CS324 · Deep Learning

Assignment 2 Report

Sicheng Zhou / 12110644

November 23, 2024

1 PyTorch MLP

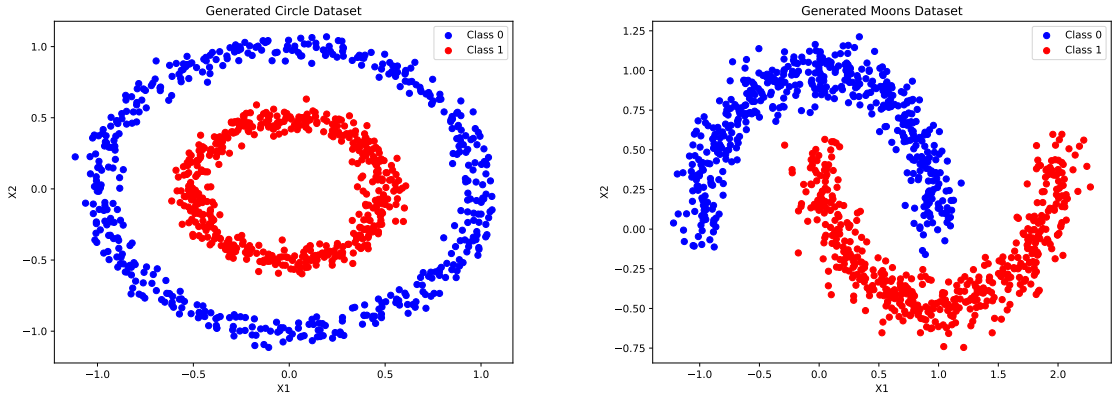
1.1 Model Structure

Input layer. Input dimension = n_inputs, output dimension = n_hidden[0].

Hidden layer. Input dimension = n_hidden[i-1], output dimension = n_hidden[i].

Output layer. Input dimension = n_hidden[-1], output dimension = n_classes.

1.2 Results



(a) Circle dataset

(b) Moon dataset

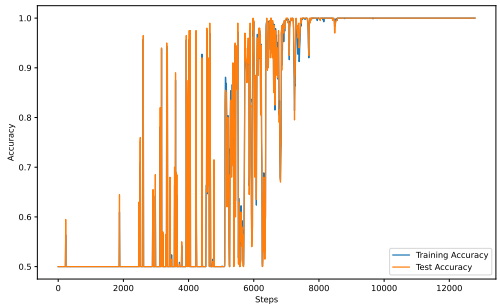
Figure 1: Two datasets. Generated using make_circles and make_moons in sklearn.datasets.

The figures show that PyTorch's MLP model is likely to converge faster than our custom implementation. This is due to several factors, including optimized libraries, better initialization techniques, and efficient computational frameworks. But after a certain number of steps, the two models tend to converge to the same accuracy.

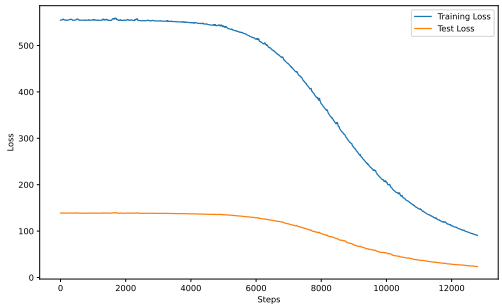
The circle dataset in sklearn.datasets is generally more challenging to train on compared to the moon dataset due to differences in their geometric properties and class separability. First, nonlinear separability. The circle dataset consists of two concentric circles. The classes are entirely interleaved, with one circle surrounded by the other. The moon dataset contains two crescent-shaped clusters. While the data is also nonlinear, the separation is easier to approximate because the two crescents are roughly linear in their respective directions, just shifted. Second, decision boundary complexity. The decision boundary in the circle dataset is circular or radial, requiring the model to learn a curved boundary. The decision boundary for the moon dataset can be approximated with a simpler piecewise linear or slightly curved boundary. Linear models with some feature engineering or neural networks with few layers can separate the two classes effectively.

1.3 CIFAR 10

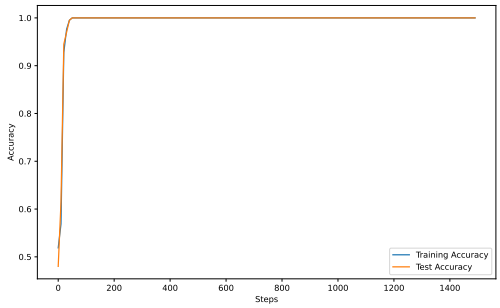
Then implement a series of optimizations on an MLP model to improve its performance on the CIFAR-10 dataset. Here are the main optimizations applied:



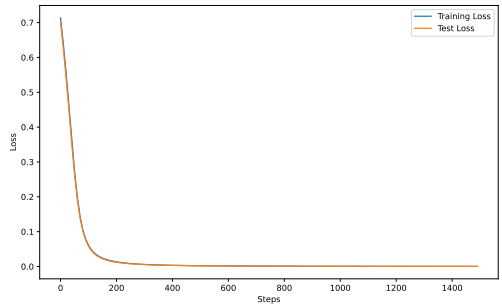
(a) numpy MLP, circle, accuracy



(b) numpy MLP, circle, loss



(c) PyTorch MLP, circle, accuracy



(d) PyTorch MLP, circle, loss

Figure 2: Results of circle dataset.

Data Augmentation. The transform variable includes data augmentation techniques including random horizontal flipping, random cropping with padding, and normalization. These methods increase dataset variability, helping the model generalize better by learning from slightly altered versions of each image.

Model Architecture and Regularization. The MLP model is defined with a hidden layer structure of [512, 256, 128] units and a dropout rate of 0.1. Dropout regularization randomly "drops" neurons during training, which helps prevent overfitting by encouraging the network to rely less on any one neuron.

Optimizer and Weight Decay. The AdamW optimizer is used with a learning rate of 0.001 and weight decay of 1e-4. AdamW is well-suited for this MLP because it helps manage sparse gradients while weight decay (L2 regularization) penalizes large weights, further mitigating overfitting.

Early Stopping. Early stopping monitors validation loss, halting training if there is no improvement for 10 epochs (determined by the patience parameter). This technique prevents overfitting by stopping the training when the model's performance on the validation set no longer improves.

The results are shown in Figure 4. The final accuracy on the test dataset is 56.66%.

2 PyTorch CNN

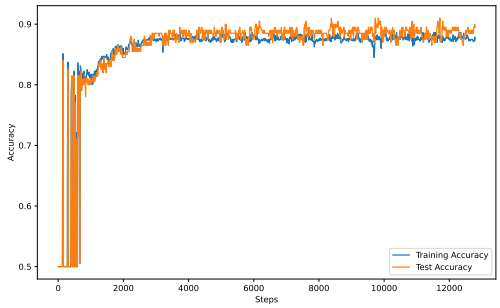
2.1 Model Structure

This model has three hidden blocks and one classifier block.

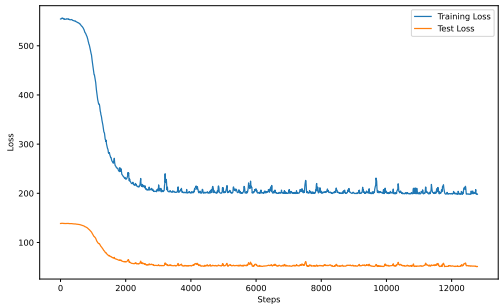
Block 1: Conv2d, a 2D convolutional layer with 64 filters, a kernel size of 3 * 3, and padding of 1 to maintain the spatial dimensions of the input;

ReLU, an activation function introducing non-linearity, applied element-wise;

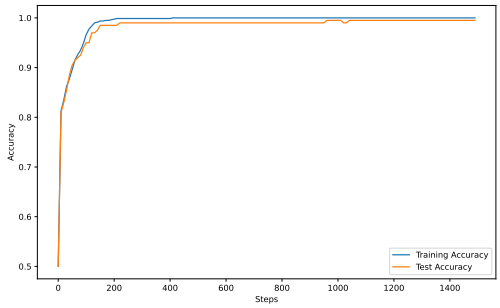
Conv2d, another convolutional layer with the same structure, increasing the complexity of the feature representation;



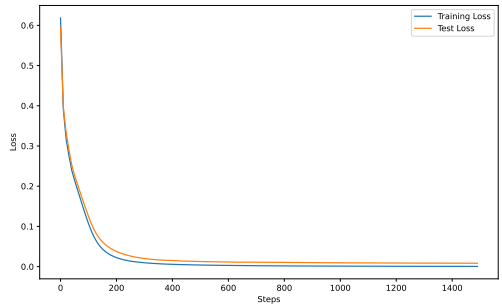
(a) numpy MLP, moon, accuracy



(b) numpy MLP, moon, loss



(c) PyTorch MLP, moon, accuracy



(d) PyTorch MLP, moon, loss

Figure 3: Results of moon dataset.

ReLU, non-linearity after the second convolution;

MaxPool2d, a pooling layer with 2 * 2 kernels and stride 2, reducing the spatial dimensions by half.

Block 2: Conv2d, a convolutional layer with 128 filters, increasing the depth of features;

ReLU, non-linearity for the newly extracted features;

Conv2d, another convolution with the same depth;

ReLU, non-linearity applied again;

MaxPool2d, reduces spatial dimensions further by half.

Block 3: Conv2d, a convolutional layer with 256 filters for deeper feature extraction;

ReLU, activation for non-linearity;

Conv2d, another convolution to refine 256-dimensional features;

ReLU, non-linearity applied again;

MaxPool2d, final pooling, reducing spatial dimensions by half.

Classifier comprises three fully connected layers and dropout for regularization.

Linear, a fully connected layer with 1024 hidden units. The input size is determined by the output of the last pooling layer (256 channels with 4 * 4 spatial dimensions);

ReLU, non-linearity for the hidden layer;

Dropout, randomly zeroes out 50% of the neurons during training to prevent overfitting;

Linear, another fully connected layer with 1024 units for further transformation;

ReLU, non-linearity applied again;

Figure 4: MLP Train and Test Loss on CIFAR10.

Dropout, regularization for the second hidden layer;

Linear, the final layer maps the features to the number of classes, producing raw scores (logits) for classification.

2.2 Training

This training script employs several optimization techniques to improve the performance and efficiency of the convolutional neural network (CNN) on the CIFAR-10 dataset. Below is a detailed explanation of the optimization methods used:

Optimization Algorithm: Adam. Adam is an adaptive learning rate optimization algorithm that combines the advantages of two other algorithms: RMSProp and Stochastic Gradient Descent with Momentum.

Loss Function: CrossEntropyLoss. The script uses torch.nn.CrossEntropyLoss as the loss function. It combines Softmax which converts raw logits into probabilities, and negative Log-Likelihood (NLL), which calculates the loss between the predicted probabilities and the ground truth labels.

Data augmentation is applied to the training dataset to artificially increase its size and diversity, improving the model's generalization capabilities. The transformations include RandomHorizontalFlip, which randomly flips images horizontally, RandomCrop, which crops a 32 * 32 patch from the image with padding to simulate slight shifts, Normalization, which scales pixel values to a range with a mean of (0.4914, 0.4822, 0.4465) and a standard deviation of (0.2023, 0.1994, 0.2010), matching the CIFAR-10 dataset's characteristics.

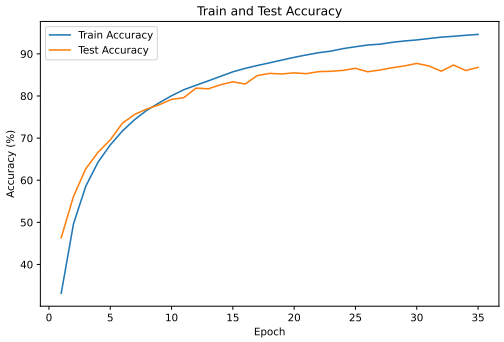
Early stopping is implemented to prevent overfitting and save computational resources. It can reduce overfitting by halting training when validation performance stops improving.

2.3 Results

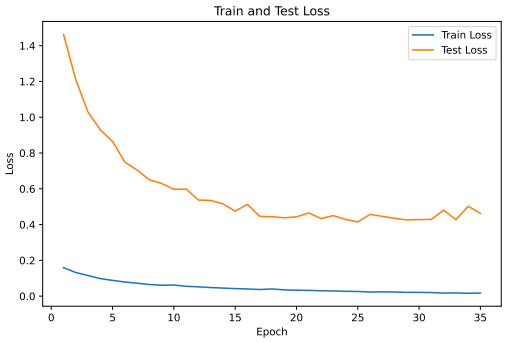
Comparing Figure 4 to Figure 5, we can see that CNN outperform MLP on the CIFAR dataset. This is primarily due to their architectural advantages and suitability for handling image data.

Local connectivity and spatial hierarchies. CNNs exploit the spatial structure of images. Convolutional layers use small filters that focus on local regions, capturing spatially local patterns like edges, textures, and simple shapes. These local features are then combined hierarchically to detect more complex structures, such as objects or regions of interest. MLPs treat each pixel in the image as an independent input feature, ignoring the spatial relationships between neighboring pixels. This lack of spatial awareness makes it harder for MLPs to learn meaningful patterns in images, especially for high-dimensional data like CIFAR.

Parameter efficiency. CNNs share parameters across different parts of the image via convolutional filters. This weight-sharing mechanism drastically reduces the number of parameters compared to MLPs. In



(a) Train and Test Accuracy



(b) Train and Test Loss

Figure 5: CNN train and test results.

an MLP, each input pixel is connected to every neuron in the next layer (fully connected), resulting in a much larger number of parameters, especially for high-resolution inputs. This can lead to overfitting and computational inefficiency, particularly for large image datasets.

Translation invariance. Convolutional operations are inherently translation-invariant, meaning that features learned by CNNs (e.g., edges or textures) can be detected regardless of their position in the image. Pooling layers (e.g., max pooling) further enhance this property by summarizing features across small regions, making the network robust to small shifts or distortions in the input. MLPs lack this built-in invariance because they process all inputs as flat vectors. As a result, an MLP may struggle to recognize the same feature in different parts of the image unless explicitly trained to do so.

3 PyTorch RNN

3.1 Model Structure

Initialization inputs:

input_length: The length of the input sequence (number of timesteps).

input_dim: The dimensionality of each input vector at a single timestep.

hidden_dim: The dimensionality of the hidden state vector.

output_dim: The dimensionality of the output vector.

Weights are initialized with random values drawn from a normal distribution. Biases are initialized to zeros.

Iterative Sequence Processing. For each timestep t in the input sequence, the input at timestep t, xt (of shape (batch_size, input_dim)), is combined with the current hidden state h using the formula:

$h = \tanh(\mathbf{x}t\mathbf{W}xh + \mathbf{h}\mathbf{W}_{hh} + \mathbf{b}_h)$

Output Computation. After processing the entire sequence, the final hidden state is used to compute the output:

output = $\mathbf{h}\mathbf{W}_{hy} + \mathbf{b}_y$

3.2 Training

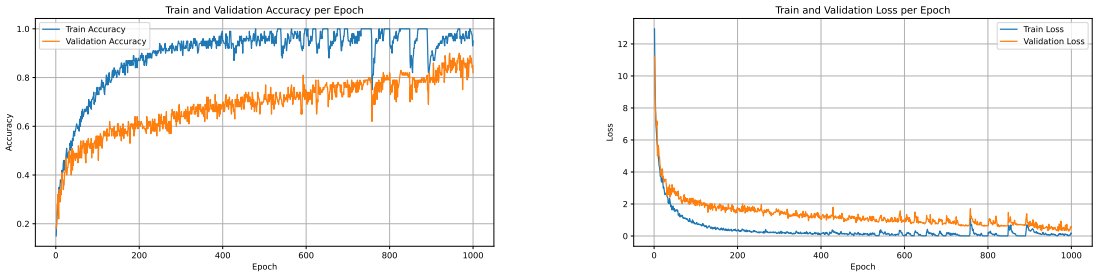
Loss Function: The training process uses the torch.nn.CrossEntropyLoss() function, which is suitable for classification tasks. It calculates the difference between the predicted probability distribution (outputs) and the target labels.

Optimizer: RMSprop is used to optimize the model's parameters. It adjusts the learning rate dynamically for each parameter by dividing the gradient by a running average of its recent magnitudes. This is particularly useful for handling the exploding or vanishing gradients often encountered in RNNs.

Gradient Clipping: To mitigate exploding gradients, the training loop applies gradient clipping using torch.nn.utils.clip_grad_norm_. The norm of the gradients is limited to a maximum value (config.max_norm, default set to 10.0). If the gradients exceed this value, they are scaled down proportionally to fit within this threshold.

Batch Training: The dataset is divided into mini-batches (of size config.batch_size, default 128). Mini-batches help improve training efficiency and reduce memory usage while stabilizing the optimization process through gradient averaging.

3.3 Results



(a) Train and Test Accuracy

(b) Train and Test Loss

Figure 6: RNN train and test results when T = 5.

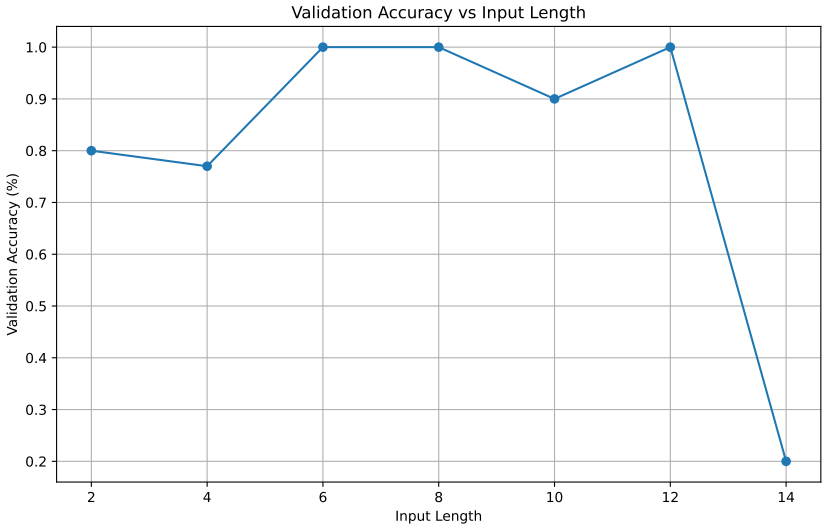


Figure 7: Accuracy versus palindrome length.

We can see from Figure 6 that we can obtain close to perfect accuracy with T = 5. We can see from Figure 7 that the prediction accuracy of the RNN become lower as we challenge it with longer and longer palindromes. This is due to the following reasons:

Vanishing gradients. RNNs rely on sequential processing, where information is propagated through hidden states across time steps. For long sequences, gradients can diminish exponentially during back-propagation, leading to the vanishing gradient problem. As a result, the RNN struggles to retain relevant

information from earlier parts of the sequence, which is critical for identifying patterns in palindromes (e.g., comparing mirrored characters from the start and end of the sequence).

Memory bottleneck. RNNs have a fixed-size hidden state vector that is used to encode the entire sequence's information. For longer palindromes, this limited capacity becomes a bottleneck, causing the model to lose important details from earlier parts of the sequence. This loss of information prevents the RNN from accurately matching characters in a palindrome, where symmetry across the sequence is essential.