Image Processing

SOS CS11

Mid-Term Report

Panav Shah
23B3323

June 2024

Contents

1 Diving into Deep Learning

Deep learning is a method in artificial intelligence (AI) that teaches computers to process data in a way that is inspired by the human brain.

1.1 Deep Learning in Image Processing

Deep learning has revolutionized the field of image processing by providing powerful techniques for tasks such as image classification, object detection, segmentation, and image generation. Here's an overview of how deep learning is applied in image processing:

1.1.1 Image Classification

Convolutional Neural Networks (CNNs): CNNs are the backbone of image classification tasks. They use layers of convolutional filters to extract features from images, followed by fully connected layers that classify the images based on the extracted features. Famous architectures include AlexNet, VGGNet, ResNet, and Inception.

1.1.2 Object Detection

Region-Based CNNs (R-CNN): The original goal of R-CNN was to take an input image and produce a set of bounding boxes as output, where each bounding box contains an object and also the category (e.g. car or pedestrian) of the object. More recently, R-CNN has been extended to perform other computer vision tasks. The following covers some of the versions of R-CNN that have been developed.

1.1.3 Image Segmentation

Semantic Segmentation: Classify the object class for each pixel within an image. That means there is a label for each pixel.

1.1.4 Image Enhancement

Super-Resolution: Increasing the resolution of images using models like Super-Resolution CNN (SRCNN) and Generative Adversarial Networks for Image Super-Resolution (SRGAN).

1.2 Key concepts in Deep Learning for Image Processing

- **Convolution :** Convolution is a general purpose filter effect for images. It is a matrix applied to an image and a mathematical operation, comprised of integers

- **Pooling :** The pooling operation involves sliding a two-dimensional filter over each channel of feature map and summarising the features lying within the region covered by the filter.

- **Activation Functions :** Non-linear functions like ReLU (Rectified Linear Unit) that introduce non-linearity into the model, allowing it to learn more complex patterns.

- **Backpropogation :** The process of updating the model's weights by calculating the gradient of the loss function with respect to each weight and adjusting the weights to minimize the loss.

- **Data Augmentation :** Techniques such as rotation, scaling, flipping, colour variation, ect. used to artificially increase the size of the training dataset, improving the model's accuracy.

1.3 Tools and Frameworks

PyTorch : PyTorch is a machine learning library based on the Torch library, used for applications such as computer vision and natural language processing, originally developed by Meta AI and now part of the Linux Foundation umbrella. It is recognized as one of the two most popular machine learning libraries alongside TensorFlow, offering free and open-source software released under the modified BSD license. Although the Python interface is more polished and the primary focus of development, PyTorch also has a C++ interface.

2 Image Classification

2.1 Classification of FashionMNIST Dataset

2.1.1 Importing computer vision libraries

```python
# Import PyTorch
import torch
from torch import nn

# Import torchvision
import torchvision
from torchvision import datasets
from torchvision import transforms
from torchvision.transforms import ToTensor

# Import matplotlib for visualisation
import matplotlib.pyplot as plt

# Check versions
print(torch.__version__)
print(torchvision.__version__)
```

```
2.3.0+cu121
0.18.0+cu121
```

2.1.2 Getting the dataset

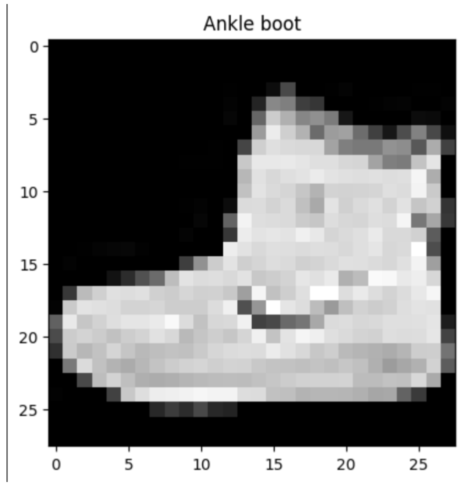
```python
# Setup training data
from torchvision import datasets
train_data = datasets.FashionMNIST(
    root="data", # where to download the data to?
    train=True, # do we want the training dataset?
    download=True, # do we want to download yes/no?
    transform=torchvision.transforms.ToTensor(), # how do we want to transform the data?
    target_transform=None # how do we want to transform the labels/targets?
)

test_data = datasets.FashionMNIST(
    root="data", # where to download the data to?
    train=False, # do we want the training dataset?
    download=True, # do we want to download yes/no?
    transform=torchvision.transforms.ToTensor(), # how do we want to transform the data?
    target_transform=None # how do we want to transform the labels/targets?
)

image, label = train_data[0]
class_names = train_data.classes
class_to_idx = train_data.class_to_idx

print(f"Image shape: {image.shape} -> [color_channels, height, width]")
print(f"Image label: {class_names[label]}")
```

```
Image shape: torch.Size([1, 28, 28]) -> [color_channels, height, width]
Image label: Ankle boot
```

2.1.3 Visualise our data

```python
plt.imshow(image.squeeze(), cmap="gray")
plt.title(class_names[label])
```

```
Text(0.5, 1.0, 'Ankle boot')
```



2.1.4 Prepare DataLoader

```
from torch.utils.data import DataLoader

# Setup the batch size hyperparameter
BATCH_SIZE = 32

# Turn datasets into iterables (batches)
train_dataloader = DataLoader(dataset=train_data,
                              batch_size=BATCH_SIZE,
                              shuffle=True)

test_dataloader = DataLoader(dataset=test_data,
                             batch_size=BATCH_SIZE,
                             shuffle=False)

print(f"Length of train_dataloader: {len(train_dataloader)} batches of {BATCH_SIZE}")
print(f"Length of test_dataloader: {len(test_dataloader)} batches of {BATCH_SIZE}")
```

```
Length of train_dataloader: 1875 batches of 32
Length of test_dataloader: 313 bat
ches of 32
```

2.1.5 Building a Convolutional Neural Network (CNN)

```
# Create a convolutional neural network
class FashionMNISTModel(nn.Module):
  """
  Model architecture that replicates the TinyVGG
  model from CNN explainer website.
  """
```

```
  def __init__(self, input_shape: int, hidden_units: int, output_shape: int):
    super().__init__()
    self.conv_block_1 = nn.Sequential(
        nn.Conv2d(in_channels=input_shape,
                  out_channels=hidden_units,
                  kernel_size=3,
                  stride=1,
                  padding=1), # values we can set ourselves in our NN's are called hyperparameters
        nn.ReLU(),
        nn.Conv2d(in_channels=hidden_units,
                  out_channels=hidden_units,
                  kernel_size=3,
                  stride=1,
                  padding=1),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size=2)
    )
    self.conv_block_2 = nn.Sequential(
        nn.Conv2d(in_channels=hidden_units,
                  out_channels=hidden_units,
                  kernel_size=3,
                  stride=1,
                  padding=1), # values we can set ourselves in our NN's are called hyperparameters
        nn.ReLU(),
        nn.Conv2d(in_channels=hidden_units,
                  out_channels=hidden_units,
                  kernel_size=3,
                  stride=1,
                  padding=1),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size=2)
    )
    self.classifier = nn.Sequential(
        nn.Flatten(),
        nn.Linear(in_features=hidden_units*7*7, # There is a trick to calculate this
                  out_features=output_shape)
    )


  def forward(self, x):
    x = self.conv_block_1(x)
    x = self.conv_block_2(x)
    x = self.classifier(x)
    return x


model = FashionMNISTModel(input_shape=1,
                              hidden_units=10,
                              output_shape=len(class_names)).to(device)

model
```

```
FashionMNISTModelV2(
  (conv_block_1): Sequential(
    (0): Conv2d(1, 10, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): Conv2d(10, 10, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU()
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (conv_block_2): Sequential(
    (0): Conv2d(10, 10, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): Conv2d(10, 10, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU()
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (classifier): Sequential(
    (0): Flatten(start_dim=1, end_dim=-1)
    (1): Linear(in_features=490, out_features=10, bias=True)
  )
)
```

2.1.6 Setting up Loss function, Optimizer and Accuracy function

```python
loss_fn = nn.CrossEntropyLoss()
optimizer=torch.optim.SGD(params=model_2.parameters(),
                          lr=0.1)
def accuracy_fn(y_true, y_pred):
    correct = torch.eq(y_true, y_pred).sum().item()
    acc = (correct / len(y_pred)) * 100
    return acc
```

2.1.7 Defining each train and test step

```python
def train_step(model: torch.nn.Module,
               data_loader: torch.utils.data.DataLoader,
               loss_fn: torch.nn.Module,
               optimizer: torch.optim.Optimizer,
               accuracy_fn,
               device: torch.device = device):
  """ Performs a training with model trying to learn on data_loader
    ."""
  train_loss, train_acc = 0, 0
  model.to(device)

  # Put model into training mode
  model.train()

  # Add a loop to loop through the training batches
  for batch, (X, y) in enumerate(data_loader):
    # Put data on target device
```

```
    X, y = X.to(device), y.to(device)

    # 1. Forward pass (outputs the raw logits from the model)
    y_pred = model(X)

    # 2. Calculate loss and accuracy (per batch)
    loss = loss_fn(y_pred, y)
    train_loss += loss # accumulate train loss
    train_acc += accuracy_fn(y_true=y,
                             y_pred=y_pred.argmax(dim=1)) # go from logits - prediction labels

    # 3. Optimizer zero grad
    optimizer.zero_grad()

    # 4. Loss backward
    loss.backward()

    # 5. Optimizer step
    optimizer.step()


  # Divide total train loss by length of train dataloader
  train_loss /= len(data_loader)
  train_acc /= len(data_loader)
  print(f"Train loss: {train_loss:.5f} | Train acc: {train_acc:.2f}%")
```

```
def test_step(model: torch.nn.Module,
              data_loader: torch.utils.data.DataLoader,
              loss_fn: torch.nn.Module,
              accuracy_fn,
              device: torch.device = device):
  """Performs a testing loop step on model going over data_loader."""
  test_loss, test_acc = 0, 0
  model.to(device)
  # Put the model in eval mode
  model.eval()

  # Turn on inference mode context manager
  with torch.inference_mode():
    for X, y in data_loader:
      # Send the data to the target device
      X, y = X.to(device), y.to(device)

      # 1. Forward pass
      test_pred = model(X)

      # 2. Calculate the loss/acc
      test_loss += loss_fn(test_pred, y)
      test_acc += accuracy_fn(y_true=y,
                              y_pred=test_pred.argmax(dim=1)) # go from logits -> prediction labels
    # Adjust metrics and print out
    test_loss /= len(data_loader)
    test_acc /= len(data_loader)
```

```
    print(f"Test loss: {test_loss:.5f} | Test acc: {test_acc:.2f}%\
    n")
```

2.1.8 Training and Testing our model using the train$_{s}tepandtest_{s}tepfunctions$

```
# Train and test model
epochs = 10
for epoch in range(epochs):
  print(f"Epoch: {epoch}\n-----")
  train_step(model=model,
             data_loader=train_dataloader,
             loss_fn=loss_fn,
             optimizer=optimizer,
             accuracy_fn=accuracy_fn,
             device=device)
  test_step(model=model,
            data_loader=test_dataloader,
            loss_fn=loss_fn,
            accuracy_fn=accuracy_fn,
            device=device)
```

```
Epoch: 0
-----
Train loss: 0.59531 | Train acc: 78.43%
Test loss: 0.39706 | Test acc: 85.53%

Epoch: 1
-----
Train loss: 0.36254 | Train acc: 87.01%
Test loss: 0.34667 | Test acc: 87.16%

Epoch: 2
-----
Train loss: 0.32573 | Train acc: 88.22%
Test loss: 0.31780 | Test acc: 88.42%

Epoch: 3
-----
Train loss: 0.30446 | Train acc: 88.88%
Test loss: 0.32878 | Test acc: 87.88%

Epoch: 4
-----
Train loss: 0.29002 | Train acc: 89.44%
Test loss: 0.30029 | Test acc: 89.22%

Epoch: 5
-----
```

```
Train loss: 0.27894 | Train acc: 89.83%
Test loss: 0.30936 | Test acc: 89.16%

Epoch: 6
-----
Train loss: 0.26996 | Train acc: 90.23%
Test loss: 0.31740 | Test acc: 88.49%

Epoch: 7
-----
Train loss: 0.26435 | Train acc: 90.36%
Test loss: 0.29987 | Test acc: 89.01%

Epoch: 8
-----
Train loss: 0.25912 | Train acc: 90.54%
Test loss: 0.30351 | Test acc: 89.90%

Epoch: 9
-----
Train loss: 0.25476 | Train acc: 90.76%
Test loss: 0.30208 | Test acc: 89.19%
```

2.1.9 Evaluate random predictions using our model

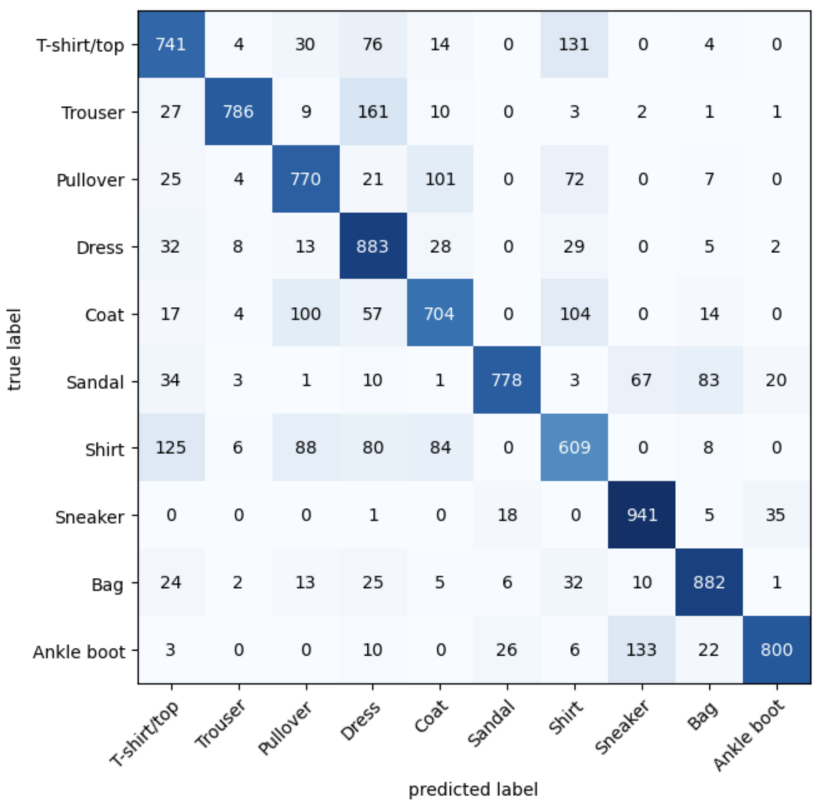
```python
def make_predictions(model: torch.nn.Module,
                     data: list,
                     device: torch.device = device):
  pred_probs = []
  model.eval()
  with torch.inference_mode():
    for sample in data:
      # Prepare the sample (add a batch dimension and pass to target device)
      sample = torch.unsqueeze(sample, dim=0).to(device)

      # Forward pass (model outputs raw logits)
      pred_logit = model(sample)

      # Get prediction probability (logit -> prediction probability)
      pred_prob = torch.softmax(pred_logit.squeeze(), dim=0)

      # Get pred_prob off the GPU for further calculations
      pred_probs.append(pred_prob.cpu())

  # Stack the pred_probs to turn list into a tensor
  return torch.stack(pred_probs)

# Make predictions
pred_probs = make_predictions(model=model,
```

```
                                  data=test_samples)

# Convert prediction probabilites to labels
pred_classes = pred_probs.argmax(dim=1)
pred_classes

# Plot predictions
plt.figure(figsize=(9, 9))
nrows = 3
ncols = 3
for i, sample in enumerate(test_samples):
  # Create subplot
  plt.subplot(nrows, ncols, i+1)

  # Plot the target image
  plt.imshow(sample.squeeze(), cmap="gray")

  # Find the prediction (in text form, e.g. "Sandal")
  pred_label = class_names[pred_classes[i]]

  # Get the truth label (in text form)
  truth_label = class_names[test_labels[i]]

  # Create a title for the plot
  title_text = f"Pred: {pred_label} | Truth: {truth_label}"

  # Check for equality between pred and truth and change color of
    the title text
  if pred_label == truth_label:
    plt.title(title_text, fontsize=10, c="g")
  else:
    plt.title(title_text, fontsize=10, c="r")

  plt.axis(False)
```

2.1.10 Plot a Confusion Matrix

```
# 1. Make predictions with trained model
y_preds = []
model_2.eval()
with torch.inference_mode():
  for X, y in test_dataloader:
    # Send the data and targets to target device
    X, y = X.to(device), y.to(device)
    # Do the forward pass
    y_logit = model_2(X)
    # Turn predictions from logits -> prediction probabilities ->
    prediction labels
    y_pred = torch.softmax(y_logit.squeeze(), dim=0).argmax(dim=1)
    # Put predications on CPU for evaluation
    y_preds.append(y_pred.cpu())

# Concatenate list of predictions into a tensor
y_pred_tensor = torch.cat(y_preds)

from torchmetrics import ConfusionMatrix
from mlxtend.plotting import plot_confusion_matrix

```

```python
# 2. Setup confusion instance and compare predictions to targets
confmat = ConfusionMatrix(num_classes=len(class_names), task="
    multiclass")
confmat_tensor = confmat(preds=y_pred_tensor,
                         target=test_data.targets)

# 3. Plot the confusion matrix
fig, ax = plot_confusion_matrix(
    conf_mat=confmat_tensor.numpy(), # matplotlib likes working
    with numpy
    class_names=class_names,
    figsize=(10, 7)
)
```



2.2 Classification of a Pizza, Steak and Sushi (a subset of the Food101 dataset

2.2.1 Importing computer vision libraries, setting up device agnostic code and importing the dataset

```python
import torch
```

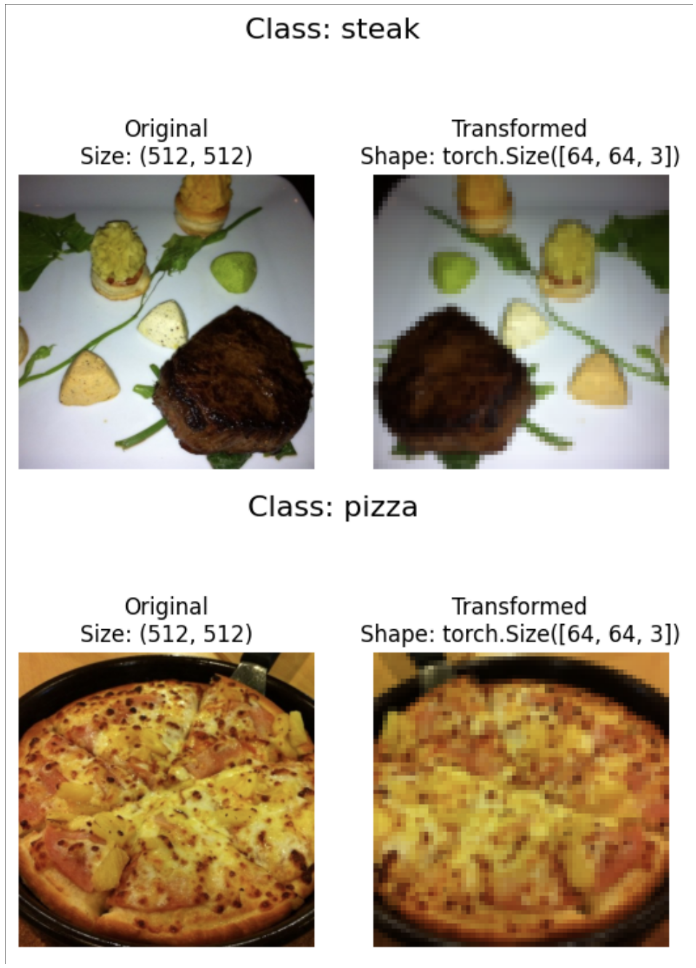
```
from torch import nn

# Setup device-agnostic code
device = "cuda" if torch.cuda.is_available() else "cpu"

import requests
import zipfile
from pathlib import Path

# Setup path to a data folder
data_path = Path("data/")
image_path = data_path / "pizza_steak_sushi"

# If the image folder doesn't exist, download it and prepare it...
if image_path.is_dir():
  print(f"{image_path} directory already exists... skipping download")
else:
  print(f"{image_path} does not exist... creating one")
  image_path.mkdir(parents=True, exist_ok=True)

  # Download pizza, steak and sushi data
  with open(data_path / "pizza_steak_sushi.zip", "wb") as f:
    request = requests.get("https://github.com/mrdbourke/pytorch-deep-learning/blob/main/data/pizza_steak_sushi.zip?raw=true")
    print("Downloading pizza, steak, sushi data...")
    f.write(request.content)

  # Unzip pizza, steak, sushi data
  with zipfile.ZipFile(data_path / "pizza_steak_sushi.zip", "r") as zip_ref:
    print("Unzipping pizza, steak and sushi data...")
    zip_ref.extractall(image_path)
```

```
data/pizza_steak_sushi does not exist... creating one
Downloading pizza, steak, sushi data...
Unzipping pizza, steak and sushi data...
```

2.2.2 Transforming data

```
import torch
from torch.utils.data import DataLoader
from torchvision import datasets, transforms

# Write a transform for image
data_transform = transforms.Compose([
    # Resize our images to 64x64
    transforms.Resize(size=(64, 64)),
    # Flip the images randomly on the horizontal
    transforms.RandomHorizontalFlip(p=0.5),
    # Turn the image into a torch.tensor
    transforms.ToTensor()
])

```

```python
def plot_transformed_images(image_path: list, transform, n=3, seed=42):
  """
  Selects random images from a path of images and loads/transforms them
  then plots the original vs the transformed version.
  """
  if seed:
    random.seed(seed)
  random_image_paths = random.sample(image_path_list, k=n)
  for image_path in random_image_paths:
    with Image.open(image_path) as f:
      fig, ax = plt.subplots(nrows=1, ncols=2)
      ax[0].imshow(f)
      ax[0].set_title(f"Original\nSize: {f.size}")
      ax[0].axis(False)

      # Transform and plt target image
      transformed_image = transform(f).permute(1, 2, 0) # note we will need to change shape
      ax[1].imshow(transformed_image)
      ax[1].set_title(f"Transformed\nShape: {transformed_image.shape}")
      ax[1].axis("off")

      fig.suptitle(f"Class: {image_path.parent.stem}", fontsize=16)

plot_transformed_images(image_path=image_path_list,
                        transform=data_transform,
                        n=2,
                        seed=42)
```



2.2.3 Loading images to a dataset

```
# Use ImageFolder to create datasets
from torchvision import datasets
train_data = datasets.ImageFolder(root=train_dir,
                                  transform=data_transform, # a
    transform for the data
                                  target_transform=None) # a
    transform for the label

test_data = datasets.ImageFolder(root=test_dir,
                                 transform=data_transform,
                                 target_transform=None)


class_names = train_data.classes
class_dict = train_data.class_to_idx
```

```python

img, label = train_data[0]
print(f"Image tensor:\n {img}")
print(f"Image shape: {img.shape}")
print(f"Image datatype: {img.dtype}")
print(f"Image label: {label}")
print(f"Label datatype: {type(label)}")
```

```
Image tensor:
 tensor([[[0.1137, 0.1020, 0.0980,  ..., 0.1255, 0.1216, 0.1176],
         [0.1059, 0.0980, 0.0980,  ..., 0.1294, 0.1294, 0.1294],
         [0.1020, 0.0980, 0.0941,  ..., 0.1333, 0.1333, 0.1333],
         ...,
         [0.1098, 0.1098, 0.1255,  ..., 0.1686, 0.1647, 0.1686],
         [0.0902, 0.0941, 0.1098,  ..., 0.1686, 0.1647, 0.1686],
         [0.0863, 0.0863, 0.0980,  ..., 0.1686, 0.1647, 0.1647]],

        [[0.0745, 0.0706, 0.0745,  ..., 0.0588, 0.0588, 0.0588],
         [0.0745, 0.0706, 0.0745,  ..., 0.0627, 0.0627, 0.0627],
         [0.0706, 0.0745, 0.0745,  ..., 0.0706, 0.0706, 0.0706],
         ...,
         [0.1255, 0.1333, 0.1373,  ..., 0.2510, 0.2392, 0.2392],
         [0.1098, 0.1176, 0.1255,  ..., 0.2510, 0.2392, 0.2314],
         [0.1020, 0.1059, 0.1137,  ..., 0.2431, 0.2353, 0.2275]],

        [[0.0941, 0.0902, 0.0902,  ..., 0.0157, 0.0196, 0.0196],
         [0.0902, 0.0863, 0.0902,  ..., 0.0196, 0.0157, 0.0196],
         [0.0902, 0.0902, 0.0902,  ..., 0.0157, 0.0157, 0.0196],
         ...,
         [0.1294, 0.1333, 0.1490,  ..., 0.1961, 0.1882, 0.1843],
         [0.1098, 0.1137, 0.1255,  ..., 0.1922, 0.1843, 0.1804],
         [0.1059, 0.0980, 0.1059,  ..., 0.1882, 0.1804, 0.1765]]])
Image shape: torch.Size([3, 64, 64])
Image datatype: torch.float32
Image label: 0
Label datatype: <class 'int'>
```

2.2.4 Turn loaded images into Dataloader's

```python
# Turn train and test datasets into DataLoader's
from torch.utils.data import DataLoader
BATCH_SIZE = 1
train_dataloader = DataLoader(dataset=train_data,
                              batch_size=BATCH_SIZE,
                              shuffle=True)
test_dataloader = DataLoader(dataset=test_data,
                             batch_size=BATCH_SIZE,
                             shuffle=False)
```

2.2.5 Create a TinyVGG model

```
# Create simple transform
simple_transform = transforms.Compose([
    transforms.Resize(size=(64, 64)),
    transforms.ToTensor()
])

# 1. Load and transfer data
from torchvision import datasets
train_data_sample = datasets.ImageFolder(root=train_dir,
                                         transform=simple_transform
    )
test_data_sample = datasets.ImageFolder(root=test_dir,
                                         transform=simple_transform
    )

# 2. Turn the datasets into Dataloaders
import os
from torch.utils.data import DataLoader

# Setup batch size and number of workers
BATCH_SIZE = 32
NUM_WORKERS = os.cpu_count()

class TinyVGG(nn.Module):
  """
  Model architecture copying TinyVGG from CNN Explainer
  """
  def __init__(self, input_shape: int,
               hidden_units: int,
               output_shape: int) -> None:
    super().__init__()
    self.conv_block_1 = nn.Sequential(
        nn.Conv2d(in_channels=input_shape,
                  out_channels=hidden_units,
                  kernel_size=3,
                  stride=1,
                  padding=0),
        nn.ReLU(),
        nn.Conv2d(in_channels=hidden_units,
                  out_channels=hidden_units,
                  kernel_size=3,
                  stride=1,
                  padding=0),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size=2,
                     stride=2)
    )
    self.conv_block_2 = nn.Sequential(
        nn.Conv2d(in_channels=hidden_units,
                  out_channels=hidden_units,
                  kernel_size=3,
                  stride=1,
                  padding=0),
        nn.ReLU(),
        nn.Conv2d(in_channels=hidden_units,
```

```
                  out_channels=hidden_units,
                  kernel_size=3,
                  stride=1,
                  padding=0),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size=2,
                     stride=2)
    )
    self.classifier = nn.Sequential(
        nn.Flatten(),
        nn.Linear(in_features=hidden_units*13*13,
                  out_features=output_shape)
    )

  def forward(self, x):
    x = self.conv_block_1(x)
    x = self.conv_block_2(x)
    x = self.classifier(x)
    return x

model_0 = TinyVGG(input_shape=3,
                  hidden_units=10,
                  output_shape=len(class_names)).to(device)
model_0
```

```
TinyVGG(
  (conv_block_1): Sequential(
    (0): Conv2d(3, 10, kernel_size=(3, 3), stride=(1, 1))
    (1): ReLU()
    (2): Conv2d(10, 10, kernel_size=(3, 3), stride=(1, 1))
    (3): ReLU()
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (conv_block_2): Sequential(
    (0): Conv2d(10, 10, kernel_size=(3, 3), stride=(1, 1))
    (1): ReLU()
    (2): Conv2d(10, 10, kernel_size=(3, 3), stride=(1, 1))
    (3): ReLU()
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (classifier): Sequential(
    (0): Flatten(start_dim=1, end_dim=-1)
    (1): Linear(in_features=1690, out_features=3, bias=True)
  )
)
```

2.2.6 Create train step, test step and train functions

```
# Create train_step()
def train_step(model: torch.nn.Module,
               dataloader: torch.utils.data.DataLoader,
```

```
               loss_fn: torch.nn.Module,
               optimizer: torch.optim.Optimizer,
               device=device):
  # Put the model in train mode
  model.train()

  # Setup train loss and train accuracy values
  train_loss, train_acc = 0, 0

  # Loop through data loader data batches
  for batch, (X, y) in enumerate(dataloader):
    # Send data to the target device
    X, y = X.to(device), y.to(device)

    # 1. Forward pass
    y_pred = model(X) # output model logits

    # 2. Calculate the loss
    loss = loss_fn(y_pred, y)
    train_loss += loss.item()

    # 3. Optimizer zero grad
    optimizer.zero_grad()

    # 4. Loss backward
    loss.backward()

    # 5. Optimizer step
    optimizer.step()

    # Calculate accuracy metric
    y_pred_class = torch.argmax(torch.softmax(y_pred, dim=1), dim=1)
    train_acc += (y_pred_class==y).sum().item()/len(y_pred)

  # Adjust metrics to get average loss and accuracy per batch
  train_loss = train_loss / len(dataloader)
  train_acc = train_acc / len(dataloader)
  return train_loss, train_acc

# Create a test step
def test_step(model: torch.nn.Module,
              dataloader: torch.utils.data.DataLoader,
              loss_fn: torch.nn.Module,
              device=device):
  # Put model in eval mode
  model.eval()

  # Setup test loss and test accuracy values
  test_loss, test_acc = 0, 0

  # Turn on inference mode
  with torch.inference_mode():
    # Loop through DataLoader batches
    for batch, (X, y) in enumerate(dataloader):
      # Send data to the target device
      X, y = X.to(device), y.to(device)
```

```python

      # 1. Forward pass
      test_pred_logits = model(X)

      # 2. Calculate the loss
      loss = loss_fn(test_pred_logits, y)
      test_loss += loss.item()

      # Calculate the accuracy
      test_pred_labels = test_pred_logits.argmax(dim=1)
      test_acc += ((test_pred_labels == y).sum().item()/len(test_pred_labels))

    # Adjust metrics to get average loss and accuracy per batch
    test_loss = test_loss / len(dataloader)
    test_acc = test_acc / len(dataloader)
    return test_loss, test_acc

from tqdm.auto import tqdm

# 1. Create a train function that takes in various model parameters + optimizer + dataloaders + loss function
def train(model: torch.nn.Module,
          train_dataloader: torch.utils.data.DataLoader,
          test_dataloader: torch.utils.data.DataLoader,
          optimizer: torch.optim.Optimizer,
          loss_fn: torch.nn.Module,
          epochs: int = 5,
          device = device):

  # 2. Create empty results dictionary
  results = {"train_loss": [],
             "train_acc": [],
             "test_loss": [],
             "test_acc": []}
  # 3. Loop through training and testing steps for a number of epochs
  for epoch in range(epochs):
    train_loss, train_acc = train_step(model=model,
                                      dataloader=train_dataloader,
                                      loss_fn=loss_fn,
                                      optimizer=optimizer,
                                      device=device)
    test_loss, test_acc = test_step(model=model,
                                    dataloader=test_dataloader,
                                    loss_fn=loss_fn,
                                    device=device)

    # 5. Print out what's happening
    print(f"Epoch: {epoch} | Train loss: {train_loss:.4f} | Train acc: {train_acc:.4f} | Test loss: {test_loss:4f} | Test acc: {test_acc:.4f}")
    results["train_loss"].append(train_loss)
    results["train_acc"].append(train_acc)
    results["test_loss"].append(test_loss)
    results["test_acc"].append(test_acc)

```

```
  # 6. Return the filled results at the end of the loop
  return results
```

2.2.7 Train and evaluate model$_0$

```
# Set number of epochs
NUM_EPOCHS = 12

# Setup loss function and optimizer
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(params=model_0.parameters(),
                             lr=0.001)

# Train model_0
model_0_results = train(model=model_0,
                        train_dataloader=train_dataloader_simple,
                        test_dataloader=test_dataloader_simple,
                        optimizer=optimizer,
                        loss_fn=loss_fn,
                        epochs=NUM_EPOCHS)
```

```
Epoch: 0 | Train loss: 1.1063 | Train acc: 0.3047 | Test loss: 1.098321 | Test acc: 0.3011
Epoch: 1 | Train loss: 1.0998 | Train acc: 0.3281 | Test loss: 1.069704 | Test acc: 0.5417
Epoch: 2 | Train loss: 1.0869 | Train acc: 0.4883 | Test loss: 1.080744 | Test acc: 0.4924
Epoch: 3 | Train loss: 1.0843 | Train acc: 0.4023 | Test loss: 1.060744 | Test acc: 0.5833
Epoch: 4 | Train loss: 1.0662 | Train acc: 0.4102 | Test loss: 1.065383 | Test acc: 0.5644
Epoch: 5 | Train loss: 1.0298 | Train acc: 0.4414 | Test loss: 1.013412 | Test acc: 0.5426
Epoch: 6 | Train loss: 0.9809 | Train acc: 0.4180 | Test loss: 0.931092 | Test acc: 0.6146
Epoch: 7 | Train loss: 0.9571 | Train acc: 0.5859 | Test loss: 1.008202 | Test acc: 0.4744
Epoch: 8 | Train loss: 0.9266 | Train acc: 0.5898 | Test loss: 1.067889 | Test acc: 0.3324
Epoch: 9 | Train loss: 1.0050 | Train acc: 0.4609 | Test loss: 1.061566 | Test acc: 0.4044
Epoch: 10 | Train loss: 0.8879 | Train acc: 0.5195 | Test loss: 0.968867 | Test acc: 0.4830
Epoch: 11 | Train loss: 0.9317 | Train acc: 0.4453 | Test loss: 0.927652 | Test acc: 0.5436
```

2.2.8 Making a prediction on a custom image

```
# Download custom image
import requests

# Setup custom image path
custom_image_path = data_path / "04-pizza-dad.jpeg"

# Download the image if it doesn't already exist
if not custom_image_path.is_file():
  with open(custom_image_path, "wb") as f:
    request = requests.get("https://github.com/mrdbourke/pytorch-deep-learning/blob/main/images/04-pizza-dad.jpeg?raw=true")
    f.write(request.content)
else:
  print(f"{custom_image_path} already exists, skipping download...")
```

```python

# Read in custom image
custom_image_uint8 = torchvision.io.read_image(custom_image_path)
plt.imshow(custom_image_uint8.permute(1, 2, 0))
```


```python
# Create transform pipeline to resize image
from torchvision import transforms
custom_image_transfom = transforms.Compose([
    transforms.Resize(size=(64, 64)),
])

# Transform targe image
custom_image_transformed = custom_image_transfom(custom_image)

plt.imshow(custom_image_transformed.permute(1, 2, 0))
```

Figure 1: Enter Caption

```
model_0.eval()
with torch.inference_mode():
  custom_image_pred = model_1(custom_image_transformed.unsqueeze(0)
    .to(device))

# Convert logits -> pred probs
custom_image_pred_probs = torch.softmax(custom_image_pred, dim=1)
print(custom_image_pred_probs)

# Convert pred prob -> pred labels
custom_image_pred_labels = torch.argmax(custom_image_pred_probs,
    dim=1)
print(custom_image_pred_labels)
print(class_names[custom_image_pred_labels]
```

```
tensor([[0.3693, 0.3537, 0.2770]], device='cuda:0')
tensor([0], device='cuda:0')
pizza
```

3 Bibliography

https://en.wikipedia.org/wiki/Region$_{B}ased_{C}onvolutional_{N}eural_{N}etworks$
https://towardsdatascience.com/review-fcn-semantic-segmentation-eb8c9b50d2d1
https://web.pdx.edu/ jduh/courses/Archive/geog481w07/Students/Ludwig$_{I}mageConvolution.pdf$
https://www.youtube.com/watch?v=V$_{x}ro1bcAuA$