CS324: Deep Learning
Assignment 2

Wai Yan Kyaw (12312638)

1 Part I: PyTorch MLP

1.1 Task 1 and Task 2

I already have numpy vesion of MLP in assignment-1 and now this is my time for implementing the same MLP in pytorch. Since there are already a lot of packages and libraries from Pytorch, it is easier for me to implement the same MLP structure in pytorch by using nn.Linear for linear layer and nn.ReLU for relu actiation function and I packed all layers using nn.Sequential() and finally use nn.CrossEntropyLoss() to compute Cross Entropy loss of the model.

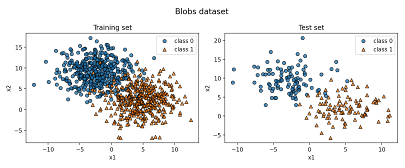
1.1.1 Model Architecture

I use these default parameters for both Pytorch and Numpy version.

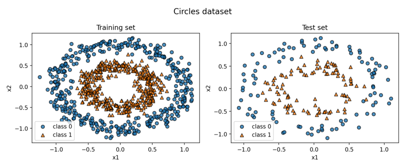
- `HIDDEN_UNITS`: [20]
- `LEARNING_RATE`: 0.01
- `MAX_STEPS`: 1500
- `EVAL_FREQ`: 50
- `LOSS_FUNCTION`: CrossEntropyLoss
- `OPTIMIZER`: SGD

So the architecture is Linear(2 → 20) → ReLU() → Linear(20 → 2) → Softmax(2→2) which exactly matches MLP Numpy version.

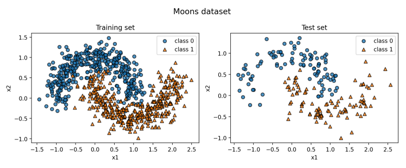
1.1.2 Datasets Creation and Training



(a) Blobs dataset



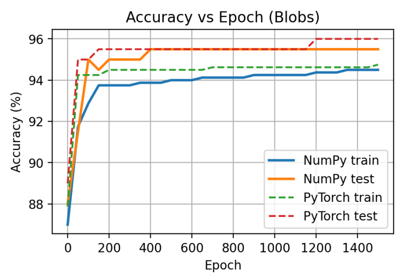
(b) Circles dataset



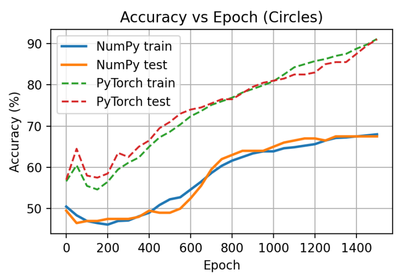
(c) Moons dataset

Figure 1: Training and test splits for the three synthetic datasets.

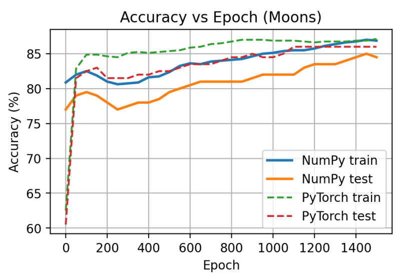
I trained both models on three datasets: `make_blobs`, `make_moons`, and `make_circles`. For each dataset, I created 1000 samples and then split 800 samples for training data and 200 for testing data as in Assignment1. These are the figures showing dataset sample points with some noise to make sure there is some difficulty for our model to learn. I use `torch.optim.SGD` as my optimizer and used batch gradient descent to train and update parameters.



(a) Blobs accuracy



(b) Circles accuracy



(c) Moons accuracy

Figure 2: Accuracy vs epoch for NumPy and PyTorch MLPs on each dataset.

1.1.3 Result Analysis

Moon Dataset Both Pytorch and Numpy version achieved final accuracy for both train and test dataset is around 85% after 1500 steps. The four curves are very similar, meaning there is no significant different between both versions and both models generalize very well.

Blobs Dataset For simple blobs dataset where two blobs do not overlap that much, both versions achieve final train and test accuracy of around 95% and there is no serious overfitting in both models.

Circle Dataset The most interesting part is when testing with circle dataset which has some difficulty for the model requiring to find the strong non linear decision boundary. As you see in the figure, the accuracy curves for Numpy version are lower than these for Pytorch version. Final accuracy for Numpy verion is around 67% and that for Pytorch version is around 91%. Therefore, I can conclude that NumPy model did not learn a correct nonlinear separation than Pytorch one because the gradients or updates may not be stable enough due to hand-coded implemention and possibly the learning rate or weight initialization in my NumPy code is suboptimal. However, for pytorch version, I can conclude that PyTorch optimizer (even plain SGD) is more numerically stable and PyTorch layers initialize weights better than my NumPy 0.1 * randn. Additionally, PyTorch autograd computes gradients more robustly. Therefore, PyTorch version learns a much better decision boundary because PyTorch is built to be robust, while hand-implemented backprop is fragile.

1.2 Task 3

1.2.1 Model Architecture

I use these parameters for this task:

- `HIDDEN_UNITS`: [1024, 512, 256]
- `LEARNING_RATE`: 0.001
- `NUM_EPOCHS`: 100
- `WEIGHT_DECAY`: 0.0001 (L2 regularization)
- `BATCH_SIZE`: 128
- `LOSS_FUNCTION`: CrossEntropyLoss

- `OPTIMIZER`: Adam

Since CIFAR-10 images have 3 channels $\times$ 32 $\times$ 32 = 3072 input features per image. I implemented the model like this: Flatten($3\times32\times32 = 3072$) $\rightarrow$ Linear($3072 \rightarrow 1024$) $\rightarrow$ ReLU $\rightarrow$ Linear($1024 \rightarrow 512$) $\rightarrow$ ReLU $\rightarrow$ Linear($512 \rightarrow 256$) $\rightarrow$ ReLU $\rightarrow$ Linear($256 \rightarrow 10$(logits)).

The first layer maps $3072 \rightarrow 1024$ and therefore has $3072 \times 1024 = 3,145,728$ weights and 1024 biases, for a total of $3,146,752$ parameters. The second layer maps $1024 \rightarrow 512$, with $1024 \times 512 = 524,288$ weights and 512 biases, for a total of $524,800$ parameters.

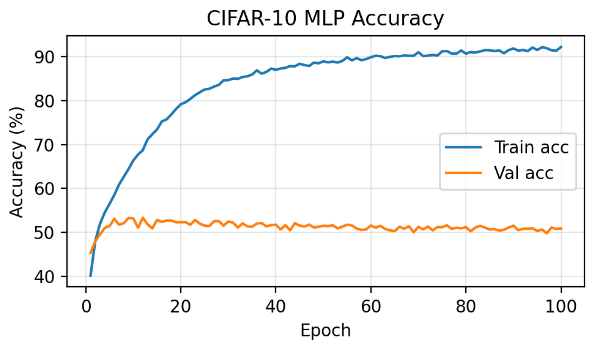
The third layer maps $512 \rightarrow 256$, with $512 \times 256 = 131,072$ weights and 256 biases, for a total of $131,328$ parameters. The final output layer maps $256 \rightarrow 10$, contributing $256 \times 10 = 2,560$ weights and 10 biases, for a total of $2,570$ parameters.

Therefore, the MLP has a total of 3,805,450 trainable parameters.

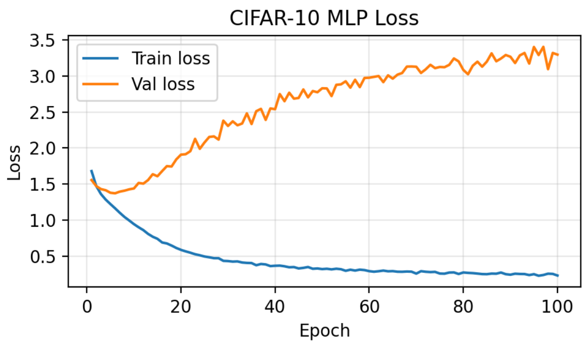
1.2.2 Dataset Creation and Training

I downloaded CIFAR-10 dataset using `torchvision`. Then, I split 45000 images for training, 5000 for validation and 10000 images for testing. Now this time I use `Adam Optimizer` to update the parameters and then track accuracy and loss.

1.2.3 Result Analysis



(a) CIFAR-10 MLP accuracy



(b) CIFAR-10 MLP loss

Figure 3: Training and validation accuracy and loss for the CIFAR-10 MLP model.

As we see in figures, training accuracy increases from 40% to 90% after 65 epoches. Final training accuracy is 92.21% which is good and training loss decreases 1.67 to 0.229 after 100 epoches. However, validation accuracy is around 50% from epoch 1 to final epoch and loss increases from 1.55 to 3.29 after 100 epoches. After that, when I test on 10000 images test data, test accuracy is 52.24% and test loss is 3.2736. Therefore, I have to conclude that the model overfits because the MLP model is not the right model for this dataset and it is bad for images. Since it flattens the 2-D structure(height $\times$ width $\times$ channels) to 1-D long vector, it loses all spatial relationships and cannot detect edges, corners, textures, and shapes. Therefore, it memorizes training set but cannot generalize.

2 Part II: PyTorch CNN

2.1 Model Architecture

The architecture is the reduced VGG-like network introduced in lecture week 8. Each ConvBlock consists of: Conv2D (kernel 3×3, stride 1, padding 1), BatchNorm2D, and ReLU activation.
I use these parameters for this architecture:

- `LEARNING_RATE`: 1e-4
- `NUM_EPOCHS`: 50
- `EVALUATION_FREQUENCY`: 1
- `BATCH_SIZE`: 32
- `LOSS_FUNCTION`: CrossEntropyLoss
- `OPTIMIZER`: Adam

The network structure is:

Layer Type	**Output Channels**	**Kernel/Stride/Padding**	**Output Spatial Size**
ConvBlock	64	3×3 / 1 / 1	32×32
MaxPool	—	3×3 / 2 / 1	16×16
ConvBlock	128	3×3 / 1 / 1	16×16
MaxPool	—	3×3 / 2 / 1	8×8
ConvBlock	256	3×3 / 1 / 1	8×8
ConvBlock	256	3×3 / 1 / 1	8×8
MaxPool	—	3×3 / 2 / 1	4×4
ConvBlock	512	3×3 / 1 / 1	4×4
ConvBlock	512	3×3 / 1 / 1	4×4
MaxPool	—	3×3 / 2 / 1	2×2
ConvBlock	512	3×3 / 1 / 1	2×2
ConvBlock	512	3×3 / 1 / 1	2×2
MaxPool	—	3×3 / 2 / 1	1×1
Flatten	—	—	512
Linear Layer	10 classes	—	—

Table 1: Model Architecture

2.1.1 parameter calculation

Convolutional layer (Conv2d). For a 2D convolution with

$$\texttt{in\_channels} = C_{\text{in}}, \quad \texttt{out\_channels} = C_{\text{out}}, \quad \texttt{kernel\_size} = K \times K,$$

and a learnable bias for each output channel, the number of parameters is

$$N_{\text{conv}} = C_{\text{out}} \cdot C_{\text{in}} \cdot K^2 + C_{\text{out}}. \tag{1}$$

Batch normalization (BatchNorm2d). Batch normalization has one learnable scale parameter γ and one learnable shift parameter β per feature map. For C_{out} channels this gives

$$N_{\text{bn}} = 2 \cdot C_{\text{out}}. \tag{2}$$

Fully-connected layer (Linear). For a linear layer with F_{in} input features and F_{out} output features (with bias) the number of parameters is

$$N_{\text{fc}} = F_{\text{out}} \cdot F_{\text{in}} + F_{\text{out}}. \tag{3}$$

Each convolutional block (`ConvBlock`) in the model consists of

$$\text{Conv2d} \rightarrow \text{BatchNorm2d} \rightarrow \text{ReLU},$$

where ReLU has no parameters. For kernel size $K = 3$ (as in the architecture), the number of parameters in a block with C_{in} input channels and C_{out} output channels is

$$\begin{align} N_{\text{block}} &= N_{\text{conv}} + N_{\text{bn}} \tag{4}\\ &= \big(C_{\text{out}} \cdot C_{\text{in}} \cdot K^2 + C_{\text{out}}\big) + 2C_{\text{out}} \tag{5}\\ &= C_{\text{out}}\big(C_{\text{in}}K^2 + 3\big). \tag{6}\end{align}$$

With $K = 3$ and $K^2 = 9$, this simplifies to

$$N_{\text{block}} = C_{\text{out}}\big(9C_{\text{in}} + 3\big). \tag{7}$$

The model uses the following sequence of convolutional blocks:

$$3 \rightarrow 64, \quad 64 \rightarrow 128, \quad 128 \rightarrow 256, \quad 256 \rightarrow 256, \quad 256 \rightarrow 512, \quad 512 \rightarrow 512, \quad 512 \rightarrow 512, \quad 512 \rightarrow 512,$$

followed by a fully-connected classifier mapping 512 features to 10 classes.
Using the formula above for each block, the total parameters per component are:

- Block 1 ($3 \rightarrow 64$): $N_{\text{block}} = 1{,}920$
- Block 2 ($64 \rightarrow 128$): $N_{\text{block}} = 74{,}112$
- Block 3 ($128 \rightarrow 256$): $N_{\text{block}} = 295{,}680$
- Block 4 ($256 \rightarrow 256$): $N_{\text{block}} = 590{,}592$
- Block 5 ($256 \rightarrow 512$): $N_{\text{block}} = 1{,}181{,}184$
- Block 6 ($512 \rightarrow 512$): $N_{\text{block}} = 2{,}360{,}832$
- Block 7 ($512 \rightarrow 512$): $N_{\text{block}} = 2{,}360{,}832$
- Block 8 ($512 \rightarrow 512$): $N_{\text{block}} = 2{,}360{,}832$
- Classifier (Linear $512 \rightarrow 10$):

$$N_{\text{fc}} = 10 \cdot 512 + 10 = 5{,}130.$$

$$\begin{align} N_{\text{total}} &= 1{,}920 + 74{,}112 + 295{,}680 + 590{,}592 + 1{,}181{,}184 \tag{8}\\ &\quad + 2{,}360{,}832 + 2{,}360{,}832 + 2{,}360{,}832 + 5{,}130 \tag{9}\\ &= 9{,}231{,}114. \tag{10}\end{align}$$

Therefore, this model contains a total of 9,231,114 learnable parameters.

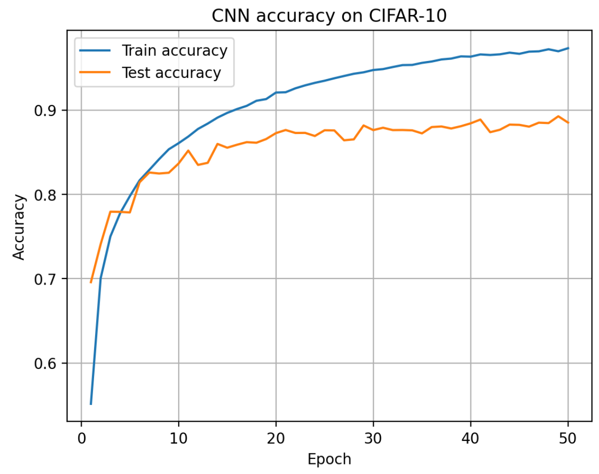
2.2 Dataset Creation

I use the CIFAR-10 as in Part1. For the training set, I use standard data augmentation and normalization to improve generalization.
The training pipeline first performs a random crop of size 32×32 with padding of 4 pixels, which introduces small spatial shifts and makes the model more robust to object position. Then, a random horizontal flip is applied with probability 0.5, which helps the network learn invariance to left–right orientation. After these augmentations, the image is converted to a tensor and channel-wise normalized using the empirical CIFAR-10 mean and standard deviation

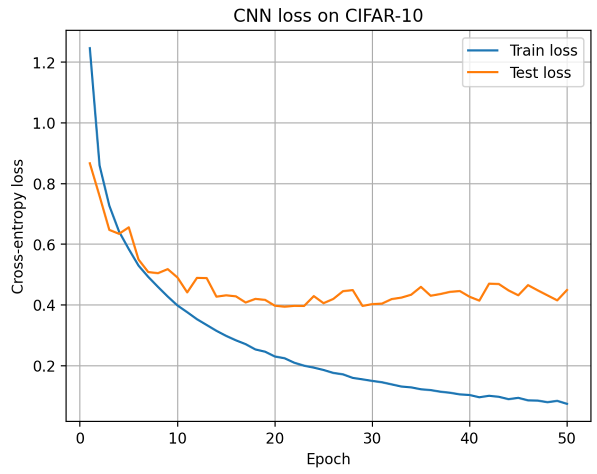
$$\mu = [0.4914,\ 0.4822,\ 0.4465], \quad \sigma = [0.2470,\ 0.2435,\ 0.2616].$$

For the test set, I do not apply any augmentation, only the same tensor conversion and normalization so that evaluation is performed on the original images.
Using identical normalization for train and test makes the model see inputs in a consistent scale during both training and evaluation. However, data augmentation is used only for the training set to avoid biasing the test performance.

2.3 Result Analysis



(a) CNN accuracy on CIFAR-10



(b) CNN loss on CIFAR-10

Figure 4: Training dynamics of the CNN on CIFAR-10: accuracy (left) and cross-entropy loss (right) as a function of epoch.

CNN performance on CIFAR-10. Training accuracy curve climbs steeply during the first few epochs and then continues to increase more slowly, reaching around 95% by epoch 50, and test accuracy rises to about 88–89% and then fluctuates within a narrow band. In the loss figure, training loss decreases from values above 1.2 down to below 0.1, and test loss drops quickly at the beginning and then stabilizes around 0.4–0.5. The training loss keeps decreasing while test loss and test accuracy essentially plateau shows the mild overfitting because the model continues to fit the training set more and more precisely but its generalization to unseen data improves only marginally after 20–30 epochs.

Comparison with MLP baseline from Assignment 1 The MLP's training accuracy becomes very high (around 90% and above) but its validation accuracy saturates early at only about 50–52% and then remains flat. So there is a large gap between training and validation curves. Similarly, MLP training loss continues to decrease while validation loss increases steadily over time,which shows a strong overfitting beacuse the network memorizes the training set but fails to learn features that generalize to new images. This contrast highlights why CNNs are far better suited for image data than fully connected MLPs because convolutions and pooling use spatial structure, share weights across the image and learn local patterns,which makes the CNN reach much higher test accuracy and significantly lower test loss on CIFAR-10 with the same dataset.

3 Part III: PyTorch RNN

3.1 Model Architecture

I implemented a simple vanilla RNN manually using three fully connected (linear) transformations: an input-to-hidden transformation, a hidden-to-hidden recurrent transformation, and a hidden-to-output transformation. I use input_dim = 1, hidden_dim = 128, and output_dim = 10.

(1) Input-to-hidden layer

T he input-to-hidden map is

$$W_{hx} : \mathbb{R}^{\text{input\_dim}} \to \mathbb{R}^{\text{hidden\_dim}}.$$

For a batch of size B:

- Input shape: $(B, 1)$.
- Output shape: $(B, 128)$.

The learnable parameters are:

- Weight matrix of size $(128, 1)$.
- No bias term in this layer.

The total number of parameters in this layer is

$$128 \times 1 = 128.$$

(2) Hidden-to-hidden recurrent layer

The recurrent hidden-to-hidden map is

$$W_{hh} : \mathbb{R}^{\text{hidden\_dim}} \to \mathbb{R}^{\text{hidden\_dim}},$$

with a bias term. It is applied at each time step to the previous hidden state.
For a batch of size B:

- Input (previous hidden state) shape: $(B, 128)$.
- Output (hidden contribution) shape: $(B, 128)$.

The learnable parameters are:

- Weight matrix of size $(128, 128)$.
- Bias vector of size (128).

The total number of parameters in this layer is

$$\text{weights: } 128 \times 128 = 16{,}384, \qquad \text{bias: } 128,$$

$$\Rightarrow \text{total} = 16{,}384 + 128 = 16{,}512.$$

(3) Hidden-to-output layer

The hidden-to-output map is

$$W_{ph} : \mathbb{R}^{\text{hidden\_dim}} \to \mathbb{R}^{\text{output\_dim}},$$

also with a bias term. It is applied once, after the last time step.
For a batch of size B:

- Input (final hidden state) shape: $(B, 128)$.
- Output (logits for 10 digits) shape: $(B, 10)$.

The learnable parameters are:

- Weight matrix of size $(10, 128)$.
- Bias vector of size (10).

The total number of parameters in this layer is

$$\text{weights: } 10 \times 128 = 1{,}280, \qquad \text{bias: } 10,$$

$$\Rightarrow \text{total} = 1{,}280 + 10 = 1{,}290.$$

Total number of parameters

$$\begin{aligned} \text{Input-to-hidden} &: 128 \\ \text{Hidden-to-hidden} &: 16{,}512 \\ \text{Hidden-to-output} &: 1{,}290 \end{aligned}$$

Therefore, the total number of learnable parameters in the model is

$$128 + 16{,}512 + 1{,}290 = 17{,}930.$$

At each time step, the model reads a single digit, updates its hidden state, and at the end of the sequence, uses the final hidden state to predict the last digit.
The parameters I use in this model are:

- `INPUT_LENGTH`: T-1
- `INPUT_DIM`: 1
- `NUM_CLASSES`: 10

- `NUM_HIDDEN`: 128
- `BATCH_SIZE`: 128
- `LEARNING_RATE`: 0.001
- `TRAIN_STEPS`: 1000
- `MAX_NORM`: 10.0
- `LOSS_FUNCTION`: CrossEntropyLoss
- `OPTIMIZER`: RMSprop

3.2 Training Procedure

For each chosen sequence length T, first I set the input sequence length to $L = T - 1$ and initialize a new RNN model. Then, palindrome dataset is constructed to generate sequences of total length T. A data loader is used to sample mini-batches of size B. For a fixed number of training steps, each iteration samples a fresh batch of inputs and targets from the data loader. Then, it performs a forward pass through the network. So, the inputs have shape $(B, L, 1)$, the hidden state is updated through time using the given recurrent equations, and the final hidden state is mapped to a vector of logits of shape $(B, 10)$. After that, the cross-entropy loss is computed using only the last digit as the target. During the backward pass, gradients are propagated back through time, their global norm is clipped to mitigate exploding gradients, and the model parameters are updated using the RMSProp optimizer.

3.3 Result Analysis

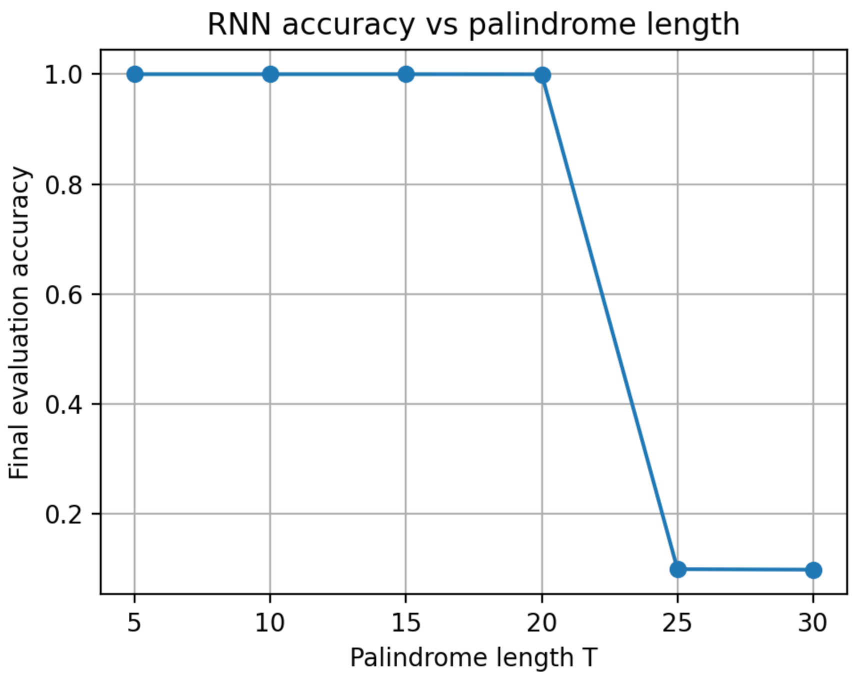
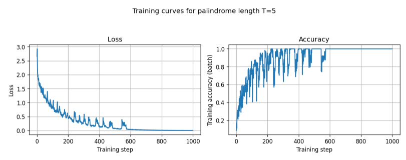
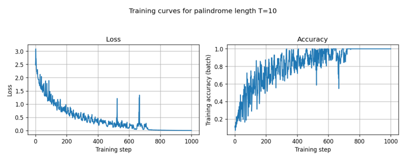


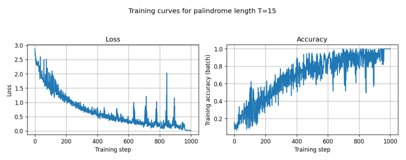
Figure 5: RNN accuracy as a function of palindrome length T.



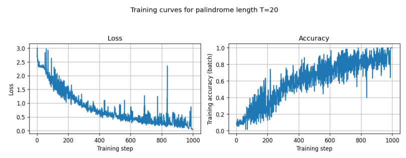
(a) $T = 5$



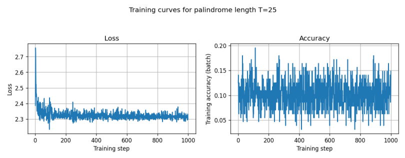
(b) $T = 10$



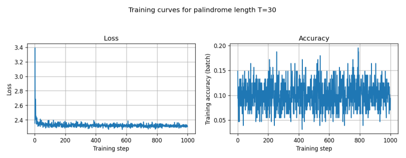
(c) $T = 15$



(d) $T = 20$



(e) $T = 25$



(f) $T = 30$

Figure 6: Training loss and batch accuracy curves for different palindrome lengths T. Each subfigure shows the loss (left) and accuracy (right) over training steps for a fixed T.

The results show that for shorter sequences ($T = 5, 10, 15, 20$), the RNN achieves almost perfect evaluation accuracy (close to 100%) on unseen palindromes. The training curves for these lengths indicate that the cross-entropy loss steadily decreases towards zero while the training accuracy rises from 0.1 to values close to 1.0. As T increases within this range, learning becomes progressively slower and noisier: the loss curves exhibit larger fluctuations and the accuracy curves take more steps to saturate, but they still converge to high performance after enough training iterations. Therefore, I can conclude that up to around 20 time steps(T=20), the vanilla RNN is able to encode the information contained in the first digit and retain it in its hidden state until the final prediction.
However, for longer palindromes ($T = 25$ and $T = 30$) the model fails to learn the task. Both the training and evaluation accuracy remain around 0.1. The training curves for these lengths show that the loss decreases only slightly from its initial value and then plateaus, and the accuracy fluctuates randomly without any upward trend. This is characteristic of the vanishing gradient problem in vanilla RNNs beacause the dependency between the first and last digit spans 24–29 recurrent steps.So the gradients associated with the early time steps become negligibly small, which prevents the network from learning a representation that preserves the first digit over such long sequences.

4 Instructions to Run Code

4.1 Part 1

Run `task2.ipynb` for Task 1 and Task 2 (Pytorch MLP).

Run `cifar10.ipynb` for Task 3 (MLP CIFAR10).

4.2 Pytorch CNN

Run `task2.ipynb`.

4.3 Pytorch RNN

Run `rnn.ipynb`.