**1 True/False Questions (30 pts)**

**Problem 1.1 – False**

Explanation: The outputs of BN module do not have to have exactly the zero mean. Batch normalization would generate the outputs such that they have the same mean value.

**Problem 1.2 – False**

Explanation: With the help of PyTorch, as long as we have defined our optimizer and loss function, we only need to write the code “loss.backward()” to let PyTorch help us compute gradients with back-propagation. Therefore, we do not need to write our own code for back-propagation.

**Problem 1.3 – False**

Explanation: Data augmentation is suitable for deep neural network, because deep neural network learns better with more data. However, data augmentation may hurt small models as they may underfit. Therefore, data augmentation is not always beneficial for all kinds of CNN models.

**Problem 1.4 – False**

Explanation: Dropout is enabled during the training phase with the purpose of combatting overfitting. Dropout will introduce extra computation cost during model training, and dropout does not decide whether a CNN model would converge (quickly) or not.

**Problem 1.5 – False**

Explanation: Dropout sometimes does not cooperate well with L-norm regularization.

**Problem 1.6 – False**

Explanation: A high sparsity means that there are many 0s in weight parameters. L2 regularization has a circular loss contour, while L1 regularization has a diamond loss contour. L1 loss usually meets the feasible solution loss on the corner of the diamond shape. Therefore, L1 regularization is more likely to obtain sparse weights.

Solutions continue in the next page.

**Problem 1.7 – True**

Explanation: Compared with ReLU, leaky ReLU could solve the problem of dead neurons because its negative side is not zero. However, the inconsistent slope makes the training process of some neural networks unstable.

**Problem 1.8 – True**

Explanation: Compare 3 \* 3 regular convolution vs. 3 \* 3 depthwise-separable convolution, the theoretical speedup could be calculated below.

If N and M are large enough, the theoretical speedup is approximately to be 9.

**Problem 1.9 – True**

Explanation: One strategy of SqueezeNet is to down-sample late in the network to spend more computation budgets (MACs) on larger activation maps. Therefore, it is correct to say that SqueezeNet puts most of the computations in later stage of the CNN design.

**Problem 1.10 – False**

Explanation: Shortcut connections in ResNet make the loss surface smoother and easier to optimize.

Solutions for Lab (1) are in the next page.

**2 Lab (1): Training SimpleNN for CIFAR-10 classification (15 pts)**

Text

Description automatically generated

Question (a) Solution:

To address the 2 hints given in this problem, I made use of some of my codes written in Homework 1. First, I randomly generated 5 data points with the image format (RGB channels \* width \* height = 3 \* 32 \* 32). Then I passed them through my SimpleNN model. The model’s output has a shape of (5, 10), where 5 is the number of data points and 10 is the number of output channels of the third fully connected layer as well as the total number of classes of all CIFAR-10 images. Second, I created a table to show the shape and the total number of weight parameters of each layer in my SimpleNN. The table is shown below.

|  |  |  |
| --- | --- | --- |
| **Layer Name** | **Layer Shape** | **Number of Weight Parameters** |
| CONV1 | (8, 3, 5, 5) | 600 |
| CONV2 | (16, 8, 3, 3) | 1152 |
| FC1 | (120, 576) | 69120 |
| FC2 | (84, 120) | 10080 |
| FC3 | (10, 84) | 840 |

For CONV1, 8 \* 3 \* 5 \* 5 = 600.

For CONV2, 16 \* 8 \* 3 \* 3 = 1152.

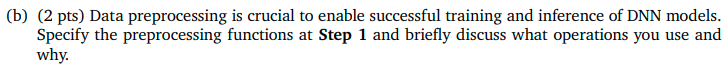
For FC1, 120 \* 576 = 69120.

For FC2, 84 \* 120 = 10080.

For FC3, 10 \* 84 = 840.

The total number of parameters of all layers is 600 + 1152 + 69120 + 10080 + 840 = 81792.

Given the table and calculations above, it is very easy to demonstrate the second hint.



Question (b) Solution:

I used two operations for data preprocessing. First, I used the “ToTensor” function because I want to convert my input NumPy array (image dimension information stored in multidimensional arrays) to a torch tensor, as well as to normalize my array (image pixel values) to a specific range from 0 to 1 (including both 0 and 1). Second, given the hint for this problem, I used the “Normalize” function because I want to normalize my input with the recommended mean and standard deviation for all the red, green and blue (RGB) channels.

Text, letter

Description automatically generated

Question (c) has been completed in code block Step 2.



Question (d) Solution:

Before model deployment, I checked whether “cuda” was available. My computer has a RTX 2070, on which I can instantiate and deploy my SimpleNN model. To verify that my model is indeed deployed on my GPU, I typed “nvidia-smi” in the Command Line to check my GPU status before and during model training.

Text

Description automatically generated

A screenshot of a computer

Description automatically generated with medium confidence

Two figures are shown above. The first figure shows my GPU status when my SimpleNN model is not training, and the second figure shows my GPU status when I am training the model. The differences among the above two figures could be concluded in the table below.

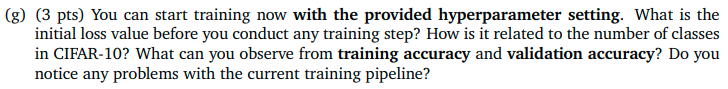
|  |  |  |
| --- | --- | --- |
| Model Status | Not Training | Training |
| GPU Power Usage | 18W | 29W |
| GPU Memory Usage | 161MiB / 8192MiB | 1139MiB / 8192MiB |
| GPU Utility | 0% | 36% |
| GPU Processes | No running processes found | Process name – python.exe |

Therefore, my GPU is indeed working during my SimpleNN model training.

Text

Description automatically generated

Question (e) and (f) have been completed in code blocks Step 4 and Step 5.



Question (g) Solution:

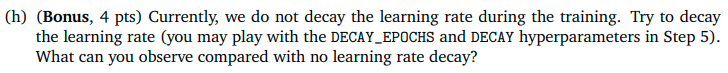
The initial loss value before any training step is 2.3086, which is close to 2.31. For SimpleNN model, I use the cross-entropy loss as the loss function, and CIFAR-10 image dataset has a total number of 10 classes. Before any training, the model weights are initialized with evenly random numbers, and the model itself is just randomly guessing the output. Therefore, will be approximately equal to , and the initial loss would be around the value of .

Question (g) Solution continues in the next page.

Chart, line chart

Description automatically generated

Given the printing result after running code blocks, the best validation accuracy is 65.84%, which is better than 63%. Given the figure shown above, we can observe that the training accuracy continuously increases with more epochs, from around 31% to approximately 90%. However, the validation accuracy slowly increases within the first 10 epochs, from around 42% to about 65%, then fluctuates or even decreases a bit for the rest of 20 epochs. Based on such observation, the current training pipeline would certainly result in model overfitting problem, as well as the problem that the trained model does not have a good validation accuracy.



Question (h) Solution:

I modified the DECAY\_EPOCHS to 5 and DECAY to 0.9, which means every 5 epochs, the learning rate will decay to 90% of its current value. Given the printing result after running code blocks, the best validation accuracy is 67.42%, which is slightly better than 65.84% with no learning rate decay.

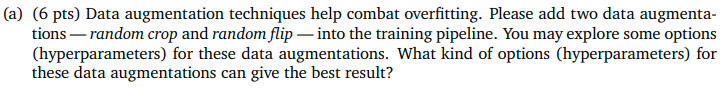
Question (h) Solution continues in the next page.

Chart, line chart

Description automatically generated

Given the figure above, compared with no learning rate decay model, the best training accuracy and validation accuracy of learning rate decay model are both slightly higher (93% vs. 90% and 67.42% vs. 65.84%), and it takes around 2 more epochs to reach model overfitting. However, the learning rate decay model still has the same two problems as the no learning rate decay model: bad validation accuracy and serious model overfitting.

**3 Lab (2): Improving the training pipeline (35 pts)**



Question (a) Solution:

Incorporating data augmentation into the training pipeline, after several attempts with different hyperparameters, I found that a probability of 0.5 (p = 0.5) for random horizontal flip and setting pad\_if\_needed as “True” would explicitly improve my model’s performance. According to the printing result after running the code block for training and validation, the best validation accuracy is 70.06%, which is better than both models in Lab (1).

Question (a) Solution continues in the next page.

Chart, line chart

Description automatically generated

Given the figure above, it is also clear that although the training accuracy can not reach 90% after 30 epochs, the model encounters overfitting problem after about 20 epochs, which demonstrates that data augmentation techniques can indeed help combat overfitting.



**Problem (b) Part 1 – Add a batch normalization layer after each convolution layer. What can you observe from the training process?**

Solution: Based on the printing result after running the training code block, the best validation accuracy is 71.82%. The figure illustrating the training process is shown below.

Chart, line chart

Description automatically generated

Given the figure shown above, my observation is that batch normalization slightly improves both training accuracies and validation accuracies, and makes model converge in a relatively faster way. Under the same training protocol, batch normalization model achieves 71.82% validation accuracy, approximately 2% better than 70.06%, the validation accuracy of model trained in Lab 2 (a).

Note: Although (a) uses 30 epochs and (b) uses 35 epochs, both models have achieved their best validation accuracies within 30 epochs, therefore their training protocols could be considered “the same”.

**Problem (b) Part 2 – Use empirical results to show that batch normalization allows a larger learning rate.**

Solution: To collect empirical results, I changed the initial learning rate to 0.1, which was 10 times larger than the previously defined initial learning rate. I also cancelled the learning rate decay during training. The figure illustrating the training process is shown below.

Chart, line chart

Description automatically generated

As the above figure shows, the model could still achieve 69.50% validation accuracy after 30 epochs, which shows that batch normalization indeed allows a larger learning rate, given other training protocols are similar.

Conceptual Analysis – To conceptually analyze why batch normalization could allow a larger learning rate, I start with the fact that although a large learning rate could speed up model training, it makes the model more susceptible to encountering either gradient vanishing or gradient exploding during back propagation. Batch normalization, however, makes layers’ parameters normalized, therefore preventing some layers’ parameters from being too small or too large, and finally resulting in less probability for gradient vanishing or exploding.

**Problem (b) Part 3 – Implement Swish [2] activation on your own and replace all of the ReLU activations in SimpleNN to Swish. Does it improve the performance of SimpleNN?**

Solution: Since the formula of Swish activation is very similar to the Sigmoid activation function, I implemented the Swish activation function directly by multiplying the input tensor with its Sigmoid in the element-wise format.

Chart, line chart

Description automatically generated

Given the above figure and printing result after running model training and validation block, the best validation accuracy is 71.80%, which is similar to the previously found best validation accuracy, 71.82%. Therefore, I think replacing all ReLU activation functions with Swish does not significantly improve the performance of my SimpleNN model.

Text

Description automatically generated

Problem (c) Part 1 Solution: To solve this problem, I trained and validated my SimpleNN model eight times given the eight learning rate values respectively. For other hyperparameters and model development techniques, I removed the learning rate decay which may influence model validation results considering learning rates themselves. However, I still kept the data augmentation, batch normalization, and Swish activation functions in my SimpleNN. As for the total number of epochs, I decided to adjust it flexibly based on my learning rates. If the learning rate is too large, I may make the total number of epochs small because the model would easily fail. If the learning rate is too small, I may increase the total number of epochs as long as it is within a maximation training budget of 100 epochs.

Report results for each model given different learning rate values.

1. Learning Rate = 1.0, Epochs = 30.

The best validation accuracy is 10.40%.

Chart, line chart

Description automatically generated

1. Learning Rate = 0.1, Epochs = 30.

The best validation accuracy is 69.74%.

Chart, line chart

Description automatically generated

1. Learning Rate = 0.05, Epochs = 30.

The best validation accuracy is 70.48%.

Chart, line chart

Description automatically generated

1. Learning Rate = 0.02, Epochs = 30.

The best validation accuracy is 71.82%.

Chart, line chart

Description automatically generated

1. Learning Rate = 0.01, Epochs = 30.

The best validation accuracy is 71.84%.

Chart, line chart

Description automatically generated

1. Learning Rate = 0.005, Epochs = 50.

The best validation accuracy is 72.12%.

Chart, line chart

Description automatically generated

1. Learning Rate = 0.002, Epochs = 70.

The best validation accuracy is 72.70%.

Chart, line chart

Description automatically generated

1. Learning Rate = 0.001, Epochs = 95.

The best validation accuracy is 72.58%.

A picture containing table

Description automatically generated

Analysis and conclusion of this problem continues in the next page.

A table that records the performance and hyperparameters of all above eight models is shown below.

|  |  |  |  |
| --- | --- | --- | --- |
| Model Number | Learning Rate | Total Number of Epochs | Best Validation Accuracy |
| Model 1 | 1.0 | 30 | 10.40% |
| Model 2 | 0.1 | 30 | 69.74% |
| Model 3 | 0.05 | 30 | 70.48% |
| Model 4 | 0.02 | 30 | 71.82% |
| Model 5 | 0.01 | 30 | 71.84% |
| Model 6 | 0.005 | 50 | 72.12% |
| **Model 7** | **0.002** | **70** | **72.70%** |
| Model 8 | 0.001 | 95 | 72.58% |

According to the above table, a large learning rate is not usually beneficial for model training. Although a large learning rate could speed up model training, it may cause model to encounter either gradient vanishing or gradient exploding, and finally result in model training failure (very bad training and validation accuracy). Although a relatively small learning rate may usually lead model training to convergence with relatively good training and validation accuracies, it needs more training epochs and therefore requires more computational power/resource/time, which also jeopardizes the trade-off between model accuracy and training cost. To conclude, the learning rate should neither be too large nor too small, it should be defined after several adjusting experiments to balance the aforementioned trade-off.

Text

Description automatically generated

Problem (c) Part 2 Solution: To solve this problem, I trained and validated my SimpleNN model 5 times given different L2 regularization strengths. I decided to apply other hyperparameters of Model 5 trained in Problem (c) Part 1, this model has a learning rate of 0.01 and 30 epochs, with the best validation accuracy of 71.84% (this model also contains data augmentation, batch normalization and Swish activation as usual). Although this is not the model with the best validation accuracy that I ever trained, 30 epochs of training could save a lot of computational power or time while maintaining relatively good validation accuracy.

Problem (c) Part 2 Solution continues in the next page.

Report the results for each regularization strength value.

1. L2 Regularization Value = 0.01.

The best validation accuracy is 65.44%.

Chart, line chart

Description automatically generated

1. L2 Regularization Value = 0.001.

The best validation accuracy is 73.08%.

Chart, line chart

Description automatically generated

1. L2 Regularization Value = 0.0001.

The best validation accuracy is 72.88%.

Chart, line chart

Description automatically generated

1. L2 Regularization Value = 0.00001.

The best validation accuracy is 72.24%.

Chart, line chart

Description automatically generated

1. L2 Regularization Value = 0.0.

The best validation accuracy is 72.84%.

Chart, line chart

Description automatically generated

A table that records the performance and hyperparameters of all above 5 models is shown below.

|  |  |  |
| --- | --- | --- |
| Model Number | L2 Regularization Value | Best Validation Accuracy |
| Model 1 | 0.01 | 65.44% |
| **Model 2** | **0.001** | **73.08%** |
| Model 3 | 0.0001 | 72.88% |
| Model 4 | 0.00001 | 72.24% |
| Model 5 | 0.0 | 72.84% |

Commentary on the importance of this hyperparameter – If the L2 regularization value is too large, which means the “punishment” upon large layer weight parameters will be severe, and therefore the model would have relatively small weight values after training. Given the above figures and summary table, this would make my model perform very unstably (validation accuracy curve fluctuates drastically for the largest L2 regularization value) during validation, as well as decrease the model’s validation accuracy. However, a properly chosen L2 regularization value could help my model combat overfitting, and therefore improve my model’s performance during validation, in terms of a higher validation accuracy shown in the above table. All in all, L2 regularization generates more stable results without too much tuning effort, which is also the reason why applying L2 regularization could not significantly improve model performance during validation, compared with the model which does not have any L2 regularization.

Text

Description automatically generated

Problem (c) Part 3 (Bonus) Solution: To compare the distribution of weight parameters after L1/L2 regularization, I decided to draw weight histogram for each layer in my SimpleNN after model training is complete.

Below are five weight histogram figures for my SimpleNN given L2 regularization value is 0.001.

Chart, histogram

Description automatically generated

Chart, histogram

Description automatically generated

Chart, histogram

Description automatically generated

Chart, histogram

Description automatically generated

Chart, histogram

Description automatically generated

I also decided to set the L1 regularization value as 0.001. After going through the same training protocol as the above L2 regularization, the L1 regularization model’s best validation accuracy is 63.00%, which is relatively bad. However, since in this problem we focus on the difference of weight distribution between L1 and L2 regularization, I will not try to optimize the L1 regularization hyperparameter. Below are five weight histogram figures for my SimpleNN given L1 regularization value is 0.001.

Chart, histogram

Description automatically generated

Chart, histogram

Description automatically generated

A picture containing icon

Description automatically generated

Icon

Description automatically generated

Icon

Description automatically generated

Given the above ten figures, it is noticeable that for L2 regularization, weight values of each layer tend to have a normal distribution. There are many small weight values between -0.5 and 0.5, but a relatively large portion of them is not 0, so the weights of a model with L2 regularization are not sparse. As for L1 regularization, there are a very large portion of weights which are 0s, and this applies to all 5 convolutional and fully connected layers in my SimpleNN model. Therefore, the weights of my SimpleNN model with L1 regularization are very sparse.

**4 Lab (3): Advanced CNN architectures (20 pts)**

After building the ResNet-20 model in “resnet-cifar10.ipynb” file, I trained my model with the hyperparameters provided in the ResNet paper. I also added the learning rate decay and defined the maximum number of epochs as 200. The initial learning rate, 0.1, would decay to its 90% every 60 epochs.

However, the best validation accuracy of my ResNet-20 model is 88.38%, which is less than 90%.

A picture containing chart

Description automatically generated