Daniel Rodriguez:

A5 Convolutional Neural Network (Total 150pts)

1. Import libraries (Total 6pts)

1.1 Import torch, torchvision, torchvision.transforms, torch.utils.data and torch.nn (6pts)

```
# TODO
import torch
import torchvision
from torchvision import transforms as tr
from torch.utils import data
import torch.nn as nn
```

2. Data Preparation (Total 32pts)

2.1 Image Transformation (12pts)

Define a transformation pipeline using torchvision.transforms.Compose.

In the pipeline, use **ColorJitter, GaussianBlur, RandomHorizontalFlip, ToTensor and Normalize** from the transforms library.

For the first four transformations, use suitable parameters of your informed choice. At the end, normalize the images with mean 0.5 and variance 0.5.

Read about these transformations here: https://pytorch.org/vision/0.9/transforms.html

```
# TODO
transform_pipeline = tr.Compose([
  tr.ColorJitter(brightness=0.5, contrast=0.5, saturation=0.5, hue=0.5),
  tr.GaussianBlur(kernel_size=(3,3), sigma=(0.5,2.0)),
  tr.RandomHorizontalFlip(p=0.5),
  tr.ToTensor(),
  tr.Normalize(mean=[0.5,0.5,0.5], std=[0.5,0.5,0.5])
])
```

2.2 Prepare train and test set by loading CIFAR10 dataset from torchvision.datasets. (4pts)

Make sure you are using the **transform** pipeline (you just wrote in task #2.1) on both train and test set.

Hint: Preparing train and test sets can be directly achieved by utilizing the class parameters.

Read about CIFAR10 dataset class in PyTorch: https://pytorch.org/vision/0.9/datasets.html#cifar

```
# TODO
train_set = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform_pipeline)
test_set = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transform_pipeline)
```

```
Files already downloaded and verified
Files already downloaded and verified
```

2.3 Use torch.utils.data.random_split() to make a validation set from the training set with 80:20 split. (3pts)

Make sure the training set you'll use after this point excludes the validation set of images

```
# TODO
num_train_samples = int(len(train_set) * 0.8)
num_val_samples = len(train_set) - num_train_samples
```

```
train_set, val_set = data.random_split(train_set, [num_train_samples, num_val_samples])
```

2.4 Prepare three dataloaders for train, validation and test set. Use an appropriate batchsize of your choice. (1+2+2+2 =7pts)

Hints:

1. Remember that choosing a batchsize is always a trade-off between efficiency and generalizability. With large batchsize, your model learns more and better in each forward pass, but each pass will require larger computation. On the other hand, with small batchsize, it might converge quicker, but each forward pass teaches features from a smaller subset, which may not be a good representation of the overall data; leading to jittery convergence.
2. During training, you will use the train and validation set for tracking the loss and avoiding overfitting. The test set will be hold out until you are ready to evaluate a trained model on new data.

Read about pytorch Dataloaders here: https://pytorch.org/tutorials/beginner/basics/data_tutorial.html#preparing-your-data-for-training-with-dataloaders

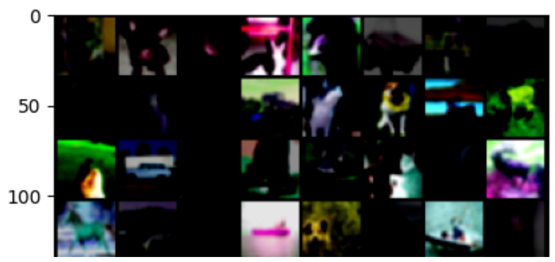
```
# TODO: set a batch size
batch_size = 64

# TODO: write dataloader for train set
train_loader = data.DataLoader(train_set, batch_size=batch_size)

# TODO: write dataloader for test set
test_loader = data.DataLoader(test_set, batch_size=batch_size)

# TODO: write dataloader for validation set
val_loader = data.DataLoader(val_set, batch_size=batch_size)
```

2.5 Load a random batch of images from the training set using the trainloader. Then use *make_grid()* from *torchvision.utils* and *imshow()* from *matplotlib.pyplot* to show the images. Also, print the corresponding true labels for those image samples. (6pts)

Hint: you may need to reshape the *make_grid()* output to comply with the format *imshow()* accepts.

```
import matplotlib.pyplot as plt
from torchvision.utils import make_grid
import numpy as np
# TODO: load a random batch of test set images
data_iterator = iter(train_loader)
images, labels = next(data_iterator)
grid = make_grid(images, nrow=8, padding=2)
grid = grid.numpy().transpose((1,2,0))

# TODO: show the images
plt.imshow(grid)
plt.show()

# TODO: print the ground truth class labels for these images
classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
print('True labels:', ' '.join(classes[labels[j]] for j in range(batch_size)))
```

```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RG
```



3. Model Design (Total 22pts)

3.1 Define a neural network model: (2+7+7 =16pts)

- Name the model class with your first name
- In the following sequential order, the model should consist:

  (1) a 2D convolution layer with 6 filters, dimension of each filter is (5, 5), stride=1, no zero padding

  (2) a Max Pool layer with filter size (2, 2), stride=2

  (3) a 2D convolution layer with 16 filters, dimension of each filter is (5, 5), stride=1, no zero padding

  (4) a 2D Max Pool layer with filter size (2, 2), stride=2

  ~ a flatten layer ~

  (5) a Dense/Fully-connected layer with 120 neurons

  ~ a ReLU activation ~

  ~ a Dropout Layer ~

  (6) a Dense/Fully-connected layer with 80 neurons

  ~ a ReLU activation ~

  (7) a Dense/Fully-connected layer with 10 neurons

Note:

1. Flatten, ReLU and Dropout are not really "layers". They are operations with specific purpose. But in model construction in pytorch, they are abstracted as layers.

   Flatten is used to convert the 4th layer output to a 1D tensor so that it can be passed through the next fully-connected layer. Since each forward pass takes a batch of data, use the *start_dim* parameter of *torch.flatten()* appropriately to keep the batch dimension intact.

   ReLU is an activation that transforms the Dense Layer's linear output to a non-linear "active" output.

   Dropout is a regularization technique. Read more in slides. In this assignment, you can drop neurons with 50% probability.

2. This dataset has 10 classes, hence the final layer consists 10 neurons.
3. The model architecture is similar to the one you saw in in-class Quiz 2, with an extra dense layer in the end.

   Read about building your custom model in pytorch here: https://pytorch.org/tutorials/beginner/introyt/modelsyt_tutorial.html

   The official pytorch documentation on conv, flatten, rely, dense are also resourceful.

```
class Daniel(nn.Module):
    def __init__(self):
        # TODO: Initialize the layers
        super(Daniel, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, kernel_size=5, stride=1 , padding=0)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(6, 16, kernel_size=5, stride=1 , padding=0)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(p=0.5)
        self.fc2 = nn.Linear(120,80)
        self.fc3 = nn.Linear(80, 10)

    def forward(self, x):
        # TODO: Define the dataflow through the layers
        x = self.pool(self.relu(self.conv1(x)))
        x = self.pool(self.relu(self.conv2(x)))
        x = torch.flatten(x, start_dim=1)
        x = self.relu(self.fc1(x))
        x =  self.dropout(x)
        x = self.relu(self.fc2(x))
        x = self.relu(self.fc3(x))
        return x
```

3.2 Create an instance of the model class that you just prepared. (2pts)

```
# TODO:
model = Daniel()
```

3.3 Set up Cross Entropy Loss as the loss function and *Adam* as the optimizer. Use a learning rate of your choice for the optimizer. (4pts)

```
# TODO: Define the loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

4. Training and Validation (Total 50pts)

4.1 Write a training loop to load data, compute model output, compute loss and backpropagating it to update model parameters. (30pts)

The # TODO tags below contain further instructions.

```
# TODO: Define number of epochs
num_epochs = 20

# TODO: Initialize empty lists to store training loss, training accuracy, validation loss, validation accuracy
# You will use these lists to plot the loss history.
train_loss_history = []
train_acc_history = []
val_loss_history = []
val_acc_history = []

# TODO: Loop through the number of epochs
for epoch in range(num_epochs):
    # TODO: set model to train mode
    model.train()
    running_loss = 0.0
    running_corrects = 0

    # TODO: iterate over the training data in batches
    for images, labels in train_loader:

        # TODO: get the image inputs and labels from current batch
        # done at the for loop

        # TODO: set the optimizer gradients to zero to avoid accumulation of gradients
        optimizer.zero_grad()

        # TODO: compute the output of the model
        outputs = model(images)

        # TODO: compute the loss on current batch
        loss = criterion(outputs, labels)

        # TODO: backpropagate the loss
        loss.backward()

        # TODO: update the optimizer parameters
        optimizer.step()

        # TODO: update the train loss and accuracy
        _, preds = torch.max(outputs, 1)
        running_loss += loss.item() * images.size(0)
        running_corrects += torch.sum(preds == labels.data)

    # TODO: compute the average training loss and accuracy and store in respective arrays
    train_loss_history.append(running_loss / len(train_set))
    train_acc_history.append(running_corrects.double() / len(train_set))

    # TODO: set the model to evaluation mode
    model.eval()
    val_running_loss = 0.0
    val_running_corrects = 0

    # TODO: keeping the gradient computation turned off, loop over batches of data from validation set.
    with torch.no_grad():
      for val_images, val_labels in val_loader:
            # TODO: compute output of the model
            val_outputs = model(val_images)

            # TODO: compute the loss
            val_loss = criterion(val_outputs, val_labels)

            # TODO: compute the validation loss and accuracy
            _, val_preds = torch.max(val_outputs, 1)

            val_running_loss += val_loss.item() * val_images.size(0)
            val_running_corrects += torch.sum(val_preds == val_labels.data)

    # TODO: compute the average validation loss and accuracy and store in respective lists
    val_loss_history.append(val_running_loss / len(val_set))
    val_acc_history.append(val_running_corrects.double() / len(val_set))

    # TODO: print the training loss, training accuracy, validation loss and validation accuracy at the end of each epoch
    print(f'Epoch {epoch+1}/{num_epochs}: '
          f'Train Loss: {(running_loss / len(train_set)):.4f}, Train Accuracy: {(running_corrects.double() / len(train_set))}, '
          f'Validation Loss: {(val_running_loss / len(val_set)):.4f}, Validation Accuracy: {(val_running_corrects.double() / len(val_set)):.4

    # TODO: save the model parameters once in every 5 epochs
    if (epoch + 1) % 5 == 0:
      torch.save(model.state_dict(), f'model_epoch_{epoch+1}.pth')
```

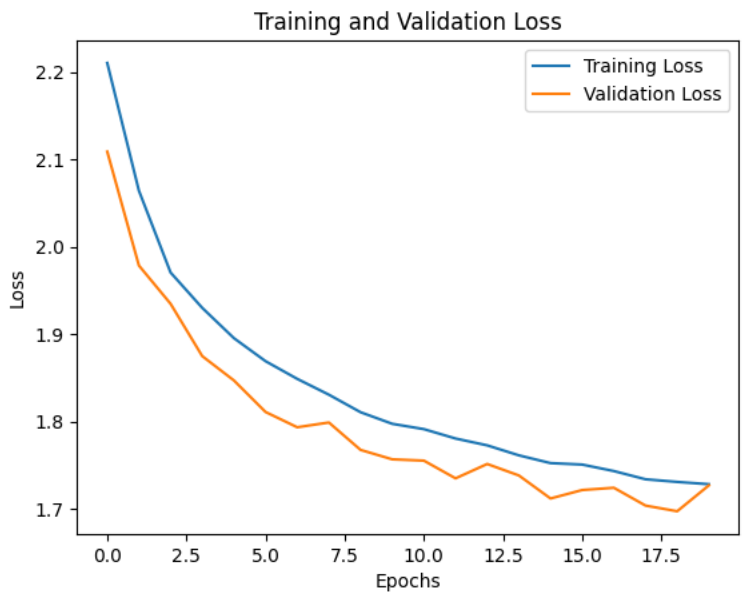
```
Epoch 1/20: Train Loss: 2.2101, Train Accuracy: 0.213775, Validation Loss: 2.1087, Validation Accuracy: 0.2919
Epoch 2/20: Train Loss: 2.0640, Train Accuracy: 0.28715, Validation Loss: 1.9783, Validation Accuracy: 0.3333
Epoch 3/20: Train Loss: 1.9704, Train Accuracy: 0.318875, Validation Loss: 1.9346, Validation Accuracy: 0.3447
Epoch 4/20: Train Loss: 1.9299, Train Accuracy: 0.34085, Validation Loss: 1.8747, Validation Accuracy: 0.3688
Epoch 5/20: Train Loss: 1.8953, Train Accuracy: 0.350425, Validation Loss: 1.8468, Validation Accuracy: 0.3718
Epoch 6/20: Train Loss: 1.8689, Train Accuracy: 0.363, Validation Loss: 1.8107, Validation Accuracy: 0.3891
Epoch 7/20: Train Loss: 1.8487, Train Accuracy: 0.37375, Validation Loss: 1.7933, Validation Accuracy: 0.4024
Epoch 8/20: Train Loss: 1.8305, Train Accuracy: 0.379175, Validation Loss: 1.7987, Validation Accuracy: 0.3988
Epoch 9/20: Train Loss: 1.8105, Train Accuracy: 0.382425, Validation Loss: 1.7674, Validation Accuracy: 0.4055
Epoch 10/20: Train Loss: 1.7972, Train Accuracy: 0.392025, Validation Loss: 1.7566, Validation Accuracy: 0.4081
Epoch 11/20: Train Loss: 1.7911, Train Accuracy: 0.39895, Validation Loss: 1.7551, Validation Accuracy: 0.4090
Epoch 12/20: Train Loss: 1.7804, Train Accuracy: 0.39925, Validation Loss: 1.7348, Validation Accuracy: 0.4135
Epoch 13/20: Train Loss: 1.7726, Train Accuracy: 0.403, Validation Loss: 1.7512, Validation Accuracy: 0.4058
Epoch 14/20: Train Loss: 1.7611, Train Accuracy: 0.408125, Validation Loss: 1.7381, Validation Accuracy: 0.4151
Epoch 15/20: Train Loss: 1.7522, Train Accuracy: 0.41175, Validation Loss: 1.7118, Validation Accuracy: 0.4248
Epoch 16/20: Train Loss: 1.7506, Train Accuracy: 0.410875, Validation Loss: 1.7215, Validation Accuracy: 0.4273
Epoch 17/20: Train Loss: 1.7431, Train Accuracy: 0.4154, Validation Loss: 1.7240, Validation Accuracy: 0.4181
Epoch 18/20: Train Loss: 1.7337, Train Accuracy: 0.4165, Validation Loss: 1.7036, Validation Accuracy: 0.4253
Epoch 19/20: Train Loss: 1.7307, Train Accuracy: 0.420675, Validation Loss: 1.6971, Validation Accuracy: 0.4332
Epoch 20/20: Train Loss: 1.7281, Train Accuracy: 0.41975, Validation Loss: 1.7268, Validation Accuracy: 0.4214
```

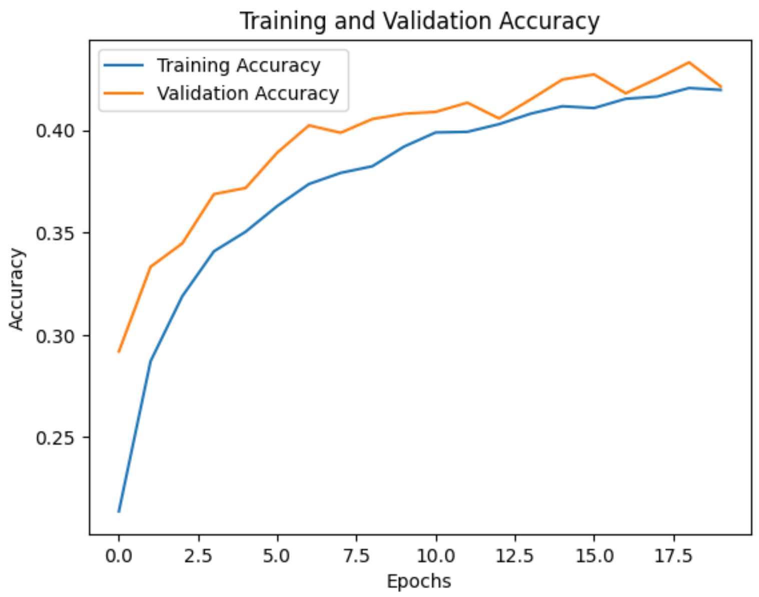
4.2 Plot and compare (5+5 =10pts)

1. training and validation loss over the number of epochs
2. training and validation accuracy over the number of epochs

(Hint: Use plot() from *matplotlib.pyplot*, import it if you haven't already done so.)

```
# TODO: plot the training and validation loss
plt.figure()
plt.plot(range(num_epochs), train_loss_history, label='Training Loss')
plt.plot(range(num_epochs), val_loss_history, label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.title('Training and Validation Loss')
plt.show()
# TODO: plot the training and validation accuracy
plt.figure()
plt.plot(range(num_epochs), train_acc_history, label='Training Accuracy')
plt.plot(range(num_epochs), val_acc_history, label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.title('Training and Validation Accuracy')
plt.show()
```





4.3 Discussion: (2*5 = 10pts)

(1) Does the training loss and accuracy improve as number of epoch increases?

(2) Does the validation loss and accuracy improve as number of epoch increases?

(3) Are there any sign of overfitting in the results? If so, when did it start to occur?

(4) How many epochs did it take for the model to converge to a good solution?

(5) What enhancement can be tried to the architecture to further improve the validation performance?

~ # TODO

(1) Yes, the training loss and accuracy improve as the number of epochs increases.

(2) Yes, the validation loss and accuracy improve as the number of epochs increases.

(3) No, there are no clear indicators of overfitting since the training loss/accuracy are not inversely related to the validation loss/accuracy. They both increase and decrease at relatively the same rates.

(4) The model plateaued/converged around epoch 16, though there were still improvements with epoch 20.

(5) Increasing the depth of the model by adding more layers.

5. Testing on new data (Total 40pts)

5.1 Load the best performing model (one with good validation accuracy and without overfitting) among the ones you saved. (4pts)

```
# TODO: instantiate a model
model = Daniel()

# TODO: load parameters from one of the saved model states
model.load_state_dict(torch.load('model_epoch_15.pth'))

# TODO: set this model to evaluation mode
model.eval()
```

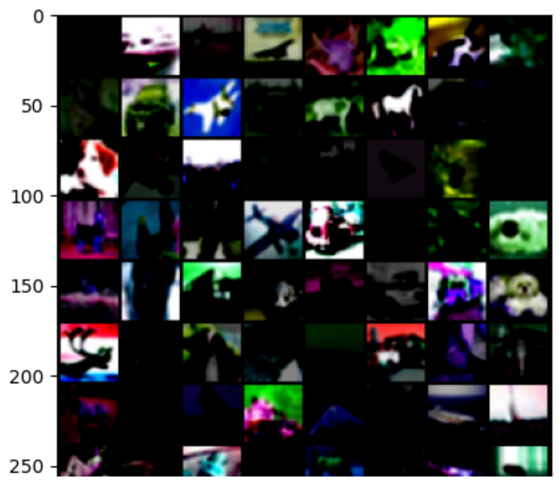
```
Daniel(
  (conv1): Conv2d(3, 6, kernel_size=(5, 5), stride=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=400, out_features=120, bias=True)
  (relu): ReLU()
  (dropout): Dropout(p=0.5, inplace=False)
  (fc2): Linear(in_features=120, out_features=80, bias=True)
  (fc3): Linear(in_features=80, out_features=10, bias=True)
)
```

5.2 Take a random batch of images from test set and show the images. Print the corresponding ground truth class labels. Then compute model output (model selected at previous step) and the predicted labels for the images in this batch. (10pts)

This is similar to task #2.5 with additional task on computing model output and printing predicted labels.

```
# TODO: load a random batch of test set images
data_iterator = iter(test_loader)
images, labels = next(data_iterator)

# TODO: show the images
grid = make_grid(images, nrow=8, padding=2)
grid = grid.numpy().transpose((1,2,0))
plt.imshow(grid)
plt.show()

# TODO: print the ground truth class labels for these images
print([label.item() for label in labels])

# TODO: compute model output
with torch.no_grad():
  outputs = model(images)

# TODO: print the predicted class labels for these images
_, predicted_labels = torch.max(outputs, 1)
print([label.item() for label in predicted_labels])
```

```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RG
```



5.3 Compute the average accuracy on test data using this model. (4+2 =6pts)

Loop over the test set, compute accuracy on each batch, lastly print the average accuracy.

```
# TODO: compute accuracy on each batch of test set
correct = 0
total = 0

# TODO: print the average accuracy
with torch.no_grad():
  for images, labels in test_loader:
    outputs = model(images)
    _, predicted_labels = torch.max(outputs, 1)
    total += labels.size(0)
    correct += (predicted_labels == labels).sum().item()

print(f'Average accuracy on test data: {(100 * correct / total)}%')
```

```
Average accuracy on test data: 41.55%
```

5.4 Compute the average accuracy for each individual class. (8+4 =12pts)

Hint: similar to #5.3. During each loop, log the accuracy for each class separately (a python/numpy dictionary can help). Then print the individual accuracy for the 10 output classes.

```
# TODO: compute per-class accuracy on each batch of test set
class_correct = {i: 0 for i in range(10)}
class_total = {i: 0 for i in range(10)}

with torch.no_grad():
  for images, labels in test_loader:
    outputs = model(images)
    _, predicted_labels = torch.max(outputs, 1)

    correct = (predicted_labels == labels)

    for i in range(labels.size(0)):
      true_label = labels[i].item()
      class_correct[true_label] += correct[i].item()
      class_total[true_label] += 1

print(f'Average accuracy on test data: {(100 * correct / total)}%')

# TODO: print per-class accuracy for 10 output classes
for i in range(10):
    accuracy = 100 * class_correct[i] / class_total[i]
    print(f'Accuracy for class {i} ({classes[i]}): {accuracy}%')
```

```
Average accuracy on test data: tensor([0.0100, 0.0100, 0.0000, 0.0100, 0.0000, 0.0000, 0.0100, 0.0100, 0.0100,
        0.0100, 0.0000, 0.0000, 0.0100, 0.0100, 0.0000, 0.0100])%
Accuracy for class 0 (plane): 45.7%
Accuracy for class 1 (car): 0.0%
Accuracy for class 2 (bird): 37.3%
Accuracy for class 3 (cat): 30.0%
Accuracy for class 4 (deer): 46.4%
Accuracy for class 5 (dog): 53.0%
Accuracy for class 6 (frog): 62.3%
Accuracy for class 7 (horse): 61.7%
Accuracy for class 8 (ship): 0.0%
Accuracy for class 9 (truck): 76.1%
```

5.5 Discussion: (2+2+4 =8pts)

1. Which class of images were detected with highest accuracy?
2. Which class of images were hardest for the model to detect?
3. Explain 1-2 possible reasons why detection of some class can be harder for the same model.

```
~ # TODO

(1) Class 9, truck, had the highest accuracy.

(2) Class 1 and 8, car and boat, had the lowest accuracy.

(3) This could be due to high intra-class variation within the images
same class being very different from each other. Or, inter-class simil
where images from different classes look very similar to each other.
```

~ # TODO

(1) Class 9, truck, had the highest accuracy.

(2) Class 1 and 8, car and boat, had the lowest accuracy.

(3) This could be due to high intra-class variation within the images of the same class being very different from each other. Or, inter-class similarity where images from different classes look very similar to each other.

✓ 13s completed at 9:30 PM