Machine Learning & Discovery *ECS 271, Fall 2024*

HW2

- You will learn how to train a deep network using PyTorch tool. Please read the following tutorial. You may skip the data parallelism section.

https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html

- CIFAR10 is a dataset of $60,000$ color images of size 32×32 from 10 categories. Please download the PyTorch tutorial code for CIFAR10 to start:

https://pytorch.org/tutorials/_downloads/cifar10_tutorial.py

- When you run the tutorial code, it will download CIFAR10 dataset for you. Please follow the instructions in the following link to install PyTorch: *https://pytorch.org/*

- To learn more, you can also find a tutorial for MNIST here:

https://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html#sphx-glr-beginner-blitz-neural-networks-tutorial-py

and the sample model for MNIST here:

https://github.com/pytorch/examples/blob/master/mnist/main.py

and the sample code for Imagenet here:

https://github.com/pytorch/examples/blob/master/imagenet/main.py

- For all the following sections, train the model for 50 epochs and plot the curve for loss, training accuracy, and test accuracy evaluated every epoch.

1. Run the tutorial code out of the box and make sure you get reasonable results. You will report these results in Section 4, so no report needed here.
2. Change the code to have only a single fully connected layer. The model will have a single layer that connects the input to the output. What is the number of parameters? In PyTorch, "nn.Linear" can be used for fully connected layer.

Answer:
The given colored image size is 32×32. Since the images are colored, we need to multiply by 3 to account for the RGB values. Therefore, the input features should be

$$32 \times 32 \times 3 = 3072$$

Additionally, since there are 10 categories as the introduction mentioned, we need to have 10 nodes for the output layer. Hence, the total number of parameters should be

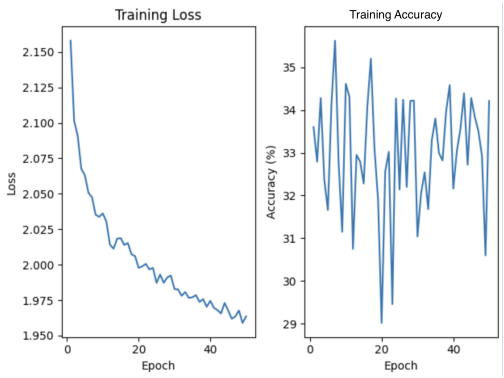
$$((32 \times 32 \times 3) + 1) \times 10 = 30730$$

Code:

```
    class Net(nn.Module):
        def __init__(self):
            super(Net, self).__init__()
            self.fc = nn.Linear(32 * 32 * 3, 10)

```

```
        def forward(self, val):
            val = val.view(-1, 32 * 32 * 3)
            val = self.fc(val)
            return val

    net = Net()
```

Plot for loss and accuracy:



3. Change the code to have multiple fully connected layers. Try having a layer from input to 110 neurons and then a layer to 74 neurons, and finally a layer to 10 neurons, one for each category. What happens if you do not use ReLU? Describe why.

Answer:
As we can tell from the results below, the training loss for the model with ReLU is less than the one without ReLU and the accuracy for using ReLU is obviously higher. Hence, we could know that the performance with ReLU is better than the one without ReLU.

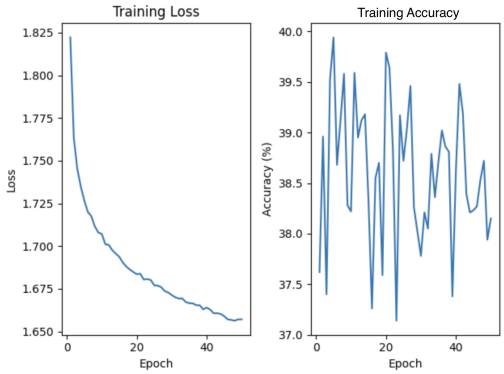
That's mainly because without ReLU, the output for each layer is consider as linear pattern. In other word, with mutiple-layers, the model is consider as linear pattern as well. However, with ReLU, it fix some problem with the non-linear patterns, which is good for the model here. And it lead to the improvement for the performance.

Code without ReLU:

```
    class Net(nn.Module):
        def __init__(self):
            super(Net, self).__init__()
            self.fc = nn.Linear(32 * 32 * 3, 110)
            self.fc1 = nn.Linear(110, 74)
            self.fc2 = nn.Linear(74, 10)

        def forward(self, val):
            val = val.view(-1, 32 * 32 * 3)
            val = self.fc(val)
            val = self.fc1(val)
            val = self.fc2(val)
```

```
            return val

    net = Net()
```

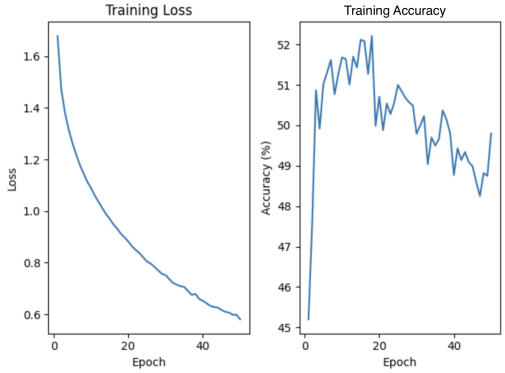
Plot for loss and accuracy without ReLU:



Code with ReLU:

```
    class Net(nn.Module):
        def __init__(self):
            super(Net, self).__init__()
            self.fc = nn.Linear(32 * 32 * 3, 110)
            self.fc1 = nn.Linear(110, 74)
            self.fc2 = nn.Linear(74, 10)

        def forward(self, val):
            val = val.view(-1, 32 * 32 * 3)
            val = F.relu(self.fc(val))
            val = F.relu(self.fc1(val))
            val = self.fc2(val)
            return val

    net = Net()
```

Plot for loss and accuracy with ReLU:



4. Change the code by adding two convolutional layers along with maxpooling layers before the fully connected layers. This will be similar to the example in the tutorial. Use this model for the following sections.

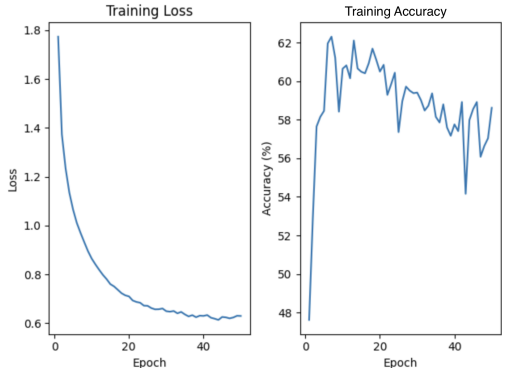
Answer:
Since the original class Net meet the requirements, I change with the training process to show the plot for training loss and training accuracy for each epoch.

Code:

```
    for epoch in range(50):  # loop over the dataset multiple times
        running_loss = 0.0
        for i, data in enumerate(trainloader, 0):
            inputs, labels = data
            optimizer.zero_grad()
            outputs = net(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()

        train_losses.append(running_loss / len(trainloader))
        correct = 0
        total = 0
        with torch.no_grad():
            for data in testloader:
                images, labels = data
                outputs = net(images)
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()

        accuracies.append(100 * correct / total)

    print('Finished Training')
    dataiter = iter(testloader)
    images, labels = next(dataiter)

    epochs = range(1, 51)
    plt.figure()

    plt.subplot(1, 2, 1)
    plt.plot(epochs, train_losses, label='Training loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.title('Training Loss')

    plt.subplot(1, 2, 2)
    plt.plot(epochs, accuracies, label='Training accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy (%)')
    plt.title('Training Accuracy')

    plt.tight_layout()
```

```
45      plt.show()
```

Plot:



5. Try multiple batch sizes to see the effect and describe the findings. Please use batch size of 1, 4, and 1000. If 1000 does not fit into the memory of your machine, please feel free to reduce it to the largest possible number.

   Answer:
   (Since 1000 is too large for me, I reduced it to 500.) From the experiments I conducted below, we can see that, for a batch size of 1, the loss and accuracy are unstable. This instability will affect the model's convergence. The accuracy is in the middle among the three. As the number of epochs increases, the accuracy decreases.

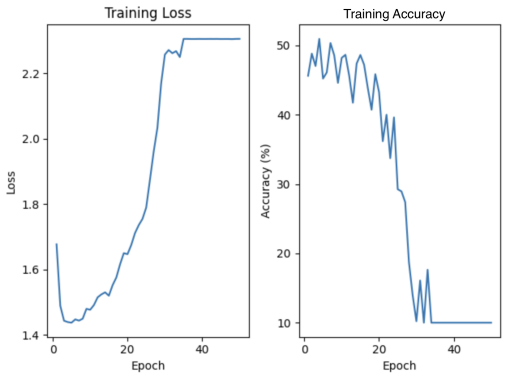
   For a batch size of 4, it is considered more stable in terms of loss and accuracy compared to batch size 1, and its performance is the best among the three.

   For a batch size of 500, the loss and accuracy are the lowest, it takes the most memory and has the worst performance, but it saves time.

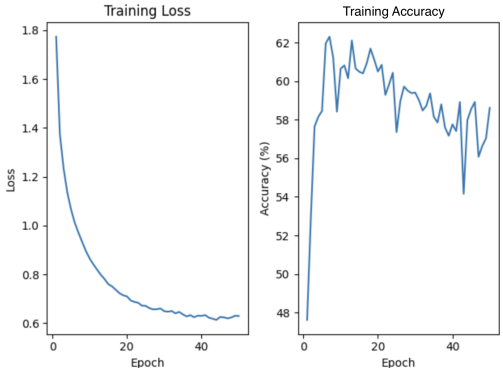
   The code I changed:

```
1       trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
2                                               download=True, transform=
                                                   transform)
3       trainloader = torch.utils.data.DataLoader(trainset, batch_size=1,
4                                                 shuffle=True,
                                                     num_workers=2)
5       # the batch size here is the one I changed for.
6
7       testset = torchvision.datasets.CIFAR10(root='./data', train=False,
8                                              download=True, transform=
                                                   transform)
9       testloader = torch.utils.data.DataLoader(testset, batch_size=1,
10                                               shuffle=False,
                                                     num_workers=2)
```

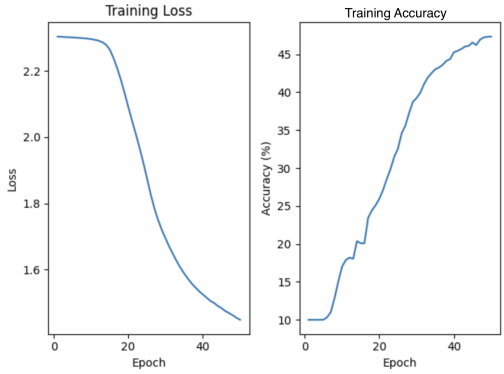
batch size of 1:



batch size of 4:



batch size of 500:



6. Try multiple learning rates to see the effect and describe the findings. Please use learning rates of 10, 0.1, 0.01, and 0.001.

   Answer:
   As seen in the experiment below, we can observe that as the learning rate decreases, both the accuracy and loss improve. In other words, the performance improves as the learning rate goes down.

   For a learning rate of 10, the training process is unstable. The loss is large, and I had to use a logarithmic operation to plot it. This instability is due to the large updates to the model's parameters caused by the high learning rate, making it difficult for the model to converge properly. The accuracy

is around 10%, which is expected since CIFAR-10 consists of 10 categories, meaning the model is essentially guessing randomly.

For a learning rate of 0.1, the training process is better than with a learning rate of 10, but it is still unstable. The accuracy is again around 10%, indicating random guessing, and the performance is poor. The model struggles to stabilize because the learning rate is still too large for proper convergence.

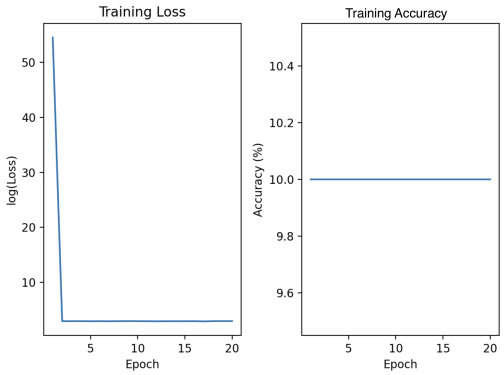
For a learning rate of 0.01, the performance is better than the two above, but it is still unstable as the epochs progress. While the model shows some improvement, it is still not fully stable, indicating that a smaller learning rate may be needed for better convergence.

For a learning rate of 0.001, the performance is the best and is considered stable and consistent among these four learning rates. However, it seems to take more time to converge compared to the others. The smaller learning rate results in more gradual updates to the model parameters, leading to a slower but more stable convergence.

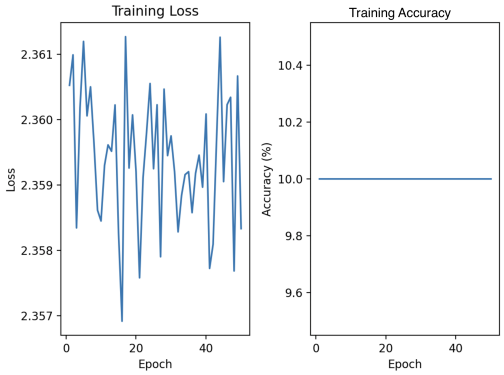
The code I changed:

```
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9) # the value lr is the one I change for.
```

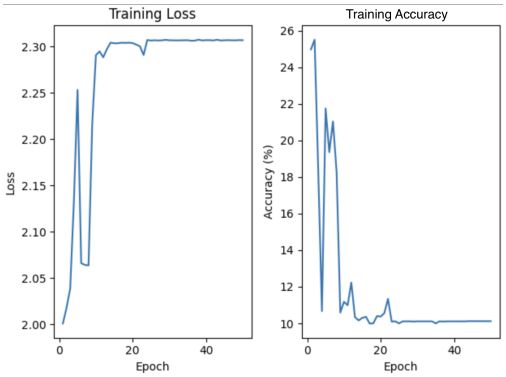
Learning rate of 10:



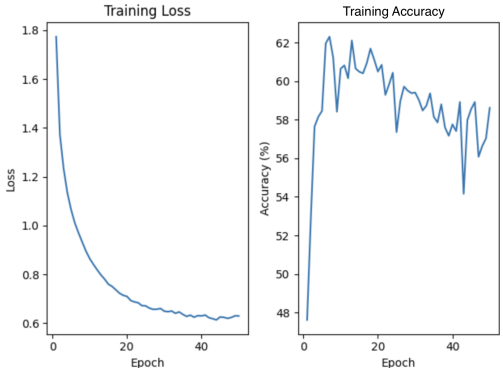
Learning rate of 0.1:



Learning rate of 0.01:



Learning rate of 0.001:



7. Please add some data augmentation to avoid overfitting. Note that you need to do this only for the trainnig and not the testing. You may use line 233-253 from Imagenet sample code:

   https://github.com/pytorch/examples/blob/master/imagenet/main.py

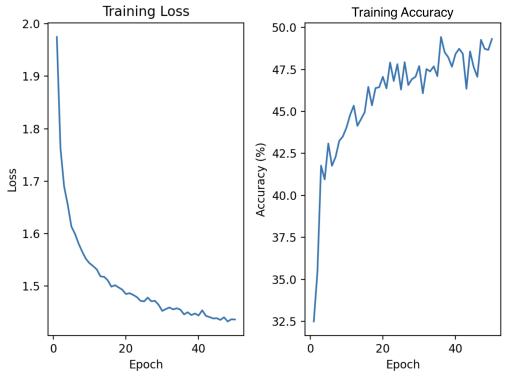
   "RandomResizedCrop" samples a random patch from the image to train the model on. "RandomHorizontalFlip" flips randomly chosen images horizontally.

   Answer:
   From the experiment, we can see that after adding data augmentation, although the training accuracy decreases, overfitting is reduced. The accuracy originally began to decrease after about epoch 10, which is overfitting. After adding data augmentation, the training accuracy completely shows an upward trend.

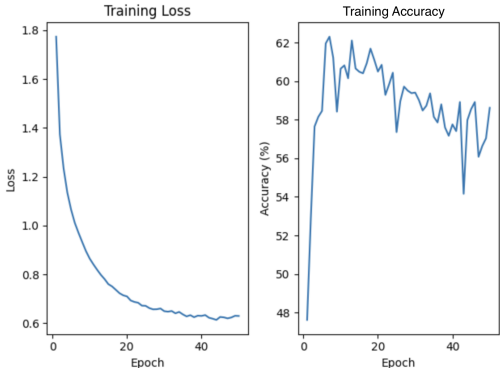
The code I changed:

```
    transform = transforms.Compose([
        transforms.RandomResizedCrop(32),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
    ])
```

plot with data augmentation:



plot without data augmentation:



8. Change the loss function from Cross Entropy to Mean Squared Error and report the effect.

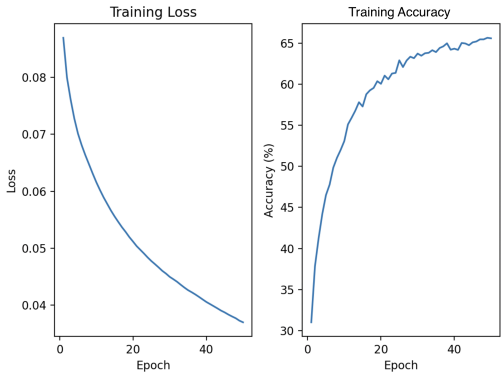
Answer:
From the following experiment, we can find that the training loss of MSE is relatively small at the beginning and is decreasing relatively steadily. That is to say, the training loss range of MSE is relatively small and the model is stable learning. The training loss of Cross Entropy starts from about 1.8, and the training loss decreases rapidly as the training progresses. Combined with theoretical knowledge, it can be found that compared with MSE, Cross Entropy has a greater penalty for wrong predictions in classification tasks.

From the perspective of training accuracy, the accuracy of MSE grows slowly but is higher in the later stage. The accuracy of Cross Entropy grows faster and quickly stabilizes at about 57%. Therefore, for such classification tasks, Cross Entropy can classify quickly and effectively.

The code I changed:

```
    criterion = nn.MSELoss()
    optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
    ...
    outputs = net(inputs)
    loss = criterion(outputs, torch.eye(10 , labels.device)[labels])
    loss.backward()
    optimizer.step()
```

plot with Mean Squared Error:



plot with Cross Entropy: