[Transfer Learning for Image Classification in PyTorch](https://jovian.ai/aakashns/transfer-learning-pytorch" \l "C0)

[Downloading the Dataset](https://jovian.ai/aakashns/transfer-learning-pytorch#C1)

[Creating a Custom PyTorch Dataset](https://jovian.ai/aakashns/transfer-learning-pytorch#C11)

[Creating Training and Validation Sets](https://jovian.ai/aakashns/transfer-learning-pytorch#C17)

[Modifying a Pretrained Model (ResNet34)](https://jovian.ai/aakashns/transfer-learning-pytorch#C23)

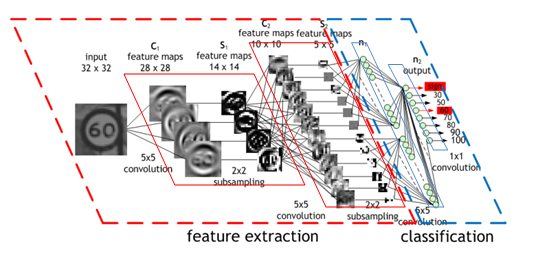
[GPU Utilities and Training Loop](https://jovian.ai/aakashns/transfer-learning-pytorch#C26)

[Finetuning the Pretrained Model](https://jovian.ai/aakashns/transfer-learning-pytorch#C31)

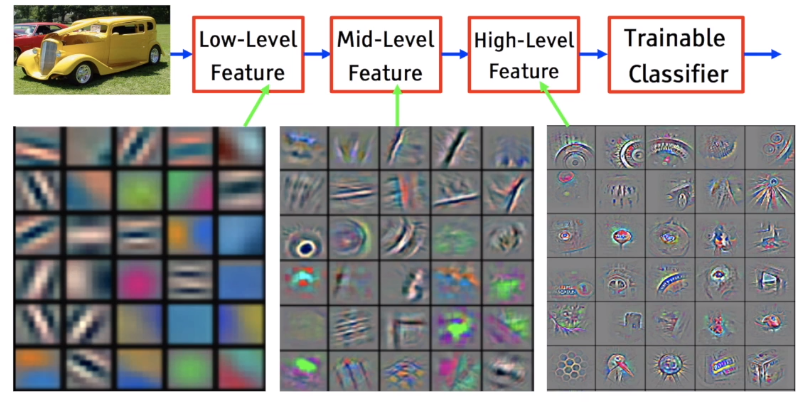
[Training a model from scratch](https://jovian.ai/aakashns/transfer-learning-pytorch#C36)

## Transfer Learning for Image Classification in PyTorch

How a CNN learns ([source](https://jovian.ai/outlink?url=https%3A%2F%2Fdeveloper.nvidia.com%2Fdiscover%2Fconvolutional-neural-network)):



Layer visualization ([source](https://jovian.ai/outlink?url=https%3A%2F%2Fmedium.com%2Fanalytics-vidhya%2Fdeep-learning-visualization-and-interpretation-of-neural-networks-2f3f82f501c5)):



### Downloading the Dataset

We'll use the Oxford-IIIT Pets dataset from [https://course.fast.ai/datasets](https://jovian.ai/outlink?url=https%3A%2F%2Fcourse.fast.ai%2Fdatasets) . 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', '.')

Using downloaded and verified file: ./oxford-iiit-pet.tgz

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]

['pug\_189.jpg',

'newfoundland\_110.jpg',

'Maine\_Coon\_52.jpg',

'leonberger\_70.jpg',

'japanese\_chin\_23.jpg']

def **parse\_breed**(fname):

parts = fname.split('\_')

return ' '.join(parts[:-1])

parse\_breed(files[4])

'japanese chin'

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: Maine Coon (32)

### 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)

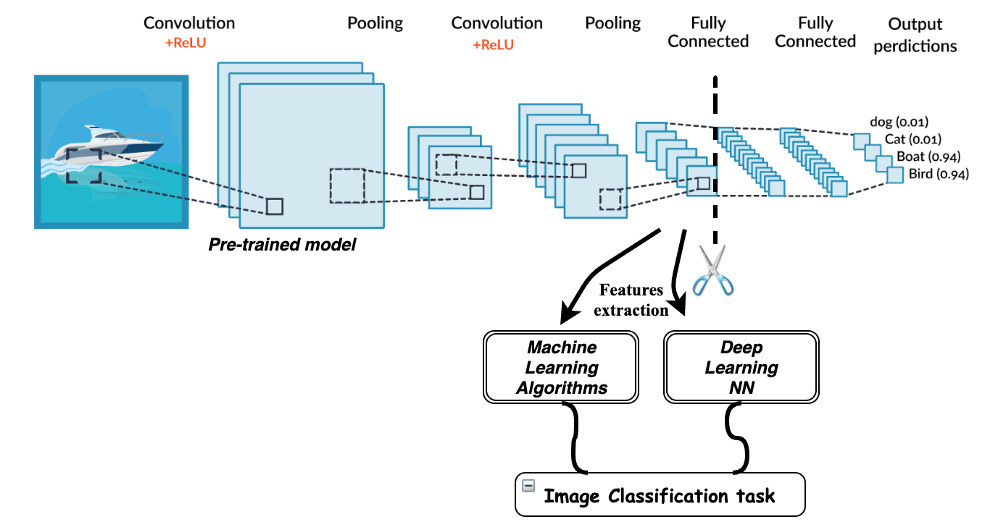
ax.imshow(make\_grid(images, nrow=8).permute(1, 2, 0))

break

show\_batch(train\_dl)

Output hidden; open in [https://colab.research.google.com](https://jovian.ai/outlink?url=https%3A%2F%2Fcolab.research.google.com) to view.

### Modifying a Pretrained Model (ResNet34)

Transfer learning ([source](https://jovian.ai/outlink?url=https%3A%2F%2Fmc.ai%2Ftransfer-learning-with-deep-learning-machine-learning-techniques%2F)):

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);

history = [evaluate(model, valid\_dl)]

history

[{'val\_acc': 0.02591117098927498, 'val\_loss': 3.8896164894104004}]

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)

HBox(children=(FloatProgress(value=0.0, max=26.0), HTML(value='')))

Epoch [0],last\_lr: 0.00589, train\_loss: 1.3628, val\_loss: 165.6215, val\_acc: 0.0240

HBox(children=(FloatProgress(value=0.0, max=26.0), HTML(value='')))

Epoch [1],last\_lr: 0.00994, train\_loss: 1.9505, val\_loss: 4.8602, val\_acc: 0.0447

HBox(children=(FloatProgress(value=0.0, max=26.0), HTML(value='')))

Epoch [2],last\_lr: 0.00812, train\_loss: 1.3101, val\_loss: 1.8901, val\_acc: 0.4405

HBox(children=(FloatProgress(value=0.0, max=26.0), HTML(value='')))

Epoch [3],last\_lr: 0.00463, train\_loss: 0.8174, val\_loss: 1.1133, val\_acc: 0.6309

HBox(children=(FloatProgress(value=0.0, max=26.0), HTML(value='')))

Epoch [4],last\_lr: 0.00133, train\_loss: 0.4924, val\_loss: 0.6684, val\_acc: 0.7638

HBox(children=(FloatProgress(value=0.0, max=26.0), HTML(value='')))

Epoch [5],last\_lr: 0.00000, train\_loss: 0.3033, val\_loss: 0.5651, val\_acc: 0.8167 CPU times: user 49.4 s, sys: 37.7 s, total: 1min 27s Wall time: 2min 59s

### 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': 0.02223292924463749, 'val\_loss': 64.20709228515625}]

%%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)

HBox(children=(FloatProgress(value=0.0, max=26.0), HTML(value='')))

Epoch [0],last\_lr: 0.00589, train\_loss: 3.5621, val\_loss: 570.1448, val\_acc: 0.0227

HBox(children=(FloatProgress(value=0.0, max=26.0), HTML(value='')))

Epoch [1],last\_lr: 0.00994, train\_loss: 3.4390, val\_loss: 4.6319, val\_acc: 0.0438

HBox(children=(FloatProgress(value=0.0, max=26.0), HTML(value='')))

Epoch [2],last\_lr: 0.00812, train\_loss: 3.2430, val\_loss: 3.2921, val\_acc: 0.1269

HBox(children=(FloatProgress(value=0.0, max=26.0), HTML(value='')))

Epoch [3],last\_lr: 0.00463, train\_loss: 2.9670, val\_loss: 3.0076, val\_acc: 0.1647

HBox(children=(FloatProgress(value=0.0, max=26.0), HTML(value='')))

Epoch [4],last\_lr: 0.00133, train\_loss: 2.7464, val\_loss: 2.9556, val\_acc: 0.1672

HBox(children=(FloatProgress(value=0.0, max=26.0), HTML(value='')))

Epoch [5],last\_lr: 0.00000, train\_loss: 2.5684, val\_loss: 2.6616, val\_acc: 0.2449 CPU times: user 48.9 s, sys: 37.4 s, total: 1min 26s Wall time: 2min 57s

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%.