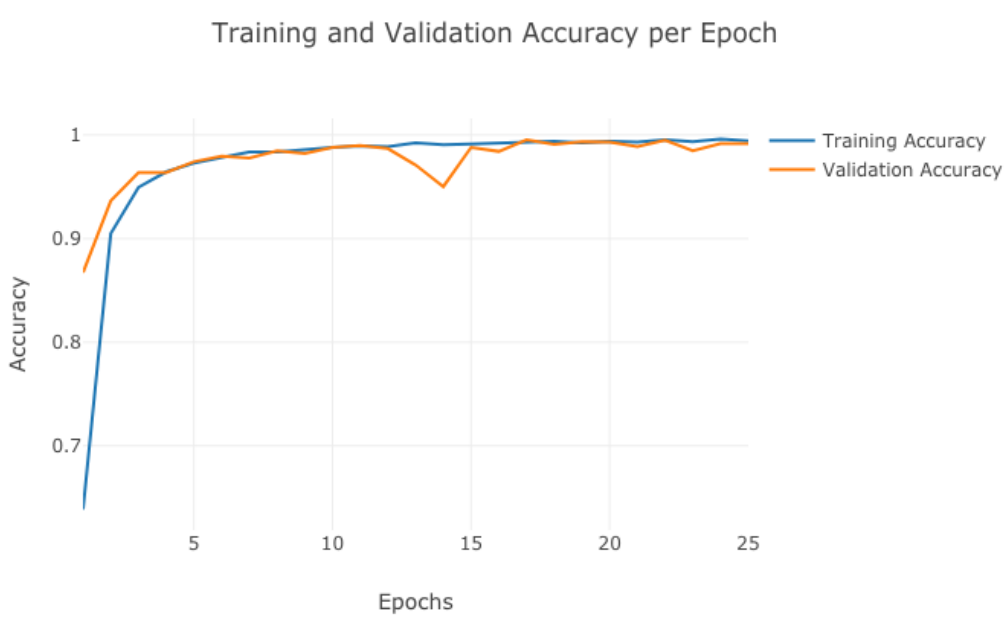
We ran our model for 10, 25,50 and 75 epochs. The results are as follows:

Epochs	Training Loss	Training Accuracy	Test Accuracy
10	0.1302	0.9883	0.98
25	0.0009	0.9941	0.98
50	0.0000	0.9970	0.99
75	0.0008	0.998	0.99

Some observations:

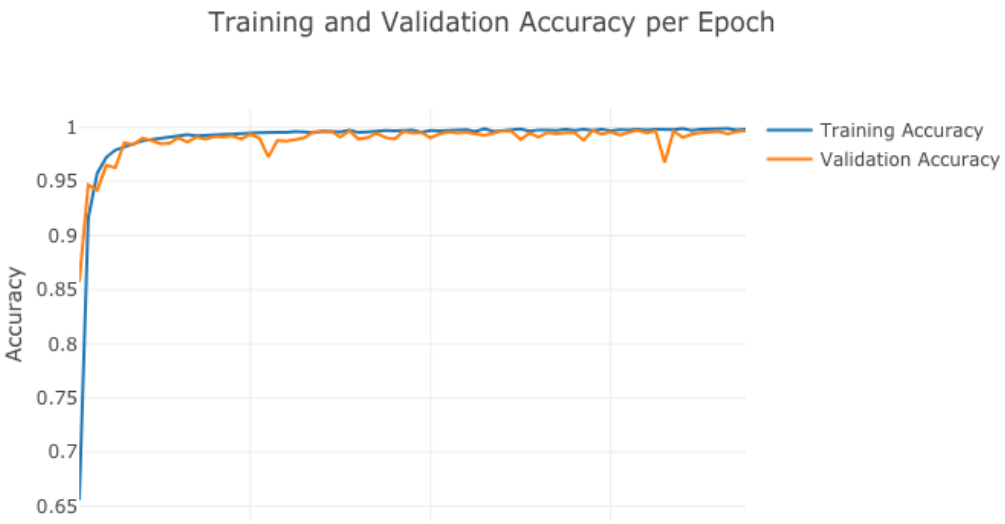
1. Our model's accuracy and loss do not fluctuate much across the number of epochs. Thus, our model is fairly robust even for 10 epochs.
2. We get fairly high testing accuracy for even ten epochs. We thus perform validation to check for any overfitting. As validation gives similar accuracy as training and testing data, the high testing accuracy comes from our model being able to learn fast.

Section 4: Data Visualisation using Plotly



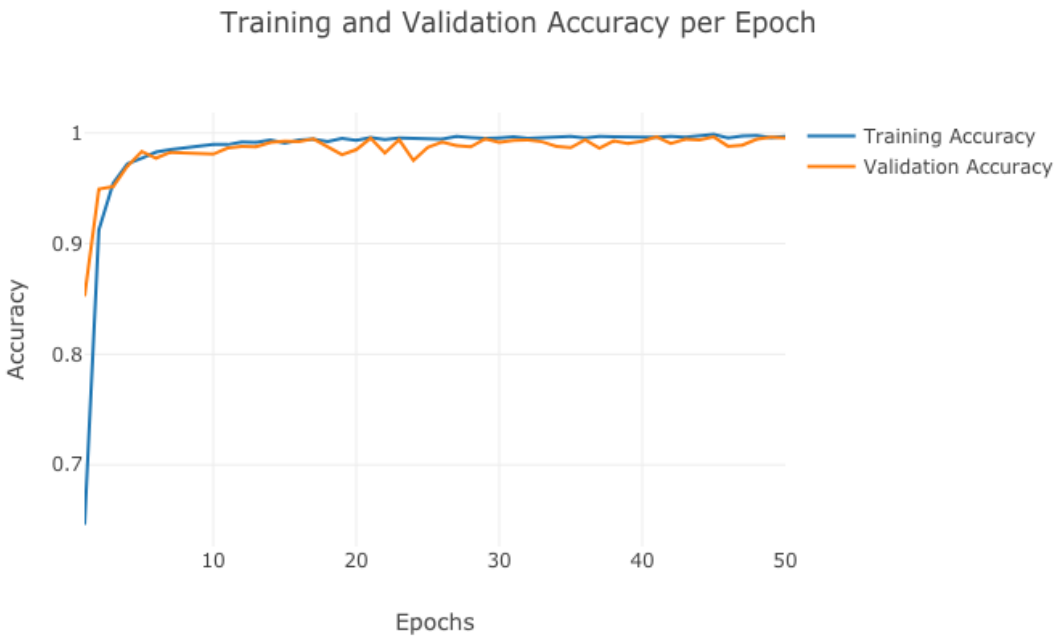
This plot shows the training and validation accuracy of the model per epoch for 25 epochs. As the two are fairly similar, overfitting is not taking place.

This plot shows the training and validation loss of the model per epoch for 25 epochs. Training loss is higher than validation loss indicating absence of overfitting.

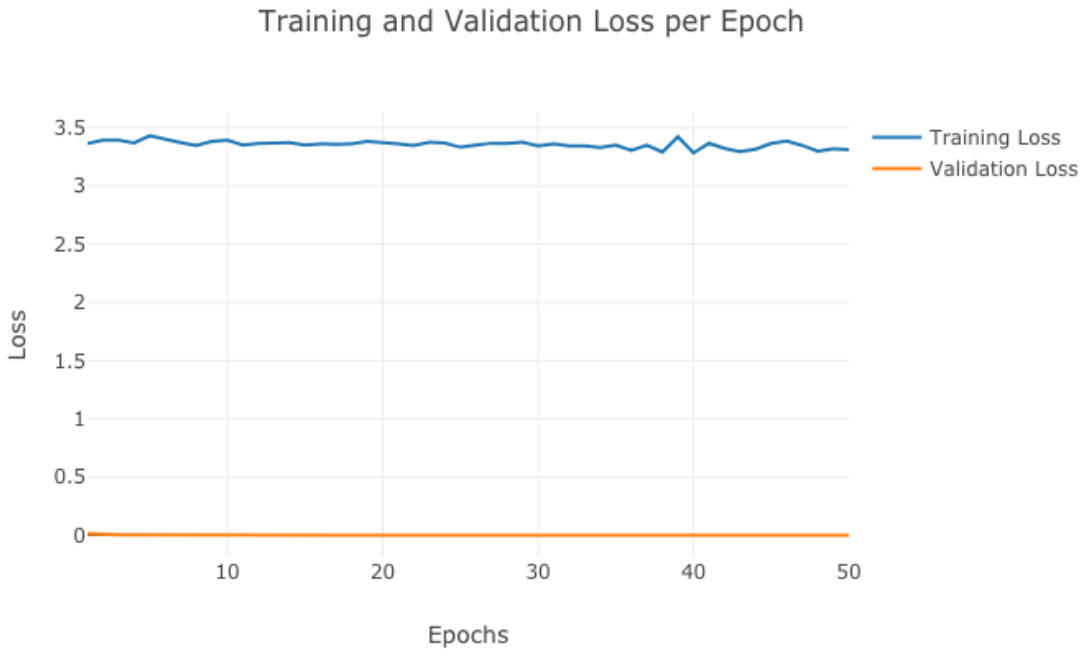


This plot shows the training and validation accuracy of the model per epoch for 75 epochs. As the two are fairly similar, overfitting is not taking place.

This plot shows the training and validation loss of the model per epoch for 75 epochs. Training loss is higher than validation loss indicating absence of overfitting.



This plot shows the training and validation accuracy of the model per epoch for 50 epochs. As the two are fairly similar, overfitting is not taking place.



This plot shows the training and validation loss of the model per epoch for 50 epochs. Training loss is higher than validation loss indicating absence of overfitting.

Bibliography

"Sign Language Recognition System Based on Prediction in Human-Computer Interaction." *Integrating Safety, Health and Environment (SHE) into the Autonomous Maintenance Activities*, by Maher Jebali, 1865-0929, pp. 565–570.

Yang, et al. "Application of Improved Sign Language Recognition and Synthesis Technology in IB." *An Introduction to Biometric Recognition - IEEE Journals & Magazine*, Wiley-IEEE Press, ieeexplore.ieee.org/document/4582795.

Dataset : https://www.kaggle.com/grassknoted/asl-alphabet