Homework 1: Convolutional Neural Networks
CSC 266 / 466
Frontiers in Deep Learning
Spring 2023

Junfei Liu - jliu137@u.rochester.edu

Deadline: See Blackboard

Instructions

Your homework solution must be typed and prepared in LaTeX. It must be output to PDF format. To use LaTeX, we suggest using `http://overleaf.com`, which is free and can be accessed online.

Your submission must cite any references used (including articles, books, code, websites, and personal communications). All solutions must be written in your own words, and you must program the algorithms yourself. **If you do work with others, you must list the people you worked with.** Submit your solutions as a PDF to Blackboard.

Your programs must be written in Python. The relevant code should be in the PDF you turn in. If a problem involves programming, then the code should be shown as part of the solution. One easy way to do this in LaTeX is to use the verbatim environment, i.e., \begin{verbatim} YOUR CODE \end{verbatim}.

You may wish to use the program *MathType*, which can easily export equations to AMS LaTeX so that you don't have to write the equations in LaTeX directly: `http://www.dessci.com/en/products/mathtype/`

Problem 1 - BatchNorm

BatchNorm is a common technique that can accelerate and improve training, especially with deeper networks. It is often used with convolutional networks, but it requires a sufficiently large batch size for it to work effectively. Note that it is not used for some tasks, e.g., training transformers or for natural language processing tasks, as the statistics tend to fluctuate greatly across instances leading to BatchNorm causing instabilities. While it is often used, BatchNorm is hard to parallelize across GPUs or multiple machines, and it is a poor choice for problems where there is domain shift (e.g., continual learning).

Part 1 - Investigations (10 points)

In this problem, you will study BatchNorm's properties for a single 'dot product' neuron, but the results are the same if you have multiple neurons in a layer or use convolutional units.

Let

$$x_i = \mathbf{w}^T \mathbf{h}_i + b$$

be the output of neuron i, where $\mathbf{h}_i \in \mathbb{R}^d$ is a vector of inputs to the neuron, $\mathbf{w} \in \mathbb{R}^d$ are the weights of the neuron, and b is the bias. BatchNorm is applied in the subsequent 'layer' before the non-linearity. After computing the activation of the neuron, the BatchNorm transformation of the output activations for a mini-batch of size m is given by

$$y_i = \beta_i + \gamma_i \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}},$$

where the mean of the activation in the mini-batch is given by

$$\mu_B = \frac{1}{m} \sum_{j=1}^{m} x_j,$$

the variance is given by

$$\sigma_B^2 = \frac{1}{m} \sum_{j=1}^{m} (x_j - \mu_B)^2,$$

and where γ_i and β_i are learned parameters.

Substitute in the neuron's activation function into the BatchNorm equations and simplify (show this). What does this tell you about how BatchNorm impacts the weights $\mathbf{h}_i$ and the bias b of neuron? How should you adjust your neural network's architecture if using BatchNorm?

Solution:
First, substitute and simplify μ_B.

$$\mu_B = \frac{1}{m}\sum_{j=1}^{m} \mathbf{w}^T\mathbf{h}_j + b,$$

$$\mu_B = b + \frac{1}{m}\sum_{j=1}^{m} \mathbf{w}^T\mathbf{h}_j,$$

Then, substitute and simplify σ_B^2.

$$\sigma_B^2 = \frac{1}{m}\sum_{j=1}^{m} \left(\mathbf{w}^T\mathbf{h}_j + b - \mu_B\right)^2,$$

$$\sigma_B^2 = \frac{1}{m}\sum_{j=1}^{m} \left(\mathbf{w}^T\mathbf{h}_j + b - (b + \frac{1}{m}\sum_{j=1}^{m} \mathbf{w}^T\mathbf{h}_j)\right)^2,$$

$$\sigma_B^2 = \frac{1}{m}\sum_{j=1}^{m} \left(\mathbf{w}^T\mathbf{h}_j - \frac{1}{m}\sum_{j=1}^{m} \mathbf{w}^T\mathbf{h}_j\right)^2,$$

Finally, substitute and simplify y_i using μ_B and σ_B^2.

$$y_i = \beta_i + \gamma_i \frac{\mathbf{w}^T\mathbf{h}_i + b - (b + \frac{1}{m}\sum_{j=1}^{m} \mathbf{w}^T\mathbf{h}_j)}{\sqrt{\sigma_B^2 + \varepsilon}},$$

$$y_i = \beta_i + \gamma_i \frac{\mathbf{w}^T\mathbf{h}_i - \frac{1}{m}\sum_{j=1}^{m} \mathbf{w}^T\mathbf{h}_j}{\sqrt{\frac{1}{m}\sum_{j=1}^{m} \left(\mathbf{w}^T\mathbf{h}_j - \frac{1}{m}\sum_{j=1}^{m} \mathbf{w}^T\mathbf{h}_j\right)^2 + \varepsilon}},$$

Part 2 - Discussion (5 points)

Discuss the pros and cons of using BatchNorm. When does BatchNorm fail? Discuss alternatives and their pros and cons. What properties would an ideal normalization method have?

Solution:
BatchNorm has the following limitations:

1. The batch size has to be large. The batch normalization is based on the statistics inside the mini-batch, which makes the result unrepresentative of the dataset if mean and standard deviations are calculated from a small sample. As a result, the iteration will not learn effectively if the batch size is not large enough.
2. It is not a good choice for some models, such as sequence models and continual learning models. In these cases, input has small batch sizes with potentially different length, which makes batchnorm tend to not perform well.
However, it is widely used because of the following reasons:
1. It is stable with large batch sizes.
2. It reduces the internal covariate shift, the amplification effect of small changes in parameters.
3. BatchNorm also has a regularizing effect which can help reduce overfitting.
4. It allows for higher learning rate.

Alternatives of BatchNorm include LayerNorm, GroupNorm, InstanceNorm, etc.
They generally perform better on small batches than BatchNorm and are designed for specific neural network architectures. For example, LayerNorm outperforms other normalization techniques in RNN.

An ideal normalization method is able to:
1. maintains the contribution of every feature
2. reduces Internal Covariate Shift
3. makes the Optimization faster

Problem 2 - Using Pre-Trained CNNs and Swin Transformers

For this problem you must use PyTorch. For this problem you will use both a CNN and a Vision Transformer known as a Swin transformer that has been trained on ImageNet-1k. We will compare the CNN ResNet-50 and the Vision Transformer Swin-T (tiny), which have 26 million and 28 million parameters, respectively, each. When loading the weights use the default weights, i.e., `ResNet50_Weights.DEFAULT` and `Swin_T_Weights.DEFAULT`. Make sure to pre-process the input image appropriately. Look at TorchVision's documentation for the pre-trained models to determine how to do this.

Part 1 - Using Pre-Trained Deep CNN (4 points)

Run ResNet50 on `fruit.jpg`. Output the top-3 predicted categories and the probabilities, e.g., "Strawberry: 33.5%", etc.

Solution:
jackfruit: 56.8%
custard apple: 1.7%
pineapple: 1.4%

Part 2 - Using Pre-Trained Deep CNN (4 points)

Run Swin-T on `fruit.jpg`. Output the top-3 predicted categories and the probabilities.

Solution:
jackfruit: 60.6%
custard apple: 9.0%
strawberry: 4.9%

Part 3 - Discussion (2 points)

The two neural networks produced different outputs, but both produced probabilities that seem imbalanced as far as recognizing the objects present in the images. Discuss why this happens in terms of the loss function used to train them. What might be a better approach for dealing with problems with detecting the presence of multiple objects in the same image?

Solution:
The cross-entropy loss function will result in imbalanced probabilities. Other loss functions such as focal loss might be a better approach.

Problem 3 - Transfer Learning (20 points)

For this problem you must use PyTorch. We will do image classification using the Oxford Pet Dataset. The dataset consists of 37 categories with about 200 images in each of them. You can learn more about the dataset here: `http://www.robots.ox.ac.uk/~vgg/data/pets/`

You should use the version of the dataset built into PyTorch. You can find the documentation here: `https://pytorch.org/vision/stable/generated/torchvision.datasets.OxfordIIITPet.html`

We will fine-tune ResNet-50's *and* Swin-T's ImageNet weights for the Pets dataset. To do this, we will replace the output layer of the network with one for the number of categories in the Pets dataset. Ensure your toolbox is appropriate transforming the images, e.g., resizing

them to a size compatible with your network, subtract the mean pixel value, divide by the standard deviation, etc.

We will train each architecture twice: 1) Once with just the output layer of the network being trained, and 2) all units fine-tuned for the task.

For data augmentation, use random flips and crops. Use PyTorch or PyTorch Lightning to do this problem, but it can be made significantly simpler by using PyTorch Lightning. You should write one piece of code where you select the neural network to fine-tune, rather than producing two pieces of code for each model. Use AdamW as your optimizer. Note that the optimal hyperparameters may differ for the optimizer between the two.

- Plot the train and test loss curves for both ResNet-50 and Swin-T as a function of epochs on the same plot. Make sure you label the lines appropriately. We recommend you do this by using Weights and Biases or with TensorBoard.
- What observations can you make from the plots regarding fine-tuning just the output layer compared to the entire network?
- Make a table to report the best accuracy of each approach.
- Analyze the errors that each model makes, compare them, and discuss. Does one architecture perform significantly better at some categories than others?

Solution:

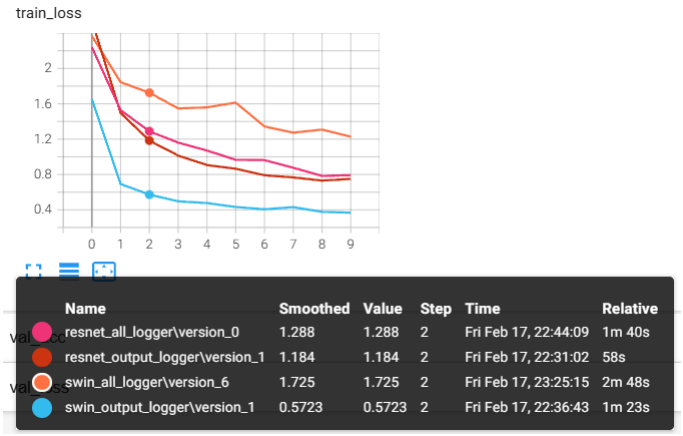

Figure 1: Train loss

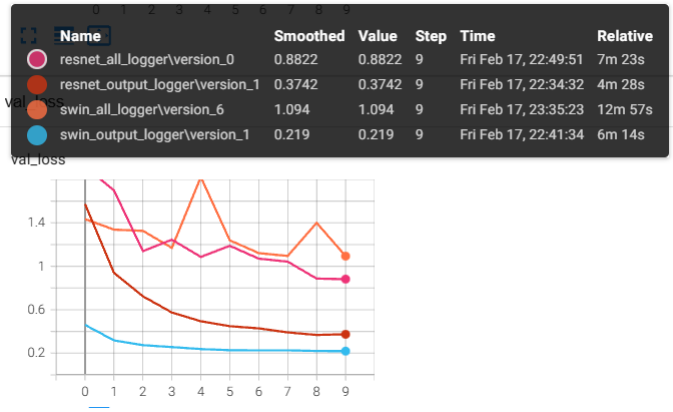

Figure 2: Test loss

I find that fine-tuning just the output layer will result in a lower loss for both resnet and swin models and a smoother loss curve than fine-tuning the entire network. Besides, the final accuracy of output layer fine-tuning is also higher.

Test accuracy after 10 epochs	Fine-tune all units	Fine-tune output layer
ResNet	0.7261	0.9043
Swin_T	0.6882	0.9272

Figure 3: Best test accuracy

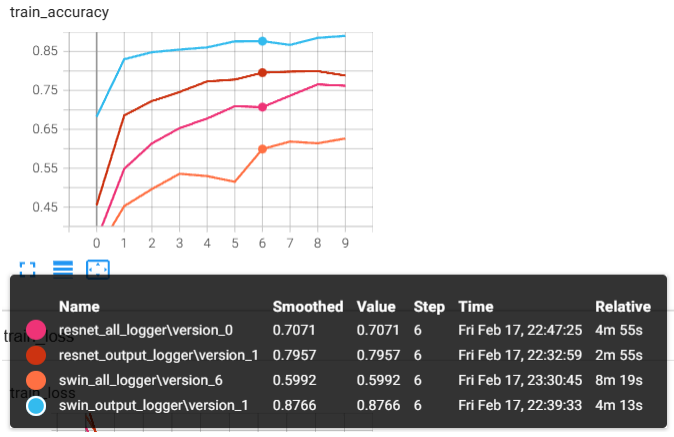

Figure 4: Train accuracy

The swin model performs better than resnet when fine-tuning the output layer, and resnet performs better when fine-tuning all units. From the experimental data demonstrated

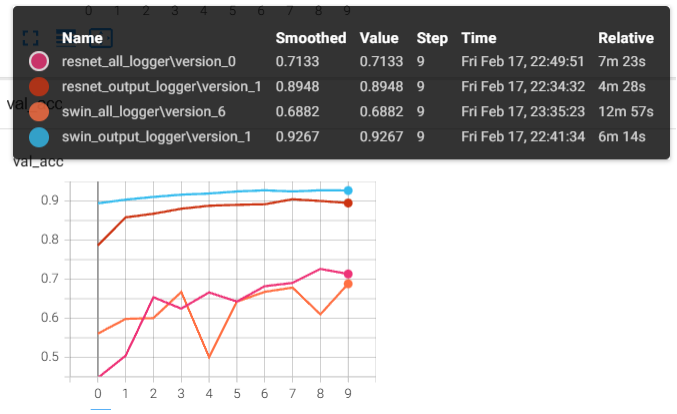

Figure 5: Test accuracy

above, there is not significant difference between these two models.
Note: use
tensorboard –logdir logs
to see charts after the program finished executing.

Problem 4 - Training a Small CNN

Part 1 (25 points)

For this problem you must use a toolbox. Train a CNN with three hidden convolutional layers that use the Mish activation function. Use 32 7×7 filters for the first layer, followed by 2×2 max pooling (stride of 2). The next two convolutional layers will use 64 3×3 filters followed by the Mish activation function. Prior to the softmax layer, you should have an average pooling layer that pools across the preceding feature map. Do not use a pre-trained CNN. Make sure to initialize your weights by explicitly calling the initializer. AdamW is the recommended optimizer. Tune your hyperparameters on a small version of the dataset to make sure your training loss goes down and things are set up correctly.

Train your model using all of the CIFAR-10 training data, and evaluate your trained system on the CIFAR-10 test data.

Visualize all of the $7 \times 7 \times 3$ filters learned by the first convolutional layer as an RGB image array (I suggest making a large RGB image that is made up of each of the smaller images, so it will have 4 rows and 8 columns). This visualization of the filters should be similar to the ones we saw in class. Note that you will need to normalize each filter to display them. Let $\mathbf{H}_t$ be the t'th filter learned by the network. The normalized version $\mathbf{H}'_t$ for

visualization is then given by

$$\mathbf{H}'_t = \frac{\mathbf{H}_t}{2\left(\|\mathbf{H}_t\|_\infty + \epsilon\right)} + \frac{1}{2},$$

where $\epsilon > 0$ (e.g., $\epsilon = 0.0001$) and the matrix norm is the infinity/max norm. This will make all values in the filter be values from 0 to 1 so that they can be displayed as an RGB image.

Display the training loss as a function of epochs. What is the accuracy on the test data? How did you initialize the weights? What optimizer did you use? Discuss your architecture and hyper-parameters.

Solution:

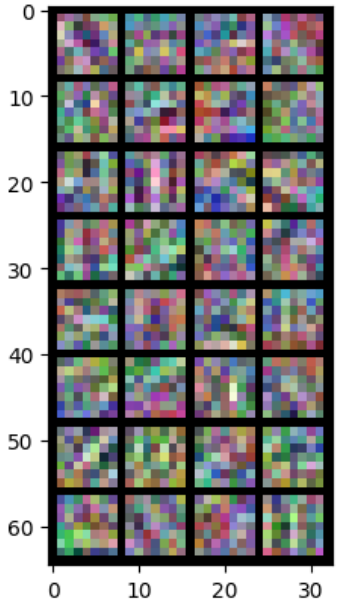

Figure 6: 7x7 Filter

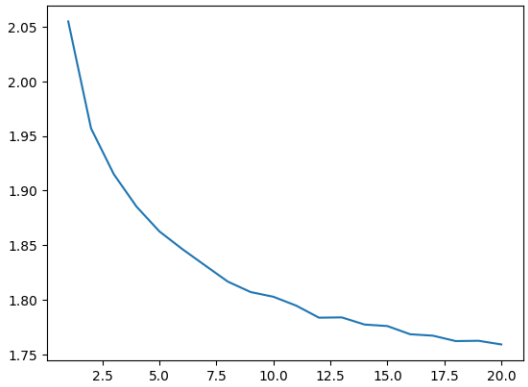

Figure 7: Loss as function of epochs

TEST DATA ACCURACY

Train Loss: 2.0455376385422923
Test Accuracy: 0.4817874203821656

Train Loss: 1.9446989842083142
Test Accuracy: 0.5362261146496815

Train Loss: 1.9041826563418065
Test Accuracy: 0.5607085987261147

Train Loss: 1.8826376900953405
Test Accuracy: 0.5638933121019108

Train Loss: 1.859487524422843
Test Accuracy: 0.5933519108280255

Train Loss: 1.8405252466421298
Test Accuracy: 0.5979299363057324

Train Loss: 1.829617742687235
Test Accuracy: 0.59484474522293

Train Loss: 1.8207104431698695
Test Accuracy: 0.6084792993630573

Train Loss: 1.8129007698934707

Test Accuracy: 0.6145501592356688

Train Loss: 1.7999518585327032
Test Accuracy: 0.6284832802547771

Train Loss: 1.7975693974653473
Test Accuracy: 0.5998208598726115

Train Loss: 1.788422469287882
Test Accuracy: 0.623109076433121

Train Loss: 1.7871669671114754
Test Accuracy: 0.6413216560509554

Train Loss: 1.7820441358534576
Test Accuracy: 0.6404259554140127

Train Loss: 1.7783465781785033
Test Accuracy: 0.631468949044586

Train Loss: 1.7746080065627232
Test Accuracy: 0.6319665605095541

Train Loss: 1.7728590232027157
Test Accuracy: 0.626890923566879

Train Loss: 1.7696576103225083
Test Accuracy: 0.6311703821656051

Train Loss: 1.7682114421863995
Test Accuracy: 0.6428144904458599

Train Loss: 1.7599914048029028
Test Accuracy: 0.6342555732484076

WEIGHT INITIALIZATION INFORMATION

The weights are initialized by the initializer function in the CNN class as below:

```
def initialize(self):
        for module in self.modules():
            if isinstance(module, nn.Conv2d):
                nn.init.kaiming_uniform_(module.weight)
            elif isinstance(module, nn.Linear):
                nn.init.kaiming_uniform_(module.weight)
```

which uses kaiming uniform initialization on convolution and linear layers.

DESCRIBE HYPER-PARAMETERS
Hyper-parameters in this part are given.

Part 2 (15 points)

Using the same architecture as in part 1, add in batch normalization between each of the hidden layers. Note that you should ensure you do not use a bias in your convolutional weight layers. Compare the training loss with and without batch normalization as a function of epochs. What is the final test accuracy? Visualize the filters.

Solution:

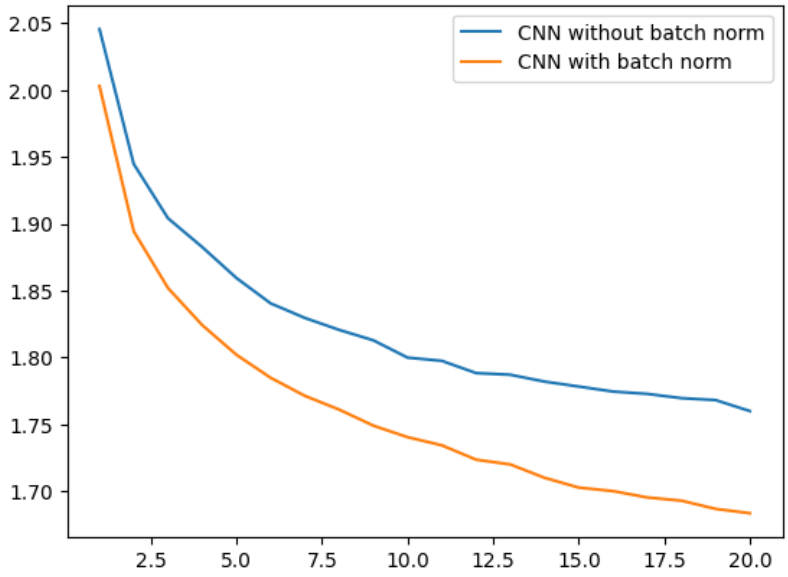

Figure 8: Loss with and without batch normalization as function of epochs

The final test accuracy at epoch 20 is 0.6851114649681529.

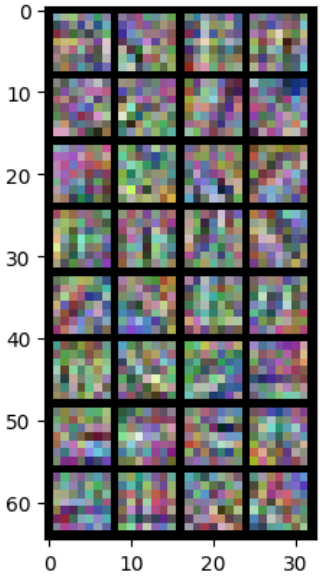

Figure 9: 7x7 filter with batch normalization

Part 3 (15 points)

Can you do better with a deeper and better network architecture? Optimize your CNN's architecture to improve performance. You may get significantly better results by using smaller filters for the first convolutional layer. Try using alternative normalization functions instead of BatchNorm and consider using more advanced optimizers than AdamW. What is your final accuracy? Describe your model's architecture and your design choices.

Note: Your model should perform better than the one in Part 1 and Part 2.

Solution:
After several attempts, BatchNorm performs better than LazyBatchNorm and performs similar to GroupNorm. I will prefer the alternative of BatchNorm, GroupNorm, as normalization function I use.
The AdamW Optimizer outperforms Adamax and SGD (with learning rate = 0.1). The final accuracy after 20 epoches with SGD optimizer is 0.7216361464968153 and it seems to have converged, around 2% lower than 0.7467157643312102 with AdamW Optimizer. Therefore, AdamW Optimizer will still be the choice.

In addition to the normalization and optimizer choices above, I added one convolution layer and one linear layer to build it deeper. In the previous parts, only one linear layer was built with 1024 input and 10 output. I built the new CNN such that the dimension

fed into the first linear layer is still 1024 but now with a hidden layer. The final accuracy reached is 0.7467157643312102 after 20 epoches and the highest accuracy reached is 0.7525875796178344 at epoch 18.

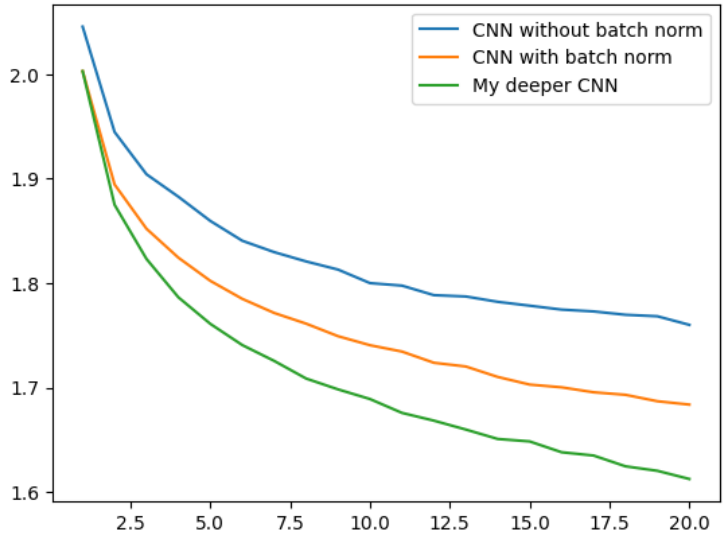

Figure 10: Loss comparison

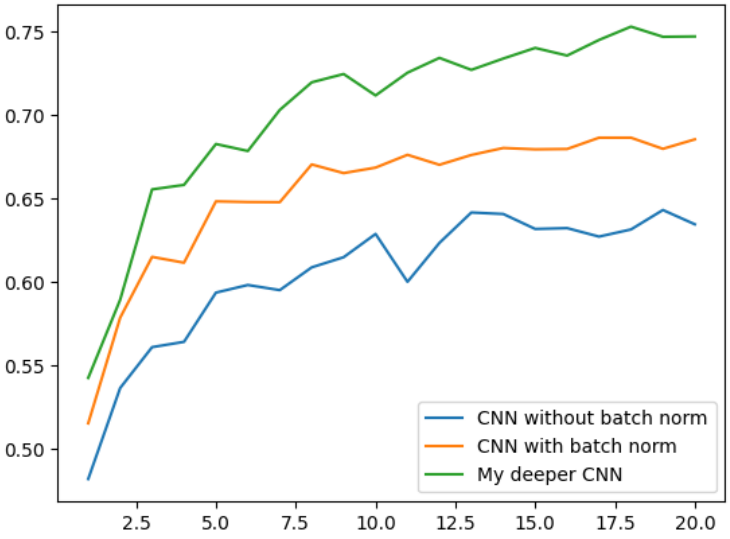

Figure 11: Accuracy comparison

Code Appendix

Discussed with Yuze Wang.
Reference:
https://gaoxiangluo.github.io/2021/08/01/Group-Norm-Batch-Norm-Instance-Norm-which-is-bett
https://pytorch.org/vision/main/models/generated/torchvision.models.resnet50.
html
https://pytorch.org/vision/main/models/generated/torchvision.models.swin_t.
html
https://github.com/Lightning-AI/lightning/issues/4102
https://medium.com/swlh/deep-learning-for-image-classification-creating-cnn-from-scratch-u

Problem 2 code:

```
from torchvision.io import read_image
from torchvision.models import resnet50, ResNet50_Weights, swin_t, Swin_T_Weights
import torch

img = read_image("./fruit.jpg")

res_weights = ResNet50_Weights.DEFAULT
model = resnet50(weights=res_weights)
model.eval()
preprocess = res_weights.transforms()
batch = preprocess(img).unsqueeze(0)

prediction = model(batch).squeeze(0).softmax(0)
class_id = prediction.argmax().item()
prediction = model(batch).squeeze(0).softmax(0)
class_id = torch.topk(prediction, 3).indices
for id in class_id:
    score = prediction[id].item()
    category_name = res_weights.meta["categories"][id]
    print(f"{category_name}: {100 * score:.1f}%")

swin_weights = Swin_T_Weights.DEFAULT
swin_model = swin_t(weights=swin_weights)
swin_model.eval()
preprocess = swin_weights.transforms()
batch = preprocess(img).unsqueeze(0)
```

```
prediction = swin_model(batch).squeeze(0).softmax(0)
class_id = prediction.argmax().item()
prediction = swin_model(batch).squeeze(0).softmax(0)
class_id = torch.topk(prediction, 3).indices
for id in class_id:
    score = prediction[id].item()
    category_name = swin_weights.meta["categories"][id]
    print(f"{category_name}: {100 * score:.1f}%")
```

Problem 3 code:

```
# In[1]:


import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
import pytorch_lightning as pl
from torchmetrics.functional import accuracy
import torchmetrics
from torch.utils.data import DataLoader
from torchvision.datasets import ImageFolder
from torchvision.models import resnet50, ResNet50_Weights, swin_t, Swin_T_Weights
# from models import FineTuningModel
from pytorch_lightning.loggers import TensorBoardLogger
import torch.optim as optim
from pytorch_lightning.callbacks import ModelCheckpoint


# In[2]:


data_dir = './data'
batch_size = 32
num_classes = 37
max_epochs = 10

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

```
#The images are resized to resize_size=[256] using interpolation=InterpolationMode.BILINEAR
#of crop_size=[224]. Finally the values are first rescaled to [0.0, 1.0] and then normalize
#and std=[0.229, 0.224, 0.225].
resnet_train_transforms = transforms.Compose([
    transforms.RandomResizedCrop(232, interpolation=transforms.InterpolationMode.BILINEAR),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])
resnet_test_transforms = transforms.Compose([
    transforms.Resize(232, interpolation=transforms.InterpolationMode.BILINEAR),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

resnet_train_dataset = torchvision.datasets.OxfordIIITPet(root = './data',
                                                  split = 'trainval',
                                                  transform = resnet_train_transforms,
                                                  download = True)

resnet_test_dataset = torchvision.datasets.OxfordIIITPet(root = './data',
                                                 split = 'test',
                                                 transform = resnet_test_transforms,
                                                 download=True)

resnet_train_loader = torch.utils.data.DataLoader(dataset = resnet_train_dataset,
                                                  batch_size = batch_size,
                                                  shuffle = True,
                                                  num_workers = 4)


resnet_test_loader = torch.utils.data.DataLoader(dataset = resnet_test_dataset,
                                                  batch_size = batch_size,
                                                  shuffle = False,
                                                  num_workers = 4)

#The images are resized to resize_size=[232] using interpolation=InterpolationMode.BICUBIC,
#of crop_size=[224]. Finally the values are first rescaled to [0.0, 1.0] and then normalize
#and std=[0.229, 0.224, 0.225].
swin_train_transforms = transforms.Compose([
```

```
    transforms.RandomResizedCrop(232, interpolation=transforms.InterpolationMode.BICUBIC),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])
swin_test_transforms = transforms.Compose([
    transforms.Resize(232, interpolation=transforms.InterpolationMode.BICUBIC),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

swin_train_dataset = torchvision.datasets.OxfordIIITPet(root = './data',
                                            split = 'trainval',
                                            transform = swin_train_transforms,
                                            download = True)

swin_test_dataset = torchvision.datasets.OxfordIIITPet(root = './data',
                                           split = 'test',
                                           transform = swin_test_transforms,
                                           download=True)

swin_train_loader = torch.utils.data.DataLoader(dataset = swin_train_dataset,
                                          batch_size = batch_size,
                                          shuffle = True,
                                          num_workers = 4)


swin_test_loader = torch.utils.data.DataLoader(dataset = swin_test_dataset,
                                          batch_size = batch_size,
                                          shuffle = False,
                                          num_workers = 4)



# In[3]:


class FineTuningModel(pl.LightningModule):
    def __init__(self, backbone, num_classes, model):
        super().__init__()
```

```
        self.backbone = backbone
        self.num_classes = num_classes

        # Replace the output layer with a new fully connected layer
        if model == "resnet":
            num_features = self.backbone.fc.in_features
            self.backbone.fc = nn.Linear(num_features, num_classes)
        elif model == "swin":
            num_features = self.backbone.head.in_features
            self.backbone.head = nn.Linear(num_features, num_classes)

        # Define the loss function
        self.loss_function = nn.CrossEntropyLoss()

    def forward(self, x):
        return self.backbone(x)

    def training_step(self, batch, batch_idx):
        x, y = batch
        y_hat = self.forward(x)
        train_loss = self.loss_function(y_hat, y)
        prediction = torch.argmax(y_hat, dim=1)
        correct = torch.sum(y == prediction).item()
        tensorboard_logs = {'train_acc_step': correct, 'train_loss_step': train_loss}

        return {'loss': train_loss, "correct": correct, "prediction_length": len(y), 'log':

    def training_epoch_end(self, outputs):
        avg_loss = torch.stack([x['loss'] for x in outputs]).mean()

        train_accuracy = sum([x['correct'] for x in outputs]) / sum(x['prediction_length']
        # tensorboard_logs = {'train_accuracy': train_accuracy, 'train_loss': avg_loss, 'st
        self.log('step', self.trainer.current_epoch)
        self.log('train_loss', avg_loss, logger=True, prog_bar=True, on_epoch=True, on_step
        self.log('train_accuracy', train_accuracy, logger=True, prog_bar=True, on_epoch=Tru
        # return {'loss': avg_loss, 'log': tensorboard_logs}

    def validation_step(self, batch, batch_idx):
        x, y = batch
        y_hat = self.forward(x)
        test_loss = self.loss_function(y_hat, y)
```

```
        prediction = torch.argmax(y_hat, dim=1)
        correct = torch.sum(y == prediction).item()
        # self.log('val_loss', test_loss)
        return {'val_loss': test_loss, "correct": correct, "prediction_length": len(y)}

    def validation_epoch_end(self, outputs):
        avg_loss = torch.stack([x['val_loss'] for x in outputs]).mean()

        val_acc = sum([x['correct'] for x in outputs]) / sum(x['prediction_length'] for x i
        # tensorboard_logs = {'val_loss': avg_loss, 'val_acc': val_acc, 'step': self.curren
        self.log('step', self.trainer.current_epoch)
        self.log('val_loss', avg_loss, logger=True, prog_bar=True, on_epoch=True, on_step=F
        self.log('val_acc', val_acc, logger=True, prog_bar=True, on_epoch=True, on_step=Fal

        # return {'log': tensorboard_logs}

    def configure_optimizers(self):
        optimizer = optim.AdamW(self.parameters())
        return optimizer



# ## Fine-tune output layer

# In[4]:


# Fine-tune ResNet-50
resnet_output = resnet50(weights=ResNet50_Weights.DEFAULT)
for param in resnet_output.parameters():
    param.requires_grad = False
resnet_output.fc.requires_grad_(True)
resnet_model_1 = FineTuningModel(backbone=resnet_output, num_classes=num_classes, model="re

resnet_output_logger = TensorBoardLogger('logs', name='resnet_output_logger')
trainer = pl.Trainer(accelerator='gpu', max_epochs=max_epochs, logger=resnet_output_logger)
trainer.fit(resnet_model_1, resnet_train_loader, resnet_test_loader)



# In[5]:
```

```
# Fine-tune Swin-T
swin_output = swin_t(weights=Swin_T_Weights.DEFAULT)
for param in swin_output.parameters():
    param.requires_grad = False
swin_output.head.requires_grad_(True)
swin_model_1 = FineTuningModel(backbone=swin_output, num_classes=num_classes, model="swin")

swin_output_logger = TensorBoardLogger('logs', name='swin_output_logger')
trainer = pl.Trainer(accelerator='gpu', max_epochs=max_epochs, logger=swin_output_logger)
trainer.fit(swin_model_1, swin_train_loader, swin_test_loader)


# ## Fine-tune all units

# In[6]:


# Fine-tune ResNet-50
resnet_all = resnet50(weights=ResNet50_Weights.DEFAULT)
#for param in resnet_all.parameters():
#    param.requires_grad = True
# resnet_all.fc.requires_grad_(True)
resnet_model_2 = FineTuningModel(backbone=resnet_all, num_classes=num_classes, model="resne

resnet_all_logger = TensorBoardLogger('logs', name='resnet_all_logger')
trainer = pl.Trainer(accelerator='gpu', max_epochs=max_epochs, logger=resnet_all_logger)
trainer.fit(resnet_model_2, resnet_train_loader, resnet_test_loader)



# In[7]:


# Fine-tune Swin-T
swin_all = swin_t(weights=Swin_T_Weights.DEFAULT)
for param in swin_all.parameters():
    param.requires_grad = True
swin_all.head.requires_grad_(True)
swin_model_2 = FineTuningModel(backbone=swin_all, num_classes=num_classes, model="swin")
```

```
swin_all_logger = TensorBoardLogger('logs', name='swin_all_logger')
trainer = pl.Trainer(accelerator='gpu', max_epochs=max_epochs, logger=swin_all_logger)
trainer.fit(swin_model_2, swin_train_loader, swin_test_loader)

# tensorboard --logdir logs
```

Problem 4 code:

```
# In[1]:


import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
from tqdm.notebook import tqdm
import matplotlib.pyplot as plt


# In[2]:


torch.cuda.is_available()


# In[3]:


device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
batch_size = 64
transforms = transforms.Compose([transforms.ToTensor(),
                                 transforms.Normalize(mean=[0.5, 0.5, 0.5],
                                                      std=[0.5, 0.5, 0.5])])

train_dataset = torchvision.datasets.CIFAR10(root = './data',
                                             train = True,
                                             transform = transforms,
                                             download = True)

test_dataset = torchvision.datasets.CIFAR10(root = './data',
```

```
                                            train = False,
                                            transform = transforms,
                                            download=True)

train_loader = torch.utils.data.DataLoader(dataset = train_dataset,
                                           batch_size = batch_size,
                                           shuffle = True)


test_loader = torch.utils.data.DataLoader(dataset = test_dataset,
                                           batch_size = batch_size,
                                           shuffle = True)


# ## Part1

# In[4]:


# Creating CNN
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()

        # 32 7x7 filters followed by 2x2 max pooling
        # W' = (W - F + 2P) / S + 1       initial W is 32
        self.convolution1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=7) # (32-

        # nn.Mish(),
        self.pool1 = nn.MaxPool2d(kernel_size = 2, stride = 2) # (26-2+2*0)/2+1 = 13

        # two 64 3x3 filters
        self.convolution2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3) # (13

        self.convolution3 = nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3) # (11
        self.pool2 = nn.AvgPool2d(kernel_size = 2, stride = 2)  # (9-2+2*0)/2+1 = 4

        self.fc = nn.Linear(64*4*4, 10)
        self.mish = nn.Mish()
        self.flatten = nn.Flatten()
        self.softmax = nn.Softmax(dim=1)
```

```
    def initialize(self):
        for module in self.modules():
            if isinstance(module, nn.Conv2d):
                nn.init.kaiming_uniform_(module.weight)
            elif isinstance(module, nn.Linear):
                nn.init.kaiming_uniform_(module.weight)

    def forward(self, x):
        x = self.convolution1(x)
        x = self.pool1(x)
        x = self.mish(self.convolution2(x))
        x = self.mish(self.pool2(self.convolution3(x)))

        x = self.flatten(x)
        x = self.fc(x)
        x = self.softmax(x)
        return x

loss_function = torch.nn.CrossEntropyLoss()
model = CNN().to(device)
model.initialize()
optimizer = torch.optim.Adam(model.parameters())


# In[5]:


def train(model):
    all_loss = 0

    for i, (images, labels) in enumerate(train_loader):
        images = images.to(device)
        labels = labels.to(device)

        outputs = model(images)
        loss = loss_function(outputs, labels)
        all_loss += loss.item()

        optimizer.zero_grad()
        loss.backward()
```

```
        optimizer.step()
    print(f"Train Loss: {all_loss/len(train_loader)}")
    return all_loss/len(train_loader)

def test(model):
    #all_loss = 0
    accuracy = []
    with torch.no_grad():
        for i, (images, labels) in enumerate(test_loader):
            images = images.to(device)
            labels = labels.to(device)

            outputs = model(images)
            #loss = loss_function(outputs, labels)
            #all_loss += loss.item()

            prediction = outputs.argmax(dim=1)
            accuracy.append((labels==prediction).sum().item() / labels.shape[0])
    print(f"Test Accuracy: {sum(accuracy)/len(accuracy)}") #  Test Loss: {all_loss/len(test
    return sum(accuracy)/len(accuracy)



# In[6]:


num_epochs = 20

train_loss_CNN = []
test_accuracy_CNN = []
for epoch in tqdm(range(num_epochs)):
    loss = train(model)
    train_loss_CNN.append(loss)
    accuracy = test(model)
    test_accuracy_CNN.append(accuracy)
    print('\n')


# In[7]:
```

```
plt.plot(range(1, num_epochs+1), train_loss_CNN)
plt.show()


# In[8]:


def visualize_filters(model):
    layer = model.convolution1.cpu()
    weight_tensor = layer.weight.detach().clone()
    print(weight_tensor.shape)
    # Ht' = Ht/(2|Ht| + e) + 0.5
    weight_tensor = weight_tensor/((torch.norm(weight_tensor, p=torch.inf) + 0.0001)*2) + 0
    visual = torchvision.utils.make_grid(weight_tensor, nrow=4, padding=1)
    plt.imshow(visual.numpy().transpose((1, 2, 0)))

visualize_filters(model)


# ## Part2

# In[9]:


# Creating CNN with batch norm
class CNN_batchnorm(nn.Module):
    def __init__(self):
        super(CNN_batchnorm, self).__init__()

        # 32 7x7 filters followed by 2x2 max pooling
        # W' = (W - F + 2P) / S + 1       initial W is 32
        self.convolution1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=7) # (32-
        self.conv1_bn = nn.BatchNorm2d(32)
        # nn.Mish(),
        self.pool1 = nn.MaxPool2d(kernel_size = 2, stride = 2) # (26-2+2*0)/2+1 = 13

        # two 64 3x3 filters
        self.convolution2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3) # (13
        self.conv2_bn = nn.BatchNorm2d(64)
        self.convolution3 = nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3) # (11
        self.conv3_bn = nn.BatchNorm2d(64)
```

```
        self.pool2 = nn.AvgPool2d(kernel_size = 2, stride = 2)  # (9-2+2*0)/2+1 = 4

        self.fc = nn.Linear(64*4*4, 10)
        self.mish = nn.Mish()
        self.flatten = nn.Flatten()
        self.softmax = nn.Softmax(dim=1)

    def initialize(self):
        for module in self.modules():
            if isinstance(module, nn.Conv2d):
                nn.init.kaiming_uniform_(module.weight)
            elif isinstance(module, nn.Linear):
                nn.init.kaiming_uniform_(module.weight)

    def forward(self, x):
        x = self.conv1_bn(self.convolution1(x))
        x = self.pool1(x)
        x = self.conv2_bn(self.mish(self.convolution2(x)))
        x = self.conv3_bn(self.mish(self.pool2(self.convolution3(x))))

        x = self.flatten(x)
        x = self.fc(x)
        x = self.softmax(x)
        return x

loss_function = torch.nn.CrossEntropyLoss()
model_batchnorm = CNN_batchnorm().to(device)
model_batchnorm.initialize()
optimizer = torch.optim.Adam(model_batchnorm.parameters())


# In[10]:


train_loss_CNN_batchnorm = []
test_accuracy_CNN_batchnorm = []
for epoch in tqdm(range(num_epochs)):
    loss = train(model_batchnorm)
    train_loss_CNN_batchnorm.append(loss)
    accuracy = test(model_batchnorm)
    test_accuracy_CNN_batchnorm.append(accuracy)
```

```
    print('\n')


# In[11]:


plt.plot(range(1, num_epochs+1), train_loss_CNN_batchnorm)
plt.show()
visualize_filters(model_batchnorm)


# ## Part3

# In[35]:


# Creating CNN with more
class CNN_new(nn.Module):
    def __init__(self):
        super(CNN_new, self).__init__()

        # W' = (W - F + 2P) / S + 1       initial W is 32

        self.convolution1 = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3)
        self.convolution_16_16_3 = nn.Conv2d(in_channels=16, out_channels=16, kernel_size=3
        self.convolution_16_32_3 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3
        self.convolution_32_32_3 = nn.Conv2d(in_channels=32, out_channels=32, kernel_size=3
        self.convolution_32_64_3 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3
        self.convolution_64_64_3 = nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3

        self.max_pool_2_2 = nn.MaxPool2d(kernel_size = 2, stride = 2)
        self.avg_pool_2_2 = nn.AvgPool2d(kernel_size = 2, stride = 2)

        self.fc1 = nn.Linear(64*4*4, 128)
        self.fc2 = nn.Linear(128, 10)
        self.mish = nn.Mish()
        self.flatten = nn.Flatten()
        self.softmax = nn.Softmax(dim=1)

        self.conv_bn_16 = nn.GroupNorm(4, 16)
        self.conv_bn_32 = nn.GroupNorm(8, 32)
```

```
        self.conv_bn_64 = nn.GroupNorm(16, 64)
        self.conv_bn_fc = nn.GroupNorm(32, 128)

    def initialize(self):
        for module in self.modules():
            if isinstance(module, nn.Conv2d):
                nn.init.kaiming_uniform_(module.weight)
            elif isinstance(module, nn.Linear):
                nn.init.kaiming_uniform_(module.weight)

    def forward(self, x):
        x = self.conv_bn_16(self.convolution1(x)) # (32 - 3)/1 + 1 = 30
        x = self.conv_bn_32(self.mish(self.convolution_16_32_3(x))) # (30 - 3)/1 + 1 = 28
        x = self.conv_bn_32(self.mish(self.convolution_32_32_3(x))) # (28 - 3)/1 + 1 = 26
        x = self.max_pool_2_2(x) # (26 - 2)/2 + 1 = 13

        x = self.conv_bn_64(self.mish(self.convolution_32_64_3(x))) # (13 - 3)/1 + 1 = 11
        x = self.conv_bn_64(self.mish(self.convolution_64_64_3(x))) # (11 - 3)/1 + 1 = 9
        x = self.avg_pool_2_2(x) # (9 - 2)/2 + 1 = 4

        x = self.flatten(x)
        x = self.conv_bn_fc(self.mish(self.fc1(x)))
        x = self.fc2(x)
        x = self.softmax(x)
        return x

loss_function = torch.nn.CrossEntropyLoss()
model_new = CNN_new().to(device)
model_new.initialize()
optimizer = torch.optim.AdamW(model_new.parameters())


# In[36]:


train_loss_CNN_new = []
test_accuracy_CNN_new = []
for epoch in tqdm(range(num_epochs)):
    loss = train(model_new)
    train_loss_CNN_new.append(loss)
    accuracy = test(model_new)
```

```
    test_accuracy_CNN_new.append(accuracy)
    print('\n')


# In[14]:


plt.plot(range(1, num_epochs+1), train_loss_CNN_new)
plt.show()
visualize_filters(model_new)


# In[37]:


plt.plot(range(1, num_epochs+1), train_loss_CNN, label="CNN without batch norm")
plt.plot(range(1, num_epochs+1), train_loss_CNN_batchnorm, label="CNN with batch norm")
plt.plot(range(1, num_epochs+1), train_loss_CNN_new, label="My deeper CNN")
plt.legend()
plt.show()


# In[38]:


plt.plot(range(1, num_epochs+1), test_accuracy_CNN, label="CNN without batch norm")
plt.plot(range(1, num_epochs+1), test_accuracy_CNN_batchnorm, label="CNN with batch norm")
plt.plot(range(1, num_epochs+1), test_accuracy_CNN_new, label="My deeper CNN")
plt.legend()
plt.show()
```