Basic Of Deep Learning - Mid Semester Project Part 2

Guy Houri (ID. 324970557), Yoav Gal (ID. 212832471)

Submitted as mid-semester project report for Basic Of Deep Learning course, Colman, 2024

1 Introduction

This project investigates the application of deep learning techniques, specifically Fully Connected Neural Networks (FCNNs), for the task of image classification of hand gestures representing digits (0-9) from the Sign Language Digits dataset. This dataset presents a challenging multi-class classification problem due to the intra-class variations in hand shapes, sizes, and orientations, as well as inter-class similarities between certain digits.

Leveraging the representational power of FCNNs, we explore various architectural configurations and hyperparameter settings to optimize performance for this task. FCNNs, provide a valuable foundation for understanding deep learning principles and offer a suitable starting point for benchmarking performance on image classification tasks. The models will learn to map raw pixel inputs to high-level feature representations through a series of linear transformations and non-linear activation functions. These learned representations are then used to predict the probability distribution over the ten digit classes using an output layer.

1.1 Data

1.1.1 Dataset Characteristics

The dataset consists of 5,000 images, each with a resolution of 28x28 pixels, depicting hand gestures corresponding to the ten digits. The data was loaded from a .npz file containing the image data and corresponding labels. The dataset was then converted to a PyTorch `Dataset` object for efficient handling during training.

1.1.2 Data Handling and Preprocessing

A custom class, inheriting from `torch.utils.data.Dataset`, was created to manage the data. Within this class, the key preprocessing step was normalization: pixel values, originally ranging from 0 to 255, were scaled to the range 0-1

by dividing by 255. The image data was converted to `torch.float32` tensors, and the labels were converted to `torch.long` tensors for compatibility with PyTorch models and loss functions.

The dataset was split into three subsets: 80% for training, 10% for validation, and 10% for testing, using PyTorch's `random_split` function. A `RandomSampler` was used with the training `DataLoader` to shuffle the data during each epoch, which is a standard practice to improve training performance and prevent the model from learning the order of the data.

1.2 Problem

This project addresses the problem of multi-class image classification of hand gestures representing digits (0-9).

2 Solution

2.1 General approach

Our preferred approach focuses on comparison, design and evaluation of Fully Connected Neural Networks (FCNNs). We will explore different FCNN architectures, varying the number of layers and neurons. We will also experiment with different hyperparameters, such as learning rate and optimizer, to optimize model performance. Performance will be evaluated using accuracy, and overfitting will be addressed.

2.2 Design

2.2.1 Environment

This project was implemented and executed using a Jupyter Notebook environment hosted on Google Colaboratory (Colab). The notebook, saved in the `.ipynb` format, provided an interactive platform for code development, experimentation, and documentation. Python, along with the PyTorch deep learning framework and supporting libraries like NumPy, was used throughout the project. Colab's cloud-based infrastructure offered access to necessary computational resources, allowing for efficient training that took no longer than several minutes in order to train the models.

2.2.2 Training Process Overview

Training a neural network involves an iterative process of feeding the model with training data, evaluating its performance, and adjusting its internal parameters (weights and biases) to improve its accuracy. Here's a high-level overview of this process:

Loss Function: measures how well the model's predictions deviate from the ground truth labels. In our case, we used the `nn.CrossEntropyLoss()` function, suitable for multi-class classification problems.

Optimizer iteratively updates the model's weights and biases based on the calculated loss. The learning rate controls the step size of these updates. We used Adam. Adam computes individual learning rates for each parameter based on the history of its gradients. This makes Adam more robust to different learning rate settings and often leads to faster convergence and better performance, especially in complex optimization landscapes. Adam also incorporates bias correction terms to account for the initialization of the moment estimates.

Epochs: Training typically occurs over multiple epochs (iterations through the entire training set). Each epoch is further divided into mini-batches of data.

Batches: *Memory Efficiency:* Processing smaller batches requires less memory, enabling training on systems with limited resources.

Faster Computation: Computing gradients and updating parameters on smaller batches is faster than processing the entire dataset at once.

Stochasticity and Generalization: The randomness introduced by mini-batch sampling provides a form of regularization. It helps the model escape local minima and generalize better to unseen data.

Training Loop:

1. Set the model to training mode.
2. Initialize epoch loss and accuracy.
3. Iterate through batches:
   (a) Transfer data and targets to the device.
   (b) Zero optimizer gradients: `optimizer.zero_grad()`.
   (c) Perform a forward pass: `output = model(data)`.
   (d) Calculate the loss: `loss = loss_fun(output, target)`.
   (e) Perform backpropagation: `loss.backward()`.
   (f) Update model weights: `optimizer.step()`.

Validation Loop: After each training epoch, a validation loop is typically executed to evaluate the model's performance on the held-out validation set. The validation process is similar to the training loop, but the model is set to evaluation mode (gradients are not computed). The validation loss and accuracy are monitored to identify potential overfitting and guide hyperparameter tuning.

Evaluation Metrics The primary evaluation metric in our case was accuracy (percentage of correct predictions on the test set). Additionally, we

employed the ‘classification_report‘ function from ‘sklearn.metrics‘ to generate a more detailed report on precision, recall, and F1-score for each digit class

2.3 Base Model

2.3.1 Architecture

The base model is a simple feedforward FCNN consisting of three fully connected layers with Adam optimizer and CrossEntropyLoss:

Input Layer: 784 neurons (flattened 28x28 images).

Hidden Layer 1: 128 neurons (ReLU activation).

Hidden Layer 2: 64 neurons (ReLU activation).

Output Layer: 10 neurons (10 digit classes, Softmax activation).

ReLU Activation The Rectified Linear Unit (ReLU) activation function sets all negative inputs to zero and leaves positive inputs unchanged. This simple non-linearity offers several advantages:

- Computational Efficiency: ReLU is computationally inexpensive, involving only a comparison and a potential assignment.
- Mitigation of Vanishing Gradients: In deep networks, gradients can become very small during backpropagation, hindering learning. ReLU helps mitigate this by having a constant gradient of 1 for positive inputs.
- Sparsity: ReLU can induce sparsity in the network, as some neurons will be inactive (outputting zero) for certain inputs. This can lead to more efficient representations.

Softmax Activation The Softmax activation function is used in the output layer for multi-class classification. It converts a vector of raw outputs into a probability distribution over the classes. It ensures that the output values are between 0 and 1 and sum up to 1, representing a valid probability distribution.

Hyperparameters

- Epochs 10: This number was selected as a balance between allowing the model sufficient training time to learn meaningful patterns from the data and avoiding excessive training time,
- Batch Size 32: Commonly used value in deep learning. It offers a good balance between computational efficiency (processing data in batches is faster than processing individual examples) and the benefits of stochastic gradient descent (smaller batches introduce more noise, which can help the model escape local minima and generalize better).

2.3.2 Results

The base model worked well with an accuracy of 0.97 in the test set.

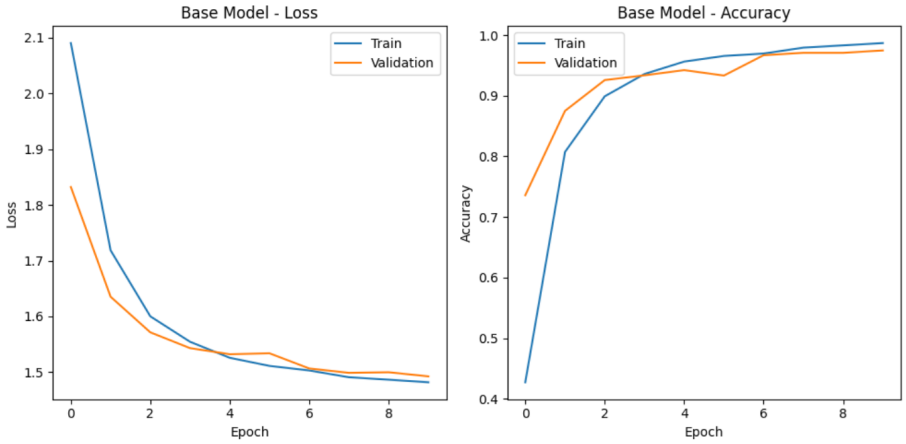

Figure 1: Base Model Results

2.4 First Experiment

This experiment investigated the effect of increasing the depth of the network by adding an additional hidden layer. We started with the base model architecture (784-128-64-10) and added a third hidden layer with 32 neurons, resulting in the architecture 784-128-64-32-10. All other hyperparameters (learning rate, batch size, number of epochs) were kept the same as in the base model.

2.4.1 Rationale

Deeper networks can learn more complex representations of the input data. By adding a hidden layer, we hypothesized that the model could capture more intricate patterns in the hand gesture images, potentially leading to improved classification accuracy.

2.4.2 Results

The following table presents the classification report for the model with the added hidden layer:

Metric	Base Model (784-128-64-10)	Deeper Model (784-128-64-32-10)
Accuracy	0.97	0.93

Table 1: Comparison of Accuracy

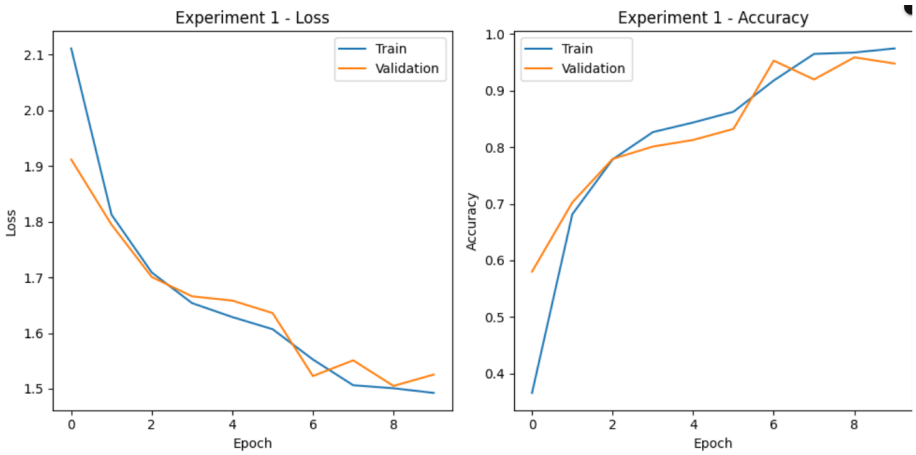

Figure 2: Expirement 1 Model Results

2.4.3 Analysis

Adding a third hidden layer did not result in a significant improvement in performance over the base model Several factors could explain this:

- **Increased Complexity Without Corresponding Benefit:** While adding a layer increases model capacity, it did not translate into a significant improvement in learning. The added layer might be redundant or not effectively contributing to feature extraction for this specific task.
- **Suboptimal Hyperparameters:** The hyperparameters, particularly the learning rate, were kept the same as the base model. It is likely that these hyperparameters are not optimal for the deeper network. The current learning rate might be too high, preventing the deeper model from converging to a better solution.
- **Data Representation:** The data might not require the increased complexity of a deeper model. The base model may already be capturing the most relevant features for classification.

2.5 Second Experiment

This experiment investigated the effect of increasing the learning rate. We used the base model architecture and increased learning rate from 0.001 to 0.01,

2.5.1 Rationale

The learning rate controls the step size of the optimizer during weight updates. A learning rate that is too high can cause the optimization process to oscillate

or diverge, preventing convergence to a good solution. We hypothesized that increasing the learning rate from the default 0.001 to 0.01 might lead to instability and worse performance.

2.5.2 Results

The following table presents the classification report for the model trained with a learning rate of 0.01:

Metric	Base Model (LR 0.001)	Model with LR 0.01
Accuracy	0.97	0.61

Table 2: Comparison of Accuracy with Different Learning Rates

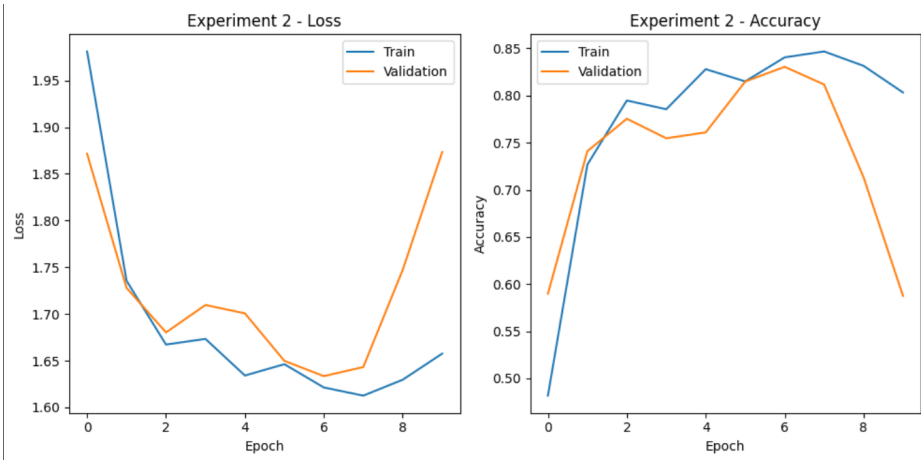

Figure 3: Experiment 2 Model Results

2.5.3 Analysis

The results clearly demonstrate the negative impact of the higher learning rate (0.01).

With a learning rate of 0.01, the optimizer likely took excessively large steps in the parameter space. This led to instability and prevented the model from converging to a good solution. The model may have overshot the optimal parameters, causing it to perform poorly on the validation set.

This experiment highlights the importance of careful learning rate selection. While a higher learning rate can sometimes speed up training, it can also lead to instability and overfitting if it is not appropriately chosen. In this case, the default learning rate of 0.001 proved to be much more suitable for this problem.

2.6 Best model Results and Metrics

The base model architecture (784-128-64-10) with the default Adam learning rate of 0.001 proved to be the most effective configuration among those tested with **accuracy of 0.97** This suggests that for this specific task and dataset, the added complexity of a deeper network was not necessary, and the original learning rate provided a more stable and effective optimization process.

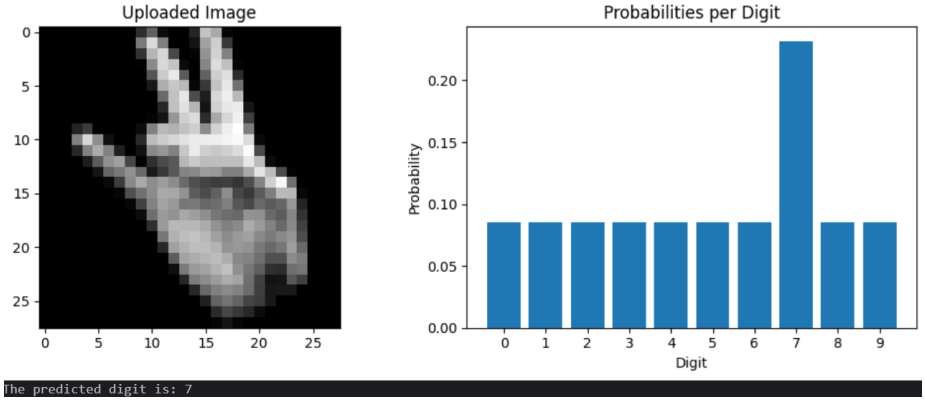

Figure 4: Base Model Prediction

3 Discussion

The experiment with a higher learning rate (0.01) demonstrated the importance of careful hyperparameter selection. As seen in the provided plots, the higher learning rate led to unstable training, with the validation loss fluctuating significantly and the accuracy dropping considerably. This highlights the sensitivity of deep learning models to the learning rate and the need for appropriate tuning. Also, we saw that in our case adding another layer for the Neural Network Wasnt beneficial.

Future experiments could explore techniques to mitigate overfitting, such as regularization or data augmentation (which are outside the scope of the current project based on the given instructions). Tuning hyperparameters specifically for the deeper model could also be beneficial.

4 Code

Link to notebook

References

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[2] Michael A. Nielsen. Neural networks and deep learning, 2018.