```
import torch
import torchvision
import numpy as np
import matplotlib.pyplot as plt
import torch.nn as nn
import torch.nn.functional as F
from torchvision.datasets import CIFAR10
from torchvision.transforms import ToTensor
from torchvision.utils import make_grid
from torch.utils.data.dataloader import DataLoader
from torch.utils.data import random_split
%matplotlib inline
```

```
dataset = CIFAR10(root='data/', download=True, transform=ToTensor())
test_dataset = CIFAR10(root='data/', train=False, transform=ToTensor())
```

```
Files already downloaded and verified
```

```
dataset_size = len(dataset)
dataset_size
```

```
50000
```

```
test_dataset_size = len(test_dataset)
test_dataset_size
```

```
10000
```

```
classes = dataset.classes
classes
```

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

```
img, label = dataset[0]
img_shape = img.shape
img_shape
```

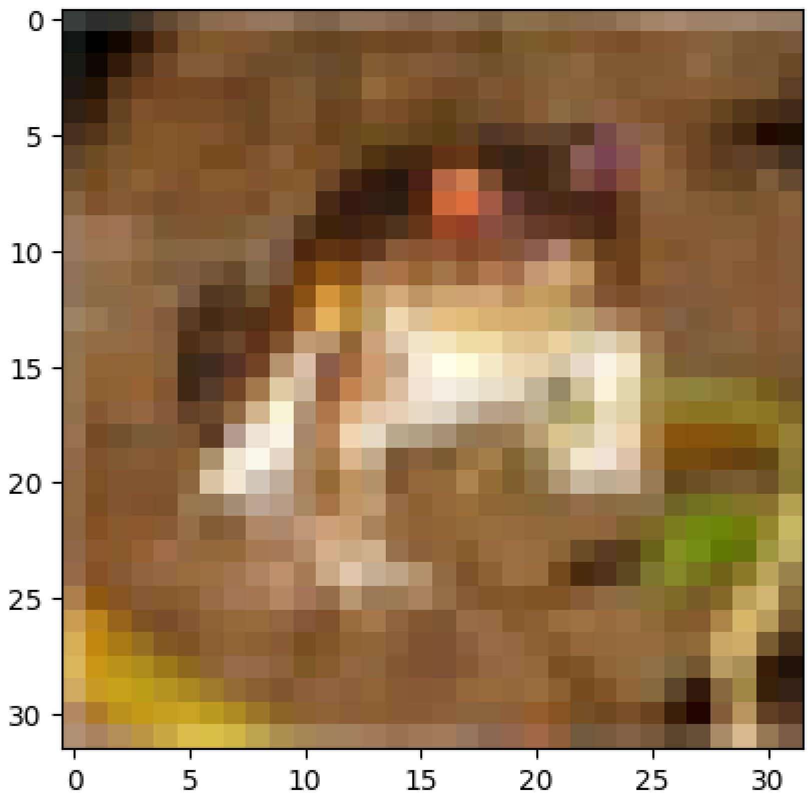
```
torch.Size([3, 32, 32])
```

```
num_classes = len(classes)
num_classes
```

```
10
```

```
img, label = dataset[0]
plt.imshow(img.permute((1, 2, 0)))
print('Label (numeric):', label)
print('Label (textual):', classes[label])
```

```
Label (numeric): 6
Label (textual): frog
```




```
torch.manual_seed(43)
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)
```

```
batch_size=128
```

```
train_loader = DataLoader(train_ds, batch_size, shuffle=True, num_workers=4, pin_memory=True)
val_loader = DataLoader(val_ds, batch_size*2, num_workers=4, pin_memory=True)
test_loader = DataLoader(test_dataset, batch_size*2, num_workers=4, pin_memory=True)
```

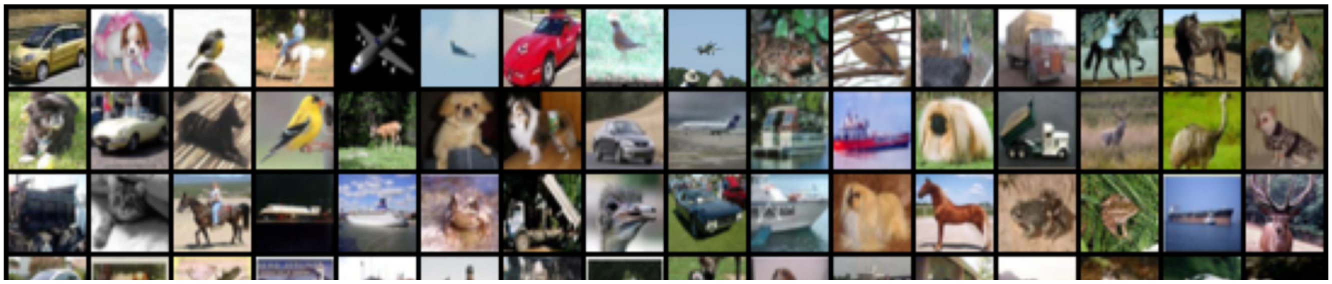
```
/usr/local/lib/python3.9/dist-packages/torch/utils/data/dataloader.py:561: UserWarning:
  warnings.warn(_create_warning_msg(
```

```
import matplotlib.pyplot as plt
from torchvision.utils import make_grid
```

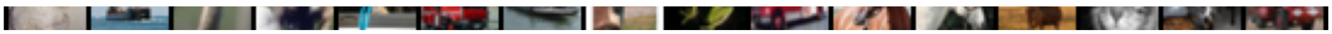
```
import matplotlib.pyplot as plt
from torchvision.utils import make_grid

for images, _ in train_loader:
    print('images.shape:', images.shape)
    plt.figure(figsize=(16,8))
    plt.axis('off')
    plt.imshow(make_grid(images, nrow=16).permute((1, 2, 0)))
    break
```

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



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



```
class ImageClassificationBase(nn.Module):
    def training_step(self, batch):
        images, labels = batch
        logits = self(images) # Generate logits
        loss = nn.CrossEntropyLoss()(logits, labels) # Calculate loss
        return loss



def validation_step(self, batch):
    images, labels = batch
    logits = self(images) # Generate logits
    loss = nn.CrossEntropyLoss()(logits, labels) # Calculate loss
    acc = accuracy(logits, 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.mean(torch.tensor(batch_losses)) # Combine losses
    batch_accs = [x['val_acc'] for x in outputs]
    epoch_acc = torch.mean(torch.tensor(batch_accs)) # Combine accuracies
    return {'val_loss': epoch_loss.item(), 'val_acc': epoch_acc.item()}



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



def evaluate(model, val_loader):
    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
        epoch_loss = 0.0
        for batch in train_loader:
            loss = model.training_step(batch)
            epoch_loss += loss.item()
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()
        epoch_loss /= len(train_loader)

        # Validation phase
        result = evaluate(model, val_loader)
        model.epoch_end(epoch, result)
        history.append(result)
    return history
```

```
import torch.cuda
```

```
import torch

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

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

```
device(type='cpu')
```

```
def to_device(data, device):
    """Move tensor(s) to chosen device"""
    # Check if the input is a tensor or a collection of tensors
    if isinstance(data, (list, tuple)):
        # If it is a collection, recursively call `to_device` on each element
        return [to_device(x, device) for x in data]
    # If it is a tensor, move it to the specified device
```

```
        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"""
            # Iterate over the batches in the wrapped dataloader
            for b in self.dl:
                # Move the batch to the specified device using `to_device`
                yield to_device(b, self.device)

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



    import matplotlib.pyplot as plt

    def plot_losses(history):
        """Plot validation loss vs. number of epochs"""
        # Extract the validation loss from the history dictionary
        losses = [x['val_loss'] for x in history]
        # Plot the validation loss as a function of epoch number
        plt.plot(losses, '-x')
        plt.xlabel('Epoch')
        plt.ylabel('Loss')
        plt.title('Validation Loss vs. No. of epochs')
        plt.show()

    def plot_accuracies(history):
        """Plot validation accuracy vs. number of epochs"""
        # Extract the validation accuracy from the history dictionary
        accuracies = [x['val_acc'] for x in history]
        # Plot the validation accuracy as a function of epoch number
        plt.plot(accuracies, '-x')
        plt.xlabel('Epoch')
        plt.ylabel('Accuracy')
        plt.title('Validation Accuracy vs. No. of epochs')
        plt.show()



    train_loader = DeviceDataLoader(train_loader, device)
    val_loader = DeviceDataLoader(val_loader, device)
    test_loader = DeviceDataLoader(test_loader, device)
```

```
input_size = 3*32*32
output_size = 10


def forward(self, xb):
    # Flatten images into vectors
    out = xb.view(xb.size(0), -1)
    # Apply layers & activation functions
    out = self.linear1(out)
    out = F.relu(out)
    out = self.linear2(out)
    out = F.relu(out)
    out = self.linear3(out)
    out = F.relu(out)
    out = self.linear4(out)
    out = F.relu(out)
    out = self.linear5(out)
    out = F.relu(out)
    out = self.linear6(out)
    out = F.softmax(out, dim=1) # <-- Use softmax activation for the output layer
    return out
```