EECS 445

Project 2 Quickstart

October 26, 2023

P2 Intro

Problem: Classifying landmarks in images. Specifically, we care about distinguishing the Pantheon from Hofburg Imperial Palace.

Approach: Build a CNN using PyTorch, experiment with *Transfer Learning* and *Data Augmentation*

Project Logistics

Due on Tuesday, 11/7 at 10:00pm

Submit write-up to Gradescope

Submit challenge CSV to Canvas

Coding questions are highlighted in green, questions with written answers are highlighted in blue.

Sections

Section	*Points*	*Recommended Completion Date*
Data Preprocessing	10 pts	Friday, 10/27
Convolutional Neural Networks	30 pts	Tuesday, 10/31
Visualizing what the CNN has learned	10 pts	Tuesday, 10/31
Transfer Learning & Data Augmentation	25 pts	Saturday, 11/4
Challenge	15 pts	Tuesday, 11/7
Code Appendix	10 pts	Tuesday, 11/7

Dataset

- 5,152 PNG Images
- Located in data/images/
- Each image: 3 x 64 x 64 (RGB color channels)
- 10 landmarks
- 4 partitions: training, validation, test, held-out
- 2 tasks (target and source)
  - Target: Classify the Pantheon and Hofburg Imperial Palace
  - Source: Classify many other landmarks
- metadata found in data/landmarks.csv

Training CNNs

- You will implement the **architecture** of the CNN using layer classes in PyTorch and the parameter values given in the spec appendix
- You will also implement the **weight initialization** of the layers and **forward pass** through the network
  - This defines the order of operations an input datapoint goes through in your model
- Note that PyTorch automatically handles the backpropagation calculations. You just need to fill in the train_epoch function
  - *Hint*: Look at the PyTorch documentation examples!
- You will also implement **early stopping**
  - This is used to prematurely stop training at a certain epoch when you observe validation loss **has not decreased** for a predetermined number of epochs (called the patience!)

Training CNNs Cont.

- The *Criterion* in the appendix refers to the **loss function** we use
  - We use Cross Entropy Loss

$$\mathcal{L}(\bar{y}, \hat{y}) = -\sum_{c} \bar{y}_c \log \hat{y}_c$$

True label $\bar{y}$

Predicted label $\hat{y}$

- The *Optimizer* in the appendix refers to the tool we use to minimize the loss function
  - We use the Adam technique, which is a form of stochastic gradient descent

Target Architecture

- Task: build CNN architecture for target task as specified in Appendix B
- Architecture (target.py)
  - __init__(): construct each layer of the convolutional neural network
  - init_weights(): initialize parameters (weights + bias) for each layer
  - forward(x): define forward propagation for a batch of input examples
- Model parameters: criterion, optimizer, learning rate, patience, batch size

Layer 0: Input image

Layer 1: Convolutional Layer 1

Layer 2: Max Pooling Layer

Layer 3: Convolutional Layer 2

Layer 4: Max Pooling Layer

Layer 5: Convolutional Layer 3

Layer 6: Fully connected layer 1

Torch.nn documentation: https://pytorch.org/docs/stable/nn.html

Example Layers

Training Parameters

- Criterion: `torch.nn.CrossEntropyLoss`
- Optimizer: `torch.optim.Adam`
- Learning rate: 10^{-3}
- Patience: 5
- Batch Size: 32

Layer 1: Convolutional Layer 1

- Number of filters: 16
- Filter size: 5×5
- Stride size: 2×2
- Padding: SAME
  - Note: Conv2D doesn't allow the argument padding='same' if stride size is not 1, so you have to calculate the padding manually.
- Activation: ReLU
- Weight initialization: normally distributed with $\mu = 0.0$, $\sigma = \frac{1}{\sqrt{5 \times 5 \times num\_input\_channels}}$
- Bias initialization: constant 0.0
- Output: $16 \times 32 \times 32$

Layer 2: Max Pooling Layer

- Kernel size: 2×2
- Stride: 2×2
- Padding: none

Layer 6: Fully connected layer 1 (Output layer)

- Input: 32
- Activation: None
- Weight initialization: normally distributed with $\mu = 0.0$, $\sigma = \frac{1}{\sqrt{input\_size}}$
- Bias initialization: constant 0.0
- Output: 2 (number of classes)

Grad-CAM

- Tool to help us visualize which areas of the image *contribute the most to each class prediction*
- You will use this paper to answer the questions for this section
  - You will manually calculate the heatmap produced by this tool for one class label
  - Intuitively, the highlighted areas of the image show characteristics that are highly correlated with a certain class label. This means you will find useful areas of the image at a particular step for predicting a certain class label

Figure 3: Grad-CAM for the 'dog' label

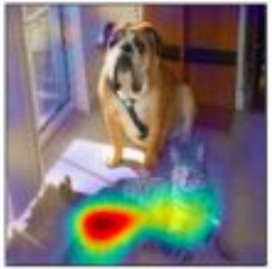
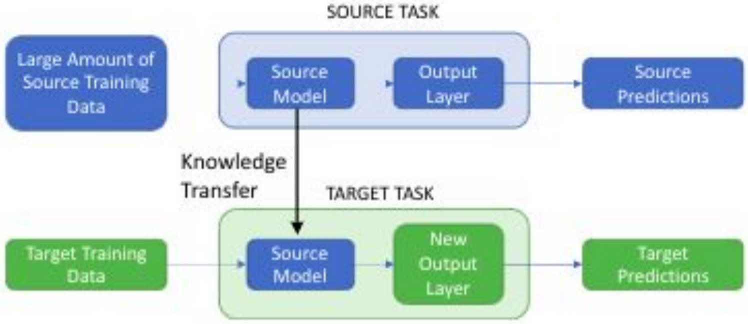


Figure 4: Grad-CAM for the 'cat' label

Transfer Learning

Transfer Learning

- Sometimes a certain task is related to other classification tasks such that the model parameters learned for one task are informative to the other
- We call the task we want to learn the **target** task and the task we already have a model for the **source** task
- The idea is to use the layers/weights from the **source** task in the model of the **target** task in the hopes that similar features will be extracted and used from the input



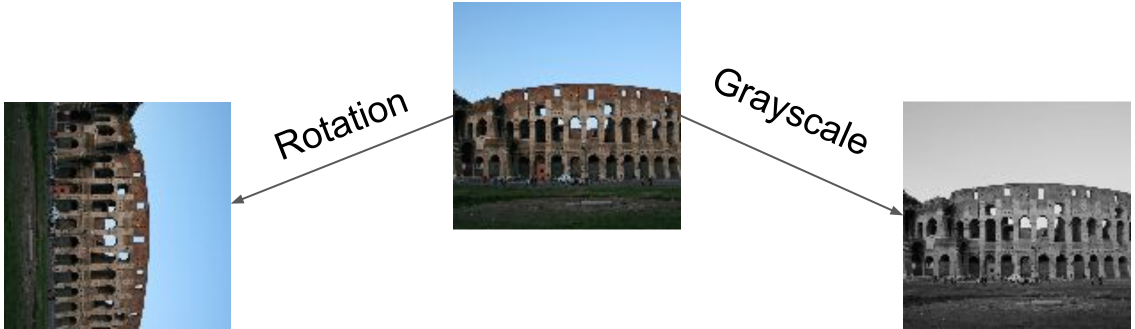
Transfer Learning Cont.

- Recall that the **target task** is classifying the Pantheon and Hofburg Imperial Palace
- The **source task** you will use in this project is classifying 8 different landmarks besides the Pantheon and Hofburg Imperial Palace
- We hypothesize that the features we learn in the **source task** (i.e. features of different landmarks) will be useful in our **target task**
- The concept of **freezing layers** allows us to control the extent to which we leverage the source task in our training of the target task

Data Augmentation

Data Augmentation

- We may also observe that our training data is not entirely representative of the possible inputs to our model
- We can introduce **new** training data by altering the existing training data slightly and adding it to the training set
- Alterations include **rotating** and **grayscaling** the images in this project
  - Can you think of more? Try them in the Challenge section!



Challenge

Train a CNN to solve the target task (classifying the Pantheon and Hofburg Imperial Palace)

Consider:

- Regularization (weight decay, dropout, etc.)
- Feature Selection
- Model Architecture
- Hyperparameters
- Transfer Learning
- Data Augmentation

PyTorch Tutorial

Pytorch Tensors

```python
import torch

A = torch.rand((3,5))
B = torch.rand((3,5))

print(A+B)             # prints element-wise sum of matrices
print(A.sum(dim=1)) # prints sum of each row of A
```

Defining a Neural Net

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


class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__() # call superclass constructor

        # Step 1. Define fully-connected layers for the neural network
        self.hidden1 = nn.Linear(2, 3)
        self.hidden2 = nn.Linear(3, 2)
        self.output = nn.Linear(2, 2)

    def forward(self, x):
        # Step 2. Override the forward function to define the forward pass
        z2 = self.hidden1(x)
        h2 = F.sigmoid(z2)
        z3 = self.hidden2(h2)
        h3 = F.sigmoid(z3)
        z4 = self.output(h3)
        return z4
```

Defining Model Architectures

To define the layers that are part of your architecture, PyTorch requires that you instantiate each layer object as a member variable of your neural network class. This is done in the constructor of the new class.

```
def __init__(self):
    super(Net, self).__init__() # call superclass constructor

    # Step 1. Define fully-connected layers for the neural network
    self.hidden1 = nn.Linear(2, 3)
    self.hidden2 = nn.Linear(3, 2)
    self.output = nn.Linear(2, 2)
```

Implementing the Forward Pass

```
    def forward(self, x):
        # Step 2. Override the forward function to define the forward pass
        z2 = self.hidden1(x)
        h2 = F.sigmoid(z2)
        z3 = self.hidden2(h2)
        h3 = F.sigmoid(z3)
        z4 = self.output(h3)
        return z4
```

The Dataset Class

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

class ExampleDataset(Dataset):
    def __init__(self, features, labels):
        self.features = features
        self.labels = labels

    def __len__(self):
        return self.features.shape[0]

    def __getitem__(self, index):
        return self.features[index], self.labels[index]
```

The DataLoader Class

Constructing a `DataLoader` requires passing in a subclass of `Dataset` to create batches from, along with the batch size required. Assuming that we use the `train_dataset` and `test_dataset` variables defined in the last code snippet, we can define wrapping data loaders as follows (where we use a mini-batch size of 32 examples for both):

```
from torch.utils.data import DataLoader

train_loader = DataLoader(train_dataset, batch_size=32)
test_loader = DataLoader(test_dataset, batch_size=32)
```

Training Neural Networks

1. Create an instance of our neural network using the definition from Section 2.

```
net  = Net()
```

2. Create training and testing datasets and data loaders using the definitions from Section 3.

```
train_dataset = ExampleDataset(X_train, y_train)
test_dataset = ExampleDataset(X_test, y_test)

train_loader = DataLoader(train_dataset, batch_size=32)
test_loader = DataLoader(test_dataset, batch_size=32)
```

3. Define your loss function (`criterion`) and your optimization algorithm (`optimizer`). Here we're using cross entropy loss and SGD with our learning rate of $\eta = 0.01$.

```
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.01)
```

Training Neural Networks

4. Define the PyTorch training loop as seen below:

```python
for epoch in range(10):  # loop over the dataset multiple times
    total_train_loss = 0.0
    for i, data in enumerate(train_loader, 0):
        # get the inputs
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + update step
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        total_train_loss += loss.item()

    # the following line disables gradient calculations
    with torch.no_grad():
        # print the average test loss for the model after training on this epoch
        total_test_loss = 0
        for inputs, labels in test_loader:
            outputs = net(inputs)
            loss = criterion(outputs, labels)
            total_test_loss += loss
    print(
        "Epoch %d : Train Loss: %.3f, Test Loss: %.3f"
        % (
            epoch + 1,
            total_train_loss / len(train_loader),
            total_test_loss / len(test_loader),
        )
    )
    running_loss = 0.0

print("Finished Training")
```

Thank you!