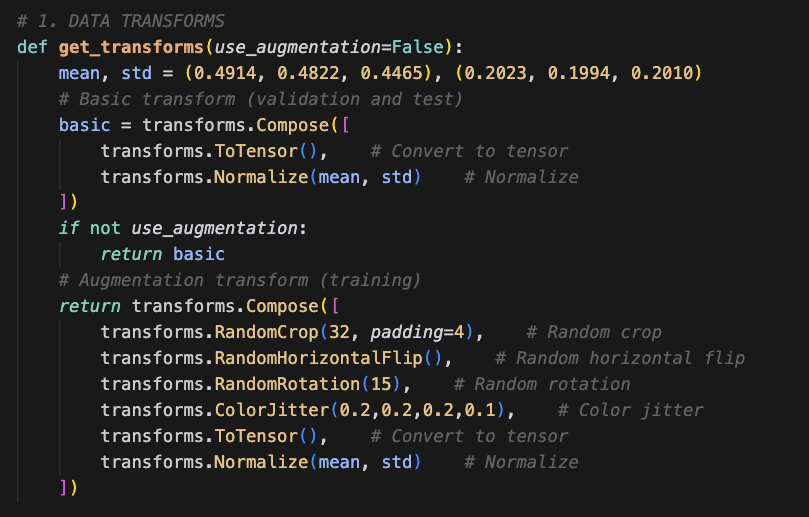
**Introduction to Deep Neural Network – Project #1**

**[Project: CNN for CIFAR-10 Classification]**

2021313549 정성수

**Code Architecture**

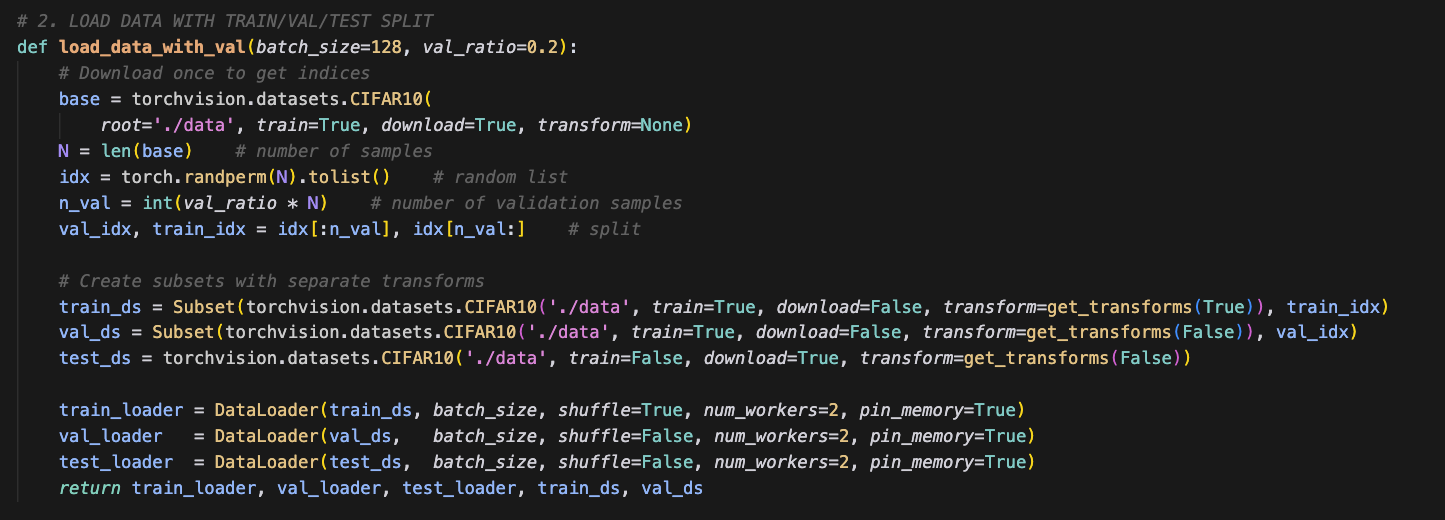
**1. Data Augmentation**



Get mean and standard deviation of CIFAR-10 dataset to normalize the data.

For training, random crop, flip, rotation, and color jitter is applied before normalizing (augmentations).

**2. Dataset load, split**



Split dataset to train, validation, and test set. The reason of the split is to train several models with the same training set, compare the accuracy with the same validation set, and then evaluate the final accuracy of the chosen model with test set after re-training.

This is needed since the selection and evaluation must be independent. (What I learned in fundamentals of machine learning class)

**3. CNN architecture**



The basic idea of architecture is originated from VGGNet, double the number of channels and half the size of image size through convolution blocks. But nothing else is replicated from the existing architectures.

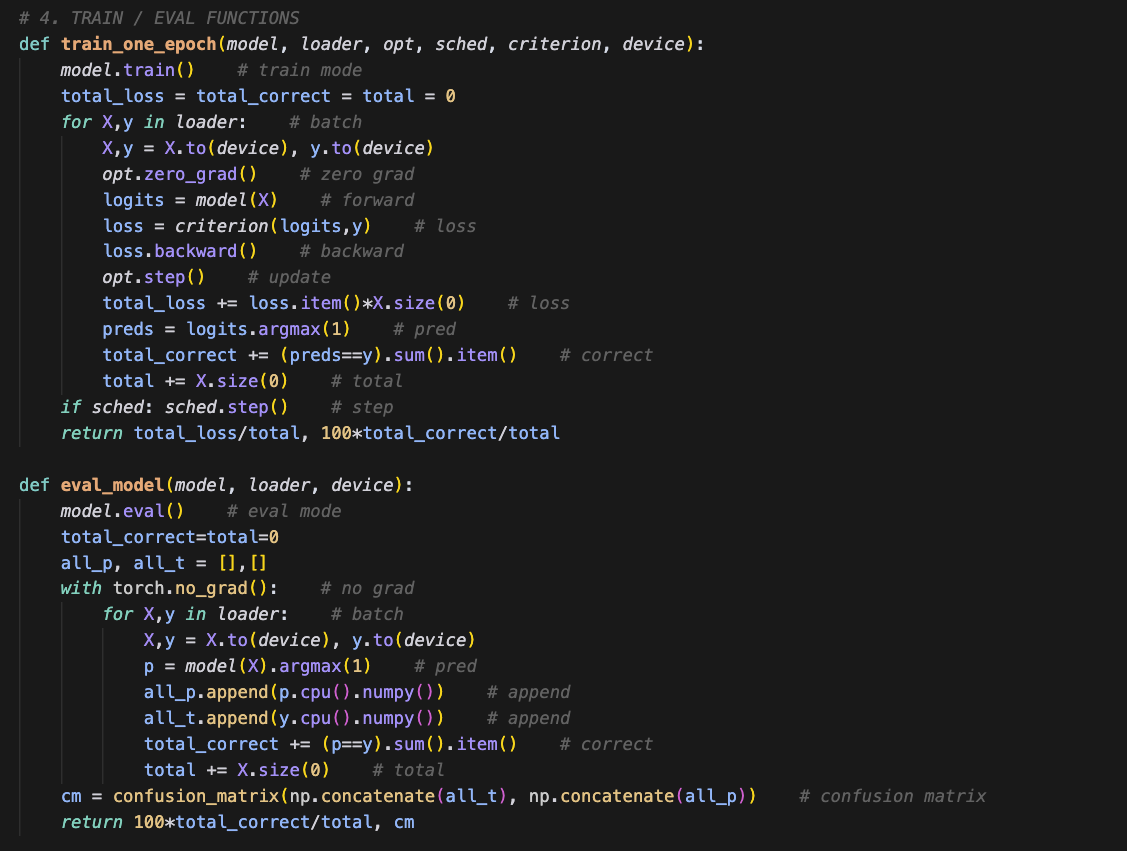
There are three convolutional blocks and each block there are two convolutional layers with a max pooling layer.

Each convolutional layer use 3x3 filter with with 1px padding, stride 1, batch normalization and ReLU activation function.

The output format looks like: (input) 3\*32\*32 => 64\*32\*32 => 64\*32\*32 => (pooling) 64\*16\*16 => 128\*16\*16 => 128\*16\*16 => (pooling) 128\*8\*8 => 256\*8\*8 => 256\*8\*8 => (pooling) 256\*4\*4 => (avg pooling) 256 => (Fully Connected) 10

Also there is dropout rate of 0.3 in FC

**4. Train**



There are two functions. One is used for training an epoch. The other is used for evaluating the given model. Both validation and evaluation use the eval function.

Each function is quite a default form. Explanation below:

==== train ====

model.train() → set dropout, batch normalization to train mode

optimizer.zero\_grad() → initialize previous gradient

logits = model(x) → forward

loss.backward() → backpropagation

opt.step() → parameter update (step)

Accumulate loss and correct: (loss.item()\*batch\_size), (preds==y)

if sched: sched.step() → if using schedular, step learning rate

return loss and accuracy (of the epoch)

==== eval ====

model.eval() + with torch.no\_grad(): → set dropout, batch normalization to test mode

confusion\_matrix(sklearn) → make confusion matrix

return 100\*total\_correct/total, cm → accuracy and confusion matrix

**5. Confusion matrix**

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

Just making confusion matrix.

**6. Main (execution & test)**

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

Setting hyper-parameters, classes, and experimental configs.

Load dataset, set Loss function to cross entropy

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

Train for each config. Accumulate results of each experiment

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

With the result, choose the best model of the best validation accuracy. Train again the chosen model with train + validation set. Then evaluate the re-trained chosen model with the test set.

**Experiment**

The experiment is conducted changing two variables: optimizer and learning rate schedular.

Optimizer : SGD, Adam

Schedular : None, StepLR, CosineAnnealingLR

So **6 different versions** are trained.

**Result & Analysis**

>> Experiment: SGD\_No

Epoch 1/50 – loss 1.6158, acc 40.07%

Epoch 2/50 – loss 1.2458, acc 55.12%

Epoch 3/50 – loss 1.0810, acc 61.51%

Epoch 4/50 – loss 0.9755, acc 65.58%

Epoch 5/50 – loss 0.8921, acc 68.79%

Epoch 6/50 – loss 0.8279, acc 71.09%

Epoch 7/50 – loss 0.7825, acc 72.73%

Epoch 8/50 – loss 0.7437, acc 73.83%

Epoch 9/50 – loss 0.7011, acc 75.69%

Epoch 10/50 – loss 0.6775, acc 76.50%

...

Epoch 41/50 – loss 0.3502, acc 87.88%

Epoch 42/50 – loss 0.3382, acc 88.33%

Epoch 43/50 – loss 0.3363, acc 88.46%

Epoch 44/50 – loss 0.3328, acc 88.47%

Epoch 45/50 – loss 0.3249, acc 88.78%

Epoch 46/50 – loss 0.3252, acc 88.64%

Epoch 47/50 – loss 0.3181, acc 89.06%

Epoch 48/50 – loss 0.3178, acc 88.94%

Epoch 49/50 – loss 0.3135, acc 88.95%

Epoch 50/50 – loss 0.3130, acc 89.23%

Validation Acc: 87.00%

>> Experiment: SGD\_Step

Epoch 1/50 – loss 1.6148, acc 39.84%

Epoch 2/50 – loss 1.2464, acc 55.18%

Epoch 3/50 – loss 1.1080, acc 60.32%

Epoch 4/50 – loss 1.0099, acc 64.45%

Epoch 5/50 – loss 0.9203, acc 67.51%

Epoch 6/50 – loss 0.8615, acc 69.95%

Epoch 7/50 – loss 0.8064, acc 72.00%

Epoch 8/50 – loss 0.7585, acc 73.69%

Epoch 9/50 – loss 0.7285, acc 74.86%

Epoch 10/50 – loss 0.6933, acc 75.89%

...

Epoch 41/50 – loss 0.3835, acc 86.58%

Epoch 42/50 – loss 0.3783, acc 87.03%

Epoch 43/50 – loss 0.3840, acc 86.74%

Epoch 44/50 – loss 0.3824, acc 86.75%

Epoch 45/50 – loss 0.3842, acc 86.75%

Epoch 46/50 – loss 0.3776, acc 87.18%

Epoch 47/50 – loss 0.3791, acc 86.92%

Epoch 48/50 – loss 0.3775, acc 86.91%

Epoch 49/50 – loss 0.3786, acc 87.04%

Epoch 50/50 – loss 0.3791, acc 87.00%

Validation Acc: 87.27%

>> Experiment: SGD\_Cosine

Epoch 1/50 – loss 1.6363, acc 39.34%

Epoch 2/50 – loss 1.2461, acc 55.12%

Epoch 3/50 – loss 1.0847, acc 61.40%

Epoch 4/50 – loss 0.9895, acc 65.08%

Epoch 5/50 – loss 0.8962, acc 68.79%

Epoch 6/50 – loss 0.8414, acc 70.41%

Epoch 7/50 – loss 0.7813, acc 72.85%

Epoch 8/50 – loss 0.7466, acc 74.03%

Epoch 9/50 – loss 0.7078, acc 75.58%

Epoch 10/50 – loss 0.6820, acc 76.55%

...

Epoch 41/50 – loss 0.2778, acc 90.54%

Epoch 42/50 – loss 0.2700, acc 90.75%

Epoch 43/50 – loss 0.2715, acc 90.63%

Epoch 44/50 – loss 0.2622, acc 91.10%

Epoch 45/50 – loss 0.2625, acc 91.03%

Epoch 46/50 – loss 0.2570, acc 91.29%

Epoch 47/50 – loss 0.2576, acc 91.11%

Epoch 48/50 – loss 0.2555, acc 91.30%

Epoch 49/50 – loss 0.2536, acc 91.38%

Epoch 50/50 – loss 0.2521, acc 91.35%

Validation Acc: 89.32%

>> Experiment: Adam\_No

Epoch 1/50 – loss 1.5432, acc 43.17%

Epoch 2/50 – loss 1.1667, acc 58.36%

Epoch 3/50 – loss 1.0164, acc 64.08%

Epoch 4/50 – loss 0.9006, acc 68.20%

Epoch 5/50 – loss 0.8214, acc 71.27%

Epoch 6/50 – loss 0.7635, acc 73.36%

Epoch 7/50 – loss 0.7161, acc 75.22%

Epoch 8/50 – loss 0.6776, acc 76.67%

Epoch 9/50 – loss 0.6521, acc 77.59%

Epoch 10/50 – loss 0.6252, acc 78.47%

...

Epoch 41/50 – loss 0.3975, acc 86.39%

Epoch 42/50 – loss 0.3908, acc 86.72%

Epoch 43/50 – loss 0.3903, acc 86.69%

Epoch 44/50 – loss 0.3854, acc 86.92%

Epoch 45/50 – loss 0.3879, acc 86.91%

Epoch 46/50 – loss 0.3825, acc 86.96%

Epoch 47/50 – loss 0.3819, acc 86.96%

Epoch 48/50 – loss 0.3833, acc 86.89%

Epoch 49/50 – loss 0.3774, acc 86.98%

Epoch 50/50 – loss 0.3800, acc 86.91%

Validation Acc: 85.87%

>> Experiment: Adam\_Step

Epoch 1/50 – loss 1.5152, acc 44.16%

Epoch 2/50 – loss 1.1357, acc 59.74%

Epoch 3/50 – loss 0.9998, acc 64.93%

Epoch 4/50 – loss 0.8888, acc 68.76%

Epoch 5/50 – loss 0.8208, acc 71.39%

Epoch 6/50 – loss 0.7633, acc 73.19%

Epoch 7/50 – loss 0.7180, acc 75.31%

Epoch 8/50 – loss 0.6834, acc 76.52%

Epoch 9/50 – loss 0.6520, acc 77.50%

Epoch 10/50 – loss 0.6341, acc 78.15%

...

Epoch 41/50 – loss 0.2519, acc 91.62%

Epoch 42/50 – loss 0.2488, acc 91.37%

Epoch 43/50 – loss 0.2480, acc 91.59%

Epoch 44/50 – loss 0.2433, acc 91.68%

Epoch 45/50 – loss 0.2490, acc 91.49%

Epoch 46/50 – loss 0.2472, acc 91.58%

Epoch 47/50 – loss 0.2424, acc 91.84%

Epoch 48/50 – loss 0.2422, acc 91.86%

Epoch 49/50 – loss 0.2428, acc 91.80%

Epoch 50/50 – loss 0.2421, acc 91.94%

Validation Acc: 89.58%

>> Experiment: Adam\_Cosine

Epoch 1/50 – loss 1.5309, acc 43.34%

Epoch 2/50 – loss 1.1601, acc 58.35%

Epoch 3/50 – loss 1.0166, acc 63.94%

Epoch 4/50 – loss 0.9103, acc 68.36%

Epoch 5/50 – loss 0.8210, acc 71.39%

Epoch 6/50 – loss 0.7650, acc 73.45%

Epoch 7/50 – loss 0.7240, acc 75.04%

Epoch 8/50 – loss 0.6795, acc 76.69%

Epoch 9/50 – loss 0.6492, acc 77.80%

Epoch 10/50 – loss 0.6253, acc 78.59%

...

Epoch 41/50 – loss 0.1896, acc 93.50%

Epoch 42/50 – loss 0.1814, acc 93.89%

Epoch 43/50 – loss 0.1803, acc 93.88%

Epoch 44/50 – loss 0.1707, acc 94.22%

Epoch 45/50 – loss 0.1655, acc 94.39%

Epoch 46/50 – loss 0.1603, acc 94.58%

Epoch 47/50 – loss 0.1596, acc 94.68%

Epoch 48/50 – loss 0.1606, acc 94.55%

Epoch 49/50 – loss 0.1535, acc 94.84%

Epoch 50/50 – loss 0.1541, acc 94.78%

Validation Acc: 90.57%

**>>> Selected on val: Adam\_Cosine (90.57%)**

Epoch 1/50 – loss 1.4317, acc 47.67%

Epoch 2/50 – loss 1.0601, acc 62.41%

Epoch 3/50 – loss 0.8964, acc 68.56%

Epoch 4/50 – loss 0.7958, acc 72.35%

Epoch 5/50 – loss 0.7184, acc 74.99%

Epoch 6/50 – loss 0.6694, acc 76.92%

Epoch 7/50 – loss 0.6356, acc 78.19%

Epoch 8/50 – loss 0.5912, acc 79.67%

Epoch 9/50 – loss 0.5644, acc 80.51%

Epoch 10/50 – loss 0.5383, acc 81.65%

...

Epoch 41/50 – loss 0.1609, acc 94.67%

Epoch 42/50 – loss 0.1551, acc 94.79%

Epoch 43/50 – loss 0.1496, acc 95.00%

Epoch 44/50 – loss 0.1478, acc 95.08%

Epoch 45/50 – loss 0.1388, acc 95.47%

Epoch 46/50 – loss 0.1346, acc 95.52%

Epoch 47/50 – loss 0.1326, acc 95.54%

Epoch 48/50 – loss 0.1321, acc 95.63%

Epoch 49/50 – loss 0.1310, acc 95.75%

Epoch 50/50 – loss 0.1306, acc 95.68%

**Final Test Acc: 91.01%**

텍스트, 스크린샷, 도표, 사각형이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

**============ANALYSIS============**

First, the selected model is the Adam optimizer with a cosine annealing scheduler. Then the final re-trained test accuracy is 91.01%.

Confusion matrix analysis:

Cat ↔ dog : 91 + 80 is wrongly classified, confusing each other. -> That’s quite obvious since they share many properties like four legs, being the same mammal, similar size, similar face... Even for humans, they look very similar.

Plane ↔ ship / plane ↔ bird / car ↔ truck : 20~30 samples are wrongly classified and confused with the other one.

-> Planes and ships are both transportation, and their environments, the sky and water, are also blue, so confusion can occur. Plane and bird both fly, and car and truck are actually hypernym and hyponym; for sure they share properties.

The final accuracy 91.01% is not an excellent score since the

**Discussion**

**Faced challenges and possible improvements:**

Maybe other data augmentations such as MixUp, CutMix, and AutoAugment, which are often used for CIFAR-10 classifiers.

Using residual terms like ResNet may improve the classification performance.

Using only dropout and batch normalization cannot prevent overfitting sufficiently. Class-specified weight decay or label smoothing can also be applied.

Early stopping/monitoring can be applied so that if training accuracy converges, then stop training to prevent overfitting.

Hyperparameter tuning can be more optimized. Actually, the code written is quite heavy, so sufficient trial and error couldn’t be made. Hyperparameter setting can be more optimized by pinpoint tuning with trial-and-error experiments.

**Insights gained:**

This was the first time to implement the CNN architecture from scratch. Had an experience of hyper-parameter tuning to make improvements to an already constructed NN, but implementing everything such as data augmentation, data loading, NN architecture design, model selecting and evaluating, and plotting a confusion matrix. Making what I’ve learned for myself was quite satisfying.

These are my insights gained during the projects:

1. Data augmentation (preprocessing) is very important.

- Actually, I have experimented with quite a lot of versions. Even ones without data augmentation (before the adjustment of the project guide), the effect of data augmentation was quite huge. The absence of data augmentation led to vulnerability toward new data.

2. Selecting an appropriate scheduler is very important.

- Of course the results show that schedular is very important; without it the accuracy was very low for both of the optimizers. But it’s very important to choose a good scheduler that is proper for the task and optimizer. The final code only does 6 experiments, but my other version of code compared nearly 12 different models, and their accuracy varied greatly by what scheduler was used with a specific optimizer. But since the code took so long to run and it consumed my computing units of Colab so fast, I chose to select which ones to include in the final code.

3. There can be so many variations even the task is very simple.

- During implementing the full-code of CNN, there were so many methods and techniques that can potentially improve the accuracy. But they were so complicated and if the code goes so messy with all those methodologies, I couldn’t catch up my own code if the problem occurs. So, I choosed ones what I have learned during the class, dropout, batch normalization, VGGNet, and ect...

**When Running the code (README)**

There are no requirements or packages to run the code in Google Colab or Jyper Notebook.

But the running time is very long (97 minutes with A100 GPU in Colab)

There was no limitation of running time in the guideline, so I didn’t reduce batch size or number of epochs. But if needed, you can easily reduce the batch size and epoch number in the main function.