Cifar10-CNN ipynb version

Image Classification using Convolutional Neural Networks(CNN) in PyTorch

1. Exploring the Cifar10 Dataset

In [1]:
```
import os
import torch
import torchvision
import tarfile
from torchvision.datasets.utils import download_url
from torch.utils.data import random_split
```

```
c:\Users\Infornet\anaconda3\envs\kingdu\lib\site-packages\tqdm\auto.py:22: TqdmWarning: IProgress not found. Please update jupyter and ipywidgets. See https://ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
c:\Users\Infornet\anaconda3\envs\kingdu\lib\site-packages\torchvision\io\image.py:13: UserWarning: Failed to load image Python extension:
  warn(f"Failed to load image Python extension: {e}")
```

In [2]:
```
# Dowload the dataset
dataset_url = "https://s3.amazonaws.com/fast-ai-imageclas/cifar10.tgz"
download_url(dataset_url, '.')
```

```
Using downloaded and verified file: .\cifar10.tgz
```

In [3]:
```
# Extract from archive
with tarfile.open('./cifar10.tgz', 'r:gz') as tar:
    tar.extractall(path='./data')
```

In [4]:
```
data_dir = './data/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']
```

두개의 폴더 내부를 살펴보면, 하나는 train, test이고 각 클래스에 동일한 수의 이미지가 있는지 확인. train은 5000개, test는 1000개

In [5]:
```
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']
```

In [6]:
```
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']
```

이미지 폴더의 class를 torchvision을 통해 pytorch tensor로 로드해본다.

In [7]:
```
from torchvision.datasets import ImageFolder
from torchvision.transforms import ToTensor
```

In [8]:
```
dataset = ImageFolder(data_dir+'/train', transform = ToTensor())
```

각 원소는 튜플이고 이미지 텐서와 라벨을 갖고있다.
cf. 튜플 : 순서가 있는 객체의 집합으로 리스트와 유사하다. 하지만 값 변경 X

- 이미지 텐서는 32*32 pixel 에 channel은 3(RGB)라서 각 이미지 shape = (3,32,32)

In [9]:
```
img, label = dataset[0]
print(img.shape, label)
img
```

Out[9]:
```
torch.Size([3, 32, 32]) 0
tensor([[[0.7922, 0.7922, 0.8000,  ..., 0.8118, 0.8039, 0.7961],
         [0.8078, 0.8078, 0.8118,  ..., 0.8235, 0.8157, 0.8078],
         [0.8235, 0.8275, 0.8314,  ..., 0.8392, 0.8314, 0.8235],
         ...,
         [0.8549, 0.8235, 0.7608,  ..., 0.9529, 0.9569, 0.9529],
         [0.8588, 0.8510, 0.8471,  ..., 0.9451, 0.9451, 0.9451],
         [0.8510, 0.8471, 0.8510,  ..., 0.9373, 0.9373, 0.9412]],

        [[0.8000, 0.8000, 0.8078,  ..., 0.8157, 0.8078, 0.8000],
         [0.8157, 0.8157, 0.8196,  ..., 0.8275, 0.8196, 0.8118],
         [0.8314, 0.8353, 0.8392,  ..., 0.8392, 0.8353, 0.8275],
         ...,
         [0.8510, 0.8196, 0.7608,  ..., 0.9490, 0.9490, 0.9529],
         [0.8549, 0.8471, 0.8471,  ..., 0.9412, 0.9412, 0.9412],
         [0.8471, 0.8431, 0.8471,  ..., 0.9333, 0.9333, 0.9333]],

        [[0.7804, 0.7804, 0.7882,  ..., 0.7843, 0.7804, 0.7765],
         [0.7961, 0.7961, 0.8000,  ..., 0.8039, 0.7961, 0.7882],
         [0.8118, 0.8157, 0.8235,  ..., 0.8235, 0.8157, 0.8078],
         ...,
         [0.8706, 0.8392, 0.7765,  ..., 0.9686, 0.9686, 0.9686],
         [0.8745, 0.8667, 0.8627,  ..., 0.9608, 0.9608, 0.9608],
         [0.8667, 0.8627, 0.8667,  ..., 0.9529, 0.9529, 0.9529]]])
```

In [10]:
```
print(dataset.classes)
```

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

여기 ipynb 상에서 시각화를 하고싶다면 matplotlib을 써야하는데 이때는 텐서 차원을
(32x32x3)으로 바꿔야함

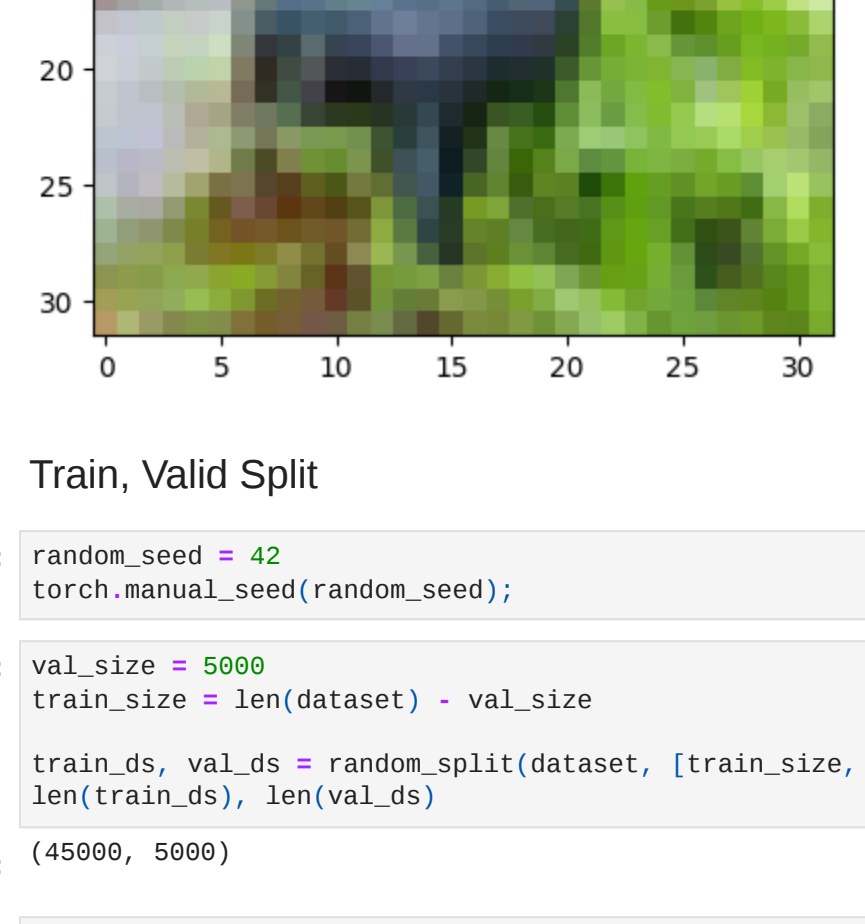
In [11]:
```
import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline
          # 요건 그림 위에 수식 없이 바로 보여지게끔 하는 코딩
matplotlib.rcParams['figure.facecolor'] = '#ffffff'
```

In [12]:
```
def show_example(img, label) :
    print('Label: ', dataset.classes[label], "(" + str(label) + ")")
    plt.imshow(img.permute(1,2,0))

# """
# # Permute() : 모든 차원들을 맞교환할 수 있음.
# ex)
# x = torch.rand(16, 32, 3)
# y = x.permute(2,1,0)    --> [3, 32, 16]
# """
```

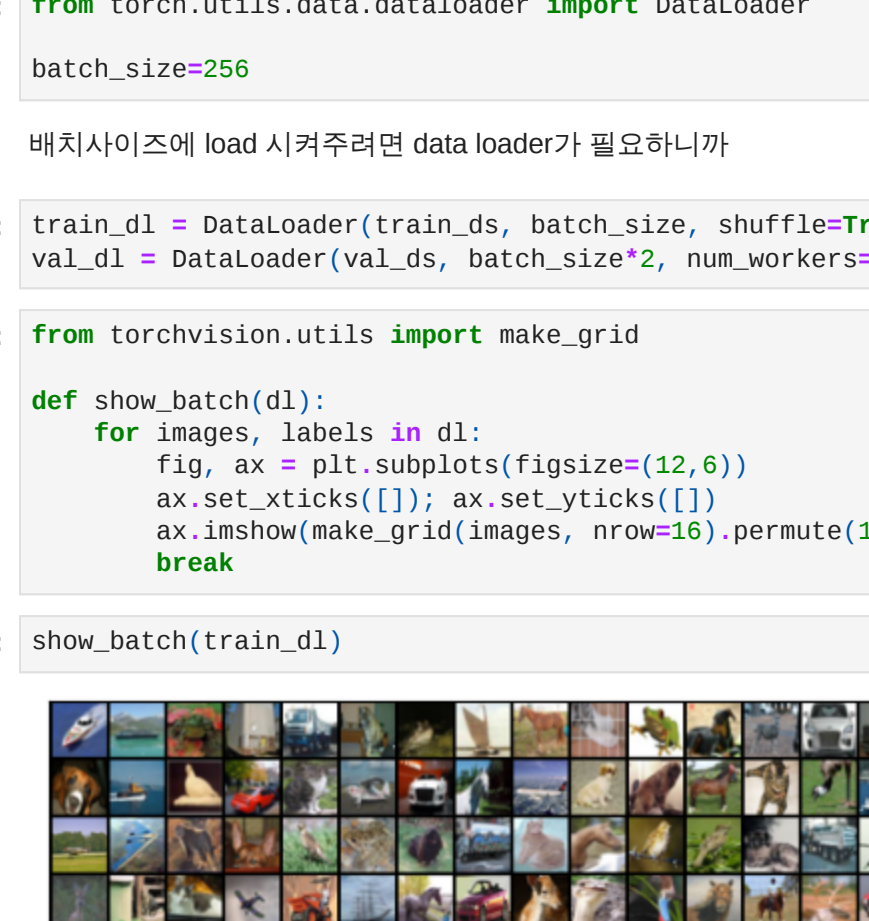
In [13]:
```
show_example(*dataset[0])
```

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



In [14]:
```
show_example(*dataset[10000])
```

```
Label:  bird (2)
```



Train, Valid Split

In [15]:
```
random_seed = 42
torch.manual_seed(random_seed);
```

In [16]:
```
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)
```

Out[16]:
```
(45000, 5000)
```

In [17]:
```
from torch.utils.data.dataloader import DataLoader

batch_size=256
```

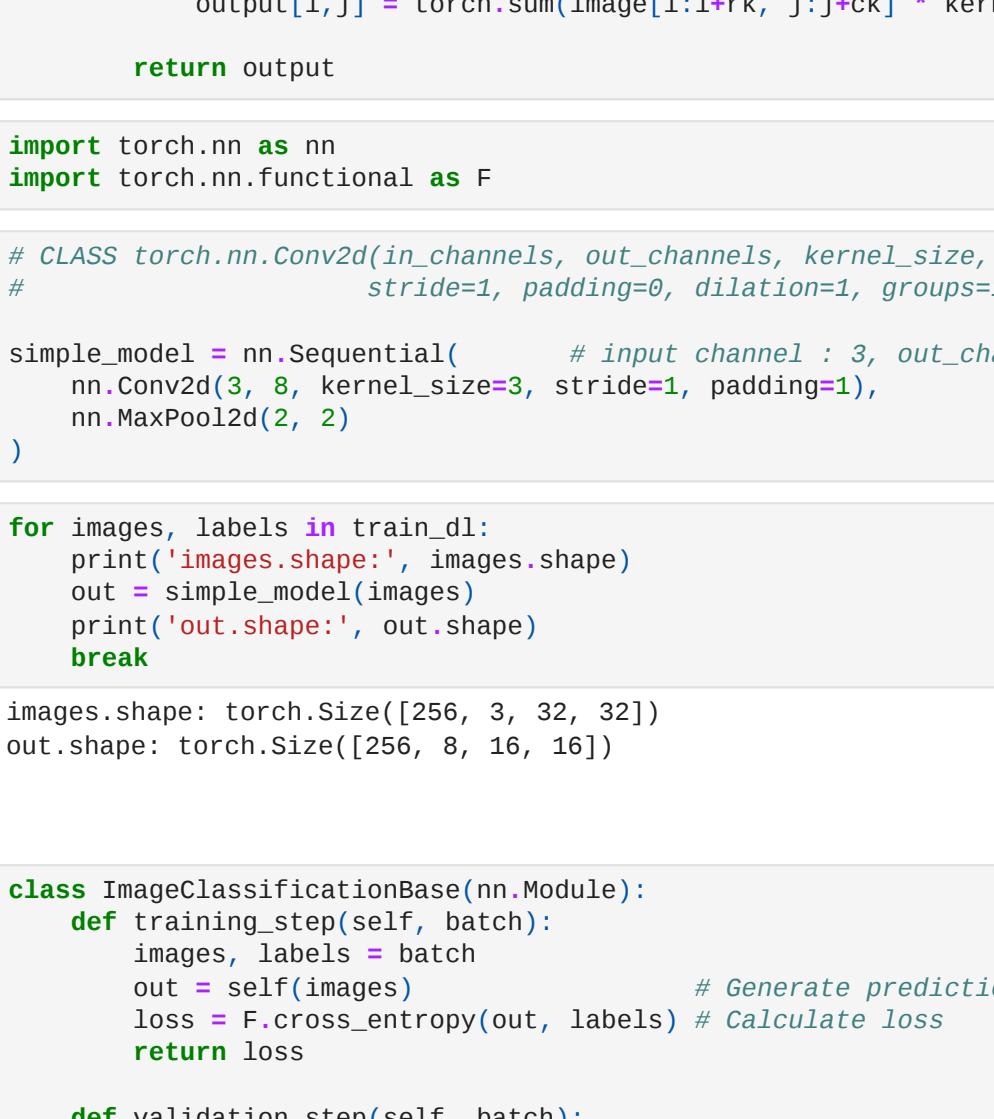
배치사이즈에 load 시켜주려면 data loader가 필요하니까

In [18]:
```
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)
```

In [19]:
```
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
```

In [20]:
```
show_batch(train_dl)
```



Defining the Model(Convolutional Neural Network)

In [21]:
```
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
```

In [22]:
```
import torch.nn as nn
import torch.nn.functional as F
```

In [23]:
```
# CLASS torch.nn.Conv2d(in_channels, out_channels, kernel_size,
#                       stride=1, padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros', device=N

simple_model = nn.Sequential(       # input channel : 3, out_channel:
    nn.Conv2d(3, 8, kernel_size=3, stride=1, padding=1),
    nn.MaxPool2d(2, 2)
)
```

In [24]:
```
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([256, 3, 32, 32])
out.shape: torch.Size([256, 8, 16, 16])
```

In [25]:
```
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))
```

nn.sequential 을 사용하여 층과 활성화함수를 하나의 network architecture로 연결시킬것이다.

In [26]:
```
# torch.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

class Cifar10CnnModel(ImageClassificationBase):
    def __init__(self):
        super().__init__()
        self.network = nn.Sequential(
            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))

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

In [27]:
```
model = Cifar10CnnModel()
model
```

Out[27]:
```
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)
  )
)
```

In [28]:
```
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([256, 3, 32, 32])
out.shape: torch.Size([256, 10])
out[0]: tensor([ 0.0239, -0.0466,  0.0067,  0.0193,  0.0044, -0.0598, -0.0188, -0.0242,
         0.0431, -0.0164], grad_fn=<SelectBackward0>)
```

GPU를 원활하게 사용하기 위해 사용 가능한 경우 몇 가지 도우미 함수(get_default_device 및 to_device)와 도우미 클래스 DeviceDataLoader를 정의하여 필요에 따라 모델 및 데이터를 GPU로 이동시킨다!

In [29]:
```
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)
```

In [30]:
```
device = get_default_device()
device
```

Out[30]:
```
device(type='cuda')
```

이제 DeviceDataLoader를 사용하여 데이터 배치를 GPU로 자동 전송(사용 가능한 경우)하고 to_device를 사용하여 모델을 GPU(사용 가능한 경우)로 이동하기 위해 교육 및 검증 데이터 로더를 래핑할 수 있습니다.

In [31]:
```
train_dl = DeviceDataLoader(train_dl, device)
val_dl = DeviceDataLoader(val_dl, device)
to_device(model, device);
```

Training the Model

In [32]:
```
@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
```

In [33]:
```
model = to_device(Cifar10CnnModel(), device)
```

In [34]:
```
evaluate(model, val_dl)
```

Out[34]:
```
{'val_loss': 2.3023250102996826, 'val_acc': 0.09949378669261932}
```

무작위에서 뽑았을때(초기정확도)는 10% 이며, 우리는 모델을 훈련하기 위해 하이퍼파라미터(학습속도, epoch수, batch_size 등)를 사용할 것이다. -> 높은 정확도를 위하여

In [35]:
```
num_epochs = 200
opt_func = torch.optim.Adagrad
lr = 0.001
```

In [36]:
```
history = fit(num_epochs, lr, model, train_dl, val_dl, opt_func)
```

```
Epoch [0], train_loss: 1.9730, val_loss: 1.7252, val_acc: 0.3597
Epoch [1], train_loss: 1.6415, val_loss: 1.5660, val_acc: 0.4234
Epoch [2], train_loss: 1.5304, val_loss: 1.5020, val_acc: 0.4508
Epoch [3], train_loss: 1.4647, val_loss: 1.4388, val_acc: 0.4764
Epoch [4], train_loss: 1.4160, val_loss: 1.4121, val_acc: 0.4876
Epoch [5], train_loss: 1.3730, val_loss: 1.3694, val_acc: 0.5059
Epoch [6], train_loss: 1.3363, val_loss: 1.3475, val_acc: 0.5103
Epoch [7], train_loss: 1.3025, val_loss: 1.2974, val_acc: 0.5313
Epoch [8], train_loss: 1.2737, val_loss: 1.2781, val_acc: 0.5435
Epoch [9], train_loss: 1.2467, val_loss: 1.2480, val_acc: 0.5535
Epoch [10], train_loss: 1.2234, val_loss: 1.2344, val_acc: 0.5585
Epoch [11], train_loss: 1.2021, val_loss: 1.2615, val_acc: 0.5484
Epoch [12], train_loss: 1.1821, val_loss: 1.1967, val_acc: 0.5704
Epoch [13], train_loss: 1.1627, val_loss: 1.1969, val_acc: 0.5793
Epoch [14], train_loss: 1.1453, val_loss: 1.1791, val_acc: 0.5859
Epoch [15], train_loss: 1.1291, val_loss: 1.1931, val_acc: 0.5721
Epoch [16], train_loss: 1.1132, val_loss: 1.1561, val_acc: 0.5881
Epoch [17], train_loss: 1.0997, val_loss: 1.1308, val_acc: 0.5968
Epoch [18], train_loss: 1.0818, val_loss: 1.1476, val_acc: 0.5863
Epoch [19], train_loss: 1.0693, val_loss: 1.1188, val_acc: 0.5989
Epoch [20], train_loss: 1.0559, val_loss: 1.1059, val_acc: 0.6058
Epoch [21], train_loss: 1.0426, val_loss: 1.0953, val_acc: 0.6086
Epoch [22], train_loss: 1.0329, val_loss: 1.0988, val_acc: 0.6132
Epoch [23], train_loss: 1.0196, val_loss: 1.0830, val_acc: 0.6192
Epoch [24], train_loss: 1.0069, val_loss: 1.0789, val_acc: 0.6200
Epoch [25], train_loss: 0.9986, val_loss: 1.1003, val_acc: 0.6078
Epoch [26], train_loss: 0.9874, val_loss: 1.0638, val_acc: 0.6238
Epoch [27], train_loss: 0.9737, val_loss: 1.0492, val_acc: 0.6320
Epoch [28], train_loss: 0.9664, val_loss: 1.0464, val_acc: 0.6354
Epoch [29], train_loss: 0.9560, val_loss: 1.0429, val_acc: 0.6350
Epoch [30], train_loss: 0.9456, val_loss: 1.0500, val_acc: 0.6286
Epoch [31], train_loss: 0.9408, val_loss: 1.0220, val_acc: 0.6440
Epoch [32], train_loss: 0.9312, val_loss: 1.0397, val_acc: 0.6275
Epoch [33], train_loss: 0.9193, val_loss: 1.0411, val_acc: 0.6328
Epoch [34], train_loss: 0.9108, val_loss: 1.0321, val_acc: 0.6419
Epoch [35], train_loss: 0.9046, val_loss: 1.0075, val_acc: 0.6462
Epoch [36], train_loss: 0.8950, val_loss: 1.0609, val_acc: 0.6365
Epoch [37], train_loss: 0.8869, val_loss: 0.9979, val_acc: 0.6560
Epoch [38], train_loss: 0.8811, val_loss: 1.0007, val_acc: 0.6517
Epoch [39], train_loss: 0.8723, val_loss: 1.0092, val_acc: 0.6500
Epoch [40], train_loss: 0.8631, val_loss: 0.9875, val_acc: 0.6575
Epoch [41], train_loss: 0.8575, val_loss: 1.0084, val_acc: 0.6530
Epoch [42], train_loss: 0.8521, val_loss: 0.9722, val_acc: 0.6688
Epoch [43], train_loss: 0.8458, val_loss: 0.9909, val_acc: 0.6577
Epoch [44], train_loss: 0.8346, val_loss: 0.9617, val_acc: 0.6694
Epoch [45], train_loss: 0.8303, val_loss: 0.9926, val_acc: 0.6573
Epoch [46], train_loss: 0.8237, val_loss: 0.9715, val_acc: 0.6602
Epoch [47], train_loss: 0.8178, val_loss: 0.9913, val_acc: 0.6527
Epoch [48], train_loss: 0.8102, val_loss: 0.9642, val_acc: 0.6675
Epoch [49], train_loss: 0.8029, val_loss: 0.9675, val_acc: 0.6687
Epoch [50], train_loss: 0.7962, val_loss: 0.9696, val_acc: 0.6644
Epoch [51], train_loss: 0.7917, val_loss: 0.9471, val_acc: 0.6768
Epoch [52], train_loss: 0.7855, val_loss: 0.9611, val_acc: 0.6693
Epoch [53], train_loss: 0.7780, val_loss: 0.9442, val_acc: 0.6705
Epoch [54], train_loss: 0.7713, val_loss: 0.9770, val_acc: 0.6581
Epoch [55], train_loss: 0.7652, val_loss: 0.9503, val_acc: 0.6737
Epoch [56], train_loss: 0.7601, val_loss: 0.9411, val_acc: 0.6763
Epoch [57], train_loss: 0.7524, val_loss: 0.9506, val_acc: 0.6704
Epoch [58], train_loss: 0.7483, val_loss: 0.9504, val_acc: 0.6699
Epoch [59], train_loss: 0.7422, val_loss: 0.9396, val_acc: 0.6773
Epoch [60], train_loss: 0.7352, val_loss: 0.9247, val_acc: 0.6873
Epoch [61], train_loss: 0.7290, val_loss: 0.9225, val_acc: 0.6824
Epoch [62], train_loss: 0.7222, val_loss: 0.9290, val_acc: 0.6832
Epoch [63], train_loss: 0.7169, val_loss: 0.9252, val_acc: 0.6843
Epoch [64], train_loss: 0.7117, val_loss: 0.9370, val_acc: 0.6806
Epoch [65], train_loss: 0.7053, val_loss: 0.9434, val_acc: 0.6797
Epoch [66], train_loss: 0.6994, val_loss: 0.9291, val_acc: 0.6841
Epoch [67], train_loss: 0.6926, val_loss: 0.9131, val_acc: 0.6928
Epoch [68], train_loss: 0.6890, val_loss: 0.9325, val_acc: 0.6842
Epoch [69], train_loss: 0.6837, val_loss: 0.9132, val_acc: 0.6915
Epoch [70], train_loss: 0.6786, val_loss: 0.9146, val_acc: 0.6936
Epoch [71], train_loss: 0.6744, val_loss: 0.9255, val_acc: 0.6857
Epoch [72], train_loss: 0.6671, val_loss: 0.9180, val_acc: 0.6905
Epoch [73], train_loss: 0.6654, val_loss: 0.9223, val_acc: 0.6939
Epoch [74], train_loss: 0.6554, val_loss: 0.9203, val_acc: 0.6909
Epoch [75], train_loss: 0.6502, val_loss: 0.9139, val_acc: 0.6935
Epoch [76], train_loss: 0.6462, val_loss: 0.9138, val_acc: 0.6909
Epoch [77], train_loss: 0.6390, val_loss: 0.9340, val_acc: 0.6910
Epoch [78], train_loss: 0.6374, val_loss: 0.9361, val_acc: 0.6890
Epoch [79], train_loss: 0.6311, val_loss: 0.9146, val_acc: 0.6995
Epoch [80], train_loss: 0.6260, val_loss: 0.9152, val_acc: 0.6966
Epoch [81], train_loss: 0.6189, val_loss: 0.9079, val_acc: 0.6991
Epoch [82], train_loss: 0.6136, val_loss: 0.9128, val_acc: 0.6964
Epoch [83], train_loss: 0.6074, val_loss: 0.9133, val_acc: 0.7007
Epoch [84], train_loss: 0.6042, val_loss: 0.9127, val_acc: 0.7009
Epoch [85], train_loss: 0.5990, val_loss: 0.9389, val_acc: 0.6914
Epoch [86], train_loss: 0.5928, val_loss: 0.9139, val_acc: 0.7003
Epoch [87], train_loss: 0.5882, val_loss: 0.9328, val_acc: 0.6891
Epoch [88], train_loss: 0.5835, val_loss: 0.9190, val_acc: 0.6998
Epoch [89], train_loss: 0.5793, val_loss: 0.9205, val_acc: 0.6947
Epoch [90], train_loss: 0.5758, val_loss: 0.9106, val_acc: 0.7020
Epoch [91], train_loss: 0.5696, val_loss: 0.9288, val_acc: 0.6953
Epoch [92], train_loss: 0.5654, val_loss: 0.9358, val_acc: 0.6948
Epoch [93], train_loss: 0.5592, val_loss: 0.9248, val_acc: 0.6985
Epoch [94], train_loss: 0.5539, val_loss: 0.9273, val_acc: 0.7001
Epoch [95], train_loss: 0.5494, val_loss: 0.9207, val_acc: 0.7006
Epoch [96], train_loss: 0.5444, val_loss: 0.9354, val_acc: 0.6934
Epoch [97], train_loss: 0.5390, val_loss: 0.9316, val_acc: 0.6991
Epoch [98], train_loss: 0.5342, val_loss: 0.9266, val_acc: 0.7000
Epoch [99], train_loss: 0.5294, val_loss: 0.9488, val_acc: 0.6969
Epoch [100], train_loss: 0.5235, val_loss: 0.9346, val_acc: 0.6995
Epoch [101], train_loss: 0.5225, val_loss: 0.9305, val_acc: 0.7009
Epoch [102], train_loss: 0.5131, val_loss: 0.9407, val_acc: 0.7028
Epoch [103], train_loss: 0.5083, val_loss: 0.9526, val_acc: 0.6975
Epoch [104], train_loss: 0.5046, val_loss: 0.9318, val_acc: 0.6991
Epoch [105], train_loss: 0.5009, val_loss: 0.9401, val_acc: 0.6982
Epoch [106], train_loss: 0.4948, val_loss: 0.9306, val_acc: 0.7025
Epoch [107], train_loss: 0.4925, val_loss: 0.9850, val_acc: 0.6892
Epoch [108], train_loss: 0.4846, val_loss: 0.9490, val_acc: 0.6960
Epoch [109], train_loss: 0.4783, val_loss: 0.9501, val_acc: 0.7018
Epoch [110], train_loss: 0.4762, val_loss: 0.9432, val_acc: 0.7014
Epoch [111], train_loss: 0.4715, val_loss: 0.9482, val_acc: 0.7065
Epoch [112], train_loss: 0.4681, val_loss: 0.9562, val_acc: 0.7023
Epoch [113], train_loss: 0.4639, val_loss: 0.9633, val_acc: 0.6965
Epoch [114], train_loss: 0.4559, val_loss: 0.9710, val_acc: 0.6920
Epoch [115], train_loss: 0.4517, val_loss: 0.9619, val_acc: 0.7008
Epoch [116], train_loss: 0.4456, val_loss: 0.9608, val_acc: 0.7054
Epoch [117], train_loss: 0.4460, val_loss: 0.9628, val_acc: 0.7060
Epoch [118], train_loss: 0.4373, val_loss: 0.9884, val_acc: 0.7015
Epoch [119], train_loss: 0.4343, val_loss: 0.9837, val_acc: 0.6997
Epoch [120], train_loss: 0.4276, val_loss: 0.9632, val_acc: 0.7011
Epoch [121], train_loss: 0.4230, val_loss: 0.9920, val_acc: 0.6968
Epoch [122], train_loss: 0.4228, val_loss: 0.9759, val_acc: 0.7008
Epoch [123], train_loss: 0.4146, val_loss: 0.9883, val_acc: 0.7009
Epoch [124], train_loss: 0.4111, val_loss: 1.0082, val_acc: 0.6963
Epoch [125], train_loss: 0.4056, val_loss: 0.9961, val_acc: 0.6957
Epoch [126], train_loss: 0.4026, val_loss: 1.0115, val_acc: 0.6997
Epoch [127], train_loss: 0.3955, val_loss: 0.9793, val_acc: 0.7034
Epoch [128], train_loss: 0.3935, val_loss: 1.0134, val_acc: 0.6997
Epoch [129], train_loss: 0.3878, val_loss: 1.0131, val_acc: 0.6980
Epoch [130], train_loss: 0.3815, val_loss: 1.0071, val_acc: 0.6986
Epoch [131], train_loss: 0.3774, val_loss: 1.0262, val_acc: 0.6986
Epoch [132], train_loss: 0.3740, val_loss: 1.0409, val_acc: 0.6949
Epoch [133], train_loss: 0.3695, val_loss: 1.0225, val_acc: 0.6966
Epoch [134], train_loss: 0.3675, val_loss: 1.0216, val_acc: 0.7045
Epoch [135], train_loss: 0.3583, val_loss: 1.0214, val_acc: 0.7015
Epoch [136], train_loss: 0.3545, val_loss: 1.0523, val_acc: 0.6997
Epoch [137], train_loss: 0.3542, val_loss: 1.0394, val_acc: 0.7000
Epoch [138], train_loss: 0.3472, val_loss: 1.0577, val_acc: 0.7010
Epoch [139], train_loss: 0.3422, val_loss: 1.0812, val_acc: 0.6908
Epoch [140], train_loss: 0.3395, val_loss: 1.0594, val_acc: 0.6944
Epoch [141], train_loss: 0.3343, val_loss: 1.0533, val_acc: 0.7019
Epoch [142], train_loss: 0.3291, val_loss: 1.0960, val_acc: 0.6960
Epoch [143], train_loss: 0.3274, val_loss: 1.0488, val_acc: 0.7034
Epoch [144], train_loss: 0.3242, val_loss: 1.0799, val_acc: 0.6928
Epoch [145], train_loss: 0.3164, val_loss: 1.0711, val_acc: 0.7028
Epoch [146], train_loss: 0.3162, val_loss: 1.0739, val_acc: 0.6973
Epoch [147], train_loss: 0.3101, val_loss: 1.0787, val_acc: 0.7041
Epoch [148], train_loss: 0.3045, val_loss: 1.1401, val_acc: 0.6823
Epoch [149], train_loss: 0.3011, val_loss: 1.1108, val_acc: 0.7009
Epoch [150], train_loss: 0.2971, val_loss: 1.0753, val_acc: 0.7016
Epoch [151], train_loss: 0.2938, val_loss: 1.1510, val_acc: 0.6934
Epoch [152], train_loss: 0.2900, val_loss: 1.1219, val_acc: 0.6974
Epoch [153], train_loss: 0.2858, val_loss: 1.0989, val_acc: 0.7070
Epoch [154], train_loss: 0.2831, val_loss: 1.1212, val_acc: 0.6951
Epoch [155], train_loss: 0.2777, val_loss: 1.1183, val_acc: 0.7009
Epoch [156], train_loss: 0.2725, val_loss: 1.1371, val_acc: 0.6984
Epoch [157], train_loss: 0.2701, val_loss: 1.1462, val_acc: 0.6940
Epoch [158], train_loss: 0.2662, val_loss: 1.1479, val_acc: 0.6976
Epoch [159], train_loss: 0.2614, val_loss: 1.1341, val_acc: 0.6976
Epoch [160], train_loss: 0.2592, val_loss: 1.1250, val_acc: 0.7037
Epoch [161], train_loss: 0.2535, val_loss: 1.1585, val_acc: 0.6992
Epoch [162], train_loss: 0.2533, val_loss: 1.1557, val_acc: 0.6974
Epoch [163], train_loss: 0.2450, val_loss: 1.1576, val_acc: 0.6976
Epoch [164], train_loss: 0.2446, val_loss: 1.2208, val_acc: 0.6866
Epoch [165], train_loss: 0.2396, val_loss: 1.1664, val_acc: 0.6996
Epoch [166], train_loss: 0.2350, val_loss: 1.1692, val_acc: 0.6972
Epoch [167], train_loss: 0.2289, val_loss: 1.1764, val_acc: 0.6982
Epoch [168], train_loss: 0.2303, val_loss: 1.1852, val_acc: 0.7020
Epoch [169], train_loss: 0.2231, val_loss: 1.2610, val_acc: 0.6905
Epoch [170], train_loss: 0.2206, val_loss: 1.2736, val_acc: 0.6814
Epoch [171], train_loss: 0.2206, val_loss: 1.2237, val_acc: 0.6965
Epoch [172], train_loss: 0.2132, val_loss: 1.2373, val_acc: 0.6964
Epoch [173], train_loss: 0.2091, val_loss: 1.2464, val_acc: 0.6947
Epoch [174], train_loss: 0.2056, val_loss: 1.2565, val_acc: 0.6993
Epoch [175], train_loss: 0.2009, val_loss: 1.2257, val_acc: 0.7009
Epoch [176], train_loss: 0.1975, val_loss: 1.2355, val_acc: 0.6992
Epoch [177], train_loss: 0.1968, val_loss: 1.2709, val_acc: 0.6972
Epoch [178], train_loss: 0.1936, val_loss: 1.2599, val_acc: 0.6959
Epoch [179], train_loss: 0.1902, val_loss: 1.2712, val_acc: 0.7022
Epoch [180], train_loss: 0.1855, val_loss: 1.2697, val_acc: 0.6983
Epoch [181], train_loss: 0.1800, val_loss: 1.2903, val_acc: 0.6933
Epoch [182], train_loss: 0.1809, val_loss: 1.2785, val_acc: 0.6990
Epoch [183], train_loss: 0.1769, val_loss: 1.3437, val_acc: 0.6902
Epoch [184], train_loss: 0.1729, val_loss: 1.3036, val_acc: 0.6972
Epoch [185], train_loss: 0.1733, val_loss: 1.3229, val_acc: 0.6976
Epoch [186], train_loss: 0.1676, val_loss: 1.2986, val_acc: 0.7016
Epoch [187], train_loss: 0.1648, val_loss: 1.3260, val_acc: 0.6986
Epoch [188], train_loss: 0.1601, val_loss: 1.3361, val_acc: 0.7009
Epoch [189], train_loss: 0.1584, val_loss: 1.3178, val_acc: 0.7004
Epoch [190], train_loss: 0.1539, val_loss: 1.3436, val_acc: 0.6969
Epoch [191], train_loss: 0.1533, val_loss: 1.3473, val_acc: 0.6984
Epoch [192], train_loss: 0.1467, val_loss: 1.3609, val_acc: 0.6998
Epoch [193], train_loss: 0.1480, val_loss: 1.3525, val_acc: 0.7005
Epoch [194], train_loss: 0.1452, val_loss: 1.3901, val_acc: 0.6950
Epoch [195], train_loss: 0.1407, val_loss: 1.3704, val_acc: 0.6965
Epoch [196], train_loss: 0.1372, val_loss: 1.3972, val_acc: 0.6998
Epoch [197], train_loss: 0.1363, val_loss: 1.3924, val_acc: 0.6975
Epoch [198], train_loss: 0.1326, val_loss: 1.4298, val_acc: 0.6929
Epoch [199], train_loss: 0.1279, val_loss: 1.4106, val_acc: 0.6979
```

In [2]:
```
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');
```

In [3]:
```
plot_accuracies(history)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_3872\2884717789.py in <module>
----> 1 plot_accuracies(history)

NameError: name 'history' is not defined
```

이 모델은 약 75% 이상 정도의 정확도에 도달하였고, epoch을 더 돌린다 하더라도 정확도가 올라갈 것 같지 않다.

In [39]:
```
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');
```

In [40]:
```
plot_losses(history)
```

처음에는 훈련과 검증 모두 손실이 낮아지다가 epoch 3부터는 훈련손실만 더 낮아지고 검증set 손실은 더 높아지는것을 확인할 수 있다. -> 과적합!

이 과적합을 피하기 위해 "노이즈"를 추가한다. 배치 정규화 및 드롭아웃과 같은 정규화 기술 사용을 통해.

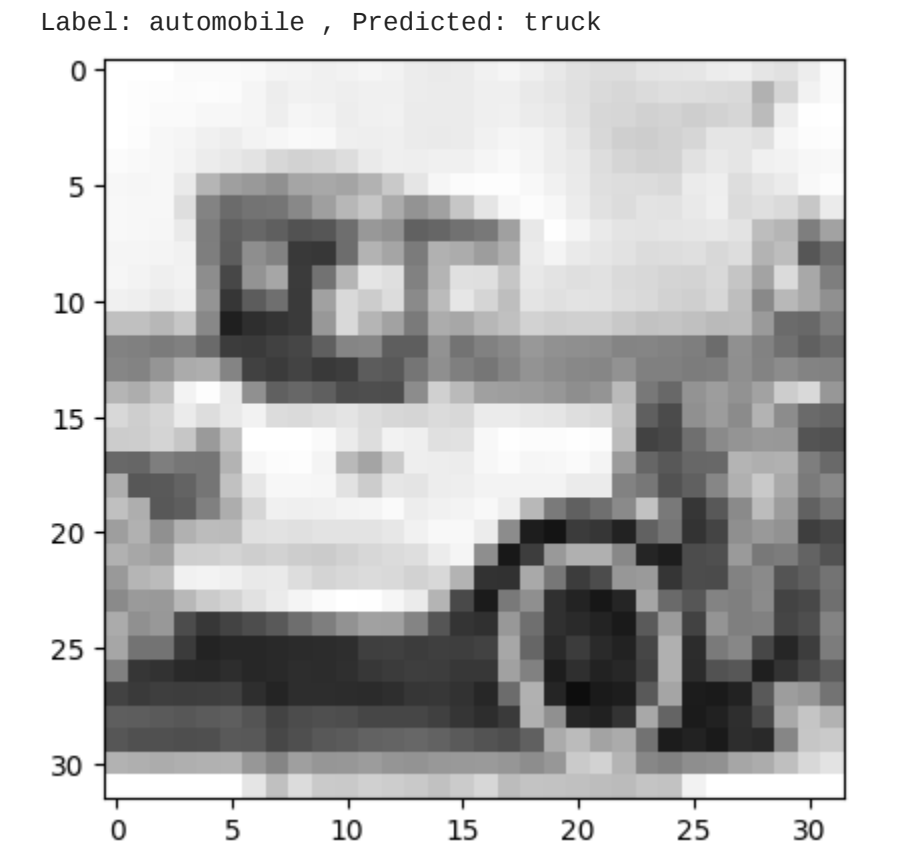
Testiong with individual images

In [41]:
```
test_dataset = ImageFolder(data_dir+'/test', transform=ToTensor())
```

In [42]:
```
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()]
```

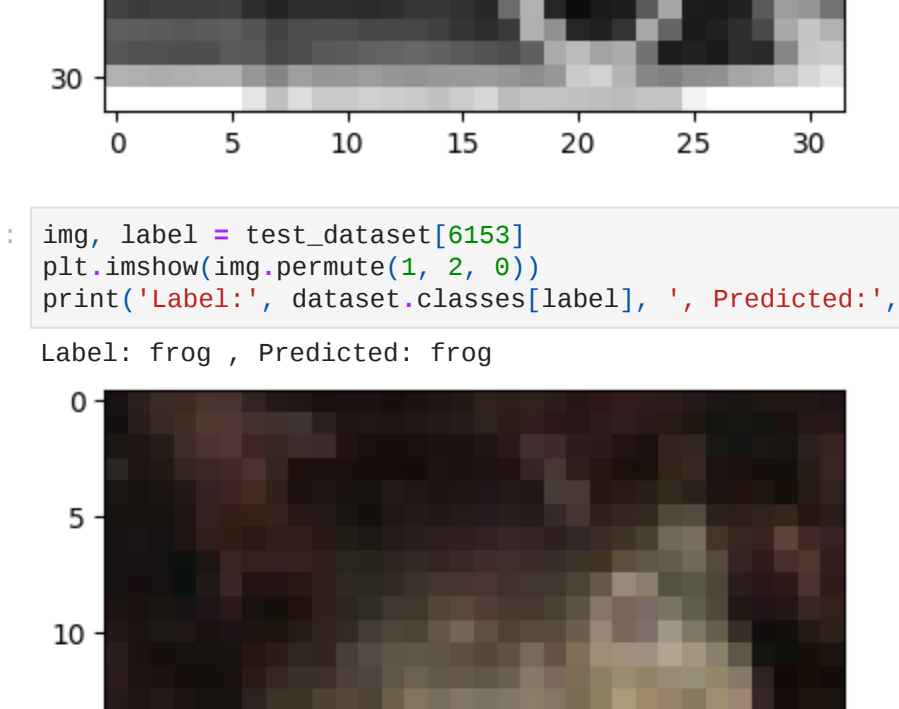
In [43]:
```
img, label = test_dataset[0]
plt.imshow(img.permute(1, 2, 0))
print('Label:', dataset.classes[label], ', Predicted:', predict_image(img, model))
```

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



In [44]:
```
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: truck
```



In [45]:
```
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
```

In [46]:
```
test_loader = DeviceDataLoader(DataLoader(test_dataset, batch_size*2), device)
result = evaluate(model, test_loader)
result
```

Out[46]:
```
{'val_loss': 1.4161564111709595, 'val_acc': 0.6883559226989746}
```

Saving and loading the model

In [ ]:
```
torch.save(model.state_dict(), 'cifar10-cnn-acc0.7637-epoch200-lr0.001-adagrad-bs256.pth')
```