ECE 661: Homework #2

Construct, Train, and Optimize CNN Models

Hai "Helen" Li
ECE Department, Duke University — September 18, 2022

Objectives

Homework #2 covers the contents of Lectures 05~08. This assignment includes basic knowledge about CNNs, detailed instructions on how to setup a training pipeline for training image classifiers on the CIFAR- 10 dataset, how to improve the training pipeline, and how to use advanced CNN architectures to improve the performance of image classifiers. In this assignment, you will gain hands-on experience training a neural network model on a real computer-vision dataset (i.e., CIFAR-10), while also learning techniques for improving the performance of your CNN model.

We encourage you to complete the Homework #2 on the JupyterLab server or Google CoLab since the model training will require the computing power of GPUs. When conducting the lab projects, actively referring to the **NumPy/PyTorch tutorial** slides on Sakai for instructions on the environment setup and NumPy/PyTorch utilities can be very helpful.

Warning: You are asked to complete the assignment independently.

This lab has 100 points plus 10 bonus points, yet your final score cannot exceed 100 points. The submission deadline will be **11:55pm, Monday, October 2**. We provide a template named simplenn-cifar10.ipynb to start with, and you are asked to develop your own code based on this template. You will need to submit three independent files including:

1. A self-contained PDF report, which provides answers to all the conceptual questions and clearly demonstrates all your lab results and observations. Remember, **do NOT generate PDF from your jupyter notebook to serve as the report**, which can increase the TA's burden of grading.
2. code.zip, a zipped code file which contains 3 jupyter notebooks simplenn-cifar10.ipynb, simplenn-cifar10-dev.ipynb, and resnet-cifar10.ipynb, respectively for the three labs.
3. predictions.csv, your predicted label for each image in the provided CIFAR-10 testing split. See detailed instructions in Lab (3).

Note that 20 percent of the grade will be deducted if the submissions doesn't follow the above guidance.

Note that TAs hold the right to adjust grading based on the returned homeworks. We make sure that the grading rule is consistent among all students. Also, the results given for the Labs (for example the reported accuracies) are obtained from the specific runtime when TAs were working on the answers. We do not expect you to get exactly the same numbers; yet, it is necessary that your results show the same trends/patterns/observations in order to receive full credits.

1 True/False Questions (30 pts)

For each question, please provide a short explanation to support your judgment.

Problem 1.1 (3 pts) Batch normalization normalizes the batch inputs by subtracting the mean, so the outputs of BN module have zero mean accordingly.

False. It depends on whether it is in training process or testing process. During training, batch normalization normalizes the inputs by subtracting the batch mean and dividing by batch variance, which makes the activations in each layer to have zero mean and unit variance. However in testing phase, accumulated moving mean and moving average are used to do batch normalization, instead of directly using information from test data, so the outputs might not have zero mean if the distribution of test data deviates a lot from the training data.

Problem 1.2 (3 pts) PyTorch provides an efficient way of tensor computation and many modularized implementation of layers. As a result, you do not necessarily need to write your own code for standard back-propagation algorithms like Adam.

True. Pytorch has many built-in layers and functions. It has implemented many optimizers including Adagrad, Adam, ASGD, RMSprop, SGD, etc. Call *torch.optim.Adam* to use Adam optimizer.

Problem 1.3 (3 pts) Data augmentation techniques are always beneficial for any kinds of CNNs and any kinds of images.

False. While data augmentation techniques can be beneficial for improving the generalization of CNN, it's not always beneficial for any kinds of CNNs or images. For small models, data augmentation might hurt its performance since the model may underfit. For certain images such as digital numbers, data augmentation may corrupt the dataset if the images after data augmentation are not reasonable.

Problem 1.4 (3 pts) Without batch normalization, the CNNs can hardly or at least converge very slowly during the training. This is also true for dropout.

False. While BN is helpful for improving model performance, it's not true that CNN can hardly converge without BN. As for dropout, it is used for regularization to reduce overfitting, but it cannot guarantee convergence. In fact, extra computation cost will be introduced in training.

Problem 1.5 (3 pts) Dropout is a common technique to combat overfitting. If L-normalizations are further incorporated at the same time, the performance can be even better.

False. Both dropout and L-normalization are used to combat overfitting. But the interaction between dropout and L- normalization can be complex, sometimes dropout does not cooperate well with L-norm regularization.

Problem 1.6 (3 pts) During training, Lasso (L1) regularizer makes the model to have a higher sparsity compared to Ridge (L2) regularizer.

True. In L1 normalization, many weights will be zero, so it tends to produce sparse weight vectors while L2 normalization tends to produce small but non-zero weights. Therefore, L1 regularizer makes the model have a higher sparsity compared to L2 regularizer.

Problem 1.7 (3 pts) Though leaky ReLU solves the problem of dead neurons compared to vanilla ReLU, it could makes training unstable.

True. Leaky ReLU solves the "dead neuron" problem by adding a small slope for negative inputs so that the negative neurons can update related weights. But the inconsistent slope will also make the training process of some NN architectures unstable.

Problem 1.8 (3 pts) MobileNets use depthwise separable convolution to improve the model efficiency. If we replace all of the 3x3 convolution layers to 3x3 depthwise separable convolution layers in ResNet architectures, we are likely to observe approximately 9x speedup for these layers.

True. Theoretically, a regular 3×3 convolution has $3 \times 3 \times M \times N \times D_F \times D_F$ MACs and $3 \times 3 \times M \times N$ parameters, while a depthwise separable convolution has $3 \times 3 \times M \times D_F \times D_F + D_F \times D_F \times M \times N$ MACs and $3 \times 3 \times M + M \times N$ parameters. When N, M is large, the theoretical speedup is calculated as $\frac{3\times3\times M\times N\times D_F\times D_F}{3\times3\times M\times D_F\times D_F+D_F\times D_F\times M\times N} \approx 9$

Problem 1.9 (3 pts) To achieve fewer parameters than early CNN designs (e.g., AlexNet) while maintaining comparable performance, SqueezeNet puts most of the computations in the later stage of the CNN design.

False. SqueezeNet down-sample late in the network to spend more computation budgets (MACs) on larger activation maps. This indicates that SqueezeNet stacks more convolutional layers at the early stage of the CNN architecture, not the later stage.

Problem 1.10 (3 pts) The shortcut connections in ResNets result in smoother loss surface.

True. From Lecture-7 page-29, the loss face with shortcut connections are smoother and easy to optimize. The optimization process is easy to find the sharp minima.

2 Lab (1): Training SimpleNN for CIFAR-10 classification (15+4 pts)

Just like in HW1, here we start with a simple CNN architecture which we term as SimpleNN. It is composed of 2 CONV layers, 2 POOL layers and 3 FC layers. The detailed structure of this model is shown in Table 1.

Name	Type	Kernel size	depth/units	Activation	Strides
Conv 1	Convolution	5	8	ReLU	1
MaxPool	MaxPool	2	N/A	N/A	2
Conv 2	Convolution	3	16	ReLU	1
MaxPool	MaxPool	2	N/A	N/A	2
FC1	Fully-connected	N/A	120	ReLU	N/A
FC2	Fully-connected	N/A	84	ReLU	N/A
FC3	Fully-connected	N/A	10	None	N/A

Table 1: SimpleNN structure. No padding is applied on both convolution layers. A flatten layer is required before FC1 to reshape the feature.

In this lab, beyond model implementation, you will learn to set up the whole training pipeline and actually train a classifier to perform image classification on the CIFAR-10 dataset [1]. CIFAR-10 is one of the most famous/popular benchmarks for image recognition/classification. It consists of 10 categories (e.g., bird, dog, car, airplane) with 32x32 RGB images. You may go to the official website for more information https://www.cs.toronto.edu/~kriz/cifar.html.

In this assignment, please refer to Jupyter Notebook simplenn-cifar10.ipynb for detailed instructions on how to construct a training pipeline for SimpleNN model. **Note, remember to unzip the provided** tools.zip **to your workspace before getting started.**

(a) (2 pts) As a sanity check, we should verify the implementation of the SimpleNN model at **Step 0**. How can you check whether the model is implemented correctly?

Hint: 1) Consider creating dummy inputs that are of the same size as CIFAR-10 images, passing them through the model, and see if the model's outputs are of the correct shape. 2) Count the total number of parameters of all conv/FC layers and see if it meets your expectation.

ANSWER:

As sanity check, I generated a random tensor with the same size of CIFAR-10 ($3 \times 32 \times 32$), then I pass it through the self-defined SimpleNN model, the shape of the model output is [1, 10], which corresponds to the 10 categories in CIFAR-10. I also calculated the number of parameters in each layer (see Table 2) and they all meet my expectations:

> For convolutional layer, the total weight elements align with the formula: $C_2 \times C_1 \times K \times K$. For example, 600=8*3*5*5 for convolutional layer 1. Other convolutional layers were also verified.
> As for fully connected layer, the total weight elements align with the formula: $Input \times Output$. For example, 69120=16*6*6*120 for fully connected layer 1. Other fully-connected layers were also verified.

As a result, the model is implemented correctly.

# of parameters	Conv1	Conv2	Fc1	Fc2	Fc3
Weight	600	1152	69120	10080	840
Bias	8	16	120	84	10

Table 2: The number of parameters of weight and bias in each layer.

(b) (2 pts) Data preprocessing is crucial to enable successful training and inference of DNN models. Specify the preprocessing functions at **Step 1** and briefly discuss what operations you use and why.

ANSWER:
I performed two preprocessing functions to both training and validation datasets.

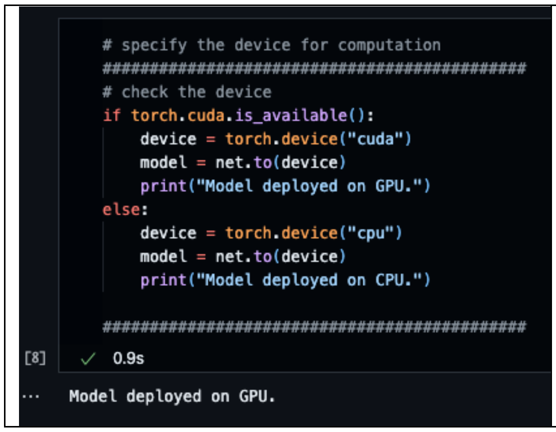
1) **Conversion to Tensor**: Images were initially in PIL format. They were converted to tensors to make them compatible with PyTorch and GPU usage.
2) **Normalization**: Tensors were normalized using the mean and standard deviation values of the CIFAR-10 dataset. This step standardizes the pixel values, which can lead to better and faster convergence during training.

(c) (2 pts) During the training, we need to feed data to the model, which requires an efficient data loading process. This is typically achieved by setting up a dataset and a dataloader. Please go to **Step 2** and build the actual training/validation datasets and dataloaders. Note, instead of using the CIFAR10 dataset class from torchvision.datasets, here you are asked to use our own CIFAR-10 dataset class, which is imported from tools.dataset. As for the dataloader, we encourage you to use torch.utils.data.DataLoader.

In notebook *simplenn-cifar10.ipynb*, I use torch.utils.data.DataLoader to download and load dataset**.**

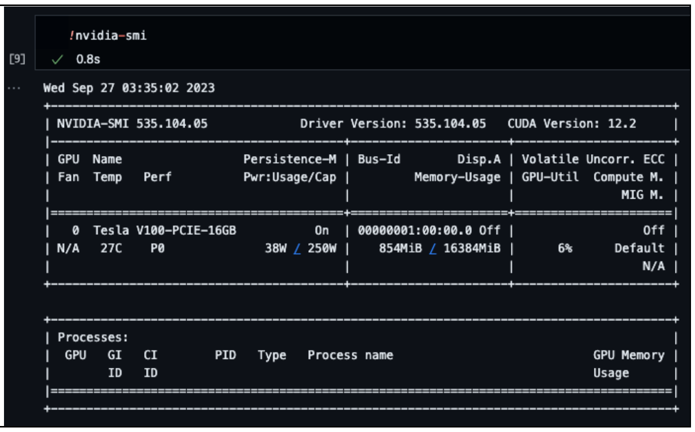
(d) (2 pts) Go to **Step 3** to instantiate and deploy the SimpleNN model on GPUs for efficient training. How can you verify that your model is indeed deployed on GPU? (Hint: use nvidia-smi command in the terminal)

ANSWER:
I print out the current device and also use the command !nvidia-smi to check the current device. It shows that the GPU is available, the CUDA version is 12.2 and model has been deployed to GPU.



```
# specify the device for computation
#############################################
# check the device
if torch.cuda.is_available():
    device = torch.device("cuda")
    model = net.to(device)
    print("Model deployed on GPU.")
else:
    device = torch.device("cpu")
    model = net.to(device)
    print("Model deployed on CPU.")

#############################################
[8]  ✓ 0.9s
...  Model deployed on GPU.
```



```
!nvidia-smi
[9]  ✓ 0.8s
...
Wed Sep 27 03:35:02 2023
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Tesla V100-PCIE-16GB           On  | 00000001:00:00.0 Off |                  Off |
| N/A   27C    P0              38W / 250W |    854MiB / 16384MiB |      6%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
+---------------------------------------------------------------------------------------+
```

(e) (2pts) Loss functions are used to encode the learning objective. Now, we need to define this problem's loss function as well as the optimizer which will update our model's parameters to minimize the loss. In **Step 4**, please fill out the loss function and optimizer part.

In notebook *simplenn-cifar10.ipynb*, I use cross entropy as loss function and SDG as optimizer.

(f) (2 pts) Please go to **Step 5** to set up the training process of SimpleNN on the CIFAR-10 dataset. Follow the detailed instructions in **Step 5** for guidance.

In notebook *simplenn-cifar10.ipynb*, I set up the training process and the model runs successfully. The best validation accuracy reached 66.06%.

(g) (3pts)You can start training now **with the provided hyperparameter setting**. What is the initial loss value before you conduct any training step? How is it related to the number of classes in CIFAR-10? What can you observe from **training accuracy** and **validation accuracy**? Do you notice any problems with the current training pipeline?

ANSWER:
The initial loss value should be -ln(1/10)=2.3. This is because the model is merely random guess before training, so the possibility of each label is the same and should be the average of n categories.

The training accuracy kept increasing during the training process while the validate accuracy plateaued after a certain point and began to fluctuate around this value, even showing signs of decreasing at certain points.

Training accuracy is much higher than the validation accuracy, which indicates overfitting problem with the current training pipeline.

(h) (**Bonus**, 4 pts) Currently, we do not decay the learning rate during the training. Try to decay the learning rate (you may play with the DECAY_EPOCHS and DECAY hyperparameters in Step 5). What can you observe compared with no learning rate decay?

ANSWER:
I implemented learning rate decay by reducing the learning rate to 70% of its previous value every 10 epochs. Compared to the model trained without learning rate decay, the validation accuracy increased from the original 66.06% to 67.46%. From the training curve plot, a smoother convergence is also observed towards the end of training. But the improvement is not huge considering other problems such as overfitting, small model architecture, etc.

At the end of Lab 1, we expect at least 65% validation accuracy if all the steps are completed properly. You are required to submit the completed version of simplenn-cifar10.ipynb for Lab (1).

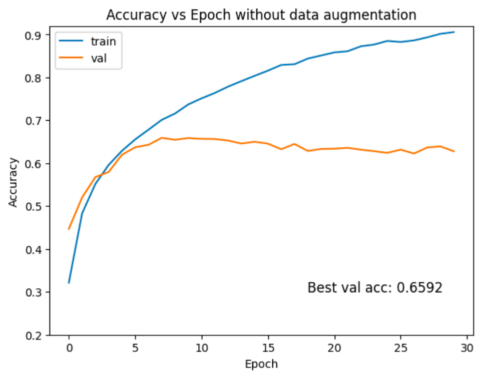
3 Lab (2): Improving the training pipeline (35+6 pts)

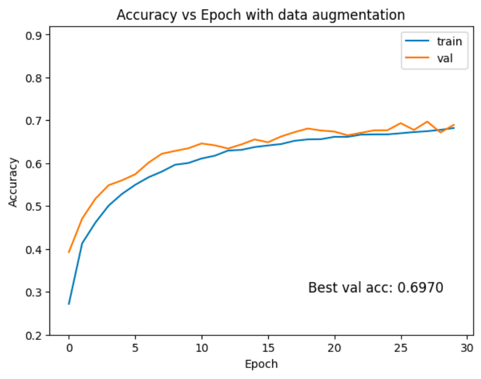
In Lab (1), we develop a simplified training pipeline. To obtain better training result, we will improve the training pipeline by employing data augmentation, improving the model design, and tuning the hyperparameters.

Before start, please duplicate the notebook in Lab (1) and name it as simplenn-cifar10-dev.ipynb**, and work on the new notebook**. You goal is to reach at least 70% validation accuracy on the CIFAR-10 dataset.

(a) (6 pts) Data augmentation techniques help combat overfitting. A typical strategy for CIFAR classification is to combine 1) *random cropping* with a *padding* of 4 and 2) *random flipping*. Train a model with such augmentation. How is the validation accuracy compared with the one without augmentation? **Note that in the following questions we all use augmentation. Also remember to reinitialize the model whenever you start a new training!**

ANSWER:

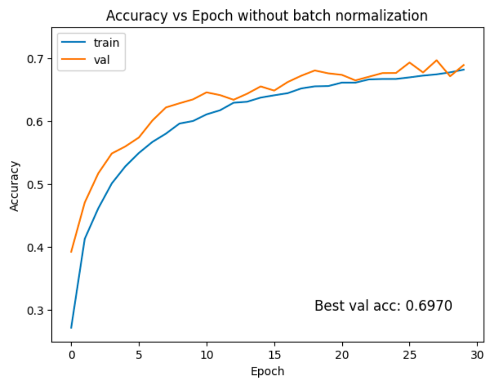


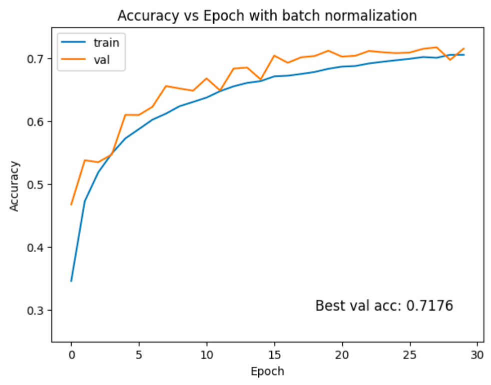


When implementing data augmentation (random cropping + random horizontal flipping), the best validation accuracy increased to 69.7% compared to the one without augmentation. Additionally, data augmentation can effectively reduce the overfitting problem in the model without augmentation.

(b) (15 pts) Model design is another important factor in determining performance on a given task. Now, modify the design of SimpleNN as instructed below:

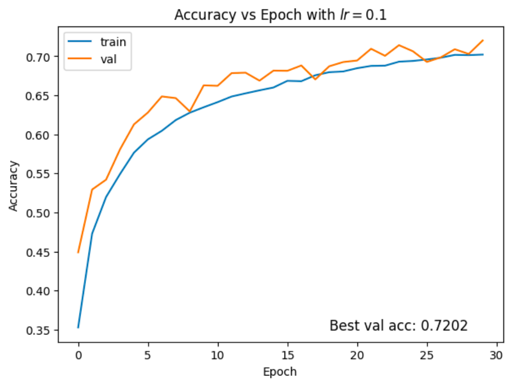
- (5 pts) Add a batch normalization (BN) layer after each convolution layer. Compared with no BN layers, how does the best validation accuracy change?





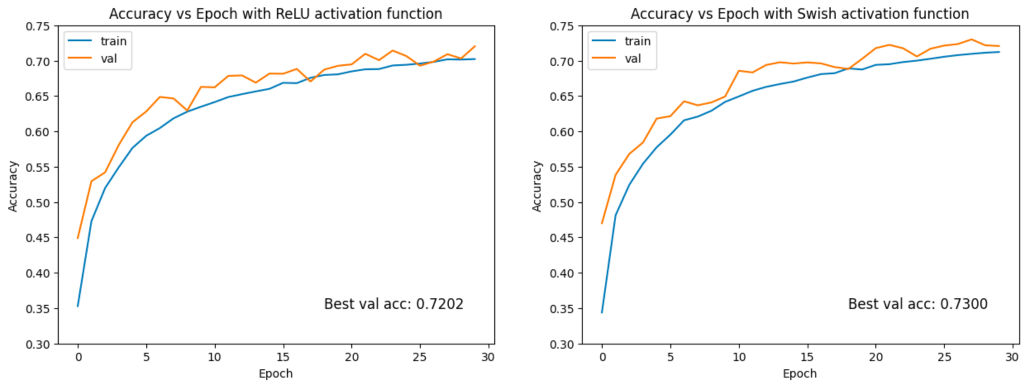
After adding a batch normalization layer after each convolution layer, the best validation accuracy increased to 71.76% compared to the one without BN (69.70%). This indicates that batch normalization played a beneficial role in the model's training process.

- ☐ (5 pts) Use empirical results to show that batch normalization allows a larger learning rate.



As mentioned in the last question, the model achieved a validation accuracy of 71.76% with a learning rate of 0.01. After increasing the learning rate to 0.1, instead of observing a decline in performance, the best validation accuracy even increased to 72.02%. This empirical observation suggests that batch normalization can effectively accommodate a higher learning rate.

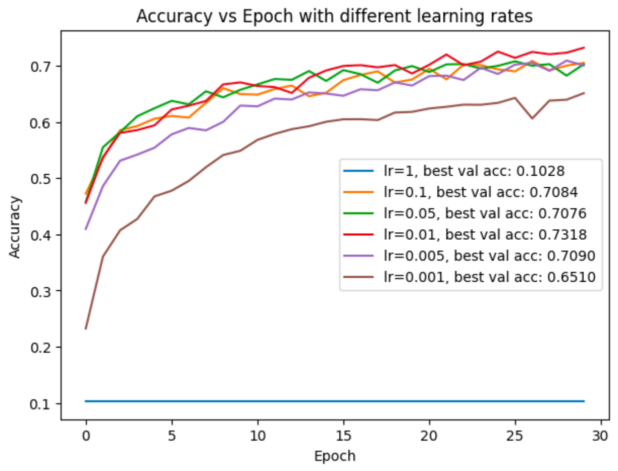
- ☐ (5 pts) Implement Swish [2] activation on you own, and replace all of the ReLU activations in SimpleNN to Swish. Train the model with BN layers and a learning rate of 0.1. Does Swish outperform ReLU?



Compared with the model with ReLU activation, which achieved a validation accuracy of 72.02%, the model with Swish activation has higher validation accuracy of 73%. This suggests that, in this context, Swish outperforms ReLU.

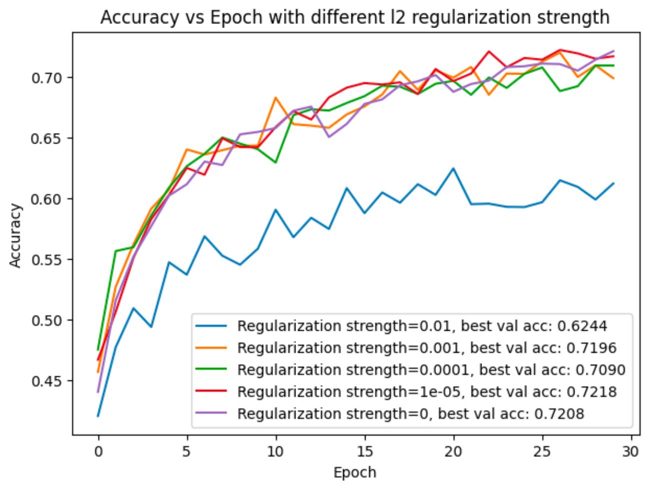
(c) (14 pts) Hyperparameter settings are very important and can have a large impact on the final model performance. Based on the improvements that you have made to the training pipeline thus far (with data augmentation and BN layers), tune some of the hyperparameters as instructed below:

- ☐ (7 pts) Apply different learning rate values: 1.0, 0.1, 0.05, 0.01, 0.005, 0.001, to see how the learning rate affects the model performance, and report results for each. Is a large learning rate beneficial for model training? If not, what can you conclude from the choice of learning rate?



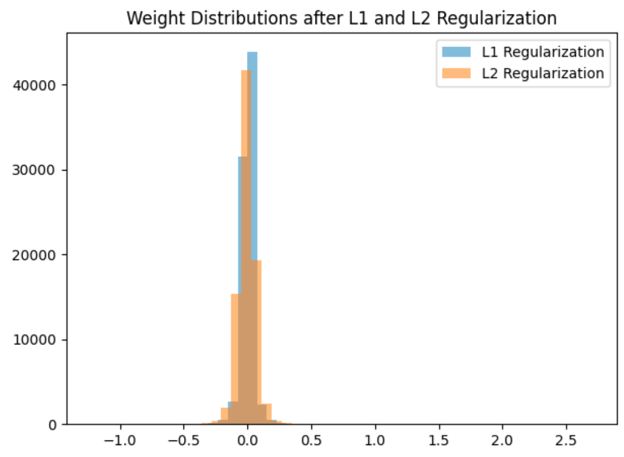
With a very high learning rate of lr=1.0, the model performance is notably poor, achieving an accuracy of just 0.1028. This is indicative of the model potentially overshooting the optimal points in the loss landscape and not converging properly. As we decrease learning rate, the accuracy starts to increase, and it peaks at lr=0.01. This suggests that this might be an optimal learning rate for the model, providing a good balance between convergence speed and accuracy. However, as we continue to decrease the learning rate further to lr=0.001, there's a noticeable drop in accuracy to 0.6510. This might indicate that the learning rate is too low, causing the model not be able to converge fast enough within the given epochs.

- ☐ (7 pts) Use different L2 regularization strengths of 1e-2, 1e-3, 1e-4, 1e-5, and 0.0 to see how the L2 regularization strength affects the model performance. In this problem use a learning rate of 0.01. Report the results for each regularization strength value along with comments on the importance of this hyperparameter.



A large weight_decay of 0.01 leads to stronger regularization. This might be too restrictive for the model, preventing it from fitting the training data well. The accuracy is the lowest in this setting, indicating potential underfitting. As the regularization strength decreased, the model's performance generally improved, reaching a peak around weight_decay=0.001 and 1e-05. When there is no regularization (weight_decay=0), the best accuracy slightly drops, indicating that a proper regularization can help reducing overfitting and thus improve model performance.

- ☐ (Bonus, 6 pts) Switch the regularization penalty from L2 penalty to L1 penalty. This means you may not use the weight_decay parameter in PyTorch builtin optimizers, as it does not support L1 regularization. Instead, you need to add L1 penalty as a part of the loss function. Compare the distribution of weight parameters after L1/L2 regularization. Describe your observations.



As shown in the weight distribution plot, there are more weight parameters reaching zero in L1 regularization compared to L2 regularization. This is because L1 regularization tends to drive some weight parameters exactly to zero, leading to a sparse model. On the other hand, L2 tends to shrink weights to near zero but doesn't necessarily force them to be exactly zero.

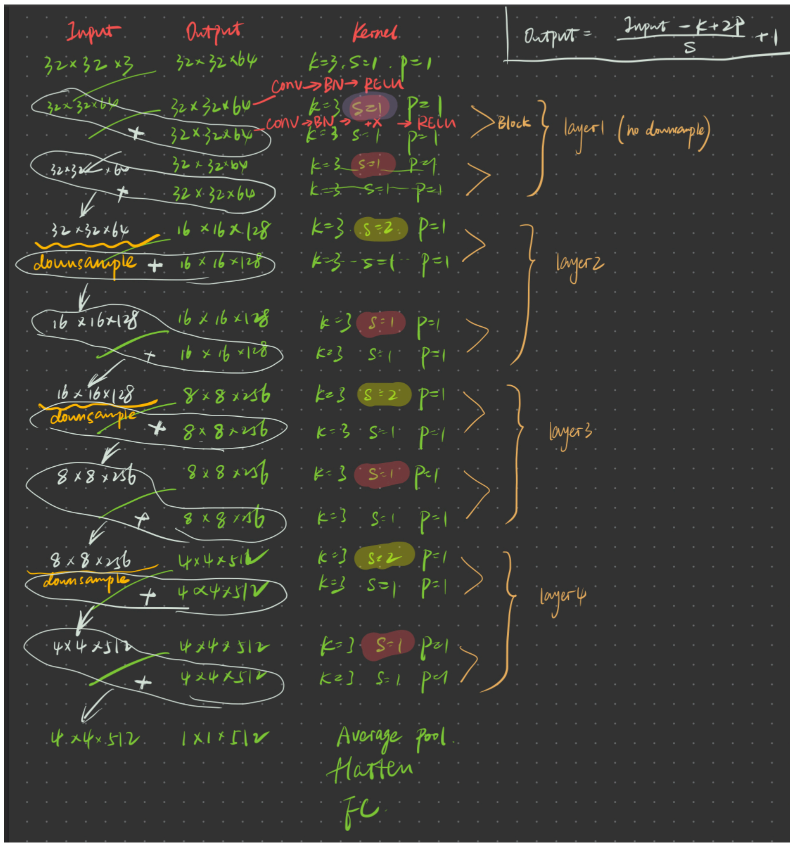
Up to now, you shall have an improved training pipeline for CIFAR-10. Remember, you are required to submit simplenn-cifar10-dev.ipynb for Lab (2).

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

The improved training pipeline for SimpleNN developed in Lab (2) still has limited performance. This is mainly because the SimpleNN has rather small capacity (learning capability) for CIFAR-10 task. Thus, in this lab we replace the SimpleNN model with a more advanced ResNet [3] architecture. We expect to see much higher accuracy on CIFAR-10 when using ResNets. **Here, you may duplicate your jupyter notebook for Lab (2) as** resnet-cifar10.ipynb **to serve as a starting point.**

(a) (8 pts) Implement the ResNet-20 architecture by following Section 4.2 of the ResNet paper [3]. This lab is designed to have you learn how to implement a DNN model yourself, **so do NOT borrow any code from online resource.**

ANSWER:
In notebook *resnet-cifar10.ipynb*, I implemented the ResNet-20 architecture. First, I created a basic block called *Residuel_Block* to store the *conv-bn-relu-conv-bn-skip sum-relu* structure. Then I created the ResNet20 class using the basic block to build the resnet-20 networks. I also used the following diagram to help me figure out the input, output shape of each layer.



Without any hyperparameter tuning, the default setting reached 90.16% accuracy on the validation dataset.

(b) (12 pts) Tune your ResNet-20 model to reach an accuracy of higher than 90% on the validation dataset. You may use all of the previous techniques that you have learned so far, including data augmentations, hyperparameter tuning, learning rate decay, etc. Training the model longer is also essential to obtaining good performance. You should be able to achieve >90% validation accuracy with a maximum of 200 epochs. **Remember to save your trained model during the training!!!** Check out this tutorial https://pytorch.org/tutorials/beginner/saving_loading_models.html on model saving/loading.

ANSWER:
During hyperparameter tuning, I tuned the learning rate, L2 regularization strength parameter, activation function, epochs, and add learning rate weight decay. The complete tuning process is shown in notebook *resnet-cifar10.ipynb*.

In the final model, I used swish function as activation, a total of 100 epochs, L2 regularization strength of 1e-4, learning rate starting from 0.05 and decay to its 80% every 10 epochs, which finally achieved 93.76% accuracy on the validation dataset.

We will grade this task by evaluating your trained model on the holdout testing dataset (which you do not have any labels). **After your ResNet-20 model is trained, you need to make predictions on test data, and save the predictions into the** predictions.csv **file**. Please use save_test_predictions.ipynb to save your predictions in required format. The saved file should look like the provided example sample_predictions.csv. Upon submission, we will directly compare your predicted labels with the ground-truth labels to compute your score.

After completing Lab (3), you are required to submit resnet-cifar10.ipynb and the your prediction results predictions.csv.

Info: Additional requirements:

- **DO NOT** train on the test set or use pretrained models to get unfair advantage. We have conducted a special preprocessing on the original CIFAR-10 dataset. As we have tested, “cheating” on the full dataset will give only 6% accuracy on our final test set, which means being unsuccessful in this assignment.
- **DO NOT** copy code directly online or from other classmates. We will check it! The result can be severe if your codes fail to pass our check.

Info: As this assignment requires much computing power of GPUs, we suggest:

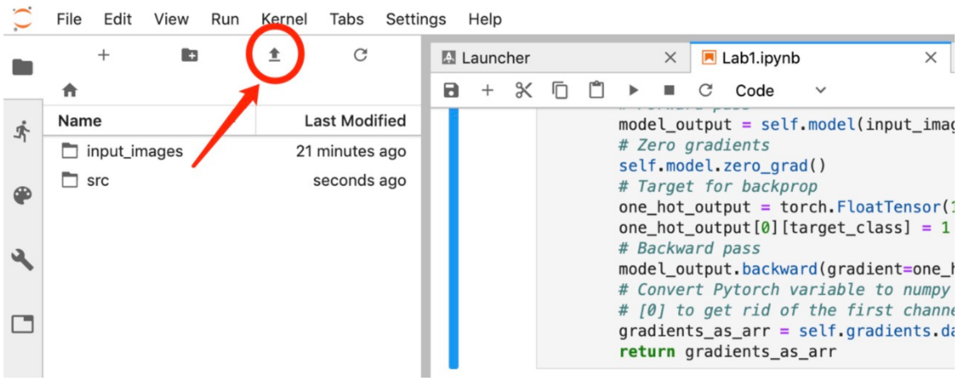
- Plan your work in advance and start early. We will **NOT** extend the deadline because of the unavailability of computing resources.
- Be considerate and kill Jupyter Notebook instances when you do not need them.
- **DO NOT** run your program forever. Please follow the recommended/maximum training budget in each lab.

References

[1] A. Krizhevsky, G. Hinton, *et al.*, “Learning multiple layers of features from tiny images,” 2009.
[2] P. Ramachandran, B. Zoph, and Q. V. Le, “Searching for activation functions,” *arXiv preprint arXiv:1710.05941*, 2017.
[3] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Appendix: Using the OIT Server

If you wish to finish the Lab questions on the OIT server, please visit https://vm-manage.oit.duke.edu/containers and log into your Jupyter Notebook Environment. You can upload the files you need to the server by clicking the button shown in Figure 1:



```
model_output = self.model(input_imag
# Zero gradients
self.model.zero_grad()
# Target for backprop
one_hot_output = torch.FloatTensor(1
one_hot_output[0][target_class] = 1
# Backward pass
model_output.backward(gradient=one_h
# Convert Pytorch variable to numpy
# [0] to get rid of the first channe
gradients_as_arr = self.gradients.da
return gradients_as_arr
```

Figure 1: Uploading files Instruction

If you are uploading an zip file, you may unzip it on the server by:

- Press the '+' button and click on "terminal" in the right-hand side "Launcher" column.
- In the terminal, type *unzip* $*$ *.zip*

Notice: After finishing the lab, please make sure you kill your current process by right-clicking on the `.ipynb` file and select "Shutdown Kernel", as shown in Figure 2:

Please note that there is a 30-minute idle timeout for GPU access set on the OIT server. If you find that you can no longer access the GPU due to the timeout, simply save your progress, log out, restart your browser and log back in, then you can keep working again.

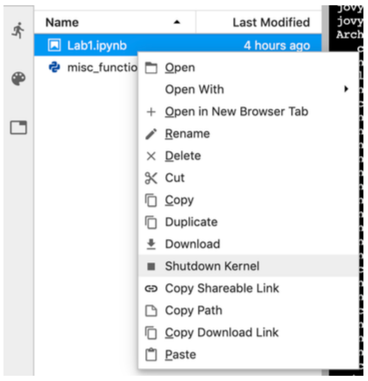


Figure 2: Shutdown kernel before exiting