Open Lab 2

Multi-Layer Networks

CSCI 7850 - Deep Learning

Due: Sep. 28 @ 11:00pm

Assignment

Here are the details of what you need to do for this assignment:

- Create a set of python scripts (`cifar10-single.py` , `cifar10-wide.py` , `cifar10-deep.py` , and `cifar10-resid.py`) that run the corresponding model type (single, wide, deep, or residual) on the corresponding data set (cifar10):
  - Remember to use the `tanh` activation function for any hidden layers.
  - Remember to keep **all models** below the 130,000 parameter threshold for this assignment - anything up to this size we will accept as a fair advantage.
  - Most settings should be maintained below (max_epochs=50, batch_size=250, etc.) so that your script only prints the results line (validation accuracy) **exactly matching the format provided below**.
  - You should try to tune your architectures to perform as well as possible given the contraints provided: some balance between depth and width will be needed for good results. *You may also explore the use of normalization or regularlization layers to see how they perform in this assignment.*
- Use your scripts to run your models on the cifar10 data set - **perform 10 independent runs for each script**.
- Perform the same process above on models for the **cifar100** data set as well. Be sure to clearly name your scripts accordingly (i.e. `cifar100-single.py`) with **no more than 300,000 parameters**.
- Compile your data into *uniquely named* text files for each dataset/model that can be read in using `np.loadtxt` .
- Create an iPython Notebook file named `OL2.ipynb` which reads in the compiled results to produce a learning curve with mean and standard error. An example of one such plot is provided below, but your notebook should contain two plots in the end: once for cifar10 and one for cifar100. No code for model training/testing should be in this notebook file, it should only read in the results text files and plot them. **Be sure to label your learning curves using the `plt.legend()` function - I have left it off on-purpose below, but you should use Single, Deep, Wide, and Resid.**
- At the end of your notebook file, create a Markdown cell and compile answers to the following questions:
  1. Which of the architectures seems to perform best overall even under the constraints above?
  2. Why do you think this may be happening?
  3. What other choices (hyperparameters, architecture changes, data prep, etc.) do you think might be explored which could impact the performance?
  4. What outcome would you expect from changing the code in this way (hypothesis)?
  5. What process would you use to attempt to confirm your hypothesis and what steps would testing it involve?
  6. Which part(s) of the lab/code/experiments are still *unclear* to you after finishing this assignment?
  7. Which parts are you interested in learning more about?

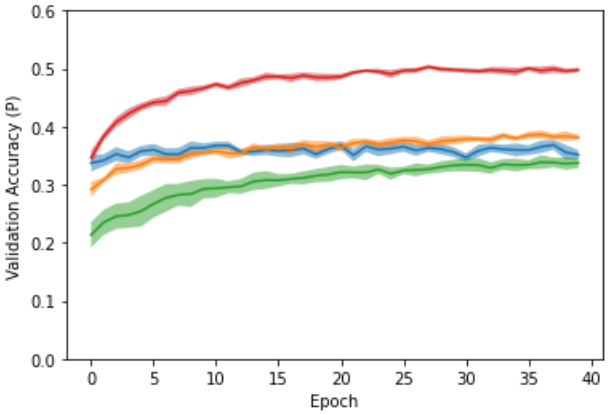
Example results plot (for illustration)

In [3]:
```
# All have this shape
results1.shape
```

Out[3]:
```
(10, 40)
```

In [4]:
```
plt.plot(np.arange(0,results1.shape[1]),
         np.mean(results1,0),label="Single")
plt.fill_between(np.arange(0,results1.shape[1]),
                 np.mean(results1,0)-(1.96*np.std(results1,0)/np.sqrt(results1.shape[0])),
                 np.mean(results1,0)+(1.96*np.std(results1,0)/np.sqrt(results1.shape[0])),
```

```
                 alpha=0.5)

plt.plot(np.arange(0,results2.shape[1]),
         np.mean(results2,0),label="Wide")
plt.fill_between(np.arange(0,results2.shape[1]),
                 np.mean(results2,0)-(1.96*np.std(results2,0)/np.sqrt(results2.shape[0])),
                 np.mean(results2,0)+(1.96*np.std(results2,0)/np.sqrt(results2.shape[0])),
                 alpha=0.5)

plt.plot(np.arange(0,results3.shape[1]),
         np.mean(results3,0),label="Deep")
plt.fill_between(np.arange(0,results3.shape[1]),
                 np.mean(results3,0)-(1.96*np.std(results3,0)/np.sqrt(results3.shape[0])),
                 np.mean(results3,0)+(1.96*np.std(results3,0)/np.sqrt(results3.shape[0])),
                 alpha=0.5)

plt.plot(np.arange(0,results4.shape[1]),
         np.mean(results4,0),label="Residual")
plt.fill_between(np.arange(0,results4.shape[1]),
                 np.mean(results4,0)-(1.96*np.std(results4,0)/np.sqrt(results4.shape[0])),
                 np.mean(results4,0)+(1.96*np.std(results4,0)/np.sqrt(results4.shape[0])),
                 alpha=0.5)

plt.ylim([0.0,0.6])
plt.ylabel('Validation Accuracy (P)')
plt.xlabel('Epoch')
# plt.legend(loc='lower right')
plt.show()
```



Submission

Create a zip archive which contains the following contents:

- cifar10-single.py
- cifar10-wide.py
- cifar10-deep.py
- cifar10-resid.py
- cifar100-single.py
- cifar100-wide.py
- cifar100-deep.py
- cifar100-resid.py
- OL2.ipynb
- **All** processed results text files (needed by np.loadtxt in your notebook)

Upload your zip archive to the course assignment system by the deadline at the top of this document.

Exploring Wide vs. Deep with CIFAR10 and CIFAR100

We will utilize some of the same coding and experimental protocols and principles that were explored to complete this lab, so be sure to refer back to Open Lab 1 if needed for review.

Let's start with some imports that will help us explain our problem of interest...

In [1]:
```
import numpy as np
import torch
import torchvision
import torchmetrics
import lightning.pytorch as pl
from torchinfo import summary
from torchview import draw_graph
import matplotlib.pyplot as plt
import pandas as pd
```

- **Numpy:** (`numpy`) for manipulating multidimensional arrays a.k.a. tensors
- **Pytorch:** (`torch`) for building/training deep learning models
- **TorchVision** (`torchvision`) for working with image data sets
- **TorchMetrics** (`torchmetrics`) for loss functions and other statistics
- **Lightning:** (`lightning.pytorch`) for building/training deep learning models quickly and easily
- **TorchInfo:** (`torchinfo`) for printing useful model details
- **TorchView:** (`torchview`) for visualizing deep learning models
- **Matplotlib:** (`matplotlib.pylot`) for visualizing the results of our simulations
- **Pandas:** (`pandas`) for loading tables with mixed data types

We also need to quickly configure PyTorch depending on the particular computing devices that we have available. Code and data have to be moved onto a device to use it, so often we need to specify which device we are placing/using certain parts of our code and data. We will see more on this below, but for now if we have an Nvidia GPU (and therefore CUDA) available then we will primarily use it. If we don't have such a device then we will just use the CPU.

In [2]:
```
if (torch.cuda.is_available()):
    device = ("cuda")
else:
    device = ("cpu")
print(torch.cuda.is_available())
```

```
True
```

The data set we will be exploring for this lab is the CIFAR10 data set which is loadable using the `torchvision` module since it is a relatively small image processing data set that's both in color and more difficult than the commonly-used MNIST data set. Models comparing MNIST are often only fractions of a percent different in performance when trained properly, and this makes it difficult to spot how architecture might influence results.

Instead, we will load up and prepare CIFAR which has 50,000 images for training/validation and 10,000 images for testing using 10 categories: 0) airplane, 1) automobile, 2) bird, 3) cat, 4) deer, 5) dog, 6) frog, 7) horse, 8) ship, and 9) truck.

Let's load the data and take a quick look at an example:

In [3]:
```
# Load the data set and scale to [-1,+1]
training_dataset = torchvision.datasets.CIFAR10(root='datasets',download=True,train=True)
testing_dataset = torchvision.datasets.CIFAR10(root='datasets',download=True,train=False)
x_train = (torch.Tensor(training_dataset.data) / 127.5) - 1.0
y_train = torch.Tensor(training_dataset.targets).to(torch.long)
x_test = (torch.Tensor(testing_dataset.data) / 127.5) - 1.0
y_test = torch.Tensor(testing_dataset.targets).to(torch.long)

# You can set a seed value here if you
# want to control the shuffling process...
rng = np.random.default_rng()
permutation = rng.permutation(x_train.shape[0])
split_point = int(x_train.shape[0] * 0.8) # 80%/20% split

# Split into validation/training - keep test
# set aside for later...
x_val = x_train[permutation][split_point:]
y_val = y_train[permutation][split_point:]
x_train = x_train[permutation][:split_point]
y_train = y_train[permutation][:split_point]

print(x_train.shape)
print(x_val.shape)
print(x_test.shape)
```

```
Files already downloaded and verified
Files already downloaded and verified
torch.Size([40000, 32, 32, 3])
torch.Size([10000, 32, 32, 3])
torch.Size([10000, 32, 32, 3])
```

In [4]:
```
x_train.shape
```

Out[4]:
```
torch.Size([40000, 32, 32, 3])
```

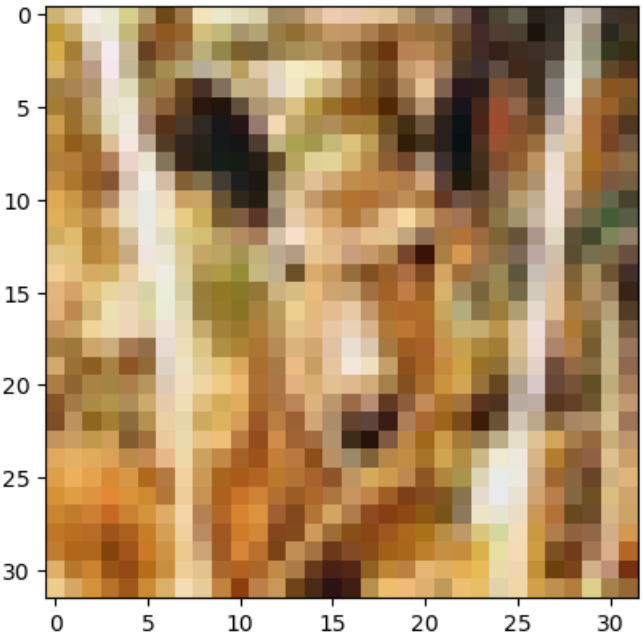
In [5]:
```
y_train.shape
```

Out[5]:
```
torch.Size([40000])
```

In [6]:
```
np.unique(y_train)
```

Out[6]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [7]:
```
plt.imshow((x_train[0].detach().numpy() + 1.0)/2.0)
plt.show()
```



In [8]:
```
y_train[0]
```

Out[8]:
```
tensor(4)
```

Note that this example is an image (very fuzzy due to pixelation of course) of a ship, so it matches that classification label (4).

You can also see that these images are 32x32 pixels, and that they are 8-bit color (red, green, blue - 3 channels).

For our experiments, we will be flattening these images and exploring the idea of using wide networks, deep networks, and residual connections to see how they compare to one-another.

Single-layer with Multi-class Targets

Let's start with the control model for our experiments, the single-layer network:

In [9]:
```
x_train.shape[1:].numel()
```

Out[9]:
```
3072
```

Carefully note the number of units in the input/output layers and the choice of activation function...

In [10]:
```
# Define model
class SingleLayerNetwork(torch.nn.Module):
    def __init__(self,
                 input_size,
                 output_size,
                 **kwargs):
```

```
        super().__init__(**kwargs)
        self.flatten_layer = torch.nn.Flatten()
        self.output_layer = torch.nn.Linear(input_size.numel(),
                                            output_size)

    def forward(self, x):
        y = x
        y = self.flatten_layer(y)
        y = self.output_layer(y)
        return y

    def predict(self, x):
        y = x
        y = self.forward(y)
        y = torch.softmax(y,-1)
        return y
```

In [11]:
```
# Define Trainable Module
class PLModel(pl.LightningModule):
    def __init__(self, module, **kwargs):
        # This is the contructor, where we typically make
        # layer objects using provided arguments.
        super().__init__(**kwargs) # Call the super class constructor
        self.module = module

        # This creates an accuracy function
        self.model_acc = torchmetrics.classification.Accuracy(task='multiclass',
                                                               num_classes=module.output_layer.out_features)
        # This creates a loss function
        self.model_loss = torch.nn.CrossEntropyLoss()

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

    def predict(self, x):
        return self.module.predict(x)

    def configure_optimizers(self):
        optimizer = torch.optim.Adam(self.parameters(), lr=0.001)
        return optimizer

    def training_step(self, train_batch, batch_idx):
        x, y_true = train_batch
        y_pred = self(x)
        acc = self.model_acc(y_pred,y_true)
        loss = self.model_loss(y_pred,y_true)
        self.log('train_acc', acc, on_step=False, on_epoch=True)
        self.log('train_loss', loss, on_step=False, on_epoch=True)
        return loss

    def validation_step(self, val_batch, batch_idx):
        x, y_true = val_batch
        y_pred = self(x)
        acc = self.model_acc(y_pred,y_true)
        loss = self.model_loss(y_pred,y_true)
        self.log('val_acc', acc, on_step=False, on_epoch=True)
        self.log('val_loss', loss, on_step=False, on_epoch=True)
        return loss

    def test_step(self, test_batch, batch_idx):
        x, y_true = test_batch
        y_pred = self(x)
        acc = self.model_acc(y_pred,y_true)
        loss = self.model_loss(y_pred,y_true)
        self.log('test_acc', acc, on_step=False, on_epoch=True)
        self.log('test_loss', loss, on_step=False, on_epoch=True)
        return loss
```

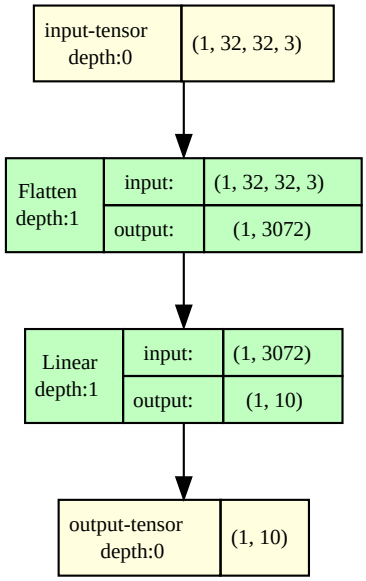
In [12]:
```
model = PLModel(SingleLayerNetwork(x_train.shape[1:],
                                   len(y_train.unique())))
```

In [13]:
```
summary(model, input_size=(1,)+x_train.shape[1:])
```

Out[13]:
```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
PLModel                                  [1, 10]                   --
├─SingleLayerNetwork: 1-1                --                        --
│    └─Flatten: 2-1                      [1, 3072]                 --
│    └─Linear: 2-2                       [1, 10]                   30,730
==========================================================================================
Total params: 30,730
Trainable params: 30,730
Non-trainable params: 0
Total mult-adds (Units.MEGABYTES): 0.03
==========================================================================================
Input size (MB): 0.01
Forward/backward pass size (MB): 0.00
Params size (MB): 0.12
Estimated Total Size (MB): 0.14
==========================================================================================
```

In [14]:
```
model_graph = draw_graph(model, input_size=(1,)+x_train.shape[1:],
                         device=device, depth=1)
model_graph.visual_graph
```

Out[14]:



In [15]:
```
predictions = model.predict(x_train[:5].to(device)).cpu().detach().numpy()
predictions
```

Out[15]:
```
array([[0.0931261 , 0.10521283, 0.10452959, 0.08475598, 0.11317062,
        0.11873344, 0.08733505, 0.14345883, 0.07305898, 0.07661855],
       [0.10041933, 0.08666887, 0.10054062, 0.12525252, 0.08890526,
        0.0920096 , 0.10732262, 0.09483743, 0.10912202, 0.09492172],
       [0.05666859, 0.09432079, 0.08927318, 0.05016853, 0.16673996,
        0.13043861, 0.09517089, 0.10667634, 0.03355034, 0.17699274],
       [0.13753262, 0.09578571, 0.09449632, 0.10602245, 0.09089877,
        0.07895108, 0.08137777, 0.09629358, 0.11924792, 0.09939376],
       [0.08209046, 0.09642393, 0.11606789, 0.05690419, 0.08389806,
        0.10205612, 0.11973917, 0.15327157, 0.07679201, 0.1127566 ]],
      dtype=float32)
```

In [16]:
```
predictions.shape
```

Out[16]:
```
(5, 10)
```

In [17]:
```
predictions.argmax(-1)
```

Out[17]:
```
array([7, 3, 9, 0, 7])
```

In [18]:
```
y_train[:5]
```

Out[18]:
```
tensor([4, 2, 5, 6, 7])
```

Outputs represent probabilities, so integer targets are mapped to the corresponding output unit (highest probability among the three predictions is assummed to the the class chosen by the network). The vectors, y_train, are therefore much more *sparse* (length 1) than the prediction vectors (length 10), so we employ a **sparse version of the accuracy and loss above**.

In [19]:
```
model.model_loss(model(x_train[:5].to(device)),
                 y_train[:5].to(device)).cpu().detach().numpy()
```

Out[19]:
```
array(2.1794205, dtype=float32)
```

In [20]:
```
model.model_acc(model(x_train[:5].to(device)),
                y_train[:5].to(device)).cpu().detach().numpy()
```

Out[20]:
```
array(0.2, dtype=float32)
```

We are **not using a softmax activation function (on the output layer)** because we have the crossentropy loss function which anticpates the outputs are not probabilities but instead represent **weighted selections (logits)**. This is just a slightly more efficient computation than using the softmax activation function: you can optionally use the softmax activation function by using the `predict()` method instead of the `forward()` function (also called using the `()` operator).

In [21]:
```
xy_train = torch.utils.data.DataLoader(list(zip(x_train,y_train)),
                                       shuffle=True, batch_size=250,
                                       num_workers=8)
xy_val = torch.utils.data.DataLoader(list(zip(x_val,y_val)),
                                     shuffle=False, batch_size=250,
                                     num_workers=8)
xy_test = torch.utils.data.DataLoader(list(zip(x_test, y_test)),
                                      shuffle=False, batch_size=250,
                                      num_workers=8)
```

```
/opt/conda/lib/python3.11/site-packages/torch/utils/data/dataloader.py:560: UserWarning: This DataLoader will creat
e 8 worker processes in total. Our suggested max number of worker in current system is 2, which is smaller than wha
t this DataLoader is going to create. Please be aware that excessive worker creation might get DataLoader running s
low or even freeze, lower the worker number to avoid potential slowness/freeze if necessary.
  warnings.warn(_create_warning_msg(
```

In [22]:
```
logger = pl.loggers.CSVLogger("logs",
                              name="OL2",
                              version="demo-0")
```

In [23]:
```
trainer = pl.Trainer(logger=logger,
                     max_epochs=50,
                     enable_progress_bar=True,
                     log_every_n_steps=0,
                     callbacks=[pl.callbacks.TQDMProgressBar(refresh_rate=20)])
```

```
GPU available: True (cuda), used: True
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
HPU available: False, using: 0 HPUs
```

In [24]:
```
trainer.validate(model, xy_val)
```

```
/opt/conda/lib/python3.11/site-packages/lightning/fabric/loggers/csv_logs.py:190: UserWarning: Experiment logs dire
ctory logs/OL2/demo-0 exists and is not empty. Previous log files in this directory will be deleted when the new on
es are saved!
  rank_zero_warn(
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
SLURM auto-requeueing enabled. Setting signal handlers.
/opt/conda/lib/python3.11/site-packages/torch/utils/data/dataloader.py:560: UserWarning: This DataLoader will creat
e 8 worker processes in total. Our suggested max number of worker in current system is 2, which is smaller than wha
t this DataLoader is going to create. Please be aware that excessive worker creation might get DataLoader running s
low or even freeze, lower the worker number to avoid potential slowness/freeze if necessary.
  warnings.warn(_create_warning_msg(
Validation: 0it [00:00, ?it/s]
```

Runningstage.validating metric	DataLoader 0
val_acc	0.12280000001192093
val_loss	2.294830560684204

Out[24]:
```
[{'val_acc': 0.12280000001192093, 'val_loss': 2.294830560684204}]
```

In [25]:
```python
trainer.fit(model, xy_train, xy_val)
```

```
/opt/conda/lib/python3.11/site-packages/lightning/pytorch/callbacks/model_checkpoint.py:615: UserWarning: Checkpoin
t directory logs/OL2/demo-0/checkpoints exists and is not empty.
  rank_zero_warn(f"Checkpoint directory {dirpath} exists and is not empty.")
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]

  | Name       | Type               | Params
--------------------------------------------------
0 | module     | SingleLayerNetwork | 30.7 K
1 | model_acc  | MulticlassAccuracy | 0
2 | model_loss | CrossEntropyLoss   | 0
--------------------------------------------------
30.7 K    Trainable params
0         Non-trainable params
30.7 K    Total params
0.123     Total estimated model params size (MB)
SLURM auto-requeueing enabled. Setting signal handlers.
Sanity Checking: 0it [00:00, ?it/s]
Training: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
Validation: 0it [00:00, ?it/s]
`Trainer.fit` stopped: `max_epochs=50` reached.
```

In [26]:
```python
trainer.validate(model, xy_val)
```

```
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
SLURM auto-requeueing enabled. Setting signal handlers.
```

Validation: 0it [00:00, ?it/s]

Runningstage.validating metric	DataLoader 0
val_acc	0.37619999051094055
val_loss	1.8291230201721191

Out[26]: [{'val_acc': 0.37619999051094055, 'val_loss': 1.8291230201721191}]

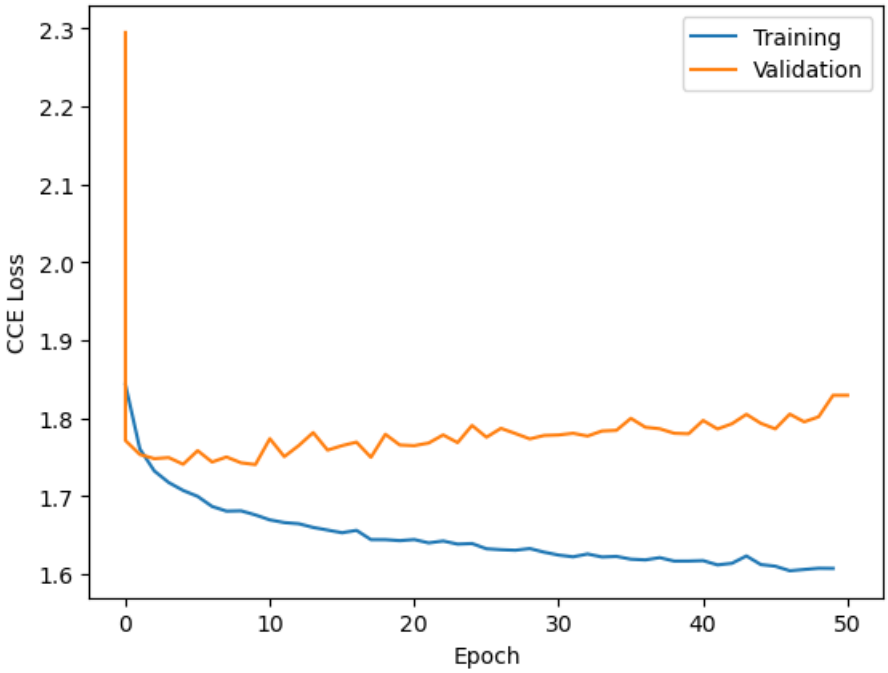
In [27]:
```
results = pd.read_csv(logger.log_dir+"/metrics.csv")
results
```

Out[27]:

	val_acc	val_loss	epoch	step	train_acc	train_loss
0	0.1228	2.294831	0	0	NaN	NaN
1	0.3900	1.771062	0	159	NaN	NaN
2	NaN	NaN	0	159	0.362700	1.842970
3	0.3966	1.753091	1	319	NaN	NaN
4	NaN	NaN	1	319	0.401400	1.760229
...	...	...	...	...	...	...
97	0.3748	1.801460	48	7839	NaN	NaN
98	NaN	NaN	48	7839	0.452175	1.607023
99	0.3762	1.829123	49	7999	NaN	NaN
100	NaN	NaN	49	7999	0.454925	1.606843
101	0.3762	1.829123	50	8000	NaN	NaN

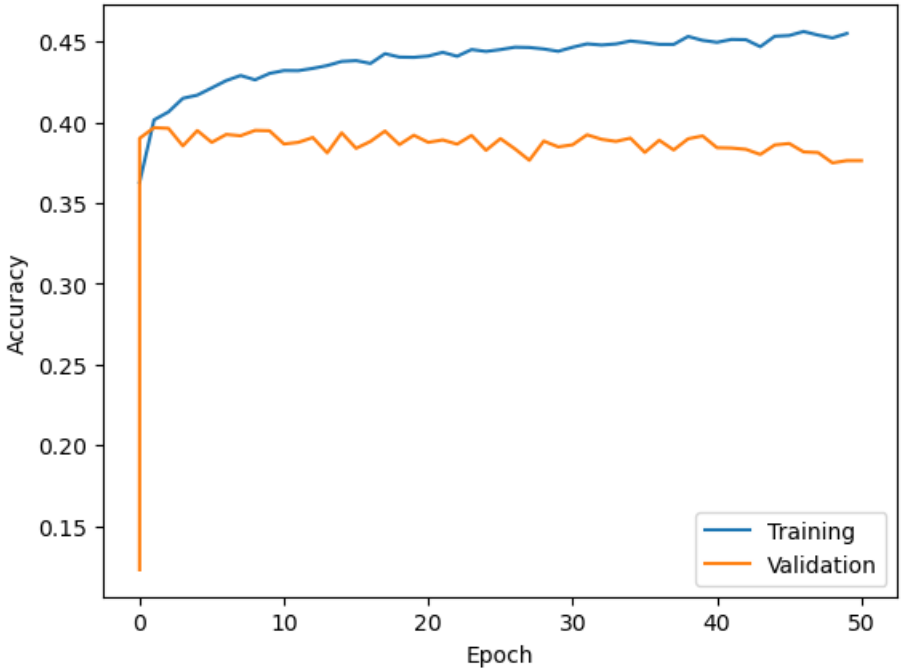
102 rows × 6 columns

In [28]:
```
plt.plot(results["epoch"][np.logical_not(np.isnan(results["train_loss"]))],
         results["train_loss"][np.logical_not(np.isnan(results["train_loss"]))],
         label="Training")
plt.plot(results["epoch"][np.logical_not(np.isnan(results["val_loss"]))],
         results["val_loss"][np.logical_not(np.isnan(results["val_loss"]))],
         label="Validation")
plt.legend()
plt.ylabel("CCE Loss")
plt.xlabel("Epoch")
plt.show()
```



In [29]:
```
plt.plot(results["epoch"][np.logical_not(np.isnan(results["train_acc"]))],
         results["train_acc"][np.logical_not(np.isnan(results["train_acc"]))],
         label="Training")
plt.plot(results["epoch"][np.logical_not(np.isnan(results["val_acc"]))],
```

```
        results["val_acc"][np.logical_not(np.isnan(results["val_acc"]))],
        label="Validation")
plt.legend()
plt.ylabel("Accuracy")
plt.xlabel("Epoch")
plt.show()
```



In [30]:
```
print("Validation accuracy:",*["%.8f"%(x) for x in
                               results['val_acc'][np.logical_not(np.isnan(results["val_acc"]))]])
```

```
Validation accuracy: 0.12280000 0.38999999 0.39660001 0.39620000 0.38530001 0.39480001 0.38749999 0.39240000 0.3916
0001 0.39489999 0.39469999 0.38650000 0.38749999 0.39050001 0.38100001 0.39340001 0.38380000 0.38810000 0.39449999
0.38609999 0.39190000 0.38749999 0.38900000 0.38640001 0.39179999 0.38260001 0.38980001 0.38350001 0.37639999 0.388
30000 0.38470000 0.38600001 0.39219999 0.38940001 0.38810000 0.39010000 0.38130000 0.38880000 0.38280001 0.38970000
0.39150000 0.38420001 0.38400000 0.38319999 0.38000000 0.38589999 0.38679999 0.38159999 0.38119999 0.37480000 0.376
19999 0.37619999
```

In [31]:
```
print("Testing accuracy:", trainer.test(model, xy_test)[0]['test_acc'])
```

```
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
SLURM auto-requeueing enabled. Setting signal handlers.
/opt/conda/lib/python3.11/site-packages/torch/utils/data/dataloader.py:560: UserWarning: This DataLoader will creat
e 8 worker processes in total. Our suggested max number of worker in current system is 2, which is smaller than wha
t this DataLoader is going to create. Please be aware that excessive worker creation might get DataLoader running s
low or even freeze, lower the worker number to avoid potential slowness/freeze if necessary.
  warnings.warn(_create_warning_msg(
Testing: 0it [00:00, ?it/s]
```

Runningstage.testing metric	DataLoader 0
test_acc	0.37869998812675476
test_loss	1.8349183797836304

```
Testing accuracy: 0.37869998812675476
```

Not very good performance here overall, but we now have a **baseline model** for performing comparisons with wide, deep, and residually connected multi-layer networks...

Widening...

To make fair comparisons, we have to keep the number of parameters in mind. Having too many extra parameters is an unfair advantage for a model. Of course, the single-layer model below is about as small as we can get with flattened input vectors, so the rest of the architectures will have more parameters by necessity.

For practicality in this assignment, keep all network models below **130,000** parameters. You can always use `summary()` to view the results of your work. We will also restrict ourselves to using only the `tanh` activation function. Here is an example that's too large, but should illustrate what to do...

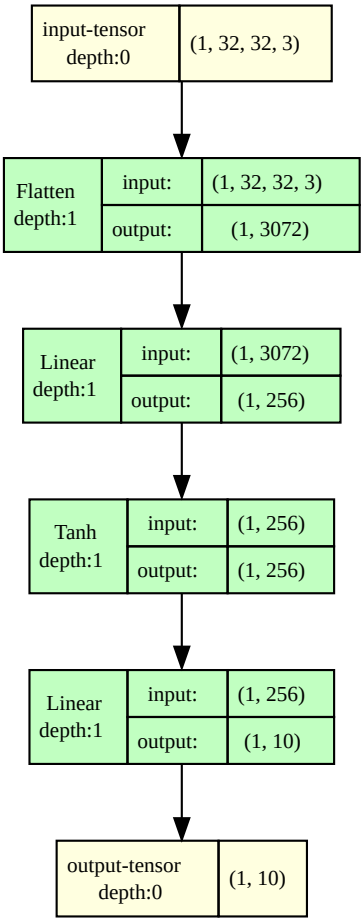
In [32]:
```
# Define model
class WideNetwork(torch.nn.Module):
    def __init__(self,
                 input_size,
                 output_size,
                 hidden_size,
                 **kwargs):
        super().__init__(**kwargs)
        self.flatten_layer = torch.nn.Flatten()
        self.hidden_layer = torch.nn.Linear(input_size.numel(),
                                            hidden_size)
        self.hidden_activation = torch.nn.Tanh()
        self.output_layer = torch.nn.Linear(hidden_size,
                                            output_size)

    def forward(self, x):
        y = x
        y = self.flatten_layer(y)
        y = self.hidden_activation(self.hidden_layer(y))
        y = self.output_layer(y)
        return y

    def predict(self, x):
        y = x
        y = self.forward(y)
        y = torch.softmax(y,-1)
        return y
```

In [33]:
```
hidden_dim = 256
model = PLModel(WideNetwork(x_train.shape[1:],
                            len(y_train.unique()),
                            hidden_dim))
summary(model, input_size=(1,)+x_train.shape[1:])
```

Out[33]:
```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
PLModel                                  [1, 10]                   --
├─WideNetwork: 1-1                       --                        --
│    └─Flatten: 2-1                      [1, 3072]                 --
│    └─Linear: 2-2                       [1, 256]                  786,688
│    └─Tanh: 2-3                         [1, 256]                  --
│    └─Linear: 2-4                       [1, 10]                   2,570
==========================================================================================
Total params: 789,258
Trainable params: 789,258
Non-trainable params: 0
Total mult-adds (Units.MEGABYTES): 0.79
==========================================================================================
Input size (MB): 0.01
Forward/backward pass size (MB): 0.00
Params size (MB): 3.16
Estimated Total Size (MB): 3.17
==========================================================================================
```

In [34]:
```
# Note the depth setting...
model_graph = draw_graph(model, input_size=(1,)+x_train.shape[1:],
                         device=device, depth=3)
model_graph.visual_graph
```

Out[34]:



Deepening...

Deep models alone have no residual connections, but focus on stacking layers to distribute the parameters. Here is an example of a deep network which is close to the same number of parameters above (still too large, and probably deeper than you might want) which illustrates the concept...

In [45]:

```
# Define model
class DeepNetwork(torch.nn.Module):
    def __init__(self,
                 input_size,
                 output_size,
                 hidden_size,
                 num_hidden_layers,
                 **kwargs):
        super().__init__(**kwargs)
        self.flatten_layer = torch.nn.Flatten()

        # Initial linear projection
        layers = [torch.nn.Linear(input_size.numel(),
                                  hidden_size)]
        for _ in range(num_hidden_layers):
            layers = layers + [torch.nn.Linear(hidden_size,
                                               hidden_size),
                               torch.nn.Tanh()]

        self.hidden_layers = torch.nn.Sequential(*layers)
        self.output_layer = torch.nn.Linear(hidden_size,
                                            output_size)

    def forward(self, x):
        y = x
        y = self.flatten_layer(y)
        y = self.hidden_layers(y)
        y = self.output_layer(y)
        return y

    def predict(self, x):
        y = x
        y = self.forward(y)
```

```
        y = torch.softmax(y,-1)
        return y
```

In [46]:
```
hidden_dim = 128
num_hidden_layers = 20
model = PLModel(DeepNetwork(x_train.shape[1:],
                            len(y_train.unique()),
                            hidden_dim,
                            num_hidden_layers))
summary(model, input_size=(1,)+x_train.shape[1:])
```

Out[46]:
```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
PLModel                                  [1, 10]                   --
├─DeepNetwork: 1-1                       --                        --
│    └─Flatten: 2-1                      [1, 3072]                 --
│    └─Sequential: 2-2                   [1, 128]                  --
│    │    └─Linear: 3-1                  [1, 128]                  393,344
│    │    └─Linear: 3-2                  [1, 128]                  16,512
│    │    └─Tanh: 3-3                    [1, 128]                  --
│    │    └─Linear: 3-4                  [1, 128]                  16,512
│    │    └─Tanh: 3-5                    [1, 128]                  --
│    │    └─Linear: 3-6                  [1, 128]                  16,512
│    │    └─Tanh: 3-7                    [1, 128]                  --
│    │    └─Linear: 3-8                  [1, 128]                  16,512
│    │    └─Tanh: 3-9                    [1, 128]                  --
│    │    └─Linear: 3-10                 [1, 128]                  16,512
│    │    └─Tanh: 3-11                   [1, 128]                  --
│    │    └─Linear: 3-12                 [1, 128]                  16,512
│    │    └─Tanh: 3-13                   [1, 128]                  --
│    │    └─Linear: 3-14                 [1, 128]                  16,512
│    │    └─Tanh: 3-15                   [1, 128]                  --
│    │    └─Linear: 3-16                 [1, 128]                  16,512
│    │    └─Tanh: 3-17                   [1, 128]                  --
│    │    └─Linear: 3-18                 [1, 128]                  16,512
│    │    └─Tanh: 3-19                   [1, 128]                  --
│    │    └─Linear: 3-20                 [1, 128]                  16,512
│    │    └─Tanh: 3-21                   [1, 128]                  --
│    │    └─Linear: 3-22                 [1, 128]                  16,512
│    │    └─Tanh: 3-23                   [1, 128]                  --
│    │    └─Linear: 3-24                 [1, 128]                  16,512
│    │    └─Tanh: 3-25                   [1, 128]                  --
│    │    └─Linear: 3-26                 [1, 128]                  16,512
│    │    └─Tanh: 3-27                   [1, 128]                  --
│    │    └─Linear: 3-28                 [1, 128]                  16,512
│    │    └─Tanh: 3-29                   [1, 128]                  --
│    │    └─Linear: 3-30                 [1, 128]                  16,512
│    │    └─Tanh: 3-31                   [1, 128]                  --
│    │    └─Linear: 3-32                 [1, 128]                  16,512
│    │    └─Tanh: 3-33                   [1, 128]                  --
│    │    └─Linear: 3-34                 [1, 128]                  16,512
│    │    └─Tanh: 3-35                   [1, 128]                  --
│    │    └─Linear: 3-36                 [1, 128]                  16,512
│    │    └─Tanh: 3-37                   [1, 128]                  --
│    │    └─Linear: 3-38                 [1, 128]                  16,512
│    │    └─Tanh: 3-39                   [1, 128]                  --
│    │    └─Linear: 3-40                 [1, 128]                  16,512
│    │    └─Tanh: 3-41                   [1, 128]                  --
│    └─Linear: 2-3                       [1, 10]                   1,290
==========================================================================================
Total params: 724,874
Trainable params: 724,874
Non-trainable params: 0
Total mult-adds (Units.MEGABYTES): 0.72
==========================================================================================
Input size (MB): 0.01
Forward/backward pass size (MB): 0.02
Params size (MB): 2.90
Estimated Total Size (MB): 2.93
==========================================================================================
```

In [ ]:
```
# Note the depth setting...
model_graph = draw_graph(model, input_size=(1,)+x_train.shape[1:],
                         device=device, depth=3)
model_graph.visual_graph
```

Residual Connections

Include residual connections can increase the number of parameters if used naively, but some adjustments to the structure can make this not so problematic...

In [58]:
```
# Define residual layer
class ResidualLayer(torch.nn.Module):
    def __init__(self,
                 hidden_size,
                 **kwargs):
        super().__init__(**kwargs)
        self.residual = torch.nn.Linear(hidden_size,
                                        hidden_size)
        self.activation = torch.nn.Tanh()

    def forward(self, x):
        y = x
        y = self.activation(self.residual(y))
        y = y + x # Add residual
        return y

class ResidualNetwork(torch.nn.Module):
    def __init__(self,
                 input_size,
                 output_size,
                 hidden_size,
                 num_hidden_layers,
                 **kwargs):
        super().__init__(**kwargs)
        self.flatten_layer = torch.nn.Flatten()

        # Initial linear projection
        layers = [torch.nn.Linear(input_size.numel(),
                                  hidden_size)]
        for _ in range(num_hidden_layers):
            layers = layers + [ResidualLayer(hidden_size)]

        self.hidden_layers = torch.nn.Sequential(*layers)
        self.output_layer = torch.nn.Linear(hidden_size,
                                            output_size)

    def forward(self, x):
        y = x
        y = self.flatten_layer(y)
        y = self.hidden_layers(y)
        y = self.output_layer(y)
        return y

    def predict(self, x):
        y = x
        y = self.forward(y)
        y = torch.softmax(y,-1)
        return y
```

In [59]:
```
hidden_dim = 128
num_hidden_layers = 20
model = PLModel(ResidualNetwork(x_train.shape[1:],
                                len(y_train.unique()),
                                hidden_dim,
                                num_hidden_layers))
summary(model, input_size=(1,)+x_train.shape[1:])
```

Out[59]:
```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
PLModel                                  [1, 10]                   --
├─ResidualNetwork: 1-1                   --                        --
│    └─Flatten: 2-1                      [1, 3072]                 --
│    └─Sequential: 2-2                   [1, 128]                  --
│    │    └─Linear: 3-1                  [1, 128]                  393,344
│    │    └─ResidualLayer: 3-2           [1, 128]                  16,512
│    │    └─ResidualLayer: 3-3           [1, 128]                  16,512
│    │    └─ResidualLayer: 3-4           [1, 128]                  16,512
│    │    └─ResidualLayer: 3-5           [1, 128]                  16,512
│    │    └─ResidualLayer: 3-6           [1, 128]                  16,512
│    │    └─ResidualLayer: 3-7           [1, 128]                  16,512
│    │    └─ResidualLayer: 3-8           [1, 128]                  16,512
│    │    └─ResidualLayer: 3-9           [1, 128]                  16,512
│    │    └─ResidualLayer: 3-10          [1, 128]                  16,512
│    │    └─ResidualLayer: 3-11          [1, 128]                  16,512
│    │    └─ResidualLayer: 3-12          [1, 128]                  16,512
│    │    └─ResidualLayer: 3-13          [1, 128]                  16,512
│    │    └─ResidualLayer: 3-14          [1, 128]                  16,512
│    │    └─ResidualLayer: 3-15          [1, 128]                  16,512
│    │    └─ResidualLayer: 3-16          [1, 128]                  16,512
│    │    └─ResidualLayer: 3-17          [1, 128]                  16,512
│    │    └─ResidualLayer: 3-18          [1, 128]                  16,512
│    │    └─ResidualLayer: 3-19          [1, 128]                  16,512
│    │    └─ResidualLayer: 3-20          [1, 128]                  16,512
│    │    └─ResidualLayer: 3-21          [1, 128]                  16,512
│    └─Linear: 2-3                       [1, 10]                   1,290
==========================================================================================
Total params: 724,874
Trainable params: 724,874
Non-trainable params: 0
Total mult-adds (Units.MEGABYTES): 0.72
==========================================================================================
Input size (MB): 0.01
Forward/backward pass size (MB): 0.02
Params size (MB): 2.90
Estimated Total Size (MB): 2.93
==========================================================================================
```

In [ ]:
```
# Note the depth setting...
model_graph = draw_graph(model, input_size=(1,)+x_train.shape[1:],
                         device=device, depth=3)
model_graph.visual_graph
```

Cifar100

An alternative data set is the cifar100 data set which has 100 different classes to predict using the images.

```
training_dataset = torchvision.datasets.CIFAR100(root='datasets',download=True,train=True)
testing_dataset = torchvision.datasets.CIFAR100(root='datasets',download=True,train=False)
```

We can use this to examine if the number of classes has any additional bearing on the results. Note that the single-layer network will be quite a bit larger in this case (about 300,000 parameters). Make sure any of the other models you build for this data set **do not exeed the size of the single-layer model**.

Scripting this process

You can always utilize the tools above outside of the notebook environment, but it's wise to remove the matplotlib tools and other interactive/exploratory parts of your code before doing so since a script will be run from the command line and therefore does not have the same level of interactivity as what we are using above. An example of doing this for the cifar10 data would look similar to the following (not all details above are included, nor all details needed for the assignment below):

```
#!/usr/bin/env python3

import numpy as np
import torch
import torchvision
import torchmetrics
import lightning.pytorch as pl
from torchinfo import summary
from torchview import draw_graph
```

```
import matplotlib.pyplot as plt
import pandas as pd

if (torch.cuda.is_available()):
    device = ("cuda")
else:
    device = ("cpu")
print(torch.cuda.is_available())

training_dataset = torchvision.datasets.CIFAR10(root='datasets',download=True,train=True)
testing_dataset = torchvision.datasets.CIFAR10(root='datasets',download=True,train=False)


# Note that other code goes here...

print("Validation accuracy:",*["%.8f"%(x) for x in
                                results['val_acc'][np.logical_not(np.isnan(results["val_acc"]))]])
```

If you put this in a file named `cifar10-single.py`, then you can use the `python3 cifar10-single.py` command from the terminal to run the script. Alternatively, you can add executable permissions to the script using `chmod +x cifar10-single.py` and then you can simply run `./cifar10-single.py` assuming the script is in the current directory.

Also, you can run this as a batch job on the SLURM scheduler on the cluster if you wrap it in a shell script which passes it through apptainer.

```
#!/bin/sh

apptainer exec /home/shared/sif/csci-2023-Fall.sif python3 cifar10-single.py
```