```
# Jovian Commit Essentials
# Please retain and execute this cell without modifying the contents for `jovian.commit` to work
!pip install jovian --upgrade -q
import jovian
jovian.utils.colab.set_colab_file_id('1ne-RJzBRocbFGgV8hgQmwfJtbX3FBlN7')
```

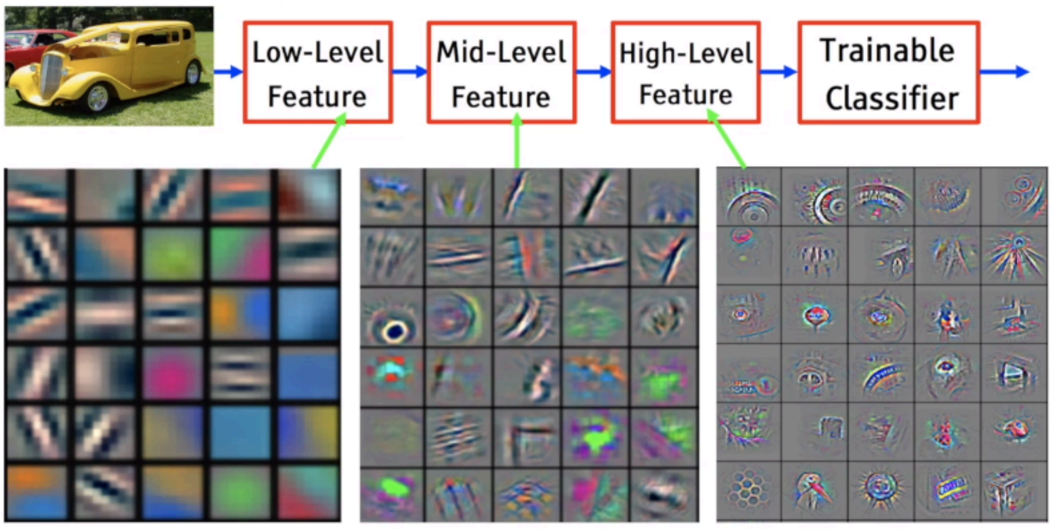
```
     |████████████████████████████████| 71kB 7.1MB/s
  Building wheel for uuid (setup.py) ... done
```

Transfer Learning for Image Classification in PyTorch

How a CNN learns (source):

cnn-learning

Layer visualization (source):



Downloading the Dataset

We'll use the Oxford-IIIT Pets dataset from https://course.fast.ai/datasets . It is 37 category (breeds) pet dataset with roughly 200 images for each class. The images have a large variations in scale, pose and lighting.

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

```
from torchvision.datasets.utils import download_url
```

```
download_url('https://s3.amazonaws.com/fast-ai-imageclas/oxford-iiit-pet.tgz', '.')
```

```
Downloading https://s3.amazonaws.com/fast-ai-imageclas/oxford-iiit-pet.tgz to ./oxford-iiit-pet.tgz
811712512/? [00:40<00:00, 37677294.82it/s]
```

```
import tarfile

with tarfile.open('./oxford-iiit-pet.tgz', 'r:gz') as tar:
    tar.extractall(path='./data')
```

```
from torch.utils.data import Dataset
```

```
import os

DATA_DIR = './data/oxford-iiit-pet/images'

files = os.listdir(DATA_DIR)
files[:5]
```

```
['Russian_Blue_130.jpg',
 'leonberger_103.jpg',
 'samoyed_197.jpg',
 'Russian_Blue_187.jpg',
 'Maine_Coon_56.jpg']
```

```
def parse_breed(fname):
    parts = fname.split('_')
    return ' '.join(parts[:-1])
```

```
parse_breed(files[4])
```

```
'Maine Coon'
```

```
from PIL import Image

def open_image(path):
    with open(path, 'rb') as f:
        img = Image.open(f)
        return img.convert('RGB')
```

Creating a Custom PyTorch Dataset

```
import os

class PetsDataset(Dataset):
    def __init__(self, root, transform):
        super().__init__()
        self.root = root
        self.files = [fname for fname in os.listdir(root) if fname.endswith('.jpg')]
        self.classes = list(set(parse_breed(fname) for fname in files))
        self.transform = transform

    def __len__(self):
        return len(self.files)

    def __getitem__(self, i):
        fname = self.files[i]
        fpath = os.path.join(self.root, fname)
        img = self.transform(open_image(fpath))
        class_idx = self.classes.index(parse_breed(fname))
        return img, class_idx
```

```
import torchvision.transforms as T

img_size = 224
imagenet_stats = ([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
dataset = PetsDataset(DATA_DIR, T.Compose([T.Resize(img_size),
                                           T.Pad(8, padding_mode='reflect'),
                                           T.RandomCrop(img_size),
                                           T.ToTensor(),
                                           T.Normalize(*imagenet_stats)]))
```

```
len(dataset)
```

```
7390
```

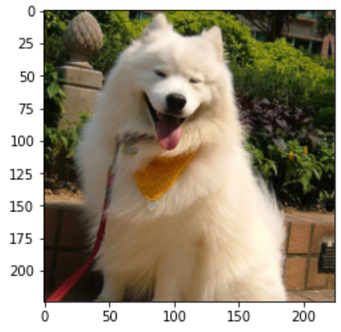
```
import torch
import matplotlib.pyplot as plt
%matplotlib inline

def denormalize(images, means, stds):
    if len(images.shape) == 3:
        images = images.unsqueeze(0)
    means = torch.tensor(means).reshape(1, 3, 1, 1)
    stds = torch.tensor(stds).reshape(1, 3, 1, 1)
    return images * stds + means

def show_image(img_tensor, label):
    print('Label:', dataset.classes[label], '(' + str(label) + ')')
    img_tensor = denormalize(img_tensor, *imagenet_stats)[0].permute((1, 2, 0))
    plt.imshow(img_tensor)
```

```
show_image(*dataset[2])
```

```
Label: samoyed (5)
```



Creating Training and Validation Sets

```
from torch.utils.data import random_split
```

```
val_pct = 0.1
val_size = int(val_pct * len(dataset))

train_ds, valid_ds = random_split(dataset, [len(dataset) - val_size, val_size])
```

```
from torch.utils.data import DataLoader
batch_size = 256

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

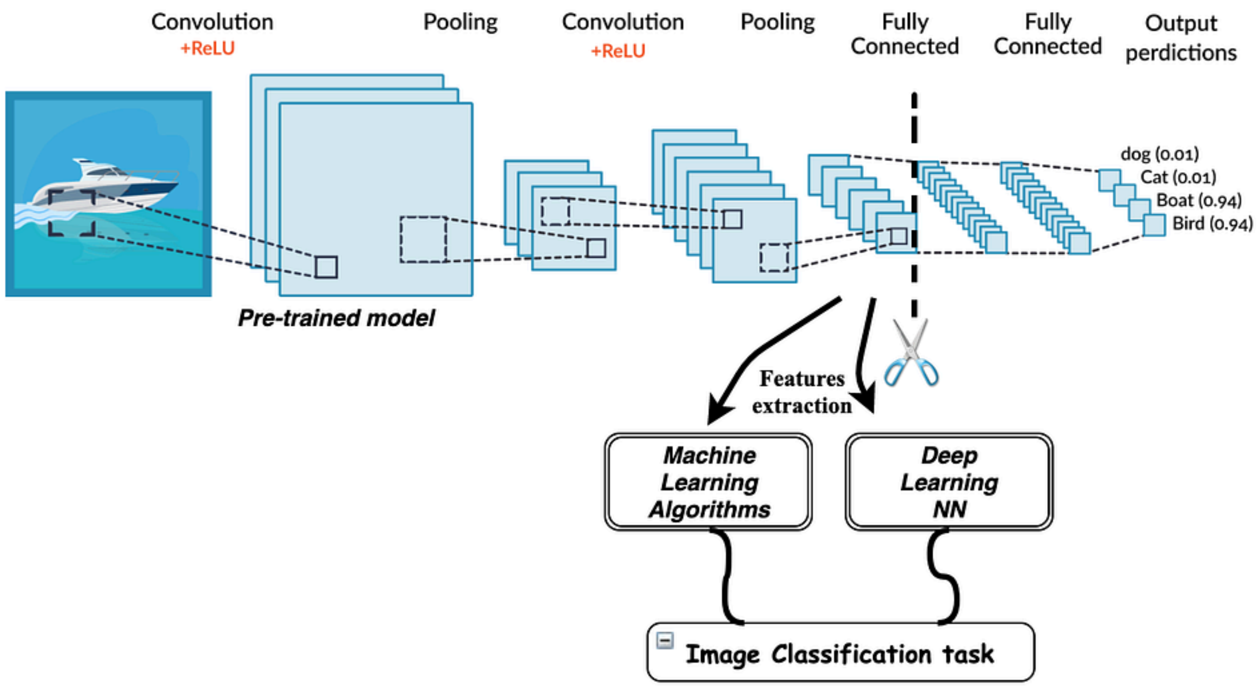
```
from torchvision.utils import make_grid

def show_batch(dl):
    for images, labels in dl:
        fig, ax = plt.subplots(figsize=(16, 16))
        ax.set_xticks([]); ax.set_yticks([])
        images = denormalize(images[:64], *imagenet_stats)
        ax.imshow(make_grid(images, nrow=8).permute(1, 2, 0))
        break
```

```
show_batch(train_dl)
```

Modifying a Pretrained Model (ResNet34)

Transfer learning (source):



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

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


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, "last_lr: {:.5f},".format(result['lrs'][-1]) if 'lrs' in result else '',
            result['train_loss'], result['val_loss'], result['val_acc']))
```

```
from torchvision import models

class PetsModel(ImageClassificationBase):
    def __init__(self, num_classes, pretrained=True):
        super().__init__()
        # Use a pretrained model
        self.network = models.resnet34(pretrained=pretrained)
        # Replace last layer
        self.network.fc = nn.Linear(self.network.fc.in_features, num_classes)

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

GPU Utilities and Training Loop

```
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)
```

```
import torch
from tqdm.notebook import tqdm

@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 tqdm(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

def get_lr(optimizer):
    for param_group in optimizer.param_groups:
        return param_group['lr']

def fit_one_cycle(epochs, max_lr, model, train_loader, val_loader,
                  weight_decay=0, grad_clip=None, opt_func=torch.optim.SGD):
    torch.cuda.empty_cache()
    history = []

    # Set up custom optimizer with weight decay
    optimizer = opt_func(model.parameters(), max_lr, weight_decay=weight_decay)
    # Set up one-cycle learning rate scheduler
    sched = torch.optim.lr_scheduler.OneCycleLR(optimizer, max_lr, epochs=epochs,
                                                steps_per_epoch=len(train_loader))

    for epoch in range(epochs):
        # Training Phase
        model.train()
        train_losses = []
        lrs = []
        for batch in tqdm(train_loader):
            loss = model.training_step(batch)
            train_losses.append(loss)
            loss.backward()

            # Gradient clipping
            if grad_clip:
                nn.utils.clip_grad_value_(model.parameters(), grad_clip)

            optimizer.step()
            optimizer.zero_grad()

            # Record & update learning rate
            lrs.append(get_lr(optimizer))
            sched.step()

        # Validation phase
        result = evaluate(model, val_loader)
        result['train_loss'] = torch.stack(train_losses).mean().item()
        result['lrs'] = lrs
        model.epoch_end(epoch, result)
        history.append(result)
    return history
```

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

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

```
train_dl = DeviceDataLoader(train_dl, device)
valid_dl = DeviceDataLoader(valid_dl, device)
```

Finetuning the Pretrained Model

```
model = PetsModel(len(dataset.classes))
to_device(model, device);
```

```
Downloading: "https://download.pytorch.org/models/resnet34-333f7ec4.pth" to /root/.cache/torch/hub/checkpoints/resnet34-
100%                                    83.3M/83.3M [02:09<00:00, 675kB/s]
```

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

```
[{'val_acc': 0.0347217433154583, 'val_loss': 3.7393031120300293}]
```

```
epochs = 6
max_lr = 0.01
grad_clip = 0.1
weight_decay = 1e-4
opt_func = torch.optim.Adam
```

```
%%time
history += fit_one_cycle(epochs, max_lr, model, train_dl, valid_dl,
                         grad_clip=grad_clip,
                         weight_decay=weight_decay,
                         opt_func=opt_func)
```

```
100%                                    26/26 [02:04<00:00, 4.78s/it]

Epoch [0],last_lr: 0.00589, train_loss: 1.2514, val_loss: 58.4321, val_acc: 0.0193
100%                                    26/26 [01:35<00:00, 3.67s/it]

Epoch [1],last_lr: 0.00994, train_loss: 2.0214, val_loss: 5.2334, val_acc: 0.0855
100%                                    26/26 [00:31<00:00, 1.22s/it]

Epoch [2],last_lr: 0.00812, train_loss: 1.3256, val_loss: 6.3855, val_acc: 0.1792
100%                                    26/26 [00:38<00:00, 1.48s/it]

Epoch [3],last_lr: 0.00463, train_loss: 0.8503, val_loss: 1.1051, val_acc: 0.6573
100%                                    26/26 [00:23<00:00, 1.09it/s]

Epoch [4],last_lr: 0.00133, train_loss: 0.5102, val_loss: 0.7788, val_acc: 0.7553
100%                                    26/26 [00:23<00:00, 1.10it/s]

Epoch [5],last_lr: 0.00000, train_loss: 0.3157, val_loss: 0.6179, val_acc: 0.7871
CPU times: user 48.4 s, sys: 35.8 s, total: 1min 24s
Wall time: 2min 50s
```

Training a model from scratch

Let's repeat the training without using weights from the pretrained ResNet34 model.

```
model2 = PetsModel(len(dataset.classes), pretrained=False)
to_device(model2, device);
```

```
history2 = [evaluate(model2, valid_dl)]
history2
```

```
[{'val_acc': 0.028113815933465958, 'val_loss': 117.01716613769531}]
```

```
%%time
history2 += fit_one_cycle(epochs, max_lr, model2, train_dl, valid_dl,
                         grad_clip=grad_clip,
                         weight_decay=weight_decay,
                         opt_func=opt_func)
```

```
100%                                    26/26 [00:24<00:00, 1.08it/s]

Epoch [0],last_lr: 0.00589, train_loss: 3.5830, val_loss: 284.6843, val_acc: 0.0240
100%                                    26/26 [00:24<00:00, 1.08it/s]

Epoch [1],last_lr: 0.00994, train_loss: 3.4900, val_loss: 13.1248, val_acc: 0.0408
100%                                    26/26 [00:23<00:00, 1.09it/s]

Epoch [2],last_lr: 0.00812, train_loss: 3.1935, val_loss: 3.4588, val_acc: 0.0953
100%                                    26/26 [00:23<00:00, 1.09it/s]

Epoch [3],last_lr: 0.00463, train_loss: 2.9598, val_loss: 3.4741, val_acc: 0.1002
100%                                    26/26 [00:24<00:00, 1.07it/s]

Epoch [4],last_lr: 0.00133, train_loss: 2.7081, val_loss: 2.8131, val_acc: 0.2354
100%                                    26/26 [00:23<00:00, 1.09it/s]

Epoch [5],last_lr: 0.00000, train_loss: 2.5174, val_loss: 2.6638, val_acc: 0.2532
CPU times: user 48.9 s, sys: 36.2 s, total: 1min 25s
Wall time: 2min 51s
```

While the pretrained model reached an accuracy of 80% in less than 3 minutes, the model without pretrained weights could only reach an accuracy of 24%.

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

```
import jovian
```