Automated Image Classification using Advanced Deep Learning Models

The goal for this Lab is it to provide you understanding of the various deep learning models that we will be using in your project(s). In this Lab we will implement AlexNet Architecture over CUB200-2011 dataset:

The CUB200-2011 dataset can be found on its website. We can either use the data zip file or import it using Kaggle and kaggle-api which needs to be installed with `pip install kaggle`. I would prefer all of you do this using Google Colab. This will also help you with Last 3 Labs.

Data Preprocessing

Downloading and extracting custom datasets

Loading custom datasets

Calculating the mean and std for normalization on custom datasets

Loading transforms to augment and normalize our data

```
# Import Libraries and Set Random Seed

import pandas as pd
import numpy as np

#!pip install torch==1.4.0
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.optim.lr_scheduler import _LRScheduler
import torch.utils.data as data

import torchvision.transforms as transforms
import torchvision.datasets as datasets
from torchvision import models

from sklearn import decomposition
from sklearn import manifold
from sklearn.metrics import confusion_matrix
from sklearn.metrics import ConfusionMatrixDisplay
```

```
from tqdm.notebook import tqdm, trange
import matplotlib.pyplot as plt

import copy
import random
import time
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-whe
Collecting torch==1.4.0
  Using cached torch-1.4.0-cp37-cp37m-manylinux1_x86_64.whl (753.4 MB)
Installing collected packages: torch
  Attempting uninstall: torch
    Found existing installation: torch 1.2.0
    Uninstalling torch-1.2.0:
      Successfully uninstalled torch-1.2.0
ERROR: pip's dependency resolver does not currently take into account all the pa
torchvision 0.4.0 requires torch==1.2.0, but you have torch 1.4.0 which is incom
torchtext 0.12.0 requires torch==1.11.0, but you have torch 1.4.0 which is incom
torchaudio 0.11.0+cu113 requires torch==1.11.0, but you have torch 1.4.0 which i
fastai 2.6.3 requires torch<1.12,>=1.7.0, but you have torch 1.4.0 which is inco
fastai 2.6.3 requires torchvision>=0.8.2, but you have torchvision 0.4.0 which i
Successfully installed torch-1.4.0
```

```
SEED = 1234

random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)
#torch.cuda.manual_seed(SEED)
#torch.backends.cudnn.deterministic = True
```

```
<torch._C.Generator at 0x7f8f8ce9c310>
```

```
# Set Data folders, TensorDatasets and DataLoaders for the datasets
```

```
ROOT = '.data'

train_data = datasets.CIFAR10(root=ROOT,
                              train=True,
                              download=True)

means = train_data.data.mean(axis=(0,1,2)) / 255
stds = train_data.data.std(axis=(0,1,2)) / 255

print(f'Calculated means: {means}')
print(f'Calculated stds: {stds}')
```

```
Files already downloaded and verified
Calculated means: [0.49139968 0.48215841 0.44653091]
Calculated stds: [0.24703223 0.24348513 0.26158784]
```

```
train_transforms = transforms.Compose([
                                       transforms.RandomRotation(5),
                                       transforms.RandomHorizontalFlip(0.5),
                                       transforms.RandomCrop(32, padding=2),
                                       transforms.ToTensor(),
                                       transforms.Normalize(mean=means,std=stds)
                                       ])

test_transforms = transforms.Compose([
                                      transforms.ToTensor(),
                                      transforms.Normalize(mean=means,std=stds)
                                      ])


train_data = datasets.CIFAR10(ROOT,
                              train=True,
                              download=True,
                              transform=train_transforms)

test_data = datasets.CIFAR10(ROOT,
                             train=False,
                             download=True,
                             transform=test_transforms)
```

```
Files already downloaded and verified
Files already downloaded and verified
```

```
VALID_RATIO = 0.9

n_train_examples = int(len(train_data) * VALID_RATIO)
n_valid_examples = len(train_data) - n_train_examples
```

```
train_data, valid_data = data.random_split(train_data,
                                           [n_train_examples, n_valid_examples])
```

```
valid_data = copy.deepcopy(valid_data)
valid_data.dataset.transform = test_transforms
```

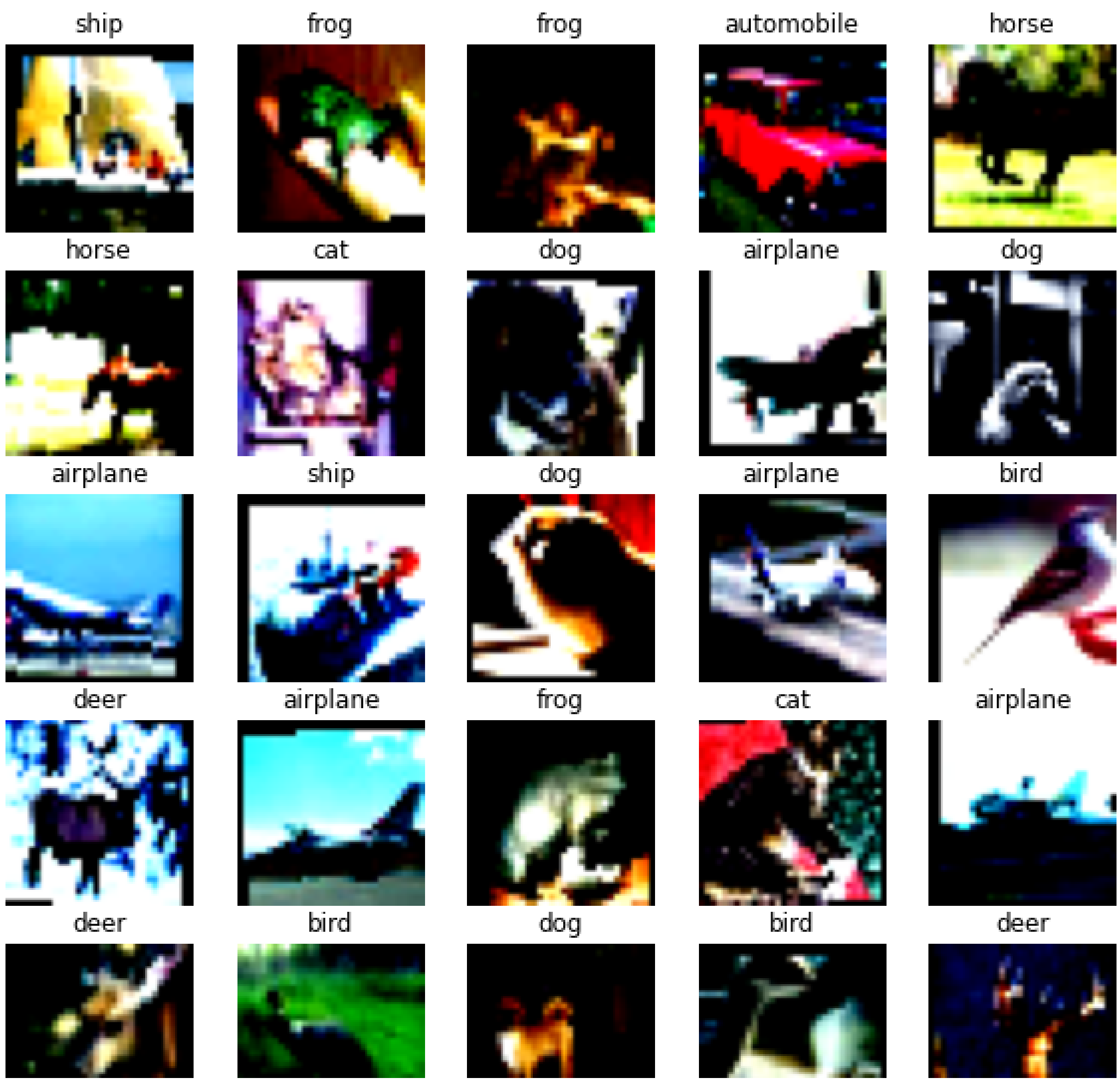
```
print(f'Number of training examples: {len(train_data)}')
print(f'Number of validation examples: {len(valid_data)}')
print(f'Number of testing examples: {len(test_data)}')
```

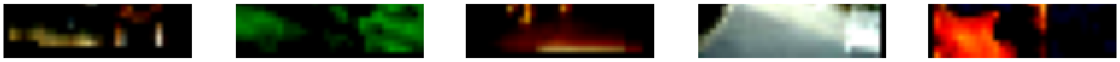
```
Number of training examples: 45000
Number of validation examples: 5000
Number of testing examples: 10000
```

```
def plot_images(images, labels, classes, normalize=False):

  n_images = len(images)

  rows = int(np.sqrt(n_images))
  cols = int(np.sqrt(n_images))

  fig = plt.figure(figsize=(10,10))

  for i in range(rows*cols):

    ax = fig.add_subplot(rows,cols,i+1)

    image = images[i]

    if normalize:
      image_min = image_min()
      image_max = image_max()
      image.clamp_(min=image_min, max=image_max)
      image.add_(-image_min).div_(image_max - image_min + 1e-5)

    ax.imshow(image.permute(1,2,0).cpu().numpy())
    ax.set_title(classes[labels[i]])
    ax.axis('off')
```

```
N_images = 25

images, labels = zip(*[(image,label) for image, label in
                       [train_data[i] for i in range(N_images)]])

classes = test_data.classes

plot_images(images, labels, classes)
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
Clipping input data to the valid range for imshow with RGB data ([0..1] for floa
```





```
def normalize_image(image):
  image_min = image.min()
  image_max = image.max()
  image.clamp_(min=image_min, max=image_max)
  image.add_(-image_min).div(image_max = image_min + 1e-5)
  return image


def plot_filter(images, filter, normalize=True):

  images = torch.cat([i.unsqueeze(0) for i in images], dim=0).cpu()
  filter = torch.FloatTensor(filter).unsqueeze(0).unsqueeze(0).cpu()
  filter = filter.repeat(3,3,1,1)

  n_images = images.shape[0]

  filtered_images = F.conv2d(images, filter)

  images = images.permute(0,2,3,1)

  fig = plt.figure(figsize=(25,5))

  for i in range(n_images):

    image = images[i]

    if normalize:
      image = normalize_image(image)

    ax = fig.add_subplot(2, n_images, i+1)
    ax.imshow(image)
    ax.set_title('Original')
    ax.axis('off')

    image = filtered_images[i]

    if normalize:
      image = normalize_image(image)

    ax = fig.add_subplot(2, n_images, n_images+i+1)
    ax.imshow(image)
    ax.set_title('Filtered')
    ax.axis('off')


from torch.functional import norm
def plot_subsample(images, pool_type, pool_size, normalize=True):
```

```
  images = torch.cat([i.unsqueeze(0) for i in images], dim=0).cpu()

  if pool_type.lower() == 'max':
    pool = F.max_pool2d
  elif pool_type.lower() in ['mean', 'avg']:
    pool = F.avg_pool2d
  else:
    raise ValueError(f'pool_type must be either max, mean, or avg, got: {pool_type}')

  n_images = images.shape[0]

  pooled_images = pool(images, kernel_size=pool_size)

  images = images.permute(0,2,3,1)
  pooled_images = pooled_images.permute(0,2,3,1)

  fig = plt.figure(figsize=(25,5))

  for i in range(n_images):

    image = images[i]

    if normalize:
      image = normalize_image(image)

    ax = fig.add_subplot(2, n_images, i+1)
    ax.imshow(image)
    ax.set_title('Original')
    ax.axis('off')

    image = pooled_images[i]

    if normalize:
      image = normalize_image(image)

    ax = fig.add_subplot(2, n_images, n_images+i+1)
    ax.imshow(image)
    ax.set_title('Subsampled')
    ax.axis('off')
```

```
BATCH_SIZE = 256

train_iterator = data.DataLoader(train_data,
                                 shuffle=True,
                                 batch_size=BATCH_SIZE)

valid_iterator = data.DataLoader(valid_data,
                                 batch_size=BATCH_SIZE)
```

```
test_iterator = data.DataLoader(test_data,
                                batch_size=BATCH_SIZE)
```

Defining a Convolutional Neural Network

Defining the AlexNet blocks

Defining a CUB200-2011 AlexNet model

```
class AlexNet(nn.Module):
  def __init__(self,output_dim):
    super().__init__()

    self.features = nn.Sequential(
        nn.Conv2d(3,64,3,2,1),
        nn.MaxPool2d(2),
        nn.ReLU(inplace=True),
        nn.Conv2d(64,192,3,padding=1),
        nn.MaxPool2d(2),
        nn.ReLU(inplace=True),
        nn.Conv2d(192,384,3,padding=1),
        nn.ReLU(inplace=True),
        nn.Conv2d(384,256,3,padding=1),
        nn.ReLU(inplace=True),
        nn.Conv2d(256,256,3,padding=1),
        nn.MaxPool2d(2),
        nn.ReLU(inplace=True)
        )

    self.classifier = nn.Sequential(
        nn.Dropout(0.5),
        nn.Linear(256*2*2, 4096),
        nn.ReLU(inplace=True),
        nn.Dropout(0.5),
        nn.Linear(4096,4096),
        nn.ReLU(inplace=True),
        nn.Linear(4096,output_dim)
        )

  def forward(self,x):
    x = self.features(x)
    h = x.view(x.shape[0], -1)
    x = self.classifier(h)
    return x, h
```

```
OUTPUT_DIM = 10
```

```
model = AlexNet(OUTPUT_DIM)

def count_parameters(model):
  return sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f'The model has {count_parameters(model):,} trainable parameters')
```

```
The model has 23,272,266 trainable parameters
```

```
def initialize_parameters(m):
  if isinstance(m,nn.Conv2d):
    nn.init.kaiming_normal_(m.weight.data, nonlinearity='relu')
    nn.init.constant_(m.bias.data, 0)
  elif isinstance(m, nn.Linear):
    nn.init.xavier_normal_(m.weight.data, gain=nn.init.calculate_gain('relu'))
    nn.init.constant_(m.bias.data, 0)


model.apply(initialize_parameters)
```

```
AlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (2): ReLU(inplace=True)
    (3): Conv2d(64, 192, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (5): ReLU(inplace=True)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (12): ReLU(inplace=True)
  )
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=1024, out_features=4096, bias=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=4096, out_features=4096, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=4096, out_features=10, bias=True)
  )
)
```

```
class ExponentialLR(_LRScheduler):
  def __init__(self, optimizer, end_lr, num_iter, last_epoch=-1):
    self.end_lr = end_lr
```

```
      self.num_iter = num_iter
      super(ExponentialLR, self).__init__(optimizer, last_epoch)

    def get_lr(self):
      curr_iter = self.last_epoch
      r = curr_iter / self.num_iter
      return [base_lr * (self.end_lr / base_lr) ** r
              for base_lr in self.base_lrs]


  class IteratorWrapper:
    def __init__(self, iterator):
      self.iterator = iterator
      self._iterator = iter(iterator)

    def __next__(self):
      try:
        inputs, labels = next(self._iterator)
      except StopIteration:
        self._iterator = iter(self.iterator)
        inputs, labels, *_ = next(self._iterator)

      return inputs, labels

    def get_batch(self):
      return next(self)


  class LRFinder:
    def __init__(self,model,optimizer,criterion,device):

      self.optimizer = optimizer
      self.model = model
      self.criterion = criterion
      self.device = device

      torch.save(model.state_dict(), 'init_params.pt')

    def range_test(self, iterator, end_lr=10, num_iter=100,
                   smooth_f=0.05, diverge_th=5):

      lrs=[]
      losses=[]
      best_loss=float('inf')

      lr_scheduler = ExponentialLR(self.optimizer, end_lr, num_iter)
      iterator = IteratorWrapper(iterator)

      for iteration in range(num_iter):

        loss = self._train_batch(iterator)
```

```
      lrs.append(lr_scheduler.get_last_lr()[0])

      #Update lr
      lr_scheduler.step()

      if iteration > 0:
        loss = smooth_f * loss + (1 - smooth_f) * losses[-1]

      if loss < best_loss:
        best_loss = loss

      losses.append(loss)

      if loss > diverge_th * best_loss:
        print('Stopping early, the loss has diverged')
        break

    #reset model to initial parameters
    model.load_state_dict(torch.load('init_params.pt'))

    return lrs, losses

  def _train_batch(self, iterator):

    self.model.train()

    self.optimizer.zero_grad()

    x, y = iterator.get_batch()

    x = x.to(self.device)
    y = y.to(self.device)

    y_pred, _ = self.model(x)

    loss = self.criterion(y_pred, y)

    loss.backward()

    self.optimizer.step()

    return loss.item()
```

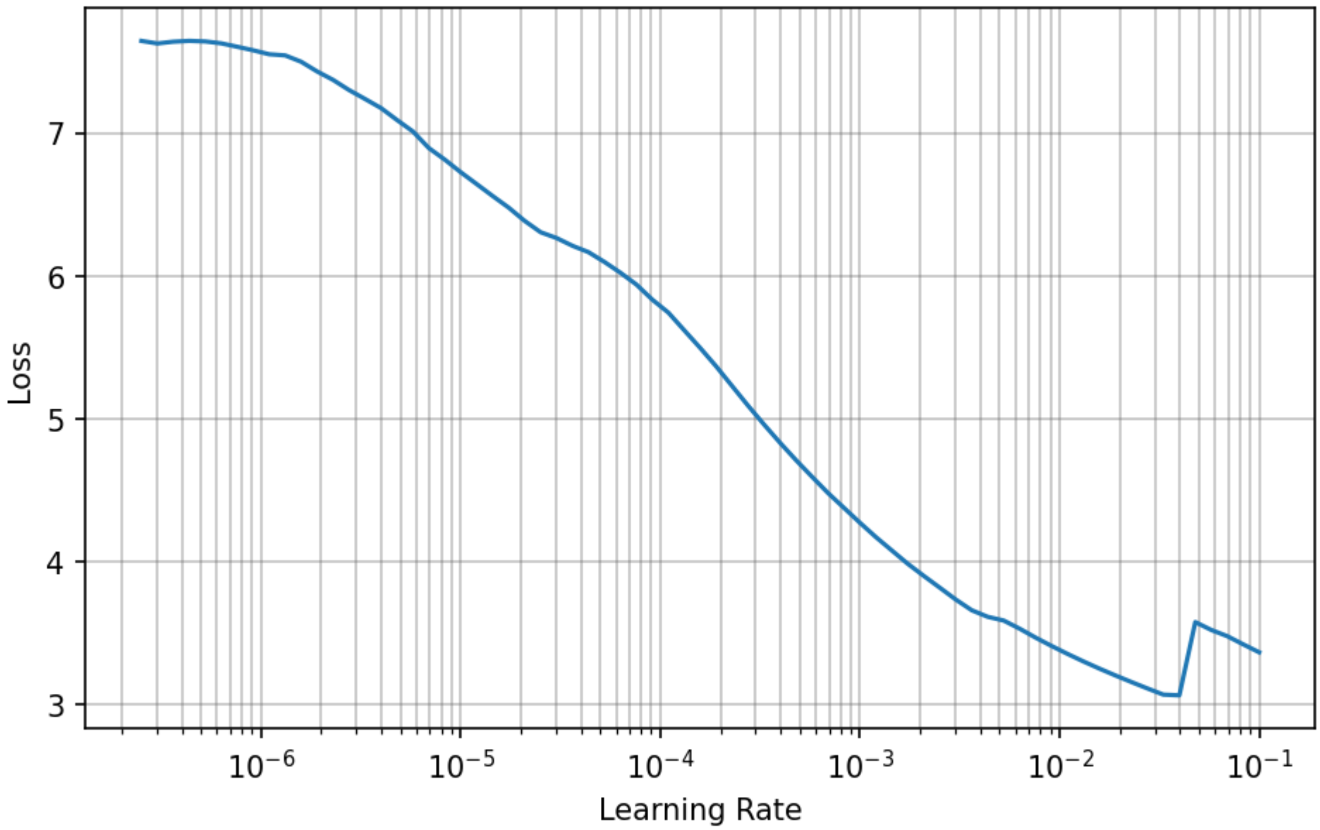
```
START_LR = 1e-7

optimizer = optim.Adam(model.parameters(), lr=START_LR)

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

criterion = nn.CrossEntropyLoss()

model = model.to(device)
criterion = criterion.to(device)
```

```
END_LR = 10
NUM_ITER = 100

lr_finder = LRFinder(model, optimizer, criterion, device)
lrs, losses = lr_finder.range_test(train_iterator, END_LR, NUM_ITER)
```

```
    Stopping early, the loss has diverged
```

```
def plot_lr_finder(lrs, losses, skip_start=5, skip_end=5):

  if skip_end == 0:
    lrs = lrs[skip_start:]
    losses = losses[skip_start:]
  else:
    lrs = lrs[skip_start:-skip_end]
    losses = losses [skip_start:-skip_end]

  fig = plt.figure(figsize=(7.5,4.5), dpi=150)
  ax = fig.add_subplot(1,1,1)
  ax.plot(lrs,losses)
  ax.set_xscale('log')
  ax.set_xlabel('Learning Rate')
  ax.set_ylabel('Loss')
  ax.grid(True, 'both', 'x', color='#666666', linestyle='-', alpha=0.4)
  ax.grid(True, 'both', 'y', color='#666666', linestyle='-', alpha=0.4)
```

```
plot_lr_finder(lrs, losses)
```




```
FOUND_LR = 1e-3

optimizer = optim.Adam(model.parameters(), lr=FOUND_LR)


def calculate_accuracy(y_pred, y):
  top_pred = y_pred.argmax(1, keepdim=True)
  correct = top_pred.eq(y.view_as(top_pred)).sum()
  acc = correct.float() / y.shape[0]
  return acc


def train(model, iterator, optimizer, criterion, device):

  epoch_loss = 0
  epoch_acc = 0

  model.train()

  for (x, y) in tqdm(iterator, desc="Training", leave=False):

    x = x.to(device)
    y = y.to(device)

    optimizer.zero_grad()

    y_pred, _ = model(x)
```

```
    loss = criterion(y_pred, y)

    acc = calculate_accuracy(y_pred, y)

    loss.backward()

    optimizer.step()

    epoch_loss += loss.item()
    epoch_acc += acc.item()

  return epoch_loss / len(iterator), epoch_acc / len(iterator)


def evaluate(model, iterator, criterion, device):

  epoch_loss = 0
  epoch_acc = 0

  model.eval()

  with torch.no_grad():

    for (x, y) in tqdm(iterator, desc="Evaluating", leave=False):

      x = x.to(device)
      y = y.to(device)

      y_pred, _ = model(x)

      loss = criterion(y_pred, y)

      acc = calculate_accuracy(y_pred, y)

      epoch_loss += loss.item()
      epoch_acc += acc.item()

  return epoch_loss / len(iterator), epoch_acc / len(iterator)


def epoch_time(start_time, end_time):
  elapsed_time = end_time - start_time
  elapsed_mins = int(elapsed_time / 60)
  elapsed_secs = int(elapsed_time - (elapsed_mins * 60))
  return elapsed_mins, elapsed_secs


EPOCHS = 25

best_valid_loss = float('inf')
```

```
for epoch in trange(EPOCHS, desc="Epochs"):

  start_time = time.monotonic()

  train_loss, train_acc = train(model, train_iterator, optimizer, criterion, device)
  valid_loss, valid_acc = evaluate(model, valid_iterator, criterion, device)

  if valid_loss < best_valid_loss:
    best_valid_loss = valid_loss
    torch.save(model.state_dict(), 'tut3-model.pt')

  end_time = time.monotonic()

  epoch_mins, epoch_secs = epoch_time(start_time, end_time)
  print(f'Epoch: {epoch+1:02} | Epoch Time: {epoch_mins}m {epoch_secs}s')
  print(f'Train Loss: {train_loss:.3f} | Train Acc: {train_acc*100:.2f}%')
  print(f'Valid Loss: {valid_loss:.3f} | Valid Acc: {valid_acc*100:.2f}%')
```

Epochs: 24% 6/25 [43:39<2:19:18, 439.93s/it]

```
Epoch: 01 | Epoch Time: 7m 19s
Train Loss: 2.463 | Train Acc: 17.92%
Valid Loss: 1.863 | Valid Acc: 27.64%
Epoch: 02 | Epoch Time: 7m 10s
Train Loss: 1.669 | Train Acc: 36.59%
Valid Loss: 1.456 | Valid Acc: 46.88%
Epoch: 03 | Epoch Time: 7m 6s
Train Loss: 1.415 | Train Acc: 48.27%
Valid Loss: 1.247 | Valid Acc: 55.22%
Epoch: 04 | Epoch Time: 7m 4s
Train Loss: 1.290 | Train Acc: 53.42%
Valid Loss: 1.177 | Valid Acc: 57.29%
Epoch: 05 | Epoch Time: 7m 32s
Train Loss: 1.195 | Train Acc: 57.26%
Valid Loss: 1.075 | Valid Acc: 62.54%
Epoch: 06 | Epoch Time: 7m 25s
Train Loss: 1.122 | Train Acc: 60.25%
Valid Loss: 1.039 | Valid Acc: 63.35%
```

Training: 97% 171/176 [07:04<00:12, 2.46s/it]

Training a Convolutional Neural Network

Loading a pre-trained model

Loading pre-trained model parameters into a defined model

Learning rate finder

Discriminative fine-tuning

One cycle learning rate scheduler

Evaluating a Convolutional Neural Network

Fine-tuning a pre-trained model to achieve ~80% top-1 accuracy and ~95% top-5 accuracy on a dataset with 200 classes and only 60 examples per class

Viewing our model's mistakes

Visualizing our data in lower dimensions with PCA and t-SNE

Viewing the learned weights of our model

Executing (50m 44s) Cell > train() > backward() > backward()