12410106 Assignment2

Zhang Sihua

DeepLearning 25 fall

Abstract. This is the report about assignment2 of DeepLearning course, covering MLP, CNN and RNN. In this report, we illustrate the implementation details and results analysis of the assignment. The report contains three parts: the PyTorch Multi-Layer Perceptron (MLP), the Convolutional Neural Network (CNN) on CIFAR-10, and the Recurrent Neural Network (RNN) for Palindrome detection. To run our code, just run the jupyter notebook in directory Part_x (x=1,2,3) or execute bash commands provided in the README file to run python scripts.

1 Part1 PyTorhc MLP

In this part we implement Multi-Layer Perceptrons using PyTorch, train and test both PyTorch and NumPy version of MLP on Moon Datatset to show the results are almost the same. We train and test our PyTorch MLP on CIFAR10, a small computer-vision dataset containing 60,000 32x32 color images and 10 object classes, and manage to get 63.83% of classification accuracy.

1.1 Task1: PyTorch Implementation

We implement a robust MLP using PyTorch in `pytorch_mlp.py`. The model architecture is the same as mlp implemented in pure numpy in last assignment . We also provide `pytorch_train_mlp.py` to train and test our model.

1.2 Task2: MLP comparison

The old NumPy MLP is in `numpy_mlp.py` and `modules.py`. We train and test both PyTorch and NumPy MLP on the same dataset created by `sklearn.datasets.make_moons`,and get similar results in `MLP.ipynb`. We also test them on other dataset, for example a circle distribution generated by `sklearn.datasets.make_circles`. Our MLP do no better than randomly guessing, as is expedcted. Our simle MLP cannot correctly classify dataset unseen during training.

1.3 Task3: MLP on CIFAR10

We use PyTorch and its units to implement a more robust 3-layer MLP in `CIFAR10.ipynb` and `mlp_cifar10.py`.We utilize `nn.BatchNorm1d` and `nn.Dropout`

(p=0.2) after each linear layer (except the output layer) to stabilize training and prevent overfitting. The activation function used is ReLU.

Data augmentation is applied to the training set, including `RandomCrop` and `RandomHorizontalFlip`, which helps the model generalize better. We use the SGD optimizer with momentum (0.9) and weight decay (5×10^{-4}), along with a Cosine Annealing learning rate scheduler. Our device is a single RTX 2080.

After training 100 epochs, we get a test accuracy of 63.83% on CIFAR10, significantly better than 10% (before training).

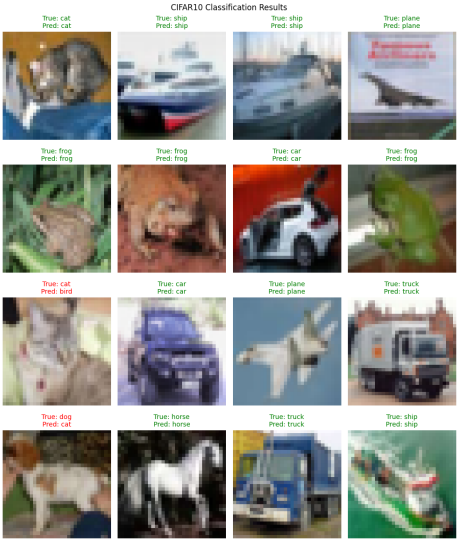

Fig. 1. CIFAR-10 Classification Results (MLP)

2 Part2 CNN on CIFAR-10

2.1 Task1: CNN Architecture

In this part, we implement a Convolutional Neural Network (CNN) in `cnn_model.py`. The architecture follows a VGG-style design, consisting of stacked convolutional layers followed by max-pooling layers.

Specifically, we define a `make_conv_layer` helper function that creates a block of `Conv2d`, `BatchNorm2d`, and `ReLU`. All `Conv2d` layers are set to maintain the size of feature map (a.k.a same padding) .The network use `Conv2d` to progressively increases the number of channels from 3 (input) to 64, 128, 256, and finally 512. Max pooling with kernel size 3 and stride 2 is used to downsample the feature maps by half each time, until only one pixel remains. A final fully connected layer maps the flattened features to the 10 class scores.

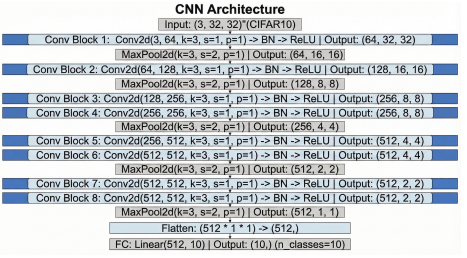

Fig. 2. CNN Architecture

2.2 Task2: Training and Results

We train the CNN on CIFAR-10 using the script `cnn_train.py`. The use of Batch Normalization significantly accelerates convergence. We use SGD with batch size 128 (MBGD), Adam optimizer with weight decay, and CosineAnnealingLR learning rate scheduler eta_min = 1e-5. Data augmentation includes RandomCroppinig, RandomHorizontalFlipping, olorJitterring, RandomGrayscaling and GaussianBlurring, which is a classical augmentation strategy on CIFAR10. These stategy help our model learn more robust features and alleviate the problem of overfitting.

We train our model on a single GPU RTX 2080 for 100 epochs and get accuracy of 96.8% on train dataset and 88.9% on test dataset.

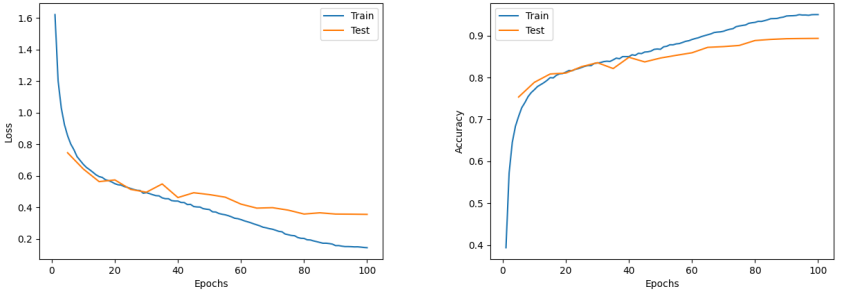

Fig. 3. CNN Training Loss and Accuracy Curves

As shown in Fig.2, the model achieves high accuracy on the test set. The loss curve shows a steady decrease, indicating stable training. How ever our model is slightly overfitting on train dataset during training process, which is common for small VGG like models. To further help with our model generalize on validation data, dropout or other regularization strategies and stronger data augmentation methods can be applied.

3 Part3 RNN on Palindrome

3.1 Task1: Vanilla RNN

In this part, we implement a Vanilla Recurrent Neural Network (RNN) in `vanilla_rnn.py`. The RNN maintains a hidden state h_t that is updated at each time step based on the current input x_t and the previous hidden state h_{t-1}:

$$h_t = \tanh(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$$

The output y_t is then computed from the hidden state:

$$o_t = W_{hy}h_t + b_y$$

The shape of W_{xh} is (hidden_size, input_size), the shape of W_{hh} is (hidden_size, hidden_size), the shape of W_{hy} is (output_size, hidden_size).

Our model output logits directly beacause `nn.CrossEntropyLoss()` do log softmax inside.

We use the RNN to solve a Palindrome prediction task. The dataset is generated dynamically in `dataset.py`. Given a sequence of digits representing the first $N-1$ digits of a palindrome of length N, the model is trained to predict the last digit. Only the output of the last time step (N-1) is used to predict the N th digit, and computing gradients.

We train the model using RMSprop optimizer. Since Vanilla RNNs can suffer from vanishing or exploding gradients, we use `torch.nn.utils.clip_grad_norm_` to clip gradients to 10. The model successfully learns the symmetry property of palindromes.

It is worth mentioning that instead of using numerical feature space (1..9), we encode them into categorical feature space using `torch.nn.functional.one_hot`. It turns out that this encoding strategy singnicifantly improve training efficiency. We infer that onehot encoding prevent our model from being confused by inherent numerical relations, like 4 is two times larger than 2, which essentially has nothing to do with our palindrome number classification.

3.2 Task2: Experiment on palindrome's length

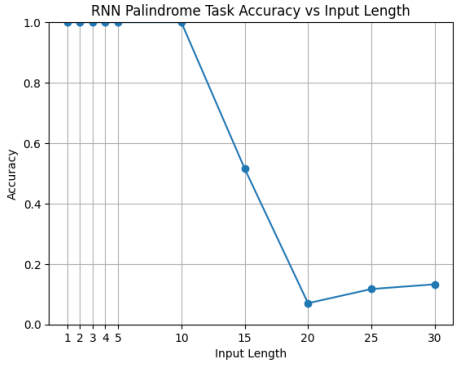

Fig. 4. Relationship Bewteen Accuracy and Palindrome's Length T

We do more experiments to explore the relationship bewteen prediction accuracy and palindrome's length T, we use the RNN model implemented in task1, train and test on palindrome with differents T. Due to the limited memory of our simple RNN, prediction accuracy should decrease while T increase. Our expeiments further prove our reasoning.