Student Name: Rohith Koti, Student Number: 241082360, Module Identifier: ECS7026P

Section 1: Dataset

The data augmentation pipeline consists of images transformed. The main transformations I used were horizontal flipping of the image, moving to tensor and lastly standardisation. Data augmentation is beneficial because we synthetically increase the size of our dataset by introducing distortions in our original images. This enables our deep learning model to be trained on a more diverse range of training examples and, therefore, increases its generalisation capabilities (reduce overfitting). Dataloader allows us to perform efficient mini-batch processing by enabling prefetching of batches, so it helps with reducing the training time. We set shuffle=True so that the model does not learn the order of the images in the datasets. Pin memory is for streamlining data transfer between the CPU and GPU. Batch size was 256.

Section 2: Intermediate and Output Block

2.1 Intermediate Block

Global Average Pooling. We calculate an average for each channel, so if we have c channels, we get c numerical values. This is stored in a c-dimensional vector m.

Dropout to reduce overfitting.

The skip connection consists of a convolutional layer followed by a batch normalisation layer. If the stride is greater than one or the number of in channels is not equal to out channels, then we have a dimensionality mismatch. Therefore, we must use a 1x1 convolution with the same number of out channels as the outputs from our main residual path (through the block). My architecture consists of alternating convolutional and batch normalisation layers. Num_layers is the number of convolutional layers L.

A convolutional layer followed by Batch normalisation for each L. Batch normalisation stabilised the training process (so we do not get large fluctuations in our accuracies from each epoch. and hence makes larger learning rates more feasible. This is because high learning rates (step size) tend to lead to instability in gradient descent, as we are more likely to overshoot the optimal point.

The attention weight tensor a is the output of the linear layer (with input m) where each entry of a is an attention weight for the weighted sum with convolutional outputs. To compute this sum, A broadcasting mechanism allows us to multiple tensors with mismatched dimensions. One of the corresponding dimensions of the two tensors must equal 1, which is why I used out.view(-1,1,1,1).

2.2 Output Block

Global average pooling gives dimension (batch size, channels, 1,1) and then reshaping to get dimension (batch_size,channels). After global average pooling, the height and width dimensions are reduced to 1, so we get dimensions after out.view(out.size(0),-1) as (batch size, channels). So, if we have a total batch size*channels*height*width pixels, then the second dimension is this product divided by the batch size (first dimension). The number of channels is 512 (given by layer 4 of our residual network). So, the tensor is passed through a fully connected layer where the number of input neurons to the fc layer is 512 (number of out channels from layer 4), and the number of neurons in the output is 10, which matches the number of classes in this image classification task.

Section 3: Training and Testing

I trained over 100 epochs, although I achieved 85-86% accuracy by epoch 50.

I had to move the dataset of features and targets in each batch to the GPU device. Also, the dimension of y from our data loader object is given by (batch size,) since this is a multi-class classification task (not multi-label, in which case the dimensions would be (batch_size,num_classes). We want the batch size to add to our running total of sample counts, so I specified y.size(0) as the batch size is in the first dimension.

Model.eval puts the model in evaluation mode, which is helpful for disabling dropout. This ensures consistency during our inference. Also, evaluation mode ensures batch normalisation uses population statistics instead of batch-specific statistics. Disable gradient calculations with torch.no_grad().

We use the package matplotlib to plot the training accuracy and test accuracy curves against the epoch number. This is useful for visualising the epoch number at which the training and testing accuracies diverge and can indicate underfitting or overfitting. If the training accuracy is consistently about 5-10% above the testing accuracy, as seen by the curve, it is a good indication of overfitting. We can, therefore, introduce early stopping to save training time if we know the model will overfit.

Section 4: Improving the Results (Hyperparameter Tuning)

Possible high-level hyperparameters we could tune are: number of epochs, batch_size, which gradient descent optimizer to use such as Adam, SVG, number of convolutional layers, number of residual blocks, dropout layer probabilities, augmentation transformations. The last four listed were the primary focus of this project.

When I changed the dropout probability to 0.5 in the intermediate block after each batch normalisation, my model, unexpectedly, was still severely overfitting, with my training accuracy reaching 97% but testing accuracy of only 88%. This is why I tried to include a dropout layer in the output block, too. After this, my model no longer overfit, and my training accuracy was not nearly as high. I tried adding a dropout

layer with a dropout probability of 0.5 after the fully connected layer in the output block. However, although my model was no longer overfitting, the testing accuracy was over 30% higher than my training accuracy for most epochs. My training accuracy stabilised around 55%, but my testing accuracy reached 87-88%. I tried to add auto augmentation but saw a decrease in testing accuracy. The testing accuracy struggled to exceed 85% with it. Regarding the number of convolutional layers, for most of my experiments, I tried L=16, 32, 64, 256, 512, but it was clear that the tuning of this hyperparameter was not influential enough to increase the testing accuracy to over 90%. Since my later tests include dropout in the output block too, overfitting should no longer be a problem. Therefore, I could focus on capturing more complex and abstract relationships.

Possible low-level hyperparameters we could tune are kernel_size (dimensions of the sliding window), stride (how many pixels does the sliding filter window move to the right, and padding; we add a layer of zeros to the edges of the image to ensure that each pixel contributes equally to the values in the output feature maps. Pixels on the edge of the image contribute less to the output feature map than more central pixels when using a sliding window, so we use padding to weigh each pixel equally.

An additional hyperparameter included weight decay (l2 regularisation) to mitigate overfitting. It adds an L2 penalty to the loss, which punishes large model weights. I tried a standard weight decay value of 1e-4.

Optimal hyperparameters found from adjustments: 256 convolutional layers, two intermediate blocks in layer 1, 5 in layer 2 & 3 and 2 in layer 4, dropout with p=0.5 in the intermediate block, and dropout p=0.1 in the output block after the fully connected layer and augmentation sequence from section 1 (more accurate than when I tried a longer pipeline). The hyperparameter combination mentioned in this paragraph gave me the best balance between high testing and high training accuracies. There was a discrepancy of about 1-2% for later epochs but less than 3% for most.