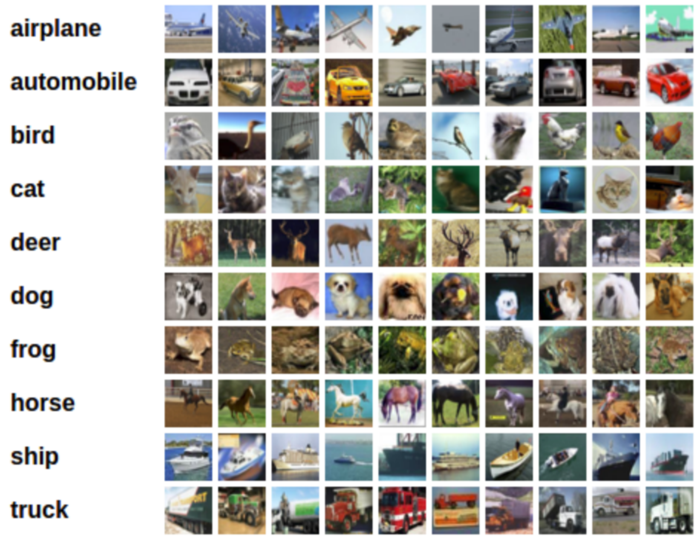
Convolutional Neural Networks - Build Model

In this notebook, we build and train a **CNN** to classify images from the CIFAR-10 database.

- The code provided here are **almost** working. You are required to build up a CNN model and train it.
- Make sure you covered implementations of the **TODO**s in this notebook

The images in this database are small color images that fall into one of ten classes; some example images are pictured below.



Optional: Use CUDA if Available

Since these are color (32x32x3) images, it may prove useful to speed up your training time by using a GPU. CUDA is a parallel computing platform and CUDA Tensors are the same as typical Tensors, but they utilize GPU's for effcient parallel computation.

In [261…

```
import torch
import numpy as np
```

```python
# check if CUDA is available
train_on_gpu = torch.cuda.is_available()

if not train_on_gpu:
    print('CUDA is not available.  Training on CPU ...')
else:
    print('CUDA is available!  Training on GPU ...')
```

```
CUDA is available!  Training on GPU ...
```

Load the Data

Downloading may take a minute. We load in the training and test data, split the training data into a training and validation set, then create DataLoaders for each of these sets of data.

In [262…

```python
from torchvision import datasets
import torchvision.transforms as transforms
from torch.utils.data.sampler import SubsetRandomSampler

# number of subprocesses to use for data loading
num_workers = 0
# how many samples per batch to load
batch_size = 20
# percentage of training set to use as validation
valid_size = 0.2

# convert data to a normalized torch.FloatTensor
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
    ])

# choose the training and test datasets
train_data = datasets.CIFAR10('data', train=True,
                              download=True, transform=transform)
test_data = datasets.CIFAR10('data', train=False,
                             download=True, transform=transform)
```

```python
# obtain training indices that will be used for validation
num_train = len(train_data)
indices = list(range(num_train))
np.random.shuffle(indices)
split = int(np.floor(valid_size * num_train))
train_idx, valid_idx = indices[split:], indices[:split]

# define samplers for obtaining training and validation batches
train_sampler = SubsetRandomSampler(train_idx)
valid_sampler = SubsetRandomSampler(valid_idx)

# prepare data loaders (combine dataset and sampler)
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,
    sampler=train_sampler, num_workers=num_workers)
valid_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,
    sampler=valid_sampler, num_workers=num_workers)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size,
    num_workers=num_workers)

# specify the image classes
classes = ['airplane', 'automobile', 'bird', 'cat', 'deer',
           'dog', 'frog', 'horse', 'ship', 'truck']
```

Visualize a Batch of Training Data

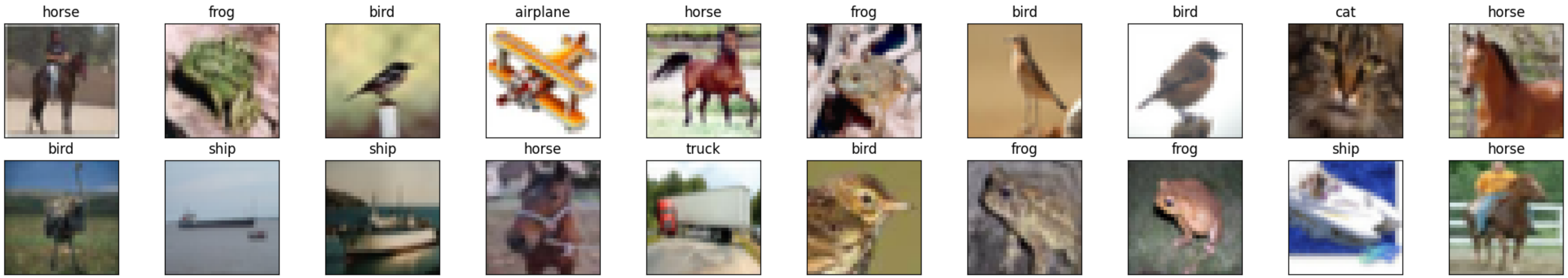
In [263…

```python
import matplotlib.pyplot as plt
%matplotlib inline

# helper function to un-normalize and display an image
def imshow(img):
    img = img / 2 + 0.5  # unnormalize
    plt.imshow(np.transpose(img, (1, 2, 0)))  # convert from Tensor image
```

In [264…

```python
# obtain one batch of training images
dataiter = iter(train_loader)
#images, labels = dataiter.next() #python, torchvision version match issue
images, labels = next(dataiter)
images = images.numpy() # convert images to numpy for display
```

```
# plot the images in the batch, along with the corresponding labels
fig = plt.figure(figsize=(25, 4))
# display 20 images
for idx in np.arange(20):
    ax = fig.add_subplot(2, int(20/2), idx+1, xticks=[], yticks=[])
    imshow(images[idx])
    ax.set_title(classes[labels[idx]])
```



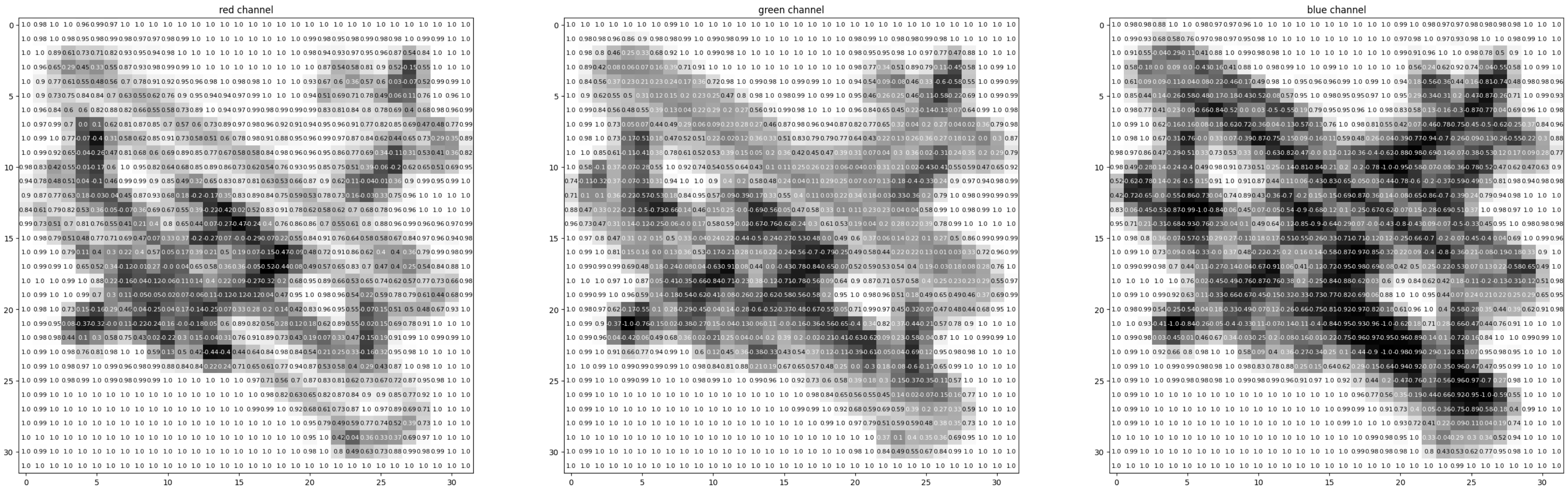
View an Image in More Detail

Here, we look at the normalized red, green, and blue (RGB) color channels as three separate, grayscale intensity images.

In [265…

```
rgb_img = np.squeeze(images[3])
channels = ['red channel', 'green channel', 'blue channel']

fig = plt.figure(figsize = (36, 36))
for idx in np.arange(rgb_img.shape[0]):
    ax = fig.add_subplot(1, 3, idx + 1)
    img = rgb_img[idx]
    ax.imshow(img, cmap='gray')
    ax.set_title(channels[idx])
    width, height = img.shape
    thresh = img.max()/2.5
    for x in range(width):
        for y in range(height):
            val = round(img[x][y],2) if img[x][y] !=0 else 0
            ax.annotate(str(val), xy=(y,x),
                    horizontalalignment='center',
```

```
                    verticalalignment='center', size=8,
                    color='white' if img[x][y]<thresh else 'black')
```



TODO: Define the Network Architecture

Build up your own Convolutional Neural Network using Pytorch API:

- nn.Conv2d(): for convolution
- nn.MaxPool2d(): for maxpooling (spatial resolution reduction)
- nn.Linear(): for last 1 or 2 layers of fully connected layer before the output layer.
- nn.Dropout(): optional, dropout can be used to avoid overfitting.
- F.relu(): Use ReLU as the activation function for all the hidden layers

The following is a skeleton example that's not completely working.

In [266…

```
import torch.nn as nn
import torch.nn.functional as F

# define the CNN architecture
```

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()

        self.conv1 = nn.Conv2d(3, 32, 3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, 3, padding=1)
        self.conv3 = nn.Conv2d(64, 128, 3, padding=1)

        pass

        # max pooling layer
        self.pool = nn.MaxPool2d(2, 2)

        pass

        # Dropout layer
        self.dropout = nn.Dropout(0.25)
        #self.dropout = nn.Dropout(0.5)
            #gets a 1% increase but not really worth it

        # Fully connected layers
        self.fc1 = nn.Linear(128 * 4 * 4, 512)
        self.fc2 = nn.Linear(512, 10)

    def forward(self, x):
        #convolution & pooling layers
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))

        # Flatten
        x = x.view(-1, 128 * 4 * 4)

        # Dropout
        x = self.dropout(x)

        # Fully connected layers
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return x
```

```
# create a complete CNN
model = Net()
print(model)

# move tensors to GPU if CUDA is available
if train_on_gpu:
    model.cuda()
```

```
Net(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (dropout): Dropout(p=0.25, inplace=False)
  (fc1): Linear(in_features=2048, out_features=512, bias=True)
  (fc2): Linear(in_features=512, out_features=10, bias=True)
)
```

Specify Loss Function and Optimizer

Decide on a loss and optimization function that is best suited for this classification task. The linked code examples from above, may be a good starting point; this PyTorch classification example Pay close attention to the value for **learning rate** as this value determines how your model converges to a small error.

The following is working code, but you can make your own adjustments.

TODO: try to compare with ADAM optimizer

In [267…

```
import torch.optim as optim

# specify loss function (categorical cross-entropy)
criterion = nn.CrossEntropyLoss()

# specify optimizer
#optimizer = optim.SGD(model.parameters(), lr=0.01)

# TODO, compare with optimizer ADAM
```

```
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

Train the Network

Remember to look at how the training and validation loss decreases over time; if the validation loss ever increases it indicates possible overfitting.

The following is working code, but you are encouraged to make your own adjustments and enhance the implementation.

In [268…

```
# number of epochs to train the model, you decide the number
n_epochs = 5
#n_epochs = 20

valid_loss_min = np.inf # track change in validation loss

for epoch in range(1, n_epochs+1):

    # keep track of training and validation loss
    train_loss = 0.0
    valid_loss = 0.0

    ###################
    # train the model #
    ###################
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        # move tensors to GPU if CUDA is available
        if train_on_gpu:
            data, target = data.cuda(), target.cuda()
        # clear the gradients of all optimized variables
        optimizer.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the batch loss
        loss = criterion(output, target)
        # backward pass: compute gradient of the loss with respect to model parameters
        loss.backward()
```

```
        # perform a single optimization step (parameter update)
        optimizer.step()
        # update training loss
        train_loss += loss.item()*data.size(0)

    ######################
    # validate the model #
    ######################
    model.eval()
    for batch_idx, (data, target) in enumerate(valid_loader):
        # move tensors to GPU if CUDA is available
        if train_on_gpu:
            data, target = data.cuda(), target.cuda()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the batch loss
        loss = criterion(output, target)
        # update average validation loss
        valid_loss += loss.item()*data.size(0)

    # calculate average losses
    train_loss = train_loss/len(train_loader.sampler)
    valid_loss = valid_loss/len(valid_loader.sampler)

    # print training/validation statistics
    print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
        epoch, train_loss, valid_loss))

    # save model if validation loss has decreased
    if valid_loss <= valid_loss_min:
        print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'.format(
        valid_loss_min,
        valid_loss))
        torch.save(model.state_dict(), 'model_trained.pt')
        valid_loss_min = valid_loss
```

```
Epoch: 1        Training Loss: 1.399143         Validation Loss: 1.178914
Validation loss decreased (inf --> 1.178914).  Saving model ...
Epoch: 2        Training Loss: 0.998880         Validation Loss: 0.881407
Validation loss decreased (1.178914 --> 0.881407).  Saving model ...
Epoch: 3        Training Loss: 0.837649         Validation Loss: 0.842238
Validation loss decreased (0.881407 --> 0.842238).  Saving model ...
Epoch: 4        Training Loss: 0.739765         Validation Loss: 0.783849
Validation loss decreased (0.842238 --> 0.783849).  Saving model ...
Epoch: 5        Training Loss: 0.666871         Validation Loss: 0.742203
Validation loss decreased (0.783849 --> 0.742203).  Saving model ...
```

Load the Model with the Lowest Validation Loss

This is the model we will use for testing, which is the model we saved in the last step

In [269…

```
model.load_state_dict(torch.load('model_trained.pt'))
```

Out[269…

```
<All keys matched successfully>
```

In [270…

```
#total parameters of the model
total_params = sum(p.numel() for p in model.parameters())
print(f"Saved model Total parameters: {total_params:,}")
```

```
Saved model Total parameters: 1,147,466
```

Test the Trained Network

Test your trained model on previously unseen data! Remember we have downloaded `train_data` and `test_data`. We will use `test_data` through `test_loader`.

A "good" result will be a CNN that gets around 70% (or more, try your best!) accuracy on these test images.

The following is working code, but you are encouraged to make your own adjustments and enhance the implementation.

In [271…

```
# track test loss
test_loss = 0.0
class_correct = list(0. for i in range(10))
```

```
class_total = list(0. for i in range(10))

model.eval()
# iterate over test data
for batch_idx, (data, target) in enumerate(test_loader):
    # move tensors to GPU if CUDA is available
    if train_on_gpu:
        data, target = data.cuda(), target.cuda()
    # forward pass: compute predicted outputs by passing inputs to the model
    output = model(data)
    # calculate the batch loss
    loss = criterion(output, target)
    # update test loss
    test_loss += loss.item()*data.size(0)
    # convert output probabilities to predicted class
    _, pred = torch.max(output, 1)
    # compare predictions to true label
    correct_tensor = pred.eq(target.data.view_as(pred))
    correct = np.squeeze(correct_tensor.numpy()) if not train_on_gpu else np.squeeze(correct_tensor.cpu().numpy())
    # calculate test accuracy for each object class
    for i in range(batch_size):
        label = target.data[i]
        class_correct[label] += correct[i].item()
        class_total[label] += 1

# average test loss
test_loss = test_loss/len(test_loader.dataset)
print('Test Loss: {:.6f}\n'.format(test_loss))

for i in range(10):
    if class_total[i] > 0:
        print('Test Accuracy of %5s: %2d%% (%2d/%2d)' % (
            classes[i], 100 * class_correct[i] / class_total[i],
            np.sum(class_correct[i]), np.sum(class_total[i])))
    else:
        print('Test Accuracy of %5s: N/A (no training examples)' % (classes[i]))

print('\nTest Accuracy (Overall): %2d%% (%2d/%2d)' % (
    100. * np.sum(class_correct) / np.sum(class_total),
    np.sum(class_correct), np.sum(class_total)))
```

```
Test Loss: 0.742453

Test Accuracy of airplane: 62% (625/1000)
Test Accuracy of automobile: 77% (777/1000)
Test Accuracy of  bird: 60% (609/1000)
Test Accuracy of   cat: 61% (611/1000)
Test Accuracy of  deer: 78% (780/1000)
Test Accuracy of   dog: 63% (631/1000)
Test Accuracy of  frog: 83% (835/1000)
Test Accuracy of horse: 80% (805/1000)
Test Accuracy of  ship: 91% (917/1000)
Test Accuracy of truck: 86% (861/1000)

Test Accuracy (Overall): 74% (7451/10000)
```

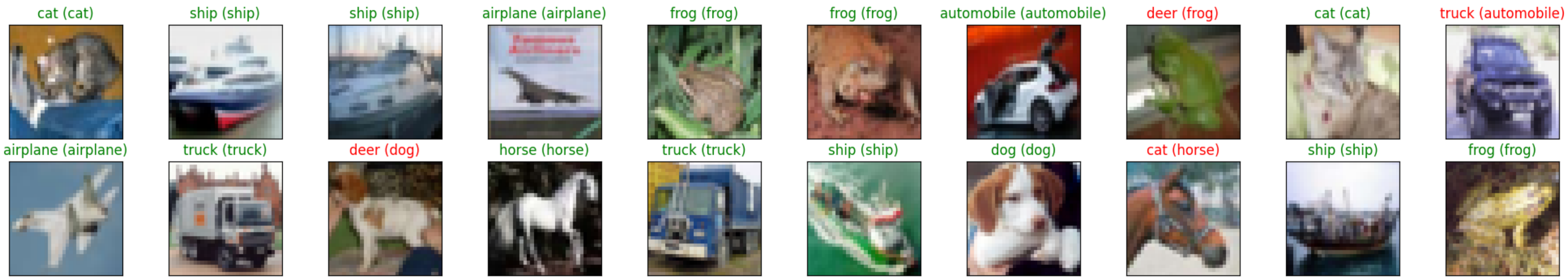
Visualize Sample Test Results

The following is working code, but you are encouraged to make your own adjustments and enhance the visualization.

In [272…

```
# 1 batch of test images
dataiter = iter(test_loader)
images, labels = next(dataiter)

# move model inputs to cuda, if GPU available
if train_on_gpu:
    images = images.cuda()
    labels = labels.cuda()

# get sample outputs
output = model(images)

# convert output probabilities to predicted class
_, preds_tensor = torch.max(output, 1)
preds = preds_tensor.cpu().numpy()
labels = labels.cpu().numpy()

# define imshow to handle GPU tensors
def imshow(img):
    img = img.cpu()
    img = img / 2 + 0.5
```

```python
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))

# plot images in the batch, and labels
fig = plt.figure(figsize=(25, 4))
for idx in np.arange(20):
    ax = fig.add_subplot(2, 10, idx+1, xticks=[], yticks=[])
    imshow(images[idx])
    ax.set_title(
        "{} ({})".format(classes[preds[idx]], classes[labels[idx]]),
        color=("green" if preds[idx] == labels[idx] else "red")
    )
```



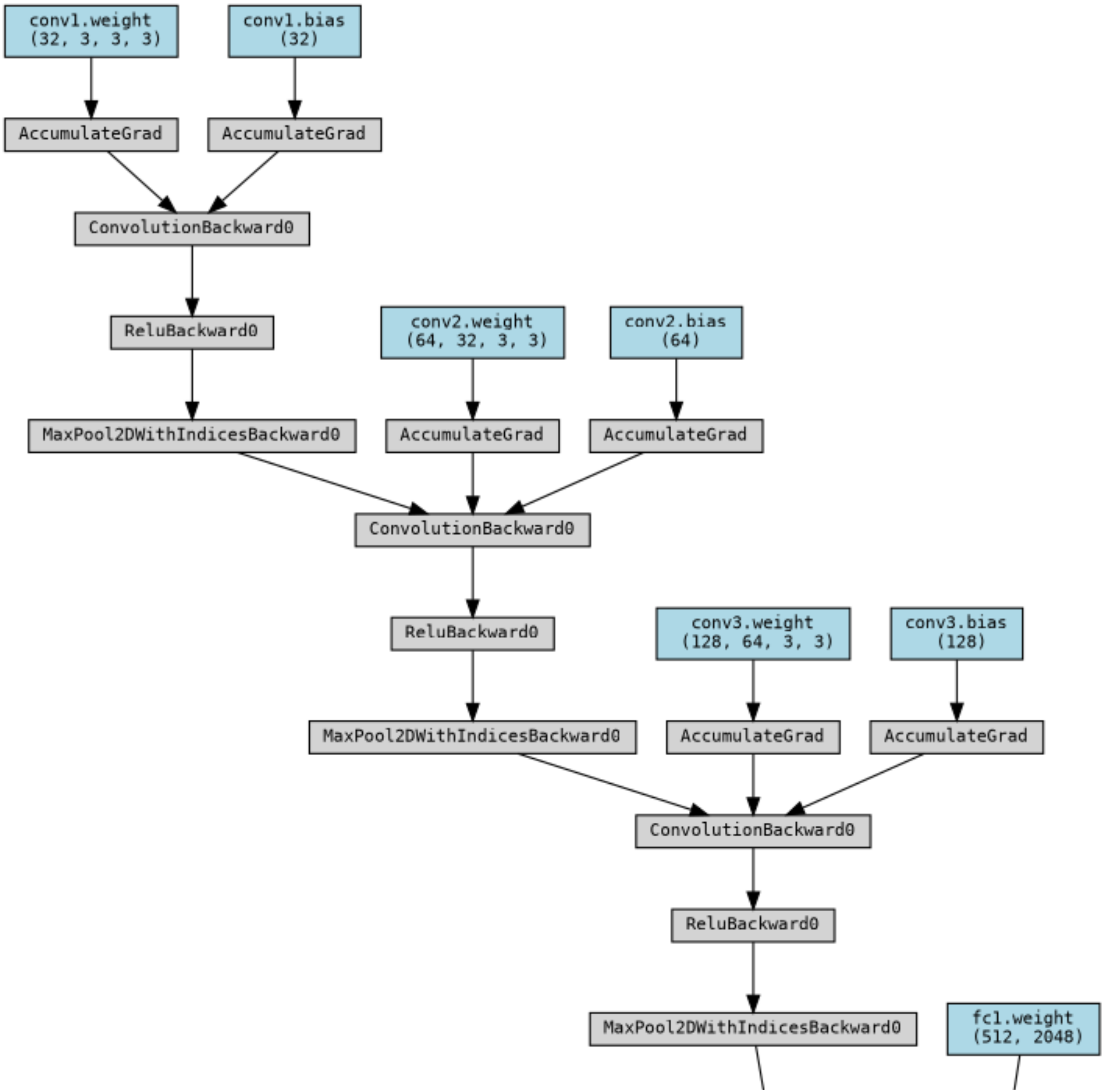
In [273…

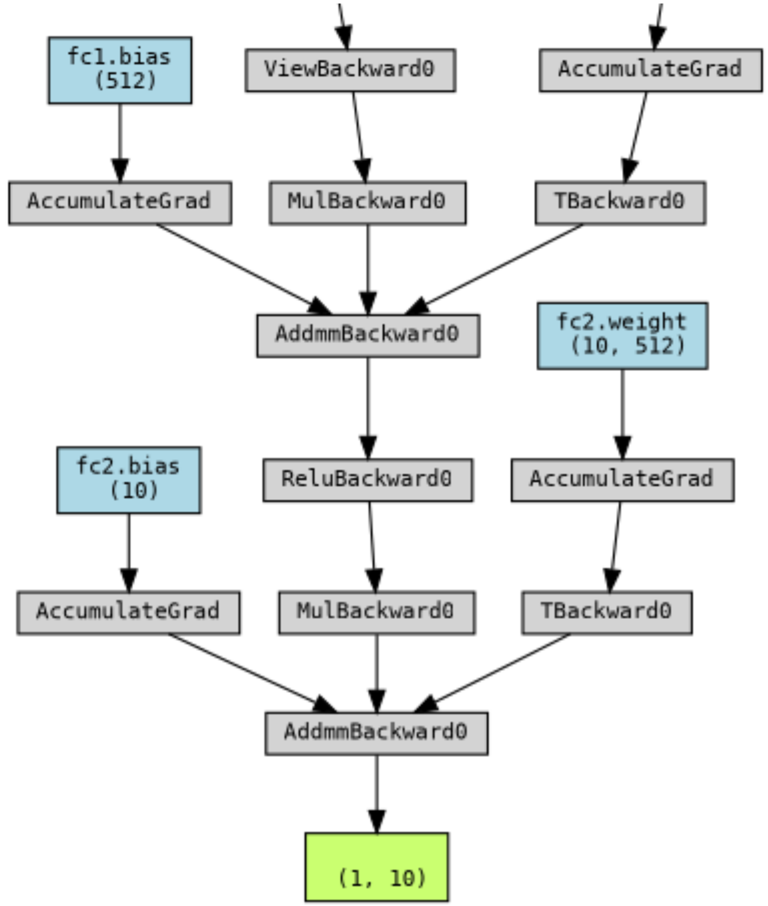
```python
#uncomment line to install torchviz for model visualization
#!pip install torchviz
```

In [274…

```python
# model visualization w/ torchviz
import torch
from torchviz import make_dot
from IPython.display import Image, display

# model
model = Net()

# dummy tensor
x = torch.randn(1, 3, 32, 32)
y = model(x)

# make graph
dot = make_dot(y, params=dict(model.named_parameters()))
```

```python
# display image
dot.format = "png"
dot.render("cnn_graph", cleanup=False)
display(Image(filename="cnn_graph.png"))
```


conv1.weight
(32, 3, 3, 3)
conv1.bias
(32)
AccumulateGrad
AccumulateGrad
ConvolutionBackward0
ReluBackward0
conv2.weight
(64, 32, 3, 3)
conv2.bias
(64)
MaxPool2DWithIndicesBackward0
AccumulateGrad
AccumulateGrad
ConvolutionBackward0
ReluBackward0
conv3.weight
(128, 64, 3, 3)
conv3.bias
(128)
MaxPool2DWithIndicesBackward0
AccumulateGrad
AccumulateGrad
ConvolutionBackward0
ReluBackward0
MaxPool2DWithIndicesBackward0
fc1.weight
(512, 2048)


fc1.bias
(512)
ViewBackward0
AccumulateGrad
AccumulateGrad
MulBackward0
TBackward0
AddmmBackward0
fc2.weight
(10, 512)
fc2.bias
(10)
ReluBackward0
AccumulateGrad
AccumulateGrad
MulBackward0
TBackward0
AddmmBackward0
(1, 10)