ECE 661: Homework #2

Construct, Train, and Optimize CNN Models

John Coogan

2024-09-28

Abstract

In this HW .

Table of contents

0.1 1. True/False Questions (15 pts)

Problem 1.1 (3pts) Any type of data augmentation techniques are always beneficial for deep learning applications.

False, data augmentation techniques are not alwaays beneficial. For instance, flipping elements of the mnist dataset can turn a 9 into a 6 while the data is still labeled as a 9. This is not beneficial for the model because now it has what looks like a 6 labeled as a 9 which will cause training issues.

Problem 1.2 (3pts) If we do not use both batch normalization and dropout, CNN training converges much slower.

False, depending on how dropout is employed, it can increase either the training time or the inference time. Depending on when dropout scaling is applied, one could see reduced

convergence time in training. Batch normalization will normally decrease the convergence time of a CNN though.

Problem 1.3 (3pts) Dropout is a common technique to combat overfitting. If L-normalizations are included alongside dropout, the performance will be even better.

False, dropout is a technique to combat overfitting. It accomplishes this by randomly setting a fraction of the neurons to zero. This forces the remaining neurons to pick up the feature representations that the dropped neurons were responsible for. The remaining neurons contribute to the output of the network in a naive way that prevents overfitting. That said, dropout does not always cooperate well with L-norm regularization.

Problem 1.4 (3pts) During training, the Lasso (L1) regularizer causes the model to have a higher sparsity compared to Ridge (L2) regularizer.

True, L1 regularization yields sparce solutions which should be used for compact models with high compression rate. L2 regularization yields more stable results without too much training effort.

Problem 1.5 (3pts) The shortcut connections in ResNets result in smoother loss surface.

True, the shortcut connections in ResNet results in a smoother loss surface. By applying residual learning and forwarding inputs via bypass further into the architecture, the gradient can be more directly propagated to earlier layers. This helps to prevent the vanishing gradient problem and allows for smoother loss surfaces.

0.2 2 Computation Questions (15 pts)

Problem 2.1 (3pts) Consider a 100x100 RGB Image as an input. If the first hidden layer consists of 100 neurons, and each neuron is fully connected to the input, how many parameters does this hidden layer have including both the weights and the bias parameters?

The number of inputs is the size of the image multiplied by the channels (R,G,B):

$100 * 100 * 3 = 30,000$

Each neuron will have a weight for each input and a bias term. Therefore, the number of parameters for the hidden layer is:

$30,000 * 100 + 100 = 3,000,100$

Problem 2.2 (3pts) Consider the same 100x100 RGB Image as an input. If you use a convolution layer with 100 filters each with size 3x3, how many parameters does this hidden layer have including both the weights and the bias parameters ?

Convolutional Layer Shape Rules:

- Number of filters N
- Convolutional kernel size K
- Stride for convolution S
- Padding for each border P

Output feature map: $C_2 \times H_2 \times W_2$, where

$W_2 = [\frac{W_1 - K + 2P}{S}] + 1$

$H_2 = [\frac{H_1 - K + 2P}{S}] + 1$

$C_2 = N$

Number of filters: $N = 100$

convolutional kernel size: $K = 3$

Stride for convolution: $S = 1$ (Assumption)

Padding for each border: $P = 0$ (Assumption)

$W_2 = [\frac{100 - 3 + 2*0}{1}] + 1 = 98$

$H_2 = [\frac{100 - 3 + 2*0}{1}] + 1 = 98$

$C_2 = 100$

Output feature map: $100 \times 98 \times 98$

Total Weight Elements (per filter): $C_1 \times K \times K = 3 \times 3 \times 3 = 27$

Number of filters $F = 100$

Total weights = $27 \times 100 = 2700$

Total bias = 100

Total parameters = $2700 + 100 = 2800$

Problem 2.3 (3pts) Consider an input volume with dimensions 100x100x16, how many parameters does a single 1x1 convolution filter have including the bias term ?

Number of filters: $N = 1$

convolutional kernel size: $K = 1$

Weights per filter: $C_1 \times K \times K = 16 \times 1 \times 1 = 16$

For a single filter the number of weights is 16 and there is only one bias term per filter so the total parameters is 17.

Problem 2.4 (3pts) Consider the input volume of 100x100x16, and we apply convolution with 32 filters each 5x5 size with a stride of 1 and no padding. What is output shape?

Number of filters: $N = 32$

convolutional kernel size: $K = 5$

Stride for convolution: $S = 1$

Padding for each border: $P = 0$

$W_2 = [\frac{W_1 - K + 2P}{S}] + 1 = [\frac{100 - 5 + 2*0}{1}] + 1 = 96$

$H_2 = [\frac{H_1 - K + 2P}{S}] + 1 = [\frac{100 - 5 + 2*0}{1}] + 1 = 96$

$C_2 = 32$

Output feature map: $32 \times 96 \times 96$

Problem 2.5 (3pts) MobileNets use depthwise separable convolution to improve the model efficiency. If we replace all the 3x3 convolution layers to 3x3 depth wise separable convolution layer in ResNet architectures, what would be the likely speedup for these layers.

Regular Convolution MACS: $3 \times 3 \times M \times N \times D_F \times D_F$

Regular Convolution Parameters: $3 \times 3 \times M \times N$

Depthwise Separable Convolution MACS: $3 \times 3 \times M \times D_F \times D_F + D_F \times D_F \times M \times N$

Depthwise Separable Convolution Parameters: $3 \times 3 \times M + M \times N$

When N,M is large, the speedup is approximately

$\frac{3 \times 3 \times M \times N \times D_F \times D_F}{3 \times 3 \times M \times D_F \times D_F + D_F \times D_F \times M \times N} \approx 9$

0.3 Lab 1

Question (a) Each image in CIFAR is an RGB image with 32x32 pixels. This means that the input size is 3x32x32. We know from the past assignment that the output size is 10. This means that we can pass a random image and check the output dimensions.

Question (b) The two preprocessing steps that are crucial is 1) to convert the image into a tensor and 2) to normalize the image. The image is converted into a tensor so that it can be passed into the model. The image is normalized so that the model can learn the weights and biases more effectively.

Question (c/d) DataLoader:

nvidia-smi shows that a new process has been added to the GPU but the usage is so low that it does not register. I have hadded lines to my training and eval code that print the amount of memory allocated (<20MB out of 4GB).

Question (e/f) See ipynb file.

Question (g) The initial loss value before we conducted any training is 1.9050 (training) and 1.5877 (validation). We would expect the cross entropy loss for an untrained model (random guessing) to be around

```python
# do NOT change these
from tools.dataset import CIFAR10
from torch.utils.data import DataLoader

# a few arguments, do NOT change these
DATA_ROOT = "./data"
TRAIN_BATCH_SIZE = 128
VAL_BATCH_SIZE = 100

#############################################
# your code here
# construct dataset
train_set = CIFAR10(
    root=DATA_ROOT, mode="train", download=True, transform=transform_train
)
val_set = CIFAR10(root=DATA_ROOT, mode="val", download=True, transform=transform_val)

# construct dataloader
train_loader = DataLoader(
    train_set,
    batch_size=TRAIN_BATCH_SIZE,
    shuffle=True,  # your code
    num_workers=4,
)
val_loader = DataLoader(
    val_set, batch_size=VAL_BATCH_SIZE, shuffle=False, num_workers=4  # your code  # your code
)
#############################################
```

Figure 1: image-2.png

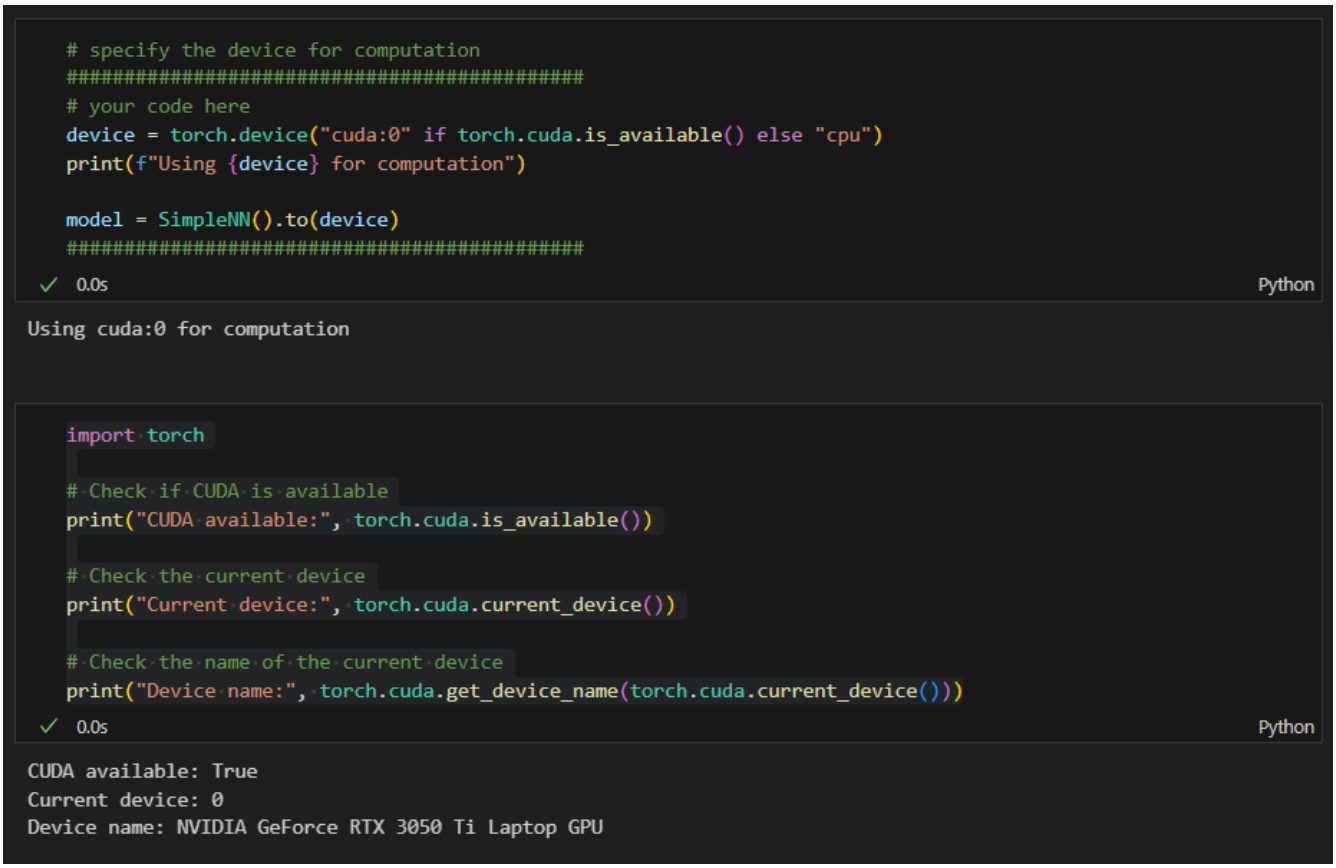


Figure 2: image.png

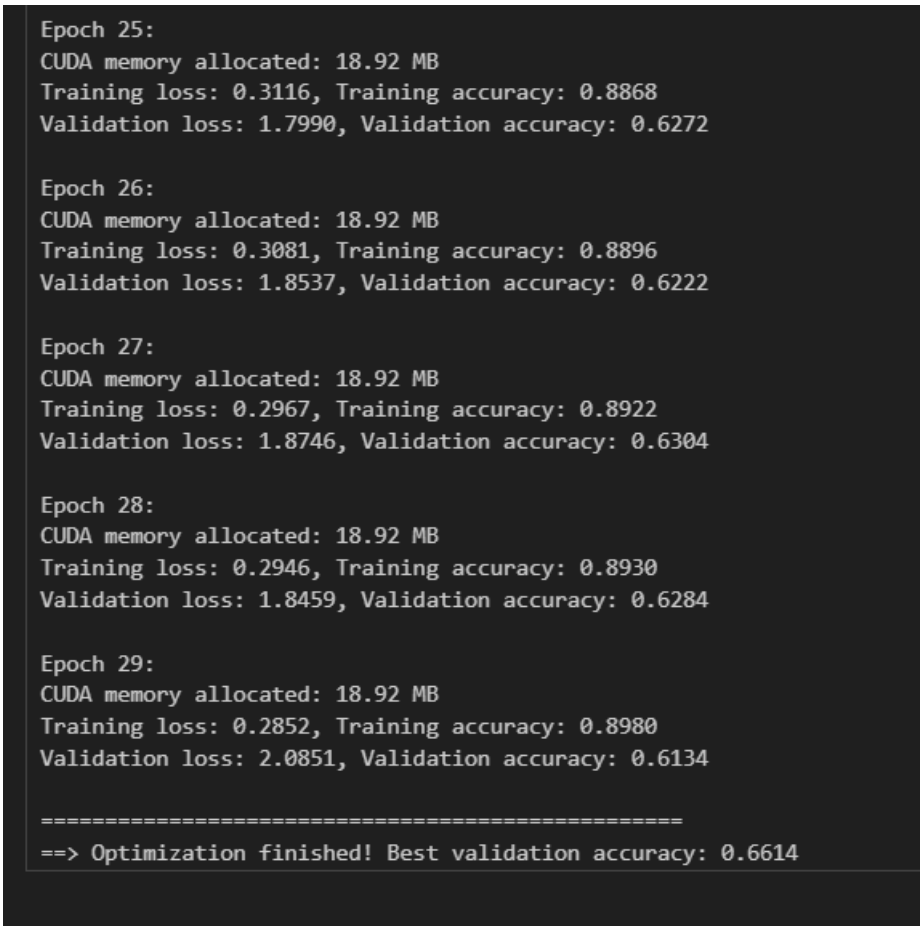
```python
import torch.nn as nn
import torch.optim as optim

# hyperparameters, do NOT change right now
# initial learning rate
INITIAL_LR = 0.01

# momentum for optimizer
MOMENTUM = 0.9

# L2 regularization strength
REG = 1e-4

##########################################
# your code here
# create loss function
criterion = nn.CrossEntropyLoss()

# Add optimizer
optimizer = optim.SGD(model.parameters(), lr=INITIAL_LR, momentum=MOMENTUM, weight_decay=REG)
##########################################
```

✓ 0.0s Python

Figure 3: image.png

Initial Loss $\approx -\log\left(\frac{1}{10}\right) = \log(10) \approx 2.3026$.

Model training output (final few epochs):



We can see that after epoch 9, we begin to overfit the data. We get our highesst validation accuracy (0.6602) while our training accuracy continues to rise, the validation accuracy starts to decrease.

```
Epoch 9:
CUDA memory allocated: 18.92 MB
Training loss: 0.7542, Training accuracy: 0.7327
Validation loss: 1.0349, Validation accuracy: 0.6602
Saving ...

Epoch 10:
CUDA memory allocated: 18.92 MB
Training loss: 0.7158, Training accuracy: 0.7461
Validation loss: 1.0727, Validation accuracy: 0.6540

Epoch 11:
CUDA memory allocated: 18.92 MB
Training loss: 0.6793, Training accuracy: 0.7594
Validation loss: 1.0825, Validation accuracy: 0.6474

Epoch 12:
CUDA memory allocated: 18.92 MB
Training loss: 0.6371, Training accuracy: 0.7749
Validation loss: 1.0794, Validation accuracy: 0.6488

Epoch 13:
CUDA memory allocated: 18.92 MB
Training loss: 0.5995, Training accuracy: 0.7860
Validation loss: 1.1427, Validation accuracy: 0.6476
```

Our final training accuracy is 0.9016 but is only this high because we have overfit to the training data.

0.4 Lab 2

Question (a) By adding a random crop with padding as well as a horizontal flip

we can improve our final validation accuracy from 0.6614 to 0.7206. This is a sizable improvement.

Question (b) We now add batch normalization to our model.

We actually see our validation accuracy decrease from our non-batch normalized model:

But we have not adjusted the batch size or the learning rate so it is still possible that we can improve our model with batch normalization.

Running our batch normalized model with a larger learning rate decreases our validation accuracy. For instance, running at a 0.1 learning rate gives us a final validation accuracy of 0.6804.

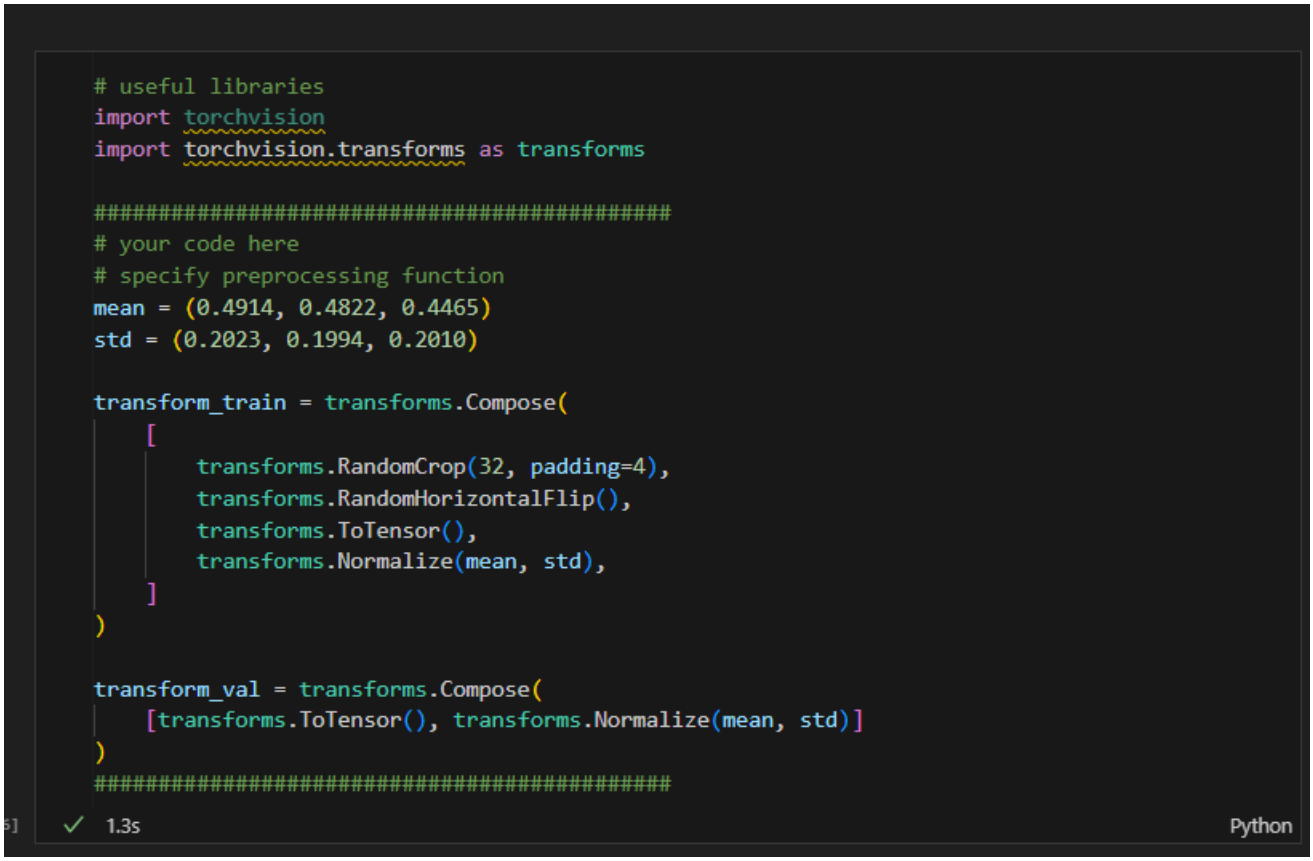


Figure 4: image-2.png

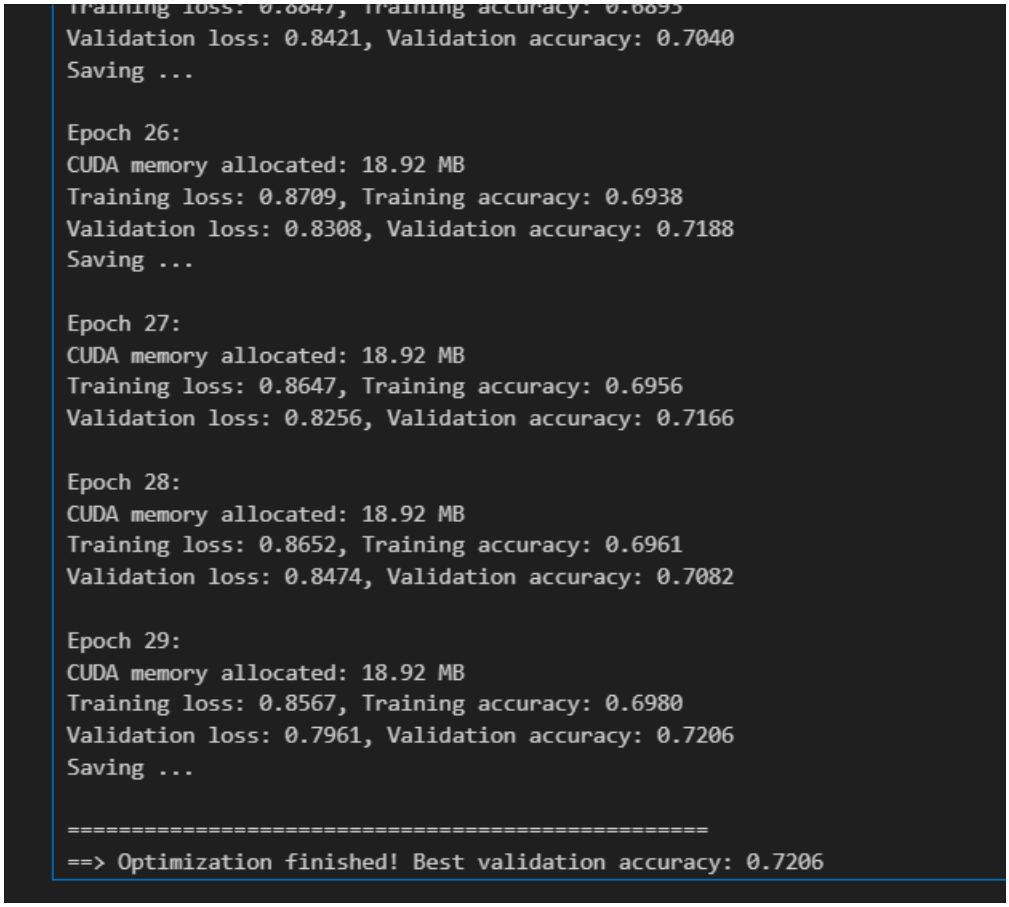
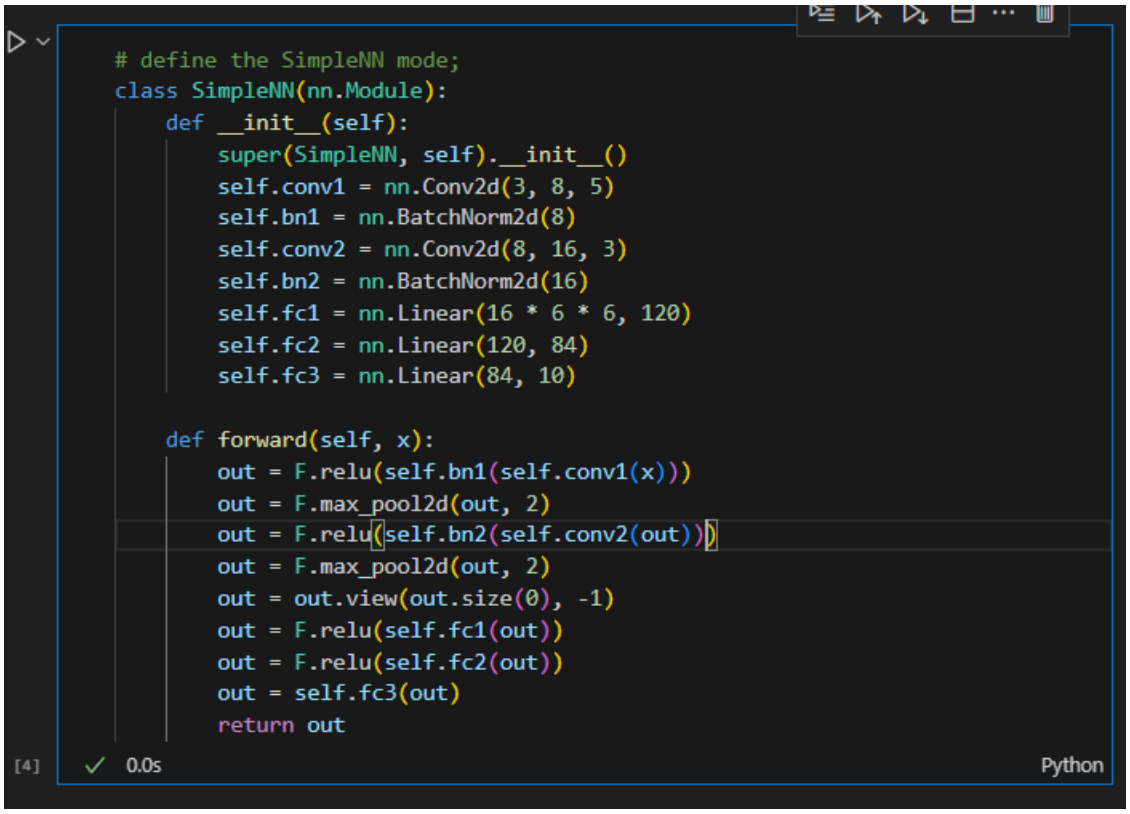


Figure 5: image.png



```python
# define the SimpleNN mode;
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 8, 5)
        self.bn1 = nn.BatchNorm2d(8)
        self.conv2 = nn.Conv2d(8, 16, 3)
        self.bn2 = nn.BatchNorm2d(16)
        self.fc1 = nn.Linear(16 * 6 * 6, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = F.max_pool2d(out, 2)
        out = F.relu(self.bn2(self.conv2(out)))
        out = F.max_pool2d(out, 2)
        out = out.view(out.size(0), -1)
        out = F.relu(self.fc1(out))
        out = F.relu(self.fc2(out))
        out = self.fc3(out)
        return out
```

[4] ✓ 0.0s Python

Figure 6: image.png

```
CUDA memory allocated: 18.93 MB
Training loss: 0.8594, Training accuracy: 0.6968
Validation loss: 0.8305, Validation accuracy: 0.7142

Epoch 27:
CUDA memory allocated: 18.93 MB
Training loss: 0.8511, Training accuracy: 0.6985
Validation loss: 0.8360, Validation accuracy: 0.7138

Epoch 28:
CUDA memory allocated: 18.93 MB
Training loss: 0.8449, Training accuracy: 0.7017
Validation loss: 0.8504, Validation accuracy: 0.6976

Epoch 29:
CUDA memory allocated: 18.93 MB
Training loss: 0.8424, Training accuracy: 0.7037
Validation loss: 0.8398, Validation accuracy: 0.7046

==================================================
==> Optimization finished! Best validation accuracy: 0.7148
```

Figure 7: image.png

Increasing the learning rate by just 0.01 to 0.02

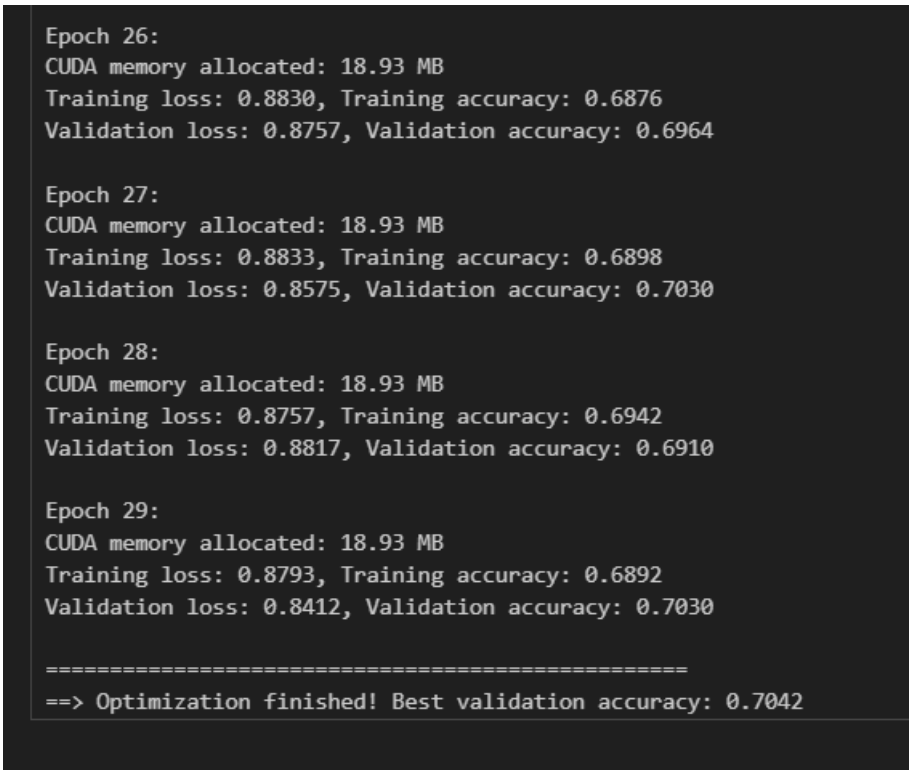


Figure 8: image.png

gives us a lower final validation accuracy at 0.7042

With respect to the SWISH function,

$$\text{SWISH}(x) = x \cdot \sigma(\beta x) = \frac{x}{1 + e^{-\beta x}}$$

or, for when $\beta = 1$, the funciton is known as the Sigmoid Weighted Linear Unit (SiLU) function. Since no β value is given, we will assume that $\beta = 1$.

Figure 9: image-2.png

Our performance with SiLU is better than ReLU: