Venelin Valkov

Get SH*T Done with PyTorch

Solve Real-World Machine Learning Problems

Get SH*T Done with PyTorch

Solve Real-World Machine Learning Problems

Venelin Valkov

This book is for sale at http://leanpub.com/getting-things-done-with-pytorch

This version was published on 2020-06-27

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

1. Getting Started with PyTorch

PyTorch[1] is:

> An open source machine learning framework that accelerates the path from research prototyping to production deployment.

In my humble opinion, PyTorch is the sweet[2] way to solve Machine Learning problems, in the real world! The vast community allows you to work state-of-the-art models and deploy them to production in no time (relatively speaking). Let's get started!

```
In [0]: !pip install -q -U torch watermark

In [0]: %load_ext watermark
        %watermark -v -p numpy,torch

Out[0]: CPython 3.6.9
        IPython 5.5.0

        numpy 1.17.5
        torch 1.4.0
```

PyTorch NumPy

Do you know NumPy? If you do, learning PyTorch will be a breeze! If you don't, prepare to learn the skills that will guide you on your journey Machine Learning Mastery!

Let's start with something simple:

[1] https://pytorch.org/
[2] https://www.youtube.com/watch?v=-h4spfuMGDI

```
In [0]: import torch
        import numpy as np

In [0]: a = np.array([1, 2])
        b = np.array([8, 9])

        c = a + b
        c

Out[0]: array([ 9, 11])
```

Adding the same arrays with PyTorch looks like this:

```
In [0]: a = torch.tensor([1, 2])
        b = torch.tensor([8, 9])

        c = a + b
        c

Out[0]: tensor([ 9, 11])
```

Fortunately, you can go from NumPy to PyTorch:

```
In [0]: a = torch.tensor([1, 2])

        a.numpy()

Out[0]: array([1, 2])
```

and vice versa:

```
In [0]: a = np.array([1, 2])
        torch.from_numpy(a)

Out[0]: tensor([1, 2])
```

The good news is that the conversions incur almost no cost on the performance of your app. The NumPy and PyTorch store data in memory in the same way. That is, PyTorch is reusing the work done by NumPy.

Tensors

Tensors are just n-dimensional number (including booleans) containers. You can find the complete list of supported data types at PyTorch's Tensor Docs[3].

So, how can you create a Tensor (try to ignore that I've already shown you how to do it)?

```
In [0]: torch.tensor([[1, 2], [2, 1]])

Out[0]: tensor([[1, 2],
                [2, 1]])
```

You can create a tensor from floats:

```
In [0]: torch.FloatTensor([[1, 2], [2, 1]])

Out[0]: tensor([[1., 2.],
                [2., 1.]])
```

Or define the type like so:

```
In [0]: torch.tensor([[1, 2], [2, 1]], dtype=torch.bool)

Out[0]: tensor([[True, True],
                [True, True]])
```

You can use a wide range of factory methods to create Tensors without manually specifying each number. For example, you can create a matrix with random numbers like this:

```
In [0]: torch.rand(3, 2)

Out[0]: tensor([[0.6686, 0.7622],
                [0.0341, 0.5835],
                [0.2423, 0.0651]])
```

Or one full of ones:

[3] https://pytorch.org/docs/stable/tensors.html

```
In [0]: torch.ones(3, 2)

Out[0]: tensor([[1., 1.],
                [1., 1.],
                [1., 1.]])
```

PyTorch has a variety of useful operations:

```
In [0]: x = torch.tensor([[2, 3], [1, 2]])
        print(x)
        print(f'sum: {x.sum()}')

Out[0]: tensor([[2, 3],
                [1, 2]])
        sum: 8
```

Get the transpose of a 2-D tensor:

```
In [0]: x.t()

Out[0]: tensor([[2, 1],
                [3, 2]])
```

Get the shape of each dimension:

```
In [0]: x.size()

Out[0]: torch.Size([2, 2])
```

Generally, performing some operation creates a new Tensor:

```
In [0]: y = torch.tensor([[2, 2], [5, 1]])
        z = x.add(y)
        z

Out[0]: tensor([[4, 5],
                [6, 3]])
```

But you can do it in-place:

```
In [0]: x.add_(y)
        x

Out[0]: tensor([[4, 5],
                [6, 3]])
```

Almost all operations have an in-place version - the name of the operation, followed by an underscore.

Running on GPU

At this point, you might be like: "Why do I need PyTorch at all? All of this is perfectly doable with NumPy?". PyTorch has three major superpowers: - you can run your operations on the GPU(s) (or something else) - Autograd: automatic differentiation[4] - A set of tools to build Neural Networks. Including several additional packages for working with text[5] or images[6].

Doing your Deep Learning computations on the GPU speeds up your experiment by a lot! And PyTorch makes it ridiculously easy to do it. Let's start by checking if GPU is available:

```
In [0]: device = torch.device("cuda") if torch.cuda.is_available() else torch.device\
("cpu")
        device

Out[0]: device(type='cuda')
```

Good, we have a CUDA[7]-enabled GPU device on our hands. Let's store a Tensor on it:

```
In [0]: x = torch.tensor([[2, 3], [1, 2]])
        x.to(device)

Out[0]: tensor([[2, 3],
                [1, 2]], device='cuda:0')
```

Notice that our Tensor is now on device `cuda:0`. What can we do with it? Pretty much everything as before:

[4] https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html
[5] https://github.com/pytorch/text
[6] https://github.com/pytorch/vision
[7] https://en.wikipedia.org/wiki/CUDA

```
In [0]: x = x.to(device)

        y = torch.tensor([[2, 2], [5, 1]])
        y = y.to(device)

        x.add(y)

Out[0]: tensor([[4, 5],
                [6, 3]], device='cuda:0')
```

Common Issues

I got to be honest with you. You will fuck up, multiple times, before understanding how this whole thing works out. That's alright!

However, there are a couple of things you can do that might minimize the frustrations along your journey:

- Doing operations between GPU and CPU Tensors is **not allowed**
- Size mismatch between Tensors occurs often and is (almost every time) easy to fix:

  In [0]: a = torch.ones(2, 2) b = torch.ones(1, 3) a * b

  Out[0]: ————————————————————————————— RuntimeError Traceback (most recent call last) <ipython-input-21-b2a4e8765762> in <module>() 1 a = torch.ones(2, 2) 2 b = torch.ones(1, 3) —-> 3 a * b

```
        RuntimeError: The size of tensor a (2) must match the size of tensor b (3) at \
non-singleton dimension 1
```

PyTorch is very descriptive in this case. When doing more complex stuff, you would want to check the shape of your Tensors obsessively, after every operation. Just print the size!

- Running out of GPU memory: You might be leaking memory or too large of a dataset/model. Faster/better GPU always helps. But remember, you can solve really large problems with a single powerful GPU these days. Think carefully if that is not enough for you - why that is?

Conclusion

Welcome to the dark side! You might've been working with Keras, TensorFlow, or another Deep Learning framework, until recently. Almost every framework is great, but PyTorch has really solid roots. Easy to use and understand, allows for fast experimentation and standard debugging tools apply! Enjoy!

References

- "PyTorch: A Modern Library for Machine Learning" with Adam Paszke[8]
- Recitation 1 | Your First Deep Learning Code[9]
- What is PyTorch?[10]

[8] https://www.youtube.com/watch?v=5bSAipCNqXo
[9] https://www.youtube.com/watch?v=KrCp_yPVOxs
[10] https://pytorch.org/tutorials/beginner/blitz/tensor_tutorial.html

2. Build Your First Neural Network with PyTorch

In this tutorial, you'll build your first Neural Network using PyTorch. You'll use it to predict whether or not is going to rain tomorrow using real weather information.

- Run the complete notebook in your browser (Google Colab)[1]
- Read the Getting Things Done with Pytorch book[2]

You'll learn how to:

- Preprocess CSV files and convert the data to Tensors
- Build your own Neural Network model with PyTorch
- Use a loss function and an optimizer to train your model
- Evaluate your model and learn about the perils of imbalanced classification

```
%reload_ext watermark
%watermark -v -p numpy,pandas,torch
```

```
CPython 3.6.9
IPython 5.5.0

numpy 1.17.5
pandas 0.25.3
torch 1.4.0
```

[1] https://colab.research.google.com/drive/1lDXVkdt7GC8jK_nGmOMKeDywXse-DY-u
[2] https://github.com/curiousily/Getting-Things-Done-with-Pytorch

```
import torch

import os
import numpy as np
import pandas as pd
from tqdm import tqdm
import seaborn as sns
from pylab import rcParams
import matplotlib.pyplot as plt
from matplotlib import rc
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, classification_report

from torch import nn, optim

import torch.nn.functional as F

%matplotlib inline
%config InlineBackend.figure_format='retina'

sns.set(style='whitegrid', palette='muted', font_scale=1.2)

HAPPY_COLORS_PALETTE = ["#01BEFE", "#FFDD00", "#FF7D00", "#FF006D", "#93D30C", "#8F0\
0FF"]

sns.set_palette(sns.color_palette(HAPPY_COLORS_PALETTE))

rcParams['figure.figsize'] = 12, 8

RANDOM_SEED = 42
np.random.seed(RANDOM_SEED)
torch.manual_seed(RANDOM_SEED)
```

Data

Our dataset contains daily weather information from multiple Australian weather stations. We're about to answer a simple question. *Will it rain tomorrow?*

The data is hosted on Kaggle[3] and created by Joe Young[4]. I've uploaded the dataset to Google Drive. Let's get it:

[3] https://www.kaggle.com/jsphyg/weather-dataset-rattle-package
[4] https://www.kaggle.com/jsphyg

```
!gdown --id 1Q1wUptbNDYdfizk5abhmoFxIQiX19Tn7
```

And load it into a data frame:

```
df = pd.read_csv('weatherAUS.csv')
```

We have a large set of features/columns here. You might also notice some *NaN*s. Let's have a look at the overall dataset size:

```
df.shape
```

```
(142193, 24)
```

Looks like we have plenty of data. But we got to do something about those missing values.

Data Preprocessing

We'll start by simplifying the problem by removing most of the data. We'll use only 4 columns for predicting whether or not is going to rain tomorrow:

```
cols = ['Rainfall', 'Humidity3pm', 'Pressure9am', 'RainToday', 'RainTomorrow']

df = df[cols]
```

Neural Networks don't work with much else than numbers. We'll convert *yes* and *no* to 1 and 0, respectively:

```
df['RainToday'].replace({'No': 0, 'Yes': 1}, inplace = True)
df['RainTomorrow'].replace({'No': 0, 'Yes': 1}, inplace = True)
```

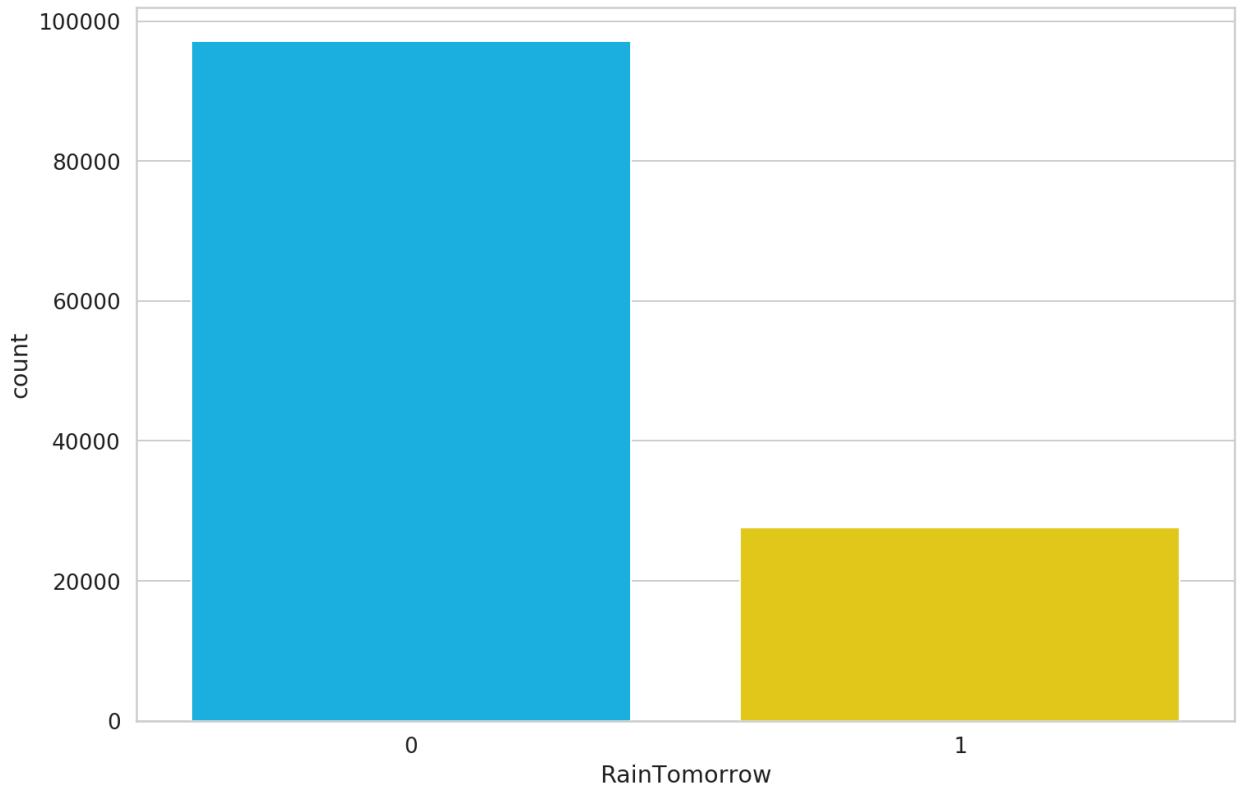
Let's drop the rows with missing values. There are better ways to do this, but we'll keep it simple:

```
df = df.dropna(how='any')
```

Finally, we have a dataset we can work with.

One important question we should answer is - *How balanced is our dataset?*. Or *How many times did it rain or not rain tomorrow?*:

```
sns.countplot(df.RainTomorrow);
```



png

```
df.RainTomorrow.value_counts() / df.shape[0]
```

```
0    0.778762
1    0.221238
Name: RainTomorrow, dtype: float64
```

Things are not looking good. About 78% of the data points have a non-rainy day for tomorrow. This means that a model that predicts there will be no rain tomorrow will be correct about 78% of the time.

You can read and apply the Practical Guide to Handling Imbalanced Datasets[5] if you want to mitigate this issue. Here, we'll just hope for the best.

The final step is to split the data into train and test sets:

[5] https://www.curiousily.com/posts/practical-guide-to-handling-imbalanced-datasets/

```
X = df[['Rainfall', 'Humidity3pm', 'RainToday', 'Pressure9am']]
y = df[['RainTomorrow']]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_stat\
e=RANDOM_SEED)
```

And convert all of it to Tensors (so we can use it with PyTorch):

```
X_train = torch.from_numpy(X_train.to_numpy()).float()
y_train = torch.squeeze(torch.from_numpy(y_train.to_numpy()).float())

X_test = torch.from_numpy(X_test.to_numpy()).float()
y_test = torch.squeeze(torch.from_numpy(y_test.to_numpy()).float())

print(X_train.shape, y_train.shape)
print(X_test.shape, y_test.shape)
```

```
torch.Size([99751, 4]) torch.Size([99751])
torch.Size([24938, 4]) torch.Size([24938])
```

Building a Neural Network

We'll build a simple Neural Network (NN) that tries to predicts will it rain tomorrow.

Our input contains data from the four columns: `Rainfall, Humidity3pm, RainToday, Pressure9am`. We'll create an appropriate input layer for that.

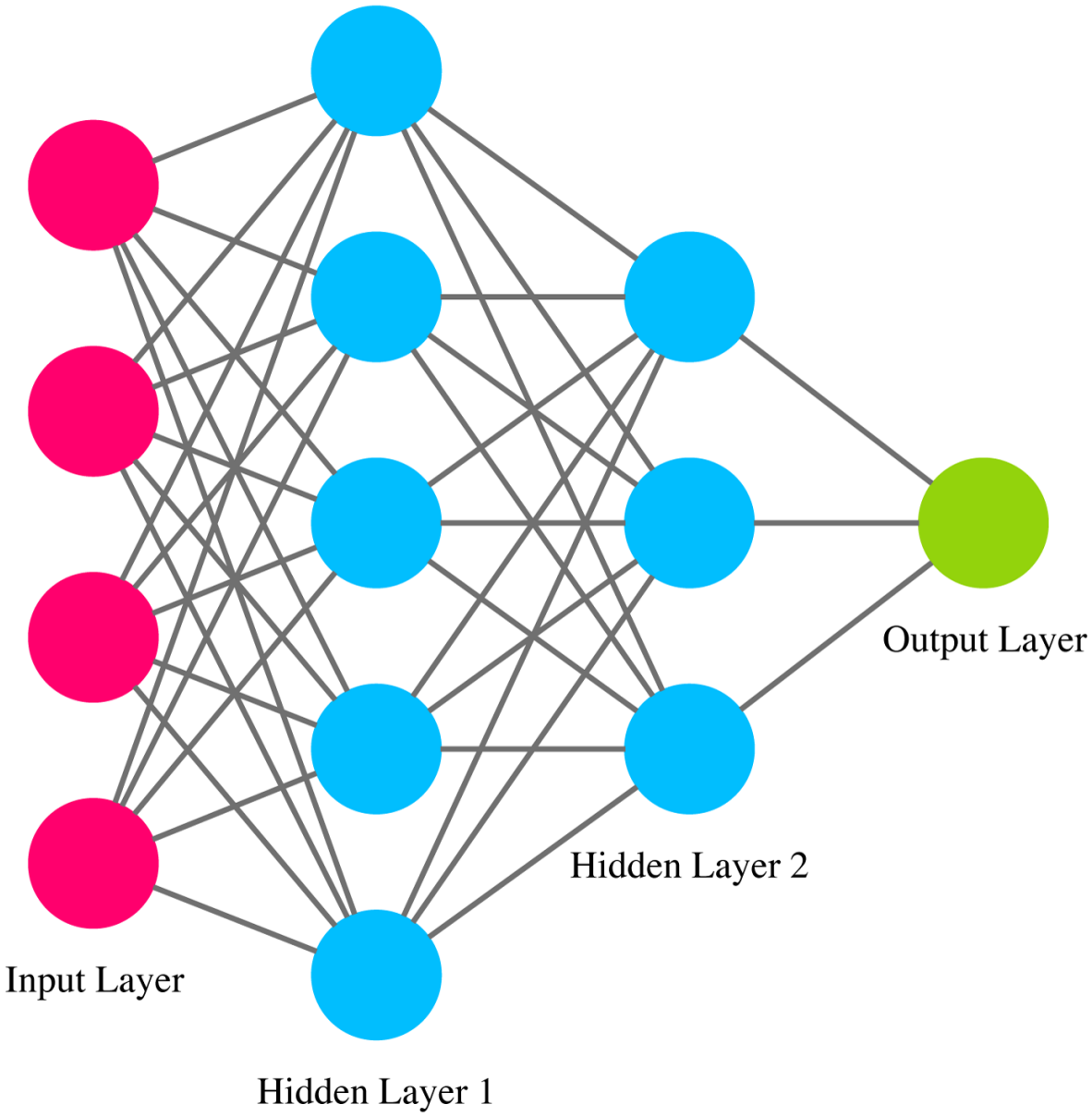
The output will be a number between 0 and 1, representing how likely (our model thinks) it is going to rain tomorrow. The prediction will be given to us by the final (output) layer of the network.

We'll two (hidden) layers between the input and output layers. The parameters (neurons) of those layer will decide the final output. All layers will be fully-connected.

One easy way to build the NN with PyTorch is to create a class that inherits from `torch.nn.Module`:

```
class Net(nn.Module):

  def __init__(self, n_features):
    super(Net, self).__init__()
    self.fc1 = nn.Linear(n_features, 5)
    self.fc2 = nn.Linear(5, 3)
    self.fc3 = nn.Linear(3, 1)

  def forward(self, x):
    x = F.relu(self.fc1(x))
    x = F.relu(self.fc2(x))
    return torch.sigmoid(self.fc3(x))
```

```
net = Net(X_train.shape[1])

ann_viz(net, view=False)
```



svg

We start by creating the layers of our model in the constructor. The `forward()` method is where the magic happens. It accepts the input `x` and allows it to flow through each layer.

There is a corresponding backward pass (defined for you by PyTorch) that allows the model to learn from the errors that is currently making.

Activation Functions

You might notice the calls to `F.relu` and `torch.sigmoid`. Why do we need those?

One of the cool *features* of Neural Networks is that they can approximate non-linear functions. In fact, it is proven that they can approximate any function[6].

Good luck approximating non-linear functions by stacking linear layers. Activation functions allow you to break from the linear world and learn (hopefully) more. You'll usually find them applied to an output of some layer.

Those functions must be hard to define, right?

ReLU

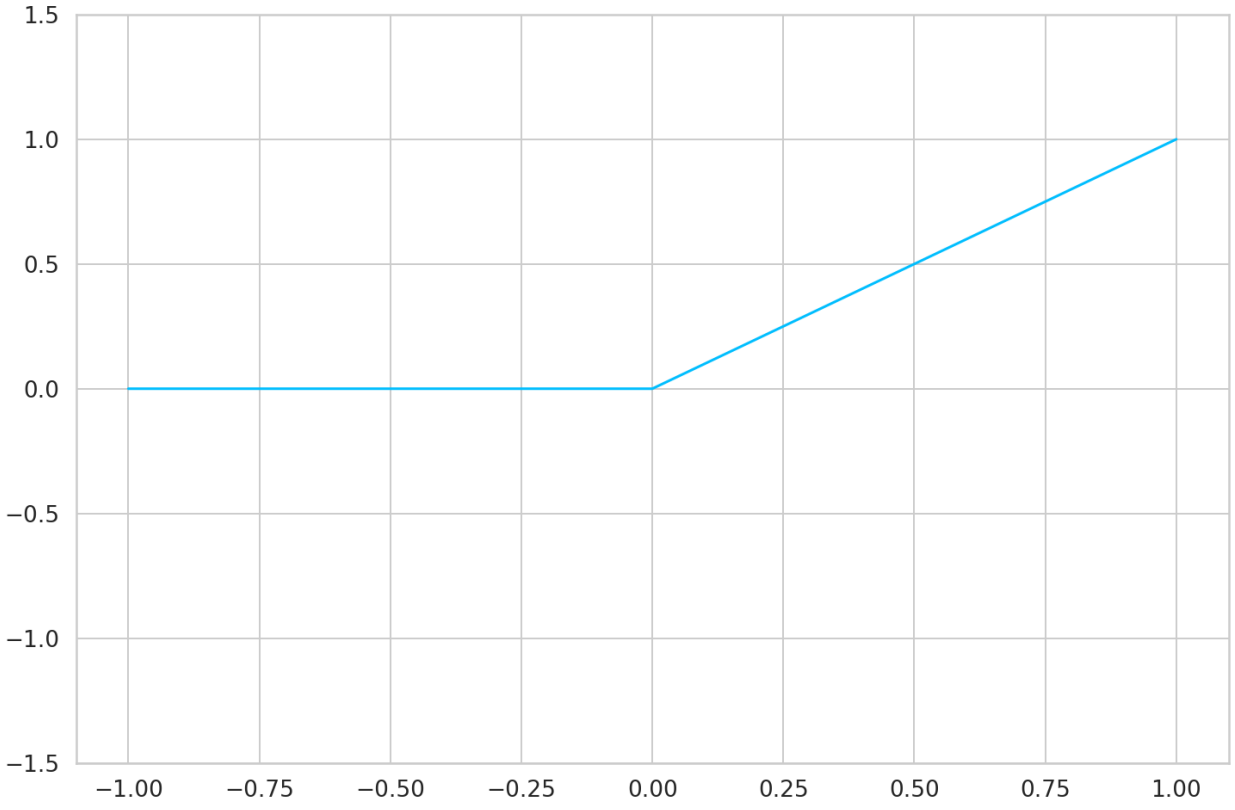
Not at all, let start with the ReLU definition (one of the most widely used activation function):

$$\text{ReLU}(x) = \max(0, x)$$

Easy peasy, the result is the maximum value of zero and the input.

```
ax = plt.gca()

plt.plot(
  np.linspace(-1, 1, 5),
  F.relu(torch.linspace(-1, 1, steps=5)).numpy()
)
ax.set_ylim([-1.5, 1.5]);
```

[6] https://en.wikipedia.org/wiki/Universal_approximation_theorem



png

Sigmoid

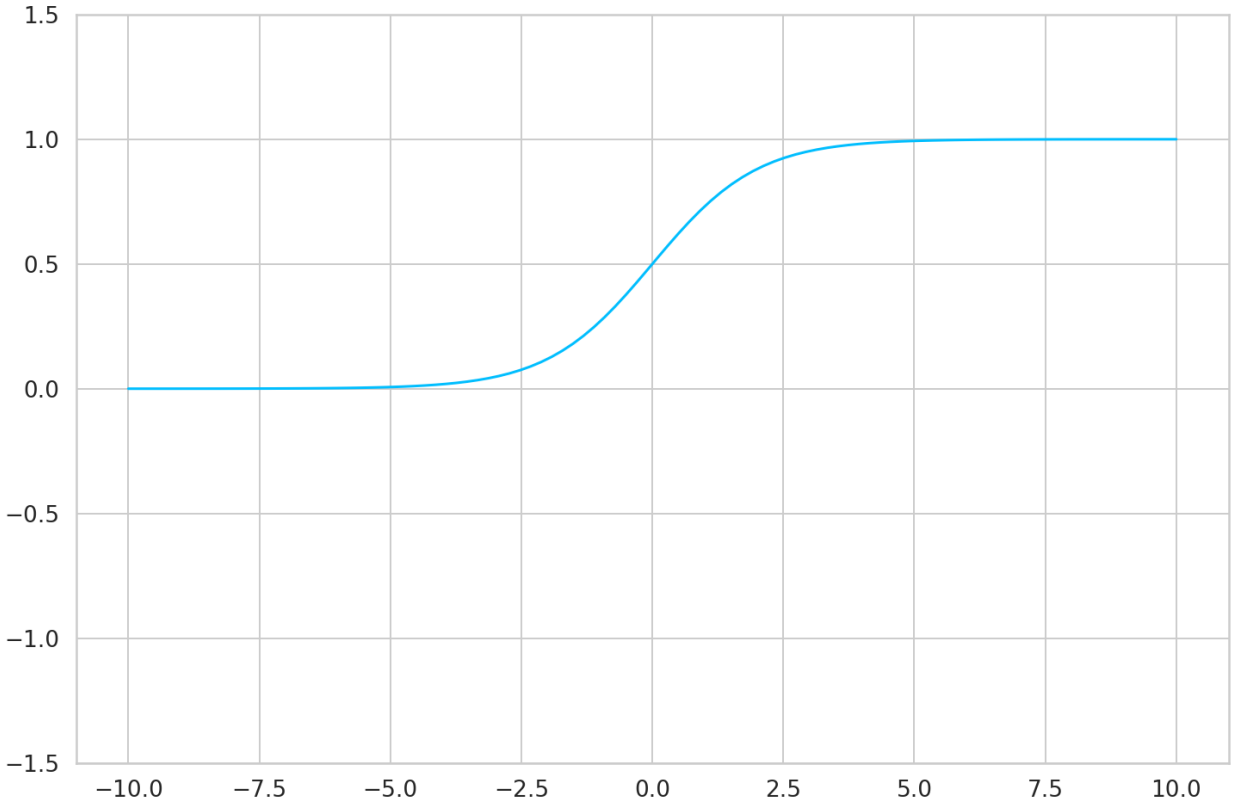
The sigmoid is useful when you need to make a binary decision/classification (answering with a *yes* or a *no*).

It is defined as:

$$\text{Sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

The sigmoid squishes the input values between 0 and 1. But in a super kind of way:

```
ax = plt.gca()

plt.plot(
  np.linspace(-10, 10, 100),
  torch.sigmoid(torch.linspace(-10, 10, steps=100)).numpy()
)
ax.set_ylim([-1.5, 1.5]);
```



png

Training

With the model in place, we need to find parameters that predict will it rain tomorrow. First, we need something to tell us how good we're currently doing:

```
1 criterion = nn.BCELoss()
```

The BCELoss[7] is a loss function that measures the difference between the two binary vectors. In our case, the predictions of our model and the real values. It expects the values to be outputed by the sigmoid function. The closer this value gets to 0, the better your model should be.

But how do we find parameters that minimize the loss function?

Optimization

Imagine that each parameter of our NN is a knob. The optimizer's job is to find the perfect positions for each knob so that the loss gets close to 0.

[7] https://pytorch.org/docs/stable/nn.html#bceloss

Real-world models can contain millions or even billions of parameters. With so many knobs to turn, it would be nice to have an efficient optimizer that quickly finds solutions.

Contrary to what you might believe, optimization in Deep Learning is just satisfying. In practice, you're content with good enough parameter values.

While there are tons of optimizers you can choose from, Adam[8] is a safe first choice. PyTorch has a well-debugged implementation you can use:

```
optimizer = optim.Adam(net.parameters(), lr=0.001)
```

Naturally, the optimizer requires the parameters. The second argument `lr` is *learning rate*. It is a tradeoff between how good parameters you're going to find and how fast you'll get there. Finding good values for this can be black magic and a lot of brute-force "experimentation".

Doing it on the GPU

Doing massively parallel computations on GPUs is one of the enablers for modern Deep Learning. You'll need nVIDIA GPU for that.

PyTorch makes it really easy to transfer all the computation to your GPU:

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

```
X_train = X_train.to(device)
y_train = y_train.to(device)

X_test = X_test.to(device)
y_test = y_test.to(device)
```

```
net = net.to(device)

criterion = criterion.to(device)
```

We start by checking whether or not a CUDA device is available. Then, we transfer all training and test data to that device. Finally, we move our model and loss function.

Finding Good Parameters

Having a loss function is great, but tracking the accuracy of our model is something easier to understand, for us mere mortals. Here's the definition for our accuracy:

[8] https://pytorch.org/docs/stable/optim.html#torch.optim.Adam

```
def calculate_accuracy(y_true, y_pred):
  predicted = y_pred.ge(.5).view(-1)
  return (y_true == predicted).sum().float() / len(y_true)
```

We convert every value below 0.5 to 0. Otherwise, we set it to 1. Finally, we calculate the percentage of correct values.

With all the pieces of the puzzle in place, we can start training our model:

```
def round_tensor(t, decimal_places=3):
  return round(t.item(), decimal_places)

for epoch in range(1000):

    y_pred = net(X_train)

    y_pred = torch.squeeze(y_pred)
    train_loss = criterion(y_pred, y_train)

    if epoch % 100 == 0:
      train_acc = calculate_accuracy(y_train, y_pred)

      y_test_pred = net(X_test)
      y_test_pred = torch.squeeze(y_test_pred)

      test_loss = criterion(y_test_pred, y_test)

      test_acc = calculate_accuracy(y_test, y_test_pred)
      print(
f'''epoch {epoch}
Train set - loss: {round_tensor(train_loss)}, accuracy: {round_tensor(train_acc)}
Test  set - loss: {round_tensor(test_loss)}, accuracy: {round_tensor(test_acc)}
''')

    optimizer.zero_grad()

    train_loss.backward()

    optimizer.step()
```

```
epoch 0
Train set - loss: 2.513, accuracy: 0.779
Test  set - loss: 2.517, accuracy: 0.778

epoch 100
Train set - loss: 0.457, accuracy: 0.792
Test  set - loss: 0.458, accuracy: 0.793

epoch 200
Train set - loss: 0.435, accuracy: 0.801
Test  set - loss: 0.436, accuracy: 0.8

epoch 300
Train set - loss: 0.421, accuracy: 0.814
Test  set - loss: 0.421, accuracy: 0.815

epoch 400
Train set - loss: 0.412, accuracy: 0.826
Test  set - loss: 0.413, accuracy: 0.827

epoch 500
Train set - loss: 0.408, accuracy: 0.831
Test  set - loss: 0.408, accuracy: 0.832

epoch 600
Train set - loss: 0.406, accuracy: 0.833
Test  set - loss: 0.406, accuracy: 0.835

epoch 700
Train set - loss: 0.405, accuracy: 0.834
Test  set - loss: 0.405, accuracy: 0.835

epoch 800
Train set - loss: 0.404, accuracy: 0.834
Test  set - loss: 0.404, accuracy: 0.835

epoch 900
Train set - loss: 0.404, accuracy: 0.834
Test  set - loss: 0.404, accuracy: 0.836
```

During the training, we show our model the data for `10,000` times. Each time we measure the loss, propagate the errors trough our model and asking the optimizer to find better parameters.

The `zero_grad()` method clears up the accumulated gradients, which the optimizer uses to find

better parameters.

What about that accuracy? 83.6% accuracy on the test set sounds reasonable, right? Well, I am about to disappoint you. But first, let's learn how to save and load our trained models.

Saving the model

Training a good model can take a lot of time. And I mean weeks, months or even years. So, let's make sure that you know how you can save your precious work. Saving is easy:

```
MODEL_PATH = 'model.pth'

torch.save(net, MODEL_PATH)
```

Restoring your model is easy too:

```
net = torch.load(MODEL_PATH)
```

Evaluation

Wouldn't it be perfect to know what kinds of errors your model makes? Of course, that's impossible. But you can get an estimate.

Using just accuracy wouldn't be a good way to do it. Recall that our data contains mostly no rain examples.

One way to delve a bit deeper into your model performance is to assess the precision and recall for each class. In our case, that will be *no rain* and *rain*:

```
classes = ['No rain', 'Raining']

y_pred = net(X_test)

y_pred = y_pred.ge(.5).view(-1).cpu()
y_test = y_test.cpu()

print(classification_report(y_test, y_pred, target_names=classes))
```

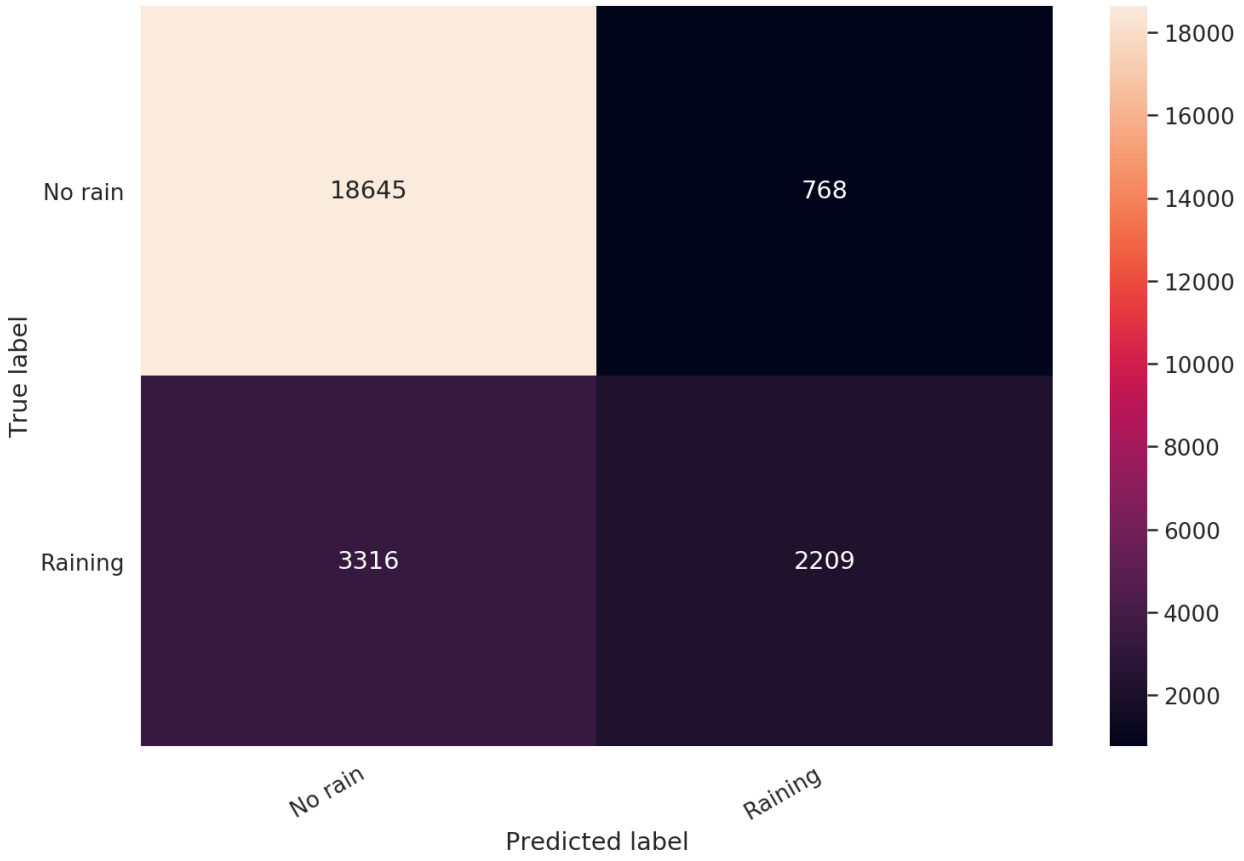
```
              precision    recall  f1-score   support

     No rain       0.85      0.96      0.90     19413
     Raining       0.74      0.40      0.52      5525

    accuracy                           0.84     24938
   macro avg       0.80      0.68      0.71     24938
weighted avg       0.83      0.84      0.82     24938
```

A maximum precision of 1 indicates that the model is perfect at identifying only relevant examples. A maximum recall of 1 indicates that our model can find all relevant examples in the dataset for this class.

You can see that our model is doing good when it comes to the *No rain* class. We have so many examples. Unfortunately, we can't really trust predictions of the *Raining* class.

One of the best things about binary classification is that you can have a good look at a simple confusion matrix:

```
cm = confusion_matrix(y_test, y_pred)
df_cm = pd.DataFrame(cm, index=classes, columns=classes)

hmap = sns.heatmap(df_cm, annot=True, fmt="d")
hmap.yaxis.set_ticklabels(hmap.yaxis.get_ticklabels(), rotation=0, ha='right')
hmap.xaxis.set_ticklabels(hmap.xaxis.get_ticklabels(), rotation=30, ha='right')
plt.ylabel('True label')
plt.xlabel('Predicted label');
```



png

You can clearly see that our model shouldn't be trusted when it says it's going to rain.

Conclusion

Well done! You now have a Neural Network that can predict the weather. Well, sort of. Building well-performing models is hard, really hard. But there are tricks you'll pick up along the way and (hopefully) get better at your craft!

- Run the complete notebook in your browser (Google Colab)[9]
- Read the Getting Things Done with Pytorch book[10]

You learned how to:

- Preprocess CSV files and convert the data to Tensors

[9] https://colab.research.google.com/drive/1lDXVkdt7GC8jK_nGmOMKeDywXse-DY-u
[10] https://github.com/curiousily/Getting-Things-Done-with-Pytorch

- Build your own Neural Network model with PyTorch
- Use a loss function and an optimizer to train your model
- Evaluate your model and learn about the perils of imbalanced classification

References

- Precision and Recall[11]
- Beyond Accuracy: Precision and Recall[12]

[11] https://en.wikipedia.org/wiki/Precision_and_recall
[12] https://towardsdatascience.com/beyond-accuracy-precision-and-recall-3da06bea9f6c

3. Transfer Learning for Image Classification using Torchvision

> TL;DR Learn how to use Transfer Learning to classify traffic sign images. You'll build a dataset of images in a format suitable for working with Torchvision. Get predictions on images from the wild (downloaded from the Internet).

In this tutorial, you'll learn how to fine-tune a pre-trained model for classifying raw pixels of traffic signs.

- Read the tutorial[1]
- Run the notebook in your browser (Google Colab)[2]
- Read the `Getting Things Done with Pytorch` book[3]

Here's what we'll go over:

- Overview of the traffic sign image dataset
- Build a dataset
- Use a pre-trained model from Torchvision
- Add a new *unknown* class and re-train the model

Will this model be ready for the real world?

```
import torch, torchvision

from pathlib import Path
import numpy as np
import cv2
import pandas as pd
from tqdm import tqdm
import PIL.Image as Image
import seaborn as sns
from pylab import rcParams
import matplotlib.pyplot as plt
from matplotlib import rc
```

[1] https://www.curiousily.com/posts/transfer-learning-for-image-classification-using-torchvision-pytorch-and-python/
[2] https://colab.research.google.com/drive/1Lk5R4pECDxDhd1uXcv26YRQ02fb_mrL9?usp=sharing
[3] https://github.com/curiousily/Getting-Things-Done-with-Pytorch

```
from matplotlib.ticker import MaxNLocator
from torch.optim import lr_scheduler
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, classification_report
from glob import glob
import shutil
from collections import defaultdict

from torch import nn, optim

import torch.nn.functional as F
import torchvision.transforms as T
from torchvision.datasets import ImageFolder
from torch.utils.data import DataLoader
from torchvision import models

%matplotlib inline
%config InlineBackend.figure_format='retina'

sns.set(style='whitegrid', palette='muted', font_scale=1.2)

HAPPY_COLORS_PALETTE = ["#01BEFE", "#FFDD00", "#FF7D00", "#FF006D", "#ADFF02", "#8F0\
0FF"]

sns.set_palette(sns.color_palette(HAPPY_COLORS_PALETTE))

rcParams['figure.figsize'] = 12, 8

RANDOM_SEED = 42
np.random.seed(RANDOM_SEED)
torch.manual_seed(RANDOM_SEED)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

Recognizing traffic signs

German Traffic Sign Recognition Benchmark (GTSRB)[4] contains more than 50,000 annotated images of 40+ traffic signs. Given an image, you'll have to recognize the traffic sign on it.

[4] http://benchmark.ini.rub.de/?section=gtsrb&subsection=dataset

```
!wget https://sid.erda.dk/public/archives/daaeac0d7ce1152aea9b61d9f1e19370/GTSRB_Fin\
al_Training_Images.zip
!unzip -qq GTSRB_Final_Training_Images.zip
```

Exploration

Let's start by getting a feel of the data. The images for each traffic sign are stored in a separate directory. How many do we have?

```
train_folders = sorted(glob('GTSRB/Final_Training/Images/*'))
len(train_folders)
```

```
    43
```

We'll create 3 helper functions that use OpenCV and Torchvision to load and show images:

```
def load_image(img_path, resize=True):
  img = cv2.cvtColor(cv2.imread(img_path), cv2.COLOR_BGR2RGB)

  if resize:
    img = cv2.resize(img, (64, 64), interpolation = cv2.INTER_AREA)

  return img

def show_image(img_path):
  img = load_image(img_path)
  plt.imshow(img)
  plt.axis('off')

def show_sign_grid(image_paths):
  images = [load_image(img) for img in image_paths]
  images = torch.as_tensor(images)
  images = images.permute(0, 3, 1, 2)
  grid_img = torchvision.utils.make_grid(images, nrow=11)
  plt.figure(figsize=(24, 12))
  plt.imshow(grid_img.permute(1, 2, 0))
  plt.axis('off');
```

Let's have a look at some examples for each traffic sign:

```
sample_images = [np.random.choice(glob(f'{tf}/*ppm')) for tf in train_folders]
show_sign_grid(sample_images)
```

png

And here is a single sign:

```
img_path = glob(f'{train_folders[16]}/*ppm')[1]

show_image(img_path)
```

png

Building a dataset

To keep things simple, we'll focus on classifying some of the most used traffic signs:

```
class_names = ['priority_road', 'give_way', 'stop', 'no_entry']

class_indices = [12, 13, 14, 17]
```

We'll copy the images files to a new directory, so it's easier to use the Torchvision's dataset helpers. Let's start with the directories for each class:

```
!rm -rf data

DATA_DIR = Path('data')

DATASETS = ['train', 'val', 'test']

for ds in DATASETS:
  for cls in class_names:
    (DATA_DIR / ds / cls).mkdir(parents=True, exist_ok=True)
```

We'll reserve 80% of the images for training, 10% for validation, and 10% test for each class. We'll copy each image to the correct dataset directory:

```
for i, cls_index in enumerate(class_indices):
  image_paths = np.array(glob(f'{train_folders[cls_index]}/*.ppm'))
  class_name = class_names[i]
  print(f'{class_name}: {len(image_paths)}')
  np.random.shuffle(image_paths)

  ds_split = np.split(
    image_paths,
    indices_or_sections=[int(.8*len(image_paths)), int(.9*len(image_paths))]
  )

  dataset_data = zip(DATASETS, ds_split)

  for ds, images in dataset_data:
    for img_path in images:
      shutil.copy(img_path, f'{DATA_DIR}/{ds}/{class_name}/')
```

```
    priority_road: 2100
    give_way: 2160
    stop: 780
    no_entry: 1110
```

We have some class imbalance, but it is not that bad. We'll ignore it.

We'll apply some image augmentation techniques to artificially increase the size of our training dataset:

```
mean_nums = [0.485, 0.456, 0.406]
std_nums = [0.229, 0.224, 0.225]

transforms = {'train': T.Compose([
  T.RandomResizedCrop(size=256),
  T.RandomRotation(degrees=15),
  T.RandomHorizontalFlip(),
  T.ToTensor(),
  T.Normalize(mean_nums, std_nums)
]), 'val': T.Compose([
  T.Resize(size=256),
  T.CenterCrop(size=224),
  T.ToTensor(),
  T.Normalize(mean_nums, std_nums)
]), 'test': T.Compose([
  T.Resize(size=256),
  T.CenterCrop(size=224),
  T.ToTensor(),
  T.Normalize(mean_nums, std_nums)
]),
}
```

We apply some random resizing, rotation, and horizontal flips. Finally, we normalize the tensors using preset values for each channel. This is a requirement of the pre-trained models[5] in Torchvision.

We'll create a PyTorch dataset for each image dataset folder and data loaders for easier training:

[5] https://pytorch.org/docs/stable/torchvision/models.html

```
image_datasets = {
  d: ImageFolder(f'{DATA_DIR}/{d}', transforms[d]) for d in DATASETS
}

data_loaders = {
  d: DataLoader(image_datasets[d], batch_size=4, shuffle=True, num_workers=4)
  for d in DATASETS
}
```

We'll also store the number of examples in each dataset and class names for later:

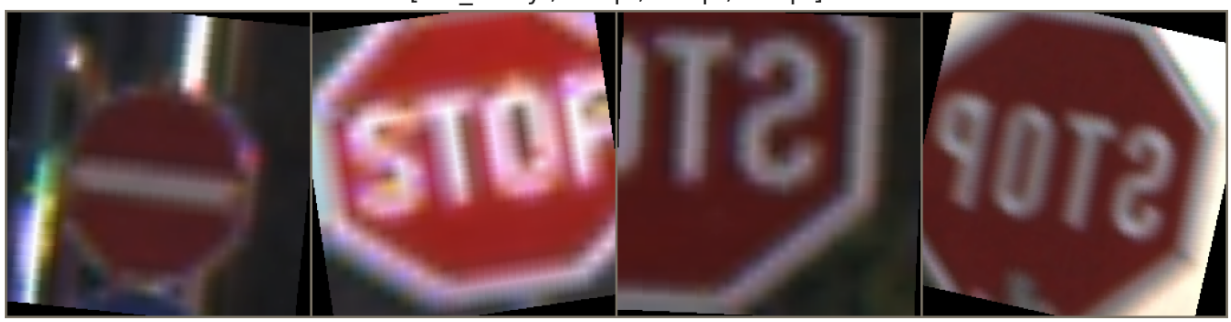
```
dataset_sizes = {d: len(image_datasets[d]) for d in DATASETS}
class_names = image_datasets['train'].classes

dataset_sizes
```

```
    {'test': 615, 'train': 4920, 'val': 615}
```

Let's have a look at some example images with applied transformations. We also need to reverse the normalization and reorder the color channels to get correct image data:

```
def imshow(inp, title=None):
  inp = inp.numpy().transpose((1, 2, 0))
  mean = np.array([mean_nums])
  std = np.array([std_nums])
  inp = std * inp + mean
  inp = np.clip(inp, 0, 1)
  plt.imshow(inp)
  if title is not None:
    plt.title(title)
  plt.axis('off')

inputs, classes = next(iter(data_loaders['train']))
out = torchvision.utils.make_grid(inputs)

imshow(out, title=[class_names[x] for x in classes])
```

['no_entry', 'stop', 'stop', 'stop']



png

Using a pre-trained model:

Our model will receive raw image pixels and try to classify them into one of four traffic signs. How hard can it be? Try to build a model from scratch.

Here, we'll use Transfer Learning[6] to copy the architecture of the very popular ResNet[7] model. On top of that, we'll use the learned weights of the model from training on the ImageNet dataset [8]. All of this is made easy to use by Torchvision:

```
def create_model(n_classes):
  model = models.resnet34(pretrained=True)

  n_features = model.fc.in_features
  model.fc = nn.Linear(n_features, n_classes)

  return model.to(device)
```

We reuse almost everything except the change of the output layer. This is needed because the number of classes in our dataset is different than ImageNet.

Let's create an instance of our model:

```
base_model = create_model(len(class_names))
```

Training

We'll write 3 helper functions to encapsulate the training and evaluation logic. Let's start with `train_epoch`:

[6] https://en.wikipedia.org/wiki/Transfer_learning
[7] https://arxiv.org/abs/1512.03385
[8] http://www.image-net.org/

```python
def train_epoch(
  model,
  data_loader,
  loss_fn,
  optimizer,
  device,
  scheduler,
  n_examples
):
  model = model.train()

  losses = []
  correct_predictions = 0

  for inputs, labels in data_loader:
    inputs = inputs.to(device)
    labels = labels.to(device)

    outputs = model(inputs)

    _, preds = torch.max(outputs, dim=1)
    loss = loss_fn(outputs, labels)

    correct_predictions += torch.sum(preds == labels)
    losses.append(loss.item())

    loss.backward()
    optimizer.step()
    optimizer.zero_grad()

  scheduler.step()

  return correct_predictions.double() / n_examples, np.mean(losses)
```

We start by turning our model into train mode and go over the data. After getting the predictions, we get the class with maximum probability along with the loss, so we can calculate the epoch loss and accuracy.

Note that we're also using a learning rate scheduler (more on that later).

```
def eval_model(model, data_loader, loss_fn, device, n_examples):
  model = model.eval()

  losses = []
  correct_predictions = 0

  with torch.no_grad():
    for inputs, labels in data_loader:
      inputs = inputs.to(device)
      labels = labels.to(device)

      outputs = model(inputs)

      _, preds = torch.max(outputs, dim=1)

      loss = loss_fn(outputs, labels)

      correct_predictions += torch.sum(preds == labels)
      losses.append(loss.item())

  return correct_predictions.double() / n_examples, np.mean(losses)
```

The evaluation of the model is pretty similar, except that we don't do any gradient calculations. Let's put everything together:

```
def train_model(model, data_loaders, dataset_sizes, device, n_epochs=3):
  optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
  scheduler = lr_scheduler.StepLR(optimizer, step_size=7, gamma=0.1)
  loss_fn = nn.CrossEntropyLoss().to(device)

  history = defaultdict(list)
  best_accuracy = 0

  for epoch in range(n_epochs):

    print(f'Epoch {epoch + 1}/{n_epochs}')
    print('-' * 10)

    train_acc, train_loss = train_epoch(
      model,
      data_loaders['train'],
      loss_fn,
```

```
      optimizer,
      device,
      scheduler,
      dataset_sizes['train']
    )

    print(f'Train loss {train_loss} accuracy {train_acc}')

    val_acc, val_loss = eval_model(
      model,
      data_loaders['val'],
      loss_fn,
      device,
      dataset_sizes['val']
    )

    print(f'Val   loss {val_loss} accuracy {val_acc}')
    print()

    history['train_acc'].append(train_acc)
    history['train_loss'].append(train_loss)
    history['val_acc'].append(val_acc)
    history['val_loss'].append(val_loss)

    if val_acc > best_accuracy:
      torch.save(model.state_dict(), 'best_model_state.bin')
      best_accuracy = val_acc

  print(f'Best val accuracy: {best_accuracy}')

  model.load_state_dict(torch.load('best_model_state.bin'))

  return model, history
```

We do a lot of string formatting and recording of the training history. The hard stuff gets delegated to the previous helper functions. We also want the best model, so the weights of the most accurate model(s) get stored during the training.

Let's train our first model:

```
%%time

base_model, history = train_model(base_model, data_loaders, dataset_sizes, device)
```

```
    Epoch 1/3
    ----------
    Train loss 0.31827690804876935 accuracy 0.8859756097560976
    Val   loss 0.0012465072916699694 accuracy 1.0

    Epoch 2/3
    ----------
    Train loss 0.12230596961529275 accuracy 0.9615853658536585
    Val   loss 0.0007955377752130681 accuracy 1.0

    Epoch 3/3
    ----------
    Train loss 0.07771141678094864 accuracy 0.9745934959349594
    Val   loss 0.0025791768387877366 accuracy 0.9983739837398374

    Best val accuracy: 1.0
    CPU times: user 2min 24s, sys: 48.2 s, total: 3min 12s
    Wall time: 3min 21s
```

Here's a little helper function that visualizes the training history for us:

```
def plot_training_history(history):
  fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(18, 6))

  ax1.plot(history['train_loss'], label='train loss')
  ax1.plot(history['val_loss'], label='validation loss')

  ax1.xaxis.set_major_locator(MaxNLocator(integer=True))
  ax1.set_ylim([-0.05, 1.05])
  ax1.legend()
  ax1.set_ylabel('Loss')
  ax1.set_xlabel('Epoch')

  ax2.plot(history['train_acc'], label='train accuracy')
  ax2.plot(history['val_acc'], label='validation accuracy')

  ax2.xaxis.set_major_locator(MaxNLocator(integer=True))
  ax2.set_ylim([-0.05, 1.05])
```

```
  ax2.legend()

  ax2.set_ylabel('Accuracy')
  ax2.set_xlabel('Epoch')

  fig.suptitle('Training history')

plot_training_history(history)
```

png

The pre-trained model is so good that we get very high accuracy and low loss after 3 epochs. Unfortunately, our validation set is too small to get some meaningful metrics from it.

Evaluation

Let's see some predictions on traffic signs from the test set:

```
def show_predictions(model, class_names, n_images=6):
  model = model.eval()
  images_handeled = 0
  plt.figure()

  with torch.no_grad():
    for i, (inputs, labels) in enumerate(data_loaders['test']):
      inputs = inputs.to(device)
      labels = labels.to(device)

      outputs = model(inputs)
```

```
      _, preds = torch.max(outputs, 1)

      for j in range(inputs.shape[0]):
        images_handeled += 1
        ax = plt.subplot(2, n_images//2, images_handeled)
        ax.set_title(f'predicted: {class_names[preds[j]]}')
        imshow(inputs.cpu().data[j])
        ax.axis('off')

        if images_handeled == n_images:
          return
```

```
show_predictions(base_model, class_names, n_images=8)
```

png

Very good! Even the almost not visible *priority road* sign is classified correctly. Let's dive a bit deeper. We'll start by getting the predictions from our model:

```python
def get_predictions(model, data_loader):
  model = model.eval()
  predictions = []
  real_values = []
  with torch.no_grad():
    for inputs, labels in data_loader:
      inputs = inputs.to(device)
      labels = labels.to(device)

      outputs = model(inputs)
      _, preds = torch.max(outputs, 1)
      predictions.extend(preds)
      real_values.extend(labels)
  predictions = torch.as_tensor(predictions).cpu()
  real_values = torch.as_tensor(real_values).cpu()
  return predictions, real_values
```

```python
y_pred, y_test = get_predictions(base_model, data_loaders['test'])
```

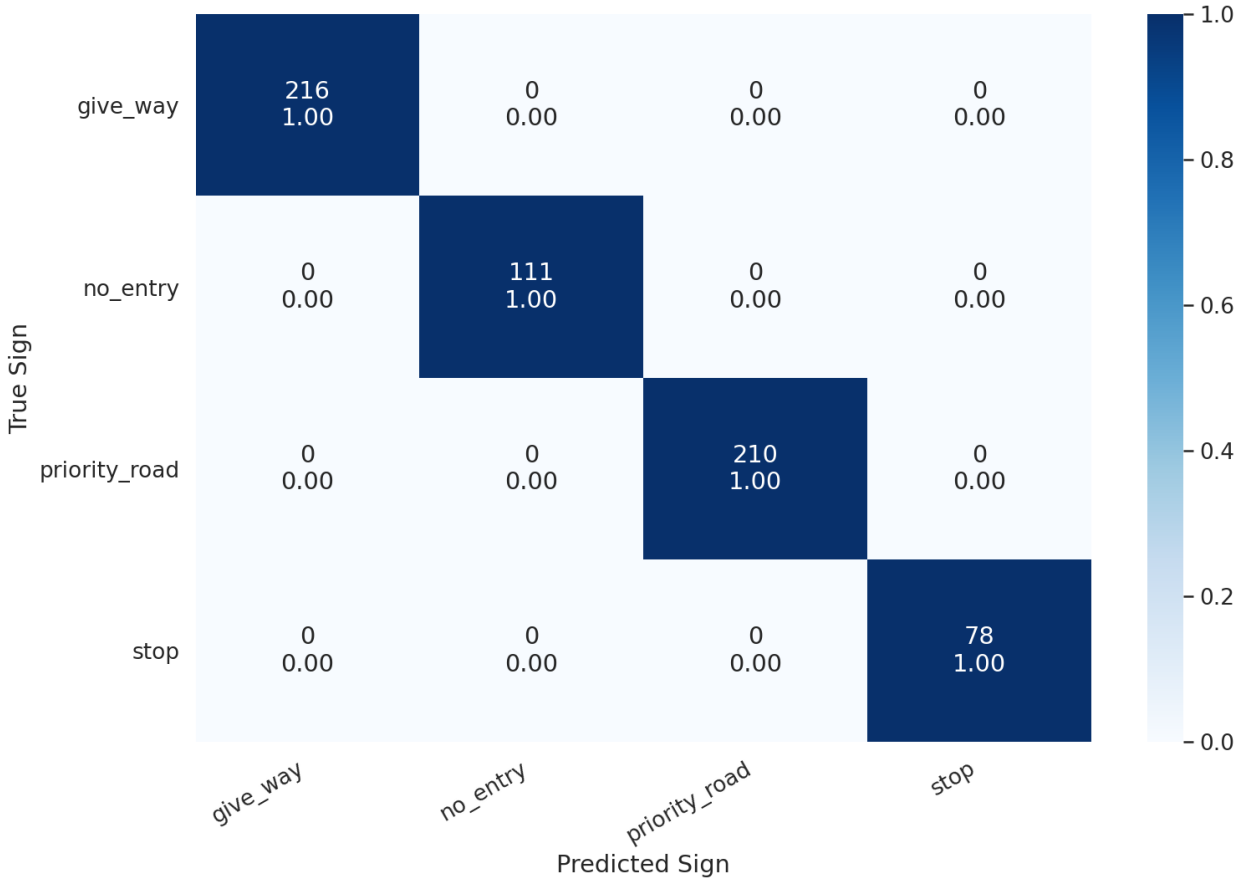
```python
print(classification_report(y_test, y_pred, target_names=class_names))
```

```
                  precision    recall  f1-score   support

       give_way       1.00      1.00      1.00       216
       no_entry       1.00      1.00      1.00       111
  priority_road       1.00      1.00      1.00       210
           stop       1.00      1.00      1.00        78

       accuracy                           1.00       615
      macro avg       1.00      1.00      1.00       615
   weighted avg       1.00      1.00      1.00       615
```

The classification report shows us that our model is perfect, not something you see every day! Does this thing make any mistakes?

```
def show_confusion_matrix(confusion_matrix, class_names):

  cm = confusion_matrix.copy()

  cell_counts = cm.flatten()

  cm_row_norm = cm / cm.sum(axis=1)[:, np.newaxis]

  row_percentages = ["{0:.2f}".format(value) for value in cm_row_norm.flatten()]

  cell_labels = [f"{cnt}\n{per}" for cnt, per in zip(cell_counts, row_percentages)]
  cell_labels = np.asarray(cell_labels).reshape(cm.shape[0], cm.shape[1])

  df_cm = pd.DataFrame(cm_row_norm, index=class_names, columns=class_names)

  hmap = sns.heatmap(df_cm, annot=cell_labels, fmt="", cmap="Blues")
  hmap.yaxis.set_ticklabels(hmap.yaxis.get_ticklabels(), rotation=0, ha='right')
  hmap.xaxis.set_ticklabels(hmap.xaxis.get_ticklabels(), rotation=30, ha='right')
  plt.ylabel('True Sign')
  plt.xlabel('Predicted Sign');
```

```
cm = confusion_matrix(y_test, y_pred)
show_confusion_matrix(cm, class_names)
```



png

No, no mistakes here!

Classifying unseen images

Ok, but how good our model will be when confronted with a real-world image? Let's check it out:

```
!gdown --id 19Qz3a610u_QSHsLeTznx8LtDBu4tbqHr
```

```
show_image('stop-sign.jpg')
```

png

For this, we'll have a look at the confidence for each class. Let's get this from our model:

```
def predict_proba(model, image_path):
  img = Image.open(image_path)
  img = img.convert('RGB')
  img = transforms['test'](img).unsqueeze(0)

  pred = model(img.to(device))
  pred = F.softmax(pred, dim=1)
  return pred.detach().cpu().numpy().flatten()
```

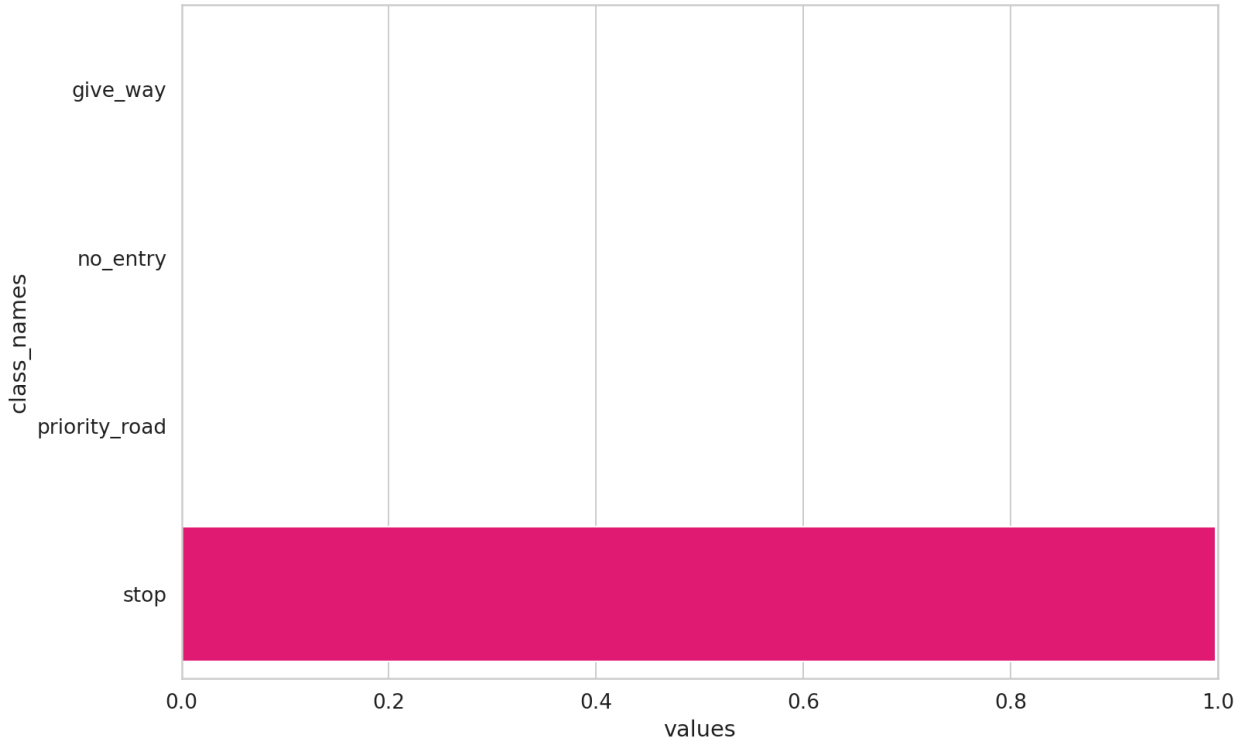
```
pred = predict_proba(base_model, 'stop-sign.jpg')
pred
```

```
    array([1.1296713e-03, 1.9811286e-04, 3.4486805e-04, 9.9832731e-01],
          dtype=float32)
```

This is a bit hard to understand. Let's plot it:

```
def show_prediction_confidence(prediction, class_names):
  pred_df = pd.DataFrame({
    'class_names': class_names,
    'values': prediction
  })
  sns.barplot(x='values', y='class_names', data=pred_df, orient='h')
  plt.xlim([0, 1]);
```

```
show_prediction_confidence(pred, class_names)
```



png

Again, our model is performing very well! Really confident in the correct traffic sign!

Classyfing unknown traffic sign

The last challenge for our model is a traffic sign that it hasn't seen before:

```
!gdown --id 1F61-iNhlJk-yKZRGcu6S9P29HxDFxF0u
```

```
show_image('unknown-sign.jpg')
```

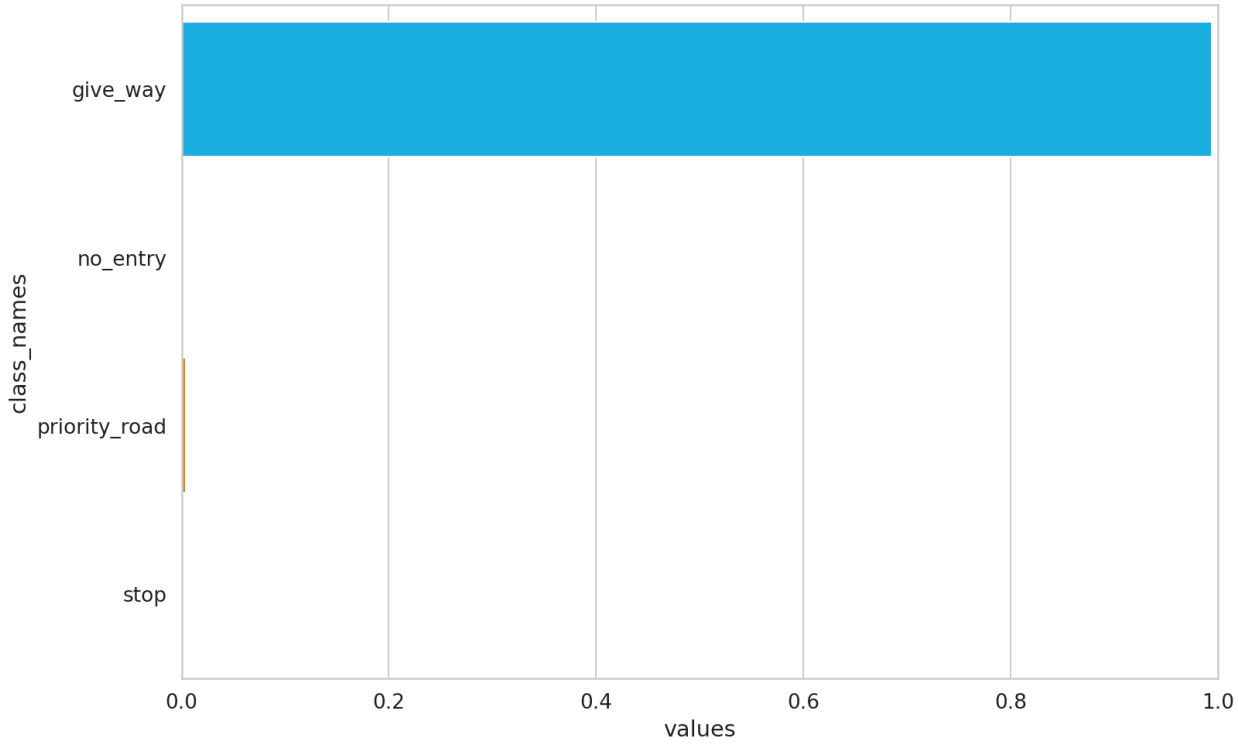
png

Let's get the predictions:

```
pred = predict_proba(base_model, 'unknown-sign.jpg')
pred
```

```
    array([9.9413127e-01, 1.1861280e-06, 3.9936006e-03, 1.8739274e-03],
          dtype=float32)
```

```
show_prediction_confidence(pred, class_names)
```



png

Our model is very certain (more than 95% confidence) that this is a *give way* sign. This is obviously wrong. How can you make your model see this?

Adding class “unknown”

While there are a variety of ways to handle this situation (one described in this paper: A Baseline for Detecting Misclassified and Out-of-Distribution Examples in Neural Networks[9]), we’ll do something simpler.

We’ll get the indices of all traffic signs that weren’t included in our original dataset:

[9] https://arxiv.org/pdf/1610.02136.pdf

```
unknown_indices = [
  i for i, f in enumerate(train_folders) \
  if i not in class_indices
]

len(unknown_indices)
```

```
    39
```

We'll create a new folder for the unknown class and copy some of the images there:

```
for ds in DATASETS:
  (DATA_DIR / ds / 'unknown').mkdir(parents=True, exist_ok=True)

for ui in unknown_indices:
  image_paths = np.array(glob(f'{train_folders[ui]}/*.ppm'))
  image_paths = np.random.choice(image_paths, 50)

  ds_split = np.split(
    image_paths,
    indices_or_sections=[int(.8*len(image_paths)), int(.9*len(image_paths))]
  )

  dataset_data = zip(DATASETS, ds_split)

  for ds, images in dataset_data:
    for img_path in images:
      shutil.copy(img_path, f'{DATA_DIR}/{ds}/unknown/')
```

The next steps are identical to what we've already done:

```
image_datasets = {
  d: ImageFolder(f'{DATA_DIR}/{d}', transforms[d]) for d in DATASETS
}

data_loaders = {
  d: DataLoader(image_datasets[d], batch_size=4, shuffle=True, num_workers=4)
  for d in DATASETS
}

dataset_sizes = {d: len(image_datasets[d]) for d in DATASETS}
```

```
class_names = image_datasets['train'].classes

dataset_sizes
```

```
    {'test': 784, 'train': 5704, 'val': 794}
```

```
%%time

enchanced_model = create_model(len(class_names))
enchanced_model, history = train_model(enchanced_model, data_loaders, dataset_sizes,\
 device)
```

```
    Epoch 1/3
    ----------
    Train loss 0.39523224640235327 accuracy 0.8650070126227208
    Val   loss 0.002290595416447625 accuracy 1.0

    Epoch 2/3
    ----------
    Train loss 0.173455789528505 accuracy 0.9446002805049089
    Val   loss 0.030148923471944415 accuracy 0.9886649874055415

    Epoch 3/3
    ----------
    Train loss 0.11575758963990512 accuracy 0.9640603085553997
    Val   loss 0.0014996432778823317 accuracy 1.0

    Best val accuracy: 1.0
    CPU times: user 2min 47s, sys: 56.2 s, total: 3min 44s
    Wall time: 3min 53s
```

```
plot_training_history(history)
```

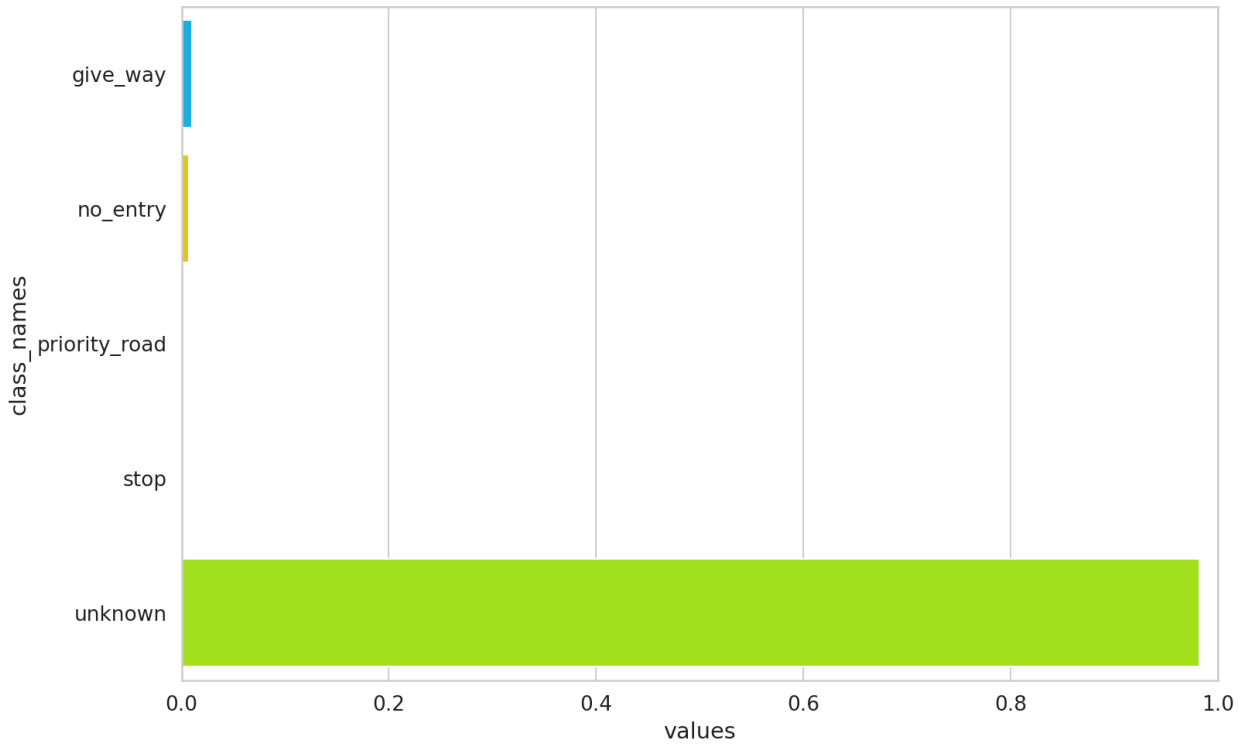
png

Again, our model is learning very quickly. Let's have a look at the sample image again:

```
1 show_image('unknown-sign.jpg')
```

png

```
pred = predict_proba(enchanced_model, 'unknown-sign.jpg')
show_prediction_confidence(pred, class_names)
```



png

Great, the model doesn't give much weight to any of the known classes. It doesn't magically know that this is a two-way sign, but recognizes is as unknown.

Let's have a look at some examples of our new dataset:

```
1 show_predictions(enchanced_model, class_names, n_images=8)
```

png

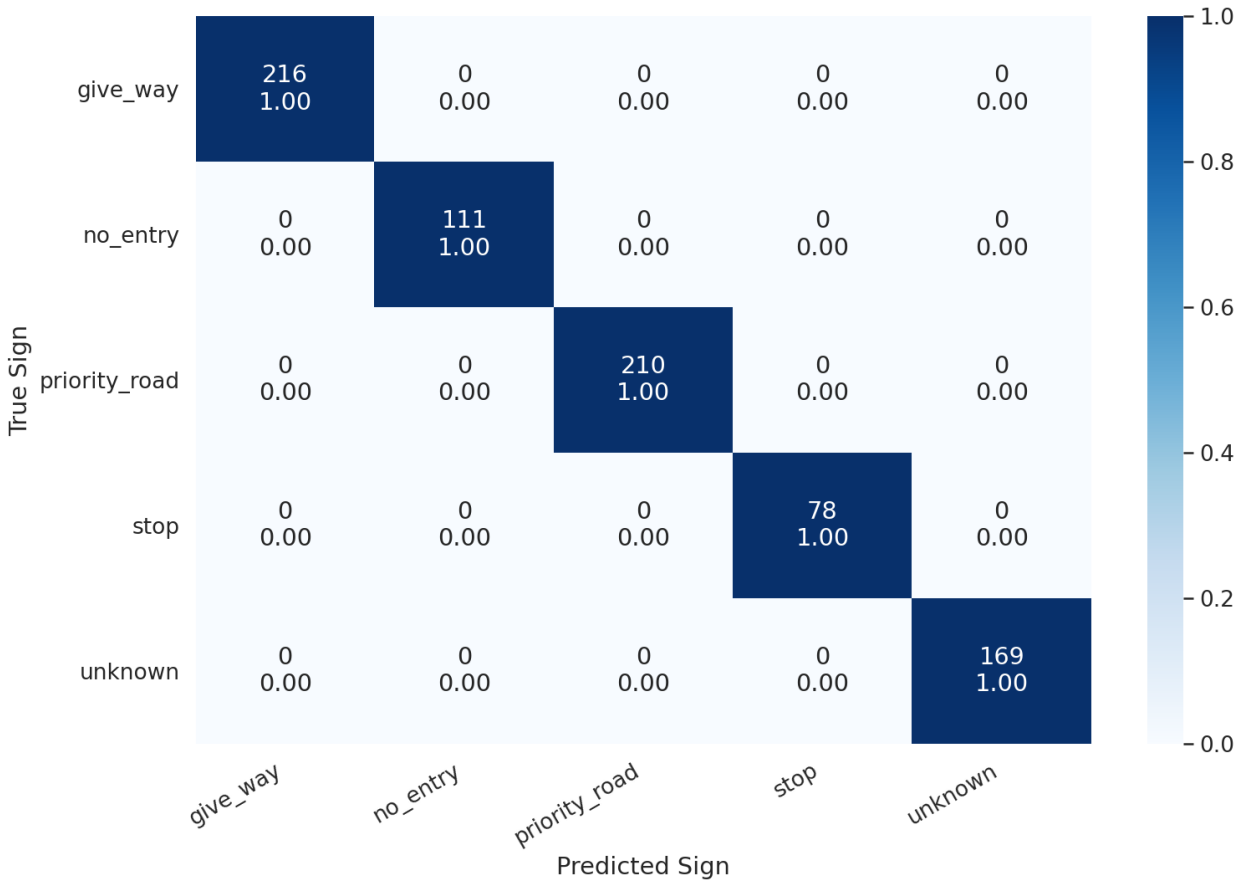
Let's get an overview of the new model's performance:

```
y_pred, y_test = get_predictions(enchanced_model, data_loaders['test'])
```

```
print(classification_report(y_test, y_pred, target_names=class_names))
```

```
                  precision    recall  f1-score   support

       give_way       1.00      1.00      1.00       216
       no_entry       1.00      1.00      1.00       111
  priority_road       1.00      1.00      1.00       210
           stop       1.00      1.00      1.00        78
        unknown       1.00      1.00      1.00       169

       accuracy                           1.00       784
      macro avg       1.00      1.00      1.00       784
   weighted avg       1.00      1.00      1.00       784
```

```
cm = confusion_matrix(y_test, y_pred)
show_confusion_matrix(cm, class_names)
```



png

Our model is still perfect. Go ahead, try it on more images!

Summary

Good job! You trained two different models for classifying traffic signs from raw pixels. You also built a dataset that is compatible with Torchvision.

- Read the tutorial[10]
- Run the notebook in your browser (Google Colab)[11]
- Read the `Getting Things Done with Pytorch` book[12]

[10] https://www.curiousily.com/posts/transfer-learning-for-image-classification-using-torchvision-pytorch-and-python/
[11] https://colab.research.google.com/drive/1Lk5R4pECDxDhd1uXcv26YRQ02fb_mrL9?usp=sharing
[12] https://github.com/curiousily/Getting-Things-Done-with-Pytorch

Here's what you've learned:

- Overview of the traffic sign image dataset
- Build a dataset
- Use a pre-trained model from Torchvision
- Add a new *unknown* class and re-train the model

Can you use transfer learning for other tasks? How do you do it? Let me know in the comments below.

References

- ResNet: the intuition behind it[13]
- Understanding ResNet Intuitively[14]
- Conv Nets: A Modular Perspective[15]
- An intuitive guide to Convolutional Neural Networks[16]
- A friendly introduction to Convolutional Neural Networks and Image Recognition[17]
- A Baseline for Detecting Misclassified and Out-of-Distribution Examples in Neural Networks[18]
- How to Train an Image Classifier in PyTorch and use it to Perform Basic Inference on Single Images[19]
- Transfer Learning with Convolutional Neural Networks in PyTorch[20]
- Image Classification with Transfer Learning and PyTorch[21]

[13] https://wiseodd.github.io/techblog/2016/10/13/residual-net/
[14] https://mc.ai/understanding-resnet-intuitively/
[15] https://colah.github.io/posts/2014-07-Conv-Nets-Modular/
[16] https://www.freecodecamp.org/news/an-intuitive-guide-to-convolutional-neural-networks-260c2de0a050/
[17] https://youtu.be/2-Ol7ZB0MmU?t=721
[18] https://arxiv.org/pdf/1610.02136.pdf
[19] https://towardsdatascience.com/how-to-train-an-image-classifier-in-pytorch-and-use-it-to-perform-basic-inference-on-single-images-99465a1e9bf5
[20] https://towardsdatascience.com/transfer-learning-with-convolutional-neural-networks-in-pytorch-dd09190245ce
[21] https://stackabuse.com/image-classification-with-transfer-learning-and-pytorch/

4. Time Series Forecasting with LSTMs for Daily Coronavirus Cases

> This tutorial is NOT trying to build a model that predicts the Covid-19 outbreak/pandemic in the best way possible. This is an example of how you can use Recurrent Neural Networks on some real-world Time Series data with PyTorch. Hopefully, there are much better models that predict the number of daily confirmed cases.

Time series data captures a series of data points recorded at (usually) regular intervals. Some common examples include daily weather temperature, stock prices, and the number of sales a company makes.

Many classical methods (e.g. ARIMA) try to deal with Time Series data with varying success (not to say they are bad at it). In the last couple of years, Long Short Term Memory Networks (LSTM)[1] models have become a very useful method when dealing with those types of data.

Recurrent Neural Networks (LSTMs are one type of those) are very good at processing sequences of data. They can "recall" patterns in the data that are very far into the past (or future). In this tutorial, you're going to learn how to use LSTMs to predict future Coronavirus cases based on real-world data.

- Run the complete notebook in your browser (Google Colab)[2]
- Read the Getting Things Done with Pytorch book[3]

Novel Coronavirus (COVID-19)

The novel Coronavirus (Covid-19) has spread around the world very rapidly. At the time of this writing, Worldometers.info[4] shows that there are more than *95,488* confirmed cases in more than *84* countries.

The top 4 worst-affected (by far) are China (the source of the virus), South Korea, Italy, and Iran. Unfortunately, many cases are currently not reported due to:

- A person can get infected without even knowing (asymptomatic)
- Incorrect data reporting
- Not enough test kits
- The symptoms look a lot like the common flu

[1] https://en.wikipedia.org/wiki/Long_short-term_memory
[2] https://colab.research.google.com/drive/1nQYJq1f7f4R0yeZOzQ9rBKgk00AfLoS0
[3] https://github.com/curiousily/Getting-Things-Done-with-Pytorch
[4] https://www.worldometers.info/coronavirus/

How dangerous is this virus?

Except for the common statistics you might see cited on the news, there are some good and some bad news:

- More than 80% of the confirmed cases recover without any need of medical attention
- 3.4% Mortality Rate estimate by the World Health Organization (WHO) as of March 3[5]
- The reproductive number which represents the average number of people to which a single infected person will transmit the virus is between 1.4 and 2.5 (WHO's estimated on Jan. 23)[6]

The last one is really scary. It sounds like we can witness some crazy exponential growth if appropriate measures are not put in place.

Let's get started!

```
%reload_ext watermark
%watermark -v -p numpy,pandas,torch
```

```
CPython 3.6.9
IPython 5.5.0

numpy 1.17.5
pandas 0.25.3
torch 1.4.0
```

```
import torch

import os
import numpy as np
import pandas as pd
from tqdm import tqdm
import seaborn as sns
from pylab import rcParams
import matplotlib.pyplot as plt
from matplotlib import rc
from sklearn.preprocessing import MinMaxScaler
from pandas.plotting import register_matplotlib_converters
from torch import nn, optim

```

[5] https://www.worldometers.info/coronavirus/coronavirus-death-rate/#who-03-03-20
[6] https://www.worldometers.info/coronavirus/#repro

```
%matplotlib inline
%config InlineBackend.figure_format='retina'

sns.set(style='whitegrid', palette='muted', font_scale=1.2)

HAPPY_COLORS_PALETTE = ["#01BEFE", "#FFDD00", "#FF7D00", "#FF006D", "#93D30C", "#8F0\
0FF"]

sns.set_palette(sns.color_palette(HAPPY_COLORS_PALETTE))

rcParams['figure.figsize'] = 14, 10
register_matplotlib_converters()

RANDOM_SEED = 42
np.random.seed(RANDOM_SEED)
torch.manual_seed(RANDOM_SEED)
```

```
<torch._C.Generator at 0x7faeaa744d30>
```

Daily Cases Dataset

The data is provided by the Johns Hopkins University Center for Systems Science and Engineering (JHU CSSE) and contains the number of reported daily cases by country. The dataset is available on GitHub[7] and is updated regularly.

We're going to take the Time Series data only for confirmed cases (number of deaths and recovered cases are also available):

```
# !wget https://raw.githubusercontent.com/CSSEGISandData/COVID-19/master/csse_covid_\
19_data/csse_covid_19_time_series/time_series_19-covid-Confirmed.csv
```

Or you can take the same dataset that I've used for this tutorial (the data snapshot is from 3 March 2020):

```
!gdown --id 1AsfdLrGESCQnRW5rbMz56A1KBc3Fe5aV
```

Data exploration

Let's load the data and have a peek:

[7] https://github.com/CSSEGISandData/COVID-19

```
df = pd.read_csv('time_series_19-covid-Confirmed.csv')
```

Two things to note here:

- The data contains a province, country, latitude, and longitude. We won't be needing those.
- The number of cases is cumulative. We'll undo the accumulation.

Let's start by getting rid of the first four columns:

```
df = df.iloc[:, 4:]
```

Let's check for missing values:

```
df.isnull().sum().sum()
```

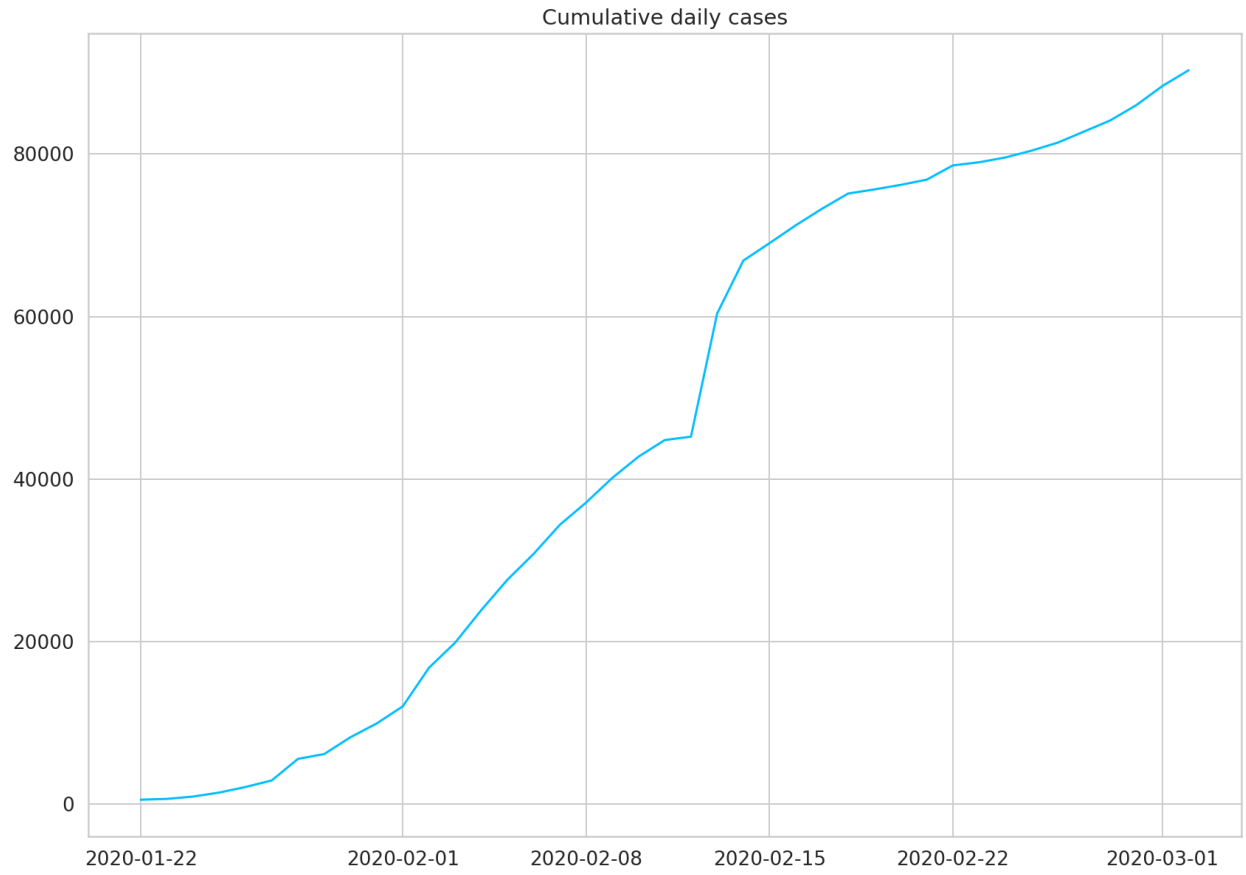
```
0
```

Everything seems to be in place. Let's sum all rows, so we get the cumulative daily cases:

```
daily_cases = df.sum(axis=0)
daily_cases.index = pd.to_datetime(daily_cases.index)
daily_cases.head()
```

```
2020-01-22     555
2020-01-23     653
2020-01-24     941
2020-01-25    1434
2020-01-26    2118
dtype: int64
```

```
plt.plot(daily_cases)
plt.title("Cumulative daily cases");
```



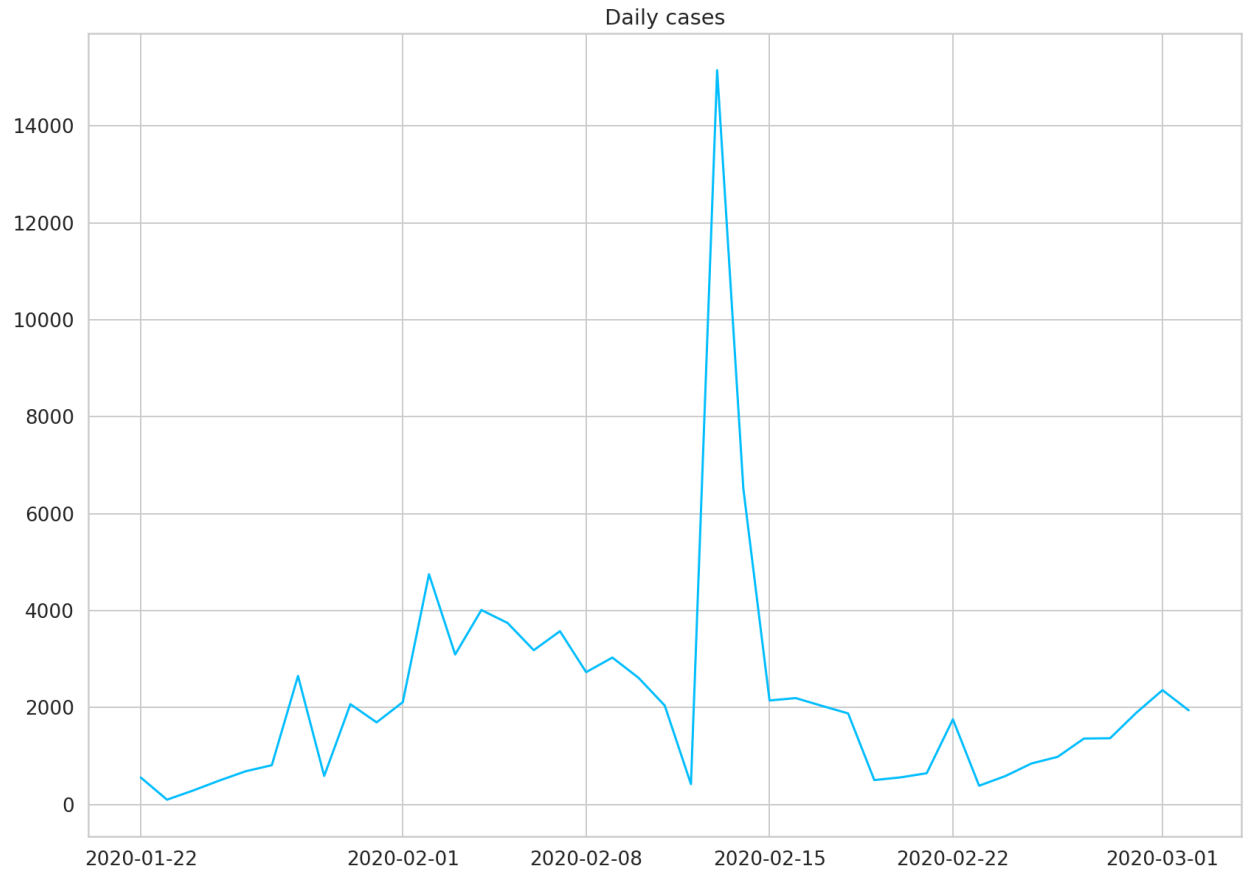
png

We'll undo the accumulation by subtracting the current value from the previous. We'll preserve the first value of the sequence:

```
daily_cases = daily_cases.diff().fillna(daily_cases[0]).astype(np.int64)
daily_cases.head()
```

```
2020-01-22    555
2020-01-23     98
2020-01-24    288
2020-01-25    493
2020-01-26    684
dtype: int64
```

```
plt.plot(daily_cases)
plt.title("Daily cases");
```



png

The huge spike (in the middle) is mostly due to a change of criteria for testing patients in China. This will certainly be a challenge for our model.

Let's check the amount of data we have:

```
daily_cases.shape
```

```
(41,)
```

Unfortunately, we have data for only 41 days. Let's see what we can do with it!

Preprocessing

We'll reserve the first 27 days for training and use the rest for testing:

```
test_data_size = 14

train_data = daily_cases[:-test_data_size]
test_data = daily_cases[-test_data_size:]

train_data.shape
```

```
(27,)
```

We have to scale the data (values will be between 0 and 1) if we want to increase the training speed and performance of the model. We'll use the `MinMaxScaler` from scikit-learn:

```
scaler = MinMaxScaler()

scaler = scaler.fit(np.expand_dims(train_data, axis=1))

train_data = scaler.transform(np.expand_dims(train_data, axis=1))

test_data = scaler.transform(np.expand_dims(test_data, axis=1))
```

Currently, we have a big sequence of daily cases. We'll convert it into smaller ones:

```
def create_sequences(data, seq_length):
    xs = []
    ys = []

    for i in range(len(data)-seq_length-1):
        x = data[i:(i+seq_length)]
        y = data[i+seq_length]
        xs.append(x)
        ys.append(y)

    return np.array(xs), np.array(ys)
```

```
seq_length = 5
X_train, y_train = create_sequences(train_data, seq_length)
X_test, y_test = create_sequences(test_data, seq_length)

X_train = torch.from_numpy(X_train).float()
y_train = torch.from_numpy(y_train).float()

X_test = torch.from_numpy(X_test).float()
y_test = torch.from_numpy(y_test).float()
```

Each training example contains a sequence of 5 data points of history and a label for the real value that our model needs to predict. Let's dive in:

```
X_train.shape
```

```
torch.Size([21, 5, 1])
```

```
X_train[:2]
```

```
tensor([[[0.0304],
         [0.0000],
         [0.0126],
         [0.0262],
         [0.0389]],

        [[0.0000],
         [0.0126],
         [0.0262],
         [0.0389],
         [0.0472]]])
```

```
y_train.shape
```

```
torch.Size([21, 1])
```

```
y_train[:2]
```

```
tensor([[0.0472],
        [0.1696]])
```

```
train_data[:10]
```

```
array([[0.03036545],
       [0.        ],
       [0.01262458],
       [0.02624585],
       [0.03893688],
       [0.04724252],
       [0.16963455],
       [0.03255814],
       [0.13089701],
       [0.10598007]])
```

Building a model

We'll encapsulate the complexity of our model into a class that extends from `torch.nn.Module`:

```
class CoronaVirusPredictor(nn.Module):

  def __init__(self, n_features, n_hidden, seq_len, n_layers=2):
    super(CoronaVirusPredictor, self).__init__()

    self.n_hidden = n_hidden
    self.seq_len = seq_len
    self.n_layers = n_layers

    self.lstm = nn.LSTM(
      input_size=n_features,
      hidden_size=n_hidden,
      num_layers=n_layers,
      dropout=0.5
    )

    self.linear = nn.Linear(in_features=n_hidden, out_features=1)

  def reset_hidden_state(self):
```

```
    self.hidden = (
        torch.zeros(self.n_layers, self.seq_len, self.n_hidden),
        torch.zeros(self.n_layers, self.seq_len, self.n_hidden)
    )

  def forward(self, sequences):
    lstm_out, self.hidden = self.lstm(
      sequences.view(len(sequences), self.seq_len, -1),
      self.hidden
    )
    last_time_step = \
      lstm_out.view(self.seq_len, len(sequences), self.n_hidden)[-1]
    y_pred = self.linear(last_time_step)
    return y_pred
```

Our `CoronaVirusPredictor` contains 3 methods:

- constructor - initialize all helper data and create the layers
- `reset_hidden_state` - we'll use a stateless LSTM, so we need to reset the state after each example
- `forward` - get the sequences, pass all of them through the LSTM layer, at once. We take the output of the last time step and pass it through our linear layer to get the prediction.

Training

Let's build a helper function for the training of our model (we'll reuse it later):

```
def train_model(
  model,
  train_data,
  train_labels,
  test_data=None,
  test_labels=None
):
  loss_fn = torch.nn.MSELoss(reduction='sum')

  optimiser = torch.optim.Adam(model.parameters(), lr=1e-3)
  num_epochs = 60

  train_hist = np.zeros(num_epochs)
  test_hist = np.zeros(num_epochs)
```

```
15
16   for t in range(num_epochs):
17     model.reset_hidden_state()
18
19     y_pred = model(X_train)
20
21     loss = loss_fn(y_pred.float(), y_train)
22
23     if test_data is not None:
24       with torch.no_grad():
25         y_test_pred = model(X_test)
26         test_loss = loss_fn(y_test_pred.float(), y_test)
27       test_hist[t] = test_loss.item()
28
29       if t % 10 == 0:
30         print(f'Epoch {t} train loss: {loss.item()} test loss: {test_loss.item()}')
31     elif t % 10 == 0:
32       print(f'Epoch {t} train loss: {loss.item()}')
33
34     train_hist[t] = loss.item()
35
36     optimiser.zero_grad()
37
38     loss.backward()
39
40     optimiser.step()
41
42   return model.eval(), train_hist, test_hist
```

Note that the hidden state is reset at the start of each epoch. We don't use batches of data our model sees every example at once. We'll use mean squared error to measure our training and test error. We'll record both.

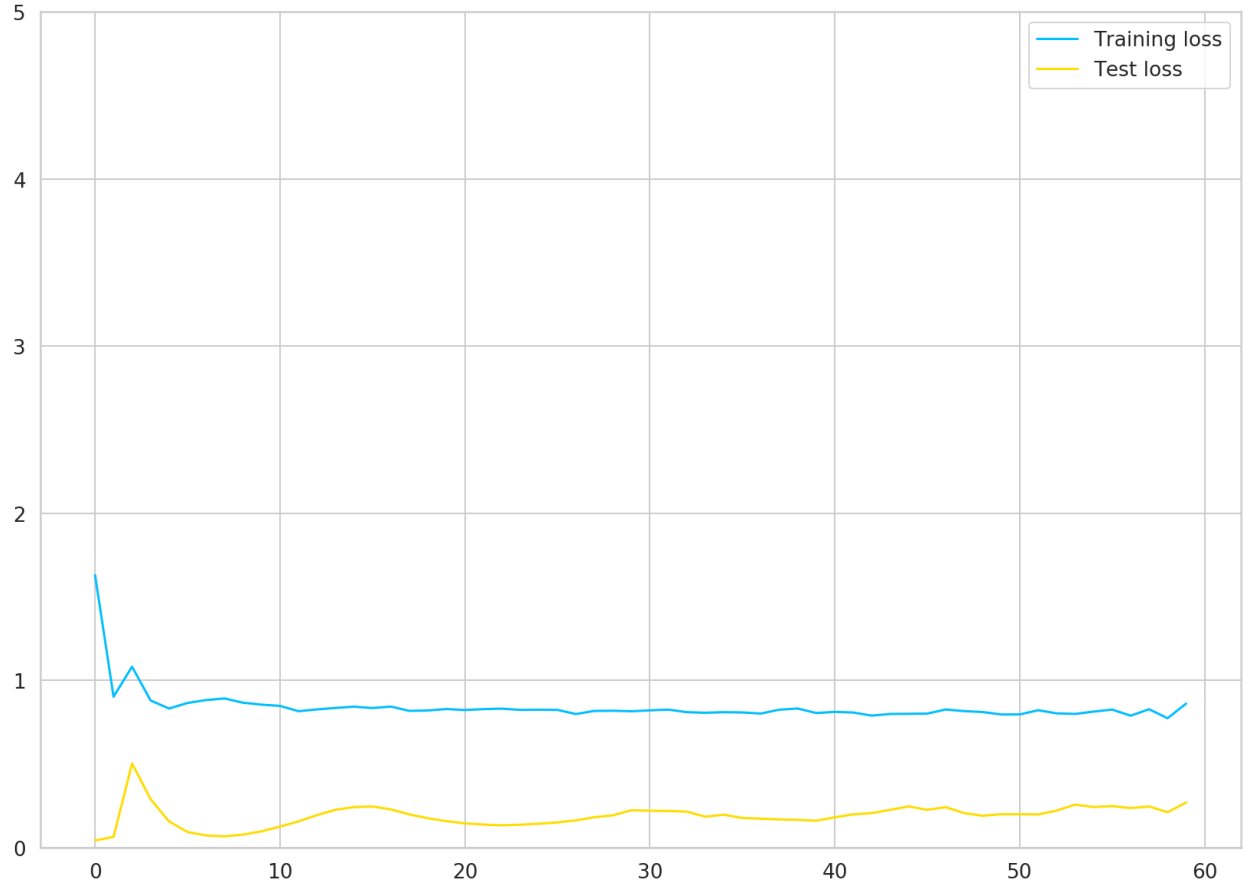
Let's create an instance of our model and train it:

```
model = CoronaVirusPredictor(
  n_features=1,
  n_hidden=512,
  seq_len=seq_length,
  n_layers=2
)
model, train_hist, test_hist = train_model(
  model,
  X_train,
  y_train,
  X_test,
  y_test
)
```

```
Epoch 0 train loss: 1.6297188997268677 test loss: 0.041186608374118805
Epoch 10 train loss: 0.8466923832893372 test loss: 0.12416432797908783
Epoch 20 train loss: 0.8219934105873108 test loss: 0.1438201516866684
Epoch 30 train loss: 0.8200693726539612 test loss: 0.2190694659948349
Epoch 40 train loss: 0.810839056968689 test loss: 0.1797715127468109
Epoch 50 train loss: 0.795730471611023 test loss: 0.19855864346027374
```

Let's have a look at the train and test loss:

```
plt.plot(train_hist, label="Training loss")
plt.plot(test_hist, label="Test loss")
plt.ylim((0, 5))
plt.legend();
```



png

Our model's performance doesn't improve after 15 epochs or so. Recall that we have very little data. Maybe we shouldn't trust our model that much?

Predicting daily cases

Our model can (due to the way we've trained it) predict only a single day in the future. We'll employ a simple strategy to overcome this limitation. Use predicted values as input for predicting the next days:

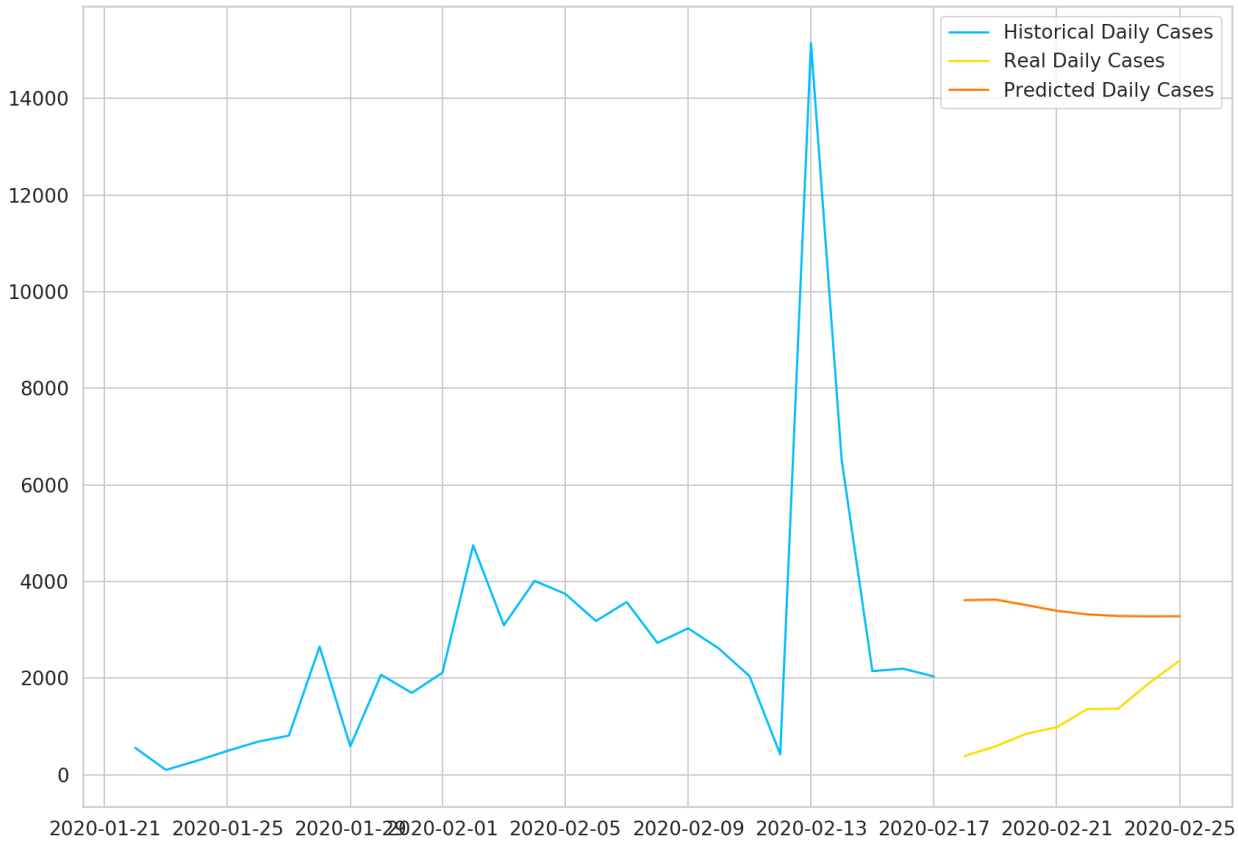
```
with torch.no_grad():
  test_seq = X_test[:1]
  preds = []
  for _ in range(len(X_test)):
    y_test_pred = model(test_seq)
    pred = torch.flatten(y_test_pred).item()
    preds.append(pred)
    new_seq = test_seq.numpy().flatten()
    new_seq = np.append(new_seq, [pred])
    new_seq = new_seq[1:]
    test_seq = torch.as_tensor(new_seq).view(1, seq_length, 1).float()
```

We have to reverse the scaling of the test data and the model predictions:

```
true_cases = scaler.inverse_transform(
    np.expand_dims(y_test.flatten().numpy(), axis=0)
).flatten()

predicted_cases = scaler.inverse_transform(
  np.expand_dims(preds, axis=0)
).flatten()
```

Let's look at the results:

```
plt.plot(
  daily_cases.index[:len(train_data)],
  scaler.inverse_transform(train_data).flatten(),
  label='Historical Daily Cases'
)

plt.plot(
  daily_cases.index[len(train_data):len(train_data) + len(true_cases)],
  true_cases,
  label='Real Daily Cases'
)

plt.plot(
  daily_cases.index[len(train_data):len(train_data) + len(true_cases)],
  predicted_cases,
  label='Predicted Daily Cases'
)

plt.legend();
```



png

As expected, our model doesn't perform very well. That said, the predictions seem to be in the right ballpark (probably due to using the last data point as a strong predictor for the next).

Use all data for training

Now, we'll use all available data to train the same model:

```
scaler = MinMaxScaler()

scaler = scaler.fit(np.expand_dims(daily_cases, axis=1))

all_data = scaler.transform(np.expand_dims(daily_cases, axis=1))

all_data.shape
```

```
(41, 1)
```

The preprocessing and training steps are the same:

```
X_all, y_all = create_sequences(all_data, seq_length)

X_all = torch.from_numpy(X_all).float()
y_all = torch.from_numpy(y_all).float()

model = CoronaVirusPredictor(
  n_features=1,
  n_hidden=512,
  seq_len=seq_length,
  n_layers=2
)
model, train_hist, _ = train_model(model, X_all, y_all)
```

```
Epoch 0 train loss: 1.9441421031951904
Epoch 10 train loss: 0.8385428786277771
Epoch 20 train loss: 0.8256545066833496
Epoch 30 train loss: 0.8023681640625
Epoch 40 train loss: 0.8125611543655396
Epoch 50 train loss: 0.8225002884864807
```

Predicting future cases

We'll use our "fully trained" model to predict the confirmed cases for 12 days into the future:

```
DAYS_TO_PREDICT = 12

with torch.no_grad():
  test_seq = X_all[:1]
  preds = []
  for _ in range(DAYS_TO_PREDICT):
    y_test_pred = model(test_seq)
    pred = torch.flatten(y_test_pred).item()
    preds.append(pred)
    new_seq = test_seq.numpy().flatten()
    new_seq = np.append(new_seq, [pred])
    new_seq = new_seq[1:]
    test_seq = torch.as_tensor(new_seq).view(1, seq_length, 1).float()
```

As before, we'll inverse the scaler transformation:

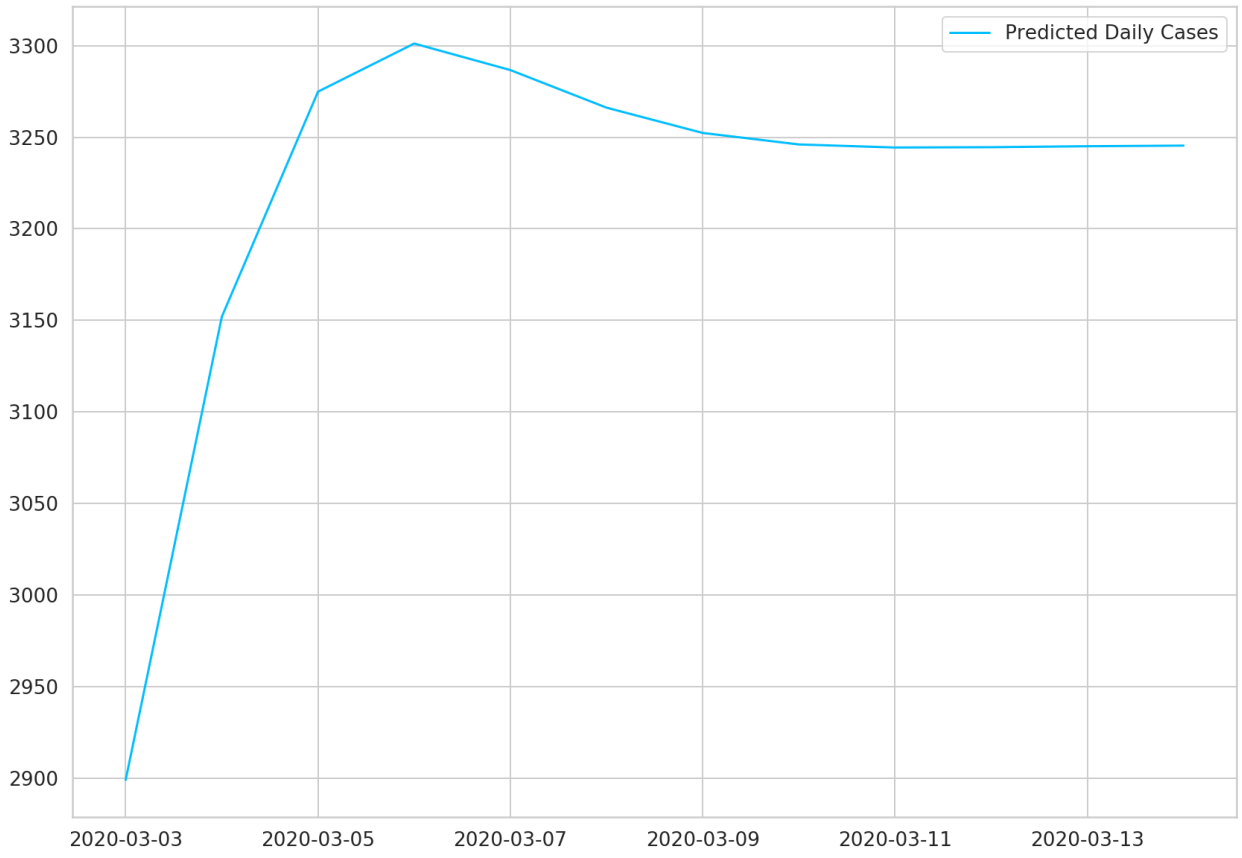
```
predicted_cases = scaler.inverse_transform(
  np.expand_dims(preds, axis=0)
).flatten()
```

To create a cool chart with the historical and predicted cases, we need to extend the date index of our data frame:

```
daily_cases.index[-1]
```

```
Timestamp('2020-03-02 00:00:00')
```

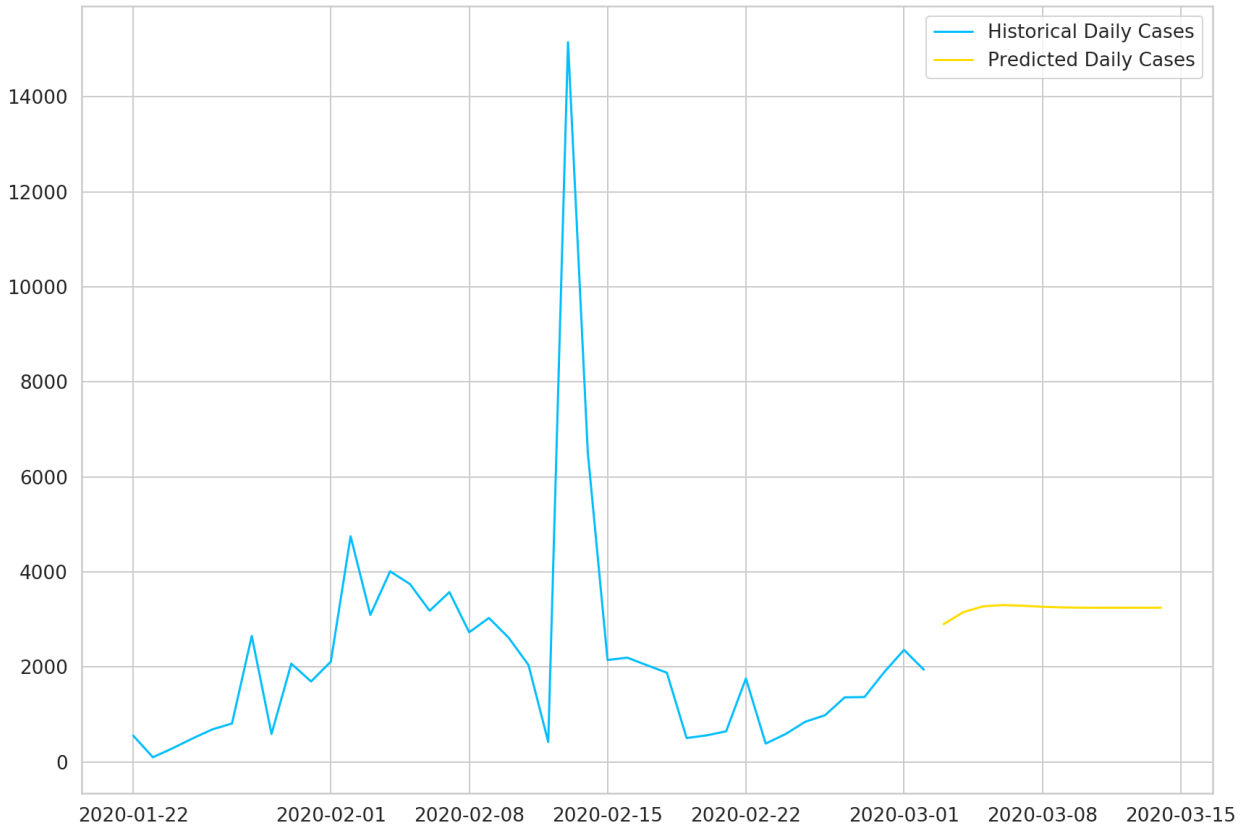
```
predicted_index = pd.date_range(
  start=daily_cases.index[-1],
  periods=DAYS_TO_PREDICT + 1,
  closed='right'
)

predicted_cases = pd.Series(
  data=predicted_cases,
  index=predicted_index
)

plt.plot(predicted_cases, label='Predicted Daily Cases')
plt.legend();
```



png

Now we can use all the data to plot the results:

```
plt.plot(daily_cases, label='Historical Daily Cases')
plt.plot(predicted_cases, label='Predicted Daily Cases')
plt.legend();
```



png

Our model thinks that things will level off. Note that the more you go into the future, the more you shouldn't trust your model predictions.

Conclusion

Well done! You learned how to use PyTorch to create a Recurrent Neural Network that works with Time Series data. The model performance is not that great, but this is expected, given the small amounts of data.

- Run the complete notebook in your browser (Google Colab)[8]
- Read the Getting Things Done with Pytorch book[9]

The problem of predicting daily Covid-19 cases is a hard one. We're amidst an outbreak, and there's more to be done. Hopefully, everything will be back to normal after some time.

[8] https://colab.research.google.com/drive/1nQYJq1f7f4R0yeZOzQ9rBKgk00AfLoS0
[9] https://github.com/curiousily/Getting-Things-Done-with-Pytorch

References

- Sequence Models PyTorch Tutorial[10]
- LSTM for time series prediction[11]
- Time Series Prediction using LSTM with PyTorch in Python[12]
- Stateful LSTM in Keras[13]
- LSTMs for Time Series in PyTorch[14]
- Novel Coronavirus (COVID-19) Cases, provided by JHU CSSE[15]
- covid-19-analysis[16]
- How does Coronavirus compare to Ebola, SARS, etc?[17]
- Worldometer COVID-19 Coronavirus Outbreak[18]
- How contagious is the Wuhan Coronavirus? (Ro)[19]
- Systemic Risk of Pandemic via Novel Pathogens - Coronavirus: A Note[20]
- Statistical Consequences of Fat Tails: Real World Preasymptotics, Epistemology, and Applications[21]

[10] https://pytorch.org/tutorials/beginner/nlp/sequence_models_tutorial.html
[11] https://towardsdatascience.com/lstm-for-time-series-prediction-de8aeb26f2ca
[12] https://stackabuse.com/time-series-prediction-using-lstm-with-pytorch-in-python/
[13] https://philipperemy.github.io/keras-stateful-lstm/
[14] https://www.jessicayung.com/lstms-for-time-series-in-pytorch/
[15] https://github.com/CSSEGISandData/COVID-19
[16] https://github.com/AaronWard/covid-19-analysis
[17] https://www.youtube.com/watch?v=6dDD2tHWWnU
[18] https://www.worldometers.info/coronavirus/
[19] https://www.worldometers.info/coronavirus/#repro
[20] https://www.academia.edu/41743064/Systemic_Risk_of_Pandemic_via_Novel_Pathogens_-_Coronavirus_A_Note
[21] https://www.researchers.one/article/2020-01-21

5. Time Series Anomaly Detection using LSTM Autoencoders

> TL;DR Use real-world Electrocardiogram (ECG) data to detect anomalies in a patient heartbeat. We'll build an LSTM Autoencoder, train it on a set of normal heartbeats and classify unseen examples as normal or anomalies

In this tutorial, you'll learn how to detect anomalies in Time Series data using an LSTM Autoencoder. You're going to use real-world ECG data from a single patient with heart disease to detect abnormal hearbeats.

- Run the complete notebook in your browser (Google Colab)[1]
- Read the Getting Things Done with Pytorch book[2]

By the end of this tutorial, you'll learn how to:

- Prepare a dataset for Anomaly Detection from Time Series Data
- Build an LSTM Autoencoder with PyTorch
- Train and evaluate your model
- Choose a threshold for anomaly detection
- Classify unseen examples as normal or anomaly

Data

The dataset[3] contains 5,000 Time Series examples (obtained with ECG) with 140 timesteps. Each sequence corresponds to a single heartbeat from a single patient with congestive heart failure.

> An electrocardiogram (ECG or EKG) is a test that checks how your heart is functioning by measuring the electrical activity of the heart. With each heart beat, an electrical impulse (or wave) travels through your heart. This wave causes the muscle to squeeze and pump blood from the heart. Source[4]

We have 5 types of hearbeats (classes):

[1] https://colab.research.google.com/drive/1_J2MrBSvsJfOcVmYAN2-WSp36BtsFZCa
[2] https://github.com/curiousily/Getting-Things-Done-with-Pytorch
[3] http://timeseriesclassification.com/description.php?Dataset=ECG5000
[4] https://www.heartandstroke.ca/heart/tests/electrocardiogram

- Normal (N)
- R-on-T Premature Ventricular Contraction (R-on-T PVC)
- Premature Ventricular Contraction (PVC)
- Supra-ventricular Premature or Ectopic Beat (SP or EB)
- Unclassified Beat (UB).

> Assuming a healthy heart and a typical rate of 70 to 75 beats per minute, each cardiac cycle, or heartbeat, takes about 0.8 seconds to complete the cycle. Frequency: 60–100 per minute (Humans) Duration: 0.6–1 second (Humans) Source[5]

The dataset is available on my Google Drive. Let's get it:

```
!gdown --id 16MIleqoIr1vYxlGk4GKnGmrsCPuWkkpT
```

```
!unzip -qq ECG5000.zip
```

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

The data comes in multiple formats. We'll load the `arff` files into Pandas data frames:

```
with open('ECG5000_TRAIN.arff') as f:
  train = a2p.load(f)

with open('ECG5000_TEST.arff') as f:
  test = a2p.load(f)
```

We'll combine the training and test data into a single data frame. This will give us more data to train our Autoencoder. We'll also shuffle it:

```
df = train.append(test)
df = df.sample(frac=1.0)
df.shape
```

```
(5000, 141)
```

We have 5,000 examples. Each row represents a single heartbeat record. Let's name the possible classes:

[5] https://en.wikipedia.org/wiki/Cardiac_cycle

```
CLASS_NORMAL = 1

class_names = ['Normal','R on T','PVC','SP','UB']
```

Next, we'll rename the last column to `target`, so its easier to reference it:

```
new_columns = list(df.columns)
new_columns[-1] = 'target'
df.columns = new_columns
```

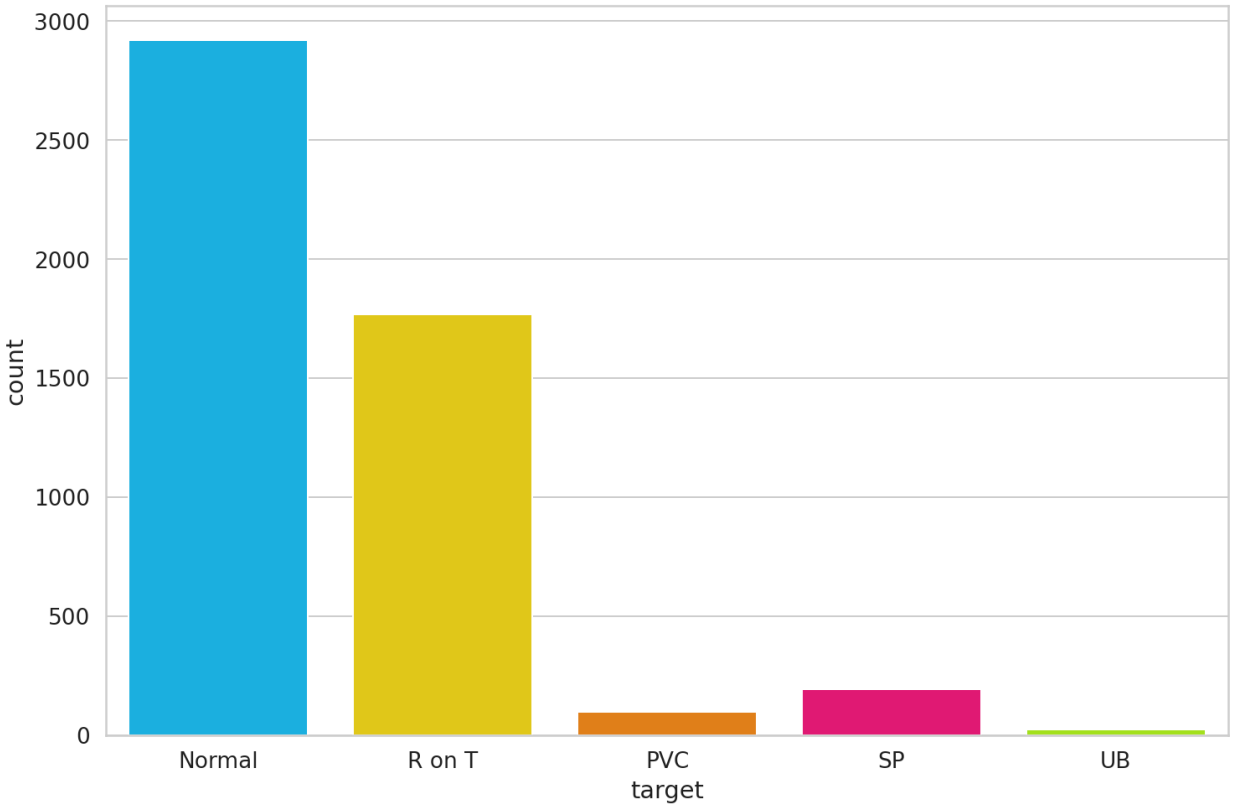
Exploratory Data Analysis

Let's check how many examples for each heartbeat class do we have:

```
df.target.value_counts()
```

```
1    2919
2    1767
4     194
3      96
5      24
Name: target, dtype: int64
```

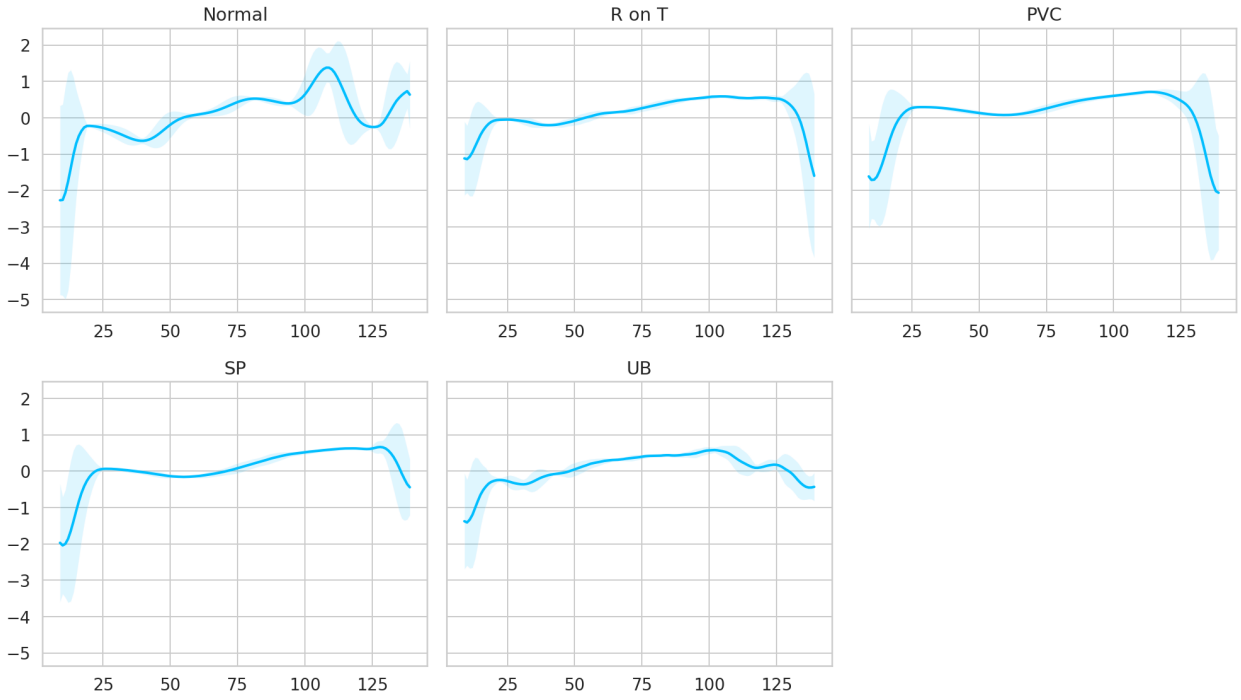
Let's plot the results:



png

The normal class, has by far, the most examples. This is great because we'll use it to train our model.

Let's have a look at an averaged (smoothed out with one standard deviation on top and bottom of it) Time Series for each class:



png

It is very good that the normal class has a distinctly different pattern than all other classes. Maybe our model will be able to detect anomalies?

LSTM Autoencoder

The Autoencoder's[6] job is to get some input data, pass it through the model, and obtain a reconstruction of the input. The reconstruction should match the input as much as possible. The trick is to use a small number of parameters, so your model learns a compressed representation of the data.

In a sense, Autoencoders try to learn only the most important features (compressed version) of the data. Here, we'll have a look at how to feed Time Series data to an Autoencoder. We'll use a couple of LSTM layers (hence the LSTM Autoencoder) to capture the temporal dependencies of the data.

To classify a sequence as normal or an anomaly, we'll pick a threshold above which a heartbeat is considered abnormal.

Reconstruction Loss

When training an Autoencoder, the objective is to reconstruct the input as best as possible. This is done by minimizing a loss function (just like in supervised learning). This function is known as

[6] https://en.wikipedia.org/wiki/Autoencoder

reconstruction loss. Cross-entropy loss and Mean squared error are common examples.

Anomaly Detection in ECG Data

We'll use normal heartbeats as training data for our model and record the *reconstruction loss*. But first, we need to prepare the data:

Data Preprocessing

Let's get all normal heartbeats and drop the target (class) column:

```
normal_df = df[df.target == str(CLASS_NORMAL)].drop(labels='target', axis=1)
normal_df.shape
```

```
(2919, 140)
```

We'll merge all other classes and mark them as anomalies:

```
anomaly_df = df[df.target != str(CLASS_NORMAL)].drop(labels='target', axis=1)
anomaly_df.shape
```

```
(2081, 140)
```

We'll split the normal examples into train, validation and test sets:

```
train_df, val_df = train_test_split(
  normal_df,
  test_size=0.15,
  random_state=RANDOM_SEED
)

val_df, test_df = train_test_split(
  val_df,
  test_size=0.33,
  random_state=RANDOM_SEED
)
```

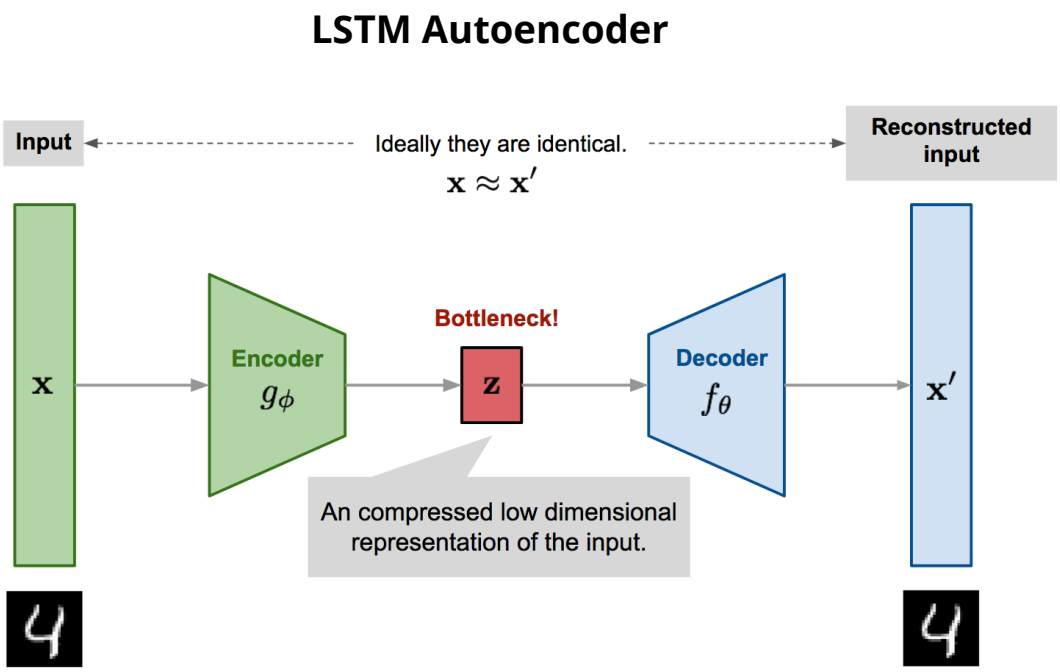
We need to convert our examples into tensors, so we can use them to train our Autoencoder. Let's write a helper function for that:

```
def create_dataset(df):

  sequences = df.astype(np.float32).to_numpy().tolist()

  dataset = [torch.tensor(s).unsqueeze(1).float() for s in sequences]

  n_seq, seq_len, n_features = torch.stack(dataset).shape

  return dataset, seq_len, n_features
```

Each Time Series will be converted to a 2D Tensor in the shape *sequence length* x *number of features* (140x1 in our case).

Let's create some datasets:

```
train_dataset, seq_len, n_features = create_dataset(train_df)
val_dataset, _, _ = create_dataset(val_df)
test_normal_dataset, _, _ = create_dataset(test_df)
test_anomaly_dataset, _, _ = create_dataset(anomaly_df)
```



Sample Autoencoder Architecture Image Source[7]

The general Autoencoder architecture consists of two components. An *Encoder* that compresses the input and a *Decoder* that tries to reconstruct it.

[7] https://lilianweng.github.io/lil-log/2018/08/12/from-autoencoder-to-beta-vae.html

We'll use the LSTM Autoencoder from this GitHub repo[8] with some small tweaks. Our model's job is to reconstruct Time Series data. Let's start with the *Encoder*:

```
class Encoder(nn.Module):

  def __init__(self, seq_len, n_features, embedding_dim=64):
    super(Encoder, self).__init__()

    self.seq_len, self.n_features = seq_len, n_features
    self.embedding_dim, self.hidden_dim = embedding_dim, 2 * embedding_dim

    self.rnn1 = nn.LSTM(
      input_size=n_features,
      hidden_size=self.hidden_dim,
      num_layers=1,
      batch_first=True
    )

    self.rnn2 = nn.LSTM(
      input_size=self.hidden_dim,
      hidden_size=embedding_dim,
      num_layers=1,
      batch_first=True
    )

  def forward(self, x):
    x = x.reshape((1, self.seq_len, self.n_features))

    x, (_, _) = self.rnn1(x)
    x, (hidden_n, _) = self.rnn2(x)

    return hidden_n.reshape((self.n_features, self.embedding_dim))
```

The *Encoder* uses two LSTM layers to compress the Time Series data input.

Next, we'll decode the compressed representation using a *Decoder*:

[8] https://github.com/shobrook/sequitur

```
class Decoder(nn.Module):

  def __init__(self, seq_len, input_dim=64, n_features=1):
    super(Decoder, self).__init__()

    self.seq_len, self.input_dim = seq_len, input_dim
    self.hidden_dim, self.n_features = 2 * input_dim, n_features

    self.rnn1 = nn.LSTM(
      input_size=input_dim,
      hidden_size=input_dim,
      num_layers=1,
      batch_first=True
    )

    self.rnn2 = nn.LSTM(
      input_size=input_dim,
      hidden_size=self.hidden_dim,
      num_layers=1,
      batch_first=True
    )

    self.output_layer = nn.Linear(self.hidden_dim, n_features)

  def forward(self, x):
    x = x.repeat(self.seq_len, self.n_features)
    x = x.reshape((self.n_features, self.seq_len, self.input_dim))

    x, (hidden_n, cell_n) = self.rnn1(x)
    x, (hidden_n, cell_n) = self.rnn2(x)
    x = x.reshape((self.seq_len, self.hidden_dim))

    return self.output_layer(x)
```

Our Decoder contains two LSTM layers and an output layer that gives the final reconstruction.

Time to wrap everything into an easy to use module:

```
class RecurrentAutoencoder(nn.Module):

  def __init__(self, seq_len, n_features, embedding_dim=64):
    super(RecurrentAutoencoder, self).__init__()

    self.encoder = Encoder(seq_len, n_features, embedding_dim).to(device)
    self.decoder = Decoder(seq_len, embedding_dim, n_features).to(device)

  def forward(self, x):
    x = self.encoder(x)
    x = self.decoder(x)

    return x
```

Our Autoencoder passes the input through the Encoder and Decoder. Let's create an instance of it:

```
model = RecurrentAutoencoder(seq_len, n_features, 128)
model = model.to(device)
```

Training

Let's write a helper function for our training process:

```
def train_model(model, train_dataset, val_dataset, n_epochs):
  optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
  criterion = nn.L1Loss(reduction='sum').to(device)
  history = dict(train=[], val=[])

  best_model_wts = copy.deepcopy(model.state_dict())
  best_loss = 10000.0

  for epoch in range(1, n_epochs + 1):
    model = model.train()

    train_losses = []
    for seq_true in train_dataset:
      optimizer.zero_grad()

      seq_true = seq_true.to(device)
      seq_pred = model(seq_true)
```

```
18
19        loss = criterion(seq_pred, seq_true)
20
21        loss.backward()
22        optimizer.step()
23
24        train_losses.append(loss.item())
25
26      val_losses = []
27      model = model.eval()
28      with torch.no_grad():
29        for seq_true in val_dataset:
30
31          seq_true = seq_true.to(device)
32          seq_pred = model(seq_true)
33
34          loss = criterion(seq_pred, seq_true)
35          val_losses.append(loss.item())
36
37      train_loss = np.mean(train_losses)
38      val_loss = np.mean(val_losses)
39
40      history['train'].append(train_loss)
41      history['val'].append(val_loss)
42
43      if val_loss < best_loss:
44        best_loss = val_loss
45        best_model_wts = copy.deepcopy(model.state_dict())
46
47      print(f'Epoch {epoch}: train loss {train_loss} val loss {val_loss}')
48
49    model.load_state_dict(best_model_wts)
50    return model.eval(), history
```

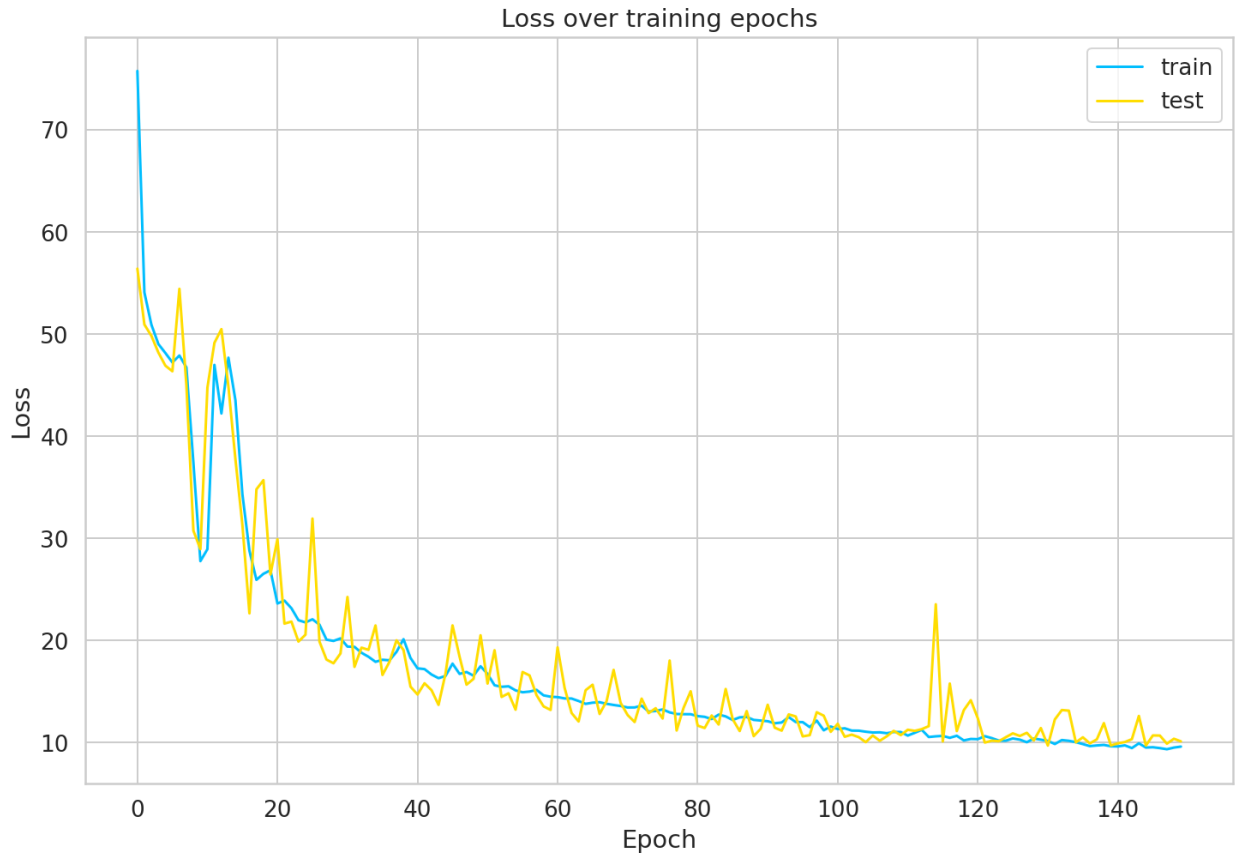
At each epoch, the training process feeds our model with all training examples and evaluates the performance on the validation set. Note that we're using a batch size of 1 (our model sees only 1 sequence at a time). We also record the training and validation set losses during the process.

Note that we're minimizing the L1Loss[9], which measures the MAE (mean absolute error). Why? The reconstructions seem to be better than with MSE (mean squared error).

We'll get the version of the model with the smallest validation error. Let's do some training:

[9] https://pytorch.org/docs/stable/nn.html#l1loss

```
model, history = train_model(
  model,
  train_dataset,
  val_dataset,
  n_epochs=150
)
```



png

Our model converged quite well. Seems like we might've needed a larger validation set to smoothen the results, but that'll do for now.

Saving the model

Let's store the model for later use:

```
MODEL_PATH = 'model.pth'

torch.save(model, MODEL_PATH)
```

Uncomment the next lines, if you want to download and load the pre-trained model:

```
# !gdown --id 1jEYx5wGsb7Ix8cZAw3l5p5pOwHs3_I9A
# model = torch.load('model.pth')
# model = model.to(device)
```

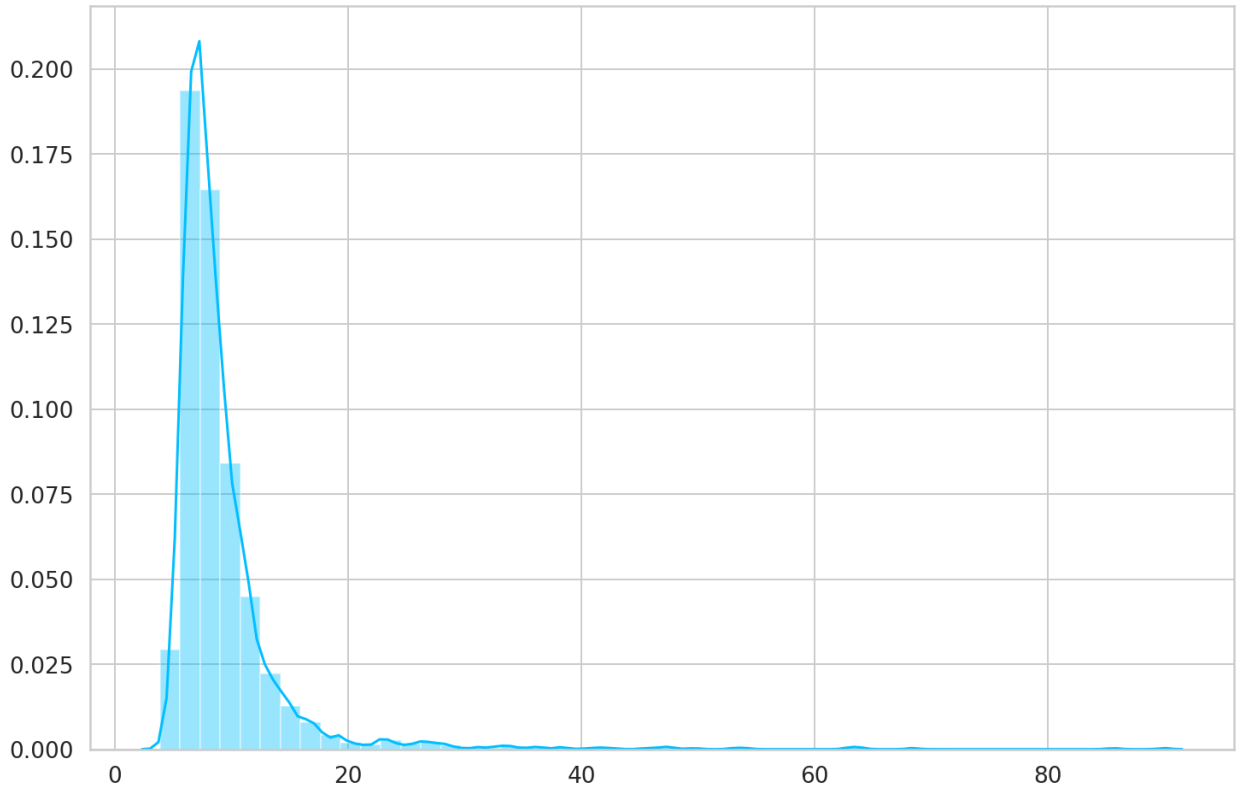
Choosing a threshold

With our model at hand, we can have a look at the reconstruction error on the training set. Let's start by writing a helper function to get predictions from our model:

```
def predict(model, dataset):
  predictions, losses = [], []
  criterion = nn.L1Loss(reduction='sum').to(device)
  with torch.no_grad():
    model = model.eval()
    for seq_true in dataset:
      seq_true = seq_true.to(device)
      seq_pred = model(seq_true)

      loss = criterion(seq_pred, seq_true)

      predictions.append(seq_pred.cpu().numpy().flatten())
      losses.append(loss.item())
  return predictions, losses
```

Our function goes through each example in the dataset and records the predictions and losses. Let's get the losses and have a look at them:

```
_, losses = predict(model, train_dataset)

sns.distplot(losses, bins=50, kde=True);
```



png

```
1 THRESHOLD = 26
```

Evaluation

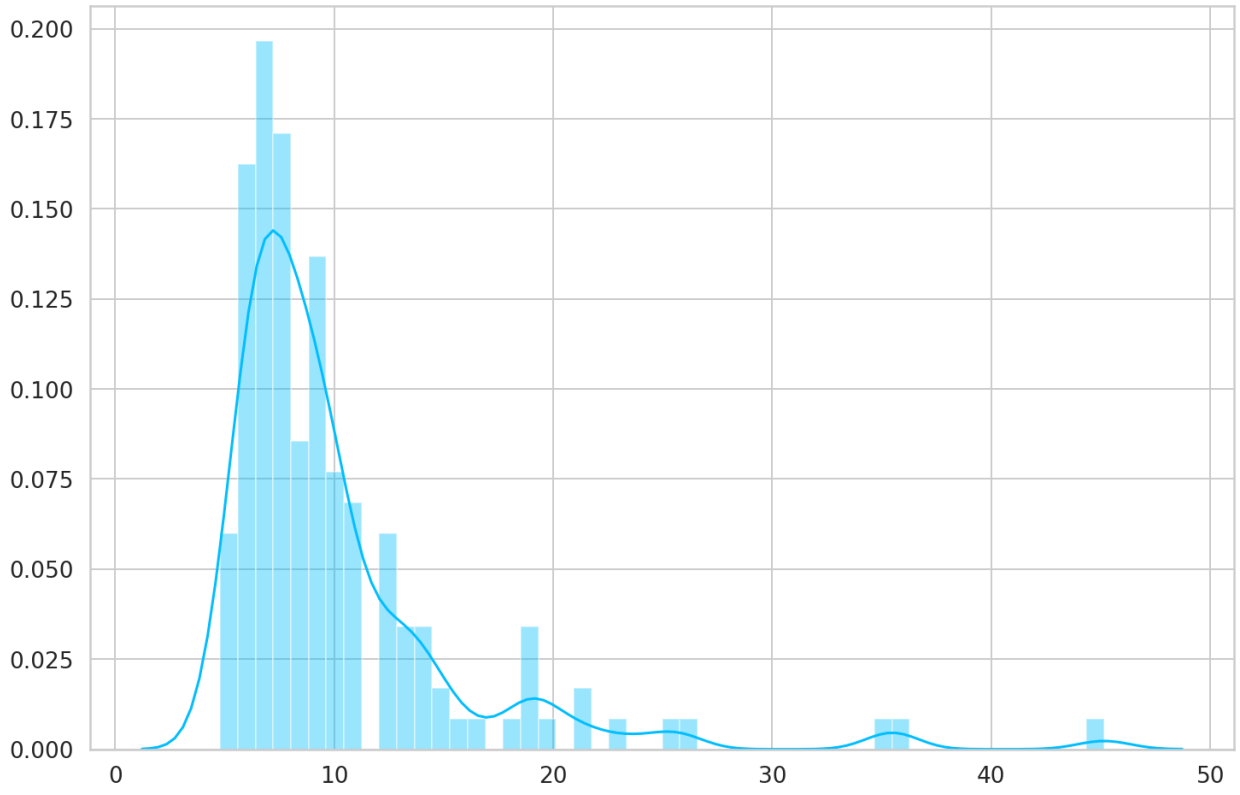
Using the threshold, we can turn the problem into a simple binary classification task:

- If the reconstruction loss for an example is below the threshold, we'll classify it as a *normal* heartbeat
- Alternatively, if the loss is higher than the threshold, we'll classify it as an anomaly

Normal hearbeats

Let's check how well our model does on normal heartbeats. We'll use the normal heartbeats from the test set (our model haven't seen those):

```
predictions, pred_losses = predict(model, test_normal_dataset)
sns.distplot(pred_losses, bins=50, kde=True);
```



png

We'll count the correct predictions:

```
correct = sum(l <= THRESHOLD for l in pred_losses)
print(f'Correct normal predictions: {correct}/{len(test_normal_dataset)}')
```

```
Correct normal predictions: 142/145
```

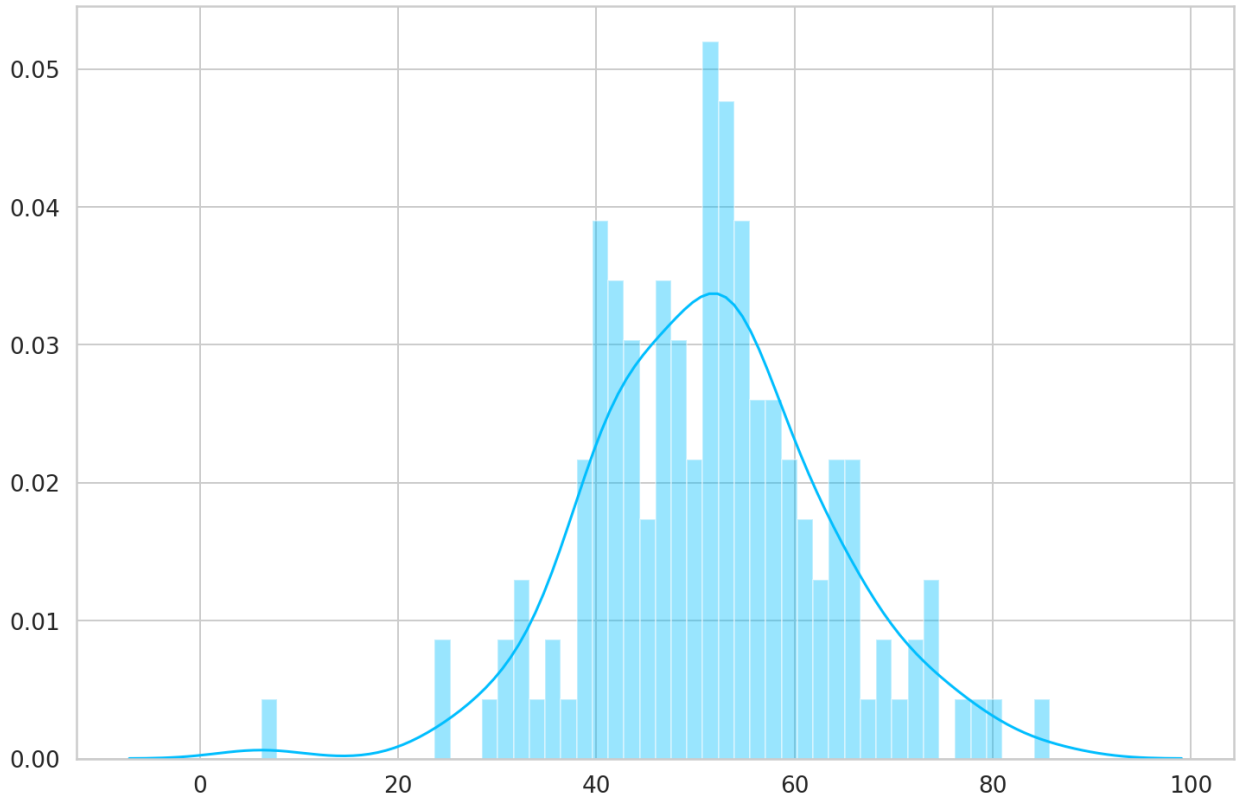
Anomalies

We'll do the same with the anomaly examples, but their number is much higher. We'll get a subset that has the same size as the normal heartbeats:

```
anomaly_dataset = test_anomaly_dataset[:len(test_normal_dataset)]
```

Now we can take the predictions of our model for the subset of anomalies:

```
predictions, pred_losses = predict(model, anomaly_dataset)
sns.distplot(pred_losses, bins=50, kde=True);
```



png

Finally, we can count the number of examples above the threshold (considered as anomalies):

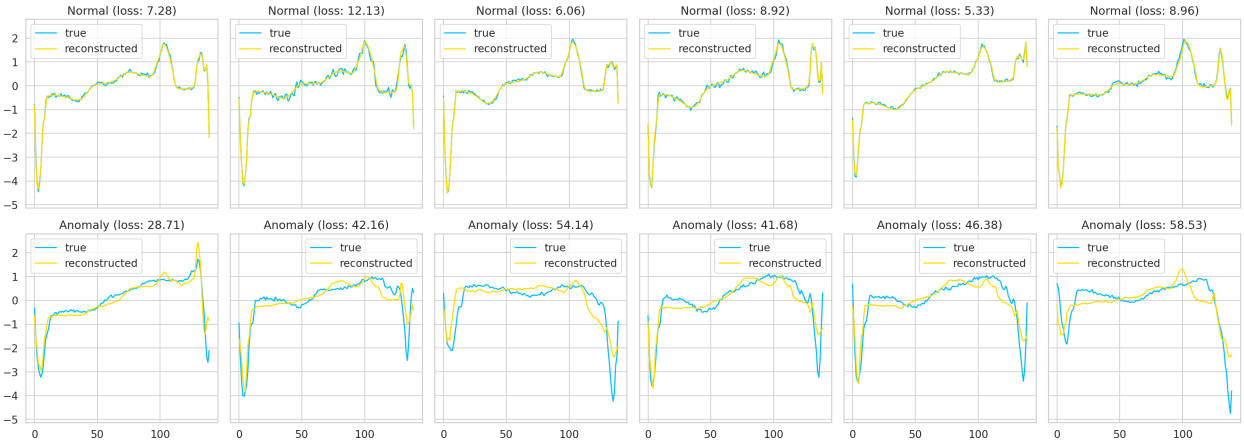
```
correct = sum(l > THRESHOLD for l in pred_losses)
print(f'Correct anomaly predictions: {correct}/{len(anomaly_dataset)}')
```

```
Correct anomaly predictions: 142/145
```

We have very good results. In the real world, you can tweak the threshold depending on what kind of errors you want to tolerate. In this case, you might want to have more false positives (normal heartbeats considered as anomalies) than false negatives (anomalies considered as normal).

Looking at Examples

We can overlay the real and reconstructed Time Series values to see how close they are. We'll do it for some normal and anomaly cases:



png

Summary

In this tutorial, you learned how to create an LSTM Autoencoder with PyTorch and use it to detect heartbeat anomalies in ECG data.

- Run the complete notebook in your browser (Google Colab)[10]
- Read the Getting Things Done with Pytorch book[11]

You learned how to:

- Prepare a dataset for Anomaly Detection from Time Series Data
- Build an LSTM Autoencoder with PyTorch
- Train and evaluate your model
- Choose a threshold for anomaly detection
- Classify unseen examples as normal or anomaly

While our Time Series data is univariate (we have only 1 feature), the code should work for multivariate datasets (multiple features) with little or no modification. Feel free to try it!

References

- Sequitur - Recurrent Autoencoder (RAE)[12]
- Towards Never-Ending Learning from Time Series Streams[13]
- LSTM Autoencoder for Anomaly Detection[14]

[10] https://colab.research.google.com/drive/1_J2MrBSvsJfOcVmYAN2-WSp36BtsFZCa
[11] https://github.com/curiousily/Getting-Things-Done-with-Pytorch
[12] https://github.com/shobrook/sequitur
[13] https://www.cs.ucr.edu/~eamonn/neverending.pdf
[14] https://towardsdatascience.com/lstm-autoencoder-for-anomaly-detection-e1f4f2ee7ccf

6. Face Detection on Custom Dataset with Detectron2

> TL;DR Learn how to prepare a custom Face Detection dataset for Detectron2 and PyTorch. Fine-tune a pre-trained model to find face boundaries in images.

Face detection is the task of finding (boundaries of) faces in images. This is useful for

- security systems (the first step in recognizing a person)
- autofocus and smile detection for making great photos
- detecting age, race, and emotional state for markering (yep, we already live in that world)

Historically, this was a really tough problem to solve. Tons of manual feature engineering, novel algorithms and methods were developed to improve the state-of-the-art.

These days, face detection models are included in almost every computer vision package/framework. Some of the best-performing ones use Deep Learning methods. OpenCV, for example, provides a variety of tools like the Cascade Classifier[1].

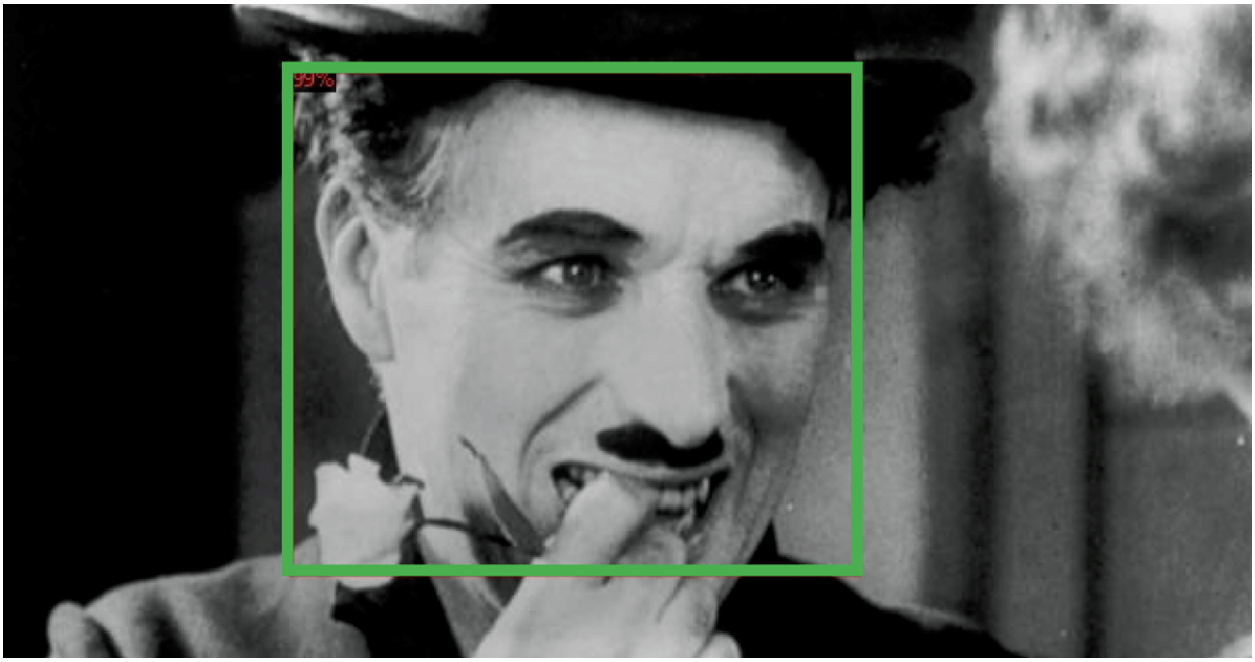
- Run the complete notebook in your browser (Google Colab)[2]
- Read the **Getting Things Done with Pytorch** book[3]

In this guide, you'll learn how to:

- prepare a custom dataset for face detection with Detectron2
- use (close to) state-of-the-art models for object detection to find faces in images
- You can extend this work for face recognition.

Here's an example of what you'll get at the end of this guide:

[1] https://docs.opencv.org/4.2.0/db/d28/tutorial_cascade_classifier.html
[2] https://colab.research.google.com/drive/1Jk4-qX9zdYGsBrTnh2vF52CV9ucuqpjk
[3] https://github.com/curiousily/Getting-Things-Done-with-Pytorch



png

Detectron 2

png

Detectron2[4] is a framework for building state-of-the-art object detection and image segmentation models. It is developed by the Facebook Research team. Detectron2 is a complete rewrite of the first version[5].

Under the hood, Detectron2 uses PyTorch (compatible with the latest version(s)) and allows for blazing fast training[6]. You can learn more at introductory blog post[7] by Facebook Research.

The real power of Detectron2 lies in the HUGE amount of pre-trained models available at the Model Zoo[8]. But what good that would it be if you can't fine-tune those on your own datasets? Fortunately,

[4] https://github.com/facebookresearch/detectron2
[5] https://github.com/facebookresearch/Detectron
[6] https://detectron2.readthedocs.io/notes/benchmarks.html
[7] https://ai.facebook.com/blog/-detectron2-a-pytorch-based-modular-object-detection-library-/
[8] https://github.com/facebookresearch/detectron2/blob/master/MODEL_ZOO.md

that's super easy! We'll see how it is done in this guide.

Installing Detectron2

At the time of this writing, Detectron2 is still in an alpha stage. While there is an official release, we'll clone and compile from the master branch. This should equal version 0.1.

Let's start by installing some requirements:

```
!pip install -q cython pyyaml==5.1
!pip install -q -U 'git+https://github.com/cocodataset/cocoapi.git#subdirectory=Pyth\
onAPI'
```

And download, compile, and install the Detectron2 package:

```
!git clone https://github.com/facebookresearch/detectron2 detectron2_repo
!pip install -q -e detectron2_repo
```

> At this point, you'll need to restart the notebook runtime to continue!

```
!pip install -q -U watermark
```

```
%reload_ext watermark
%watermark -v -p numpy,pandas,pycocotools,torch,torchvision,detectron2
```

```
CPython 3.6.9
IPython 5.5.0

numpy 1.17.5
pandas 0.25.3
pycocotools 2.0
torch 1.4.0
torchvision 0.5.0
detectron2 0.1
```

```
import torch, torchvision
import detectron2
from detectron2.utils.logger import setup_logger
setup_logger()

import glob

import os
import ntpath
import numpy as np
import cv2
import random
import itertools
import pandas as pd
from tqdm import tqdm
import urllib
import json
import PIL.Image as Image

from detectron2 import model_zoo
from detectron2.engine import DefaultPredictor, DefaultTrainer
from detectron2.config import get_cfg
from detectron2.utils.visualizer import Visualizer, ColorMode
from detectron2.data import DatasetCatalog, MetadataCatalog, build_detection_test_lo\
ader
from detectron2.evaluation import COCOEvaluator, inference_on_dataset
from detectron2.structures import BoxMode

import seaborn as sns
from pylab import rcParams
import matplotlib.pyplot as plt
from matplotlib import rc

%matplotlib inline
%config InlineBackend.figure_format='retina'

sns.set(style='whitegrid', palette='muted', font_scale=1.2)

HAPPY_COLORS_PALETTE = ["#01BEFE", "#FFDD00", "#FF7D00", "#FF006D", "#ADFF02", "#8F0\
0FF"]

sns.set_palette(sns.color_palette(HAPPY_COLORS_PALETTE))

```

```
rcParams['figure.figsize'] = 12, 8

RANDOM_SEED = 42
np.random.seed(RANDOM_SEED)
torch.manual_seed(RANDOM_SEED)
```

Face Detection Data

Our dataset is provided by Dataturks[9], and it is hosted on Kaggle[10]. Here's an excerpt from the description:

> Faces in images marked with bounding boxes. Have around 500 images with around 1100 faces manually tagged via bounding box.

I've downloaded the JSON file containing the annotations and uploaded it to Google Drive. Let's get it:

```
!gdown --id 1K79wJgmPTWamqb04Op2GxW0SW9oxw8KS
```

Let's load the file into a Pandas dataframe:

```
faces_df = pd.read_json('face_detection.json', lines=True)
```

Each line contains a single face annotation. Note that multiple lines might point to a single image (e.g. multiple faces per image).

Data Preprocessing

The dataset contains only image URLs and annotations. We'll have to download the images. We'll also normalize the annotations, so it's easier to use them with Detectron2 later on:

[9] https://dataturks.com/
[10] https://www.kaggle.com/dataturks/face-detection-in-images

```
os.makedirs("faces", exist_ok=True)

dataset = []

for index, row in tqdm(faces_df.iterrows(), total=faces_df.shape[0]):
    img = urllib.request.urlopen(row["content"])
    img = Image.open(img)
    img = img.convert('RGB')

    image_name = f'face_{index}.jpeg'

    img.save(f'faces/{image_name}', "JPEG")

    annotations = row['annotation']
    for an in annotations:

      data = {}

      width = an['imageWidth']
      height = an['imageHeight']
      points = an['points']

      data['file_name'] = image_name
      data['width'] = width
      data['height'] = height

      data["x_min"] = int(round(points[0]["x"] * width))
      data["y_min"] = int(round(points[0]["y"] * height))
      data["x_max"] = int(round(points[1]["x"] * width))
      data["y_max"] = int(round(points[1]["y"] * height))

      data['class_name'] = 'face'

      dataset.append(data)
```

Let's put the data into a dataframe so we can have a better look:

```
df = pd.DataFrame(dataset)
```

```
print(df.file_name.unique().shape[0], df.shape[0])
```

```
409 1132
```

We have a total of 409 images (a lot less than the promised 500) and 1132 annotations. Let's save them to the disk (so you might reuse them):

```
df.to_csv('annotations.csv', header=True, index=None)
```

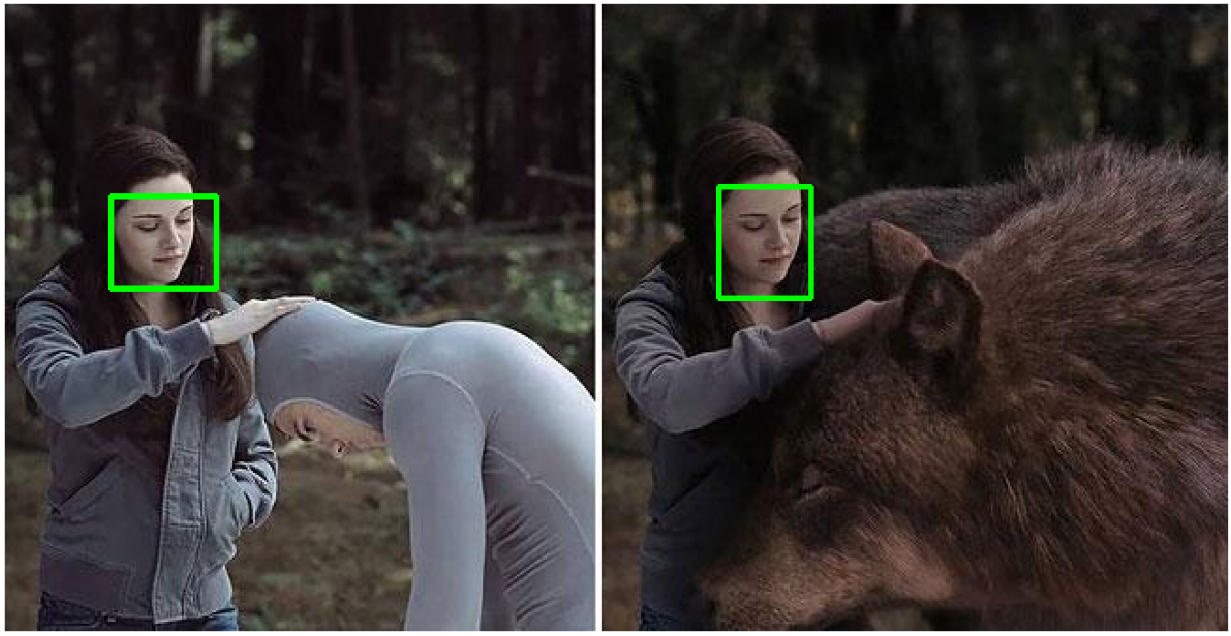
Data Exploration

Let's see some sample annotated data. We'll use OpenCV to load an image, add the bounding boxes, and resize it. We'll define a helper function to do it all:

```
def annotate_image(annotations, resize=True):
  file_name = annotations.file_name.to_numpy()[0]
  img = cv2.cvtColor(cv2.imread(f'faces/{file_name}'), cv2.COLOR_BGR2RGB)

  for i, a in annotations.iterrows():
    cv2.rectangle(img, (a.x_min, a.y_min), (a.x_max, a.y_max), (0, 255, 0), 2)

  if not resize:
    return img

  return cv2.resize(img, (384, 384), interpolation = cv2.INTER_AREA)
```

Let's start by showing some annotated images:

```
img_df = df[df.file_name == df.file_name.unique()[0]]
img = annotate_image(img_df, resize=False)

plt.imshow(img)
plt.axis('off');
```



png

```
img_df = df[df.file_name == df.file_name.unique()[1]]
img = annotate_image(img_df, resize=False)

plt.imshow(img)
plt.axis('off');
```

png

Those are good ones, the annotations are clearly visible. We can use torchvision to create a grid of images. Note that the images are in various sizes, so we'll resize them:

```
sample_images = [annotate_image(df[df.file_name == f]) for f in df.file_name.unique(\
)[:10]]
sample_images = torch.as_tensor(sample_images)
```

```
sample_images.shape
```

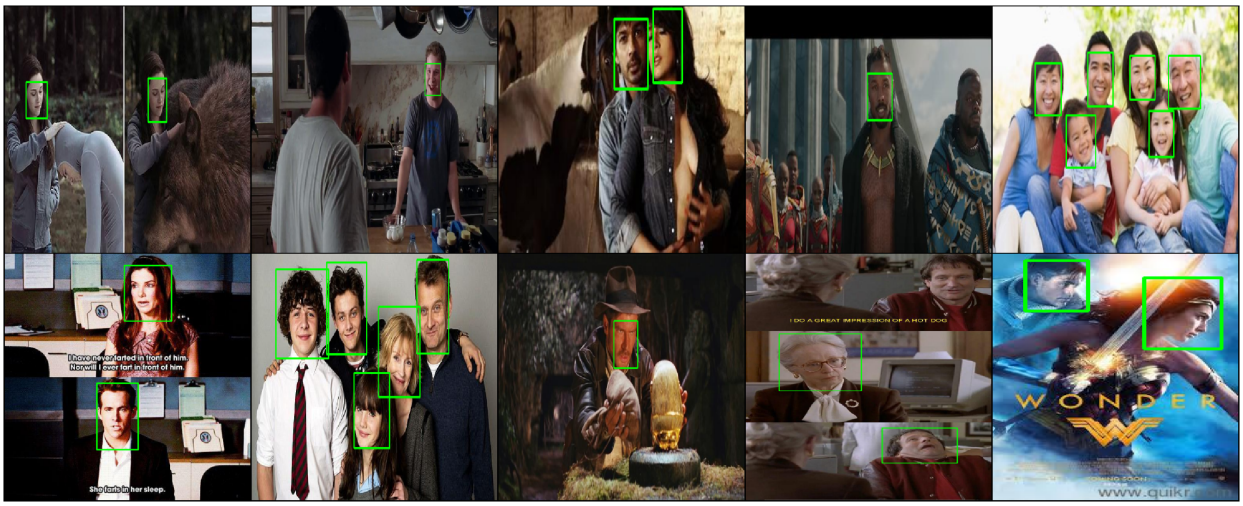
```
torch.Size([10, 384, 384, 3])
```

```
sample_images = sample_images.permute(0, 3, 1, 2)
```

```
sample_images.shape
```

```
torch.Size([10, 3, 384, 384])
```

```
plt.figure(figsize=(24, 12))
grid_img = torchvision.utils.make_grid(sample_images, nrow=5)

plt.imshow(grid_img.permute(1, 2, 0))
plt.axis('off');
```



png

You can clearly see that some annotations are missing (column 4). That's real life data for you, sometimes you have to deal with it in some way.

Face Detection with Detectron 2

It is time to go through the steps of fine-tuning a model using a custom dataset. But first, let's save 5% of the data for testing:

```
df = pd.read_csv('annotations.csv')

IMAGES_PATH = f'faces'

unique_files = df.file_name.unique()

train_files = set(np.random.choice(unique_files, int(len(unique_files) * 0.95), repl\
ace=False))
train_df = df[df.file_name.isin(train_files)]
test_df = df[~df.file_name.isin(train_files)]
```

The classical train_test_split won't work here, cause we want a split amongst the file names.

The next parts are written in a bit more generic way. Obviously, we have a single class - face. But adding more should be as simple as adding more annotations to the dataframe:

```
classes = df.class_name.unique().tolist()
```

Next, we'll write a function that converts our dataset into a format that is used by Detectron2:

```
def create_dataset_dicts(df, classes):
  dataset_dicts = []
  for image_id, img_name in enumerate(df.file_name.unique()):

    record = {}

    image_df = df[df.file_name == img_name]

    file_path = f'{IMAGES_PATH}/{img_name}'
    record["file_name"] = file_path
    record["image_id"] = image_id
    record["height"] = int(image_df.iloc[0].height)
    record["width"] = int(image_df.iloc[0].width)

    objs = []
    for _, row in image_df.iterrows():

      xmin = int(row.x_min)
      ymin = int(row.y_min)
      xmax = int(row.x_max)
      ymax = int(row.y_max)

      poly = [
          (xmin, ymin), (xmax, ymin),
          (xmax, ymax), (xmin, ymax)
      ]
      poly = list(itertools.chain.from_iterable(poly))

      obj = {
        "bbox": [xmin, ymin, xmax, ymax],
        "bbox_mode": BoxMode.XYXY_ABS,
        "segmentation": [poly],
        "category_id": classes.index(row.class_name),
        "iscrowd": 0
      }
      objs.append(obj)
```

```
37
38      record["annotations"] = objs
39      dataset_dicts.append(record)
40    return dataset_dicts
```

We convert every annotation row to a single record with a list of annotations. You might also notice that we're building a polygon that is of the exact same shape as the bounding box. This is required for the image segmentation models in Detectron2.

You'll have to register your dataset into the dataset and metadata catalogues:

```
1  for d in ["train", "val"]:
2    DatasetCatalog.register("faces_" + d, lambda d=d: create_dataset_dicts(train_df if\
3   d == "train" else test_df, classes))
4    MetadataCatalog.get("faces_" + d).set(thing_classes=classes)
5
6  statement_metadata = MetadataCatalog.get("faces_train")
```

Unfortunately, evaluator for the test set is not included by default. We can easily fix that by writing our own trainer:

```
1  class CocoTrainer(DefaultTrainer):
2
3    @classmethod
4    def build_evaluator(cls, cfg, dataset_name, output_folder=None):
5
6      if output_folder is None:
7          os.makedirs("coco_eval", exist_ok=True)
8          output_folder = "coco_eval"
9
10     return COCOEvaluator(dataset_name, cfg, False, output_folder)
```

The evaluation results will be stored in the `coco_eval` folder if no folder is provided.

Fine-tuning a Detectron2 model is nothing like writing PyTorch code. We'll load a configuration file, change a few values, and start the training process. But hey, it really helps if you know what you're doing

For this tutorial, we'll use the Mask R-CNN X101-FPN model. It is pre-trained on the COCO dataset[11] and achieves very good performance. The downside is that it is slow to train.

Let's load the config file and the pre-trained model weights:

[11] http://cocodataset.org/#home

```
cfg = get_cfg()

cfg.merge_from_file(
  model_zoo.get_config_file(
    "COCO-InstanceSegmentation/mask_rcnn_X_101_32x8d_FPN_3x.yaml"
  )
)

cfg.MODEL.WEIGHTS = model_zoo.get_checkpoint_url(
  "COCO-InstanceSegmentation/mask_rcnn_X_101_32x8d_FPN_3x.yaml"
)
```

Specify the datasets (we registered those) we'll use for training and evaluation:

```
cfg.DATASETS.TRAIN = ("faces_train",)
cfg.DATASETS.TEST = ("faces_val",)
cfg.DATALOADER.NUM_WORKERS = 4
```

And for the optimizer, we'll do a bit of magic to converge to something nice:

```
cfg.SOLVER.IMS_PER_BATCH = 4
cfg.SOLVER.BASE_LR = 0.001
cfg.SOLVER.WARMUP_ITERS = 1000
cfg.SOLVER.MAX_ITER = 1500
cfg.SOLVER.STEPS = (1000, 1500)
cfg.SOLVER.GAMMA = 0.05
```

Except for the standard stuff (batch size, max number of iterations, and learning rate) we have a couple of interesting params:

- `WARMUP_ITERS` - the learning rate starts from `0` and goes to the preset one for this number of iterations
- `STEPS` - the checkpoints (number of iterations) at which the learning rate will be reduced by `GAMMA`

Finally, we'll specify the number of classes and the period at which we'll evaluate on the test set:

```
cfg.MODEL.ROI_HEADS.BATCH_SIZE_PER_IMAGE = 64
cfg.MODEL.ROI_HEADS.NUM_CLASSES = len(classes)

cfg.TEST.EVAL_PERIOD = 500
```

Time to train, using our custom trainer:

```
os.makedirs(cfg.OUTPUT_DIR, exist_ok=True)

trainer = CocoTrainer(cfg)
trainer.resume_or_load(resume=False)
trainer.train()
```

Evaluating Object Detection Models

Evaluating object detection models is a bit different when compared to evaluating standard classification or regression models.

The main metric you need to know about is IoU (intersection over union). It measures the overlap between two boundaries - the predicted and ground truth one. It can get values between 0 and 1.

$$\text{IoU} = \frac{\text{area of overlap}}{\text{area of union}}$$

Using IoU, one can define a threshold (e.g. >0.5) to classify whether a prediction is a true positive (TP) or a false positive (FP).

Now you can calculate average precision (AP) by taking the area under the precision-recall curve.

Now AP@X (e.g. AP50) is just AP at some IoU threshold. This should give you a working understanding of how object detection models are evaluated.

I suggest you read the mAP (mean Average Precision) for Object Detection[12] tutorial by Jonathan Hui if you want to learn more on the topic.

I've prepared a pre-trained model for you, so you don't have to wait for the training to complete. Let's download it:

```
!gdown --id 18Ev2bpdKsBaDufhVKf0cT6RmM3FjW3nL
!mv face_detector.pth output/model_final.pth
```

We can start making predictions by loading the model and setting a minimum threshold of 85% certainty at which we'll consider the predictions as correct:

[12] https://medium.com/@jonathan_hui/map-mean-average-precision-for-object-detection-45c121a31173

```
cfg.MODEL.WEIGHTS = os.path.join(cfg.OUTPUT_DIR, "model_final.pth")
cfg.MODEL.ROI_HEADS.SCORE_THRESH_TEST = 0.85
predictor = DefaultPredictor(cfg)
```

Let's run the evaluator with the trained model:

```
evaluator = COCOEvaluator("faces_val", cfg, False, output_dir="./output/")
val_loader = build_detection_test_loader(cfg, "faces_val")
inference_on_dataset(trainer.model, val_loader, evaluator)
```

Finding Faces in Images

Next, let's create a folder and save all images with predicted annotations in the test set:

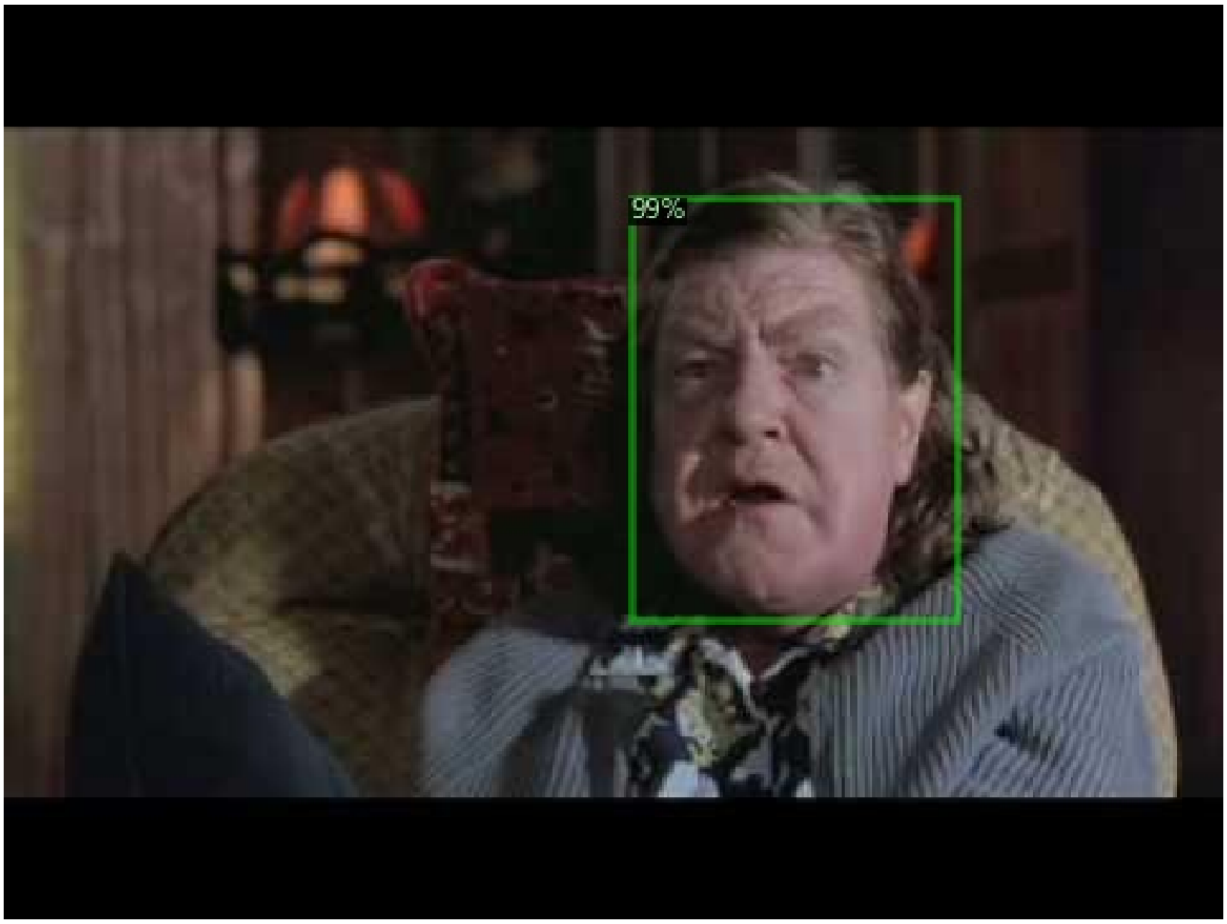
```
os.makedirs("annotated_results", exist_ok=True)

test_image_paths = test_df.file_name.unique()
```

```
for clothing_image in test_image_paths:
  file_path = f'{IMAGES_PATH}/{clothing_image}'
  im = cv2.imread(file_path)
  outputs = predictor(im)
  v = Visualizer(
    im[:, :, ::-1],
    metadata=statement_metadata,
    scale=1.,
    instance_mode=ColorMode.IMAGE
  )
  instances = outputs["instances"].to("cpu")
  instances.remove('pred_masks')
  v = v.draw_instance_predictions(instances)
  result = v.get_image()[:, :, ::-1]
  file_name = ntpath.basename(clothing_image)
  write_res = cv2.imwrite(f'annotated_results/{file_name}', result)
```

Let's have a look:

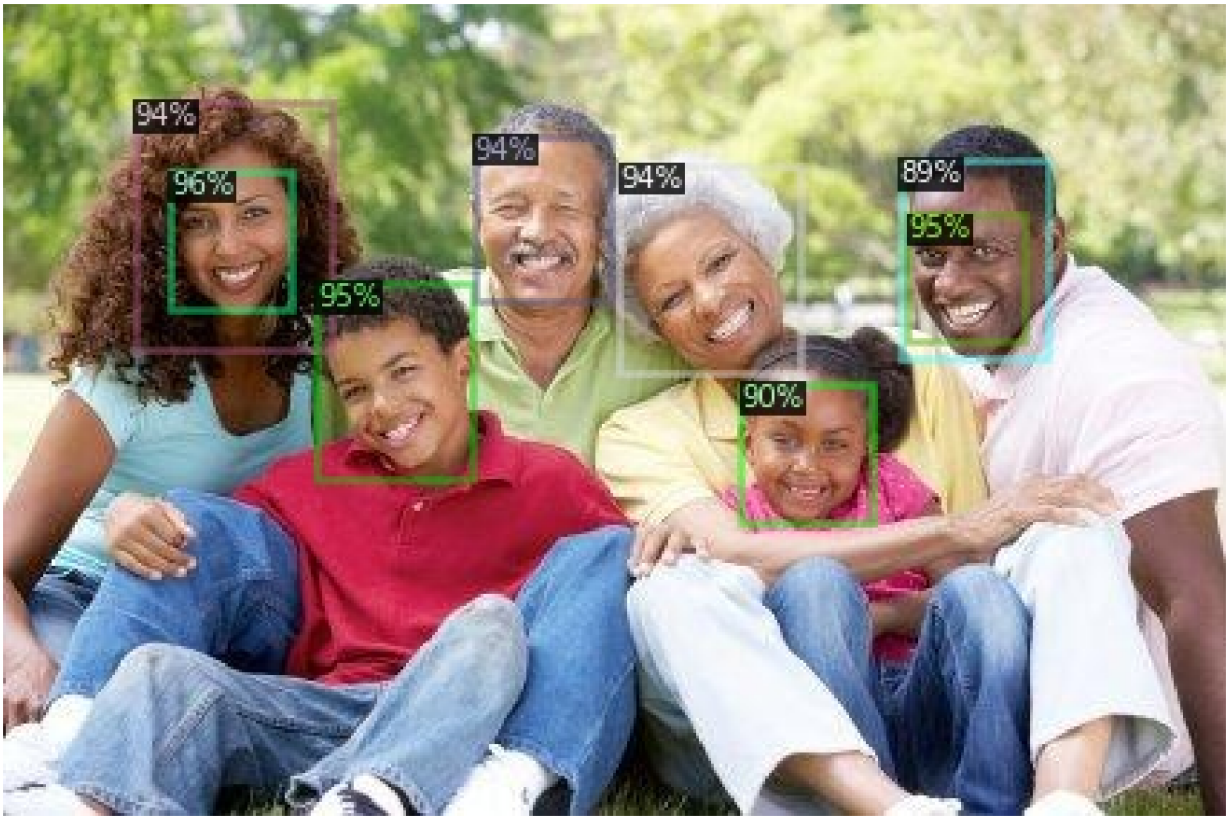
```
annotated_images = [f'annotated_results/{f}' for f in test_df.file_name.unique()]
```

```
img = cv2.cvtColor(cv2.imread(annotated_images[0]), cv2.COLOR_BGR2RGB)

plt.imshow(img)
plt.axis('off');
```



png

```
img = cv2.cvtColor(cv2.imread(annotated_images[1]), cv2.COLOR_BGR2RGB)

plt.imshow(img)
plt.axis('off');
```



png

```
img = cv2.cvtColor(cv2.imread(annotated_images[3]), cv2.COLOR_BGR2RGB)

plt.imshow(img)
plt.axis('off');
```

png

```
img = cv2.cvtColor(cv2.imread(annotated_images[4]), cv2.COLOR_BGR2RGB)

plt.imshow(img)
plt.axis('off');
```

png

Not bad. Not bad at all. I suggest you explore more images on your own, too!

Note that some faces have multiple bounding boxes (on the second image) with different degrees of certainty. Maybe training the model longer will help? How about adding more or augmenting the existing data?

Conclusion

Congratulations! You now know the basics of Detectron2 for object detection! You might be surprised by the results, given the small dataset we have. That's the power of large pre-trained models for you

- Run the complete notebook in your browser (Google Colab)[13]
- Read the **Getting Things Done with Pytorch** book[14]

You learned how to:

- prepare a custom dataset for face detection with Detectron2
- use (close to) state-of-the-art models for object detection to find faces in images
- You can extend this work for face recognition.

[13] https://colab.research.google.com/drive/1Jk4-qX9zdYGsBrTnh2vF52CV9ucuqpjk
[14] https://github.com/curiousily/Getting-Things-Done-with-Pytorch

References

- Face Detection in Images[15]
- Detectron2 on GitHub[16]
- mAP (mean Average Precision) for Object Detection[17]

[15] https://www.kaggle.com/dataturks/face-detection-in-images
[16] https://github.com/facebookresearch/detectron2/
[17] https://medium.com/@jonathan_hui/map-mean-average-precision-for-object-detection-45c121a31173

7. Create Dataset for Sentiment Analysis by Scraping Google Play App Reviews

> TL;DR In this tutorial, you'll learn how to create a dataset for Sentiment Analysis by scraping user reviews for Android apps. You'll convert the app and review information into Data Frames and save that to CSV files.

- Run the notebook in your browser (Google Colab)[1]
- Read the `Getting Things Done with Pytorch` book[2]

You'll learn how to:

- Set a goal and inclusion criteria for your dataset
- Get real-world user reviews by scraping Google Play
- Use Pandas to convert and save the dataset into CSV files

Setup

Let's install the required packages and setup the imports:

```
%watermark -v -p pandas,matplotlib,seaborn,google_play_scraper
```

```
CPython 3.6.9
IPython 5.5.0

pandas 1.0.3
matplotlib 3.2.1
seaborn 0.10.0
google_play_scraper 0.0.2.3
```

[1] https://colab.research.google.com/drive/1GDJIpz7BXw55jl9wTOMQDool9m8DIOyp
[2] https://github.com/curiousily/Getting-Things-Done-with-Pytorch

```
import json
import pandas as pd
from tqdm import tqdm

import seaborn as sns
import matplotlib.pyplot as plt

from pygments import highlight
from pygments.lexers import JsonLexer
from pygments.formatters import TerminalFormatter

from google_play_scraper import Sort, reviews, app

%matplotlib inline
%config InlineBackend.figure_format='retina'

sns.set(style='whitegrid', palette='muted', font_scale=1.2)
```

The Goal of the Dataset

You want to get feedback for your app. Both negative and positive are good. But the negative one can reveal critical features that are missing or downtime of your service (when it is much more frequent).

Lucky for us, Google Play has plenty of apps, reviews, and scores. We can scrape app info and reviews using the google-play-scraper[3] package.

You can choose plenty of apps to analyze. But different app categories contain different audiences, domain-specific quirks, and more. We'll start simple.

We want apps that have been around some time, so opinion is collected organically. We want to mitigate advertising strategies as much as possible. Apps are constantly being updated, so the time of the review is an important factor.

Ideally, you would want to collect every possible review and work with that. However, in the real world data is often limited (too large, inaccessible, etc). So, we'll do the best we can.

Let's choose some apps that fit the criteria from the *Productivity* category. We'll use AppAnnie[4] to select some of the top US apps:

[3] https://github.com/JoMingyu/google-play-scraper

[4] https://www.appannie.com/apps/google-play/top-chart/?country=US&category=29&device=&date=2020-04-05&feed=All&rank_sorting_type=rank&page_number=1&page_size=100&table_selections=

```
app_packages = [
  'com.anydo',
  'com.todoist',
  'com.ticktick.task',
  'com.habitrpg.android.habitica',
  'cc.forestapp',
  'com.oristats.habitbull',
  'com.levor.liferpgtasks',
  'com.habitnow',
  'com.microsoft.todos',
  'prox.lab.calclock',
  'com.gmail.jmartindev.timetune',
  'com.artfulagenda.app',
  'com.tasks.android',
  'com.appgenix.bizcal',
  'com.appxy.planner'
]
```

Scraping App Information

Let's scrape the info for each app:

```
app_infos = []

for ap in tqdm(app_packages):
  info = app(ap, lang='en', country='us')
  del info['comments']
  app_infos.append(info)
```

```
100%|□□□□□□□□□□□□| 15/15 [00:02<00:00,  6.34it/s]
```

We got the info for all 15 apps. Let's write a helper function that prints JSON objects a bit better:

```
def print_json(json_object):
  json_str = json.dumps(
    json_object,
    indent=2,
    sort_keys=True,
    default=str
  )
  print(highlight(json_str, JsonLexer(), TerminalFormatter()))
```

Here is a sample app information from the list:

```
print_json(app_infos[0])
```

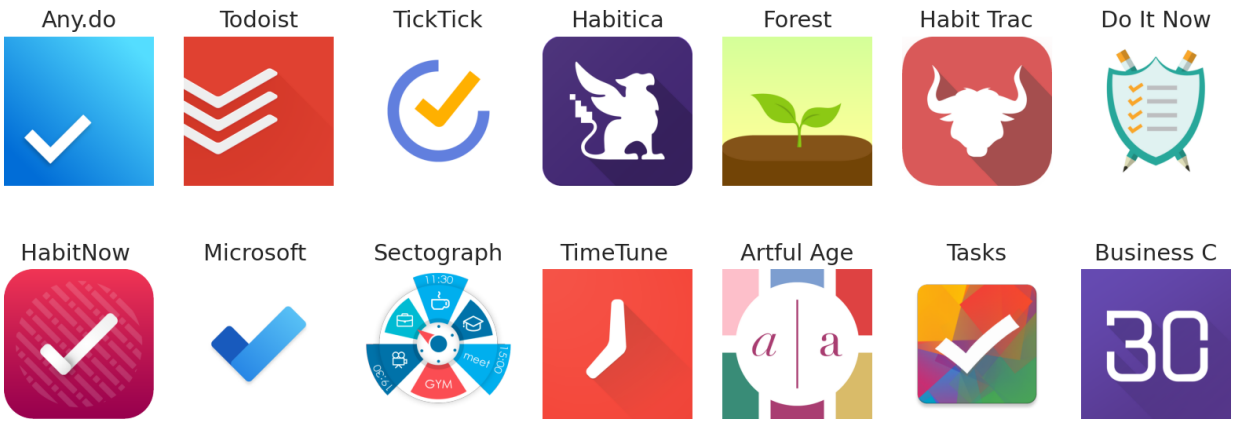

```
{
  "adSupported": null,
  "androidVersion": "Varies",
  "androidVersionText": "Varies with device",
  "appId": "com.anydo",
  "containsAds": null,
  "contentRating": "Everyone",
  "contentRatingDescription": null,
  "currency": "USD",
  "description": "<b>\ud83c\udfc6 Editor's Choice by Google</b>\r\n\r\nAny.do is a T\
o Do List, Calendar, Planner...",
  "descriptionHTML": "<b>\ud83c\udfc6 Editor&#39;s Choice by Google</b><br><br>Any.d\
o is a To Do List, Calendar, Planner...",
  "developer": "Any.do Calendar & To-Do List",
  "developerAddress": "Any.do Inc.\n\n6  Agripas Street, Tel Aviv\n6249106 ISRAEL",
  "developerEmail": "feedback+androidtodo@any.do",
  "developerId": "5304780265295461149",
  "developerInternalID": "5304780265295461149",
  "developerWebsite": "https://www.any.do",
  "free": true,
  "genre": "Productivity",
  "genreId": "PRODUCTIVITY",
  "headerImage": "https://lh3.googleusercontent.com/dZknnlk1LM8fYS3wjOvVHOmWKOGH1HAe\
691Yuh7LAeBj6a730A1CQqZnXxjNahAYUFFw",
  "histogram": [27291, 9246, 13735, 29904, 262997],
  "icon": "https://lh3.googleusercontent.com/zgOLUXCHkF91H8xuMTMLT17smwgLPwSBjUlKVWF\
-cZRFjlv-Uvtman7DiHEii54fbEE",
  "installs": "10,000,000+",
  "minInstalls": 10000000,
```

```
  "offersIAP": true,
  "price": 0,
  "privacyPolicy": "https://www.any.do/privacy",
  "ratings": 343174,
  "recentChanges": "Faster and smoother for better user experience!",
  "recentChangesHTML": "Faster and smoother for better user experience!",
  "released": "Nov 10, 2011",
  "reviews": 122170,
  "score": 4.43388,
  "screenshots": [
    "https://lh3.googleusercontent.com/C-L3_FPMlKVrZItAORaszhnQzlzMyXcqF_-oGaabHm_On\
wUW1jz02BXBVSKi0HRUtQ",
    "https://lh3.googleusercontent.com/uAP6G5ANQcgVs4Uj6yrcsAo4OUhejTJRVCXOxnAVA5Efi\
t_OtAnrOYyL1SUHj1rv",
    "https://lh3.googleusercontent.com/AI5mLFu0Atsl0km2FO9_IwJXNy_1q1_X6Ua3EVMZNedp0\
dsDToDRaWQ1UDvI6mb1-I0",
    "https://lh3.googleusercontent.com/bYCAn3mjgB4ugSY0PL-PCcMBfbvXCSFkzL-pLSIIbZ8sQ\
ByQPerHboPQ2fA126K4LDtU",
    "https://lh3.googleusercontent.com/u-dX4lpTepsvXs33ds4xxYpApuGS4JBAEb0UsvY_fPbpt\
xnF0QxaKNW0-tJVXaP8a1E",
    "https://lh3.googleusercontent.com/qvUz_9IXHQd6FSLUALZo8NKLx-s4uDGyElPOGRsU28TCE\
ficQc0BoNRloRRLqUkH2A",
    "https://lh3.googleusercontent.com/tEyGs6MGlY97ccLc4c_HxV9xNOpsvwQyHz6uGAezkVtxm\
1ydAaTj5EZSUgqlg69qrrk",
    "https://lh3.googleusercontent.com/StN0i2BskOs6HCfaPO0DMBOCQMCag3okWVI_SlFJtMytw\
bgNMBnD5i9hbSqdNlGxffmn",
    "https://lh3.googleusercontent.com/GRKqWfo-PLzCKwpgZ8fej4PGsUp1q9eM5a3LQeiYCOW-K\
UpCOIHXOp3mteZWbJ-pz4My",
    "https://lh3.googleusercontent.com/pFQQ_qi8u92duWCNXpEcNKpH2lVpD_hFd5f-UlTP_f6wf\
t3YyYLMzwLitxt-UI6G8vs",
    "https://lh3.googleusercontent.com/AoeCU6bT1x0eHRvJwvQyOSKJ31oSayox959qMNVaSzz3u\
N9bvk1cGek5zyRDe1BdtA",
    "https://lh3.googleusercontent.com/vICme1f4J9vFt8wY3xBY-LshGgYyvSbsa4TLJyEtNsy0a\
lUI0i9oMQVq8oJ4l_yR1Aw",
    "https://lh3.googleusercontent.com/7sn9m__iVM-peiG6_jkKBuE-QVH_xDaycF_oR1XJlwcAC\
45ybNZ_Exor09ENOJ41Q2U",
    "https://lh3.googleusercontent.com/9I_m2ZXgPtiU4Po4cw_cyIaEpZxynxQ1n3YkhFgakATfb\
u63a8_f8vGQDxKOHYITzew"
  ],
  "size": "Varies with device",
  "summary": "Task Manager \u2705 Organizer \ud83d\udcc5 Agenda \ud83d\udcdd Daily R\
eminders \ud83d\udd14 All-in-One Simple App.",
  "summaryHTML": "Task Manager \u2705 Organizer \ud83d\udcc5 Agenda \ud83d\udcdd Dai\
```

```
ly Reminders \ud83d\udd14 All-in-One Simple App.",
  "title": "Any.do: To do list, Calendar, Planner & Reminders",
  "updated": 1586258773,
  "url": "https://play.google.com/store/apps/details?id=com.anydo&hl=en&gl=us",
  "version": "Varies with device",
  "video": "https://www.youtube.com/embed/2nkllLD0x6o?ps=play&vq=large&rel=0&autohid\
e=1&showinfo=0",
  "videoImage": "https://i.ytimg.com/vi/2nkllLD0x6o/hqdefault.jpg"
}
```

This contains lots of information including the number of ratings, number of reviews and number of ratings for each score (1 to 5). Let's ignore all of that and have a look at their beautiful icons:

```
def format_title(title):
  sep_index = title.find(':') if title.find(':') != -1 else title.find('-')
  if sep_index != -1:
    title = title[:sep_index]
  return title[:10]

fig, axs = plt.subplots(2, len(app_infos) // 2, figsize=(14, 5))

for i, ax in enumerate(axs.flat):
  ai = app_infos[i]
  img = plt.imread(ai['icon'])
  ax.imshow(img)
  ax.set_title(format_title(ai['title']))
  ax.axis('off')
```



png

We'll store the app information for later by converting the JSON objects into a Pandas dataframe and saving the result into a CSV file:

```
app_infos_df = pd.DataFrame(app_infos)
app_infos_df.to_csv('apps.csv', index=None, header=True)
```

Scraping App Reviews

In an ideal world, we would get all the reviews. But there are lots of them and we're scraping the data. That wouldn't be very polite. What should we do?

We want:

- Balanced dataset - roughly the same number of reviews for each score (1-5)
- A representative sample of the reviews for each app

We can satisfy the first requirement by using the scraping package option to filter the review score. For the second, we'll sort the reviews by their helpfulness, which are the reviews that Google Play thinks are most important. Just in case, we'll get a subset from the newest, too:

```
app_reviews = []

for ap in tqdm(app_packages):
  for score in list(range(1, 6)):
    for sort_order in [Sort.MOST_RELEVANT, Sort.NEWEST]:
      rvs, _ = reviews(
        ap,
        lang='en',
        country='us',
        sort=sort_order,
        count= 200 if score == 3 else 100,
        filter_score_with=score
      )
      for r in rvs:
        r['sortOrder'] = 'most_relevant' if sort_order == Sort.MOST_RELEVANT else 'n\
ewest'
        r['appId'] = ap
      app_reviews.extend(rvs)
```

```
100%|□□□□□□□□□□□□| 15/15 [00:45<00:00,  3.01s/it]
```

Note that we're adding the app id and sort order to each review. Here's an example for one:

```
print_json(app_reviews[0])
```


```
{
  "appId": "com.anydo",
  "at": "2020-04-05 22:25:57",
  "content": "Update: After getting a response from the developer I would change my \
rating to 0 stars if possible. These guys hide behind confusing and opaque terms and\
 refuse to budge at all. I'm so annoyed that my money has been lost to them! Really \
terrible customer experience. Original: Be very careful when signing up for a free t\
rial of this app. If you happen to go over they automatically charge you for a full \
years subscription and refuse to refund. Terrible customer experience and the app is\
 just OK.",
  "repliedAt": "2020-04-07 14:09:03",
  "replyContent": "Our policy and TOS are completely transparent and can be found in\
 the Help Center and our main page. In addition, a payment can only be made upon the\
 user's authorization via the app and Google Play. We provide users with a full 7 da\
ys trial to test the app with an additional 48 hours for a refund, along with priori\
ty support for all issues.",
  "reviewCreatedVersion": "4.17.0.3",
  "score": 1,
  "sortOrder": "most_relevant",
  "thumbsUpCount": 37,
  "userImage": "https://lh3.googleusercontent.com/a-/AOh14GiHdfNEu1DwwcJ6yNyju8Yvn4J\
wjpzuXvD74aVmDA",
  "userName": "Andrew Thomas"
}
```


`repliedAt` and `replyContent` contain the developer response to the review. Of course, they can be missing.

How many app reviews did we get?

```
len(app_reviews)
```

```
15750
```

Let's save the reviews to a CSV file:

```
app_reviews_df = pd.DataFrame(app_reviews)
app_reviews_df.to_csv('reviews.csv', index=None, header=True)
```

Summary

Well done! You now have a dataset with more than 15k user reviews from 15 productivity apps. Of course, you can go crazy and get much much more.

- Run the notebook in your browser (Google Colab)[5]
- Read the `Getting Things Done with Pytorch` book[6]

You learned how to:

- Set goals and expectations for your dataset
- Scrape Google Play app information
- Scrape user reviews for Google Play apps
- Save the dataset to CSV files

Next, we're going to use the reviews for sentiment analysis with BERT. But first, we'll have to do some text preprocessing!

References

- Google Play Scraper for Python[7]

[5] https://colab.research.google.com/drive/1GDJIpz7BXw55jl9wTOMQDool9m8DIOyp
[6] https://github.com/curiousily/Getting-Things-Done-with-Pytorch
[7] https://github.com/JoMingyu/google-play-scraper

8. Sentiment Analysis with BERT and Transformers by Hugging Face

> TL;DR In this tutorial, you'll learn how to fine-tune BERT for sentiment analysis. You'll do the required text preprocessing (special tokens, padding, and attention masks) and build a Sentiment Classifier using the amazing Transformers library by Hugging Face!

- Run the notebook in your browser (Google Colab)[1]
- Read the `Getting Things Done with Pytorch` book[2]

You'll learn how to:

- Intuitively understand what BERT is
- Preprocess text data for BERT and build PyTorch Dataset (tokenization, attention masks, and padding)
- Use Transfer Learning to build Sentiment Classifier using the Transformers library by Hugging Face
- Evaluate the model on test data
- Predict sentiment on raw text

Let's get started!

What is BERT?

BERT (introduced in this paper[3]) stands for Bidirectional Encoder Representations from Transformers. If you don't know what most of that means - you've come to the right place! Let's unpack the main ideas:

- Bidirectional - to understand the text you're looking you'll have to look back (at the previous words) and forward (at the next words)
- Transformers - The Attention Is All You Need[4] paper presented the Transformer model. The Transformer reads entire sequences of tokens at once. In a sense, the model is non-directional, while LSTMs read sequentially (left-to-right or right-to-left). The attention mechanism allows for learning contextual relations between words (e.g. `his` in a sentence refers to Jim).

[1] https://colab.research.google.com/drive/1PHv-IRLPCtv7oTcIGbsgZHqrB5LPvB7S
[2] https://github.com/curiousily/Getting-Things-Done-with-Pytorch
[3] https://arxiv.org/abs/1810.04805
[4] https://arxiv.org/abs/1706.03762

- (Pre-trained) contextualized word embeddings - The ELMO paper[5] introduced a way to encode words based on their meaning/context. Nails has multiple meanings - fingernails and metal nails.

BERT was trained by masking 15% of the tokens with the goal to guess them. An additional objective was to predict the next sentence. Let's look at examples of these tasks:

Masked Language Modeling (Masked LM)

The objective of this task is to guess the masked tokens. Let's look at an example, and try to not make it harder than it has to be:

That's `[mask]` she `[mask]` -> That's what she said

Next Sentence Prediction (NSP)

Given a pair of two sentences, the task is to say whether or not the second follows the first (binary classification). Let's continue with the example:

Input = `[CLS]` That's `[mask]` she `[mask]`. [SEP] Hahaha, nice! [SEP]

Label = *IsNext*

Input = `[CLS]` That's `[mask]` she `[mask]`. [SEP] Dwight, you ignorant `[mask]`! [SEP]

Label = *NotNext*

The training corpus was comprised of two entries: Toronto Book Corpus[6] (800M words) and English Wikipedia (2,500M words). While the original Transformer has an encoder (for reading the input) and a decoder (that makes the prediction), BERT uses only the decoder.

BERT is simply a pre-trained stack of Transformer Encoders. How many Encoders? We have two versions - with 12 (BERT base) and 24 (BERT Large).

Is This Thing Useful in Practice?

The BERT paper was released along with the source code[7] and pre-trained models.

The best part is that you can do Transfer Learning (thanks to the ideas from OpenAI Transformer) with BERT for many NLP tasks - Classification, Question Answering, Entity Recognition, etc. You can train with small amounts of data and achieve great performance!

[5] https://arxiv.org/abs/1802.05365v2
[6] https://arxiv.org/abs/1506.06724
[7] https://github.com/google-research/bert

Setup

We'll need the Transformers library[8] by Hugging Face:

```
!pip install -qq transformers
```

```
%reload_ext watermark
%watermark -v -p numpy,pandas,torch,transformers
```

```
CPython 3.6.9
IPython 5.5.0

numpy 1.18.2
pandas 1.0.3
torch 1.4.0
transformers 2.8.0
```

```
import transformers
from transformers import BertModel, BertTokenizer, AdamW, get_linear_schedule_with_w\
armup
import torch

import numpy as np
import pandas as pd
import seaborn as sns
from pylab import rcParams
import matplotlib.pyplot as plt
from matplotlib import rc
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, classification_report
from collections import defaultdict
from textwrap import wrap

from torch import nn, optim
from torch.utils.data import Dataset, DataLoader

%matplotlib inline
%config InlineBackend.figure_format='retina'
```

[8] https://huggingface.co/transformers/

```
22
23 sns.set(style='whitegrid', palette='muted', font_scale=1.2)
24
25 HAPPY_COLORS_PALETTE = ["#01BEFE", "#FFDD00", "#FF7D00", "#FF006D", "#ADFF02", "#8F0\
26 0FF"]
27
28 sns.set_palette(sns.color_palette(HAPPY_COLORS_PALETTE))
29
30 rcParams['figure.figsize'] = 12, 8
31
32 RANDOM_SEED = 42
33 np.random.seed(RANDOM_SEED)
34 torch.manual_seed(RANDOM_SEED)
35 device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

Data Exploration

We'll load the Google Play app reviews dataset, that we've put together in the previous part:

```
1 !gdown --id 1S6qMioqPJjyBLpLVz4gmRTnJHnjitnuV
2 !gdown --id 1zdmewp7ayS4js4VtrJEHzAheSW-5NBZv
```

```
1 df = pd.read_csv("reviews.csv")
2 df.shape
```

```
1 (15746, 11)
```

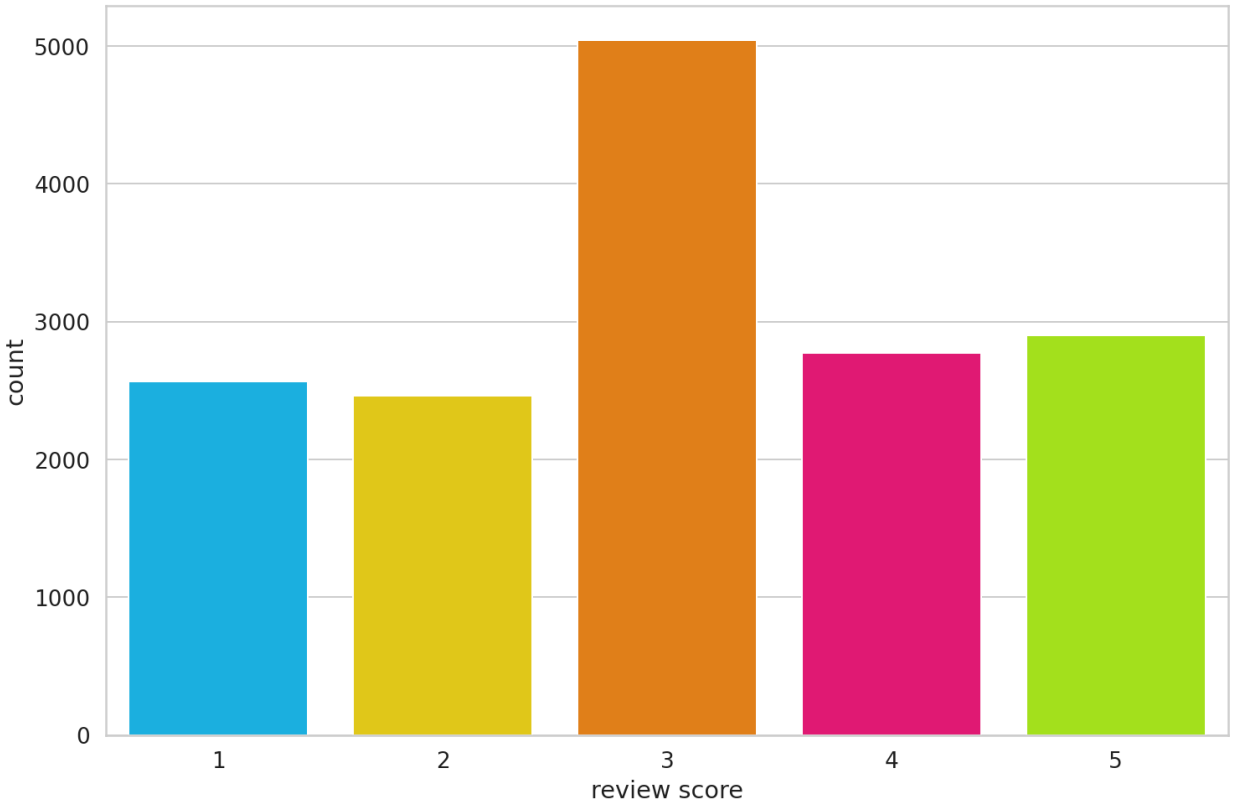
We have about 16k examples. Let's check for missing values:

```
1 df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 15746 entries, 0 to 15745
Data columns (total 11 columns):
 #   Column                Non-Null Count  Dtype 
---  ------                --------------  ----- 
 0   userName              15746 non-null  object
 1   userImage             15746 non-null  object
 2   content               15746 non-null  object
 3   score                 15746 non-null  int64 
 4   thumbsUpCount         15746 non-null  int64 
 5   reviewCreatedVersion  13533 non-null  object
 6   at                    15746 non-null  object
 7   replyContent          7367 non-null   object
 8   repliedAt             7367 non-null   object
 9   sortOrder             15746 non-null  object
 10  appId                 15746 non-null  object
dtypes: int64(2), object(9)
memory usage: 1.3+ MB
```

Great, no missing values in the score and review texts! Do we have class imbalance?

```
sns.countplot(df.score)
plt.xlabel('review score');
```



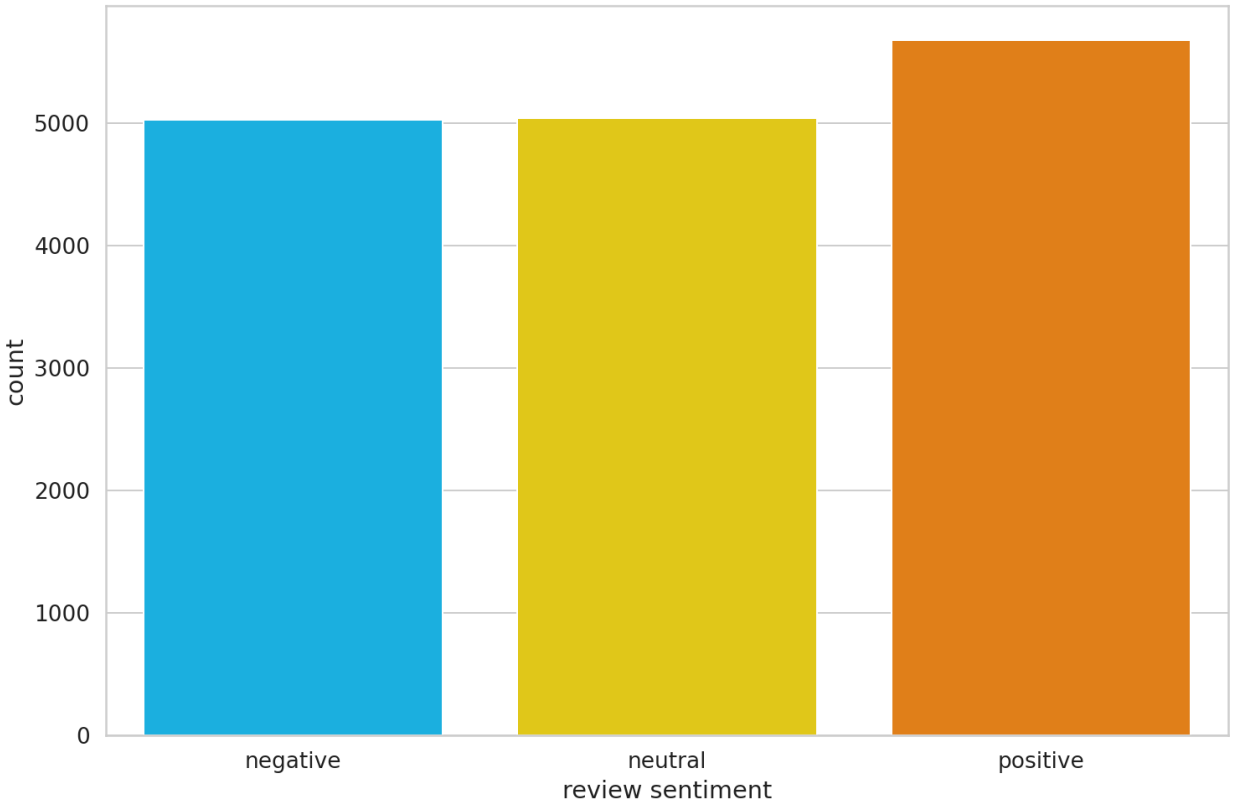
png

That's hugely imbalanced, but it's okay. We're going to convert the dataset into negative, neutral and positive sentiment:

```
def to_sentiment(rating):
  rating = int(rating)
  if rating <= 2:
    return 0
  elif rating == 3:
    return 1
  else:
    return 2

df['sentiment'] = df.score.apply(to_sentiment)
```

```
class_names = ['negative', 'neutral', 'positive']
```

```
ax = sns.countplot(df.sentiment)
plt.xlabel('review sentiment')
ax.set_xticklabels(class_names);
```



png

The balance was (mostly) restored.

Data Preprocessing

You might already know that Machine Learning models don't work with raw text. You need to convert text to numbers (of some sort). BERT requires even more attention (good one, right?). Here are the requirements:

- Add special tokens to separate sentences and do classification
- Pass sequences of constant length (introduce padding)
- Create array of 0s (pad token) and 1s (real token) called *attention mask*

The Transformers library provides (you've guessed it) a wide variety of Transformer models (including BERT). It works with TensorFlow and PyTorch! It also includes prebuild tokenizers that do the heavy lifting for us!

```
PRE_TRAINED_MODEL_NAME = 'bert-base-cased'
```

> You can use a cased and uncased version of BERT and tokenizer. I've experimented with both. The cased version works better. Intuitively, that makes sense, since "BAD" might convey more sentiment than "bad".

Let's load a pre-trained BertTokenizer[9]:

```
tokenizer = BertTokenizer.from_pretrained(PRE_TRAINED_MODEL_NAME)
```

We'll use this text to understand the tokenization process:

```
sample_txt = 'When was I last outside? I am stuck at home for 2 weeks.'
```

Some basic operations can convert the text to tokens and tokens to unique integers (ids):

```
tokens = tokenizer.tokenize(sample_txt)
token_ids = tokenizer.convert_tokens_to_ids(tokens)

print(f' Sentence: {sample_txt}')
print(f'   Tokens: {tokens}')
print(f'Token IDs: {token_ids}')
```

```
 Sentence: When was I last outside? I am stuck at home for 2 weeks.
   Tokens: ['When', 'was', 'I', 'last', 'outside', '?', 'I', 'am', 'stuck', 'at', 'h\
ome', 'for', '2', 'weeks', '.']
Token IDs: [1332, 1108, 146, 1314, 1796, 136, 146, 1821, 5342, 1120, 1313, 1111, 123\
, 2277, 119]
```

Special Tokens

`[SEP]` - marker for ending of a sentence

```
tokenizer.sep_token, tokenizer.sep_token_id
```

[9] https://huggingface.co/transformers/model_doc/bert.html#berttokenizer

```
('[SEP]', 102)
```

`[CLS]` - we must add this token to the start of each sentence, so BERT knows we're doing classification

```
tokenizer.cls_token, tokenizer.cls_token_id
```

```
('[CLS]', 101)
```

There is also a special token for padding:

```
tokenizer.pad_token, tokenizer.pad_token_id
```

```
('[PAD]', 0)
```

BERT understands tokens that were in the training set. Everything else can be encoded using the `[UNK]` (unknown) token:

```
tokenizer.unk_token, tokenizer.unk_token_id
```

```
('[UNK]', 100)
```

All of that work can be done using the `encode_plus()`[10] method:

```
encoding = tokenizer.encode_plus(
  sample_txt,
  max_length=32,
  add_special_tokens=True, # Add '[CLS]' and '[SEP]'
  return_token_type_ids=False,
  pad_to_max_length=True,
  return_attention_mask=True,
  return_tensors='pt',  # Return PyTorch tensors
)

encoding.keys()
```

[10] https://huggingface.co/transformers/main_classes/tokenizer.html#transformers.PreTrainedTokenizer.encode_plus

```
dict_keys(['input_ids', 'attention_mask'])
```

The token ids are now stored in a Tensor and padded to a length of 32:

```
print(len(encoding['input_ids'][0]))
encoding['input_ids'][0]
```

```
32
tensor([ 101, 1332, 1108,  146, 1314, 1796,  136,  146, 1821, 5342, 1120, 1313,
        1111,  123, 2277,  119,  102,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0])
```

The attention mask has the same length:

```
print(len(encoding['attention_mask'][0]))
encoding['attention_mask']
```

```
32
tensor([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0]])
```

We can inverse the tokenization to have a look at the special tokens:

```
tokenizer.convert_ids_to_tokens(encoding['input_ids'][0])
```

```
['[CLS]',
 'When',
 'was',
 'I',
 'last',
 'outside',
 '?',
 'I',
 'am',
 'stuck',
 'at',
 'home',
 'for',
 '2',
 'weeks',
```

```
 '.',
 '[SEP]',
 '[PAD]',
 '[PAD]',
 '[PAD]',
 '[PAD]',
 '[PAD]',
 '[PAD]',
 '[PAD]',
 '[PAD]',
 '[PAD]',
 '[PAD]',
 '[PAD]',
 '[PAD]',
 '[PAD]',
 '[PAD]']
```

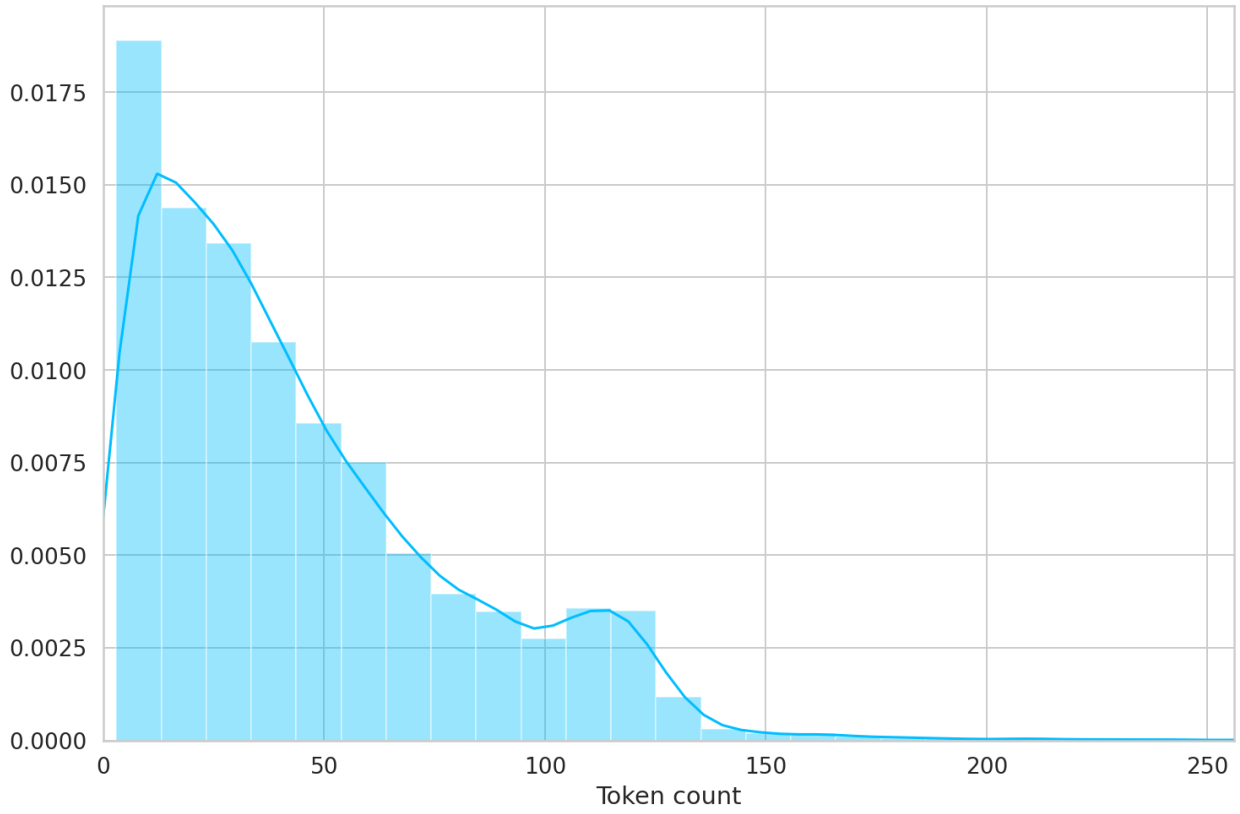
Choosing Sequence Length

BERT works with fixed-length sequences. We'll use a simple strategy to choose the max length. Let's store the token length of each review:

```
token_lens = []

for txt in df.content:
  tokens = tokenizer.encode(txt, max_length=512)
  token_lens.append(len(tokens))
```

and plot the distribution:

```
sns.distplot(token_lens)
plt.xlim([0, 256]);
plt.xlabel('Token count');
```



png

Most of the reviews seem to contain less than 128 tokens, but we'll be on the safe side and choose a maximum length of 160.

```
MAX_LEN = 160
```

We have all building blocks required to create a PyTorch dataset. Let's do it:

```
class GPReviewDataset(Dataset):

  def __init__(self, reviews, targets, tokenizer, max_len):
    self.reviews = reviews
    self.targets = targets
    self.tokenizer = tokenizer
    self.max_len = max_len

  def __len__(self):
    return len(self.reviews)

  def __getitem__(self, item):
```

```
    review = str(self.reviews[item])
    target = self.targets[item]

    encoding = self.tokenizer.encode_plus(
      review,
      add_special_tokens=True,
      max_length=self.max_len,
      return_token_type_ids=False,
      pad_to_max_length=True,
      return_attention_mask=True,
      return_tensors='pt',
    )

    return {
      'review_text': review,
      'input_ids': encoding['input_ids'].flatten(),
      'attention_mask': encoding['attention_mask'].flatten(),
      'targets': torch.tensor(target, dtype=torch.long)
    }
```

The tokenizer is doing most of the heavy lifting for us. We also return the review texts, so it'll be easier to evaluate the predictions from our model. Let's split the data:

```
df_train, df_test = train_test_split(
  df,
  test_size=0.1,
  random_state=RANDOM_SEED
)
df_val, df_test = train_test_split(
  df_test,
  test_size=0.5,
  random_state=RANDOM_SEED
)
```

```
df_train.shape, df_val.shape, df_test.shape
```

```
((14171, 12), (787, 12), (788, 12))
```

We also need to create a couple of data loaders. Here's a helper function to do it:

```
def create_data_loader(df, tokenizer, max_len, batch_size):
  ds = GPReviewDataset(
    reviews=df.content.to_numpy(),
    targets=df.sentiment.to_numpy(),
    tokenizer=tokenizer,
    max_len=max_len
  )

  return DataLoader(
    ds,
    batch_size=batch_size,
    num_workers=4
  )
```

```
BATCH_SIZE = 16

train_data_loader = create_data_loader(df_train, tokenizer, MAX_LEN, BATCH_SIZE)
val_data_loader = create_data_loader(df_val, tokenizer, MAX_LEN, BATCH_SIZE)
test_data_loader = create_data_loader(df_test, tokenizer, MAX_LEN, BATCH_SIZE)
```

Let's have a look at an example batch from our training data loader:

```
data = next(iter(train_data_loader))
data.keys()
```

```
dict_keys(['review_text', 'input_ids', 'attention_mask', 'targets'])
```

```
print(data['input_ids'].shape)
print(data['attention_mask'].shape)
print(data['targets'].shape)
```

```
torch.Size([16, 160])
torch.Size([16, 160])
torch.Size([16])
```

Sentiment Classification with BERT and Hugging Face

There are a lot of helpers that make using BERT easy with the Transformers library. Depending on the task you might want to use BertForSequenceClassification[11], BertForQuestionAnswering[12] or something else.

But who cares, right? We're *hardcore*! We'll use the basic BertModel[13] and build our sentiment classifier on top of it. Let's load the model:

```
bert_model = BertModel.from_pretrained(PRE_TRAINED_MODEL_NAME)
```

And try to use it on the encoding of our sample text:

```
last_hidden_state, pooled_output = bert_model(
  input_ids=encoding['input_ids'],
  attention_mask=encoding['attention_mask']
)
```

The `last_hidden_state` is a sequence of hidden states of the last layer of the model. Obtaining the `pooled_output` is done by applying the BertPooler[14] on `last_hidden_state`:

```
last_hidden_state.shape
```

```
torch.Size([1, 32, 768])
```

We have the hidden state for each of our 32 tokens (the length of our example sequence). But why 768? This is the number of hidden units in the feedforward-networks. We can verify that by checking the config:

```
bert_model.config.hidden_size
```

```
768
```

You can think of the `pooled_output` as a summary of the content, according to BERT. Albeit, you might try and do better. Let's look at the shape of the output:

[11] https://huggingface.co/transformers/model_doc/bert.html#bertforsequenceclassification
[12] https://huggingface.co/transformers/model_doc/bert.html#bertforquestionanswering
[13] https://huggingface.co/transformers/model_doc/bert.html#bertmodel
[14] https://github.com/huggingface/transformers/blob/edf0582c0be87b60f94f41c659ea779876efc7be/src/transformers/modeling_bert.py#L426

```
pooled_output.shape
```

```
torch.Size([1, 768])
```

We can use all of this knowledge to create a classifier that uses the BERT model:

```
class SentimentClassifier(nn.Module):

  def __init__(self, n_classes):
    super(SentimentClassifier, self).__init__()
    self.bert = BertModel.from_pretrained(PRE_TRAINED_MODEL_NAME)
    self.drop = nn.Dropout(p=0.3)
    self.out = nn.Linear(self.bert.config.hidden_size, n_classes)

  def forward(self, input_ids, attention_mask):
    _, pooled_output = self.bert(
      input_ids=input_ids,
      attention_mask=attention_mask
    )
    output = self.drop(pooled_output)
    return self.out(output
```

Our classifier delegates most of the heavy lifting to the BertModel. We use a dropout layer for some regularization and a fully-connected layer for our output. Note that we're returning the raw output of the last layer since that is required for the cross-entropy loss function in PyTorch to work.

This should work like any other PyTorch model. Let's create an instance and move it to the GPU

```
model = SentimentClassifier(len(class_names))
model = model.to(device)
```

We'll move the example batch of our training data to the GPU:

```
input_ids = data['input_ids'].to(device)
attention_mask = data['attention_mask'].to(device)

print(input_ids.shape) # batch size x seq length
print(attention_mask.shape) # batch size x seq length
```

```
torch.Size([16, 160])
torch.Size([16, 160])
```

To get the predicted probabilities from our trained model, we'll apply the softmax function to the outputs:

```
F.softmax(model(input_ids, attention_mask), dim=1)
```

```
tensor([[0.5879, 0.0842, 0.3279],
        [0.4308, 0.1888, 0.3804],
        [0.4871, 0.1766, 0.3363],
        [0.3364, 0.0778, 0.5858],
        [0.4025, 0.1040, 0.4935],
        [0.3599, 0.1026, 0.5374],
        [0.5054, 0.1552, 0.3394],
        [0.5962, 0.1464, 0.2574],
        [0.3274, 0.1967, 0.4759],
        [0.3026, 0.1118, 0.5856],
        [0.4103, 0.1571, 0.4326],
        [0.4879, 0.2121, 0.3000],
        [0.3811, 0.1477, 0.4712],
        [0.3354, 0.1354, 0.5292],
        [0.3999, 0.2822, 0.3179],
        [0.5075, 0.1684, 0.3242]], device='cuda:0', grad_fn=<SoftmaxBackward>)
```

Training

To reproduce the training procedure from the BERT paper, we'll use the AdamW[15] optimizer provided by Hugging Face. It corrects weight decay, so it's similar to the original paper. We'll also use a linear scheduler with no warmup steps:

[15] https://huggingface.co/transformers/main_classes/optimizer_schedules.html#adamw

```
EPOCHS = 10

optimizer = AdamW(model.parameters(), lr=2e-5, correct_bias=False)
total_steps = len(train_data_loader) * EPOCHS

scheduler = get_linear_schedule_with_warmup(
  optimizer,
  num_warmup_steps=0,
  num_training_steps=total_steps
)

loss_fn = nn.CrossEntropyLoss().to(device)
```

How do we come up with all hyperparameters? The BERT authors have some recommendations for fine-tuning:

- Batch size: 16, 32
- Learning rate (Adam): 5e-5, 3e-5, 2e-5
- Number of epochs: 2, 3, 4

We're going to ignore the number of epochs recommendation but stick with the rest. Note that increasing the batch size reduces the training time significantly, but gives you lower accuracy.

Let's continue with writing a helper function for training our model for one epoch:

```
def train_epoch(
  model,
  data_loader,
  loss_fn,
  optimizer,
  device,
  scheduler,
  n_examples
):
  model = model.train()

  losses = []
  correct_predictions = 0

  for d in data_loader:
    input_ids = d["input_ids"].to(device)
    attention_mask = d["attention_mask"].to(device)
    targets = d["targets"].to(device)
```

```
19
20      outputs = model(
21        input_ids=input_ids,
22        attention_mask=attention_mask
23      )
24
25      _, preds = torch.max(outputs, dim=1)
26      loss = loss_fn(outputs, targets)
27
28      correct_predictions += torch.sum(preds == targets)
29      losses.append(loss.item())
30
31      loss.backward()
32      nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
33      optimizer.step()
34      scheduler.step()
35      optimizer.zero_grad()
36
37    return correct_predictions.double() / n_examples, np.mean(losses)
```

Training the model should look familiar, except for two things. The scheduler gets called every time a batch is fed to the model. We're avoiding exploding gradients by clipping the gradients of the model using clip*grad_norm*[16].

Let's write another one that helps us evaluate the model on a given data loader:

```
1  def eval_model(model, data_loader, loss_fn, device, n_examples):
2    model = model.eval()
3
4    losses = []
5    correct_predictions = 0
6
7    with torch.no_grad():
8      for d in data_loader:
9        input_ids = d["input_ids"].to(device)
10       attention_mask = d["attention_mask"].to(device)
11       targets = d["targets"].to(device)
12
13       outputs = model(
14         input_ids=input_ids,
15         attention_mask=attention_mask
16       )
```

[16] https://pytorch.org/docs/stable/nn.html#clip-grad-norm

```
      _, preds = torch.max(outputs, dim=1)

      loss = loss_fn(outputs, targets)

      correct_predictions += torch.sum(preds == targets)
      losses.append(loss.item())

  return correct_predictions.double() / n_examples, np.mean(losses)
```

Using those two, we can write our training loop. We'll also store the training history:

```
%%time

history = defaultdict(list)
best_accuracy = 0

for epoch in range(EPOCHS):

  print(f'Epoch {epoch + 1}/{EPOCHS}')
  print('-' * 10)

  train_acc, train_loss = train_epoch(
    model,
    train_data_loader,
    loss_fn,
    optimizer,
    device,
    scheduler,
    len(df_train)
  )

  print(f'Train loss {train_loss} accuracy {train_acc}')

  val_acc, val_loss = eval_model(
    model,
    val_data_loader,
    loss_fn,
    device,
    len(df_val)
  )

  print(f'Val   loss {val_loss} accuracy {val_acc}')
  print()
```

```
33
34    history['train_acc'].append(train_acc)
35    history['train_loss'].append(train_loss)
36    history['val_acc'].append(val_acc)
37    history['val_loss'].append(val_loss)
38
39    if val_acc > best_accuracy:
40      torch.save(model.state_dict(), 'best_model_state.bin')
41      best_accuracy = val_acc
```

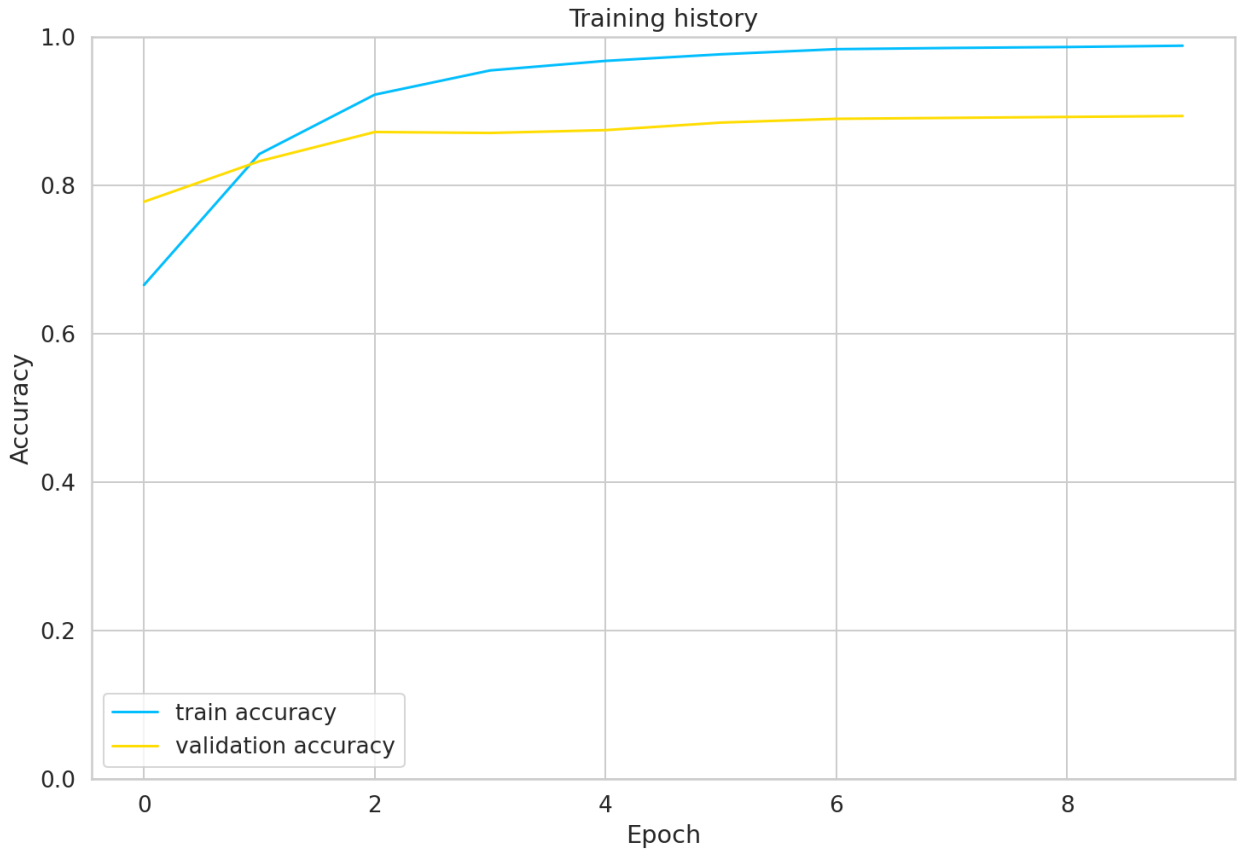
```
1  Epoch 1/10
2  ----------
3  Train loss 0.7330631300571541 accuracy 0.6653729447463129
4  Val   loss 0.5767546480894089 accuracy 0.7776365946632783
5
6  Epoch 2/10
7  ----------
8  Train loss 0.4158683338330777 accuracy 0.8420012701997036
9  Val   loss 0.5365073362737894 accuracy 0.832274459974587
10
11 Epoch 3/10
12 ----------
13 Train loss 0.24015077009679367 accuracy 0.922023851527768
14 Val   loss 0.5074492372572422 accuracy 0.8716645489199493
15
16 Epoch 4/10
17 ----------
18 Train loss 0.16012676668187295 accuracy 0.9546962105708843
19 Val   loss 0.6009970247745514 accuracy 0.8703939008894537
20
21 Epoch 5/10
22 ----------
23 Train loss 0.11209654617575301 accuracy 0.9675393409074872
24 Val   loss 0.7367783848941326 accuracy 0.8742058449809403
25
26 Epoch 6/10
27 ----------
28 Train loss 0.08572274737026433 accuracy 0.9764307388328276
29 Val   loss 0.7251267762482166 accuracy 0.8843710292249047
30
31 Epoch 7/10
32 ----------
```

```
Train loss 0.06132202987342602 accuracy 0.9833462705525369
Val   loss 0.7083295831084251 accuracy 0.889453621346887

Epoch 8/10
----------
Train loss 0.050604159273123096 accuracy 0.9849693035071626
Val   loss 0.753860274553299 accuracy 0.8907242693773825

Epoch 9/10
----------
Train loss 0.04373276197092931 accuracy 0.9862395032107826
Val   loss 0.7506809896230697 accuracy 0.8919949174078781

Epoch 10/10
----------
Train loss 0.03768671146314381 accuracy 0.9880036694658105
Val   loss 0.7431786182522774 accuracy 0.8932655654383737

CPU times: user 29min 54s, sys: 13min 28s, total: 43min 23s
Wall time: 43min 43s
```

Note that we're storing the state of the best model, indicated by the highest validation accuracy.

Whoo, this took some time! We can look at the training vs validation accuracy:

```
plt.plot(history['train_acc'], label='train accuracy')
plt.plot(history['val_acc'], label='validation accuracy')

plt.title('Training history')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend()
plt.ylim([0, 1]);
```



png

The training accuracy starts to approach 100% after 10 epochs or so. You might try to fine-tune the parameters a bit more, but this will be good enough for us.

Don't want to wait? Uncomment the next cell to download my pre-trained model:

```
# !gdown --id 1V8itWtowCYnb2Bc9K1K9SxGff9WwmogA

# model = SentimentClassifier(len(class_names))
# model.load_state_dict(torch.load('best_model_state.bin'))
# model = model.to(device)
```

Evaluation

So how good is our model on predicting sentiment? Let's start by calculating the accuracy on the test data:

```
test_acc, _ = eval_model(
  model,
  test_data_loader,
  loss_fn,
  device,
  len(df_test)
)

test_acc.item()
```

```
0.883248730964467
```

The accuracy is about 1% lower on the test set. Our model seems to generalize well.

We'll define a helper function to get the predictions from our model:

```
def get_predictions(model, data_loader):
  model = model.eval()

  review_texts = []
  predictions = []
  prediction_probs = []
  real_values = []

  with torch.no_grad():
    for d in data_loader:

      texts = d["review_text"]
      input_ids = d["input_ids"].to(device)
      attention_mask = d["attention_mask"].to(device)
      targets = d["targets"].to(device)

      outputs = model(
        input_ids=input_ids,
        attention_mask=attention_mask
      )
      _, preds = torch.max(outputs, dim=1)

      review_texts.extend(texts)
      predictions.extend(preds)
      prediction_probs.extend(outputs)
      real_values.extend(targets)
```

```
27
28    predictions = torch.stack(predictions).cpu()
29    prediction_probs = torch.stack(prediction_probs).cpu()
30    real_values = torch.stack(real_values).cpu()
31    return review_texts, predictions, prediction_probs, real_values
```

This is similar to the evaluation function, except that we're storing the text of the reviews and the predicted probabilities:

```
y_review_texts, y_pred, y_pred_probs, y_test = get_predictions(
  model,
  test_data_loader
)
```

Let's have a look at the classification report

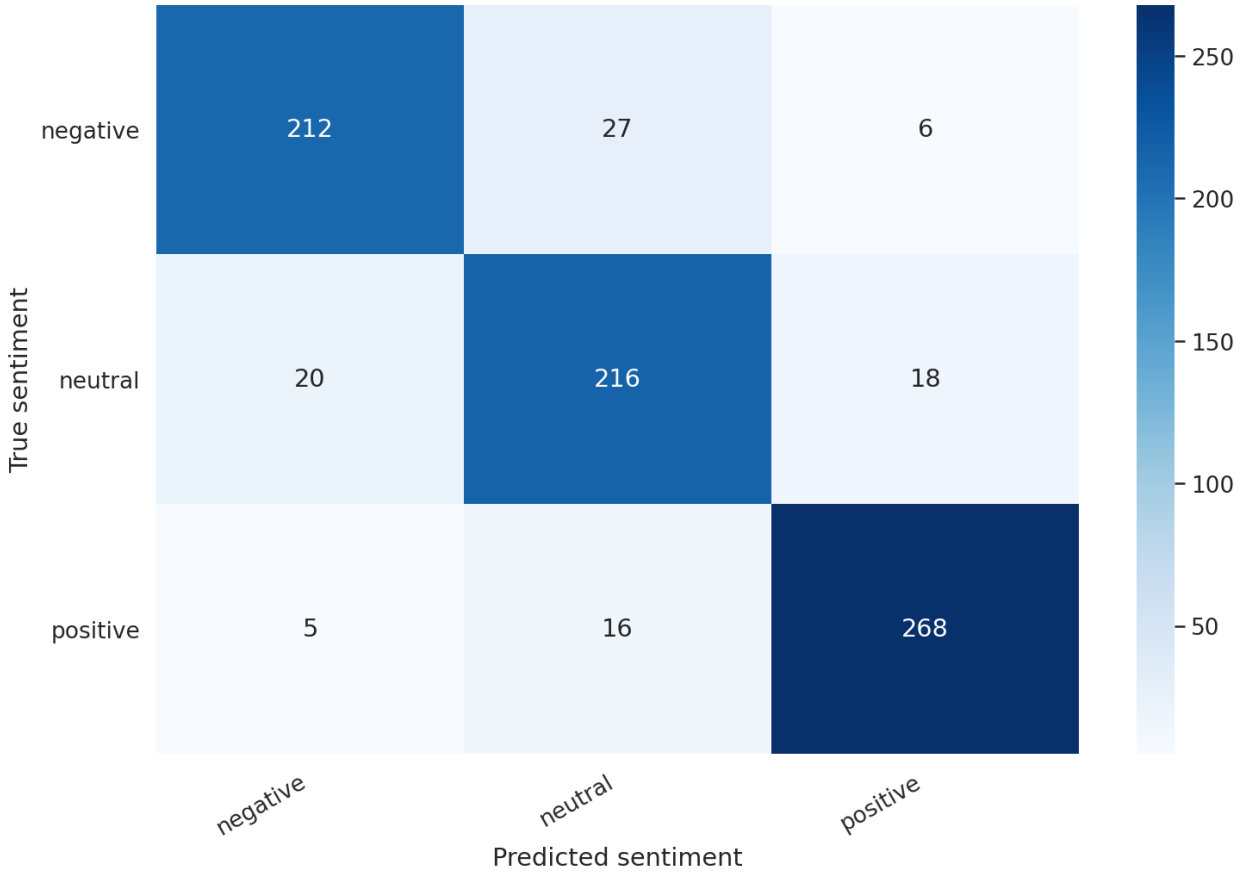
```
print(classification_report(y_test, y_pred, target_names=class_names))
```

```
              precision    recall  f1-score   support

    negative       0.89      0.87      0.88       245
     neutral       0.83      0.85      0.84       254
    positive       0.92      0.93      0.92       289

    accuracy                           0.88       788
   macro avg       0.88      0.88      0.88       788
weighted avg       0.88      0.88      0.88       788
```

Looks like it is really hard to classify neutral (3 stars) reviews. And I can tell you from experience, looking at many reviews, those are hard to classify.

We'll continue with the confusion matrix:

```
def show_confusion_matrix(confusion_matrix):
  hmap = sns.heatmap(confusion_matrix, annot=True, fmt="d", cmap="Blues")
  hmap.yaxis.set_ticklabels(hmap.yaxis.get_ticklabels(), rotation=0, ha='right')
  hmap.xaxis.set_ticklabels(hmap.xaxis.get_ticklabels(), rotation=30, ha='right')
  plt.ylabel('True sentiment')
  plt.xlabel('Predicted sentiment');

cm = confusion_matrix(y_test, y_pred)
df_cm = pd.DataFrame(cm, index=class_names, columns=class_names)
show_confusion_matrix(df_cm)
```



png

This confirms that our model is having difficulty classifying neutral reviews. It mistakes those for negative and positive at a roughly equal frequency.

That's a good overview of the performance of our model. But let's have a look at an example from our test data:

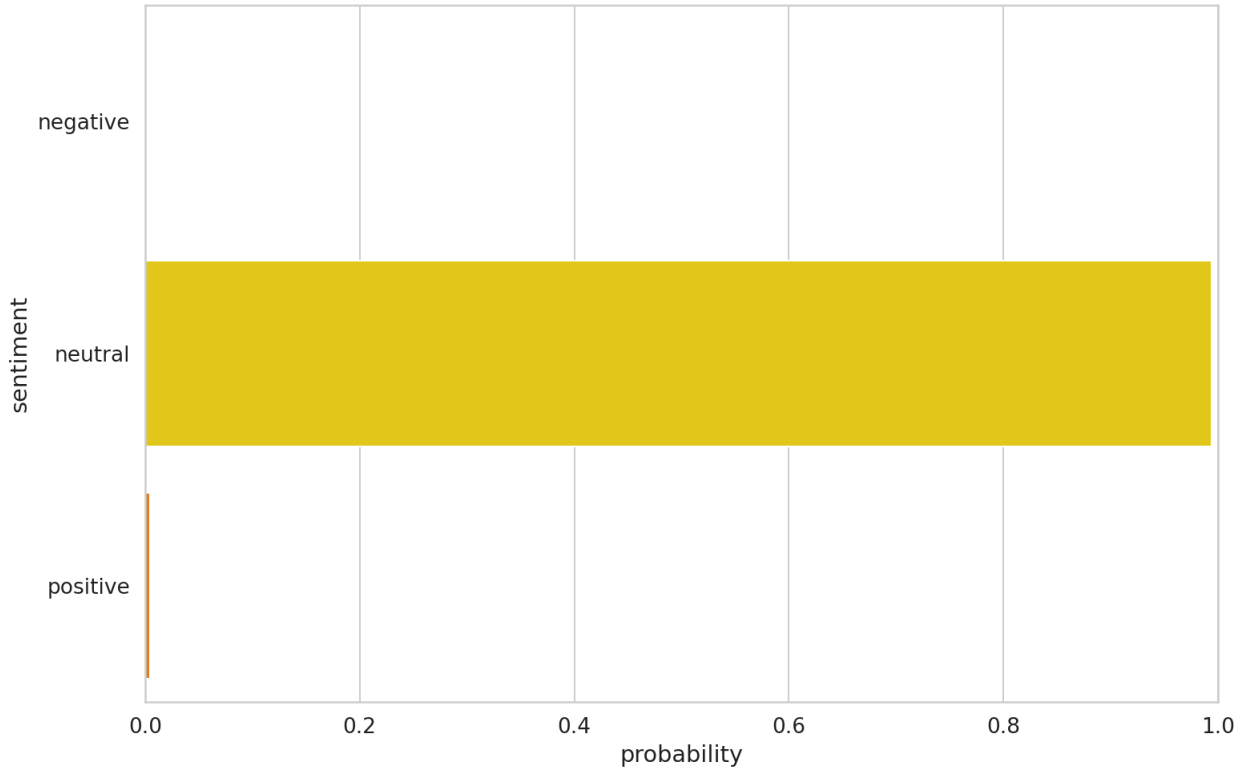
```
idx = 2

review_text = y_review_texts[idx]
true_sentiment = y_test[idx]
pred_df = pd.DataFrame({
  'class_names': class_names,
  'values': y_pred_probs[idx]
})
```

```
print("\n".join(wrap(review_text)))
print()
print(f'True sentiment: {class_names[true_sentiment]}')
```

```
I used to use Habitica, and I must say this is a great step up. I'd
like to see more social features, such as sharing tasks - only one
person has to perform said task for it to be checked off, but only
giving that person the experience and gold. Otherwise, the price for
subscription is too steep, thus resulting in a sub-perfect score. I
could easily justify $0.99/month or eternal subscription for $15. If
that price could be met, as well as fine tuning, this would be easily
worth 5 stars.

True sentiment: neutral
```

Now we can look at the confidence of each sentiment of our model:

```
sns.barplot(x='values', y='class_names', data=pred_df, orient='h')
plt.ylabel('sentiment')
plt.xlabel('probability')
plt.xlim([0, 1]);
```



png

Predicting on Raw Text

Let's use our model to predict the sentiment of some raw text:

```
review_text = "I love completing my todos! Best app ever!!!"
```

We have to use the tokenizer to encode the text:

```
encoded_review = tokenizer.encode_plus(
  review_text,
  max_length=MAX_LEN,
  add_special_tokens=True,
  return_token_type_ids=False,
  pad_to_max_length=True,
  return_attention_mask=True,
  return_tensors='pt',
)
```

Let's get the predictions from our model:

```
input_ids = encoded_review['input_ids'].to(device)
attention_mask = encoded_review['attention_mask'].to(device)

output = model(input_ids, attention_mask)
_, prediction = torch.max(output, dim=1)

print(f'Review text: {review_text}')
print(f'Sentiment  : {class_names[prediction]}')
```

```
Review text: I love completing my todos! Best app ever!!!
Sentiment  : positive
```

Summary

Nice job! You learned how to use BERT for sentiment analysis. You built a custom classifier using the Hugging Face library and trained it on our app reviews dataset!

- Run the notebook in your browser (Google Colab)[17]
- Read the `Getting Things Done with Pytorch` book[18]

You learned how to:

- Intuitively understand what BERT is
- Preprocess text data for BERT and build PyTorch Dataset (tokenization, attention masks, and padding)
- Use Transfer Learning to build Sentiment Classifier using the Transformers library by Hugging Face
- Evaluate the model on test data
- Predict sentiment on raw text

Next, we'll learn how to deploy our trained model behind a REST API and build a simple web app to access it.

[17] https://colab.research.google.com/drive/1PHv-IRLPCtv7oTcIGbsgZHqrB5LPvB7S
[18] https://github.com/curiousily/Getting-Things-Done-with-Pytorch

References

- BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding[19]
- L11 Language Models - Alec Radford (OpenAI)[20]
- The Illustrated BERT, ELMo, and co.[21]
- BERT Fine-Tuning Tutorial with PyTorch[22]
- How to Fine-Tune BERT for Text Classification?[23]
- Huggingface Transformers[24]
- BERT Explained: State of the art language model for NLP[25]

[19] https://arxiv.org/abs/1810.04805
[20] https://www.youtube.com/watch?v=BnpB3GrpsfM
[21] https://jalammar.github.io/illustrated-bert/
[22] https://mccormickml.com/2019/07/22/BERT-fine-tuning/
[23] https://arxiv.org/pdf/1905.05583.pdf
[24] https://huggingface.co/transformers/
[25] https://towardsdatascience.com/bert-explained-state-of-the-art-language-model-for-nlp-f8b21a9b6270

9. Deploy BERT for Sentiment Analysis as REST API using FastAPI

> TL;DR Learn how to create a REST API for Sentiment Analysis using a pre-trained BERT model

- Project on GitHub[1]
- Run the notebook in your browser (Google Colab)[2]
- `Getting Things Done with Pytorch` on GitHub[3]

In this tutorial, you'll learn how to deploy a pre-trained BERT model as a REST API using FastAPI. Here are the steps:

- Initialize a project using Pipenv
- Create a project skeleton
- Add the pre-trained model and create an interface to abstract the inference logic
- Update the request handler function to return predictions using the model
- Start the server and send a test request

Project setup

We'll manage our dependencies using Pipenv[4]. Here's the complete Pipfile:

```
[[source]]
name = "pypi"
url = "https://pypi.org/simple"
verify_ssl = true

[dev-packages]
black = "==19.10b0"
isort = "*"
flake8 = "*"
gdown = "*"
```

[1] https://github.com/curiousily/Deploy-BERT-for-Sentiment-Analysis-with-FastAPI
[2] https://colab.research.google.com/drive/154jf65arX4cHGaGXl2_kJ1DT8FmF4Lhf
[3] https://github.com/curiousily/Getting-Things-Done-with-Pytorch
[4] https://pipenv.pypa.io/en/latest/

```
11
12 [packages]
13 fastapi = "*"
14 uvicorn = "*"
15 pydantic = "*"
16 torch = "*"
17 transformers = "*"
18
19 [requires]
20 python_version = "3.8"
21
22 [pipenv]
23 allow_prereleases = true
```

The backbone of our REST API will be:

- FastAPI[5] - lets you easily set up a REST API (some say it might be fast, too)
- Uvicorn[6] - server that lets you do async programming with Python (pretty cool)
- Pydantic[7] - data validation by introducing types for our request and response data.

Some tools will help us write some better code (thanks to Momchil Hardalov[8] for the configs):

- Black[9] - code formatting
- isort[10] - imports sorting
- flake8[11] - check for code style (PEP 8) compliance

Building a skeleton REST API

Let's start by creating a skeleton structure for our project. Your directory should look like this:

[5] https://fastapi.tiangolo.com/
[6] https://www.uvicorn.org/
[7] https://pydantic-docs.helpmanual.io/
[8] https://github.com/mhardalov
[9] https://black.readthedocs.io/en/stable/
[10] https://timothycrosley.github.io/isort/
[11] https://flake8.pycqa.org/en/latest/

```
.
├── Pipfile
├── Pipfile.lock
└── sentiment_analyzer
    ├── api.py
```

We'll start by creating a dummy/stubbed response to test that everything is working end-to-end. Here are the contents of `api.py`:

```
from typing import Dict

from fastapi import Depends, FastAPI
from pydantic import BaseModel

app = FastAPI()


class SentimentRequest(BaseModel):
    text: str


class SentimentResponse(BaseModel):

    probabilities: Dict[str, float]
    sentiment: str
    confidence: float


@app.post("/predict", response_model=SentimentResponse)
def predict(request: SentimentRequest):
    return SentimentResponse(
        sentiment="positive",
        confidence=0.98,
        probabilities=dict(negative=0.005, neutral=0.015, positive=0.98)
    )
```

Our API expects a text - the review for sentiment analysis. The response contains the sentiment, confidence (softmax output for the sentiment) and all probabilities for each sentiment.

Adding our model

Here's the file structure of the complete project:

```
.
├── assets
│   └── model_state_dict.bin
├── bin
│   └── download_model
├── config.json
├── Pipfile
├── Pipfile.lock
└── sentiment_analyzer
    ├── api.py
    ├── classifier
    │   ├── model.py
    │   └── sentiment_classifier.py
```

We'll need the pre-trained model. We'll write the `download_model` script for that:

```
#!/usr/bin/env python
import gdown

gdown.download(
    "https://drive.google.com/uc?id=1V8itWtowCYnb2Bc9K1K9SxGff9WwmogA",
    "assets/model_state_dict.bin",
)
```

The model can be downloaded from my Google Drive. Let's get it:

```
python bin/download_model
```

Our pre-trained model is stored as a PyTorch state dict. We need to load it and use it to predict the text sentiment.

Let's start with the config file `config.json`:

```
{
    "BERT_MODEL": "bert-base-cased",
    "PRE_TRAINED_MODEL": "assets/model_state_dict.bin",
    "CLASS_NAMES": [
        "negative",
        "neutral",
        "positive"
    ],
    "MAX_SEQUENCE_LEN": 160
}
```

Next, we'll define the `sentiment_classifier.py`:

```
import json

from torch import nn
from transformers import BertModel

with open("config.json") as json_file:
    config = json.load(json_file)


class SentimentClassifier(nn.Module):
    def __init__(self, n_classes):
        super(SentimentClassifier, self).__init__()
        self.bert = BertModel.from_pretrained(config["BERT_MODEL"])
        self.drop = nn.Dropout(p=0.3)
        self.out = nn.Linear(self.bert.config.hidden_size, n_classes)

    def forward(self, input_ids, attention_mask):
        _, pooled_output = self.bert(input_ids=input_ids, attention_mask=attention_m\
ask)
        output = self.drop(pooled_output)
        return self.out(output)
```

This is the same model we've used for training. It just uses the config file.

Recall that BERT requires some special text preprocessing. We need a place to use the tokenizer from Hugging Face. We also need to do some massaging of the model outputs to convert them to our API response format.

The `Model` provides a nice abstraction (a Facade) to our classifier. It exposes a single `predict()` method and should be pretty generalizable if you want to use the same project structure as a template for your next deployment. The `model.py` file:

```
import json

import torch
import torch.nn.functional as F
from transformers import BertTokenizer

from .sentiment_classifier import SentimentClassifier

with open("config.json") as json_file:
    config = json.load(json_file)


```

```
13 class Model:
14     def __init__(self):
15 
16         self.device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
17 
18         self.tokenizer = BertTokenizer.from_pretrained(config["BERT_MODEL"])
19 
20         classifier = SentimentClassifier(len(config["CLASS_NAMES"]))
21         classifier.load_state_dict(
22             torch.load(config["PRE_TRAINED_MODEL"], map_location=self.device)
23         )
24         classifier = classifier.eval()
25         self.classifier = classifier.to(self.device)
26 
27     def predict(self, text):
28         encoded_text = self.tokenizer.encode_plus(
29             text,
30             max_length=config["MAX_SEQUENCE_LEN"],
31             add_special_tokens=True,
32             return_token_type_ids=False,
33             pad_to_max_length=True,
34             return_attention_mask=True,
35             return_tensors="pt",
36         )
37         input_ids = encoded_text["input_ids"].to(self.device)
38         attention_mask = encoded_text["attention_mask"].to(self.device)
39 
40         with torch.no_grad():
41             probabilities = F.softmax(self.classifier(input_ids, attention_mask), di\
42 m=1)
43         confidence, predicted_class = torch.max(probabilities, dim=1)
44         predicted_class = predicted_class.cpu().item()
45         probabilities = probabilities.flatten().cpu().numpy().tolist()
46         return (
47             config["CLASS_NAMES"][predicted_class],
48             confidence,
49             dict(zip(config["CLASS_NAMES"], probabilities)),
50         )
51 
52 
53 model = Model()
54 
55 
```

```
56 def get_model():
57     return model
```

We'll do the inference on the GPU, if one is available. We return the name of the predicted sentiment, the confidence, and the probabilities for each sentiment.

But why don't we define all that logic in our request handler function? For this tutorial, this is an example of overengeneering. But in the real world, when you start testing your implementation, this will be such a nice bonus.

You see, mixing everything in the request handler logic will result in countless sleepless nights. When shit hits the fan (and it will) you'll wonder if your REST or model code is wrong. This way allows you to test them, separately.

The `get_model()` function ensures that we have a single instance of our Model (Singleton). We'll use it in our API handler.

Putting everything together

Our request handler needs access to the model to return a prediction. We'll use the Dependency Injection framework[12] provided by FastAPI to inject our model. Here's the new `predict` function:

```
1 @app.post("/predict", response_model=SentimentResponse)
2 def predict(request: SentimentRequest, model: Model = Depends(get_model)):
3     sentiment, confidence, probabilities = model.predict(request.text)
4     return SentimentResponse(
5         sentiment=sentiment, confidence=confidence, probabilities=probabilities
6     )
```

The model gets injected by `Depends` and our Singleton function `get_model`. You can really appreciate the power of abstraction by looking at this!

But does it work?

Testing the API

Let's fire up the server:

```
1 uvicorn sentiment_analyzer.api:app
```

This should take a couple of seconds to load everything and start the HTTP server.

[12] https://fastapi.tiangolo.com/tutorial/dependencies/

```
http POST http://localhost:8000/predict text="This app is a total waste of time!"
```

Here's the response:

```
{
    "confidence": 0.999885082244873,
    "probabilities": {
        "negative": 0.999885082244873,
        "neutral": 8.876612992025912e-05,
        "positive": 2.614063305372838e-05
    },
    "sentiment": "negative"
}
```

Let's try with a positive one:

```
http POST http://localhost:8000/predict text="OMG. I love how easy it is to stick to\
 my schedule. Would recommend to everyone!"
```

```
{
    "confidence": 0.999932050704956,
    "probabilities": {
        "negative": 1.834999602579046e-05,
        "neutral": 4.956663542543538e-05,
        "positive": 0.999932050704956
    },
    "sentiment": "positive"
}
```

Both results are on point. Feel free to tryout with some real reviews from the Play Store.

Summary

You should now be a proud owner of ready to deploy (kind of) Sentiment Analysis REST API using BERT. Of course, you're missing lots of stuff to be production-ready - logging, monitoring, alerting, containerization, and much more. But hey, you did good!

- Project on GitHub[13]

[13] https://github.com/curiousily/Deploy-BERT-for-Sentiment-Analysis-with-FastAPI

- Run the notebook in your browser (Google Colab)[14]
- `Getting Things Done with Pytorch` on GitHub[15]

You learned how to:

- Initialize a project using Pipenv
- Create a project skeleton
- Add the pre-trained model and create an interface to abstract the inference logic
- Update the request handler function to return predictions using the model
- Start the server and send a test request

Go on then, deploy and make your users happy!

References

- FastAPI Homepage[16]
- fastAPI ML quickstart[17]

[14] https://colab.research.google.com/drive/154jf65arX4cHGaGXl2_kJ1DT8FmF4Lhf
[15] https://github.com/curiousily/Getting-Things-Done-with-Pytorch
[16] https://fastapi.tiangolo.com/
[17] https://github.com/cosmic-cortex/fastAPI-ML-quickstart

10. Object Detection on Custom Dataset with YOLO (v5)

> TL;DR Learn how to build a custom dataset for YOLO v5 (darknet compatible) and use it to fine-tune a large object detection model. The model will be ready for real-time object detection on mobile devices.

In this tutorial, you'll learn how to fine-tune a pre-trained YOLO v5 model for detecting and classifying clothing items from images.

- Run the notebook in your browser (Google Colab)[1]
- Read the `Getting Things Done with Pytorch` book[2]

Here's what we'll go over:

- Install required libraries
- Build a custom dataset in YOLO/darknet format
- Learn about YOLO model family history
- Fine-tune the largest YOLO v5 model
- Evaluate the model
- Look at some predictions

How good our final model is going to be?

Prerequisites

Let's start by installing some required libraries by the YOLOv5 project:

```
!pip install torch==1.5.1+cu101 torchvision==0.6.1+cu101 -f https://download.pytorch\
.org/whl/torch_stable.html
!pip install numpy==1.17
!pip install PyYAML==5.3.1
!pip install git+https://github.com/cocodataset/cocoapi.git#subdirectory=PythonAPI
```

We'll also install Apex by NVIDIA[3] to speed up the training of our model (this step is optional):

[1] https://colab.research.google.com/drive/1e4zvS6LyhOAayEDh3bz8MXFTJcVFSvZX?usp=sharing
[2] https://github.com/curiousily/Getting-Things-Done-with-Pytorch
[3] https://nvidia.github.io/apex/

```
!git clone https://github.com/NVIDIA/apex && cd apex && pip install -v --no-cache-di\
r --global-option="--cpp_ext" --global-option="--cuda_ext" . --user && cd .. && rm -\
rf apex
```

Build a dataset

<YouTube youTubeId=”NsxDrEJTgRw” />

The dataset contains annotations for clothing items - bounding boxes around shirts, tops, jackets, sunglasses. The dataset is from DataTurks[4] and is on Kaggle[5].

```
!gdown --id 1uWdQ2kn25RSQITtBHa9_zayplm27IXNC
```

The dataset contains a single JSON file with URLs to all images and bounding box data.

Let’s import all required libraries:

```
from pathlib import Path
from tqdm import tqdm
import numpy as np
import json
import urllib
import PIL.Image as Image
import cv2
import torch
import torchvision
from IPython.display import display
from sklearn.model_selection import train_test_split

import seaborn as sns
from pylab import rcParams
import matplotlib.pyplot as plt
from matplotlib import rc

%matplotlib inline
%config InlineBackend.figure_format='retina'
sns.set(style='whitegrid', palette='muted', font_scale=1.2)
rcParams['figure.figsize'] = 16, 10

np.random.seed(42)
```

[4] https://dataturks.com/
[5] https://www.kaggle.com/dataturks/clothing-item-detection-for-ecommerce

Each line in the dataset file contains a JSON object. Let's create a list of all annotations:

```
clothing = []
with open("clothing.json") as f:
    for line in f:
        clothing.append(json.loads(line))
```

Here's an example annotation:

```
clothing[0]
```

```
{'annotation': [{'imageHeight': 312,
   'imageWidth': 147,
   'label': ['Tops'],
   'notes': '',
   'points': [{'x': 0.02040816326530612, 'y': 0.2532051282051282},
    {'x': 0.9931972789115646, 'y': 0.8108974358974359}]}],
 'content': 'http://com.dataturks.a96-i23.open.s3.amazonaws.com/2c9fafb063ad2b650163\
b00a1ead0017/4bb8fd9d-8d52-46c7-aa2a-9c18af10aed6___Data_xxl-top-4437-jolliy-origina\
l-imaekasxahykhd3t.jpeg',
 'extras': None}
```

We have the labels, image dimensions, bounding box points (normalized in 0-1 range), and an URL to the image file.

Do we have images with multiple annotations?

```
for c in clothing:
  if len(c['annotation']) > 1:
    display(c)
```

```
{'annotation': [{'imageHeight': 312,
   'imageWidth': 265,
   'label': ['Jackets'],
   'notes': '',
   'points': [{'x': 0, 'y': 0.6185897435897436},
    {'x': 0.026415094339622643, 'y': 0.6185897435897436}]},
  {'imageHeight': 312,
   'imageWidth': 265,
   'label': ['Skirts'],
   'notes': '',
```

```
    'points': [{'x': 0.01509433962264151, 'y': 0.03205128205128205},
     {'x': 1, 'y': 0.9839743589743589}]}],
 'content': 'http://com.dataturks.a96-i23.open.s3.amazonaws.com/2c9fafb063ad2b650163\
b00a1ead0017/b3be330c-c211-45bb-b244-11aef08021c8___Data_free-sk-5108-mudrika-origin\
al-imaf4fz626pegq9f.jpeg',
 'extras': None}
```

Just a single example. We'll need to handle it, though.

Let's get all unique categories:

```
categories = []
for c in clothing:
  for a in c['annotation']:
    categories.extend(a['label'])
categories = list(set(categories))
categories.sort()
categories
```

```
    ['Jackets',
     'Jeans',
     'Shirts',
     'Shoes',
     'Skirts',
     'Tops',
     'Trousers',
     'Tshirts',
     'sunglasses']
```

We have 9 different categories. Let's split the data into a training and validation set:

```
train_clothing, val_clothing = train_test_split(clothing, test_size=0.1)
len(train_clothing), len(val_clothing)
```

```
(453, 51)
```

Sample image and annotation

Let's have a look at an image from the dataset. We'll start by downloading it:

```
row = train_clothing[10]

img = urllib.request.urlopen(row["content"])
img = Image.open(img)
img = img.convert('RGB')

img.save("demo_image.jpeg", "JPEG")
```

Here's how our sample annotation looks like:

```
row
```

```
{'annotation': [{'imageHeight': 312,
   'imageWidth': 145,
   'label': ['Tops'],
   'notes': '',
   'points': [{'x': 0.013793103448275862, 'y': 0.22756410256410256},
    {'x': 1, 'y': 0.7948717948717948}]}],
 'content': 'http://com.dataturks.a96-i23.open.s3.amazonaws.com/2c9fafb063ad2b650163\
b00a1ead0017/ec339ad6-6b73-406a-8971-f7ea35d47577___Data_s-top-203-red-srw-original-\
imaf2nfrxdzvhh3k.jpeg',
 'extras': None}
```

We can use OpenCV to read the image:

```
img = cv2.cvtColor(cv2.imread(f'demo_image.jpeg'), cv2.COLOR_BGR2RGB)
img.shape
```

```
(312, 145, 3)
```

Let's add the bounding box on top of the image along with the label:

```
for a in row['annotation']:
  for label in a['label']:

    w = a['imageWidth']
    h = a['imageHeight']

    points = a['points']
    p1, p2 = points

    x1, y1 = p1['x'] * w, p1['y'] * h
    x2, y2 = p2['x'] * w, p2['y'] * h

    cv2.rectangle(
      img,
      (int(x1), int(y1)),
      (int(x2), int(y2)),
      color=(0, 255, 0),
      thickness=2
    )

    ((label_width, label_height), _) = cv2.getTextSize(
        label,
        fontFace=cv2.FONT_HERSHEY_PLAIN,
        fontScale=1.75,
        thickness=2
    )

    cv2.rectangle(
      img,
      (int(x1), int(y1)),
      (int(x1 + label_width + label_width * 0.05), int(y1 + label_height + label_hei\
ght * 0.25)),
      color=(0, 255, 0),
      thickness=cv2.FILLED
    )

    cv2.putText(
      img,
      label,
      org=(int(x1), int(y1 + label_height + label_height * 0.25)), # bottom left
      fontFace=cv2.FONT_HERSHEY_PLAIN,
      fontScale=1.75,
      color=(255, 255, 255),
```

```
        thickness=2
    )
```

The point coordinates are converted back to pixels and used to draw rectangles over the image. Here's the result:

```
plt.imshow(img)
plt.axis('off');
```

png

Convert to YOLO format

YOLO v5 requires the dataset to be in the *darknet format*. Here's an outline of what it looks like:

- One txt with labels file per image
- One row per object
- Each row contains: `class_index bbox_x_center bbox_y_center bbox_width bbox_height`
- Box coordinates must be normalized between 0 and 1

Let's create a helper function that builds a dataset in the correct format for us:

```
def create_dataset(clothing, categories, dataset_type):

  images_path = Path(f"clothing/images/{dataset_type}")
  images_path.mkdir(parents=True, exist_ok=True)

  labels_path = Path(f"clothing/labels/{dataset_type}")
  labels_path.mkdir(parents=True, exist_ok=True)

  for img_id, row in enumerate(tqdm(clothing)):

    image_name = f"{img_id}.jpeg"

    img = urllib.request.urlopen(row["content"])
    img = Image.open(img)
    img = img.convert("RGB")

    img.save(str(images_path / image_name), "JPEG")

    label_name = f"{img_id}.txt"

    with (labels_path / label_name).open(mode="w") as label_file:

      for a in row['annotation']:

        for label in a['label']:

          category_idx = categories.index(label)

          points = a['points']
          p1, p2 = points

```

```
            x1, y1 = p1['x'], p1['y']
            x2, y2 = p2['x'], p2['y']

            bbox_width = x2 - x1
            bbox_height = y2 - y1

            label_file.write(
              f"{category_idx} {x1 + bbox_width / 2} {y1 + bbox_height / 2} {bbox_widt\
h} {bbox_height}\n"
            )
```

We'll use it to create the train and validation datasets:

```
create_dataset(train_clothing, categories, 'train')
create_dataset(val_clothing, categories, 'val')
```

Let's have a look at the file structure:

```
!tree clothing -L 2
```

```
clothing
├── images
│   ├── train
│   └── val
└── labels
├── train
└── val

6 directories, 0 files
```

And a single annotation example:

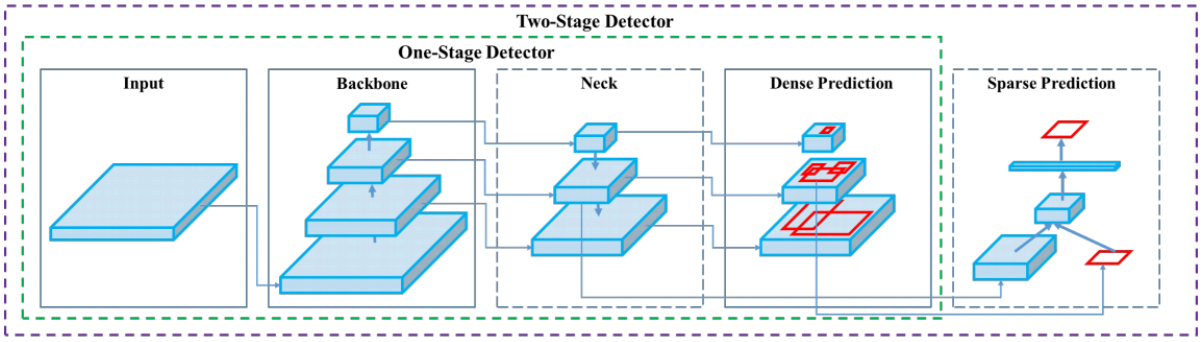
```
!cat clothing/labels/train/0.txt
```

```
4 0.525462962962963 0.5432692307692308 0.9027777777777778 0.9006410256410257
```

Fine-tuning YOLO v5

<YouTube youTubeId="XNRzZkZ-Byg" />

The YOLO abbreviation stands for You Only Look Once. YOLO models are one stage object detectors.



One-stage vs two-stage object detectors. Image from the YOLO v4 paper

YOLO models are very light and fast. They are not the most accurate object detections around[6], though. Ultimately, those models are the choice of many (if not all) practitioners interested in real-time object detection (FPS >30)[7].

Contreversy

Joseph Redmon introduced YOLO v1 in the 2016 paper You Only Look Once: Unified, Real-Time Object Detection[8]. The implementation uses the Darknet Neural Networks library[9].

He also co-authored the YOLO v2 paper in 2017 YOLO9000: Better, Faster, Stronger[10]. A significant improvement over the first iteration with much better localization of objects.

The final iteration, from the original author, was published in the 2018 paper YOLOv3: An Incremental Improvement[11].

Then things got a bit wacky. Alexey Bochkovskiy published YOLOv4: Optimal Speed and Accuracy of Object Detection[12] on April 23, 2020. The project has an open-source repository on GitHub[13].

YOLO v5 got open-sourced on May 30, 2020[14] by Glenn Jocher[15] from ultralytics. There is no published paper, but the complete project is on GitHub[16].

[6] https://paperswithcode.com/sota/object-detection-on-coco
[7] https://paperswithcode.com/sota/real-time-object-detection-on-coco
[8] https://arxiv.org/pdf/1506.02640.pdf
[9] https://pjreddie.com/darknet/
[10] https://arxiv.org/pdf/1612.08242.pdf
[11] https://arxiv.org/pdf/1804.02767.pdf
[12] https://arxiv.org/abs/2004.10934
[13] https://github.com/AlexeyAB/darknet
[14] https://github.com/ultralytics/yolov5/commit/1e84a23f38fad9e52b59101e9f1246d93066ed1e
[15] https://github.com/glenn-jocher
[16] https://github.com/ultralytics/yolov5

The community at Hacker News got into a heated debate about the project naming[17]. Even the guys at Roboflow wrote Responding to the Controversy about YOLOv5[18] article about it. They also did a great comparison between YOLO v4 and v5.

My opinion? As long as you put out your work for the whole world to use/see - I don't give a flying fuck. I am not going to comment on points/arguments that are obvious.

YOLO v5 project setup

YOLO v5 uses PyTorch, but everything is abstracted away. You need the project itself (along with the required dependencies).

Let's start by cloning the GitHub repo and checking out a specific commit (to ensure reproducibility):

```
!git clone https://github.com/ultralytics/yolov5
%cd yolov5
!git checkout ec72eea62bf5bb86b0272f2e65e413957533507f
```

We need two configuration files. One for the dataset and one for the model we're going to use. Let's download them:

```
!gdown --id 1ZycPS5Ft_0vlfgHnLsfvZPhcH6qOAqBO -O data/clothing.yaml
!gdown --id 1czESPsKbOWZF7_PkCcvRfTiUUJfpx12i -O models/yolov5x.yaml
```

The model config changes the number of classes to 9 (equal to the ones in our dataset). The dataset config `clothing.yaml` is a bit more complex:

```
train: ../clothing/images/train/
val: ../clothing/images/val/

nc: 9

names:
  [
    "Jackets",
    "Jeans",
    "Shirts",
    "Shoes",
    "Skirts",
    "Tops",
    "Trousers",
```

[17] https://news.ycombinator.com/item?id=23478151
[18] https://blog.roboflow.ai/yolov4-versus-yolov5/

```
    "Tshirts",
    "sunglasses",
  ]
```

This file specifies the paths to the training and validation sets. It also gives the number of classes and their names (you should order those correctly).

Training

Fine-tuning an existing model is very easy. We'll use the largest model **YOLOv5x** (89M parameters), which is also the most accurate.

In our case, we don't really care about speed. We just want the best accuracy you can get. The checkpoint you're going to use for a different problem(s) is contextually specific. Take a look at the overview of the pre-trained checkpoints[19].

To train a model on a custom dataset, we'll call the `train.py` script. We'll pass a couple of parameters:

- img 640 - resize the images to 640x640 pixels
- batch 4 - 4 images per batch
- epochs 30 - train for 30 epochs
- data ./data/clothing.yaml - path to dataset config
- cfg ./models/yolov5x.yaml - model config
- weights yolov5x.pt - use pre-trained weights from the YOLOv5x model
- name yolov5x_clothing - name of our model
- cache - cache dataset images for faster training

```
!python train.py --img 640 --batch 4 --epochs 30 \
  --data ./data/clothing.yaml --cfg ./models/yolov5x.yaml --weights yolov5x.pt \
  --name yolov5x_clothing --cache
```

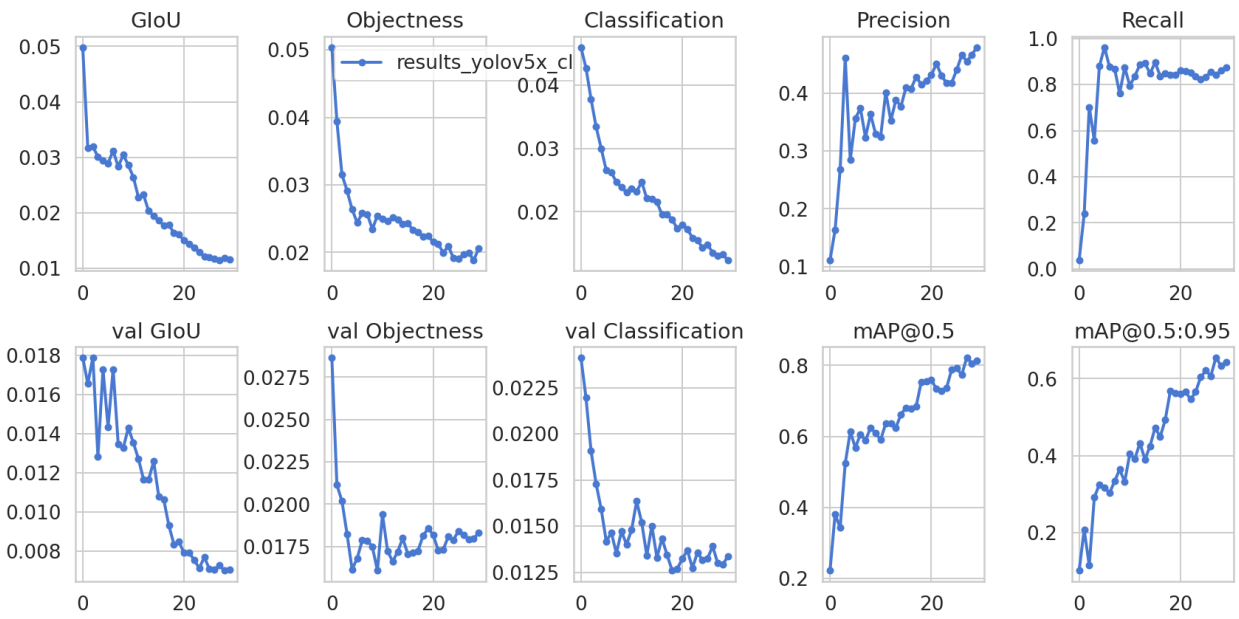
The training took around 30 minutes on Tesla P100. The best model checkpoint is saved to `weights/best_yolov5x_clothing.pt`.

Evaluation

The project includes a great utility function `plot_results()` that allows you to evaluate your model performance on the last training run:

[19] https://github.com/ultralytics/yolov5/blob/f9ae460eeccd30bdc43a89a37f74b9cc7b93d52f/README.md#pretrained-checkpoints

```
from utils.utils import plot_results

plot_results();
```



png

Looks like the mean average precision (mAP) is getting better throughout the training. The model might benefit from more training, but it is good enough.

Making predictions

Let's pick 50 images from the validation set and move them to `inference/images` to see how our model does on those:

```
!find ../clothing/images/val/ -maxdepth 1 -type f | head -50 | xargs cp -t "./infere\
nce/images/"
```

We'll use the `detect.py` script to run our model on the images. Here are the parameters we're using:

- weights weights/best_yolov5x_clothing.pt - checkpoint of the model
- img 640 - resize the images to 640x640 px
- conf 0.4 - take into account predictions with confidence of 0.4 or higher
- source ./inference/images/ - path to the images

```
!python detect.py --weights weights/best_yolov5x_clothing.pt \
  --img 640 --conf 0.4 --source ./inference/images/
```

We'll write a helper function to show the results:

```
def load_image(img_path: Path, resize=True):
  img = cv2.cvtColor(cv2.imread(str(img_path)), cv2.COLOR_BGR2RGB)
  img = cv2.resize(img, (128, 256), interpolation = cv2.INTER_AREA)
  return img

def show_grid(image_paths):
  images = [load_image(img) for img in image_paths]
  images = torch.as_tensor(images)
  images = images.permute(0, 3, 1, 2)
  grid_img = torchvision.utils.make_grid(images, nrow=11)
  plt.figure(figsize=(24, 12))
  plt.imshow(grid_img.permute(1, 2, 0))
  plt.axis('off');
```

Here are some of the images along with the detected clothing:

```
img_paths = list(Path("inference/output").glob("*.jpeg"))[:22]
show_grid(img_paths)
```

png

To be honest with you. I am really blown away with the results!

Summary

You now know how to create a custom dataset and fine-tune one of the YOLO v5 models on your own. Nice!

- Run the notebook in your browser (Google Colab)[20]
- Read the `Getting Things Done with Pytorch` book[21]

Here's what you've learned:

- Install required libraries
- Build a custom dataset in YOLO/darknet format
- Learn about YOLO model family history
- Fine-tune the largest YOLO v5 model
- Evaluate the model
- Look at some predictions

How well does your model do on your dataset? Let me know in the comments below.

In the next part, you'll learn how to deploy your model a mobile device.

References

- Clothing Item Detection for E-Commerce dataset[22]
- YOLOv5 GitHub[23]
- YOLOv5 Train on Custom Data[24]
- NVIDIA Apex on GitHub[25]
- YOLOv4: Optimal Speed and Accuracy of Object Detection[26]

[20] https://colab.research.google.com/drive/1e4zvS6LyhOAayEDh3bz8MXFTJcVFSvZX?usp=sharing
[21] https://github.com/curiousily/Getting-Things-Done-with-Pytorch
[22] https://www.kaggle.com/dataturks/clothing-item-detection-for-ecommerce
[23] https://github.com/ultralytics/yolov5
[24] https://github.com/ultralytics/yolov5/wiki/Train-Custom-Data
[25] https://github.com/NVIDIA/apex
[26] https://arxiv.org/pdf/2004.10934.pdf