CV1 Final Project Part2

Luis Zerkowski, Tanya Kaintura, Jelle van der Lee

October 21, 2023

Q1.1 CIFAR-100

The CIFAR-100 dataset contains 100 classes which can be combines as 20 superclasses. The dataset is divided into train and test set. the trainset has 50000 data points and test has 10000 data points. The imshow function unnormalizes and displays images, making it easier to visualize the dataset. The code iterates through the training data, collecting images for classes with labels from 0 to 4 as shown in Fig 1

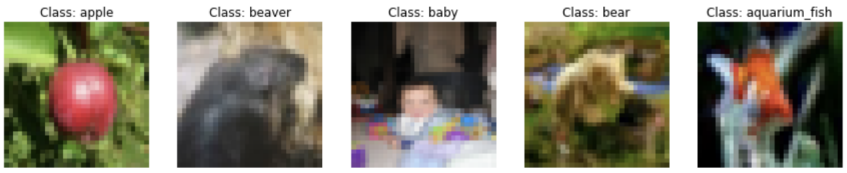

Figure 1: Visualization for classes 0-4

Q1.2 Architecture understanding

The TwoLayerNet class defines a neural network with two fully connected layers, transforming input data from size 3x32x32 to the number of output classes(100) with Relu activation. For ConvNet class, it follows a structure similar to LeNet-5 as shown in Fig 2. LeNet was groundbreaking in its time and featured several innovations, including the use of the tanh activation function in its neurons. The tanh activation function squashes the neuron outputs into a range between -1 and 1, which was well-suited for the symmetric nature of handwritten digit data. However, when adapting the architecture we changed the tanh to Relu activation. It is computationally more efficient due to its simple thresholding operation, mitigating the vanishing gradient problem. Also our architecture contain 3 channels.

The choice of an odd kernel size 5x5 in convolutional layers is a common practice in CNNs. It makes sure the output feature map will have the same spatial dimensions as the input

feature map (with appropriate padding). A larger kernel size allows the convolutional layer to capture a larger portion of the input image, which can be useful for learning complex features. There is no direct relationship between the number of channels and the kernel size; they serve different purposes in the architecture of a CNN. In F6, the input size is 120 and output is 84. Therefore, the total number of parameters in F6 (self.fc2) is: 120 (input size) x 84 (output size) + 1(biases)*84 = 10164 parameters.

```
ConvNet(
  (conv1): Conv2d(3, 6, kernel_size=(5, 5), stride=(1, 1))
  (pool1): AvgPool2d(kernel_size=2, stride=2, padding=0)
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (pool2): AvgPool2d(kernel_size=2, stride=2, padding=0)
  (fc1): Linear(in_features=400, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=100, bias=True)
)
```

Figure 2: ConvNet architecture

Q1.3 Preparation of training

```python
class CIFAR100_loader(torch.utils.data.Dataset):
    def __init__(self, root, train=True, transform=None):
        '''CIFAR-100 dataset loader'''
        ################################
        self.data = torchvision.datasets.CIFAR100(
            root=root, train=train, download=True, transform=transform)
        ################################

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

    def __getitem__(self, item):
        ################################
        img, target = self.data[item]
        ################################
        return img, target
```

Figure 3: CIFAR100 loader

```python
# Define data transformations for training and testing
transform_train = transforms.Compose([
    transforms.RandomHorizontalFlip(0.5),
    transforms.RandomVerticalFlip(0.2),
    transforms.RandomRotation(30),
    transforms.RandomAdjustSharpness(0.4),
    transforms.RandomCrop(32, padding=4),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
])

transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
])

def initialize_optimizer(model, learning_rate):
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)
    return optimizer
```

Figure 4: Tranform and Optimizer

In figure 3, the CIFAR100_loader class initializes a CIFAR-100 dataset loader, defining the dataset's parameters in the init method and retrieving specific items, consisting of images and targets, in the getitem method.

As you see in figure 4, there are several data transformation used. The data transformations applied to the images serve various purposes. For training data, transformations like random horizontal and vertical flips, random rotation, and random sharpening are used to augment the dataset. These augmentations introduce diversity in the training set, making the model more robust to different image variations and orientations. Random cropping with padding helps the model learn from various parts of the image. Additionally, normalization standardizes pixel values to a mean of 0.5 and a standard deviation of 0.5, ensuring numerical stability and faster convergence during training. For testing data, transformations include

only normalization. The choice of the Adam optimizer is based on its efficiency and popularity. Adam combines the advantages of two other popular optimizers, AdaGrad and RMSProp, providing adaptive learning rates for each parameter. This adaptability often leads to faster convergence and efficient training, making it a suitable choice for deep learning models.

Implementation Approach

For *Q1.3* we have to first complement the CIFAR100_loader, Transform function and the Optimizer and then train the *TwolayerNet* and *ConvNet* with these implemented functions. In the train method we have used Cross-Entropy Loss (or Logarithmic Loss) as it measures the dissimilarity between predicted class probabilities and the true class labels. Then we initialize data loaders for the CIFAR-100 dataset for training and testing with specified transformations, 64 batch size. The models are trained for 10 epochs with a learning rate of 0.001, and its accuracy is evaluated on the test set. The initialize_optimizer, train, and valid functions are used to handle training and evaluation. The *TwolayerNet* uses a hidden size of 64.

Results Analysis

We tested both of the models under the same setting. In a comprehensive analysis of the results for the TwoLayerNet and ConvNet models on the CIFAR-100 dataset, distinct performance patterns emerge. The TwoLayerNet shows a gradual reduction in training loss from 4.162 to 3.649 over 10 epochs, indicating a learning process, albeit at a slow pace. However, its test accuracy is relatively low at **11%**, suggesting challenges in generalizing to unseen data, and class-wise accuracy exhibits significant variation. In contrast, the ConvNet model also demonstrates a decline in training loss from 4.162 to 3.328, highlighting effective learning throughout the training period. Notably, the ConvNet exhibits superior generalization, achieving a test accuracy of **22%**, outperforming the TwoLayerNet. Nevertheless, class-wise accuracy varies alot. The architecture plays a pivotal role in distinguishing their performance, with the ConvNet, benefiting from convolutional layers, exhibiting a marked advantage in capturing intricate image features and spatial hierarchies. This architectural disparity contributes significantly to the ConvNet's better test accuracy, demonstrating its adeptness at generalizing across diverse data. Comparing the class-wise accuracy, we can see that the ConvNet outperforms the TwoLayerNet for most of the classes. The ConvNet achieves higher accuracy for classes like "plain," "lawn-mower," "orange," and "skunk," among others.

Q1.4 Setting up the hyperparameters

Architecture Implementation

In Q1.4 to achieve better accuracy we have changed the architecture of ConvNet and TwolayerNet as shown in Figure 5 and 6 respectively.

```
TwoLayerNet(
  (fc1): Linear(in_features=3072, out_features=128, bias=True)
  (fc2): Linear(in_features=128, out_features=128, bias=True)
  (fc3): Linear(in_features=128, out_features=128, bias=True)
  (fc4): Linear(in_features=128, out_features=100, bias=True)
)
```

Figure 5: Updated TwolayerNet architecture

In TwolayerNet, we have added 2 more linear layers as it can significantly enhance its accuracy for a variety of reasons. Firstly, these added layers augment the network's capacity, enabling it to capture and model intricate patterns and relationships within the data. Such complexity is especially beneficial when dealing with datasets that exhibit hierarchical and multifaceted features. Each new layer introduces non-linear activation functions, such as Rectified Linear Units (ReLU), which are crucial for modeling complex, non-linear relationships in the data. As these non-linear transformations accumulate, the network gains more expressive power, resulting in more accurate representations. These additional layers also play a role in implicit regularization, guarding against overfitting by learning robust, abstract features that minimize noise interference. It is essential to note that while adding layers can be beneficial, their effectiveness depends on proper weight initialization, hyperparameter tuning, and the characteristics of the training dataset. The depth of the architecture should be carefully considered, as an excessive number of layers could lead to overfitting, and deeper networks may demand greater computational resources.

```
ConvNet(
  (conv1): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (pool): AvgPool2d(kernel_size=2, stride=2, padding=0)
  (fc1): Linear(in_features=8192, out_features=512, bias=True)
  (fc2): Linear(in_features=512, out_features=100, bias=True)
)
```

Figure 6: Updated ConvNet architecture

As shown in the Figure 6, adding BatchNorm layers and having only 1 average pooling layer instead of 2 max pooling layers could help improving the accuracy. Batch normalization (nn.BatchNorm2d) inserted after each convolutional layer helps stabilize the training process. It normalizes the inputs to each layer, reducing internal covariate shift. This normalization

leads to smoother and faster convergence during training. The use of average pooling retains more information compared to max pooling, which can be more aggressive in downsampling. This can be especially beneficial when capturing fine details and features in the data. Moreover, the reduction of pooling layers to one in the first ConvNet allows the network to retain spatial information, enabling it to learn more detailed features and achieve better accuracy. A deeper network can capture more complex and hierarchical features, which is especially beneficial for classifying 100 different classes. We didn't change the activation function and are still using Relu as explained above.

Also tried other architecture for Convnet with 2 convolutions with kernel 5, 2 batch norm layer and 2 max pooling layers. Although it didn't yield good accuracy. Multiple pooling layers, especially with max pooling, can lead to a significant reduction in the size of feature maps. When this reduction is excessive, it may lead to a loss of essential information for accurate classification. It can even lead to vanishing gradient problem. When maxpooling is combined with BatchNorm, it may affect the diversity of the learned features in the network.

Results and Hyperparameterization

In *Q1.4* our goal is to find such hyperparameters for the functions of ConvNet and TwolayerNet, that we reach the accuracy of the models as high as we could. The final accuracy we have reached is 56% for ConvNet and 22% for TwolayerNet. In order to reach this accuracy we have not only modified the Dataloader, transform and Optimizer functions, but also the architectures of the two Nets. The hyperparameters and additions to the architectures that resulted in the highest results are shown in the Table 1 and Figure 6.

Table 1: Best Hyper-parameters ConvNet

Parameter	**Setting**
Transforms	Random Horizontal Flip, Random Crop, Color Jitter, Tensor, and Normalization
Optimizer	SGD with Momentum and Weight Decay
Batch Size	128
Epochs	50
Learning Rate	0.01

Explain what did not work and why the best params mentioned did work?: Next to the best working parameters, we have also tried multiple different additions to the model that resulted in a lower accuracy than the final result. We tried different things mentioned below:
1. Different Optimizers (Adam, RMS, Adagrad): Alternative optimizers were less effective in

finding the global optima. SGD's momentum and weight decay offered better trade-offs, allowing the model to escape local optima and generalize better. SGD with momentum and weight decay is a robust optimizer for training deep neural networks. It offers a compromise between escaping local optima and avoiding divergence. The momentum allows the model to move more decisively in directions that lead to lower loss, while weight decay helps prevent overfitting.
2. Too High or low Learning Rates (e.g., 0.5, 0.1, 0.001): Excessive learning rates led to instability in the training process. Very high learning rates caused the loss to diverge, while too low rates led to slow convergence. A rate of 0.01 was optimal in balancing rapid convergence and stability.
3. Too Many Transformations4: Overly complex data augmentations can introduce noise and hinder the model's ability to learn. The best results were achieved with a balanced set of transformations that provided valuable diversity without overcomplicating the training process. The combination of transformations, including random horizontal flip, random crop, color jitter, tensor transformation, and normalization, provides a diverse and augmented dataset. These augmentations help the model generalize better by exposing it to variations in the training data.
4. Epochs : With 20 epochs the model was still converging so we then tested to 50 epochs.

Table 2: Best Hyper-parameters TwolayerNet

Parameter	**Setting**
Transforms	Random Horizontal Flip, Random Crop, Color Jitter, Tensor, and Normalization
Optimizer	Adam with weight decay
Batch Size	128
Epochs	20
Learning Rate	0.001
hidden size	64

Explain what did not work and why the best params mentioned did work?: We tested multiple things to increase teh accuracy more, we managed to get it up from 11% to 22%. Different things we tried are mentione below:
1. Batch size : it was increased from 64 to 128. A batch size of 64 may not have performed as well as 128 because it provides a more noisy gradient estimate due to its smaller sample size, potentially leading to slower convergence and a higher risk of getting stuck in suboptimal local minima. In contrast, a batch size of 128 offers a more stable gradient estimate, which can lead to faster convergence and better generalization during training, resulting in improved model

performance.
2. Hidden size: Increasing the hidden size to 512 did not improved accuracy because it introduced excessive model capacity, leading to overfitting. With a large hidden size, the model can memorize the training data rather than learning meaningful patterns, which can severely hinder its ability to generalize to unseen examples, resulting in poor test performance. To remove the overfitting we did weight decay in Adam optimizer but best accuracy we could get was 15%.
3. Optimizer : For this network adam with weight decay worked better than SGD. Adam is known for its adaptive learning rates and often works well for a wide range of problems. SGD with weight decay may require careful tuning of the learning rate and other hyperparameters to achieve good results.
4. Epochs : Higher number of epochs also did not help much the accuracy was alsmost same fro 20 epochs and 50 epochs. The model might have already converged to its optimal or near-optimal performance within the first 20 epochs. Additional epochs do not bring substantial improvements because the model has learned most of what it can from the available data.
5. Learnign rate : Higher learning rate did not perform well. A high learning rate can lead to training problems, such as divergence, where the model's updates are too large and it fails to converge; overshooting, causing instability; and difficulty in learning, where it overlooks subtle improvements in the loss landscape, hindering parameter fine-tuning. This makes a lower, more controlled learning rate like 0.001 a safer choice for many training scenarios.

Compare and explain the differences of these two networks regarding the architecture, performances, and learning rates.
"TwolayerNet" and "ConvNet," each designed to handle image data but with different levels of complexity. TwolayerNet employs a relatively straightforward architecture with only linear layers, relying on fully connected layers and ReLU activation functions for feature extraction and class prediction. In contrast, ConvNet adopts a more intricate architecture better suited for image data. It integrates convolutional layers, batch normalization, and average pooling, which enable it to capture spatial and hierarchical features within images.

In terms of performance, ConvNet significantly outperforms TwolayerNet, achieving an accuracy of 56% compared to TwolayerNet's 22%. This performance discrepancy can be attributed to the architectural differences. ConvNet's convolutional layers allow it to capture intricate and spatial features present in image data, making it more effective in classifying a large number of classes. Additionally, the inclusion of batch normalization stabilizes the training process, contributing to better accuracy. Regarding learning rates, both networks displayed a sensitivity to this hyperparameter. In the case of ConvNet, the best learning rate was found to be 0.01, offering a balanced trade-off between rapid convergence and training stability. In

contrast, TwolayerNet demonstrated optimal performance with a lower learning rate of 0.001. This discrepancy can be attributed to the complexity of the architectures. ConvNet's intricate structure benefitted from a moderately higher learning rate, while TwolayerNet's simplicity required a lower learning rate to ensure stable training.

In conclusion, the architectural differences between TwolayerNet and ConvNet led to distinct levels of performance, with ConvNet outperforming its counterpart due to its ability to capture complex spatial features in image data. The choice of learning rate, closely tied to network complexity, played a vital role in achieving stable and efficient training for both networks. Lower learning rates were preferable for simpler architectures like TwolayerNet, while a balanced learning rate was more suitable for the complex ConvNet, ultimately leading to improved accuracy.

Q2.1 Create the STL10_Dataset

Implementation Approach

The networks implemented in the preceding sections are trained on the CIFAR-100 dataset, consisting of images with 32x32x3 dimensions. The STL10 dataset contains images of 96x96x3 dimensions and we start by correctly downloading and processing the dataset. In *Q2.1* we compliment STL10_dataset by filtering the data for the five specific classes from 0-4 (indexing error if we start from 1-5) {0: 'car', 1:'deer', 2:'horse', 3:'monkey', 4:'truck'} and implements the essential methods __len__ and __getitem__ to provide the dataset's length and retrieve individual items while applying transformations if specified.

Q2.2 Finetuning from ConvNet

For fine-tuning ConvNet on the dataset of STL10, we first loaded our model and parameters that were pre-trained on the dataset CIFAR-100. Based on this we first modified the output layer to have 5 classes and second last layer with size (73728, 512) to handle the difference in input size. Then we defined the parameters, from which the optimal results were received by using the parameters as seen in table 3. In *Q2.2* and *Q3* we are trying to get the highest accuracy on the test set by fine-tuning our hyper-parameters for the function of ConvNet. The highest final accuracy we recorded was 82.58% for the epochs of 20 and 82.67% for the epochs value of 60. While the accuracies are very similar, we concluded that the model is able to converge at 20 epochs and running this amount is sufficient. For the train accuracies we found 98.60% for 20 epochs and 99.56% for 60 epochs. Our optimal hyper-parameters and final accuracy are presented in table 3.

Table 3: Final Accuracy and the associated best Hyper-parameters from ConvNet

Final Accuracy	82.58%
Parameter	**Setting**
Transforms	Random Horizontal Flip, Random Rotation, Tensor, and Normalization
Optimizer	SGD with Momentum (0.9) and Weight Decay ($1e^{-4}$)
Batch Size	128
Epochs	20
Learning Rate	0.01

The parameters from table 3 are based on the optimal parameters found in *Q1.4*. We have used the same parameters as from table 1, with a small change in the Transforms parameters for even better optimization.

Results

For each set of parameters we trained the model to find the highest accuracy on our test data. In figure 7 we show the Accuracy and Loss per epoch during the training of our model, for both 20 and 60 epochs. As seen in the figure Test and Train accuracy grows per epoch and the Training loss decreases per epoch where the Test loss stays around the same value.

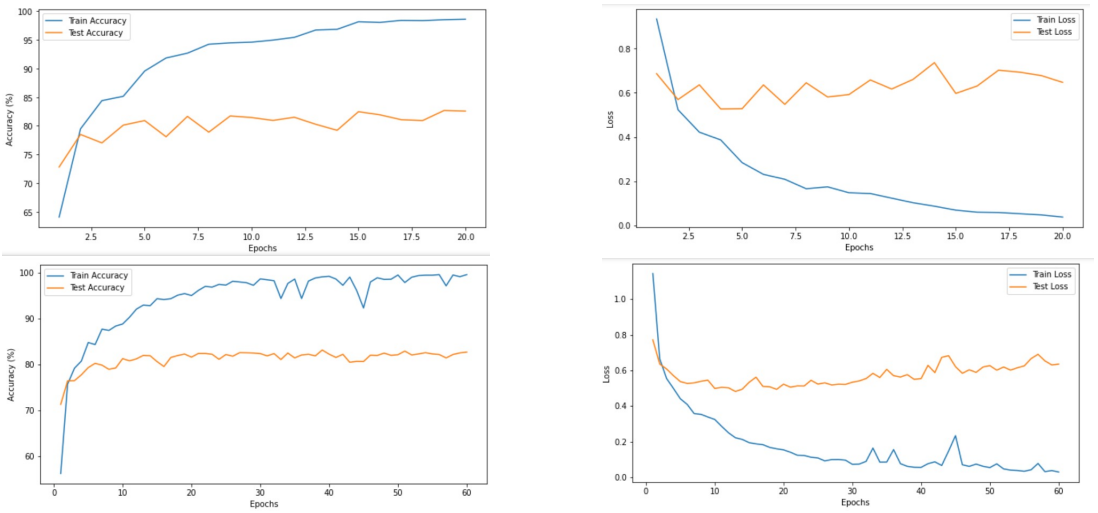

Figure 7: Accuracy and Loss per epoch

In order to create an understanding of the feature space, we have implemented the dimensionality reduction method of *t-SNE* to visualize the feature space in figure 8. This is done by using the pre-defined TSNE function and reducing our feature vectors to the dimension of 2D. The resulting figure shows clusters where the images represented in each cluster contain similar features. As we can see, our five different classes are correctly displayed as separate clusters in the feature space. However, there still exists some overlap between the classes which might make it difficult to correctly classify these images.

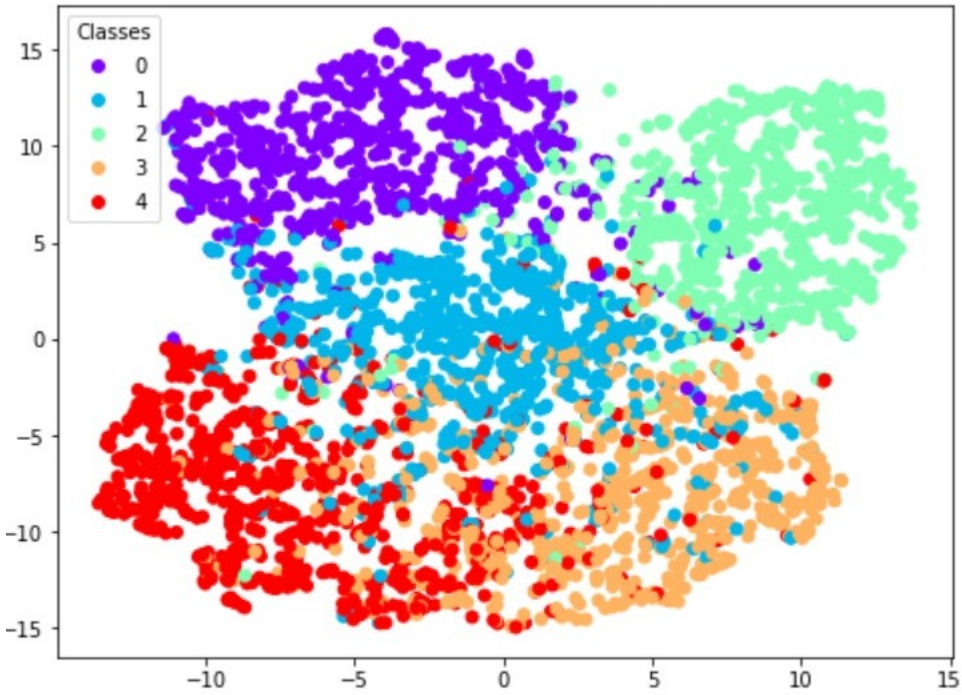

Figure 8: t-SNE Visualization of Feature Space

Q3 Bonus

In this section we tried multiple things from changing the learning rate, batch size, epochs etc but the highest accuracy we found was a Test Accuracy of 82.67% and a Train Accuracy of 99.56%, with 60 epochs, by adjusting the hyper-parameters and changing the structures. Our strategies are broadly explained in the section of *Q1.4* from page 5 until 8.