# Classifying images of everyday objects using a neural network

The ability to try many different neural network architectures to address a problem is what makes deep learning really powerful, especially compared to shallow learning techniques like linear regression, logistic regression etc.

In this assignment, you will:

1. Explore the CIFAR10 dataset: https://www.cs.toronto.edu/~kriz/cifar.html
2. Set up a training pipeline to train a neural network on a GPU
3. Experiment with different network architectures & hyperparameters

As you go through this notebook, you will find a **???** in certain places. Your job is to replace the **???** with appropriate code or values, to ensure that the notebook runs properly end-to-end. Try to experiment with different network structures and hypeparameters to get the lowest loss.

You might find these notebooks useful for reference, as you work through this notebook:

- https://jovian.ml/aakashns/04-feedforward-nn
- https://jovian.ml/aakashns/fashion-feedforward-minimal

```
# Uncomment and run the commands below if imports fail
# !conda install numpy pandas pytorch torchvision cpuonly -c pytorch -y
# !pip install matplotlib --upgrade --quiet
```

```
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
```

```
# Project name used for jovian.commit
project_name = '03-cifar10-feedforward'
```

## Exploring the CIFAR10 dataset

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

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to data/cifar-10-
```

```
python.tar.gz
  0%|          | 0/170498071 [00:00<?, ?it/s]
Extracting data/cifar-10-python.tar.gz to data/
```

**Q: How many images does the training dataset contain?**

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

```
50000
```

**Q: How many images does the test dataset contain?**

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

```
10000
```

**Q: How many output classes does the dataset contain? Can you list them?**

Hint: Use `dataset.classes`

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

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

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

```
10
```

**Q: What is the shape of an image tensor from the dataset?**

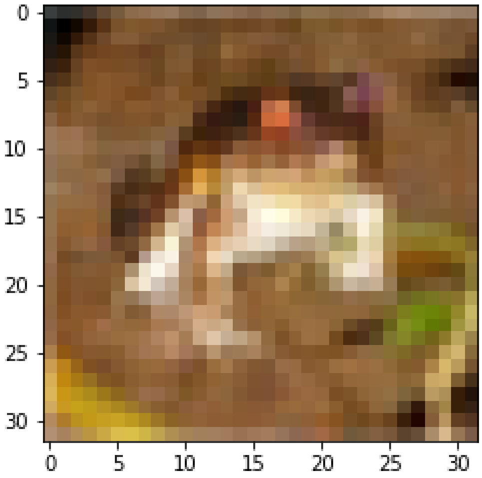
```
img, label = dataset[0]
img_shape = img.shape
img_shape
```

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

Note that this dataset consists of 3-channel color images (RGB). Let us look at a sample image from the dataset. `matplotlib` expects channels to be the last dimension of the image tensors (whereas in PyTorch they are the first dimension), so we'll the `.permute` tensor method to shift channels to the last dimension. Let's also print the label for the image.

```
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
```




**(Optional) Q: Can you determine the number of images belonging to each class?**

Hint: Loop through the dataset.

```
img_class = dict()
for img, label in dataset:
  if classes[label] not in img_class:
    img_class[classes[label]]=1
  else:
    img_class[classes[label]]+=1

img_class
```

```
{'airplane': 5000,
 'automobile': 5000,
 'bird': 5000,
 'cat': 5000,
 'deer': 5000,
 'dog': 5000,
 'frog': 5000,
 'horse': 5000,
 'ship': 5000,
 'truck': 5000}
```

Let's save our work to Jovian, before continuing.

```
!pip install jovian --upgrade --quiet
```

```
import jovian
```

```
jovian.commit(project=project_name, environment=None)
```

```
[jovian] Detected Colab notebook...
[jovian] Please enter your API key ( from https://jovian.ai/ ):
API KEY: ··········
[jovian] Uploading colab notebook to Jovian...
Committed successfully! https://jovian.ai/tannu945/03-cifar10-feedforward
```

```
'https://jovian.ai/tannu945/03-cifar10-feedforward'
```

## Preparing the data for training

We'll use a validation set with 5000 images (10% of the dataset). To ensure we get the same validation set each time, we'll set PyTorch's random number generator to a seed value of 43.

```
torch.manual_seed(43)
val_size = 5000
train_size = len(dataset) - val_size
```

Let's use the `random_split` method to create the training & validation sets

```
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 to load the data in batches.

```
batch_size=128
```

```
train_loader = DataLoader(train_ds, batch_size, shuffle=True, num_workers=4, pin_memory
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)
```

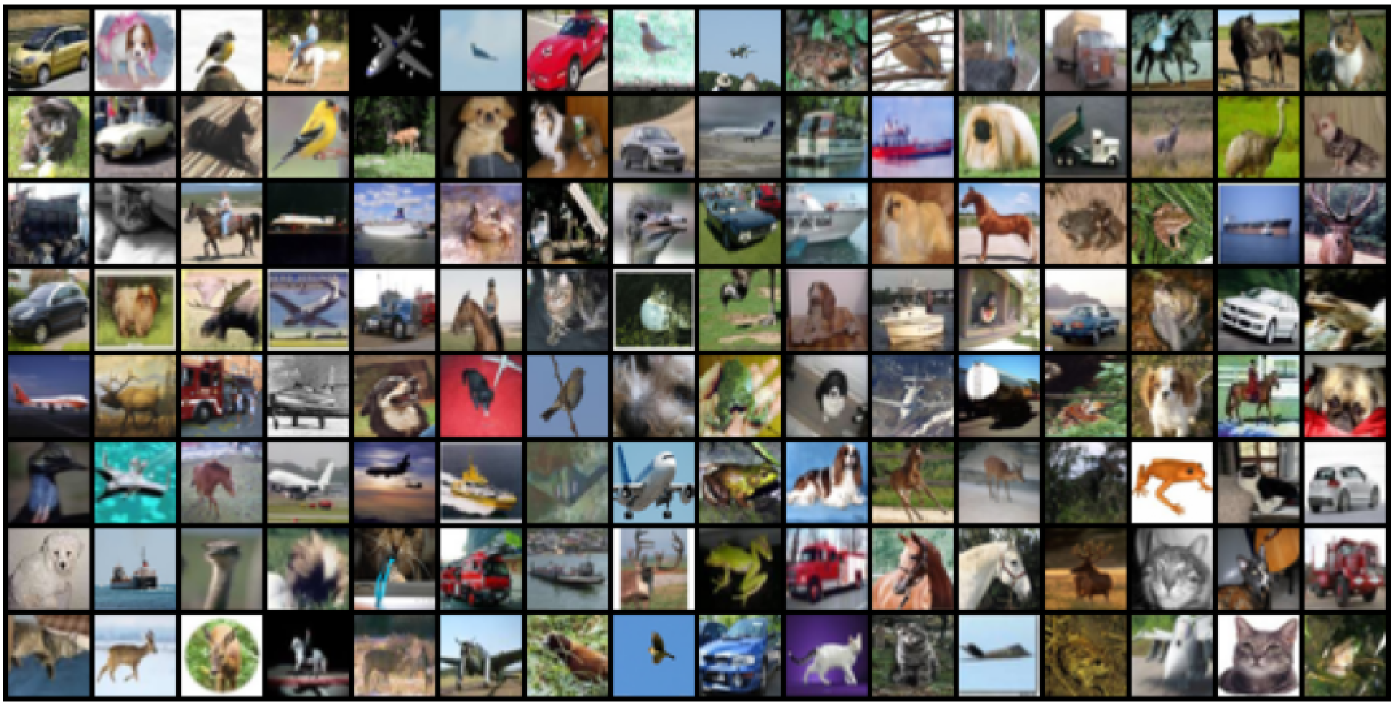
```
/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:490: UserWarning:
This DataLoader will create 4 worker processes in total. Our suggested max number of
worker in current system is 2, which is smaller than what this DataLoader is going to
create. Please be aware that excessive worker creation might get DataLoader running
slow or even freeze, lower the worker number to avoid potential slowness/freeze if
necessary.
  cpuset_checked))
```

Let's visualize a batch of data using the `make_grid` helper function from Torchvision.

```
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
```

```
/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:490: UserWarning:
This DataLoader will create 4 worker processes in total. Our suggested max number of
worker in current system is 2, which is smaller than what this DataLoader is going to
create. Please be aware that excessive worker creation might get DataLoader running
slow or even freeze, lower the worker number to avoid potential slowness/freeze if
necessary.
  cpuset_checked))
```

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



Can you label all the images by looking at them? Trying to label a random sample of the data manually is a good way to estimate the difficulty of the problem, and identify errors in labeling, if any.

## Base Model class & Training on GPU

Let's create a base model class, which contains everything except the model architecture i.e. it wil not contain the `__init__` and `__forward__` methods. We will later extend this class to try out different architectures. In fact, you can extend this model to solve any image classification problem.

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

```python
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 [{}], val_loss: {:.4f}, val_acc: {:.4f}".format(epoch, result['val
```

We can also use the exact same training loop as before. I hope you're starting to see the benefits of refactoring our code into reusable functions.

```python
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
        for batch in train_loader:
            loss = model.training_step(batch)
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()
        # Validation phase
        result = evaluate(model, val_loader)
        model.epoch_end(epoch, result)
        history.append(result)
    return history
```

Finally, let's also define some utilities for moving out data & labels to the GPU, if one is available.

```python
torch.cuda.is_available()
```

```
True
```

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

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

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

```python
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)
```

Let us also define a couple of helper functions for plotting the losses & accuracies.

```python
def plot_losses(history):
    losses = [x['val_loss'] for x in history]
    plt.plot(losses, '-x')
    plt.xlabel('epoch')
    plt.ylabel('loss')
    plt.title('Loss vs. No. of epochs');
```

```python
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');
```

Let's move our data loaders to the appropriate device.

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

# Training the model

We will make several attempts at training the model. Each time, try a different architecture and a different set of learning rates. Here are some ideas to try:

- Increase or decrease the number of hidden layers
- Increase of decrease the size of each hidden layer
- Try different activation functions
- Try training for different number of epochs
- Try different learning rates in every epoch

What's the highest validation accuracy you can get to? **Can you get to 50% accuracy? What about 60%?**

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

**Q: Extend the `ImageClassificationBase` class to complete the model definition.**

Hint: Define the `__init__` and `forward` methods.

```
class CIFAR10Model(ImageClassificationBase):
    def __init__(self):
        super().__init__()
        self.linear1 = nn.Linear(input_size, 512)
        # Hidden layers
        self.linear2 = nn.Linear(512, 256)
        self.linear3 = nn.Linear(256, 128)
        self.linear4 = nn.Linear(128, 64)
        self.linear5 = nn.Linear(64, 32)
        # Output layer
        self.linear6 = nn.Linear(32, output_size)

    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.relu(out)

        return out
```

You can now instantiate the model, and move it the appropriate device.

```
model = to_device(CIFAR10Model(), device)
```

Before you train the model, it's a good idea to check the validation loss & accuracy with the initial set of weights.

```
history = [evaluate(model, val_loader)]
history
```

```
/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:490: UserWarning: This DataLoader will create 4 worker processes in total. Our suggested max number of worker in current system is 2, which is smaller than what this DataLoader is going to create. Please be aware that excessive worker creation might get DataLoader running slow or even freeze, lower the worker number to avoid potential slowness/freeze if necessary.
  cpuset_checked))
```

```
[{'val_acc': 0.09564568102359772, 'val_loss': 2.3040597438812256}]
```

**Q: Train the model using the `fit` function to reduce the validation loss & improve accuracy.**

Leverage the interactive nature of Jupyter to train the model in multiple phases, adjusting the no. of epochs & learning rate each time based on the result of the previous training phase.

```
history += fit(10, 0.15, model, train_loader, val_loader)
```

```
/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:490: UserWarning: This DataLoader will create 4 worker processes in total. Our suggested max number of worker in current system is 2, which is smaller than what this DataLoader is going to create. Please be aware that excessive worker creation might get DataLoader running slow or even freeze, lower the worker number to avoid potential slowness/freeze if necessary.
  cpuset_checked))
```

```
Epoch [0], val_loss: 2.2033, val_acc: 0.1660
Epoch [1], val_loss: 2.1011, val_acc: 0.2387
Epoch [2], val_loss: 2.1332, val_acc: 0.2484
Epoch [3], val_loss: 1.9815, val_acc: 0.3192
```

```
Epoch [4], val_loss: 1.9652, val_acc: 0.3228
Epoch [5], val_loss: 2.0434, val_acc: 0.3185
Epoch [6], val_loss: 1.8378, val_acc: 0.3796
Epoch [7], val_loss: 1.8119, val_acc: 0.3995
Epoch [8], val_loss: 1.7429, val_acc: 0.4220
Epoch [9], val_loss: 1.9108, val_acc: 0.3668
```

```
history += fit(20, 0.1, model, train_loader, val_loader)
```

```
/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:490: UserWarning:
This DataLoader will create 4 worker processes in total. Our suggested max number of
worker in current system is 2, which is smaller than what this DataLoader is going to
create. Please be aware that excessive worker creation might get DataLoader running
slow or even freeze, lower the worker number to avoid potential slowness/freeze if
necessary.
  cpuset_checked))
```

```
Epoch [0], val_loss: 1.7957, val_acc: 0.4072
Epoch [1], val_loss: 1.7322, val_acc: 0.4233
Epoch [2], val_loss: 1.7349, val_acc: 0.4201
Epoch [3], val_loss: 1.8997, val_acc: 0.3861
Epoch [4], val_loss: 1.6685, val_acc: 0.4482
Epoch [5], val_loss: 1.7538, val_acc: 0.4189
Epoch [6], val_loss: 1.6843, val_acc: 0.4464
Epoch [7], val_loss: 1.6700, val_acc: 0.4261
Epoch [8], val_loss: 1.6861, val_acc: 0.4232
Epoch [9], val_loss: 1.6573, val_acc: 0.4276
Epoch [10], val_loss: 1.6343, val_acc: 0.4511
Epoch [11], val_loss: 1.6176, val_acc: 0.4477
Epoch [12], val_loss: 1.6209, val_acc: 0.4583
Epoch [13], val_loss: 1.6276, val_acc: 0.4455
Epoch [14], val_loss: 1.5902, val_acc: 0.4665
Epoch [15], val_loss: 1.6731, val_acc: 0.4412
Epoch [16], val_loss: 1.6282, val_acc: 0.4735
Epoch [17], val_loss: 1.6278, val_acc: 0.4592
Epoch [18], val_loss: 1.5480, val_acc: 0.4849
Epoch [19], val_loss: 1.5933, val_acc: 0.4804
```

```
history += fit(10, 0.001, model, train_loader, val_loader)
```

```
/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:490: UserWarning:
This DataLoader will create 4 worker processes in total. Our suggested max number of
worker in current system is 2, which is smaller than what this DataLoader is going to
create. Please be aware that excessive worker creation might get DataLoader running
slow or even freeze, lower the worker number to avoid potential slowness/freeze if
```

```
necessary.
  cpuset_checked))
```

```
Epoch [0], val_loss: 1.5450, val_acc: 0.5177
Epoch [1], val_loss: 1.5474, val_acc: 0.5156
Epoch [2], val_loss: 1.5473, val_acc: 0.5160
Epoch [3], val_loss: 1.5477, val_acc: 0.5158
Epoch [4], val_loss: 1.5537, val_acc: 0.5164
Epoch [5], val_loss: 1.5524, val_acc: 0.5162
Epoch [6], val_loss: 1.5518, val_acc: 0.5177
Epoch [7], val_loss: 1.5527, val_acc: 0.5170
Epoch [8], val_loss: 1.5555, val_acc: 0.5160
Epoch [9], val_loss: 1.5578, val_acc: 0.5169
```

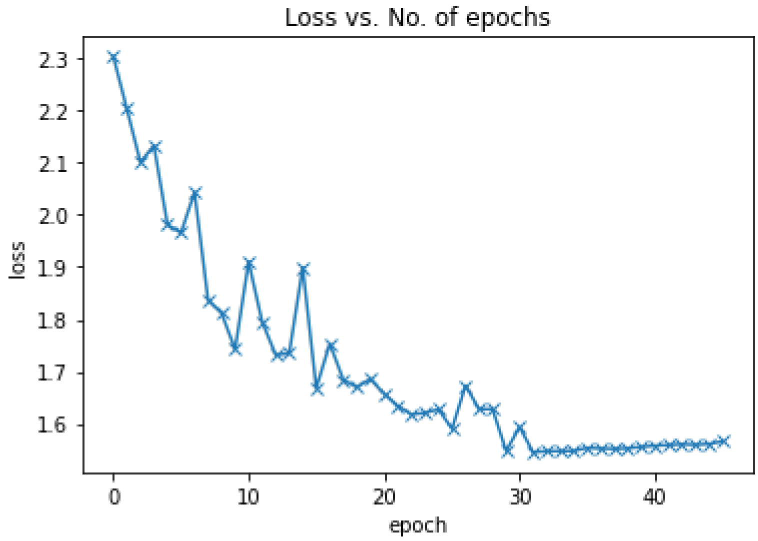
```
history += fit(5, 0.001, model, train_loader, val_loader)
```

```
/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:490: UserWarning:
This DataLoader will create 4 worker processes in total. Our suggested max number of
worker in current system is 2, which is smaller than what this DataLoader is going to
create. Please be aware that excessive worker creation might get DataLoader running
slow or even freeze, lower the worker number to avoid potential slowness/freeze if
necessary.
  cpuset_checked))
```

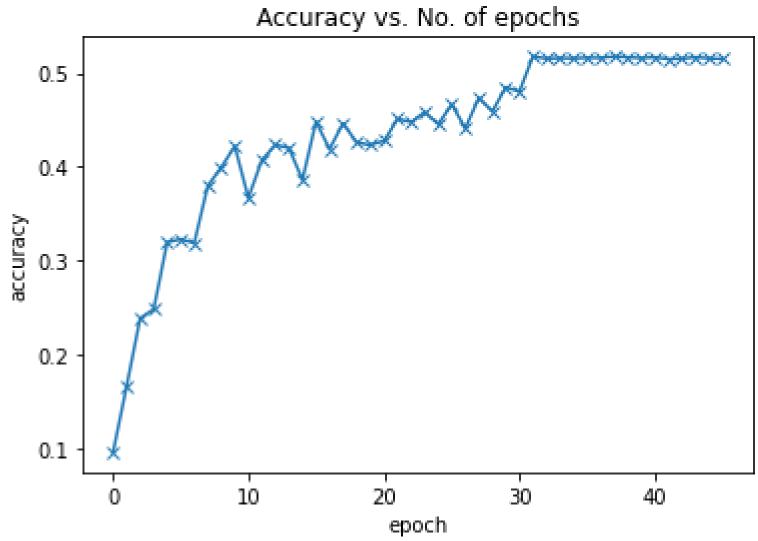
```
Epoch [0], val_loss: 1.5590, val_acc: 0.5150
Epoch [1], val_loss: 1.5607, val_acc: 0.5154
Epoch [2], val_loss: 1.5596, val_acc: 0.5170
Epoch [3], val_loss: 1.5612, val_acc: 0.5158
Epoch [4], val_loss: 1.5662, val_acc: 0.5152
```

Plot the losses and the accuracies to check if you're starting to hit the limits of how well your model can perform on this dataset. You can train some more if you can see the scope for further improvement.

```
plot_losses(history)
```




```
plot_accuracies(history)
```




Finally, evaluate the model on the test dataset report its final performance.

```
evaluate(model, test_loader)
```

```
/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:490: UserWarning: This DataLoader will create 4 worker processes in total. Our suggested max number of worker in current system is 2, which is smaller than what this DataLoader is going to create. Please be aware that excessive worker creation might get DataLoader running slow or even freeze, lower the worker number to avoid potential slowness/freeze if necessary.
  cpuset_checked))
```

```
{'val_acc': 0.5218750238418579, 'val_loss': 1.5275291204452515}
```

Are you happy with the accuracy? Record your results by completing the section below, then you can come back and try a different architecture & hyperparameters.

# Recoding your results

As your perform multiple experiments, it's important to record the results in a systematic fashion, so that you can review them later and identify the best approaches that you might want to reproduce or build upon later.

**Q: Describe the model's architecture with a short summary.**

E.g. `"3 layers (16,32,10)"` (16, 32 and 10 represent output sizes of each layer)

```
arch = "4 layers (128, 64, 32, 10)"
```

**Q: Provide the list of learning rates used while training.**

```
lrs = [0.15, 0.1, 0.01, 0.001]
```

**Q: Provide the list of no. of epochs used while training.**

```
epochs = [10, 20, 20, 5]
```

**Q: What were the final test accuracy & test loss?**

```
evalu1 = evaluate(model, val_loader)
test_acc = evalu1['val_acc']
test_loss = evalu1['val_loss']
```

```
/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:490: UserWarning:
This DataLoader will create 4 worker processes in total. Our suggested max number of
worker in current system is 2, which is smaller than what this DataLoader is going to
create. Please be aware that excessive worker creation might get DataLoader running
slow or even freeze, lower the worker number to avoid potential slowness/freeze if
necessary.
  cpuset_checked))
```

Finally, let's save the trained model weights to disk, so we can use this model later.

```
torch.save(model.state_dict(), 'cifar10-feedforward.pth')
```

The `jovian` library provides some utility functions to keep your work organized. With every version of your notebok, you can attach some hyperparameters and metrics from your experiment.

```
# Clear previously recorded hyperparams & metrics
jovian.reset()
```

```
jovian.log_hyperparams(arch=arch,
                       lrs=lrs,
                       epochs=epochs)
```

```
jovian.log_metrics(test_loss=test_loss, test_acc=test_acc)
```

Finally, we can commit the notebook to Jovian, attaching the hypeparameters, metrics and the trained model weights.

```
jovian.commit(project=project_name, outputs=['cifar10-feedforward.pth'], environment=No
```

Once committed, you can find the recorded metrics & hyperprameters in the "Records" tab on Jovian. You can find the saved model weights in the "Files" tab.

# Continued experimentation

Now go back up to the "**Training the model**" section, and try another network architecture with a different set of hyperparameters. As you try different experiments, you will start to build an undestanding of how the different architectures & hyperparameters affect the final result. Don't worry if you can't get to very high accuracy, we'll make some fundamental changes to our model in the next lecture.

Once you have tried multiple experiments, you can compare your results using the "**Compare**" button on Jovian.

Compare Versions

View Diff | Filter | Configure

| ID | Title | Created | Author | Hyperparameters | | | Metrics | | | | Notes |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | epochs | lr | opt | acc ↓ | loss | val_acc | val_loss | |
| 7 | Resnet18 | 1 year ago | init27 | 2 | 0.005 | | 0.9826 | 0.0558 | 0.9788 | 0.0683 | |
| 10 | Efficient Net 18 | 7 months ago | aakashns | 1 | 0.001 | RMSprop | 0.9783 | 0.0748 | 0.9759 | 0.0814 | deployed to prod |
| 11 | Version 11 | 7 months ago | aakashns | 1 | 0.001 | RMSprop | 0.9783 | 0.0748 | 0.9759 | 0.0814 | early stopping |
| 12 | Version 12 | 7 months ago | aakashns | 1 | 0.001 | RMSprop | 0.9783 | 0.0748 | 0.9759 | 0.0814 | |
| 13 | Version 13 | 7 months ago | aakashns | 1 | 0.001 | RMSprop | 0.9783 | 0.0748 | 0.9759 | 0.0814 | |
| 5 | 2-layer linear | 1 year ago | donjhoe | 2 | 0.01 | | 0.8964 | 0.3482 | 0.9227 | 0.2542 | simple w/ dropout |

# (Optional) Write a blog post

Writing a blog post is the best way to further improve your understanding of deep learning & model training, because it forces you to articulate your thoughts clearly. Here'are some ideas for a blog post:

- Report the results given by different architectures on the CIFAR10 dataset
- Apply this training pipeline to a different dataset (it doesn't have to be images, or a classification problem)
- Improve upon your model from Assignment 2 using a feedfoward neural network, and write a sequel to your previous blog post
- Share some Strategies for picking good hyperparameters for deep learning
- Present a summary of the different steps involved in training a deep learning model with PyTorch
- Implement the same model using a different deep learning library e.g. Keras ( https://keras.io/ ), and present a comparision.