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.

```
import torch
import numpy as np

# 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.

```
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)

# 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']
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to data/cifar-10-python.tar.gz
100%|██████████| 170M/170M [00:04<00:00, 36.5MB/s]
Extracting data/cifar-10-python.tar.gz to data
Files already downloaded and verified
```

Visualize a Batch of Training Data

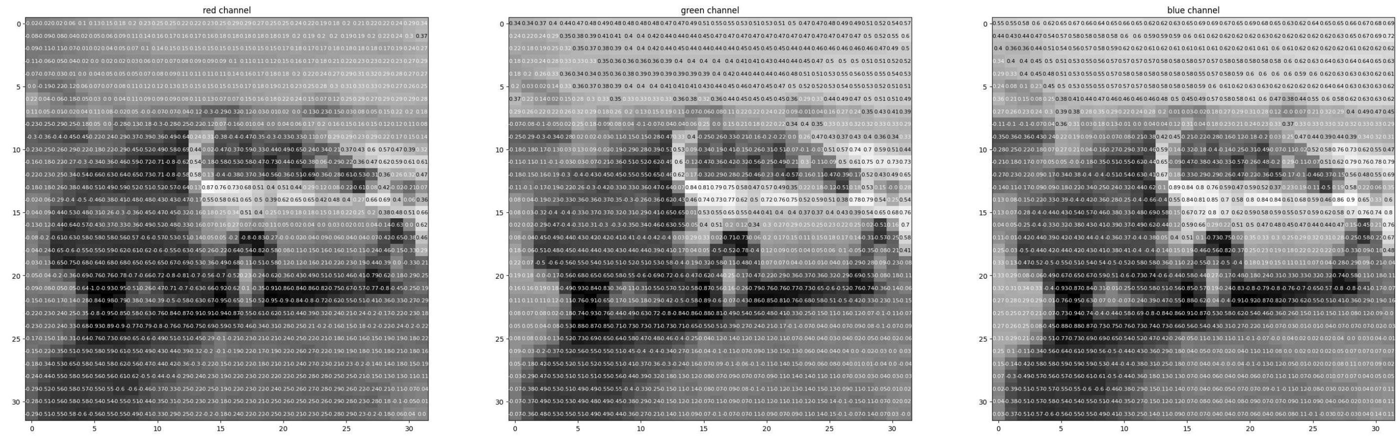
```
import matplotlib.pyplot as plt
%matplotlib inline

# 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


# 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.

```
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.

```
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__()

        # Convolutional Layers (Extract features from the input image)
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3, stride=1, padding=1)  # (32x32x3) -> (32x32x32)
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, stride=1, padding=1)  # (32x32x32) -> (32x32x64)
        self.conv3 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, stride=1, padding=1)  # (32x32x64) -> (32x32x128)

        # Max Pooling Layer (Reduce spatial dimensions)
        self.pool = nn.MaxPool2d(2, 2)  # Halves the spatial size

        # Fully Connected Layers
        self.fc1 = nn.Linear(128 * 4 * 4, 512)  # Flattened size after pooling
        self.fc2 = nn.Linear(512, 10)  # Output layer (for 10 classes)

        # Dropout Layer (To prevent overfitting)
        self.dropout = nn.Dropout(0.25)

    def forward(self, x):
        # Convolutional + ReLU + Pooling
        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 before fully connected layers
        x = x.view(-1, 128 * 4 * 4)
```

```
        # Fully connected layers with ReLU activation
        x = F.relu(self.fc1(x))
        x = self.dropout(x)  # Dropout for regularization
        x = self.fc2(x)  # Output layer

        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)
  (fc1): Linear(in_features=2048, out_features=512, bias=True)
  (fc2): Linear(in_features=512, out_features=10, bias=True)
  (dropout): Dropout(p=0.25, inplace=False)
)
```

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

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

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

# specify optimizer
optimizer = optim.Adam(model.parameters(), lr=0.001)
import torch.optim as optim

# Specify loss function (Categorical Cross-Entropy)
criterion = nn.CrossEntropyLoss()
```

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.

```
# number of epochs to train the model, you decide the number
n_epochs = 10

valid_loss_min = np.inf # track change in validation loss

# Lists to store losses for plotting
train_losses = []
valid_losses = []

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_10ep_adam.pt')
        valid_loss_min = valid_loss
    train_losses.append(train_loss)
    valid_losses.append(valid_loss)
```

```
Epoch: 1        Training Loss: 1.362433         Validation Loss: 1.017644
Validation loss decreased (inf --> 1.017644).  Saving model ...
Epoch: 2        Training Loss: 0.939519         Validation Loss: 0.888765
Validation loss decreased (1.017644 --> 0.888765).  Saving model ...
Epoch: 3        Training Loss: 0.761178         Validation Loss: 0.822422
Validation loss decreased (0.888765 --> 0.822422).  Saving model ...
Epoch: 4        Training Loss: 0.641792         Validation Loss: 0.774537
Validation loss decreased (0.822422 --> 0.774537).  Saving model ...
Epoch: 5        Training Loss: 0.536255         Validation Loss: 0.827330
Epoch: 6        Training Loss: 0.451659         Validation Loss: 0.853996
Epoch: 7        Training Loss: 0.380369         Validation Loss: 0.928480
Epoch: 8        Training Loss: 0.328165         Validation Loss: 0.938229
Epoch: 9        Training Loss: 0.280818         Validation Loss: 0.937594
Epoch: 10       Training Loss: 0.242974         Validation Loss: 1.078554
```

```
train_losses
```

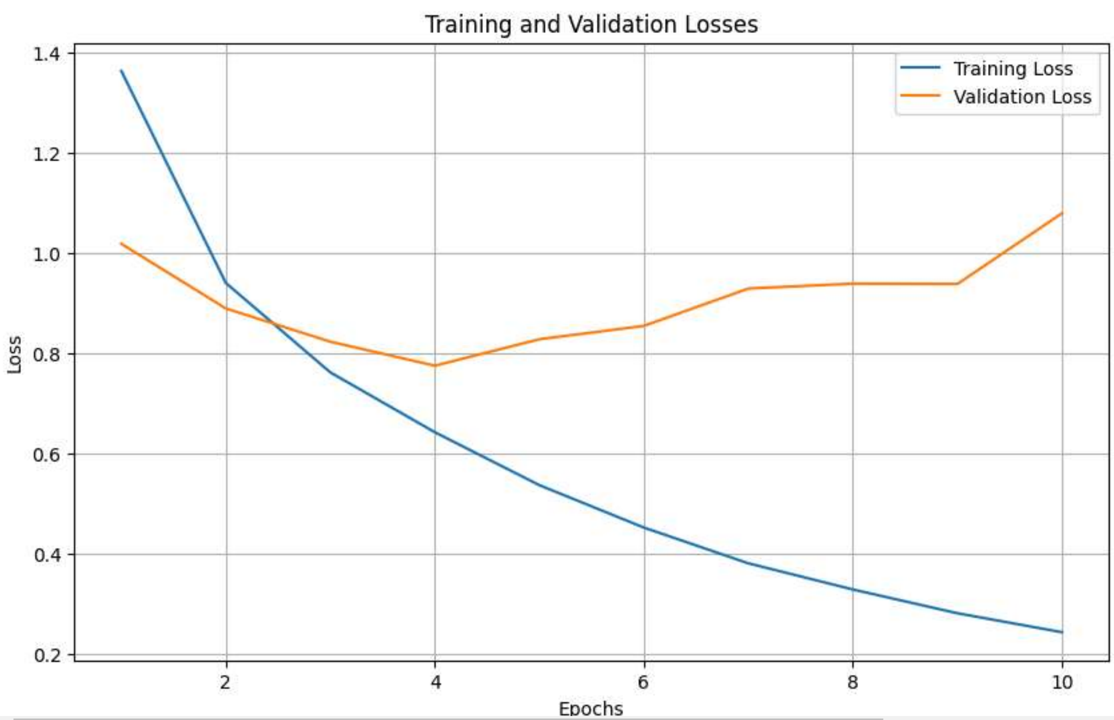
```
[1.3624334735274315,
 0.9395193465203047,
 0.7611784198358655,
 0.6417920807078481,
```

```
 0.5362550069727003,
 0.4516589296441525,
 0.380368607936427,
 0.32816509737446903,
 0.2808178300103173,
 0.242974249629071]
```

```
valid_losses
```

```
[1.0176435064077378,
 0.888764798283577,
 0.8224223788380622,
 0.7745373221635818,
 0.8273299932032824,
 0.8539961712062358,
 0.9284799632132054,
 0.9382290343046188,
 0.9375938406847417,
 1.078553899526596]
```

```
# Plot the training and validation losses
import matplotlib.pyplot as plt
iepochs = len(valid_losses)
plt.figure(figsize=(10, 6))
plt.plot(range(1, iepochs + 1), train_losses, label='Training Loss')
plt.plot(range(1, iepochs + 1), valid_losses, label='Validation Loss')

plt.title('Training and Validation Losses')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
plt.show()
```



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

```
model.load_state_dict(torch.load('model_trained_10ep_adam.pt'))
```

```
<ipython-input-12-19a4568ebf93>:1: FutureWarning: You are using `torch.load` with `weights_only=False` (the current default value), whic
  model.load_state_dict(torch.load('model_trained_10ep_adam.pt'))
<All keys matched successfully>
```

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.

```
# 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)))
```

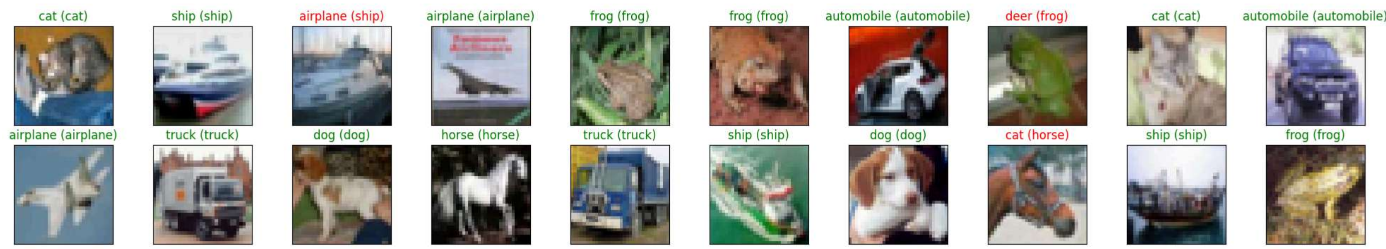
```
Test Loss: 0.782643

Test Accuracy of airplane: 81% (817/1000)
Test Accuracy of automobile: 86% (860/1000)
Test Accuracy of  bird: 70% (706/1000)
Test Accuracy of   cat: 54% (542/1000)
Test Accuracy of  deer: 69% (694/1000)
Test Accuracy of   dog: 61% (610/1000)
Test Accuracy of  frog: 73% (733/1000)
Test Accuracy of horse: 66% (667/1000)
Test Accuracy of  ship: 86% (868/1000)
Test Accuracy of truck: 79% (790/1000)

Test Accuracy (Overall): 72% (7287/10000)
```

Visualize Sample Test Results

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

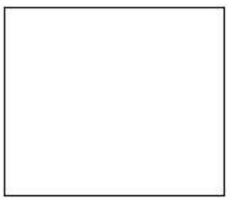
```
# obtain one batch of test images
dataiter = iter(test_loader)
```

```
images, labels = next(dataiter)

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

# get sample outputs
output = model(images)

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

# move the predictions to CPU if they're on the GPU and then convert to numpy
preds = np.squeeze(preds_tensor.cpu().numpy()) if not train_on_gpu else np.squeeze(preds_tensor.cpu().numpy())

# plot the images in the batch, along with predicted and true labels
fig = plt.figure(figsize=(25, 4))
for idx in np.arange(20):
    ax = fig.add_subplot(2, int(20/2), idx+1, xticks=[], yticks=[])
    imshow(images[idx].cpu())  # move the image to CPU for plotting if necessary
    ax.set_title("{} ({})".format(classes[preds[idx]], classes[labels[idx]]),
                 color=("green" if preds[idx] == labels[idx].item() else "red"))
```




```
# obtain one batch of test images
dataiter = iter(test_loader)
images, labels = next(dataiter)
images.numpy()

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

# get sample outputs
output = model(images)
# convert output probabilities to predicted class
_, preds_tensor = torch.max(output, 1)
preds = np.squeeze(preds_tensor.numpy()) if not train_on_gpu else np.squeeze(preds_tensor.cpu().numpy())

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

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/usr/local/lib/python3.11/dist-packages/numpy/core/fromnumeric.py in _wrapfunc(obj, method, *args, **kwds)
     58     try:
---> 59         return bound(*args, **kwds)
     60     except TypeError:

TypeError: transpose() received an invalid combination of arguments - got (tuple), but expected one of:
 * (int dim0, int dim1)
 * (name dim0, name dim1)


During handling of the above exception, another exception occurred:

TypeError                                 Traceback (most recent call last)
                                  5 frames
/usr/local/lib/python3.11/dist-packages/torch/_tensor.py in __array__(self, dtype)
   1147             return handle_torch_function(Tensor.__array__, (self,), self, dtype=dtype)
   1148         if dtype is None:
-> 1149             return self.numpy()
   1150         else:
   1151             return self.numpy().astype(dtype, copy=False)

TypeError: can't convert cuda:0 device type tensor to numpy. Use Tensor.cpu() to copy the tensor to host memory first.
```



Next steps: Explain error

Start coding or generate with AI.