Final Report CV1 - Part 2

Kristiyan Hristov (12899437)
Macha Meijer (13136011)
Karim Abdel Sadek (15079015)

October 2023

1 Part 1 - Classification on CIFAR 100 Dataset

1.1 Dataset

In the first part of the assignment we were asked to implement a simple Two Layer Neural Net and LeNet-5 for the purpose of classifying the images in CIFAR-100 Dataset. It contains 100 different classes. Each image has the shape of 32 x 32 x 3 (RGB image with width and height of 32).

1.2 Training Loop, Optimizer and Augmentations

In this section, we delve into the intricacies of the training mechanism employed for our deep learning models, highlighting the strategies, tools, and methodologies that guide the model through its learning process. We will explain only the final training that was utilized for achieving the results that we submit in the notebook. It is important to note however, that in order to reach this state of training configuration hyperparameters and train augmentations we had to iterate multiple times and experiment with different settings. The explanation below is the best we managed to get.

1.2.1 Training loop

The training loop is crucial in shaping the model's learning trajectory. We trained our model over a series of epochs, where each epoch consists of a forward pass, loss computation, backward pass, and parameters update. Additionally, we calculated the accuracy on the training data at the end of each epoch to monitor the model's learning progression.
To prevent the model from being biased towards a particular class and to enhance its generalization capabilities, we employed extensive data augmentation techniques on the training data. Furthermore, at the end of each epoch, we evaluated the model's performance on a separate test dataset to ensure that it generalizes well to unseen data.
The PyTorch library was utilized to implement the training loop, ensuring a

robust and efficient training process. The model, along with the training data, were transferred to a CUDA-enabled GPU device if available, to expedite the training process.
Important Note: the training procedure used for all models in our work is finally the same! No training is done under different setting in our final submission!

1.2.2 Optimizer and Learning Rate Scheduler

We experimented with different optimizers Adam, AdamaW, Lion and SGD. After lots of iterations we chose the Stochastic Gradient Descent (SGD) optimizer since it gave the best performance. The best hyperparameters we found are: learning rate of 1×10^{-3}, weight decay of 1×10^{-4}, and momentum of 0.9. These parameters were carefully chosen to balance the trade-off between training speed and model stability.
To enhance the training process further, we incorporated a OneCycleLR learning rate scheduler. This scheduler dynamically adjusts the learning rate during training, following a cyclical schedule that starts low, gradually increases, and then decreases (see Figure 1). This strategy not only aids in faster convergence but also results in a more accurate and robust model.// **Important Note:** the optimizer and learning rate scheduler used for all models in our work is finally the same! No training is done under different setting in our final submission!

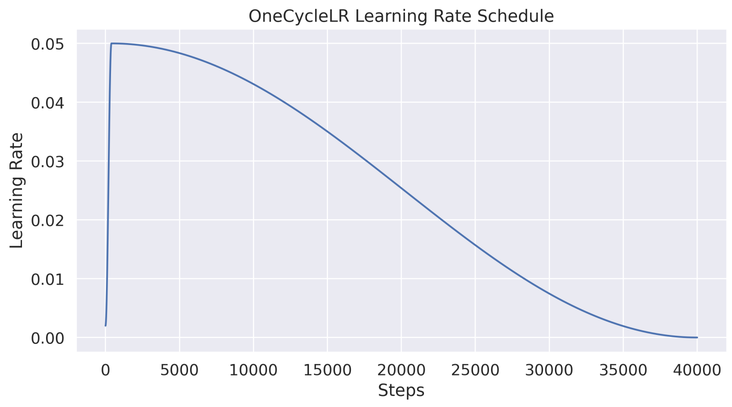

Figure 1: Learning Rate Scheduler

1.2.3 Augmentations

Data augmentation plays a vital role in training deep learning models, especially when the dataset is limited in size (this matters a lot for the finetunning phase). For our training data, we applied a series of augmentation techniques including random resizing and cropping, horizontal and vertical flips, color jittering,

affine transformations, perspective transformations, and random grayscaling. We intentionally resize the CIFAR-100 images to 96 x 96 in order to match the dimensions of our target dataset (STL-10) for which we have to finetune the network. The resolution has a huge impact on the training - a model that is trained to "see" features in smaller resolution will try to match this size to the second dataset - for example if our kernel size is 3 x 3 the 3 x 3 patches in an image 32 x 32 will represent bigger features than 3 x 3 patches in the 96 x 96 images. This however will introduce a blurring effect on our 32 x 32 CIFAR-100 images which is addressed architecturally in the ConvNet and will be explained in the following sections.
On the other hand, the test data underwent minimal preprocessing, consisting of resizing, tensor conversion, and normalization. These preprocessing steps are crucial to ensure that the input data is in the right format and range for the model to perform well.
These augmentation and preprocessing techniques not only helped in reducing overfitting but also aided the model in learning a more robust representation of the data, ultimately resulting in better performance on unseen data. **Important Note:** the augmentations used for all models in our work is finally the same! No training is done under different setting in our final submission!

1.3 Base Networks

1.3.1 TwoLayerNet

Our choice for the architecture of the TwoLayerNet were simple: we need only one parameter there - the hidden size of the first Linear layer. We chose 512. The output layer must be with 100 nodes since that is how many classes we have in our Dataset. The results we obtained are the following:
Final Train accuracy: 20.938% ; Final Test accuracy: 16.04%

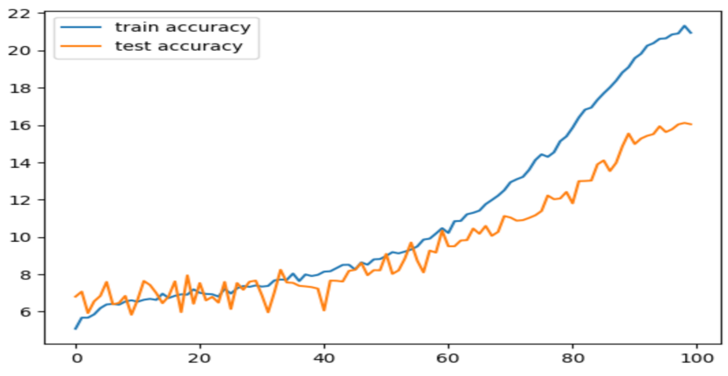

Figure 2: TwoLayerNet Accuracy (epochs/accuracy %)

1.3.2 LeNet-5

For the LeNet-5 we didn't have any choices since we were asked to implement the model from the original paper, so we just replicated it. The input channels for the first layer are 3 since we have RGB image and not monochrome. Kernel size and output channels are as in the paper for the whole Network except again the final Linear layer which we had to change from 10 to 100 nodes. In the forward pass after the last Convolutional layer (and pooling) we needed to flatten the Channel, Width and Height dimensions of the tensors in order for the Linear layer to be able to process them. The computation is very simple - the number of input features to the Linear layer is equal to $C * H * W$. For an input with resolution 32 x 32 we will have 16 x 5 x 5. 16 is the number of output channels from the last Convolutional Layer and 5 x 5 are the with and height of the the feature map. In the notebook you will see 16 x 21 x 21 since in the train transform we upscale the images to 96 x 96 (see the augmentations section above). The results we obtained are the following:
Final Train accuracy: 38.112% ; Final Test accuracy: 43.12%

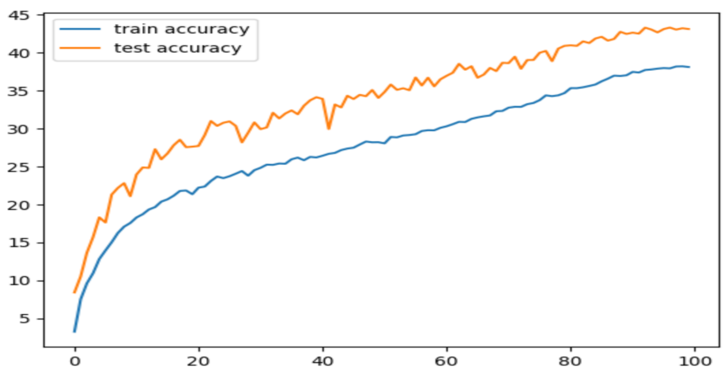

Figure 3: LeNet-5 Accuracy (epochs/accuracy %)

So we see that the LeNet-5 does a way better job at classifying the images than a simple Two Layer Perceptron. This of course is expected since CNNs we specifically invented to work on image data (Convolutional layers are translation invariant and represent locality and geometric structure).

Important Note: Since we don't have a validation set we took the liberty of monitoring the test set accuracy during training. We know this is something fundamentally wrong in the process of developing Machine Learning models but since it was never stated to be prohibitive in this assignment we did it.

1.4 Improving the Networks

1.4.1 TwoLayerNet

Firstly we will focus on the Improvements of the TwoLayerNet since this is the one that is not important for the final goal of finetuning for the STL-10 Detaset. Ass instructed we added two more linear layers (with 512 and 256 nodes respectfully) to the network and also increased the hidden size of the first linear layer from 512 to 1024 nodes. The result we got are the following:
Final Train accuracy: 34.412% ; Final Test accuracy: 24.06%. This is a big improvement over the simple TwoLayerNet: Test accuracy **improved with 8% - from 16.04% to 24.06%**. If we wanted we could have possibly improved it even further - we suspect that the learning rate we set might not be optimal for such a Network but we focused most of out time improving the ConvNet since it is the one we have to finetune for Part 2.

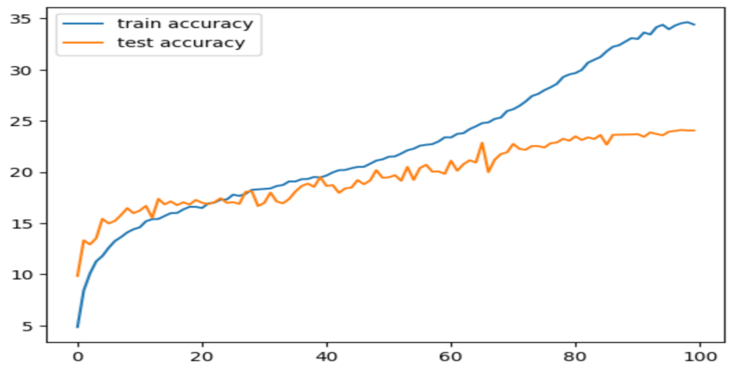

Figure 4: Improved TwoLayerNet Accuracy (epochs/accuracy %)

1.4.2 ConvNet

Improving the LeNet-5 was the part where we spent almost all our time. The main focus was to get as high accuracy as possible on CIFAR-100 Dataset so that later for the finetuning on STL-10 Dataset we will have the best start possible. The architecture and all hyperparametrs we present here were found after extensive iteration - we did around more than 50 trainings throught the two weeks we had to find them.

Firstly we designed our custom Conv block which is the main building block of our network. It is defined as follows:

- **Initialization:** The constructor takes several parameters:
  - `in_channels`: Number of input channels.

  - `out_channels`: Number of output channels.
  - `kernel_size`: Size of the convolutional kernel.
  - `downsample` (default: `True`): Whether to downsample the input.
  - `stride` (default: 1): Stride of the convolution.
  - `padding` (default: `None`): Padding applied to the input. If `None`, it is set to `kernel_size // 2`.

  Inside the constructor, the following layers and operations are defined:

  - `conv`: A 2D convolutional layer.
  - `bn`: Batch normalization.
  - `relu`: ReLU activation function.
  - `pool`: Max pooling layer if `downsample` is `True`, otherwise an identity operation.

- **Forward Pass:** The forward pass of the block is standard - it passes the through all parts of the block.

Secondly we will outline the final model architecture. We stack 5 such blocks one after the other followed by 3 linear layers similarly to the original LeNet-5 (only difference we add 1 dropout layer between the second and third Linear Layer).

- **Conv Block 1 (Stem):**
  - `input channels`: 3 (RGB image).
  - `output channels`: 64.
  - `kernel size`: 3x3
  - `stride`: 3
  - `downsample`: False
  - `padding`: 0
- **Conv Block 2:**
  - `input channels`: 64.
  - `output channels`: 128.
  - `kernel size`: 5x5
  - `stride`: 1
  - `downsample`: True
  - `padding`: 2

- **Conv Blocks 3 - 4:** The next 3 Conv blocks follow a similar pattern to **Conv Block 2**: $InputChannels = OutputChannels$ of the previous layer and $OutputChannels = 2 * InputChannels$. The only difference with Conv Block 2 is that for the next 3 blocks we use a kernel of 3 x 3.
- **Conv Block 5:** Follows the same pattern explained for Conv Block 3 and 4 but since this is our last convolution before the Linear Layers we don't do Max Pooling but instead Global Average Pooling that shrinks the resolution of each feature map to just 1 pixel. By doing this we end up with a tensor with shape: 128 ($BatchSize$) x 1024 (number of $OutputChannels$ of the last Convolution) x 1 ($Width$) x 1 ($Height$). This tensor when sqeezed on the last 2 dimensions becomes exactly in the form which the Linear Layers expect: $BatchSize$ x $NumberOfFeatures$.
- **Linear Layer 1:** $InputFeatures$: 1024, $OutputFeatures$: 512, *ReLU* activation.
- **Linear Layer 2:** $InputFeatures$: 512, $OutputFeatures$: 512, *ReLU* activation.
- **Dropout :** 50% .
- **Linear Layer 3:** $InputFeatures$: 512, $OutputFeatures$: 100 (number of classes).

Now we will explain our motives behind designing the model in this way:
Firstly a big change from LeNet-5 is the addition of BatchNorm. Batch normalization is a technique used in neural networks to normalize the input of a layer by adjusting and scaling the activations. It helps in speeding up the training, allows for higher learning rates, improves gradient flow, and makes the network more robust and stable, enabling the training of deeper networks.
Our second change is the introduction of a big stride in the first Conv Block (so called Stem). By using $stride = 3$ we reduce the resolution of the image 3 times after going through this layer. This way the CIFAR-100 image that we upscaled to 96 x 96 (the resolution of STL-10 images) after just 1 layer is returned to its original size. This way we are unifying in a better way what the model "sees" during the pretraining with what it will "see" in later in the finetuning. This initial downsampling also allows for the extraction of low-level features while enabling deeper layers of the network to focus on more abstract and complex features.
We experimented a bit with the number of channels following the main practice of using powers of 2 and decided that these ones work well.
We explained above the choice of Global Average Pooling after the last convolution. So the final big change we introduced is the addition of Dropout. Dropout is a regularization technique used in neural networks to prevent overfitting by randomly deactivating a subset of neurons during training. This encourages the network to develop redundant pathways, improving its ability to generalize from the training data to unseen data. We tried adding a Dropout Layer also

after the First Linear Layer but this dropped our test accuracy so we removed it. Now the learning curve actually starts to resemble overfitting (trains grows fast, test doesn't seem to improve) but in fact this is not the case. In order to be in the overfit setting not only the train needs to reach really high values (as in our case) but test accuracy needs to **drop** which is certainly not the case with us. When observed closely (you can refer to the notebook for the numbers) the test accuracy also continues to grow! We must admit that with one more dropout (that we just mentioned) the train accuracy stays close to the test accuracy and doesn't reach such huge values. However, in the same training setting with two dropout layers we achieve 70% test accuracy whereas without the first drpout we get 72%.

The results we observe with the above-explained network are the following: **Final Train accuracy: 90.548% ; Final Test accuracy: 72.1%.** This is an enormous improvement over LeNet-5: Test accuracy **improved with 29% - from 43.12% to 72.1%**.

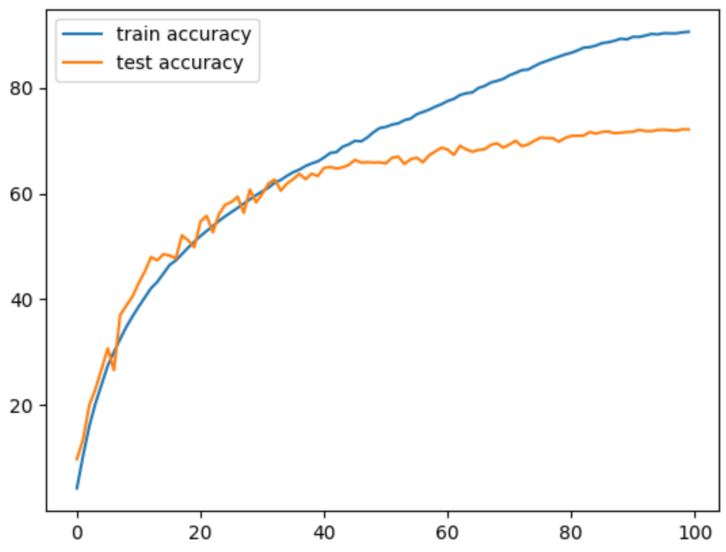

Figure 5: Improved ConvNet Accuracy (epochs/accuracy %)

2 Part 2 - Finetuning for STL-10 Dataset

2.1 Dataset

The STL-10 Dataset has much less images than CIFAR-100 so if a model is trained from scratch on it it is most likely to overfit on the training data or just will not reach the same accuracy as a model that was pretrained on a larger dataset (such as CIFAR-100).The Dataset consists of 10 classes from which we have to use just 5 - 'car', 'deer', 'horse', 'monkey', 'truck'. These classes are not in the folders from 1 to 5 so we accounted for that in the DataSet class in the notebook. The images have shapes of 96 x 96 x 3 as mentioned above.

2.2 Finetuning

In order for our ConvNet to work for 5 classes we have to replace the final linear layer of the network which has 100 output nodes with a layer with just 5 output nodes. Besides that we keep the same training procedure we did for CIFAR-100 (same augmentations, same learning rate, same Scheduler, same number of epochs, same batch size). We experimented mostly with dropping the learning rate but this worsened our test score. Also we tried without a scheduler or with a scheduler set up differently but once again we got lower accuracy and in general the worse training. Also with different training setups not only our final accuracy was not good enough but also the accuracy after the first epoch was significantly lower (so it not only ends worse but starts worse). To give a concrete example whit the current setting after just 1 epoch we get $\tilde{7}0\%$ accuracy. With any other learning rate (higher or lower) the accuracy after the first epoch drops as low as 5%. This of course makes sense since the final layer is initialized at random and with a very small learning rate the weight need much longer to reach good values. Because of this observations we concluded that we managed to find (almost) optimal setting and it just happened to be the same as the one used for the pretraining of the ConvNet. The results we observed in the end are the following:
Final Train accuracy: 98.84% ; Final Test accuracy: 89.55%

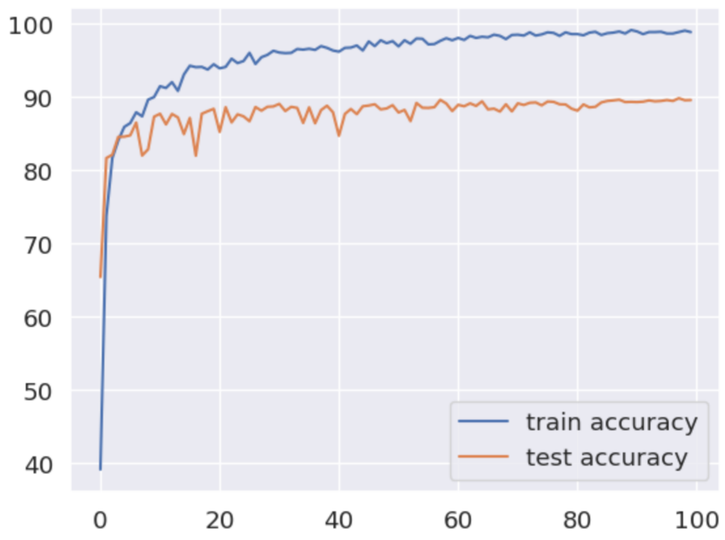

Figure 6: ConvNet Accuracy on STL-10 Dataset (epochs/accuracy %)

Similarly to our explanation above for the CIFAR-100 Dataset here we are not overfitting the data! The test accuracy throughout the whole training increases. Here however this is more surprising since we use the same learning rates and train for the same amount of epochs as in the pretraining. Having much less training examples we assumed that at some point our test accuracy will start to diminish because of that. But in reality our model almost perfectly learns the train data without loosing its ability to generalize!

The results per class look as follows:

Class	Accuracy (%)
Car	**91%**
Deer	**89%**
Horse	**88%**
Monkey	**83%**
Truck	**93%**

Table 1: Accuracy per class