University of Amsterdam

Computer Vision 1

Lab Project - Part 2

October 23, 2023

Students:
Fien De Kok
12672289

Jakub Tomaszewski
15178331

Joan Velja
14950480

Group:
G4

1 Introduction

Image classification is a cornerstone task in the field of computer vision and has wide-ranging applications that span from medical imaging to autonomous navigation systems. The problem is typically formulated as a supervised learning task, where the goal is to learn a function $f : \mathcal{X} \to \mathcal{Y}$ from a set of labeled examples, forming the pairs $\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$. $\mathcal{X}$ represents the input space, which in our context would be images represented as multi-dimensional arrays. $\mathcal{Y}$ represents the output label space, which is a set of classes that the algorithm aims to categorize input images into.

Mathematically, the objective of image classification can be thought of as an optimization problem, where the goal is to find the model parameters θ that minimize a certain loss function $\mathcal{L}$. In the case of neural networks, the model parameters θ consist of weights and biases, and the loss function is in our case the categorical cross-entropy loss given by

$$\mathcal{L}(\theta) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{C} y_{ij} \log(p_{ij}),$$

where N is the number of samples, C is the number of classes, y_{ij} is the true label, and p_{ij} is the predicted probability of sample i belonging to class j.

This report examines two neural network architectures: LeNet-5 and a 2-Layer Perceptron. LeNet-5, the cornerstone of convolutional neural networks, introduced pivotal concepts like spatial hierarchies and parameter sharing through its architecture of alternating convolutional and pooling layers. Designed originally for handwritten digit recognition, LeNet-5 employs a series of 5 layers, including three convolutional layers intervaled with average pooling layers and concluded by a fully connected block, utilizing tanh as activation functions. On the other hand, the 2-Layer Perceptron is a simpler neural architecture consisting of an input layer followed by one hidden layer and an output layer, using as an activation function ReLU.

To evaluate these architectures, we utilize the CIFAR-100 dataset, which is an extension of the simpler CIFAR-10 dataset. Comprising 60,000 color images with a resolution of 32×32 pixels, CIFAR-100 is categorized into 100 classes, with each class having 600 instances. Divided into 500 training images and 100 testing images per class, CIFAR-100 is a challenging array of objects and scenes, especially for such rudimentary architectures like the ones we are implementing in this experiment.

The remainder of this report offers a rigorous analysis of the aforementioned architectures, the experimental results of the implementations and some intuition behind the functioning of these models. Initially, we delineate the experimental methodology, elaborating on choices of loss functions, hyperparameters, and optimization algorithms. Subsequently, we scrutinize the performance of each model using evaluation metrics such as accuracy, F1-score, and confusion matrices. Following this first part of the analysis, the report aims at delving deeper into transfer learning in the context of retraining the Convolutional Network on another dataset, namely STL-10.

2 Part 1 - CIFAR-100

CIFAR-100 is a labeled subset of 80 million tiny images dataset where CIFAR stands for Canadian Institute For Advanced Research. The images were collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. The dataset consists of 60000 colored images (50000 training and 10000 test) of 32 $\times$ 32 pixels. CIFAR-100 has 100 classes containing 600 images each, where there are 500 training images and 100 testing images per class. The 100 classes in the CIFAR-100 are grouped into 20 superclasses. Each image comes with a ”fine” label (the class to which it belongs) and a ”coarse” label (the superclass to which it belongs). Some example of classes in CIFAR-100 (Krizhevsky 2009) are beaver, shark, roses, bowls, skyscraper, forest, mountain, lobster, man, woman, bus, tank and so on. As these few classes can tell already, the dataset is very eterogeneous, further underlying the complexity of the task at hand.

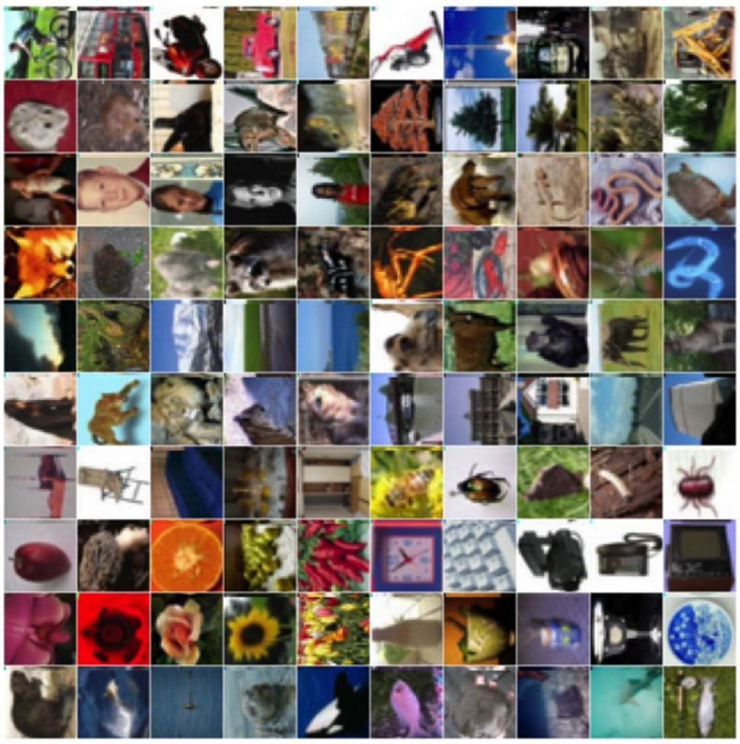


Figure 1: Sample images from the dataset

3 Architecture overview

3.1 2 Layer perceptron

The 2-Layer Perceptron, one of the simplest forms of artificial neural networks, is a feedforward neural network with a single hidden layer. Despite its simplicity, this model can still be used as a classifier for our task at hand.

Mathematically, a 2-Layer Perceptron aims to approximate a function $f : \mathbb{R}^d \to \mathbb{R}^k$ where d is the dimensionality of the input and k is the number of classes. Given an input $\mathbf{x} \in \mathbb{R}^d$, the output $\mathbf{y}$ can be calculated as:

$$\mathbf{y} = \sigma(W_2 \cdot \sigma(W_1 \cdot \mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2),$$

where W_1 and W_2 are the weight matrices, $\mathbf{b}_1$ and $\mathbf{b}_2$ are the bias vectors, and σ represents the activation function. Our implementation contains a 2LP having 1024 neurons in its hidden layer. We found this to be a good compromise between a lower number of neurons, which would fail to capture the intricacies of the dataset, and a larger number of neurons, which would lead to overfitting the training set.

While a 2-Layer Perceptron is computationally less intensive, it's often inadequate for complex image classification tasks. First, it lacks the ability to capture local and hierarchical patterns in images, feature that is instead present in CNNs, which are critical for understanding the semantics of complex objects and scenes.

3.2 LeNet-5

LeNet-5 (LeCun, Bottou, Bengio, & Haffner 1998), developed by Yann LeCun in 1998, is the first convolutional neural networks that laid the groundwork for the subsequent developments in the field of computer vision and deep learning. Despite being designed over two decades ago, we can still make use of it for our image classification task.

LeNet-5 employs convolutional layers to learn spatial hierarchies of features. The architecture consists of two sets of convolutional and average pooling layers, followed by a flattening convolutional layer, and two fully-connected layers. Specifically, it uses:

- Convolutional Layer (C1): 5×5 kernel, 6 filters;
- Average Pooling (S2)
- Convolutional Layer (C3): 5×5 kernel, 16 filters
- Average Pooling (S4)
- Convolutional Layer (C5): Fully connected, 120 units
- Fully-Connected Layer (F6): 84 units
- Output Layer: k units, based on the task at hand.

LeNet-5 originally employs tanh activation functions, although modern implementations often use the Rectified Linear Unit (ReLU) for better gradient propagation. For the sake of this experiment though, we will stick to the original implementation of the network, so tanh activations are enforced in our notebook.

LeNet-5 was originally designed for grayscale image classification tasks, particularly for handwritten digit recognition on the MNIST dataset. Its architecture is also too shallow to capture the complex features present in a more intricate, high-dimensional dataset like CIFAR-100. Moreover, it uses average pooling instead of max-pooling, which is generally less effective at capturing the most relevant features in an image, but since Max-Pooling methods were not implemented by LeCun himself, we decided to stick to the original architecture once again.

4 Experimental setting

We followed the training schedule originally proposed by Andrej Karpathy in a famous blogpost (Karpathy 2019), namely setting a (toy) baseline, then trying to overfit the data and finally applying some regularization to tame the overfitting. As a baseline for comparison purposes, we decided to run our experiments under equal settings, i.e.:

- **Batch size**: 64

- **Learning rate**: 0.0001
- **Optimizer**: SGD
- **Epochs**: 100

Under these assumptions, the two classifiers attain the following losses/accuracies:

Metric	2LP	LeNet-5
Testing Accuracy	0.1493	0.0259
Training Accuracy	0.2340	0.0259
Training Loss	3.2051	4.442

Table 1: Comparison of 2LP and LeNet-5 under the baseline assumptions, averaged over 15 runs

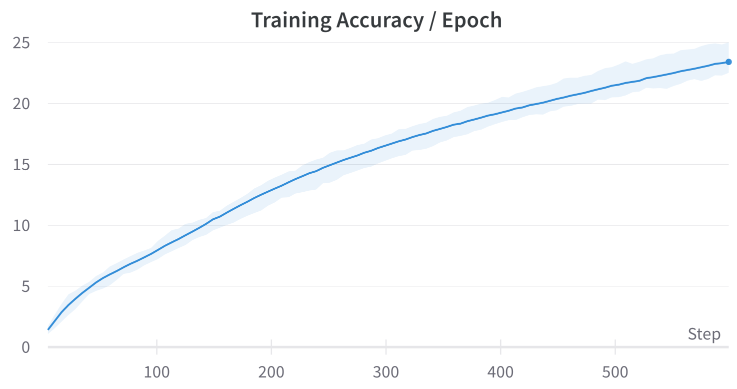


Figure 2: 2LP average training accuracy over 15 runs

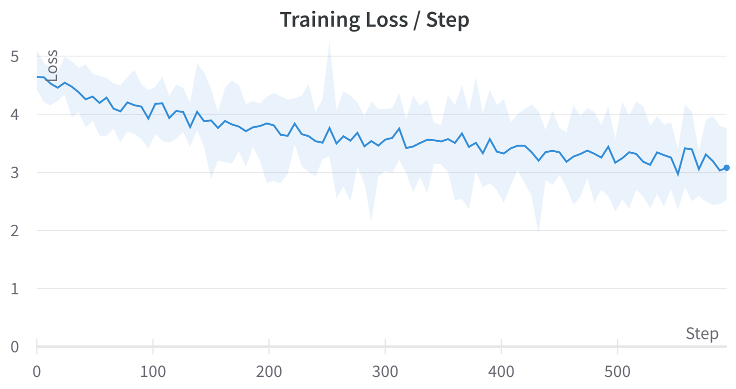


Figure 3: 2LP average training loss over 15 runs

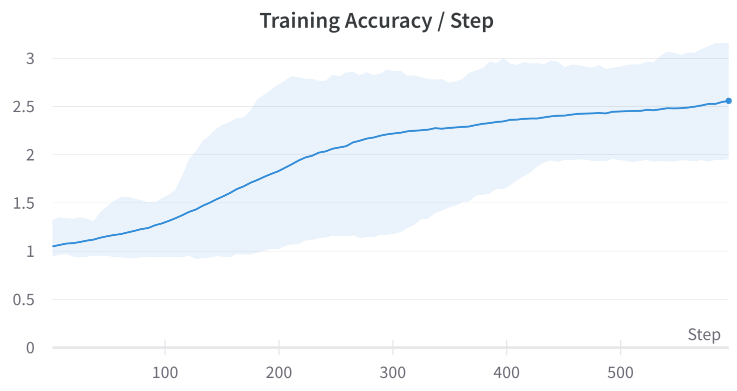


Figure 4: CNN average training accuracy over 15 runs

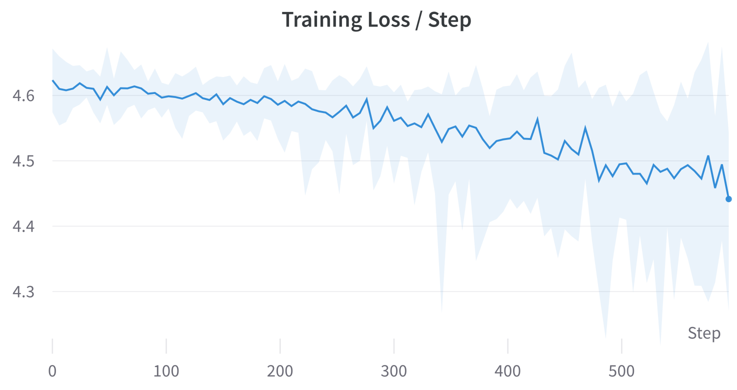


Figure 5: 2LP average training loss over 15 runs

From the above results, it's clear that some hyperparameter tuning in both architectures is needed, and especially to LeNet-5: it is in fact unthinkable that a feedforward neural network outperforms a CNN in an image classification task. This assumption is further confirmed by the training loss plot of LeNet-5, where we can clearly see that the learning rate is too small for this purpose. We will justify tweaks to hyperparameters such as the transform and Optimizer.

4.1 Experiment #1

In this part of the assignment, we kept the same architectures but made the following changes:

- customized `learning rates`. The reason for this is given by the fact that we noticed different behaviors by the two classifiers when training our baseline models: the training loss would go down much quicker for the 2LP, whereas the CNN would be much slower, hinting at a different approach when training;
- Addition of regularization. Since we aim at increasing the training accuracy/lower training loss, we have to take measures to avoid overfitting and preserve generalization. Because of this we implemented `gradient clipping` at 1 and `weight_decay` at 0.001. Gradient clipping helps prevent exploding gradients by rescaling them when they exceed a threshold, ensuring stable training, whereas Weight Decay is L2 Regularization of the weights of the network;
- Addition of transformation techniques;

- Addition of One-Cycle scheduler. Given the slow convergence emerged from the baseline experiments, we will implement One-Cycle learning rate scheduler (Smith & Topin 2017): the authors study an interesting approach for training neural networks that allows the speed of training to be increased by an order of magnitude. The basic approach is to perform a single, triangular learning rate cycle with a large maximum learning rate, then allow the learning rate to decay below the minimum value of this cycle at the end of training. In addition, the momentum is cycled in the opposite direction of the learning rate (typically in the range [0.85, 0.95]).

4.1.1 Experiment #1 for 2LP

Given the baseline results, we will try to lower the training error and then regularize to avoid overfitting using weight decay. We tried different approaches w.r.t. hyperparameter tuning:

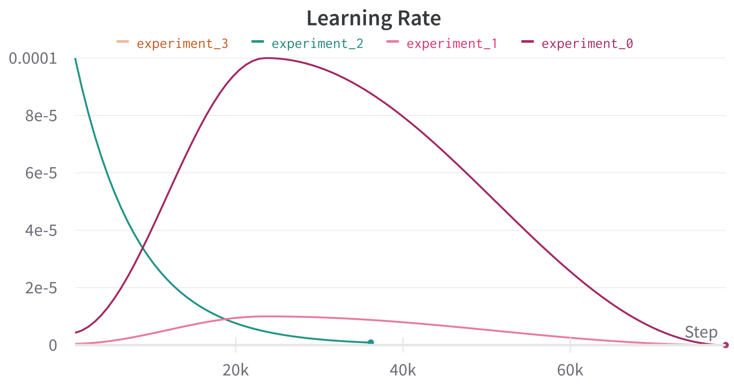


Figure 6: relevant learning rates tested

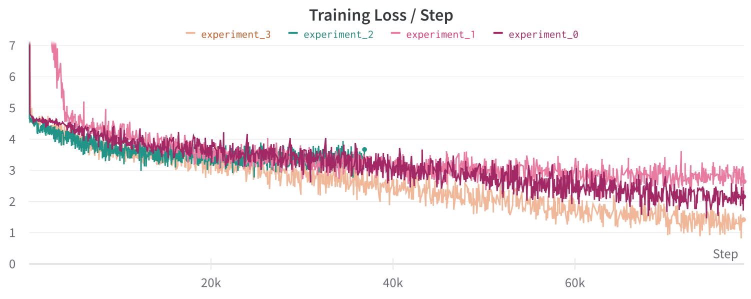


Figure 7: relevant training losses

As we said before, we changed the learning rate schedule. We tried schedulers such as cosine annealing, cycle schedule and exponential decay of the LR. We only show for the sake of the report the relevant ones, i.e., those who resulted in substantial increase in accuracy of the test set (1 Cycle scheduling) or those who surprisingly failed (exponential decay). The best performing 2LP was the one starting from learning rate equal to $4e - 6$, and reaching as a peak 0.0001. By doing this, we let the model explore the loss landscape in the beginning: the idea of setting an increasingly large learning rate in the early stages of training, aims to enable a more thorough exploration of the loss landscape. This way we avoid saddle points and local minima, since a larger learning rate can help the model to skip over shallow local minima or saddle points in the loss landscape. Moreover, an increasing LR enables faster convergence: initially, when the model is far from optimal, a larger learning rate speeds up the training process. On top of this, we implemented a new set of transformations, namely:

```
transforms.RandomHorizontalFlip(0.1),
    transforms.RandomRotation(20),
    transforms.RandomAdjustSharpness(sharpness_factor = 2, p = 0.1),
    transforms.ColorJitter(brightness = 0.1, contrast = 0.1, saturation = 0.1),
    transforms.Normalize((train mean), (train std)),
```

We found these transformation to be pretty common in online competitions based on CIFAR 100 and 10, thus we believed they could help with our task.

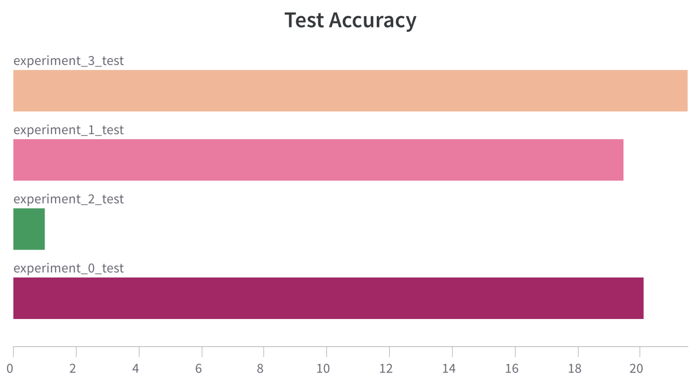


Figure 8: relevant training losses

The modifications allowed us to jump to a test accuracy of 21.51% and a loss of 1.35. It is a stark improvement from the non finetuned 2LP (+6% testing accuracy, halved training loss)!

4.1.2 Experiment #1 for ConvNet

Similarly to what we did to the 2LP, we leverage 1 Cycle LR and transformations for LeNet-5 given the poor performance obtained by the baseline model. Given the training results, we thought about increasing the learning rate along with the regularization techniques mentioned in the previous subsection. We believe that the lack of generalization of LeNet-5 in the baseline scenario was given by a low LR that would get stuck in (sharp) local minimas. We once again tried different schedulers and combinations of batch size and learning rates, but we got satisfying results only with the combination we will describe below. This set of modifications yielded an sharp increase in training accuracy and then testing accuracy too:

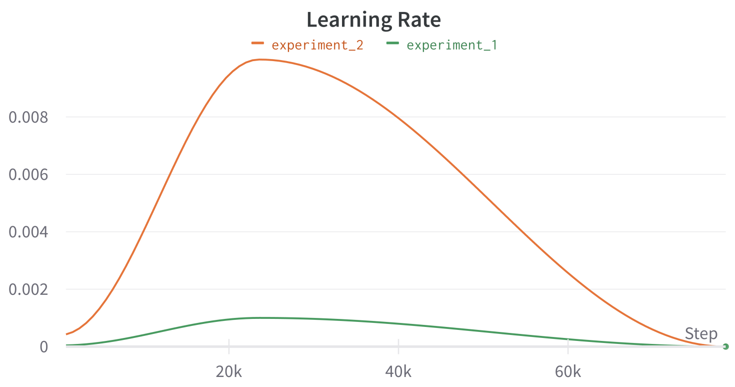


Figure 9: relevant learning rates

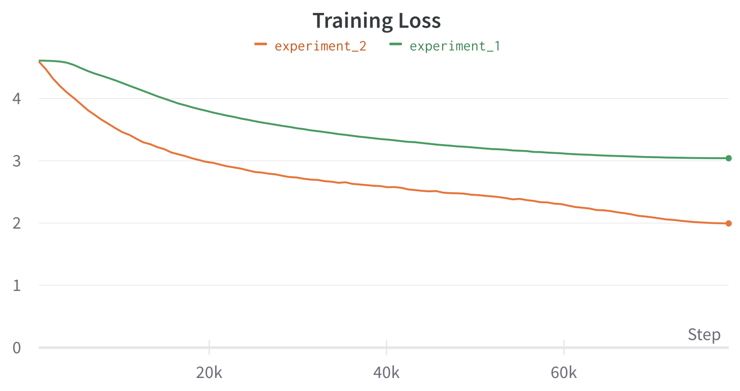


Figure 10: relevant learning rates

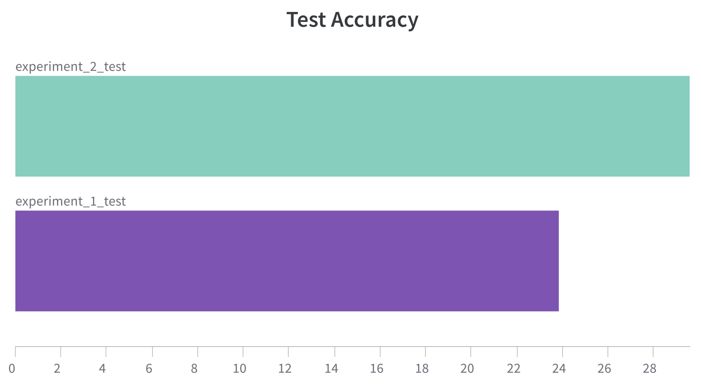


Figure 11: relevant learning rates

As we can see, a 1 Cycle schedule that starts at $2e-4$ and peaks at 0.01 is the one that performs best: we thought that the loss landscape for the CNN had lots of sharp local minimas, so we increased our peaks so to "jump" them. This way we attained a testing accuracy of 29.7%, a stark increase from the dummy baseline of around 2%!

4.2 Experiment #2

In this part of the assignment we will try adding two layers to each of the two architectures given the results of the above section.

4.2.1 Experiment #2 for 2LP

We decided to add one fully connected layer to increase the power of the Neural Network and a dropout layer to prevent overfitting. These modifications yielded an increase of 4% on the test accuracy, which shoots up at 26.82%.

4.2.2 Experiment #2 for ConvNet

We decided to add another fully connected layer to better classify the features coming from the convolution blocks. In order to train faster and not to get stuck in local minima, we added a Batch Normalization Layer after the convolution block. This strategy did not prove to be successful, as the model would start overfitting the training set resulting in a decrease of the test accuracy at around 27%. Even by trying to add another convolution to the architecture without

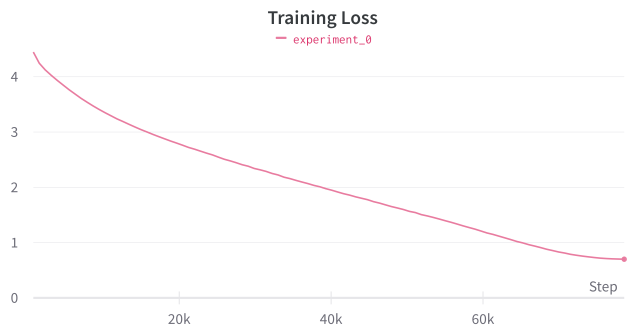


Figure 12: 3LP + Dropout loss

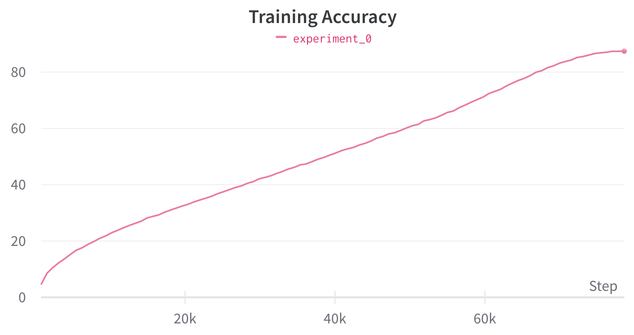


Figure 13: 3LP + Dropout training accuracy

altering the overall structure, the results were not satisfying: the final output of the convolution block would end up being very small dimensionally, thus not being able to convey much information about the image itself. We break down some of the conclusions that we draw from this experiment below.

5 Results - Part 1

The results above confirm our prior knowledge on the given architectures. CNNs are expected to be better than Feedforward Neural Networks. Although we had some trouble fitting the model to the training data with the CNN model, we still achieved a slightly higher accuracy on the final test set. We assume these issues to arise from the fact that, as we mentioned in the beginning, LeNet-5 is very outdated and would require a lot of tweaks within the original architecture itself to perform well on complex data such as CIFAR-100: to begin, we would at least need to change all tanh activations to state-of-the-art functions such as ReLU or derivatives. Moreover, stacking small sized convolutions has proven in recent literature to be very effective for feature extraction, whereas LeNet-5 only implements 2 as we saw. Nonetheless, it still outperforms Feedforward neural nets, leveraging the ability to learn spatial features from the image through convolutions.

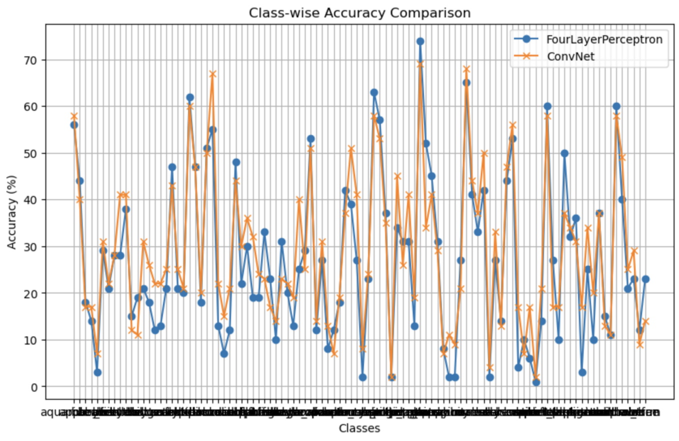
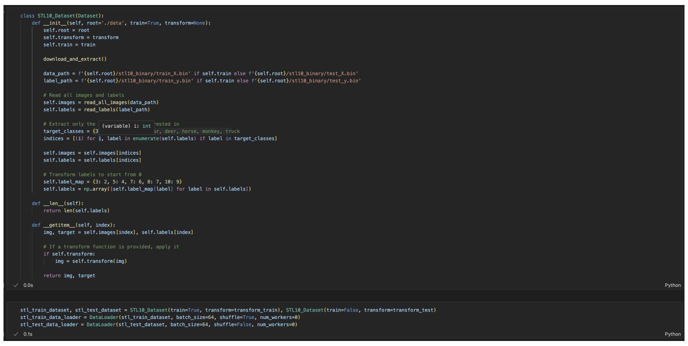


Figure 14: Class-wise comparison of accuracy on test set

6 Part 2: STL-10

The STL-10 dataset is an image recognition dataset for developing unsupervised feature learning, deep learning, self-taught learning algorithms (Coates, Ng & Lee 2011). It is inspired by the CIFAR-10 dataset but with some modifications: each class has fewer labeled training examples than in CIFAR-10 and STL-10 images have a higher resolution (96x96 pixels) compared to CIFAR-10 or CIFAR-100 images (32x32 pixels). Moreover, STL-10 has 10 classes (airplane, bird, car, cat, deer, dog, horse, monkey, ship, truck), from which we extract only 5 (car, deer, horse, monkey, truck) with 500 training images and 800 test images per class. This is a more limited set of categories compared to CIFAR-100, and a more limited number of training datapoints. Finally, the images in STL-10 are, as we said, of higher quality and more realistic, whereas CIFAR images are low-res and are sometimes be hard to recognize, even by a human.

We created the dataset for STL-5 based on the suggesitions provided in the notebook. We then loaded the pretrained parameters from ConvNet (LeNet-5) applying the following modifications:



```python
class STL10_Dataset(Dataset):
    def __init__(self, root='./data', train=True, transform=None):
        self.root = root
        self.transform = transform
        self.train = train

        download_and_extract()

        data_path = f'{self.root}/stl10_binary/train_X.bin' if self.train else f'{self.root}/stl10_binary/test_X.bin'
        label_path = f'{self.root}/stl10_binary/train_y.bin' if self.train else f'{self.root}/stl10_binary/test_y.bin'

        # Read all images and labels
        self.images = read_all_images(data_path)
        self.labels = read_labels(label_path)

        # Extract only the (variable) i: int ested in
        target_classes = {3 ar, deer, horse, monkey, truck
        indices = [(i) for i, label in enumerate(self.labels) if label in target_classes]

        self.images = self.images[indices]
        self.labels = self.labels[indices]

        # Transform labels to start from 0
        self.label_map = {3: 2, 5: 4, 7: 6, 8: 7, 10: 9}
        self.labels = np.array([self.label_map[label] for label in self.labels])

    def __len__(self):
        return len(self.labels)

    def __getitem__(self, index):
        img, target = self.images[index], self.labels[index]

        # If a transform function is provided, apply it
        if self.transform:
            img = self.transform(img)

        return img, target
```

✓ 0.0s Python

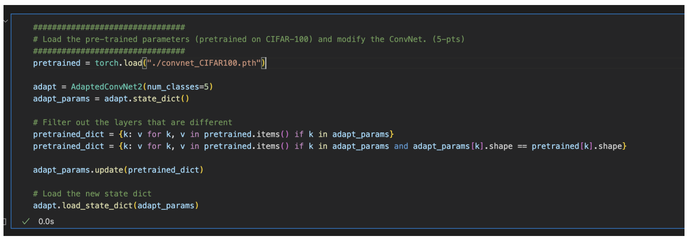
```python
stl_train_dataset, stl_test_dataset = STL10_Dataset(train=True, transform=transform_train), STL10_Dataset(train=False, transform=transform_test)
stl_train_data_loader = DataLoader(stl_train_dataset, batch_size=64, shuffle=True, num_workers=0)
stl_test_data_loader = DataLoader(stl_test_dataset, batch_size=64, shuffle=False, num_workers=0)
```

✓ 0.1s Python

Figure 15: Data loading

- Addressing different input size;
- Addressing different categorization (100 to 5 classes).

From here, we proceeded by loading the weights of the layers which stayed the same from one model to the other (namely the convolutions and two fully connected layers) through the following script:



```
################################
# Load the pre-trained parameters (pretrained on CIFAR-100) and modify the ConvNet. (5-pts)
################################
pretrained = torch.load("./convnet_CIFAR100.pth")

adapt = AdaptedConvNet2(num_classes=5)
adapt_params = adapt.state_dict()

# Filter out the layers that are different
pretrained_dict = {k: v for k, v in pretrained.items() if k in adapt_params}
pretrained_dict = {k: v for k, v in pretrained.items() if k in adapt_params and adapt_params[k].shape == pretrained[k].shape}

adapt_params.update(pretrained_dict)

# Load the new state dict
adapt.load_state_dict(adapt_params)
```
✓ 0.0s

Figure 16: Weights loading

The results of the transfer learning training and subsequent testing on the test-set of STL-5 can be found at the following WandB https://api.wandb.ai/links/1000kelvin/6llv4l2s.

7 Bibliography

Krizhevsky, A. (2009). Learning Multiple Layers of Features from Tiny Images. 32–33.

LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-Based Learning Applied to Document Recognition. Proceedings of the IEEE, 86, 2278–2324. Retrieved from http://citeseerx.ist.psu.edu/view

Karpathy, A. (2019). A recipe for training neural networks. Andrej Karpathy Blog. Retrieved from http://karpathy.github.io/2019/04/25/recipe/

Smith, L. N., & Topin, N. (2017). Super-Convergence: Very Fast Training of Residual Networks Using Large Learning Rates. CoRR, abs/1708.07120. Retrieved from http://arxiv.org/abs/1708.07120

Coates, A., Ng, A., & Lee, H. (2011). An Analysis of Single-Layer Networks in Unsupervised Feature Learning. In G. Gordon, D. Dunson, & M. Dudík (Eds.), Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (pp. 215–223). Retrieved from https://proceedings.mlr.press/v15/coates11a.html