In [2]:
```
import torch
import torchvision
import torch.nn as nn  # All neural network modules, nn.Linear, nn.Conv2d, BatchNorm, Loss functions
import torch.optim as optim  # For all Optimization algorithms, SGD, Adam, etc.
import torch.nn.functional as F  # All functions that don't have any parameters
from torch.utils.data import (
    DataLoader,
)  # Gives easier dataset managment and creates mini batches
import torchvision.datasets as datasets  # Has standard datasets we can import in a nice way
import torchvision.transforms as transforms  # Transformations we can perform on our dataset
```

In [3]:
```
# Set device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Hyperparameters
num_classes = 10
learning_rate = 1e-3
batch_size = 1024
num_epochs = 50
```

In [4]:
```
# Simple Identity class that let's input pass without changes
class Identity(nn.Module):
    def __init__(self):
        super(Identity, self).__init__()

    def forward(self, x):
        return x
```

1. Load in a pretrained model (VGG16)

In [5]:
```
model = torchvision.models.vgg16(pretrained=True)
```

```
/home/ruturajpatil/.local/lib/python3.10/site-packages/torchvision/models/_utils.py:208: UserWarning: The parameter 'pr
etrained' is deprecated since 0.13 and may be removed in the future, please use 'weights' instead.
  warnings.warn(
/home/ruturajpatil/.local/lib/python3.10/site-packages/torchvision/models/_utils.py:223: UserWarning: Arguments other t
han a weight enum or `None` for 'weights' are deprecated since 0.13 and may be removed in the future. The current behav
ior is equivalent to passing `weights=VGG16_Weights.IMAGENET1K_V1`. You can also use `weights=VGG16_Weights.DEFAULT` to
get the most up-to-date weights.
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /home/ruturajpatil/.cache/torch/hub/checkpoint
s/vgg16-397923af.pth
100.0%
```

In [6]:
```
print(model)
```

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (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(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (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(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=True)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace=True)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=True)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace=True)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace=True)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace=True)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
```

```
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

2. Freezing parameters in model's lower layers

In [7]:
```
# If you want to do finetuning then set requires_grad = False
for param in model.parameters():
    param.requires_grad = False
```

In [8]:
```
## Freezing the average pool layer of the model and add a custom classifier
model.avgpool = Identity()
```

3. Add custom classifier with several layers of trainable parameters to mode

In [9]:
```
model.classifier = nn.Sequential(
    nn.Linear(512, 100), nn.ReLU(),
    nn.Linear(100, num_classes)
)
model.to(device)
```

Out[9]:
```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (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(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (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(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=True)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace=True)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=True)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace=True)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace=True)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace=True)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): Identity()
```

```
  (classifier): Sequential(
    (0): Linear(in_features=512, out_features=100, bias=True)
    (1): ReLU()
    (2): Linear(in_features=100, out_features=10, bias=True)
  )
)
```

4. Train classifier layers on training data available for task

In [10]:
```
# Load Data
train_dataset = datasets.CIFAR10(
    root="dataset/", train=True, transform=transforms.ToTensor(), download=True
)
train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True)
```

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

100.0%

Extracting dataset/cifar-10-python.tar.gz to dataset/
```

In [11]:
```
# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
```

In [13]:
```
# Train Network
for epoch in range(num_epochs):
    losses = []

    for batch_idx, (data, targets) in enumerate(train_loader):
        # Get data to cuda if possible
        data = data.to(device=device)
        targets = targets.to(device=device)

        # forward
        scores = model(data)
        loss = criterion(scores, targets)

        losses.append(loss.item())
        # backward
        optimizer.zero_grad()
        loss.backward()

        # gradient descent or adam step
        optimizer.step()

    print(f"Cost at epoch {epoch} is {sum(losses)/len(losses):.5f}")
```

```
Cost at epoch 0 is 0.74452
Cost at epoch 1 is 0.73771
Cost at epoch 2 is 0.73279
Cost at epoch 3 is 0.72999
Cost at epoch 4 is 0.72231
Cost at epoch 5 is 0.72027
Cost at epoch 6 is 0.71365
Cost at epoch 7 is 0.70845
Cost at epoch 8 is 0.70504
Cost at epoch 9 is 0.70078
Cost at epoch 10 is 0.69512
Cost at epoch 11 is 0.68973
Cost at epoch 12 is 0.68553
Cost at epoch 13 is 0.68230
Cost at epoch 14 is 0.67616
Cost at epoch 15 is 0.67199
Cost at epoch 16 is 0.66721
```

```
Cost at epoch 17 is 0.66456
Cost at epoch 18 is 0.65900
Cost at epoch 19 is 0.65321
Cost at epoch 20 is 0.64894
Cost at epoch 21 is 0.64827
Cost at epoch 22 is 0.64559
Cost at epoch 23 is 0.63880
Cost at epoch 24 is 0.63768
Cost at epoch 25 is 0.63080
Cost at epoch 26 is 0.62703
Cost at epoch 27 is 0.62392
Cost at epoch 28 is 0.62043
Cost at epoch 29 is 0.61719
Cost at epoch 30 is 0.61401
Cost at epoch 31 is 0.61015
Cost at epoch 32 is 0.60704
Cost at epoch 33 is 0.60256
Cost at epoch 34 is 0.60116
Cost at epoch 35 is 0.59558
Cost at epoch 36 is 0.59220
Cost at epoch 37 is 0.59090
Cost at epoch 38 is 0.58601
Cost at epoch 39 is 0.58393
Cost at epoch 40 is 0.57806
Cost at epoch 41 is 0.57598
Cost at epoch 42 is 0.57466
Cost at epoch 43 is 0.57111
Cost at epoch 44 is 0.56705
Cost at epoch 45 is 0.56460
Cost at epoch 46 is 0.56366
Cost at epoch 47 is 0.55820
Cost at epoch 48 is 0.55514
Cost at epoch 49 is 0.55108
```

5. Checking accuracy and fine tuning if required.

In [14]:
```
def check_accuracy(loader, model):
    if loader.dataset.train:
        print("Checking accuracy on training data")
    else:
        print("Checking accuracy on test data")

    num_correct = 0
    num_samples = 0
    model.eval()

    with torch.no_grad():
        for x, y in loader:
            x = x.to(device=device)
            y = y.to(device=device)

            scores = model(x)
            _, predictions = scores.max(1)
            num_correct += (predictions == y).sum()
            num_samples += predictions.size(0)

        print(
            f"Got {num_correct} / {num_samples} with accuracy {float(num_correct)/float(num_samples)*100:.2f}"
        )

    model.train()
```

In [ ]:
```
check_accuracy(train_loader, model)
```

```
Checking accuracy on training data
```

In [ ]:
```

```