Classifying images of lung tomographys

Table of contents

Summary

Multi Layer Perceptron - 0.46256
XGBOOST Clasifier - 0.64102
VGG16 CNN - 0.67282

Final models: VGG16 CNN and XGBOOST

Introduction

CT scans provide a lot of insight into pulmonary affections or other diseases. The use of ML in this domain has accelerated in the past few years, especially with the coronavirus crisis. Research has shown that using machine learning can provide valuable information and diagnostics. In this documentation I will present my approaches in trying to discriminate CT scans between one of the three classes (native, arterial, venous).

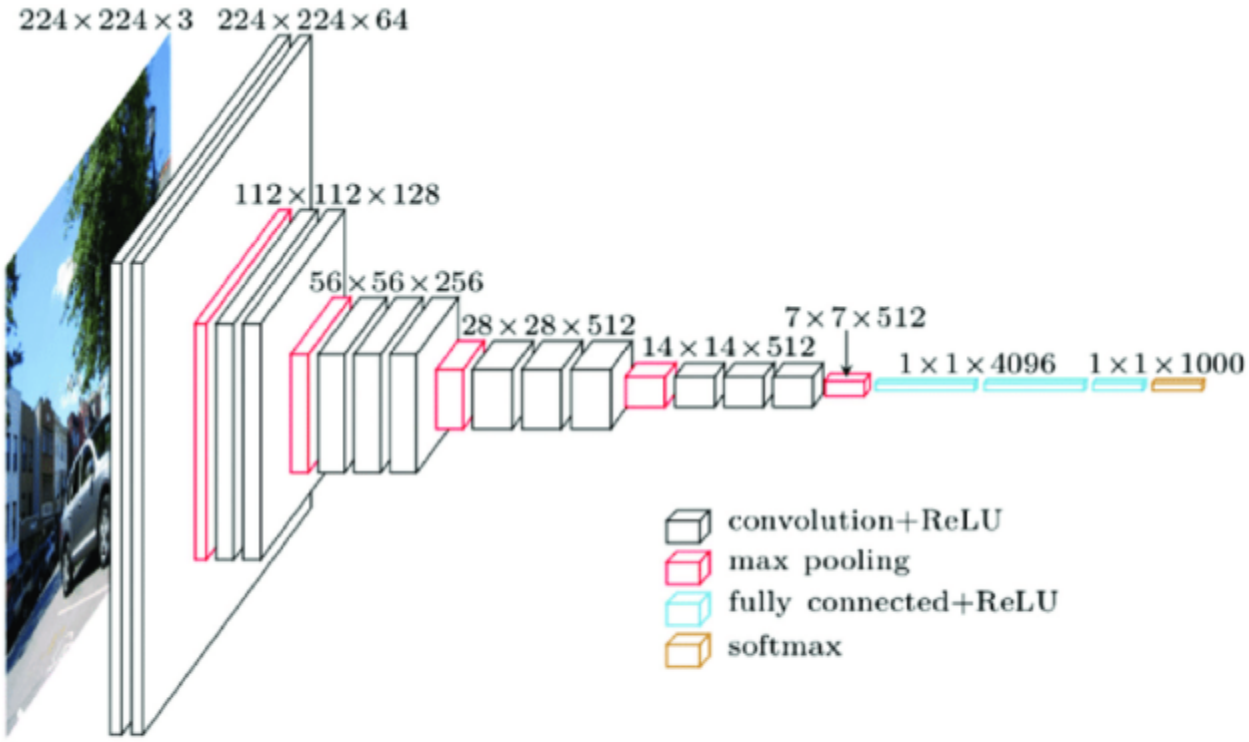
Chosen Features

The dataset is comprised of CT scans from 3 classes. Since the dataset is visual, I opted to use the images themselves as the feature. I applied preprocessing to the data, mainly scaling it. I used the StandardScaler from sklearn in order to aid the model in learning the differences between the classes. This approach proved to be lucrative and improved the results. I am sure that with more detailed preprocessing I could have boosted my accuracy higher.

Chosen model

Since this task is a image classification one, I chose traditional models and newer ones that are used in similar tasks. I started with a Multilayer Perceptron that yielded decent results, but the complexity of the data was too big for this model to be effective. I moved onto a XGBoost classifier, which proved to be a better option than the MLP Classifier. With a bit of tweaking I managed to get a big improvement over the first model. Afterwards, I tried to use a VGG16-like network, which yielded the best results from all of the models I tried.

The CNN uses multiple convolutional layers which transform the image into features. After each set of convolutions, the input is then passed through a Max Pooling layer. Then, I applied regularization, using Dropout layers, with p=0.25. After all of these layers, the input is fed through a standard neural network, in order to process the features into one of the 3 available classes. As said earlier, this model, with a bit of fine-tuning and adjustment, proved to be my best attempt at the task.



vgg-16 architecture
https://www.researchgate.net/publication/328966158/figure/fig2/AS:693278764720129@1542301946576/An-overview-of-the-VGG-16-model-architecture-this-model-uses-simple-convolutional-blocks.png

Chosen approach

I chose to implement the VGG-16 CNN in pytorch, since it is a very flexible and widely-used library. For image manipulation and imports I used pillow. Other libraries include sklearn (for scaling), numpy (for vector manipulation) and pandas (for the initial data import).

```
import pandas as pd
import numpy as np
from PIL import Image
from tqdm import tqdm
import torch.optim as optim
import torch.nn.functional as F
import torch.nn as nn
import torch
from torch.utils.data import TensorDataset, DataLoader
import os
import matplotlib.pyplot as plt
import csv
```

Since pytorch is a flexible but verbose library, I had to implement the VGG16 classifier by hand. I used already existing architecture descriptions for the implementation. I started with the Conv2d layer clusters, using a kernel size of 3. The max pool has a size of 2. The linear fully connected layers are a standard size of 4096 neurons, into the output of 3 classes. I used relu for the neuron activation and dropouts on the fully connected layers in order to reduce bias and overfit. As for the loss function I used cross entropy, since it is widely used in multi-class classification tasks.

I chose to train on the GPU since it is faster, thus I had to install CUDA and use it as a device. The loading of the data is simple, I looped over the file entries and appended the images to their respective train, validation and test arrays. I then created the tensor datasets and respective data loaders to be used in the training part. I used the Adam optimizer, since it converges quite quickly.

```python
class VGG(nn.Module):
    def __init__(self):
        super().__init__()

        # 1 channel input, since we are using grayscale images
        self.layer1_1 = nn.Conv2d(1, 32, kernel_size=3, padding=1)
        self.layer1_2 = nn.Conv2d(32, 32, kernel_size=3, padding=1)
        self.layer1_mp = nn.MaxPool2d(2, 2)

        self.layer2_1 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.layer2_2 = nn.Conv2d(64, 64, kernel_size=3, padding=1)
        self.layer2_mp = nn.MaxPool2d(2, 2)

        self.layer3_1 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.layer3_2 = nn.Conv2d(128, 128, kernel_size=3, padding=1)
        self.layer3_3 = nn.Conv2d(128, 128, kernel_size=3, padding=1)
        self.layer3_mp = nn.MaxPool2d(2, 2)

        self.layer4_1 = nn.Conv2d(128, 256, kernel_size=3, padding=1)
        self.layer4_2 = nn.Conv2d(256, 256, kernel_size=3, padding=1)
        self.layer4_3 = nn.Conv2d(256, 256, kernel_size=3, padding=1)
        self.layer4_mp = nn.MaxPool2d(2, 2)

        self.layer5_1 = nn.Conv2d(256, 512, kernel_size=3, padding=1)
        self.layer5_2 = nn.Conv2d(512, 512, kernel_size=3, padding=1)
        self.layer5_3 = nn.Conv2d(512, 512, kernel_size=3, padding=1)
        self.layer5_mp = nn.MaxPool2d(2, 2)

        # conv layers done, time for linear
        self.fc1 = nn.Linear(512 * 1 * 1, 4096)
        self.fc2 = nn.Linear(4096, 4096)
        self.fc3 = nn.Linear(4096, 3)
```

```
def forward(self, x):
    x = torch.unsqueeze(x, 1)

    x = F.relu(self.layer1_1(x))
    x = F.relu(self.layer1_2(x))
    x = self.layer1_mp(x)

    x = F.relu(self.layer2_1(x))
    x = F.relu(self.layer2_2(x))
    x = self.layer2_mp(x)

    x = F.relu(self.layer3_1(x))
    x = F.relu(self.layer3_2(x))
    x = F.relu(self.layer3_3(x))
    x = self.layer3_mp(x)

    x = F.relu(self.layer4_1(x))
    x = F.relu(self.layer4_2(x))
    x = F.relu(self.layer4_3(x))
    x = self.layer4_mp(x)

    x = F.relu(self.layer5_1(x))
    x = F.relu(self.layer5_2(x))
    x = F.relu(self.layer5_3(x))
    x = self.layer5_mp(x)

    # we need to flatten x for the linear layers
    x = x.view(-1, 512 * 1 * 1)

    x = F.relu(self.fc1(x))
    x = F.dropout(x, 0.25)
    x = F.relu(self.fc2(x))
    x = F.dropout(x, 0.25)
    x = self.fc3(x)

    return x
```

Hyperparameter choices

The main hyperparameter I tweaked with is the learning rate. A learning rater of 5e-5 proved to be the most lucrative. Other values were tried but they either converged too fast and overfitted, or took way too long to get anywhere. For this reason I used Adam over SGD as the optimiser. Despite being jittery, it converges way faster. Also, by using weight decay, it can be made to converge slower after the initial burst of learning. In addition, I also played around with regularization (dropout probability, I settled on 0.25) and other loss functions (binary cross entropy with logits, etc), but the plain cross entropy proved to be the best for this scenario.

Results

From the confusion matrix we can see that the class with the least false positives was the first one. The other two classes had a lot of false positives, probably because they were the ones with the least visual differences.

```
[[1305   92  103]
 [ 315  748  437]
 [ 176  327  997]]
              precision    recall  f1-score   support

           0       0.73      0.87      0.79      1500
           1       0.64      0.50      0.56      1500
           2       0.65      0.66      0.66      1500
```

Conclusion

Overall, I have achieved meager results on this dataset, but I am sure that with more preprocessing of the images a better accuracy would have been achieved.