CT Scan Classification

Moroianu Theodor

May 19, 2021

Contents

1 Introduction **3**

2 Discrepancy between Validation and Testing **3**

3 First Approaches: KNN, Naive-Bayes **3**

4 Second Approach: Keras API **3**

5 Third Approach: Pytorch **4**
5.1 Defining Network 4
5.2 Ray Tune 5
5.3 FastAI 6

6 Final Approach: Pytorch + Transfer-Learning **7**
6.1 Torchvision Models 7
6.2 Data Augmentation 7
6.3 Training 8
6.4 Dataset Modifications 8

7 Benchmarks **9**
7.1 Pytorch Resnet 9
7.2 Pytorch CNN 10

8 Hyperparameter tuning **11**

1 Introduction

In this paper I will talk about my approaches, failures and results while building a model able to classify the CT Scan images given in the competition. [2]

As neither in the course nor in the laboratory we were shown how to use high-level libraries like *Pytorch*[6] or *Tensorflow*[7], I found the need to search online deep-learning courses. I found an awesome course on *Udacity*[4], where I learned how to use *Pytorch* for training deep-learning models.

I tried most of the state-of-the-art deep-learning frameworks (*Pytorch*, *Tensorflow*, *Keras* and *FastAI*), my best results being with pure *Pytorch*. I also used, without any success, the *scikit-learn* library, for implementing *KNN* and *Naive-Bayes* classifiers.

2 Discrepancy between Validation and Testing

While submitting my answers to *Kaggle*, I noticed a non-negligible drop in accuracy on the visible part of the test data, in comparison to the validation data. I am talking about more a than 15% difference in accuracy, which couldn't appear because of validation over-fitting, as I of course didn't train my models on the validation data, and only used the validation data to know when to early stop.

As a 15% drop is statistically impossible, I asked the teacher if the testing data is part of the same dataset and has the same class distribution as the train and validation data, which he confirmed isn't the case. I talked with other students which confirmed that validation accuracy isn't tightly related to testing accuracy, and at the end found a way to reduce the accuracy drop from 15% to just over 7%.

3 First Approaches: KNN, Naive-Bayes

Based on the belief that the competition was solvable with knowledge accumulated at the lab, I tried to generate a *KNN* approach. It performed really badly on validation data, and while I didn't submit it (we only have 2 submissions per day), someone else had the exact same approach, and scored barely above 50% on Kaggle. This should come at no surprises, as *KNN* is known to perform poorly on image classification tasks [5]. The *Naive-Bayes* classifier performed even worse, so I dismissed the idea of such approaches.

4 Second Approach: Keras API

Keras is one of the most known deep-learning libraries out there, so I decided to give it a try. A relatively complex network was fairly easy to build, and after the tedious task of installing the *cuda* version of *Tensorflow* I was ready to train my network.

The network giving me the best results is the following:

```
model = Sequential()
model.add(Conv2D(32, 7, activation='relu', input_shape=(50, 50, 1)))
```

```
model.add(BatchNormalization())
model.add(Conv2D(64, 5, activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(32, 3, activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(100, activation='relu',
            kernel_regularizer=tf.keras.regularizers.l2(0.001)))
model.add(Dropout(0.5))
model.add(Dense(100, activation='relu',
            kernel_regularizer=tf.keras.regularizers.l2(0.01)))
model.add(Dense(3, activation='softmax'))
model.compile(loss='categorical_crossentropy',
        optimizer='adam', metrics=['accuracy'])
```

While it gave ok-ish results, I disliked the lack of control *Keras* gave me. On the other hand, pure *Tensorflow* was too complicated when compared to other frameworks, so I abandoned the *Keras* approach.

5 Third Approach: Pytorch

5.1 Defining Network

For defining the network, I used the *torch.nn.Module* class functionalities. As such, my network is the following:

```
class ConvUnit(nn.Module):
    def __init__(self, in_f, out_f, ker=3, dropout=0.05, max_pull=False):
        super().__init__()
        if max_pull == False:
            self.net = nn.Sequential (
                nn.Conv2d(in_f, out_f, kernel_size=ker, padding=ker//2),
                nn.BatchNorm2d(out_f),
                nn.Dropout2d(dropout),
                nn.ReLU()
            )
        else:
            self.net = nn.Sequential (
                nn.Conv2d(in_f, out_f, kernel_size=ker,
                    padding=ker//2, stride=2),
                nn.BatchNorm2d(out_f),
                nn.Dropout2d(dropout),
```

```
                nn.ReLU()
            )
    def forward(self, x):
        return self.net(x)

class FcUnit(nn.Module):
    def __init__(self, in_f, out_f, dropout=0.3):
        super().__init__()
        self.net = nn.Sequential (
            nn.Linear(in_f, out_f),
            nn.BatchNorm1d(out_f),
            nn.Dropout(dropout),
            nn.ReLU()
        )
    def forward(self, x):
        return self.net(x)

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            ConvUnit(3, 16, ker=5),
            ConvUnit(16, 32, ker=3, max_pull=True, dropout=0.1),
            ConvUnit(32, 64, ker=3, max_pull=True, dropout=0.1),
            ConvUnit(64, 128, ker=3, max_pull=True, dropout=0.1),

            nn.Flatten(),

            FcUnit(128 * 13 * 13, 1024, dropout=0.4),
            FcUnit(1024, 512, dropout=0.4),
            FcUnit(512, 3, dropout=0.)
        )

    def forward(self, x):
        return self.net(x)
```

As one can see, the network is highly flexible, and is made of two parts:

- The features extraction, made with the help of 2d convolutions.
- The classification, made with the help of fully connected layers.

5.2 Ray Tune

For improving the network defined above, I decided to perform hyper-parameter tuning. I decided to use the *ray.tune* library, which helped automate the hyper-parameter search[8].

The model I used with Ray-Tune is:

```
class Model(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.net = nn.Sequential(
            ConvUnit(1, 32, ker=config['ker1']),
            ConvUnit(32, config['d1'], ker=config['ker1'],
                    max_pull=True, dropout=config['drop1']),
            ConvUnit(config['d1'], config['d2'], ker=3,
                    max_pull=True, dropout=config['drop1']),

            nn.Flatten(),

            FcUnit(config['d2'] * 13 * 13, config['fc1'],
                    dropout=config['drop2']),
            FcUnit(config['fc1'], config['fc2'], dropout=config['drop3']),
            FcUnit(config['fc2'], 3, dropout=0.)
        )

    def forward(self, x):
        return self.net(x)
```

We can see that the *config* object, given by *Ray.Tune*. Finding the best configuration can be done by specifying the search grid:

```
config = {
    "ker1": tune.choice([3, 5, 7]),
    "d1": tune.choice([32, 64, 128]),
    "d2": tune.choice([64, 128, 256]),
    "fc1": tune.choice([100, 500, 1000]),
    "fc2": tune.choice([100, 250, 500]),
    "drop1": tune.sample_from(lambda _: np.random.uniform(0., 0.2)),
    "drop2": tune.sample_from(lambda _: np.random.uniform(0.3, 0.7)),
    "drop3": tune.sample_from(lambda _: np.random.uniform(0., 0.7)),
    "batch_size": tune.choice([32, 64])
}
```

While *Ray* helped me find better hyper-parameters, it didn't drastically improve the network.

5.3 FastAI

FastAI is both a course and a *Pytorch* deep-learning library[3], which offers features similar to the automated learning in *Keras*:

```
net = Model()
def opt_func(params, **kwargs):
    return OptimWrapper(optim.AdamW(params, **kwargs))

dls = DataLoaders(train_loader, validation_loader)

learn = Learner(dls, net, loss_func=criterion, cbs=[CudaCallback])
learn.fit(300)
```

The code above trains the *net* network automatically. However, *FastAI* was removing some of the liberty I was looking for in *Pytorch*, so I decided to stop using it.

6 Final Approach: Pytorch + Transfer-Learning

6.1 Torchvision Models

In my final attempt to implement an easy-to-use and powerful network, I turned to the *torchvision.models* module of *Pytorch*.

Using a *resnet34* architecture, my final network looks like this:

```
class Resnet(nn.Module):
    def __init__(self):
        super().__init__()
        self.resnet = models.resnet34(pretrained=True)
        self.fc = nn.Sequential(
            nn.Dropout(0.3),
            nn.Linear(1000, 3)
        )

    def forward(self, x):
        x = self.fc(self.resnet(x))
        return x
```

6.2 Data Augmentation

As we have a small training set, I decided to use data augmentation, which can be easily implemented in *Pytorch* with the help of a custom transformation:

```
RESIZE = 200
p_enc = PositionalEncodingPermute2D(3)
p_enc_filter = p_enc(th.zeros(1, 3, RESIZE, RESIZE))[0]
datapath = "data"

class PEnc(object):
```

```
    def __init__(self):
        pass
    def __call__(self, x):
        return x + p_enc_filter / 10

train_transform = transforms.Compose([
        transforms.Resize(RESIZE),
        transforms.ToTensor(),
        transforms.RandomRotation(degrees=20),
        transforms.RandomCrop(RESIZE, padding=10),
        transforms.RandomAutocontrast(),
        transforms.RandomAdjustSharpness(0.95),
        transforms.RandomResizedCrop(RESIZE, scale=(0.8, 1)),
        PEnc(),
])
```

We can observe that:

- I resized the images to $200x200$ as the resnet architecture I used didn't really like small images.
- I applied some randomized data augmentation like random rotations, random contrasts and random resized crops.
- I added a 2d positional encoding to the image, to help with the model's attention[1]. Note that the positional encoding gave mixed results, the model performing better sometimes without it.

6.3 Training

I trained the model with the *Adam* optimizer, with various learning rates between $3e-4$ and $5e-7$. The model gave best performances and started to converge after 30 epochs, each of which took me 100 seconds on my *GTX 1060* graphics card.

6.4 Dataset Modifications

For enhancing the model's performances, I decided during the last training cycles of my final model to modify the division of the training and validation data.

As such, before the modification, the dataset was devided into:

- $5000x3$ images for training.
- $1500x3$ images for validation.

After my modification, the dataset contained:

- $6000x3$ images for training.

- $500x3$ images for validation.

While this modification reduced the precision of the validation accuracy, I already had a good idea of the performance of the model, and just wanted to train it on more data.

7 Benchmarks

I bench-marked my best two models.

7.1 Pytorch Resnet

Train is made by adjusting the learning rate each few epochs. For the first few epoches, the resnet layers are frozen, to train the classifier.

When the network starts converging, the rest of the layers are un-frozen.

Epoch	Train Loss	Train Accuracy	Validation Loss	Validation Accuracy
1	0.699	66.77%	0.78	60.51%
2	0.527	75.55%	0.642	68.71
3	0.425	81%	0.546	74.71%
4	0.34	85.1%	0.51	75.91%
5	0.26	88.81%	0.54	78.98%
6	0.22	91%	0.57	79.2%
7	0.17	92.9%	0.45	83%

The accuracy graph is the following:

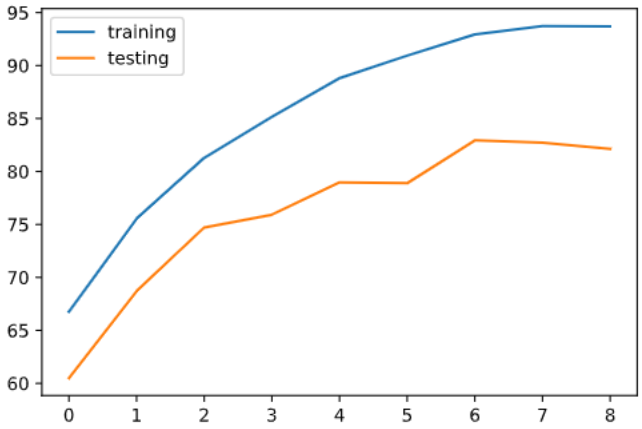

Figure 1: Training and Validation Accuracy of the Resnet Model

The confusion matrix, interpreted as an image is the following:

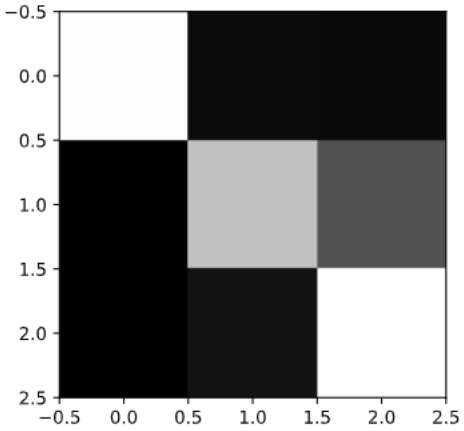

Figure 2: Confusion Matrix of the Resnet Model

The confusion matrix's numeric values are:

1323	89	88
37	1016	447
32	132	1336

7.2 Pytorch CNN

The first few training epochs are given below:

Epoch	Train Loss	Train Accuracy	Validation Loss	Validation Accuracy
1	1.0553	43.99%	1.0464	49.36%
2	0.9986	46.89%	0.9136	56.51%
3	1.0062	47.21%	0.9492	45.31%
4	0.8651	59.63%	0.7922	61.11%
5	0.8345	61.17%	0.7668	61.71%
6	0.8181	61.88%	0.7639	62.89%
7	0.8072	63.47%	0.7501	64.18%
8	0.8099	63.50%	0.7529	63.67%

The accuracy graph is the following:

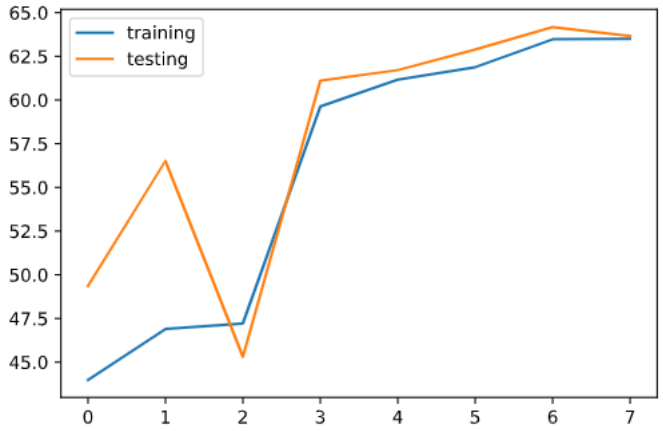

Figure 3: Training and Validation Accuracy of the CNN Model

The confusion matrix, interpreted as an image is the following:

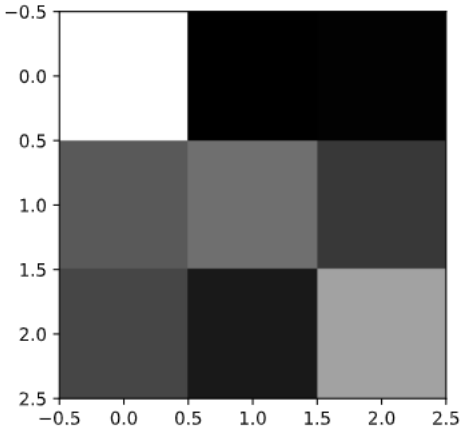

Figure 4: Confusion Matrix of the CNN Model

The confusion matrix's numeric values are:

1380	53	67
523	631	346
418	185	897

8 Hyperparameter tuning

For finding the best hyperparameters, I first tried to use the *ray.tune* library. As the library was really slow, and hard to use, I decided to stop using it, and to manually perform a random-search of the best hyperparameters.

Parameters I have tested are:

- Data augmentation filters. This includes, between others:
  - Image resizing. I found that a resize to $200x200$ worked best.
  - Image rotation. Overfitting was minimized when rotating training images with a random value between $-20°$ and $20°$.
  - Image cropping. Cropping 10-20 pixels from the image's borders worked best.
  - Various strengths of sharpness / brightness.
- Various depths of the network.
- Various dropdown strengths.
- Various CNN kernel sizes.
- Various fully connected sizes, of CNN filters.

Note that my final network, as it is using a pretrained *resnet* architecture, doesn't require most of the hyperparameter search.

As plotting the results for all the parameters over all the epoches would be too long, I made a table ilustrating the results of random manual search on the *resnet* architecture.

The table below represents results after **one** epoch. While I trained it longer, representing all this data would be too tedious. All the trials were ran with the *Adam* optimizer, with a learning rate of 10^{-4}

Trial	RRotation	RCrop	Resize	CDropout	TLoss	VLoss	Time
1	20	0	50	0.3	0.57	0.76	33.1
2	25	10	100	0.3	0.49	0.61	49.2
3	10	15	150	0.1	0.4	0.47	86.3
4	20	10	200	0.2	0.57	0.9	132
5	5	20	200	0.3	0.38	0.54	135
6	30	25	150	0.4	0.49	0.57	91
7	45	15	150	0.1	0.53	0.56	92
8	20	40	150	0.2	0.41	0.45	89

Note that:

- *Trial* represents the ID of the trial.
- *RRotation* means the maximal magnitude of random rotations.
- *RCrop* means the maximal crop of the images.
- *Resize* means the resize dimension of the images.
- *CDropout* means the probability of dropout in the classifier.
- *CSize* means the size of the fully connected layer from the classifier.

- *TLoss* is the training loss after the first epoch.
- *VLoss* is the validation loss after the first epoch.

We can plot each run, where scatter points represent train / validation loss, and point size represent run time:

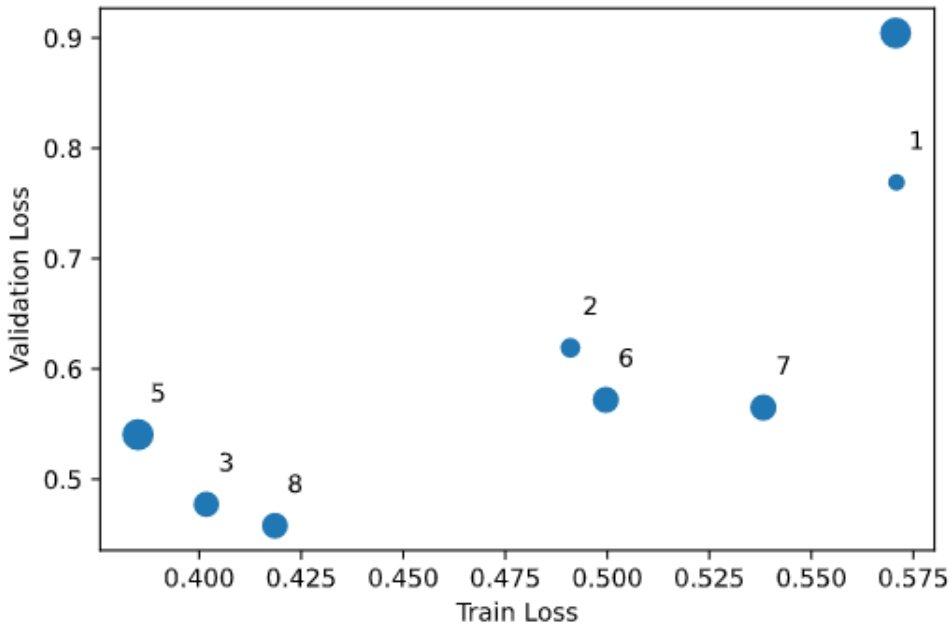

Figure 5: Random Search Results

References

[1] *Attention Is All You Need*. URL: https://arxiv.org/pdf/1706.03762.pdf.

[2] *CT Scan Classification*. URL: https://www.kaggle.com/c/ai-unibuc-23-31-2021/.

[3] *FastAI in Pytorch Course*. URL: https://course.fast.ai/.

[4] *Intro to Deep Learning with PyTorch*. URL: https://classroom.udacity.com/courses/ud188.

[5] *k-NN classifier for image classification*. URL: https://www.pyimagesearch.com/2016/08/08/k-nn-classifier-for-image-classification/.

[6] *Pytorch*. URL: https://pytorch.org/.

[7] *TensorFlow*. URL: https://www.tensorflow.org/.

[8] *Tune: Scalable Hyperparameter Tuning*. URL: https://docs.ray.io/en/master/tune/index.html.