Image Classification using Convolutional Neural Networks in PyTorch

This tutorial covers the following topics:

- Downloading an image dataset from web URL
- Understanding convolution and pooling layers
- Creating a convolutional neural network (CNN) using PyTorch
- Training a CNN from scratch and monitoring performance
- Underfitting, overfitting and how to overcome them

How to run the code

Option 1: Running using free online resources (1-click, recommended)

The easiest way to start executing the code is to click the **Run** button at the top of this page and select **Run on Colab**. Google Colab is a free online platform for running Jupyter notebooks using Google's cloud infrastructure. You can also select "Run on Binder" or "Run on Kaggle" if you face issues running the notebook on Google Colab.

Option 2: Running on your computer locally

To run the code on your computer locally, you'll need to set up Python, download the notebook and install the required libraries. We recommend using the Conda distribution of Python. Click the **Run** button at the top of this page, select the **Run Locally** option, and follow the instructions.

Using a GPU for faster training

You can use a Graphics Processing Unit (GPU) to train your models faster if your execution platform is connected to a GPU manufactured by NVIDIA. Follow these instructions to use a GPU on the platform of your choice:

- *Google Colab*: Use the menu option "Runtime > Change Runtime Type" and select "GPU" from the "Hardware Accelerator" dropdown.
- *Kaggle*: In the "Settings" section of the sidebar, select "GPU" from the "Accelerator" dropdown. Use the button on the top-right to open the sidebar.
- *Binder*: Notebooks running on Binder cannot use a GPU, as the machines powering Binder aren't connected to any GPUs.
- *Linux*: If your laptop/desktop has an NVIDIA GPU (graphics card), make sure you have installed the NVIDIA CUDA drivers.
- *Windows*: If your laptop/desktop has an NVIDIA GPU (graphics card), make sure you have installed the NVIDIA CUDA drivers.
- *macOS*: macOS is not compatible with NVIDIA GPUs

If you do not have access to a GPU or aren't sure what it is, don't worry, you can execute all the code in this tutorial just fine without a GPU.

Exploring the CIFAR10 Dataset

For this tutorial, we'll use the CIFAR10 dataset, which consists of 60000 32x32 px colour images in 10 classes. Here are some sample images from the dataset:

```
import os
import torch
import torchvision
import tarfile
from torchvision.datasets.utils import download_url
from torch.utils.data import random_split
```

We'll download the images in PNG format from this page, using some helper functions from the `torchvision` and `tarfile` packages.

```
# Dowload the dataset
dataset_url = "https://s3.amazonaws.com/fast-ai-imageclas/cifar10.tgz"
download_url(dataset_url, '../../data/downloads/')
```

```
Using downloaded and verified file: ../../data/downloads/cifar10.tgz
```

```
# Extract from archive
if not os.path.exists('../../data/downloads/cifar10'):
    with tarfile.open('../../data/downloads/cifar10.tgz', 'r:gz') as
tar:
        tar.extractall(path='../../data/downloads/')
```

The dataset is extracted to the directory `data/cifar10`. It contains 2 folders `train` and `test`, containing the training set (50000 images) and test set (10000 images) respectively. Each of them contains 10 folders, one for each class of images. Let's verify this using `os.listdir`.

```
data_dir = '../../data/downloads/cifar10/'

print(os.listdir(data_dir))
classes = os.listdir(data_dir + "/train")
print(classes)
```

```
['test', 'train']
['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog',
'horse', 'ship', 'truck']
```

Let's look inside a couple of folders, one from the training set and another from the test set. As an exercise, you can verify that that there are an equal number of images for each class, 5000 in the training set and 1000 in the test set.

```
airplane_files = os.listdir(data_dir + "/train/airplane")
print('No. of training examples for airplanes:', len(airplane_files))
print(airplane_files[:5])
```

```
No. of training examples for airplanes: 5000
['0001.png', '0002.png', '0003.png', '0004.png', '0005.png']
```

```
ship_test_files = os.listdir(data_dir + "/test/ship")
print("No. of test examples for ship:", len(ship_test_files))
print(ship_test_files[:5])
```

```
No. of test examples for ship: 1000
['0001.png', '0002.png', '0003.png', '0004.png', '0005.png']
```

The above directory structure (one folder per class) is used by many computer vision datasets, and most deep learning libraries provide utilites for working with such datasets. We can use the `ImageFolder` class from `torchvision` to load the data as PyTorch tensors.

```
from torchvision.datasets import ImageFolder
#from torchvision.transforms import ToTensor
import torchvision.transforms as tt

#dataset = ImageFolder(data_dir+'/train', transform =
tt.Compose([tt.ToTensor(), tt.RandomCrop(16),
tt.RandomRotation(180)]))
#dataset = ImageFolder(data_dir+'/train', transform = tt.ToTensor())
dataset = ImageFolder(data_dir+'/train', transform =
tt.Compose([tt.RandAugment(), tt.ToTensor()]))
#RandAugment
#Augmix
```

Let's look at a sample element from the training dataset. Each element is a tuple, containing a image tensor and a label. Since the data consists of 32x32 px color images with 3 channels (RGB), each image tensor has the shape `(3, 32, 32)`.

```
img, label = dataset[0]
print(img.shape, label)
img
```

```
torch.Size([3, 32, 32]) 0
```

```
tensor([[[0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
         ...,
         [0.7569, 0.6549, 0.5216,  ..., 0.8314, 0.9843, 0.9686],
         [0.7294, 0.6235, 0.4824,  ..., 0.9373, 0.9725, 0.9686],
         [0.7412, 0.6235, 0.4667,  ..., 0.9647, 0.9647, 0.9647]],

        [[0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
         ...,
         [0.7647, 0.6627, 0.5294,  ..., 0.8235, 0.9725, 0.9608],
         [0.7373, 0.6314, 0.4902,  ..., 0.9255, 0.9608, 0.9686],
         [0.7490, 0.6353, 0.4745,  ..., 0.9529, 0.9529, 0.9569]],
        [[0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000],
         ...,
         [0.7725, 0.6706, 0.5373,  ..., 0.8431, 0.9843, 0.9725],
         [0.7451, 0.6392, 0.4980,  ..., 0.9451, 0.9765, 0.9765],
         [0.7569, 0.6392, 0.4824,  ..., 0.9686, 0.9725, 0.9725]]])
```

The list of classes is stored in the `.classes` property of the dataset. The numeric label for each element corresponds to index of the element's label in the list of classes.

```
print(dataset.classes)
```

```
['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog',
'horse', 'ship', 'truck']
```

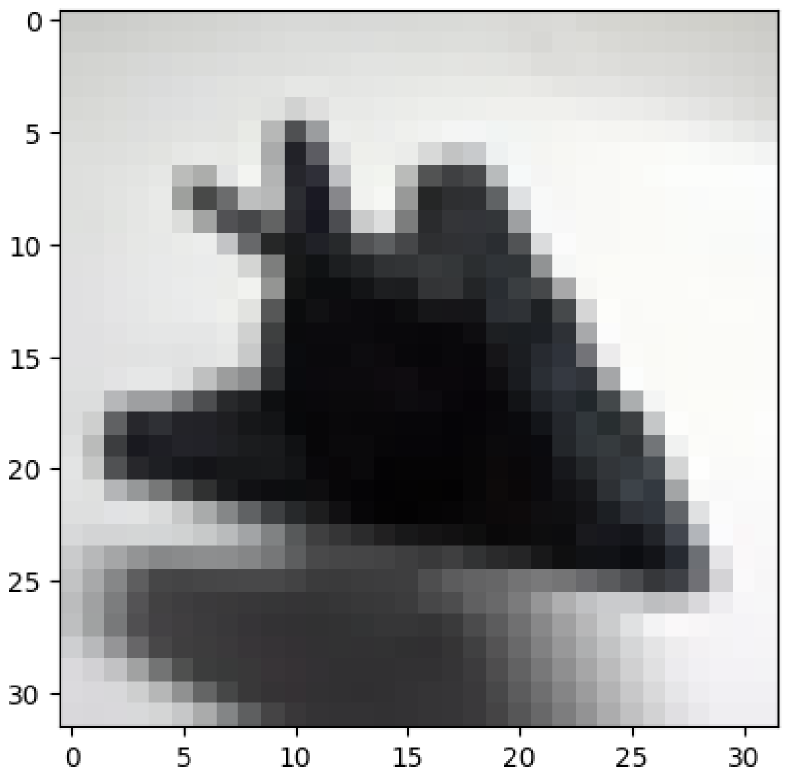
We can view the image using `matplotlib`, but we need to change the tensor dimensions to `(32,32,3)`. Let's create a helper function to display an image and its label.

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

matplotlib.rcParams['figure.facecolor'] = '#ffffff'


def show_example(img, label):
    print('Label: ', dataset.classes[label], "("+str(label)+")")
    plt.imshow(img.permute(1, 2, 0))
```

Let's look at a couple of images from the dataset. As you can tell, the 32x32px images are quite difficult to identify, even for the human eye. Try changing the indices below to view different images.

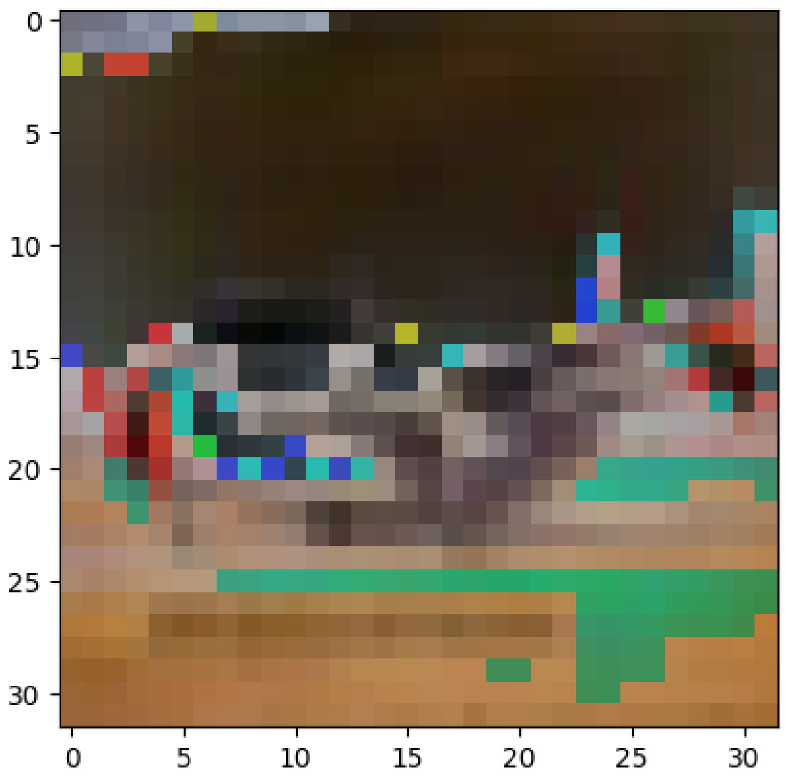
```
show_example(*dataset[0])
```

```
Label:  airplane (0)
```



```
show_example(*dataset[1099])
```

```
Label:  airplane (0)
```



Training and Validation Datasets

While building real world machine learning models, it is quite common to split the dataset into 3 parts:

1. **Training set** - used to train the model i.e. compute the loss and adjust the weights of the model using gradient descent.
2. **Validation set** - used to evaluate the model while training, adjust hyperparameters (learning rate etc.) and pick the best version of the model.
3. **Test set** - used to compare different models, or different types of modeling approaches, and report the final accuracy of the model.

Since there's no predefined validation set, we can set aside a small portion (5000 images) of the training set to be used as the validation set. We'll use the `random_split` helper method from PyTorch to do this. To ensure that we always create the same validation set, we'll also set a seed for the random number generator.

```
random_seed = 42
torch.manual_seed(random_seed);

val_size = 5000
train_size = len(dataset) - val_size

train_ds, val_ds = random_split(dataset, [train_size, val_size])
len(train_ds), len(val_ds)
```

```
(45000, 5000)
```

We can now create data loaders for training and validation, to load the data in batches

```
from torch.utils.data.dataloader import DataLoader

batch_size=128

train_dl = DataLoader(train_ds, batch_size, shuffle=True,
num_workers=4, pin_memory=True)
val_dl = DataLoader(val_ds, batch_size*2, num_workers=4,
pin_memory=True)
```

We can look at batches of images from the dataset using the `make_grid` method from `torchvision`. Each time the following code is run, we get a different bach, since the sampler shuffles the indices before creating batches.

```
from torchvision.utils import make_grid

def show_batch(dl):
    for images, labels in dl:
        fig, ax = plt.subplots(figsize=(12, 6))
        ax.set_xticks([]); ax.set_yticks([])
        ax.imshow(make_grid(images, nrow=16).permute(1, 2, 0))
        break

show_batch(train_dl)
```

Defining the Model (Convolutional Neural Network)

For this tutorial, we will use a convolutional neural network, using the `nn.Conv2d` class from PyTorch.

> The 2D convolution is a fairly simple operation at heart: you start with a kernel, which is simply a small matrix of weights. This kernel “slides” over the 2D input data, performing an elementwise multiplication with the part of the input it is currently on, and then summing up the results into a single output pixel.

Let us implement a convolution operation on a 1 channel image with a 3x3 kernel.

```python
def apply_kernel(image, kernel):
    ri, ci = image.shape       # image dimensions
    rk, ck = kernel.shape      # kernel dimensions
    ro, co = ri-rk+1, ci-ck+1  # output dimensions
    output = torch.zeros([ro, co])
    for i in range(ro):
        for j in range(co):
            output[i,j] = torch.sum(image[i:i+rk,j:j+ck] * kernel)
    return output
```

```python
sample_image = torch.tensor([
    [3, 3, 2, 1, 0],
    [0, 0, 1, 3, 1],
    [3, 1, 2, 2, 3],
    [2, 0, 0, 2, 2],
    [2, 0, 0, 0, 1]
], dtype=torch.float32)

sample_kernel = torch.tensor([
    [0, 1, 2],
    [2, 2, 0],
    [0, 1, 2]
], dtype=torch.float32)

apply_kernel(sample_image, sample_kernel)
```

```
tensor([[12., 12., 17.],
        [10., 17., 19.],
        [ 9.,  6., 14.]])
```

For multi-channel images, a different kernel is applied to each channels, and the outputs are added together pixel-wise.

Checking out the following articles to gain a better understanding of convolutions:

1. Intuitively understanding Convolutions for Deep Learning by Irhum Shafkat
2. Convolutions in Depth by Sylvian Gugger (this article implements convolutions from scratch)

There are certain advantages offered by convolutional layers when working with image data:

- **Fewer parameters**: A small set of parameters (the kernel) is used to calculate outputs of the entire image, so the model has much fewer parameters compared to a fully connected layer.
- **Sparsity of connections**: In each layer, each output element only depends on a small number of input elements, which makes the forward and backward passes more efficient.
- **Parameter sharing and spatial invariance**: The features learned by a kernel in one part of the image can be used to detect similar pattern in a different part of another image.

We will also use a max-pooling layers to progressively decrease the height & width of the output tensors from each convolutional layer.

Before we define the entire model, let's look at how a single convolutional layer followed by a max-pooling layer operates on the data.

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

simple_model = nn.Sequential(
    nn.Conv2d(3, 8, kernel_size=3, stride=1, padding=1),
    nn.MaxPool2d(2, 2)
)

for images, labels in train_dl:
    print('images.shape:', images.shape)
    out = simple_model(images)
    print('out.shape:', out.shape)
    break
```

```
images.shape: torch.Size([128, 3, 32, 32])
out.shape: torch.Size([128, 8, 16, 16])
```

The `Conv2d` layer transforms a 3-channel image to a 16-channel *feature map*, and the `MaxPool2d` layer halves the height and width. The feature map gets smaller as we add more layers, until we are finally left with a small feature map, which can be flattened into a vector. We can then add some fully connected layers at the end to get vector of size 10 for each image.

Let's define the model by extending an `ImageClassificationBase` class which contains helper methods for training & validation.

```
class ImageClassificationBase(nn.Module):
    def training_step(self, batch):
        images, labels = batch
        out = self(images)                  # Generate predictions
        loss = F.cross_entropy(out, labels) # Calculate loss
        return loss

    def validation_step(self, batch):
        images, labels = batch
        out = self(images)                    # Generate predictions
        loss = F.cross_entropy(out, labels)   # Calculate loss
        acc = accuracy(out, labels)           # Calculate accuracy
        return {'val_loss': loss.detach(), 'val_acc': acc}

    def validation_epoch_end(self, outputs):
        batch_losses = [x['val_loss'] for x in outputs]
        epoch_loss = torch.stack(batch_losses).mean()   # Combine
losses
        batch_accs = [x['val_acc'] for x in outputs]
        epoch_acc = torch.stack(batch_accs).mean()      # Combine
accuracies
        return {'val_loss': epoch_loss.item(), 'val_acc':
epoch_acc.item()}

    def epoch_end(self, epoch, result):
        print("Epoch [{}], train_loss: {:.4f}, val_loss: {:.4f},
val_acc: {:.4f}".format(
            epoch, result['train_loss'], result['val_loss'],
result['val_acc']))

def accuracy(outputs, labels):
    _, preds = torch.max(outputs, dim=1)
    return torch.tensor(torch.sum(preds == labels).item() /
len(preds))
```

We'll use `nn.Sequential` to chain the layers and activations functions into a single network architecture.

```
class Cifar10CnnModel(ImageClassificationBase):
    def __init__(self):
        super().__init__()
        self.network = nn.Sequential(
            #nn.Conv2d(in_channels, out_channels, kernel_size,
stride=1, padding=0, dilation=1, groups=1, bias=True,
padding_mode='zeros', device=None, dtype=None)
            nn.Conv2d(3, 32, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2), # output: 64 x 16 x 16

            nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2), # output: 128 x 8 x 8

            nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2), # output: 256 x 4 x 4

            nn.Flatten(),
            nn.Linear(256*4*4, 1024),
            nn.ReLU(),
            nn.Linear(1024, 512),
            nn.ReLU(),
            nn.Linear(512, 10)
        )

        """
        self.network = nn.Sequential(
            #nn.Conv2d(in_channels, out_channels, kernel_size,
stride=1, padding=0, dilation=1, groups=1, bias=True,
padding_mode='zeros', device=None, dtype=None)

            nn.Conv2d(3, 32, kernel_size=3, padding='same'),
            nn.SiLU(),
            nn.Conv2d(32, 128, kernel_size=3, stride=1,
padding='same'),
            nn.SiLU(),
            nn.MaxPool2d(2, 2), # output: 256 x 16 x 16
            nn.Conv2d(128, 256, kernel_size=3, stride=1,
padding='same'),
            nn.SiLU(),
            nn.MaxPool2d(2, 2), # output: 256 x 8 x 8
            nn.Conv2d(256, 256, kernel_size=3, stride=1,
padding='same'),
            nn.MaxPool2d(2, 2), # output: 256 x 4 x 4

            nn.Flatten(),
            nn.Linear(256*4*4, 2048),
            nn.SiLU(),
            nn.Linear(2048, 1024),
            nn.SiLU(),
            nn.Linear(1024, 512),
            nn.SiLU(),
            nn.Linear(512, 256),
            nn.SiLU(),
            nn.Linear(256, 128),
            nn.SiLU(),
            nn.Linear(128, 64),
            nn.SiLU(),
            nn.Linear(64, 10)
        )
        """

    def forward(self, xb):
        return self.network(xb)

model = Cifar10CnnModel()
model
```

```
Cifar10CnnModel(
  (network): Sequential(
    (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (1): ReLU()
    (2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (3): ReLU()
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
    (6): ReLU()
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
    (8): ReLU()
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
    (11): ReLU()
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
    (13): ReLU()
    (14): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (15): Flatten(start_dim=1, end_dim=-1)
    (16): Linear(in_features=4096, out_features=1024, bias=True)
    (17): ReLU()
    (18): Linear(in_features=1024, out_features=512, bias=True)
    (19): ReLU()
    (20): Linear(in_features=512, out_features=10, bias=True)
  )
)
```

Let's verify that the model produces the expected output on a batch of training data. The 10 outputs for each image can be interpreted as probabilities for the 10 target classes (after applying softmax), and the class with the highest probability is chosen as the label predicted by the model for the input image.

```
for images, labels in train_dl:
    print('images.shape:', images.shape)
    out = model(images)
    print('out.shape:', out.shape)
    print('out[0]:', out[0])
    break
```

```
images.shape: torch.Size([128, 3, 32, 32])
out.shape: torch.Size([128, 10])
out[0]: tensor([ 0.0240, -0.0467,  0.0066,  0.0193,  0.0043, -0.0598,
-0.0187, -0.0242,
         0.0431, -0.0163], grad_fn=<SelectBackward0>)
```

To seamlessly use a GPU, if one is available, we define a couple of helper functions (`get_default_device` & `to_device`) and a helper class `DeviceDataLoader` to move our model & data to the GPU as required.

```
def get_default_device():
    """Pick GPU if available, else CPU"""
    if torch.cuda.is_available():
        return torch.device('cuda')
    else:
        return torch.device('cpu')

def to_device(data, device):
    """Move tensor(s) to chosen device"""
    if isinstance(data, (list,tuple)):
        return [to_device(x, device) for x in data]
    return data.to(device, non_blocking=True)

class DeviceDataLoader():
    """Wrap a dataloader to move data to a device"""
    def __init__(self, dl, device):
        self.dl = dl
        self.device = device

    def __iter__(self):
        """Yield a batch of data after moving it to device"""
        for b in self.dl:
            yield to_device(b, self.device)

    def __len__(self):
        """Number of batches"""
        return len(self.dl)
```

Based on where you're running this notebook, your default device could be a CPU (`torch.device('cpu')`) or a GPU (`torch.device('cuda')`)

```
device = get_default_device()
device
```

```
device(type='cuda')
```

We can now wrap our training and validation data loaders using `DeviceDataLoader` for automatically transferring batches of data to the GPU (if available), and use `to_device` to move our model to the GPU (if available).

```
train_dl = DeviceDataLoader(train_dl, device)
val_dl = DeviceDataLoader(val_dl, device)
to_device(model, device);
```

Training the Model

We'll define two functions: `fit` and `evaluate` to train the model using gradient descent and evaluate its performance on the validation set.

```
@torch.no_grad()
def evaluate(model, val_loader):
    model.eval()
    outputs = [model.validation_step(batch) for batch in val_loader]
    return model.validation_epoch_end(outputs)

def fit(epochs, lr, model, train_loader, val_loader,
opt_func=torch.optim.SGD):
    history = []
    optimizer = opt_func(model.parameters(), lr)
    for epoch in range(epochs):
        # Training Phase
        model.train()
        train_losses = []
        for batch in train_loader:
            loss = model.training_step(batch)
            train_losses.append(loss)
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()
        # Validation phase
        result = evaluate(model, val_loader)
        result['train_loss'] = torch.stack(train_losses).mean().item()
        model.epoch_end(epoch, result)
        history.append(result)
    return history
```

Before we begin training, let's instantiate the model once again and see how it performs on the validation set with the initial set of parameters.

```
model = to_device(Cifar10CnnModel(), device)
```

```
evaluate(model, val_dl)
```

```
{'val_loss': 2.3022453784942627, 'val_acc': 0.10039062798023224}
```

The initial accuracy is around 10%, which is what one might expect from a randomly intialized model (since it has a 1 in 10 chance of getting a label right by guessing randomly).

We'll use the following *hyperparmeters* (learning rate, no. of epochs, batch_size etc.) to train our model. As an exercise, you can try changing these to see if you have achieve a higher accuracy in a shorter time.

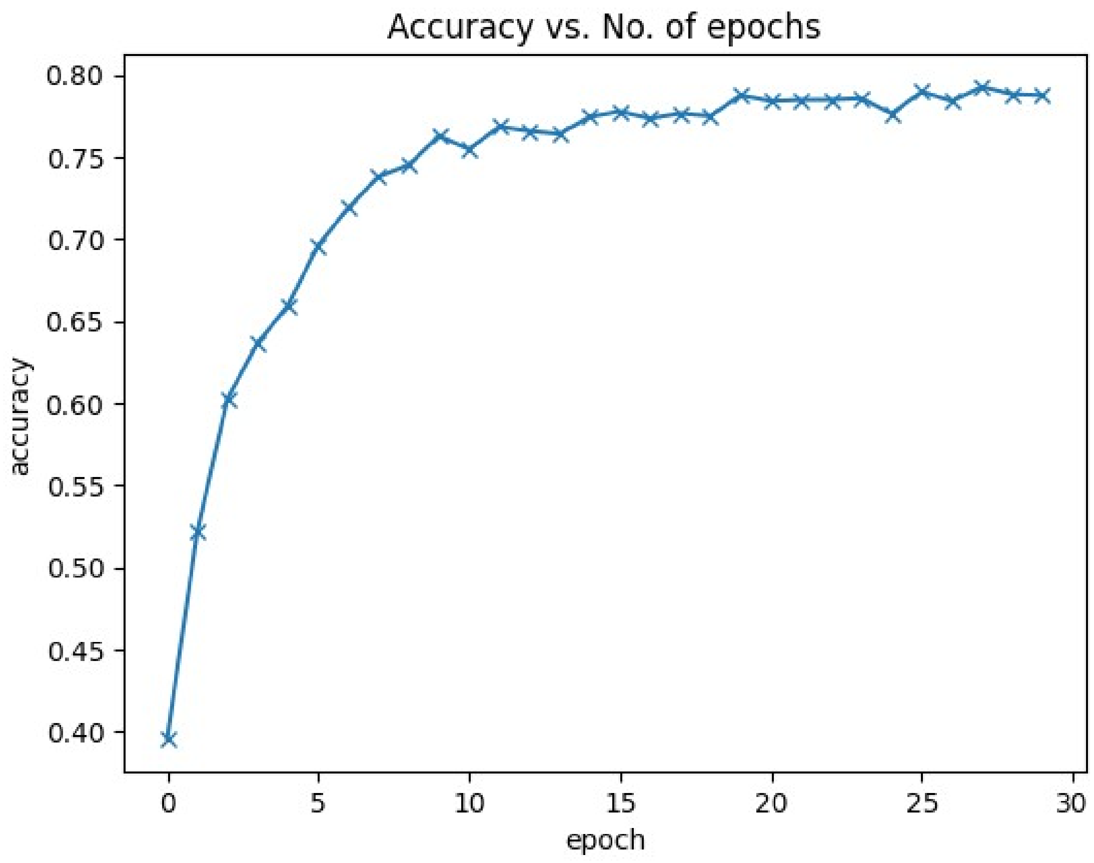
```
#num_epochs = 10
#lr = 0.001
num_epochs = 30
opt_func = torch.optim.Adam
#nAdam
lr = 0.001

history = fit(num_epochs, lr, model, train_dl, val_dl, opt_func)
```

```
Epoch [0], train_loss: 1.9109, val_loss: 1.6457, val_acc: 0.3958
Epoch [1], train_loss: 1.4787, val_loss: 1.3145, val_acc: 0.5220
Epoch [2], train_loss: 1.2250, val_loss: 1.1170, val_acc: 0.6028
Epoch [3], train_loss: 1.0385, val_loss: 1.0373, val_acc: 0.6368
Epoch [4], train_loss: 0.9118, val_loss: 0.9805, val_acc: 0.6594
Epoch [5], train_loss: 0.8317, val_loss: 0.8434, val_acc: 0.6961
Epoch [6], train_loss: 0.7492, val_loss: 0.7992, val_acc: 0.7189
Epoch [7], train_loss: 0.6878, val_loss: 0.7654, val_acc: 0.7380
Epoch [8], train_loss: 0.6404, val_loss: 0.7426, val_acc: 0.7450
Epoch [9], train_loss: 0.5939, val_loss: 0.7055, val_acc: 0.7627
Epoch [10], train_loss: 0.5576, val_loss: 0.7254, val_acc: 0.7546
Epoch [11], train_loss: 0.5302, val_loss: 0.6928, val_acc: 0.7683
Epoch [12], train_loss: 0.5016, val_loss: 0.6908, val_acc: 0.7657
Epoch [13], train_loss: 0.4705, val_loss: 0.7144, val_acc: 0.7640
Epoch [14], train_loss: 0.4452, val_loss: 0.6791, val_acc: 0.7743
Epoch [15], train_loss: 0.4283, val_loss: 0.6978, val_acc: 0.7777
Epoch [16], train_loss: 0.4100, val_loss: 0.7096, val_acc: 0.7733
Epoch [17], train_loss: 0.3923, val_loss: 0.6975, val_acc: 0.7765
Epoch [18], train_loss: 0.3798, val_loss: 0.6888, val_acc: 0.7748
Epoch [19], train_loss: 0.3590, val_loss: 0.6941, val_acc: 0.7875
Epoch [20], train_loss: 0.3411, val_loss: 0.7043, val_acc: 0.7839
Epoch [21], train_loss: 0.3271, val_loss: 0.6897, val_acc: 0.7847
Epoch [22], train_loss: 0.3283, val_loss: 0.7412, val_acc: 0.7847
Epoch [23], train_loss: 0.3181, val_loss: 0.7083, val_acc: 0.7856
Epoch [24], train_loss: 0.2988, val_loss: 0.7457, val_acc: 0.7763
Epoch [25], train_loss: 0.2934, val_loss: 0.7002, val_acc: 0.7896
Epoch [26], train_loss: 0.2892, val_loss: 0.7380, val_acc: 0.7839
Epoch [27], train_loss: 0.2855, val_loss: 0.6952, val_acc: 0.7926
Epoch [28], train_loss: 0.2709, val_loss: 0.7212, val_acc: 0.7881
Epoch [29], train_loss: 0.2695, val_loss: 0.7283, val_acc: 0.7874
```

We can also plot the valdation set accuracies to study how the model improves over time.

```
def plot_accuracies(history):
    accuracies = [x['val_acc'] for x in history]
    plt.plot(accuracies, '-x')
    plt.xlabel('epoch')
    plt.ylabel('accuracy')
    plt.title('Accuracy vs. No. of epochs');
```

```
plot_accuracies(history)
```

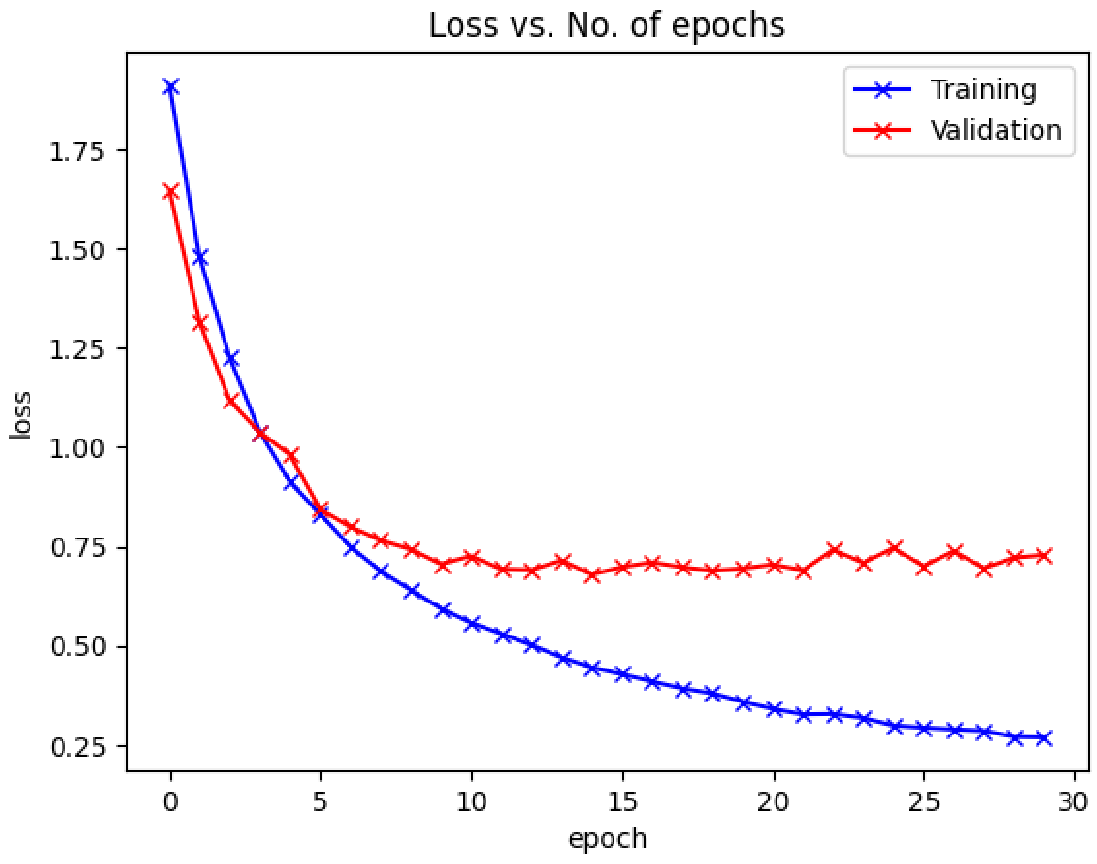


Our model reaches an accuracy of around 75%, and by looking at the graph, it seems unlikely that the model will achieve an accuracy higher than 80% even after training for a long time. This suggests that we might need to use a more powerful model to capture the relationship between the images and the labels more accurately. This can be done by adding more convolutional layers to our model, or incrasing the no. of channels in each convolutional layer, or by using regularization techniques.

We can also plot the training and validation losses to study the trend.

```
def plot_losses(history):
    train_losses = [x.get('train_loss') for x in history]
    val_losses = [x['val_loss'] for x in history]
    plt.plot(train_losses, '-bx')
    plt.plot(val_losses, '-rx')
    plt.xlabel('epoch')
    plt.ylabel('loss')
    plt.legend(['Training', 'Validation'])
    plt.title('Loss vs. No. of epochs');
```

```
plot_losses(history)
```



Initialy, both the training and validation losses seem to decrease over time. However, if you train the model for long enough, you will notice that the training loss continues to decrease, while the validation loss stops decreasing, and even starts to increase after a certain point!

This phenomenon is called **overfitting**, and it is the no. 1 why many machine learning models give rather terrible results on real-world data. It happens because the model, in an attempt to minimize the loss, starts to learn patters are are unique to the training data, sometimes even memorizing specific training examples. Because of this, the model does not generalize well to previously unseen data.

Following are some common stragegies for avoiding overfitting:

- Gathering and generating more training data, or adding noise to it
- Using regularization techniques like batch normalization & dropout
- Early stopping of model's training, when validation loss starts to increase

We will cover these topics in more detail in the next tutorial in this series, and learn how we can reach an accuracy of **over 90%** by making minor but important changes to our model.

Testing with individual images

While we have been tracking the overall accuracy of a model so far, it's also a good idea to look at model's results on some sample images. Let's test out our model with some images from the predefined test dataset of 10000 images. We begin by creating a test dataset using the `ImageFolder` class.

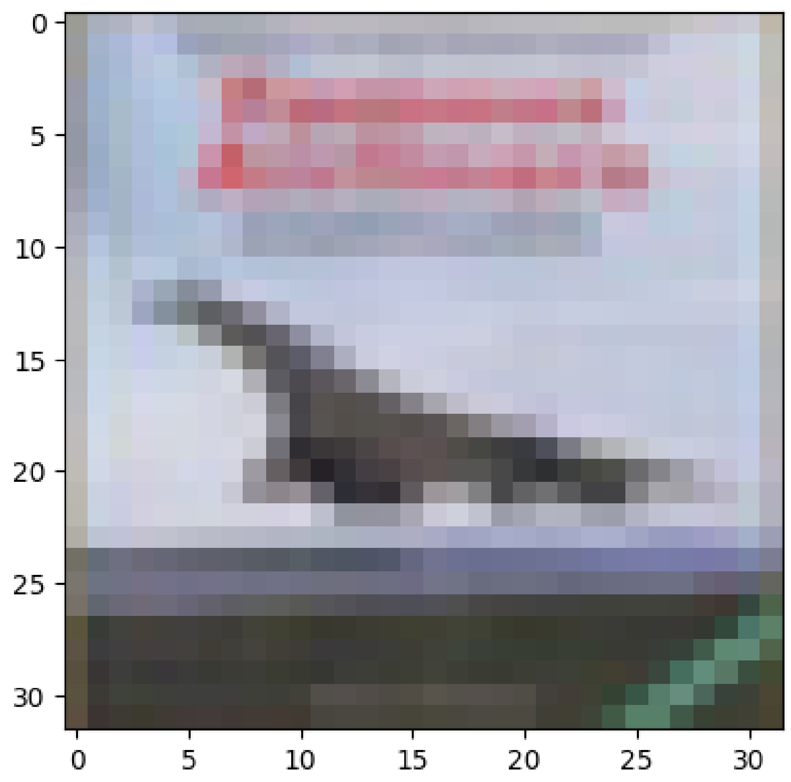
```
test_dataset = ImageFolder(data_dir+'/test', transform=tt.ToTensor())
```

Let's define a helper function `predict_image`, which returns the predicted label for a single image tensor.

```
def predict_image(img, model):
    # Convert to a batch of 1
    xb = to_device(img.unsqueeze(0), device)
    # Get predictions from model
    yb = model(xb)
    # Pick index with highest probability
    _, preds  = torch.max(yb, dim=1)
    # Retrieve the class label
    return dataset.classes[preds[0].item()]
```

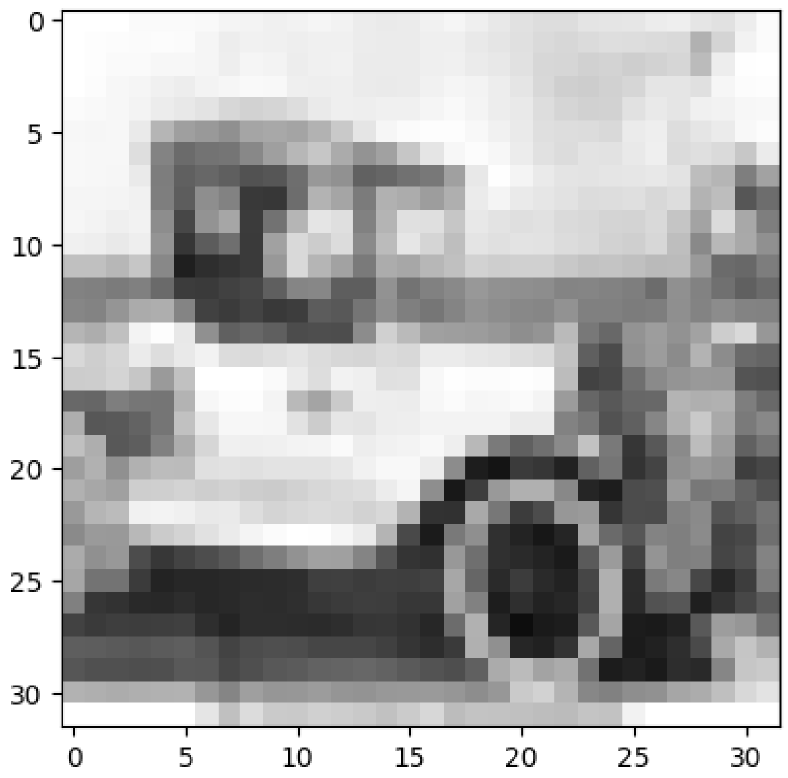
```
img, label = test_dataset[0]
plt.imshow(img.permute(1, 2, 0))
print('Label:', dataset.classes[label], ', Predicted:',
predict_image(img, model))
```

```
Label: airplane , Predicted: airplane
```



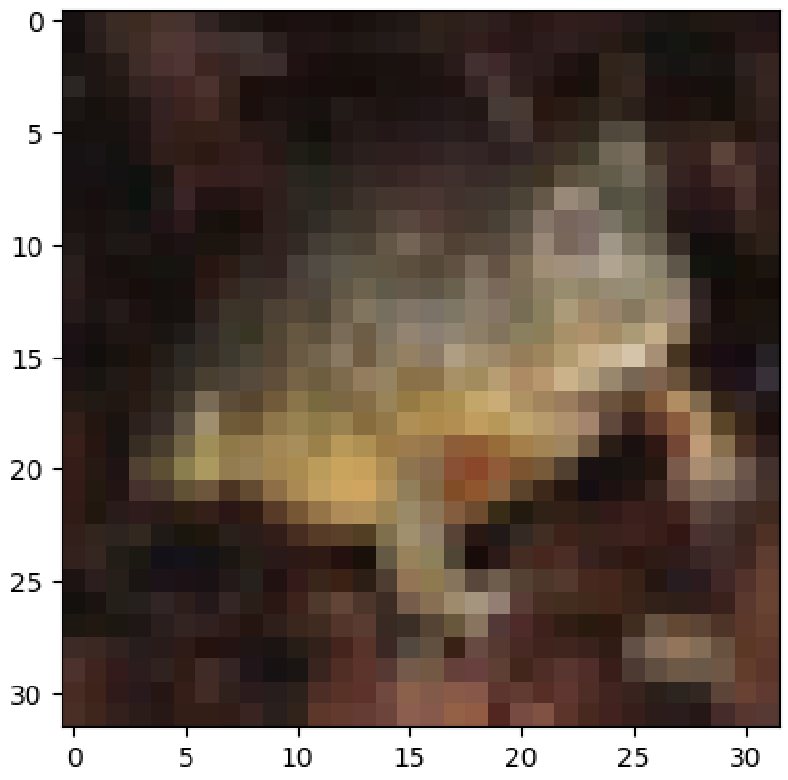
```
img, label = test_dataset[1002]
plt.imshow(img.permute(1, 2, 0))
print('Label:', dataset.classes[label], ', Predicted:',
predict_image(img, model))
```

```
Label: automobile , Predicted: automobile
```



```
img, label = test_dataset[6153]
plt.imshow(img.permute(1, 2, 0))
print('Label:', dataset.classes[label], ', Predicted:',
predict_image(img, model))
```

```
Label: frog , Predicted: frog
```



```
test_loader = DeviceDataLoader(DataLoader(test_dataset, batch_size*2),
device)
#{'val_loss': 0.8212520480155945, 'val_acc': 0.7828124761581421}
# Final {'val_loss': 0.6101927161216736, 'val_acc': 0.827832043170929}
result = evaluate(model, test_loader)
result
```

```
{'val_loss': 0.6101927161216736, 'val_acc': 0.827832043170929}
```

Saving and loading the model

Since we've trained our model for a long time and achieved a resonable accuracy, it would be a good idea to save the weights of the model to disk, so that we can reuse the model later and avoid retraining from scratch. Here's how you can save the model.

```
torch.save(model.state_dict(), 'cifar10-cnn.pth') #pth is an already
trained model
```

The `.state_dict` method returns an `OrderedDict` containing all the weights and bias matrices mapped to the right attributes of the model. To load the model weights, we can redefine the model with the same structure, and use the `.load_state_dict` method.

```
model2 = to_device(Cifar10CnnModel(), device)
```

```
model2.load_state_dict(torch.load('cifar10-cnn.pth'))
```

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

Just as a sanity check, let's verify that this model has the same loss and accuracy on the test set as before.

```
evaluate(model2, test_loader)
```

```
{'val_loss': 0.6101927161216736, 'val_acc': 0.827832043170929}
```

Summary and Further Reading/Exercises

We've covered a lot of ground in this tutorial. Here's quick recap of the topics:

- Introduction to the CIFAR10 dataset for image classification
- Downloading, extracing and loading an image dataset using `torchvision`
- Show random batches of images in a grid using `torchvision.utils.make_grid`
- Creating a convolutional neural network using with `nn.Conv2d` and `nn.MaxPool2d` layers
- Capturing dataset information, metrics and hyperparameters
- Training a convolutional neural network and visualizing the losses and errors
- Understanding overfitting and the strategies for avoiding it (more on this later)
- Generating predictions on single images from the test set
- Saving and loading the model weights, and attaching them to the eperiment

There's a lot of scope to experiment here, and I encourage you to use the interactive nature of Jupyter to play around with the various parameters. Here are a few ideas:

- Try chaging the hyperparameters to achieve a higher accuracy within fewer epochs.
- Try adding more convolutional layers, or increasing the number of channels in each convolutional layer
- Try using a feedforward neural network and see what's the maximum accuracy you can achieve
- Read about some of the startegies mentioned above for reducing overfitting and achieving better results, and try to implement them by looking into the PyTorch docs.
- Modify this notebook to train a model for a different dataset (e.g. CIFAR100 or ImageNet)