ECE 661: Homework #2
Construct, Train, and Optimize CNN Models

Yiran Chen

ECE Department, Duke University — September 15, 2022

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, but the outputs of BN module do *not* necessarily have zero mean.

True. One example is that you can not use the batch statistics of the test data, such that you do not update moving mean/moving variance with test data! The output of the batch normalization model when performing inference will then not be zero mean, due to the differences in statistics between the training and testing distribution. The batch normalization does ensure the output is zero mean in expectation, however, differences in the distributions can affect the mean.

Problem 1.2 (3 pts) PyTorch provides an efficient way of tensor computation and many modularized implementation of layers. However, you still need to write your own code for back-propagation.

False, One just calculate the total loss, and then run loss.backward() to find the gradients and optimizer.step() to update the weights based on the gradients. PyTorch uses Autograd to perform the automatic differentiation and the step function to update- all one must do is run these commands.

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

False. Data augmentation may hurt small models as they may underfit (slide 28, lecture 6). Another way to think about it is in cases where the data is already sufficiently large and representative of what the testing distribution will be, or in cases like handwriting recognition where performing a flip can introduce confusion (between a 6b and a 9).

Problem 1.4 (3 pts) Without Dropout, the CNNs can hardly or at least converge very slowly during the training.

False. Dropout is used as a regularization technique to prevent overfitting, and has little to do with model convergence (however according to lecture 6, slide 11, it can actually introduce computational cost, meaning it may take more time to converge). In fact, according to lecture 6 slide 11, Batch Norm is prevalent but Dropout is not used that often anymore.

Problem 1.5 (3 pts) L-norm regularization can effectively combat overfitting. If Dropout is further incorporated at the same time, the performance can be even better.

False. Sometimes dropout does not cooperate well with L-norm regularization (slide 11, lecture 6). Thus, while Dropout and L-norm regularization are separately effective techniques to reduce overfitting, adding Dropout to L-norm regularization can actually hurt upon model performance.

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

True. Due to the diamond contour of L1 regularization, L1 loss isocontours usually meets the feasible solution loss on the corner of the diamond shape (where a parameter is 0), whereas with L2 the isocontours meet on the circumference of the circle (where a parameter may be shrunk to very tiny but non-zero) (slides 7-8. lecture 6). Thus, L2 actually performs feature selection, whereas L1 merely shrinks coefficients.

Problem 1.7 (3 pts) Compared with ReLU, leaky ReLU not only solves the problem of dead neurons, but also improves the training stability.

False. The inconsistent slope of Leaky ReLU can introduce instability into certain neural network architectures and makes the training process of some neural often increases convergence time (lecture 5, slide 17).

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. a 3×3 regular convolution has $3 \times 3 \times \mathrm{M} \times \mathrm{N} \times D_F \times D_F$ MACs and $3 \times 3 \times \mathrm{M} \times \mathrm{N}$ parameters. A depthwise separable convolution has $3 \times 3 \times \mathrm{M} \times D_F \times D_F + D_F \times D_F \times \mathrm{M} \times \mathrm{N}$ MACs and $3 \times 3 \times \mathrm{M} + \mathrm{M} \times \mathrm{N}$ parameters (lecture 8, slide 17). The regular convolution has 9 times as many MACs, creating a theretical speedup of 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.

True. The SqueezeNet downample late in the network to spend more computation (MACs) on larger activation maps (lecture 8, slide 8). It has a small architecture, uses 1x1 filter to save computation, and downsamples later to keep a larger feature map

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

True. As shown in slide 29, lecture 7, the shortcut connections of ResNet help the loss landscape remain smooth rather than chaotic, which is what allowed it to use deeper networks and improve the state-of-the-art at the time.

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

Lab 1 (15 points)

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.

The first check I perform is shape. I passing in a vector of size (5,3,32,32), representing 5 32x3x RGB images. I then expect 5 outputs of length 10, representing a vector of size 10 for each image with the softmax outputs for each class.

The second check I perform is parameters. I print out the module.weight shape, which includes the number of parameters in a given layer. In checking this with manual calculations at each layer of what we would expect the number of weights to be based on the given table, I can better know if everything was input correctly. The outputs from the print statements in the notebook should match the calculations below (and they do).

Conv 1: Weights = fmap_in (depth) * fmap_out (num kernels) * kernel_height * kernel_width = 3 * 8 * 5 * 5 = 600 weights

Input Conv 1: (3, 32,32) Output of Conv 1: (8,28,28) As p = 0, k = 5, s = 1, depth = 8. Subtract k-1 for image dimensions and change depth to new depth.

Output of Pool 1: (8,14,14) As s = 2. Divide image dimensions by 2.

Conv 2: Weights = fmap_in (depth) * fmap_out (num kernels) * kernel_height * kernel_width = 8 * 16 * 3 * 3 = 1152 weights

Output of Conv 2: (16,12,12) As p = 0, k = 3, s = 1, depth = 16. Subtract k-1 for image dimensions and change depth to new depth.

Output of Pool 2: (16,6,6) As s = 2. Divide image dimensions by 2.

FC 1: Weights = Input features * output features = (16 * 6 * 6) * 120 = 69120 weights

FC 2: Weights = Input features * output features = 120 * 84 = 10080 weights

FC 3: Weights = Input features * output features = 84 * 10 = 840 weights

(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.

First, I convert the input into torch tensors to have the data fit within the PyTorch training framework. I then normalize the inputs by the mean and standard deviation. This is important as it leads to faster training as it puts the various data features on a smaller scale and within the context of the data distribution. Further preprocessing (augmentations, etc) is not performed as this represents a Simple NN for Lab 1.

(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`.

Lab 2

(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)

In running nvidia-smi command, it is clear that after running the net.to(device) command, we see that the memory usage drastically increases, as the net has been deployed to the GPU. Thus, we see in the images below the first image has (0MiB / 2203 MiB) and second image has (1171MiB / 2203 MiB)

```
!nvidia-smi
```

```
Thu Sep 29 23:54:26 2022
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 495.29.05    Driver Version: 460.73.01    CUDA Version: 11.2     |
|-------------------------------+----------------------+----------------------+
| 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  GeForce RTX 208...  On   | 00000000:1B:00.0 Off |                  N/A |
| 29%   24C    P8    37W / 250W |      0MiB /  2203MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

```
net = net.to(device)
```

```
Run on GPU...
```

```
: !nvidia-smi
```

```
Thu Sep 29 23:55:32 2022
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 495.29.05    Driver Version: 460.73.01    CUDA Version: 11.2     |
|-------------------------------+----------------------+----------------------+
| 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  GeForce RTX 208...  On   | 00000000:1B:00.0 Off |                  N/A |
| 29%   27C    P2    67W / 250W |   1171MiB /  2203MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

Lab 3

(2 pts) 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.

(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.

(g) (3 pts) 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?

The initial loss value before I conduct any training step is 2.304, which is approximately equal to ln(0.1) (which is equal to 2.303). This is because in a random scenario, we would expect the prediction of each class to be 0.1 (since there are 10 classes), leading to the cross entropy being the sum of 0.1 * ln(0.1) for each of the ten classes, such that we have 10 * 0.1 * ln(0.1)= ln(0.1).

The current training pipeline does seem to have several problems. Firstly, it reaches the highest performance on validation around 10-20 epochs. The other epochs do not seem to improve performance. The training pipeline could benefit from hyperparameter tuning, as well as decreased learning rate as the epochs reach the 10-20 range in order to better search the loss landscape for a local minima. Additionally, the difference between train and validation performance could be remedied by some augmentations in order to provide the network more data. Finally, it seems that the network itself is not complex enough to truly surpass the high benchmarks set by deeper networks that reach 90 percent.

(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?

I notice that no learning rate decay model reaches the peak performance within 10-20 epochs, and does not really improve but rather stays around there in terms of validation performance. Meanwhile, introducing learning rate decay causes the later epochs to be smaller steps that do not overstep minima as much, and slowly but steadily eke out performance gain in the later epochs, as demonstrated by the light performance boost.

DECAY_EPOCHS	Decay	LR	Activation	L2 Reg	Val Accuracy
N/A	N/A	0.01	ReLU	1e-4	0.6570
10	0.1	0.01	ReLU	1e-4	0.6802
15	0.1	0.01	ReLU	1e-4	0.6668

At the end of Lab 1, we expect at least 64% 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 pts)

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.

Lab 4 (35 points)

(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!**

Decay?	LR	Augmentations	Batch Norm?	Activation	L2 Reg	Val Accuracy
N	0.01	N	N	ReLU	1e-4	0.6570
N	0.01	Y	N	ReLU	1e-4	0.7286

As shown in the table, there is a significant improvement in validation accuracy when introducing augmentations. This is likely due to the fact that augmentations introduce diversity to the dataset that helps account for the differences that may occur in the data distribution between train and validation, such as simple rotations/flips and shifts.

(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?

Decay?	LR	Augmentations	Batch Norm?	Activation	L2 Reg	Val Accuracy
N	0.01	Y	N	ReLU	1e-4	0.7286
N	0.01	Y	Y	ReLU	1e-4	0.7400

As shown in the table, there is a slight improvement in validation accuracy when introducing batch normalization.

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

As shown in the table below, increasing the learning rate increases the disparity between model performance when using versus batch normalization versus not. The models with batch normalization remain above 70 percent (and at 0.64 for a very large 0.2 learning rate), whereas the models without batch normalization quickly decrease in performance down to 50 percent and 30 percent. This demonstrates the power of batch normalization to allow for higher learning rates, which is especially helpful in speeding up the training process.

Decay?	LR	Augmentations	Batch Norm?	Activation	L2 Reg	Val Accuracy
N	0.05	Y	N	ReLU	1e-4	0.6454
N	0.1	Y	N	ReLU	1e-4	0.5218
N	0.2	Y	N	ReLU	1e-4	0.3182
N	0.05	Y	Y	ReLU	1e-4	0.7314
N	0.1	Y	Y	ReLU	1e-4	0.7012
N	0.2	Y	Y	ReLU	1e-4	0.6430

- (5 pts) Implement Swish [**?**] 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?

Decay?	LR	Augmentations	Batch Norm?	Activation	L2 Reg	Val Accuracy
N	0.01	Y	Y	ReLU	1e-4	0.7400
N	0.01	Y	Y	Swish	1e-4	0.7526

As shown in the table, there is a slight improvement in validation accuracy when introducing the Swish activation. With a single run of these hyperparameters, Swish outperformed 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:

Lab 5

- (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?

Decay?	LR	Augmentations	Batch Norm?	Activation	L2 Reg	Val Accuracy
N	0.001	Y	Y	ReLU	1e-4	0.6830
N	0.005	Y	Y	ReLU	1e-4	0.7288
N	0.01	Y	Y	ReLU	1e-4	0.7400
N	0.05	Y	Y	ReLU	1e-4	0.7314
N	0.1	Y	Y	ReLU	1e-4	0.7012
N	1	Y	Y	ReLU	1e-4	0.1054

  A larger learning rate is not necessarily better, as demonstrated by the poor performance of the model with learning rate of 1. Smaller learning rates take longer to converge, such that the smaller learning rates here likely have not yet converged. Meanwhile, larger learning rates are generally able to quickly reduce the loss, however, they often overstep local minima. Thus, it is ideal to start with a larger learning rate to help reduce the loss, before lowering the learning rate as the model does not improve in order to allow it to not step too far as it approaches the minima.

- (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.

Decay?	LR	Augmentations	Batch Norm?	Activation	L2 Reg	Val Accuracy
N	0.1	Y	Y	ReLU	0.0	0.7352
N	0.1	Y	Y	ReLU	1e-5	0.7374
N	0.1	Y	Y	ReLU	1e-4	0.7400
N	0.1	Y	Y	ReLU	1e-3	0.7308
N	0.1	Y	Y	ReLU	1e-2	0.6624

- (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.

  Images of distributions added to appendix. Based on the visualizations, the weights for each layer using L1 normalization are unimodal with a large amount of variables shrunk to near zero. While they are unimodal, they do not appear to be approximately normal given the dominatingly high concentration of values at the mode of the distribution. Meanwhile, with L2 normalization, the peak of the distribution is a lot wider (as many variables are shrunk to close to zero but do not fall within the bin). There is higher variance in these distributions, and distributions such as the FC1, FC2, and Conv 2 distributions even appear approximately normal.

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 [**?**] 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.**

Lab 6 (20 points)

(a) (8 pts) Implement the ResNet-20 architecture by following Section 4.2 of the ResNet paper [**?**]. Note that you can use either "option A" or "option B" when the input and output shape don't match in the shortcut (see Section 3.3 "Residual Network"). This lab is designed to have you learn how to implement a DNN model yourself, **so do NOT borrow any code from online resource.**

(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.

We will grade this task mainly 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`. We will compare your predictions with the ground-truth labels to evaluate your model performance. To this end we host a kaggle competition at https://www.kaggle.com/t/a6201262c9884fe3870dec7760576577, where you need to submit your `predictions.csv`.

In addition, in the PDF report, you are asked to comment on which techniques you find most useful (in terms of validation/test accuracy) when training the ResNet. If you make observations and write down your thoughts/analyses/hypothesis, you will get good score on this question even if your model do not achieve the highest accuracy (which requires some luck).

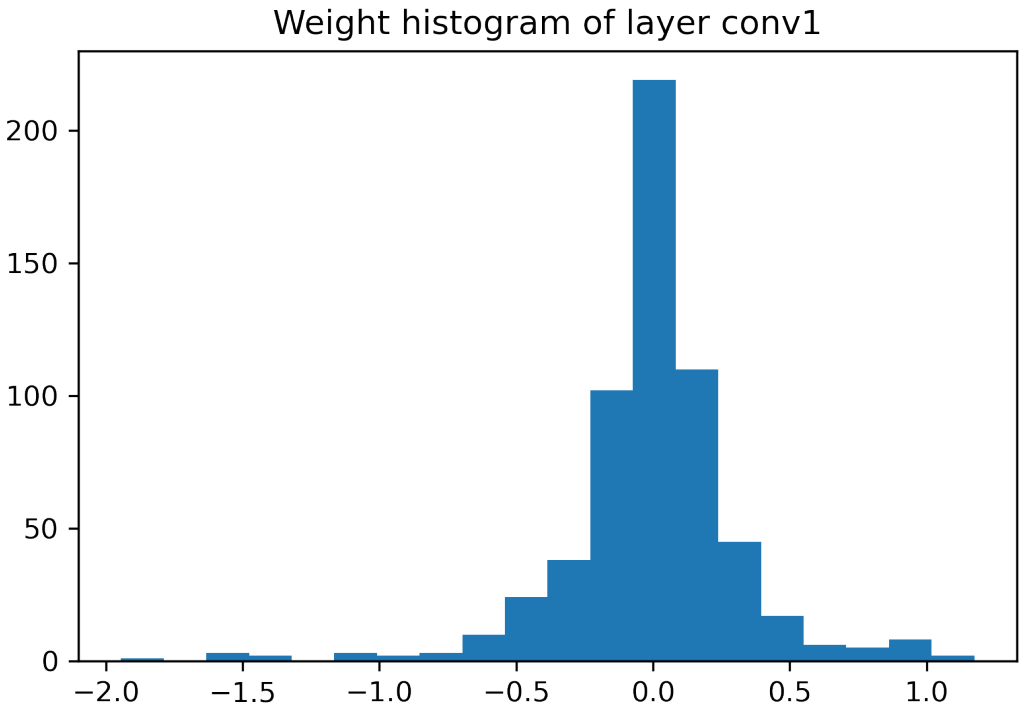
All in all, I found the ResNet parameters to be fairly strong, and the key to training ResNet was a learning rate scheduler that makes a few decays (multiply by 0.1 or 0.2 every 40-60 epochs) in order to be able to reach local minima and slowly but steadily eke out performance gain as the model reaches 89-90 percent and seeks 91-92 percent. Additionally, I found that the model performs better with augmentations, batch normalization, and having a smaller but non-zero L2 regularization in order to prevent the model from overfitting.

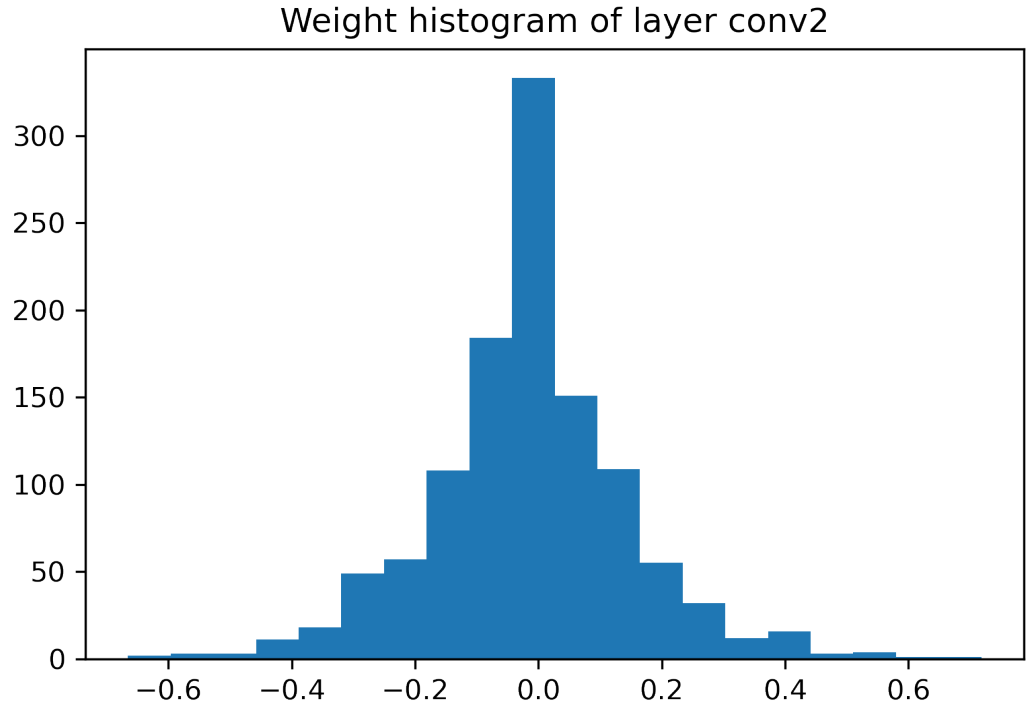
In terms of training the ResNet, there are several hyperparameters that have various affects on model performance. My hypotheses before running the model were that the biggest impact would be made by tuning learning rate and learning rate decay. In running several models with different parameters, I found this to be true. I tried running models with learning rate = 0.1 and different decay schedules (0.2 every 40, 0.1 every 40, 0.1 every 50 epochs, 0.1 every 66 epochs). I found that these performed similarly around 91-92 percent (on validation), but the biggest factor was the actual inclusion of weight decay, as that led to much faster convergence and models that were consistently and easily reaching 91 percent on validation and hit 90 percent with one or two learning rate decays. I found that running it for 200 epochs was not necessary, and was able to reach 90 percent performance from models with decay between 50-100 epochs consistently (although slight performance gains could be eked out with a good learning rate scheduler). I found that a learning rate of 0.1 was the best given the learning rate decay, but a smaller learning rate worked better when I did not use decay, as it was able to reach local minima in the later epochs (whereas 0.1 other may overstep). The inclusion of batch normalization and augmentations were critical to a good pipeline, as it led to much faster training and higher similarity between train and validation performance (due to more diverse data from augmentations). Changing L2 regularization to 0 led to a model that was not regularized well, causing a larger difference in training and validation performance, such that the L2 regularization was kept at 1e-4, which seemed to be a good medium between reduced bias (0) and variance (1e-2). Finally, ReLU and Swish seemed to have similar performance for me with the ResNet.

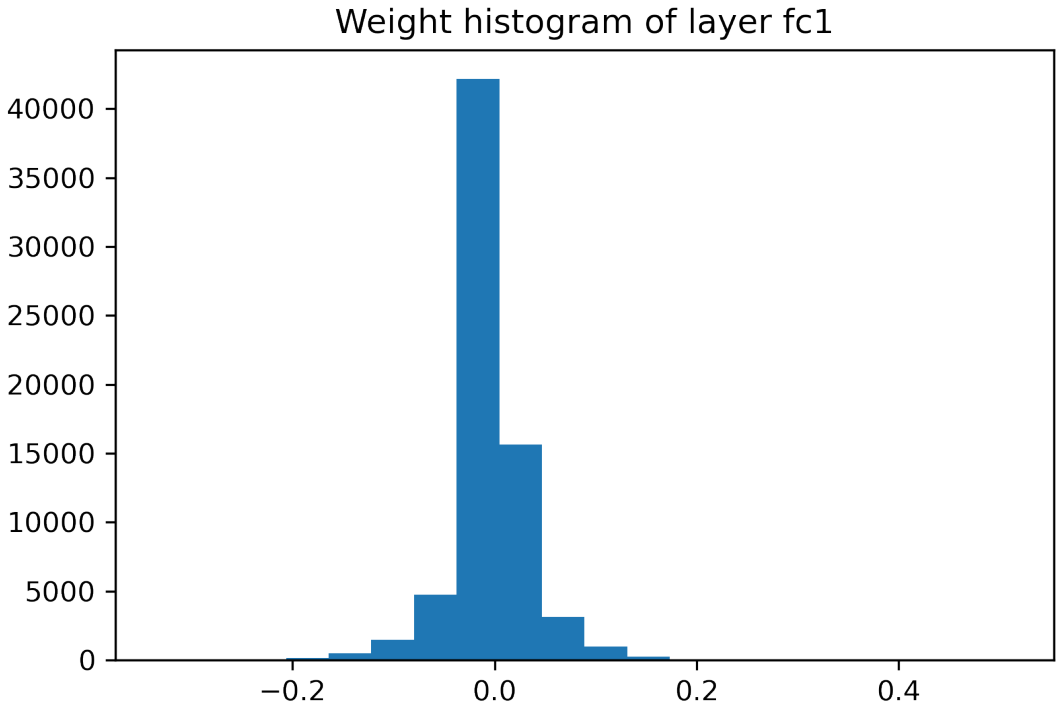
Here, I will discuss the hyperparameters I did not get to tune. I kept the optimizer as SGD in order to stay true to the paper, but the use of the Adam paper may have led to a slight improvement, as the adaptive learning rate is more informed than my guesses for learning rate decay. Additionally, I did not have time to try multiple momentum values, but a smaller momentum may have helped to have the step more in the direction of the gradient (and larger momentum would emphasize previous step in order to leave local minima). I also did not get to test L1 regularization against L2 regularization.

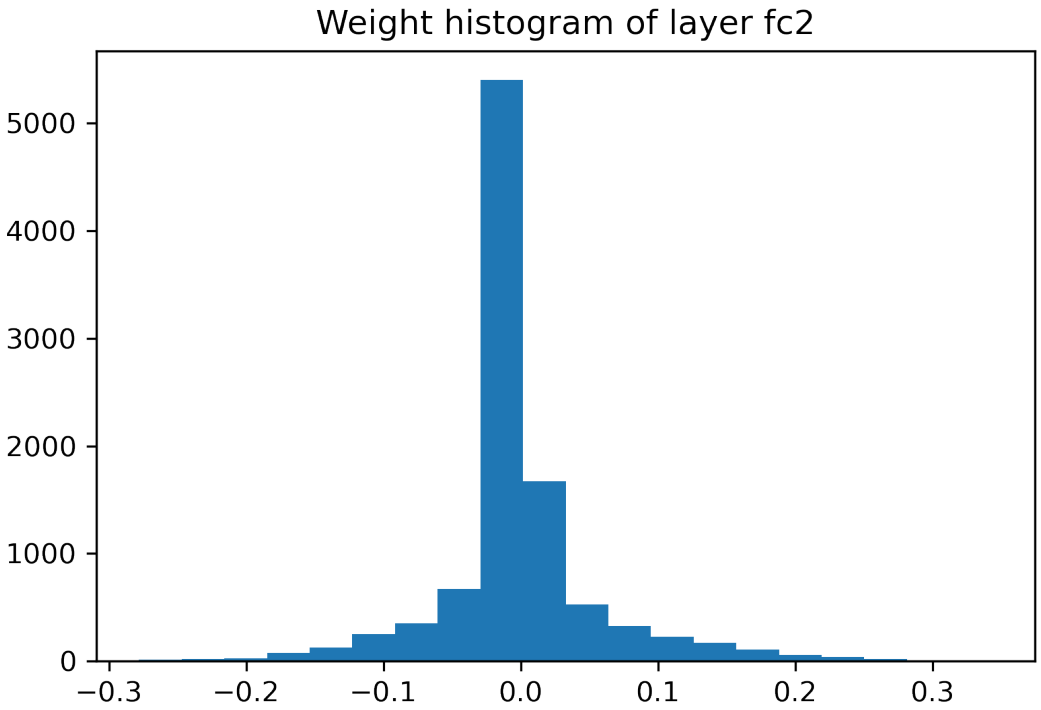
5 Appendix

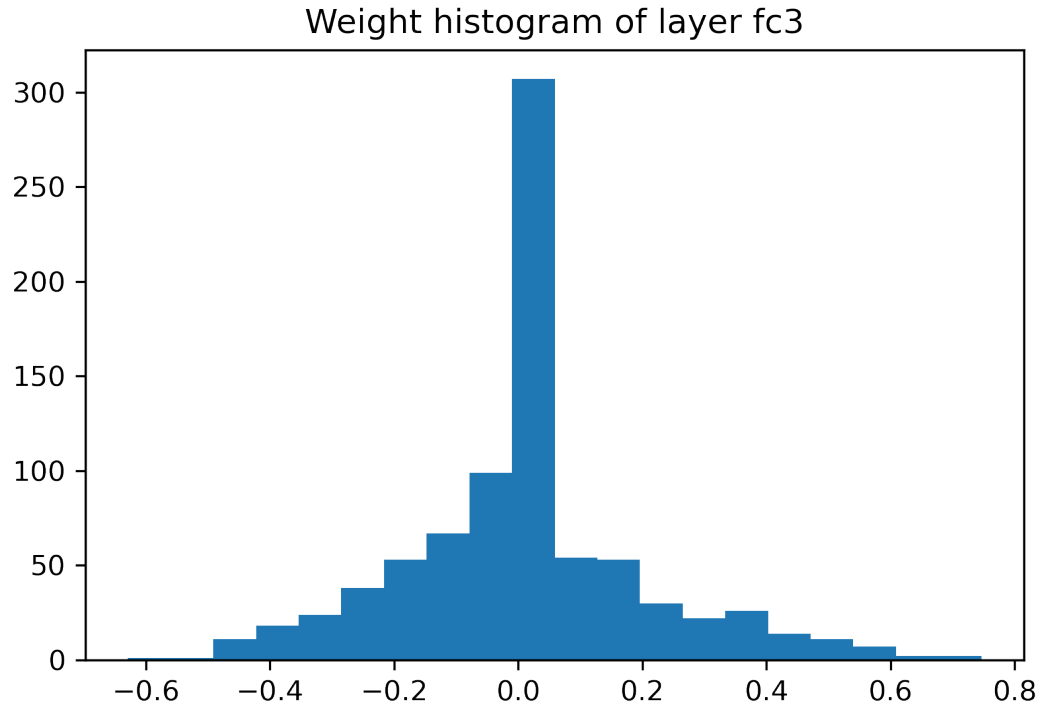
5.1 L1 Regularization Weight Distribution



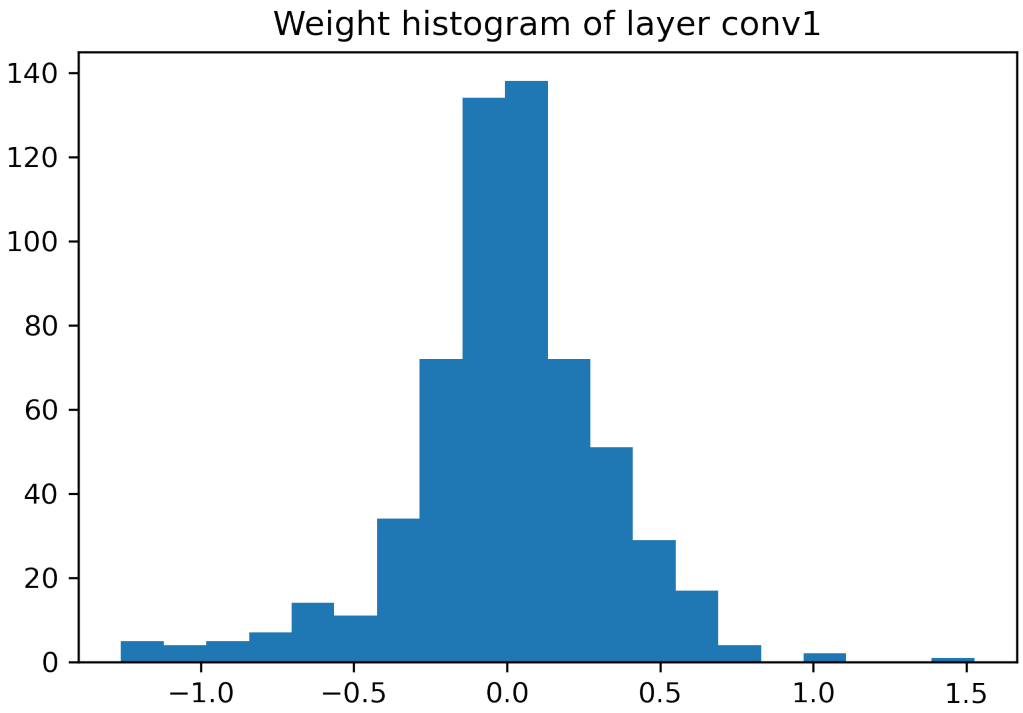

Weight histogram of layer conv2
300
250
200
150
100
50
0
−0.6
−0.4
−0.2
0.0
0.2
0.4
0.6

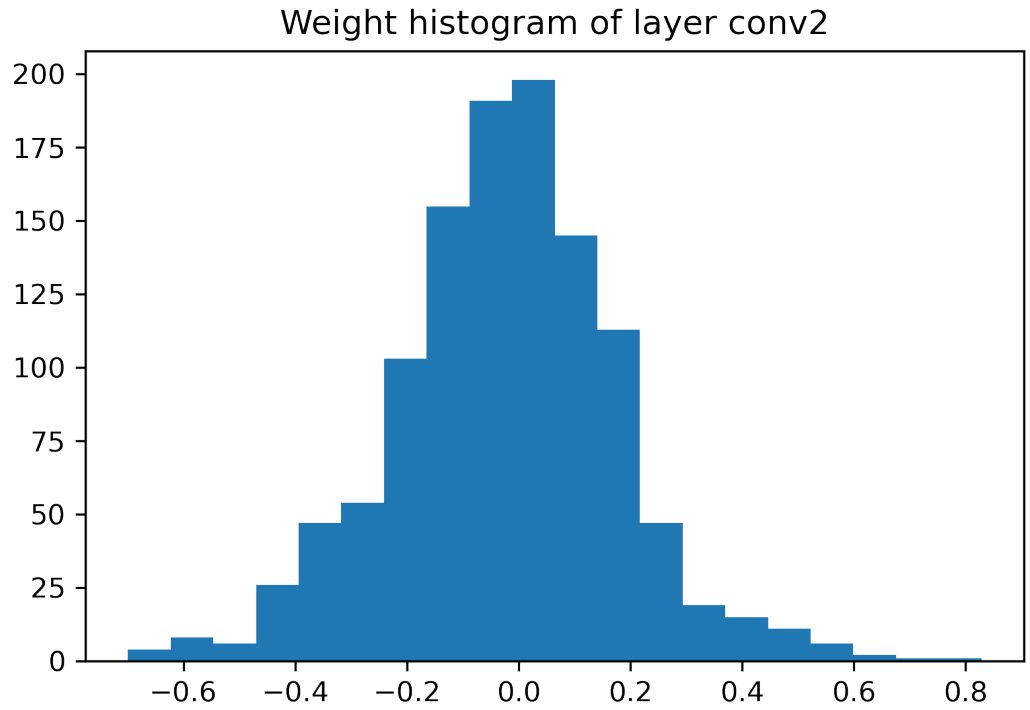

Weight histogram of layer fc1
40000
35000
30000
25000
20000
15000
10000
5000
0
−0.2
0.0
0.2
0.4

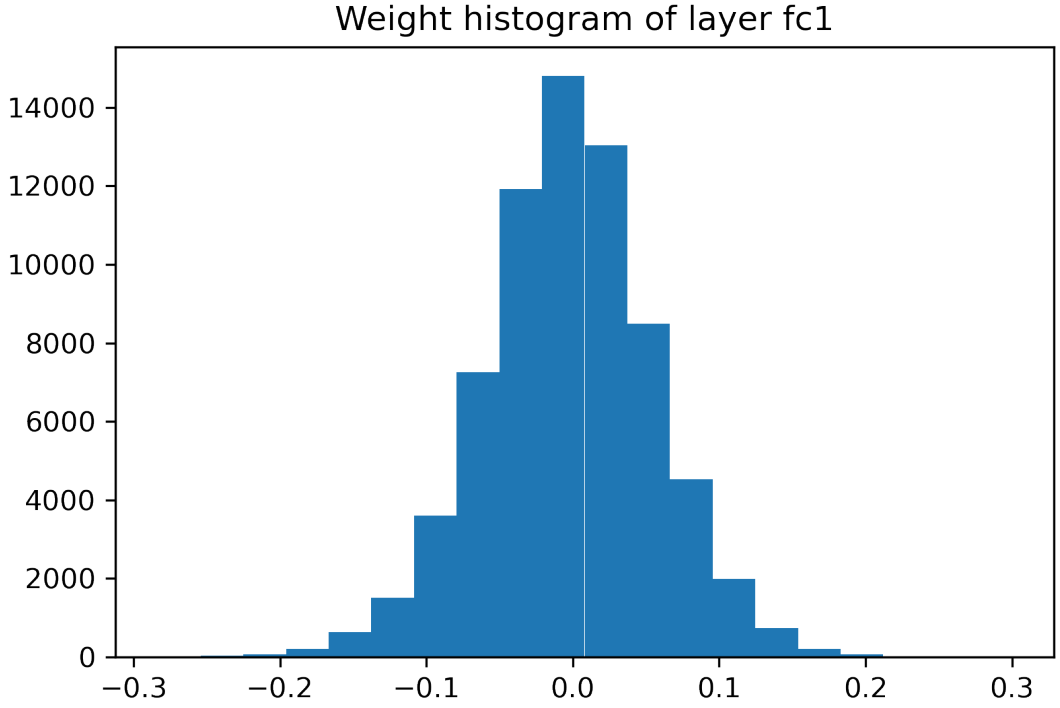

Weight histogram of layer fc2
5000
4000
3000
2000
1000
0
−0.3
−0.2
−0.1
0.0
0.1
0.2
0.3

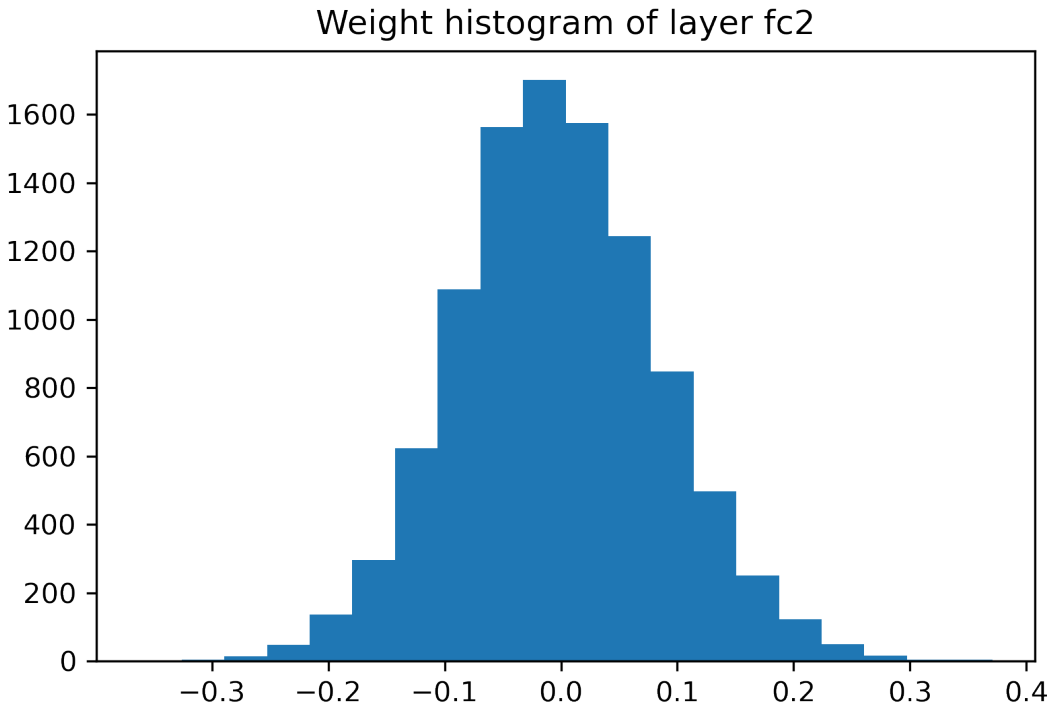

Weight histogram of layer fc3
300
250
200
150
100
50
0
−0.6
−0.4
−0.2
0.0
0.2
0.4
0.6
0.8

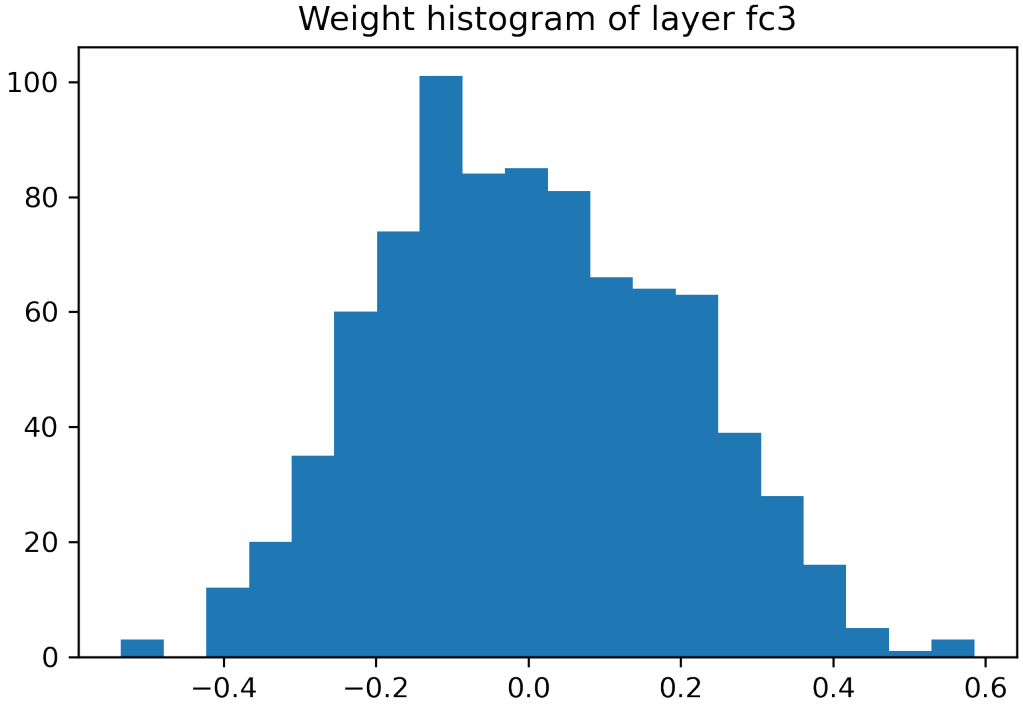
5.2 L2 Regularization Weight Distribution




Weight histogram of layer conv2
200
175
150
125
100
75
50
25
0
−0.6
−0.4
−0.2
0.0
0.2
0.4
0.6
0.8


Weight histogram of layer fc1
14000
12000
10000
8000
6000
4000
2000
0
−0.3
−0.2
−0.1
0.0
0.1
0.2
0.3


Weight histogram of layer fc2
1600
1400
1200
1000
800
600
400
200
0
−0.3
−0.2
−0.1
0.0
0.1
0.2
0.3
0.4


Weight histogram of layer fc3
100
80
60
40
20
0
−0.4
−0.2
0.0
0.2
0.4
0.6