Lecture 6: Advanced Time Series Modelling

Contents

Lecture Outline

Lecture Learning Objectives

- Develop a neural network for time series forecasting in `pytorch`.
- Develop a convolutional neural network for time series forecasting in `pytorch`.
- Understand how a recurrent neural network differs to a standard feed-forward network and why they can be good for modelling sequential data.
- Describe how a long short-term memory network differs to a recurrent neural network.
- Remember to check this lecture for advanced methods and packages related to time series modelling.

Imports

```
import numpy as np
import pandas as pd
import warnings
warnings.filterwarnings('ignore')
import plotly.express as px
# import mxnet as mx
import lightning.pytorch as pl
import gluonts
import torch
from torch import nn, optim
from torch.utils.data import DataLoader, TensorDataset
from torchinfo import summary
from sklearn.preprocessing import MinMaxScaler, StandardScaler
# Custom modules
from scripts.utils import *
from scripts.plotting import *
# Plot defaults
px.defaults.height = 400; px.defaults.width = 550
```

1. Deep learning with time series

Traditionally, classical time-series methods such as exponential smoothing or ARIMA have been the workhorse in forecasting. These methods are well understood, easy to intepret, and highly effective in univariate time-series applications, or time-series with a strong trend/seasonality. However, they suffer from some limitations:

- Focus on linear relationships between x & y
- Inputs must be specified, not learned automatically
- Focus on univariate time-series, lack support for multi-input or multi-output
- Focus on regression, lack support for classification
- Assume complete, non-missing data

Neural networks provide a flexible architecture that can be valuable in forecasting:

- Robust to noise/missing value
- Support non-linear relationships, learn arbitrary mapping functions from inputs to outputs
- Support multivariate inputs/outputs
- Features can be learned, don't need to specify in advance (CNN)
- Temporal dependecies can be learned (RNN, LSTM)

Common applications of deep learning for time-series:

- Sensorial data, data generated with high frequency, data with no clear trend/seasonality
- Multivariate time-series
- Hierachical time-series
- Time-series classification
- Anomaly detection

1.1. Neural networks

- Back in lecture 4, I showed you how to use machine learning for time series modelling
- The same concepts discussed there apply to using plain old feed-forward neural networks for time series modelling. We wrangle some features from our time series and use those to train a network.
- I actually did this in lecture 5, where I trained a simple network with `pytorch` for quantile regression.

- When training an ML model or feed-forward network as above, the order of the features doesn't actually matter, our features could look like this:

Target	Lag-1	Lag-2	Lag-3
4	8	2	5
7	3	1	6
...	...	...	...

- Or like this:

Target	Lag-3	Lag-1	Lag-2
4	5	8	2
7	6	3	1
...	...	...	...

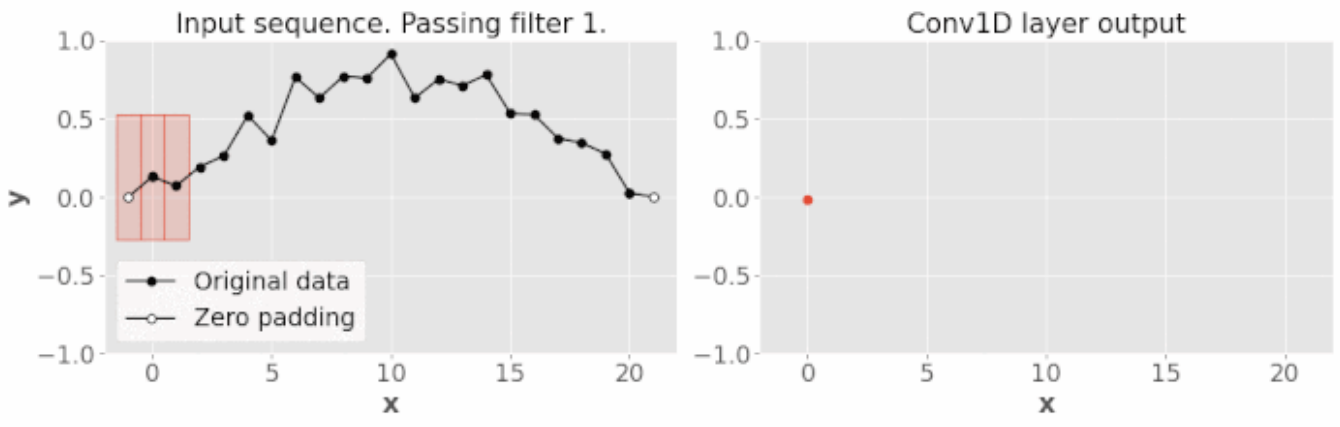
- Our model would learn the same mapping from features to target.
- But one thing I've been stressing in these lectures is that what makes time series data different is its inherent temporal structure, which we're more or less throwing away in the approach above. So a natural question is, can we retain and make use of the structure?
- In a previous course, we learned that when working with image data, convolutional neural networks are effective because they retain the structure of the image. We can apply the same principle here!

1.2. Convolutional neural networks

- We know that CNNs are useful when we wish to preserve the structure of our data. We have previously used them to work with images, which are 3D data (channels x width x height). But we can also use CNNs to work with time series data, which is 2D (features x length).
- With image data, we use the `pytorch` class `torch.nn.Conv2d()`. With time series data, we use `torch.nn.Conv1d()`. But these layers are really doing the same thing, just in a different dimensionality.
- For example, rather than an image, we're now taking in some sequence of values and passing our filters over them to generate "filtered" versions of the series that focus on

particular patterns/elements (some might highlight noise, some might smooth, some might highlight seasonality, etc.)

- For example, let's see a sequence of 20 values, filtered with 4 kernels of size 3:



- In `pytorch` code:

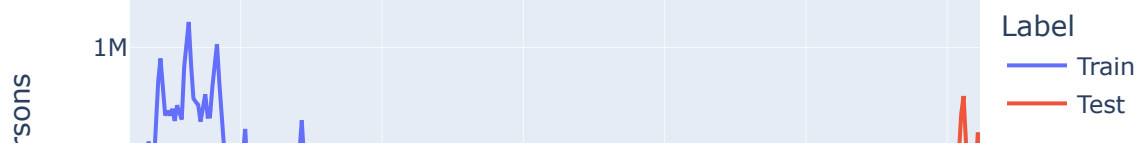
```
series = torch.randn(1, 1, 20)  # (batch size, features, sequence length)
conv_layer = nn.Conv1d(in_channels=1, out_channels=4, kernel_size=3, padding=1
conv_layer(series).shape
```

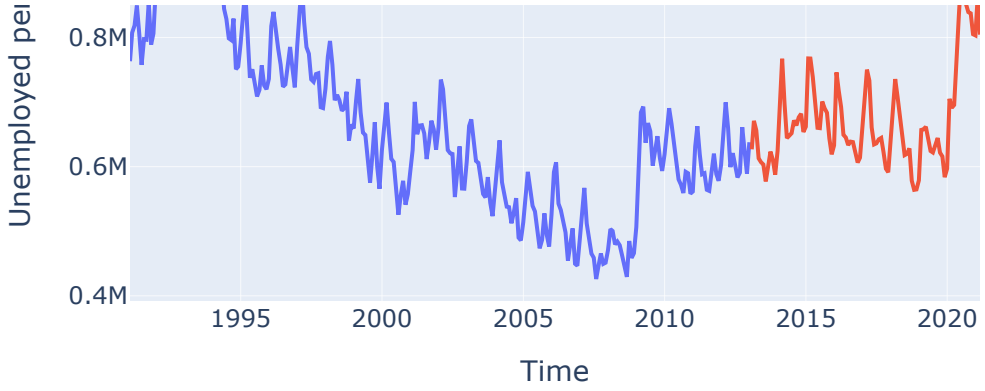
```
torch.Size([1, 4, 20])
```

- Okay, let's try making a CNN on some real data:

```
# Load data
train = (pd.read_csv("data/unemployed_train.csv", index_col=0, parse_dates=Tru
           .resample("1M").mean()
           .assign(Label="Train")
         )
test = (pd.read_csv("data/unemployed_valid.csv", index_col=0, parse_dates=True
           .resample("1M").mean()
           .assign(Label="Test")
         )
px.line(pd.concat((train, test)), y="Unemployed persons", color="Label", width
```

Australian Unemployment





- The input data to our model should be shape `(batch size x features x sequence length)`:
  - `batch size` is up to you, as usual;
  - `features` is the number of features we have, for a univariate time series, that's one. I'll show a multivariate model after this.
  - `sequence length` is the length of the segment of time series we wish to feed in, this is something that can be tuned, but if there's seasonality, you can usually set it to one or two times the seasonal period.
- We first need to prepare our time series for our network. Our features will be sequences of size `sequence length` and our target will be the next value in the sequence (i.e., the one we wish to predict with our sequence). Here's some examples for a sequence length of 24:

```
SEQUENCE_LENGTH = 24
BATCH_SIZE = 16
# generate sequences
cnn_data = lag_df(train[["Unemployed persons"]], SEQUENCE_LENGTH).dropna()
cnn_data = cnn_data[cnn_data.columns[::-1]]  # flip sequence for more logical
cnn_data.iloc[:3]
```

	Unemployed persons-24	Unemployed persons-23	Unemployed persons-22	Unemployed persons-21	Unemployed persons-20	Ur p
Time						
1993-01-31	763321.6342	807546.6346	818667.6259	849493.4822	805407.9362	75
1993-02-28	807546.6346	818667.6259	849493.4822	805407.9362	757682.9427	79
1993-03-31	818667.6259	849493.4822	805407.9362	757682.9427	797379.0700	79

3 rows × 25 columns

- Let's standardize, coerce this data into tensors, and create a data loader to train a `pytorch` model with:

```
def prepare_dataloader(df, sequence_length, batch_size, shuffle=True, lstm=Fal
    # generate sequences
    scaler = StandardScaler()  # standardize data
    scaled_data = df.copy()
    scaled_data["Unemployed persons"] = scaler.fit_transform(scaled_data[["Une
    scaled_data = lag_df(scaled_data[["Unemployed persons"]], sequence_length)
    scaled_data = scaled_data[scaled_data.columns[::-1]].to_numpy()  # flip se
    # split into X and y
    X_train = scaled_data[:, :-1]
    y_train = scaled_data[:, -1]
    # coerce to dataloader
    if lstm:
        dataset = TensorDataset(torch.tensor(X_train, dtype=torch.float32).uns
                                torch.tensor(y_train, dtype=torch.float32))
    else:
        dataset = TensorDataset(torch.tensor(X_train, dtype=torch.float32).uns
                                torch.tensor(y_train, dtype=torch.float32))
    dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=shuffle, d
    return dataloader, scaled_data, scaler
```

```
SEQUENCE_LENGTH = 24
BATCH_SIZE = 16
dataloader, scaled_data, scaler = prepare_dataloader(train, SEQUENCE_LENGTH, B
# example batch
X, y = next(iter(dataloader))
print(f"X shape: {X.shape}")
print(f"y shape: {y.shape}")
```

```
X shape: torch.Size([16, 1, 24])
y shape: torch.Size([16])
```

- Now, let's create a simple model of a single convolutoinal layer:

```
class CNN(nn.Module):
    def __init__(self, input_size, output_size):
        super().__init__()
        self.main = nn.Sequential(
            nn.Conv1d(input_size, 24, kernel_size=3),
            nn.ReLU(),
            nn.BatchNorm1d(24),
            nn.MaxPool1d(3),
            nn.Flatten(),
            nn.Linear(168, 50),
            nn.ReLU(),
            nn.Linear(50, output_size),
        )

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

```
# instaniate model framework
model = CNN(input_size=1, output_size=1)
optimizer = optim.Adam(model.parameters())
criterion = nn.MSELoss()
# train
EPOCHS = 200
for epoch in range(1, EPOCHS + 1):
    for X, y in dataloader:
        optimizer.zero_grad()
        y_hat = model(X)
        loss = criterion(y_hat.flatten(), y)
        loss.backward()
        optimizer.step()
    if epoch % 20 == 0: print(f"Epoch {epoch:03}. Loss = {loss.item():.4f}")
```

```
Epoch 020. Loss = 0.0706
```

```
Epoch 040. Loss = 0.0707
```

```
Epoch 060. Loss = 0.0148
```

```
Epoch 080. Loss = 0.0194
```

```
Epoch 100. Loss = 0.0838
```

```
Epoch 120. Loss = 0.0092
```

```
Epoch 140. Loss = 0.0049
```

```
Epoch 160. Loss = 0.0033
```

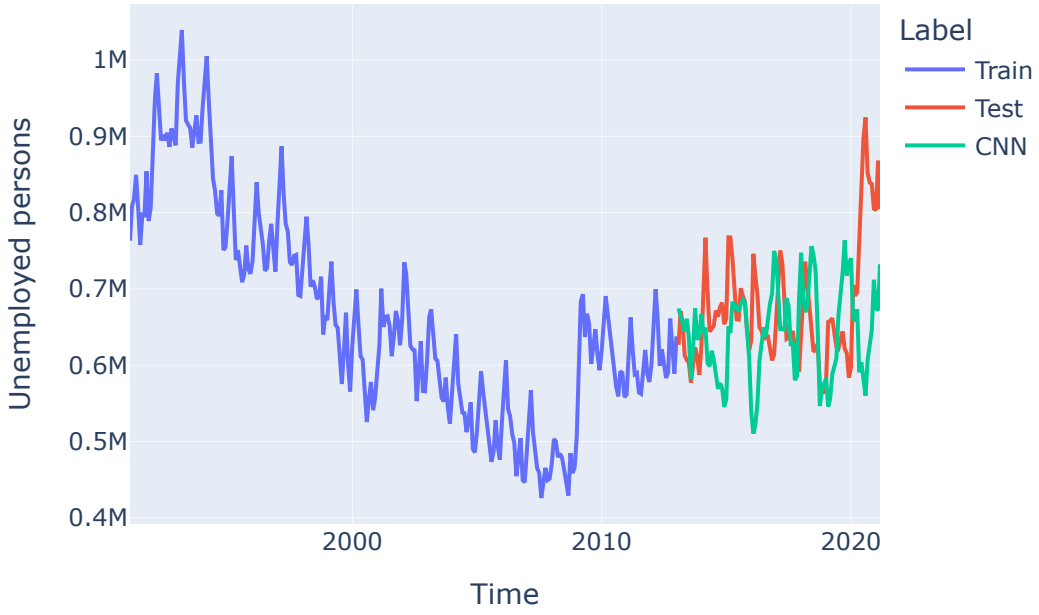
```
Epoch 180. Loss = 0.0048
```

```
Epoch 200. Loss = 0.0020
```

- Let's check out our predictions, I'll make them recursively:

```
def recursive_CNN_forecast(input_data, model, n=20, responses=1):
    forecast = np.empty((n, responses))
    for i, n in enumerate(range(n)):
        forecast[i] = model(input_data).detach().numpy()
        input_data = torch.cat((torch.tensor([forecast[i]], dtype=torch.float3
                                input_data[:, :, :-1]), -1)
    return forecast
```

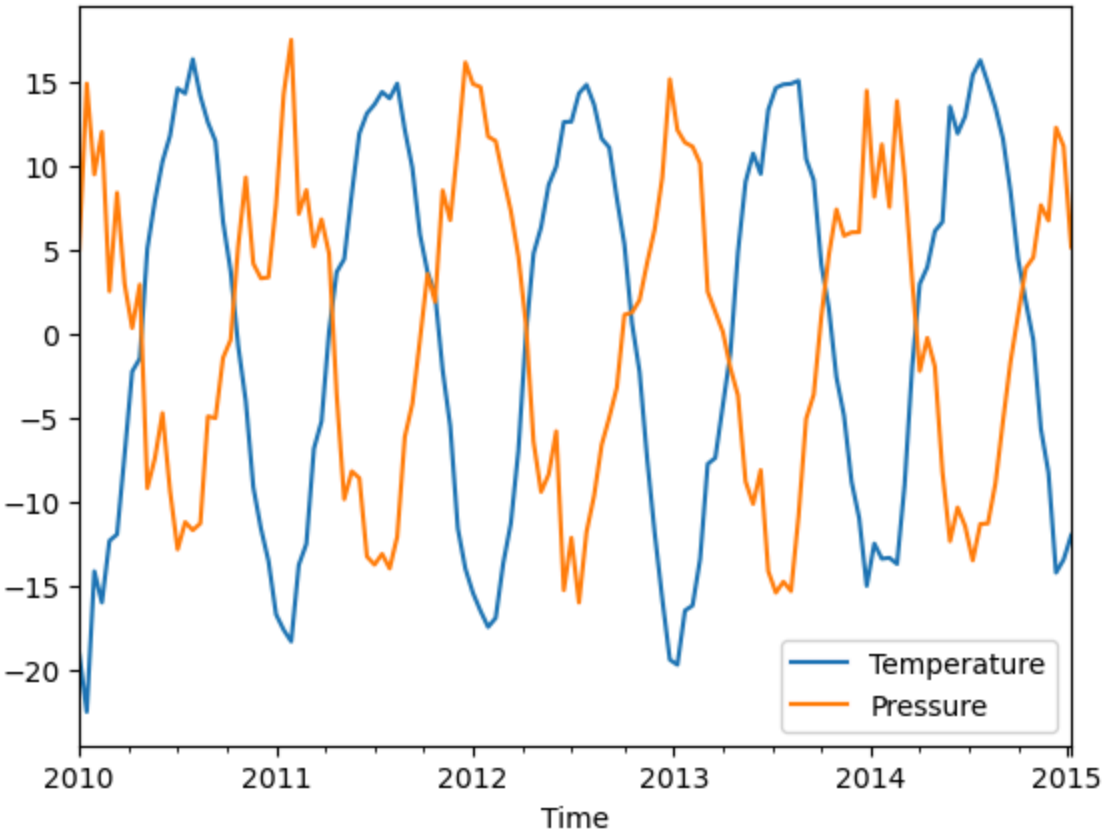
```
input_data = torch.tensor(scaled_data[-1, 1:], dtype=torch.float32).view(1, 1,
forecasts = recursive_CNN_forecast(input_data, model, len(test))
forecasts = scaler.inverse_transform(forecasts)
cnn = pd.DataFrame(
    {
        "Unemployed persons": forecasts.flatten(),
        "Label": "CNN",
    },
    index=test.index,
)

px.line(pd.concat((train, test, cnn)), y="Unemployed persons", color="Label")
```



- Even though this is just a random architecture and we didn't tune at all, you can already see the network picking up some temporal patterns - cool!
- There are tons of variations we can make here too:
  - Single input -> single output (we just did this)
  - Multiple inputs -> single output
  - Single input -> multiple outputs
  - Multiple inputs -> multiple outputs
- Let's quickly try a "multiple inputs -> multiple outputs"

```
df = (pd.read_csv("data/pollution.csv", index_col=0, parse_dates=True, usecols
        .resample("2W")
        .mean()
    )
df -= df.mean()
df.plot()
```

```
<Axes: xlabel='Time'>
```




```
# generate sequences
cnn_data = lag_df(df, lag=SEQUENCE_LENGTH).dropna()
X_train = (cnn_data.drop(columns=["Temperature", "Pressure"])
                   .to_numpy()
                   .reshape(-1, 2, SEQUENCE_LENGTH)
          )
y_train = cnn_data[["Temperature", "Pressure"]].to_numpy()
# coerce to dataloader
dataset = TensorDataset(torch.tensor(X_train, dtype=torch.float32),
                        torch.tensor(y_train, dtype=torch.float32))
dataloader = DataLoader(dataset, batch_size=BATCH_SIZE, shuffle=True)
# example batch
X, y = next(iter(dataloader))
print(f"X shape: {X.shape}")
print(f"y shape: {y.shape}")
```

```
X shape: torch.Size([16, 2, 24])
y shape: torch.Size([16, 2])
```

```
# instantiate model framework
model = CNN(input_size=2, output_size=2)
optimizer = optim.Adam(model.parameters())
criterion = nn.MSELoss()
# train
EPOCHS = 200
for epoch in range(1, EPOCHS + 1):
    for X, y in dataloader:
        optimizer.zero_grad()
        y_hat = model(X)
        loss = criterion(y_hat, y)
        loss.backward()
        optimizer.step()
    if epoch % 20 == 0: print(f"Epoch {epoch}. Loss = {loss.item():.2f}")
```

```
Epoch 20. Loss = 3.28
```

```
Epoch 40. Loss = 2.21
```

```
Epoch 60. Loss = 2.09
```

```
Epoch 80. Loss = 1.54
```

```
Epoch 100. Loss = 0.89
```

```
Epoch 120. Loss = 0.82
```

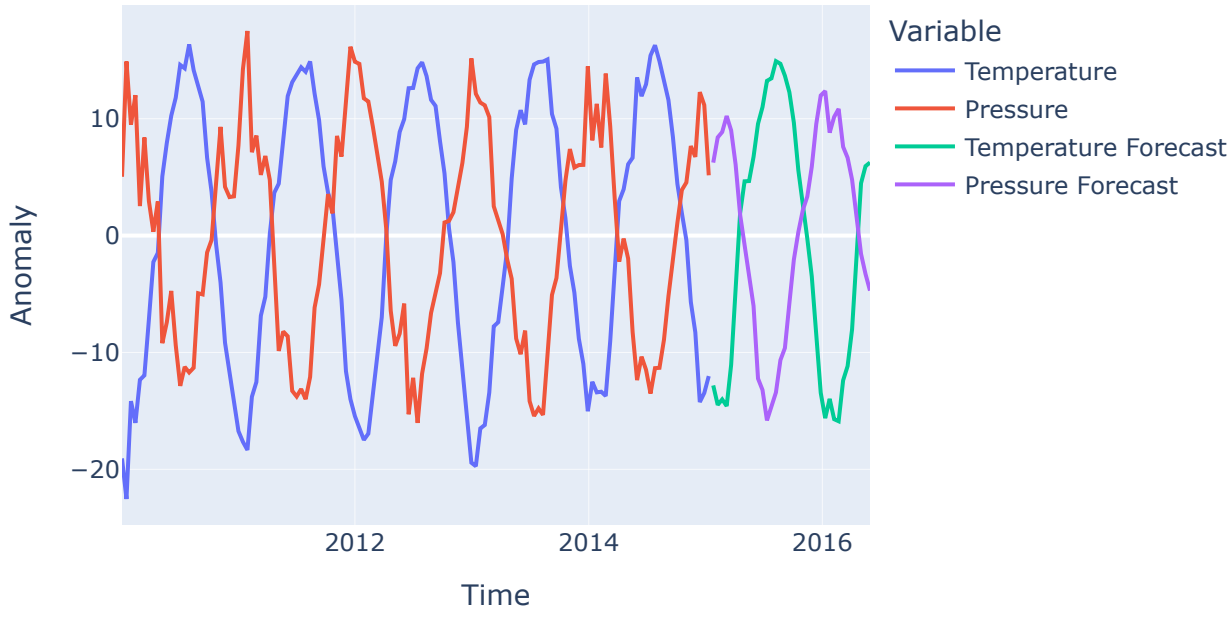
```
Epoch 140. Loss = 0.80
```

```
Epoch 160. Loss = 0.32
```

```
Epoch 180. Loss = 0.25
```

```
Epoch 200. Loss = 0.18
```

```
# Make predictions
forecast_index = create_forecast_index(df.index[-1], 36, freq="2W")
input_data = torch.tensor(X_train[-1, :], dtype=torch.float32).unsqueeze(0)
cnn_multi = pd.DataFrame(recursive_CNN_forecast(input_data, model, n=36, respo
                         columns=["Temperature Forecast", "Pressure Forecast"]
                         index=forecast_index)
# Plot
px.line(pd.concat((df.melt(var_name="Variable", value_name="Anomaly", ignore_i
                   cnn_multi.melt(var_name="Variable", value_name="Anomaly", i
        y="Anomaly", color="Variable", width=650)
```



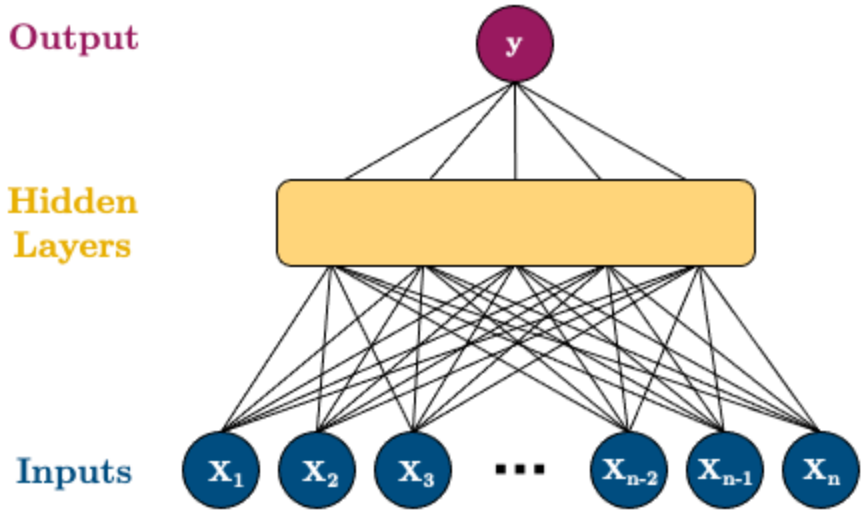
1.3. Recurrent neural networks

- So we can get reasonable results with CNNs because we preserve the structure of the data.
- CNNs extract patterns based on looking at local groups of values - a filter of some size is passed over the data to extract patterns. Those patterns are used to help make predictions. Think back to when we were working with image data, each filter looks at a small group of pixels each time and extracts some information from them, and then moves on.
- But one thing we're missing is memory. What I mean by that, is that our CNN doesn't really think about how influential values far in the past should be relative to values in the

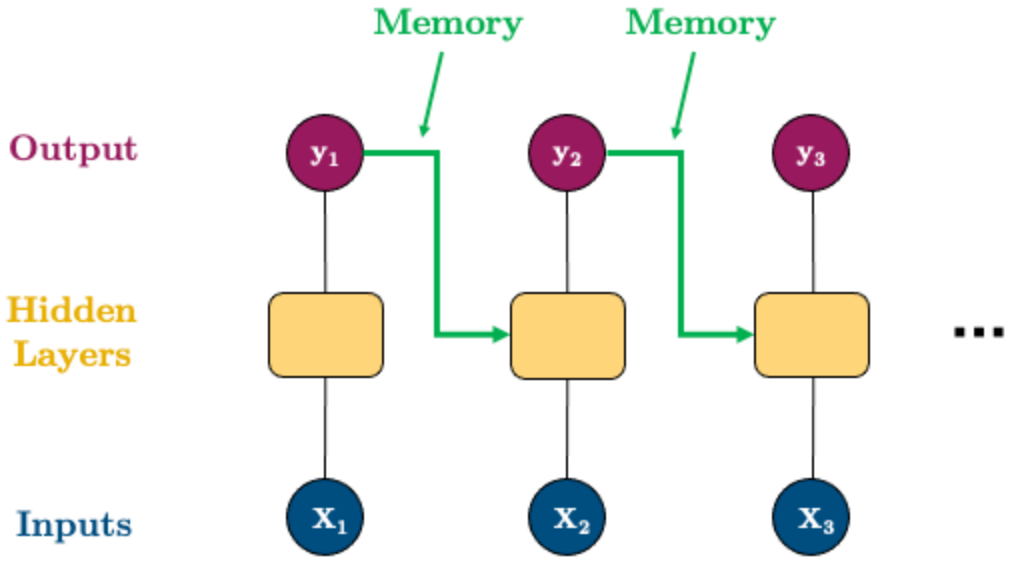
recent past.

- This is where recurrent neural networks come in! You'll be learning about RNNs in detail in 575, so the goal here is to give you a high-level overview of how they work and how they relate to the previous architectures and time series.

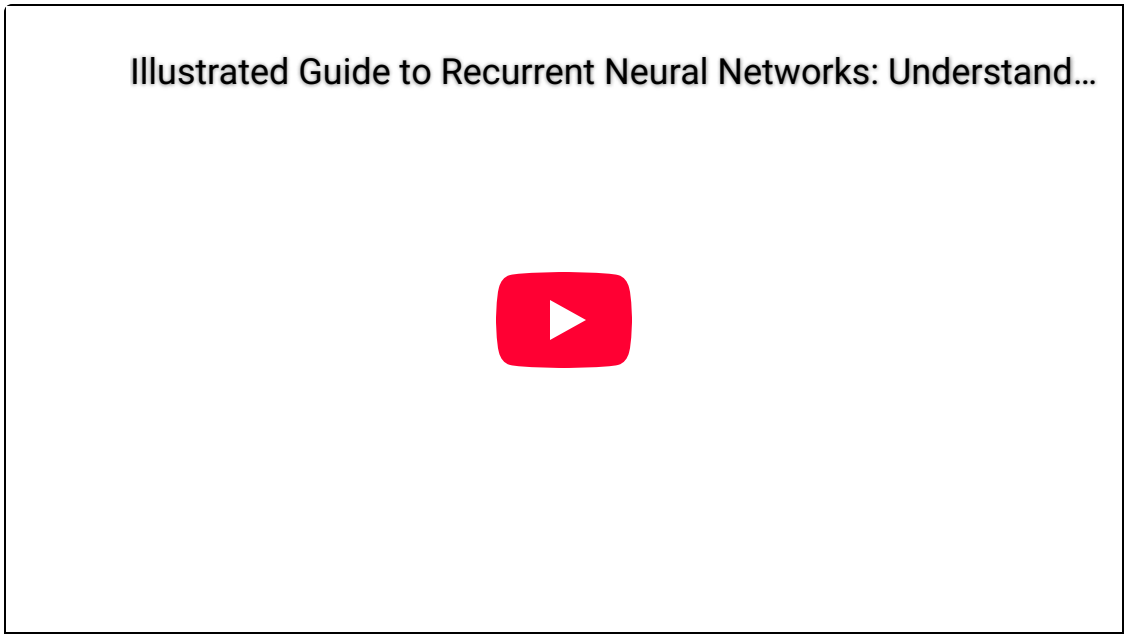
- Let's start with a regular neural network:



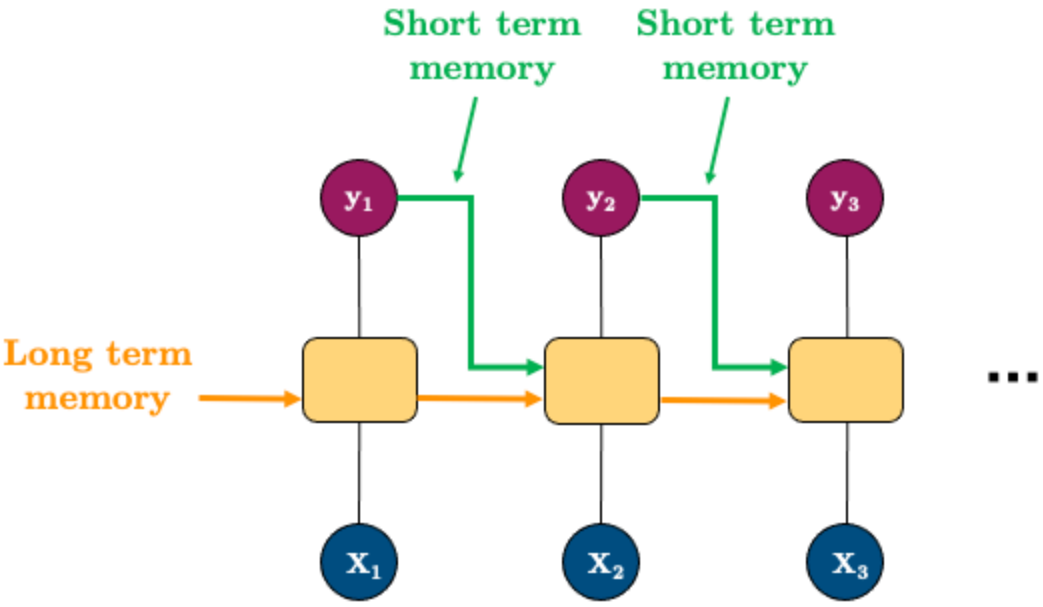
- The problem with the above is that we have no memory or notion of temporal dependence - if we swap the order of the inputs, nothing would change.
- Instead, let's split this up and process one time-step at a time (i.e., sequentially). We can then feed the output of each time-step into the next as "memory":



```
from IPython.display import HTML
# Youtube
HTML('<iframe width="560" height="315" src="https://www.youtube.com/embed/LHXX
```



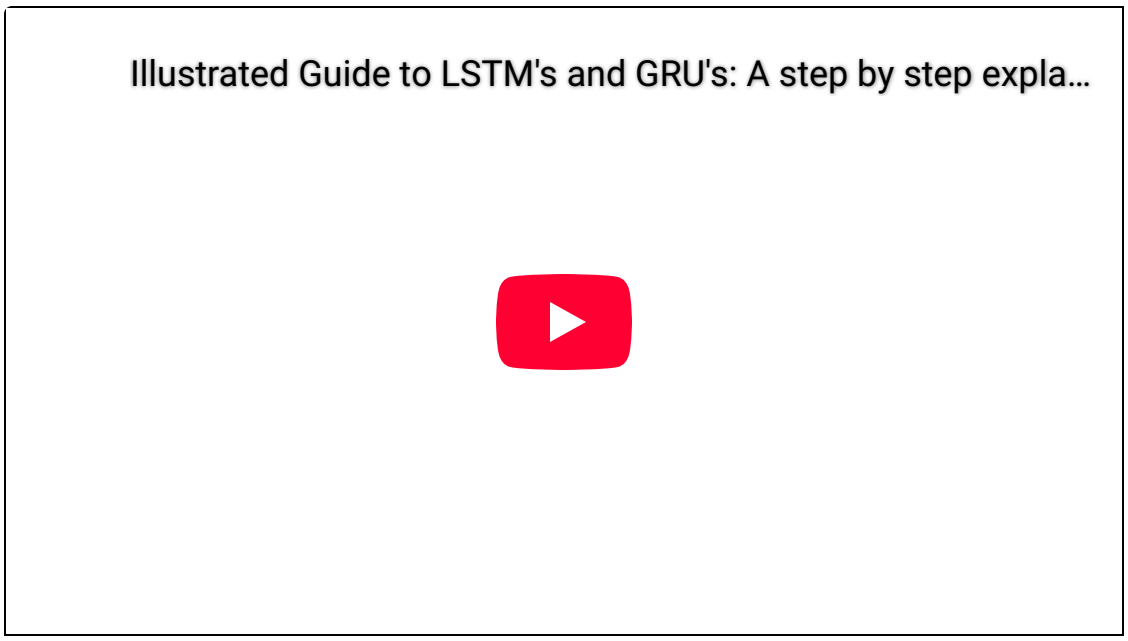
- That's an RNN!
- The problem with RNN's is that they forget things quickly. That's where LSTMs come in, they add an additional "long-term" memory component to the above framework:



- Things are starting to look complicated but it's not too bad!
- All those yellow boxes, which we'll now call "LSTM cells" are identical. They share the same weights and the same architecture

- We don't need to go into the details of how they work, the core idea is to combine the current input with the long-term and short-term memory to provide an output, a new short-term memory, and update the long-term memory

```
from IPython.display import HTML
# Youtube
HTML('<iframe width="560" height="315" src="https://www.youtube.com/embed/8HyC
```



- The architecture of an LSTM cell above is fairly arbitrary - it's been found to work well in practice but there are plenty of variants of it. A popular one is GRU.
- Recall with CNNs that we found a lot of utility in using models that others had created. You can do the same thing with LSTMs as I'll show later in the lecture. But just for fun, let's create a simple LSTM from scratch now with `pytorch` because I want you to see that you can do it! It's a little more involved, but I'll try to make it as simple as possible:

```
class LSTM(nn.Module):
    def __init__(self, input_size, output_size, hidden_dim):
        super().__init__()

        self.hidden_dim = hidden_dim
        self.lstm = nn.LSTM(input_size, hidden_dim, batch_first=True, num_laye
        self.fc = nn.Linear(hidden_dim, output_size)

    def forward(self, x, hidden):
        # x :           (batch_size, seq_length, input_size)
        # hidden :      (short_mem, long_mem), each (n_layers, batch_size, hid
        # output :  (batch_size, seq_length, input_size)
        # note that the output will have the same shape as the input because
        # it makes a forecast for each time step in the sequence
        # but we only care about the last prediction (the forecast after the s
        # so I'll only take the last value of the output

        prediction, (short_mem, long_mem) = self.lstm(x, hidden)
        output = self.fc(prediction)
        return output[:, -1, :], short_mem, long_mem
```

```
SEQUENCE_LENGTH = 36
BATCH_SIZE = 16
dataloader, scaled_data, scaler = prepare_dataloader(train, SEQUENCE_LENGTH, B
# example batch
X, y = next(iter(dataloader))
print(f"X shape: {X.shape}")
print(f"y shape: {y.shape}")
```

```
X shape: torch.Size([16, 36, 1])
y shape: torch.Size([16])
```

```
# instantiate model framework
model = LSTM(1, 1, 51)
optimizer = optim.Adam(model.parameters())
criterion = nn.MSELoss()
# train
EPOCHS = 300
for epoch in range(1, EPOCHS + 1):
    for X, y in dataloader:
        memory = None
        optimizer.zero_grad()
        y_hat, short_mem, long_mem = model(X, memory)
        loss = criterion(y_hat.flatten(), y)
        loss.backward()
        optimizer.step()
    if epoch % 20 == 0: print(f"Epoch {epoch:03}. Loss = {loss.item():.4f}")
```

```
Epoch 020. Loss = 0.0733
```

```
Epoch 040. Loss = 0.0651
```

```
Epoch 060. Loss = 0.0539
```

```
Epoch 080. Loss = 0.0397
```

```
Epoch 100. Loss = 0.0360
```

```
Epoch 120. Loss = 0.0392
```

```
Epoch 140. Loss = 0.0308
```

```
Epoch 160. Loss = 0.0269
```

```
Epoch 180. Loss = 0.0320
```

```
Epoch 200. Loss = 0.0186
```

```
Epoch 220. Loss = 0.0189
```

```
Epoch 240. Loss = 0.0166
```

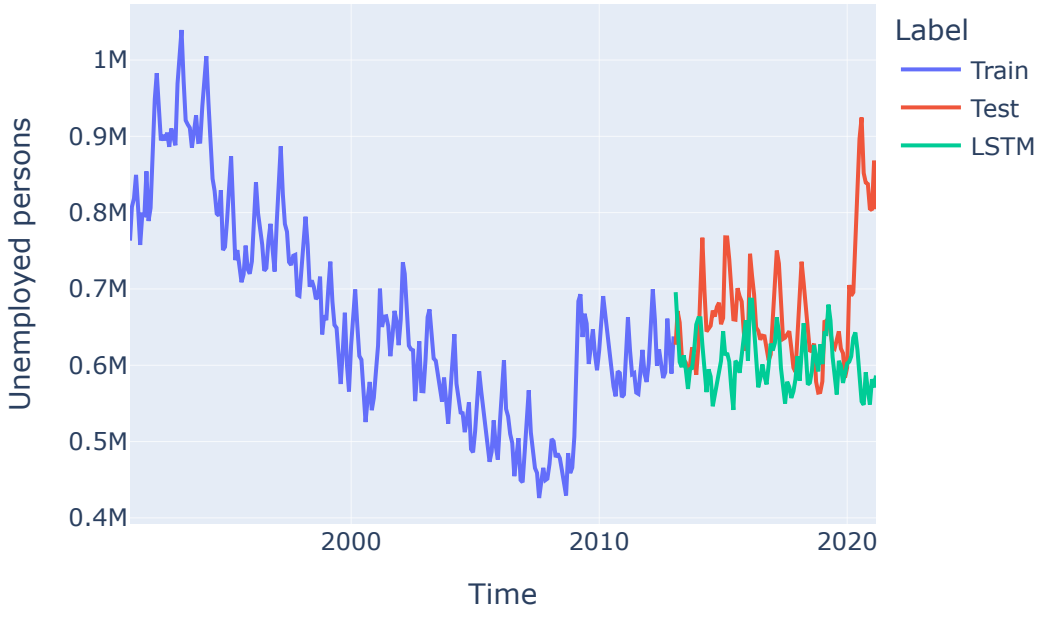
```
Epoch 260. Loss = 0.0155
```

```
Epoch 280. Loss = 0.0141
```

```
Epoch 300. Loss = 0.0171
```

```
def recursive_LSTM_forecast(input_data, model, memory, n=20, no_memory=False):
    forecast = np.empty(n)
    for i, n in enumerate(range(n)):
        with torch.no_grad():
            output, short_mem, long_mem = model(input_data, memory)
            if no_memory:
                memory = None
            else:
                memory = (short_mem.data, long_mem.data)
            forecast[i] = output.detach().numpy()
        input_data = torch.cat((torch.tensor([[[forecast[i]]]], dtype=torch.fl
                                input_data[:, :-1, :]), 1)
    return forecast
```

```
input_data = torch.tensor(scaled_data[-1:, 1:], dtype=torch.float32).unsqueeze
forecasts = recursive_LSTM_forecast(input_data, model, memory, len(test))
forecasts = scaler.inverse_transform(forecasts.reshape(-1, 1))
lstm = pd.DataFrame(
    {
        "Unemployed persons": forecasts.flatten(),
        "Label": "LSTM",
    },
    index=test.index,
)

px.line(pd.concat((train, test, lstm)), y="Unemployed persons", color="Label")
```



- Above, I'm implementing a "stateless" LSTM
- I'm not remembering the "memory" between batches as I train, I believe all necessary time-dependent information is in the sequence of each example
- For my data, this makes sense, because probably most of the information needed to make a prediction is in the last 36 months
- But what if I used a much shorter time series? In that case, I probably want the states to persist between batches, a "stateful" LSTM.

2. Useful time series packages

2.1. Prophet

- Facebook's Prophet is another popular model for time series forecasting
- It's actually not that complex of model, it basically fits non-linear regressions at various seasonalities (you can read the paper here). I like to think of it as a bit of a Frankenstein of classical time series methods we've seen like decomposition, regresion, exponential smoothing, etc.
- Prophet can be useful for modelling data with complex/multiple seasonality, changing trends and holiday effects, and aims to automate and speed up model fitting as much as possible to help with forecasting data at scale.
- Prophet was originally developed for modelling daily data. It has since expanded to include longer periods (weekly, monthly, yearly, etc.). I haven't found Prophet to be overly useful in my work, but I also don't typically deal with the kind of data it was really suited for. There's plenty of comparisions out there of Prophet vs ARIMA vs ETS vs machine learning - but as always, there's no free lunch.
- Anyway, let's give it a quick go, Prophet requires a dataframe with two columns:
    1. "ds": the "datestamps"
    2. "y": the time series values

```
from prophet import Prophet
```

```
prophet_df = pd.DataFrame({"ds": train.index,
                           "y": train["Unemployed persons"].to_numpy()}, index
prophet_df.head()
```

	ds	y
0	1991-01-31	763321.6342
1	1991-02-28	807546.6346
2	1991-03-31	818667.6259
3	1991-04-30	849493.4822
4	1991-05-31	805407.9362


```
prophet_df = pd.DataFrame({"ds": train.index,
                           "y": train["Unemployed persons"].to_numpy()}, index
prophet_df.head()
```

	ds	y
0	1991-01-31	763321.6342
1	1991-02-28	807546.6346
2	1991-03-31	818667.6259
3	1991-04-30	849493.4822
4	1991-05-31	805407.9362

- We can make an instance of the `Prophet()` class and then `.fit()` it, similar to the `sklearn` API:

```
model = Prophet(interval_width=0.95)
model.fit(prophet_df);
```

```
12:58:17 - cmdstanpy - INFO - Chain [1] start processing
```

```
12:58:17 - cmdstanpy - INFO - Chain [1] done processing
```

- Let's make a dataframe of times to forecast over using the `.make_future_dataframe()` method:

```
test_dates = model.make_future_dataframe(periods=len(test), freq="M", include_
test_dates.head()
```

	ds
0	2013-01-31
1	2013-02-28
2	2013-03-31
3	2013-04-30
4	2013-05-31

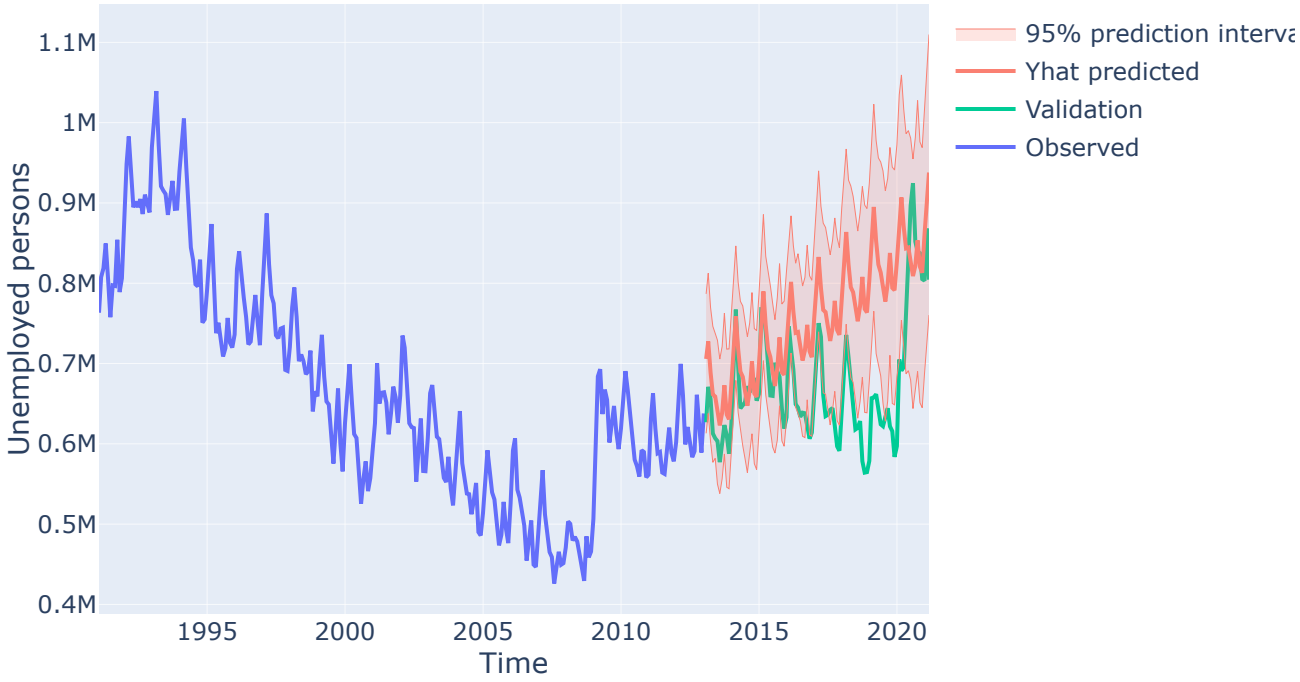
- And finally, we can make our forecasts:

```
forecasts = model.predict(test_dates)
forecasts.head()
```

	ds	trend	yhat_lower	yhat_upper	trend_lower	tre
0	2013-01-31	646252.707343	613119.982272	786742.651706	646178.009510	6462
1	2013-02-28	648268.268396	637885.172944	812506.730136	647933.913766	6485
2	2013-03-31	650499.782418	602632.594777	769526.073353	649777.953687	6511
3	2013-04-30	652659.312118	576951.174358	746331.371513	651340.621875	6537
4	2013-05-31	654890.826141	581941.739283	737643.636599	652954.727412	6563

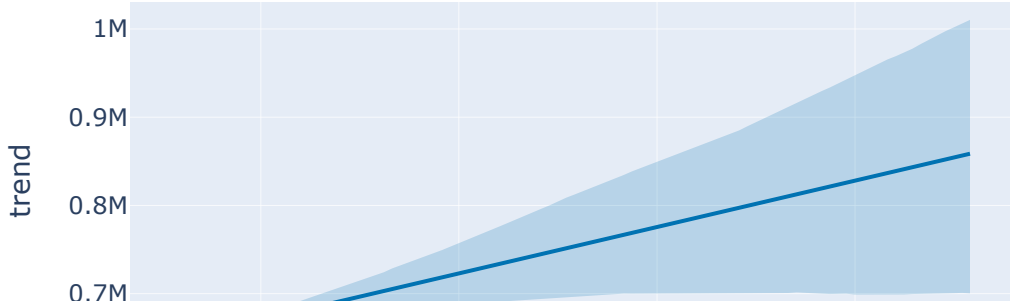
```
```

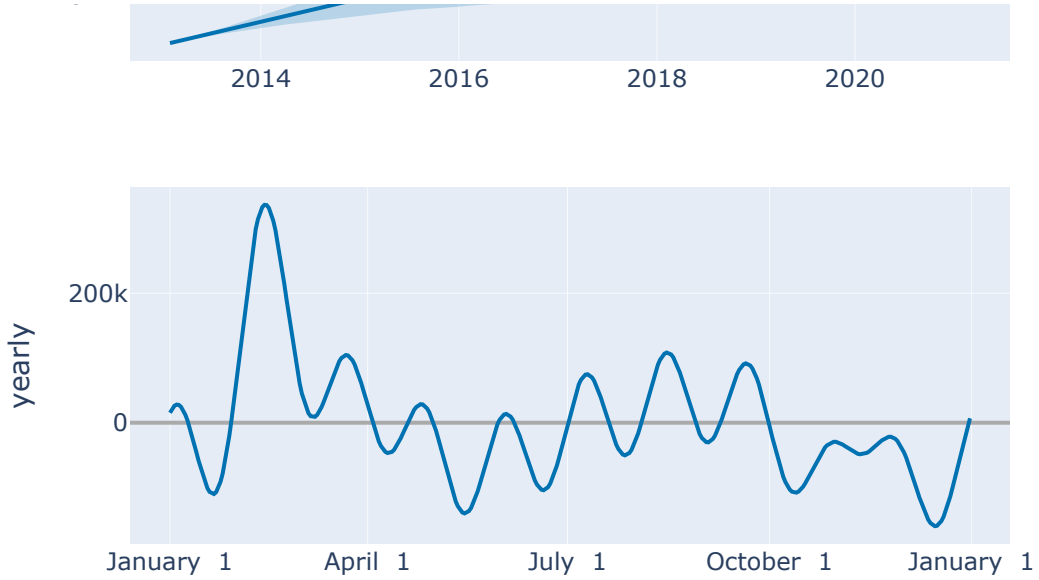
```
prophet = (
    forecasts.loc[:, ["yhat", "yhat_lower", "yhat_upper"]]
    .rename(columns={"yhat_lower": "pi_lower", "yhat_upper": "pi_upper"})
    .set_index(test.index)
)
plot_prediction_intervals(
    train["Unemployed persons"], prophet, "yhat", valid=test["Unemployed perso
)
```



- We can also plot each of the modelled components:

```
from prophet.plot import plot_components_plotly
plot_components_plotly(model, forecasts, figsize=(600, 300))
```





2.2. Machine-learning based

2.1.1. Gluon-TS

- GluonTS is a time series package built on the MXNet deep-learning framework (an alternative to `pytorch`)
- It has a simple and easy to use Python API with plenty of ready-to-train state of the are deep learning models
- Other popular models available in `gluonts` include:
  - DeepAR
  - DeepVAR
  - GPVAR
  - DeepState
  - LSTNet
  - N-Beats

2.1.2. PyTorch Forecasting

- pytorch-forecasting is a relatively new package released late 2020 that aims to provide a high-level API for neural network based time series modelling
- I haven't used it myself and it remains under active development but I'll be keeping an eye on it
- Comparing to `gluonts`, they claim that `pytorch-forecasting` has the advantage of being built on `pytorch` which is more popular than `MXNet` and that the interface will be simpler than `gluonts` - we'll see!

2.1.3. Other

- Two other popular libraries for time series forecasting with machine learning and an `sklearn` interface:
  1. sktime
  2. tslearn
- But there are plenty of other libraries out there too
- I prefer to do a lot of my non deep-learning machine learning work manually and transparently, so I tend not to use these libraries myself, but it's important to know they exist!

2.3. R

- If you prefer R, there's plenty of time series functionality to go round!
- Time series packages in R have recently been getting a huge upgrade in the form of the tidyverts ecosystem
- As the name suggests, this ecosystem is made to work nicely with tibbles and the tidyverse!
- The `tidyverts` ecosystem is under active development, but it contains all the same time series functionality we've been using in `statsmodels` and other packages

3. Additional topics

- **Heirachical/grouped time series**:
  - Time series that have multiple levels. For example we may wish to model unemployment in Australia on a per-country, and per-state level (Australia has 6 states and 2 territories)
  - Bottom-up approach: model the lowest level first (per-state unemployment) and then sum to produce higher levels (country unemployment)
  - Top-down approach: model the highest level first (country unemployment) and then proportion down to lower levels (per-state unemployment)
- **Multiple seasonality**:
  - Decompose/model independently
  - Decompose/model simultaneously using e.g., prophet, or statsmodels
- **Multivariate series**:
  - We actually saw how to easily forecast multivariate series in Lecture 4 and earlier this lecture using machine learning
  - The idea is to allow the series to interact and affect each others' values
  - Another popular model for this, which is a generalisation of autoregression is Vector Autoregression
- **Explanatory variables**:
  - Very easy to add a features to machine learning models
  - ARIMA models also support this, usually in the form of "regression with ARIMA errors", which basically means, fit a linear regression to your data and then model the residual with an ARIMA model. Try it out by using the `exog` parameters of `statsmodels` `ARIMA()` class.
- **Time series classification**:
  - I usually go for a machine learning approach here
  - There are other options too, like hidden Markov models (which you learned about in 575), you can also try Googling: "discrete time series forecasting"

4. Summary

Key takeaways:

- Time series data is not independent, it contains temporal dependence. The main patterns we look for are:

```
- Trend
- Seasonality
- Cyclicity
- Short-term autocorrelation
```

- The temporal patterns in a series help us choose suitable models. There are 4 main classes of models we can use to model time series:

```
- Baseline models (naive, average, drift, etc.)
- Exponential smoothing models
- ARIMA
- Machine learning
```

Now, as an in-class exercise, pair up with some people around you and compare these methods and discuss some of their advantages and disadvantages.

> Each of the above classes model time series in different ways. There's no free lunch, but ensembles (combinations) of different models can work well. The statistical models remain popular in industry because of their speed, ease-of-use, and interpretibility. The machine learning approaches take much more effort to build, maintain, and forecast with - but they show potential!

A good workflow when presented with a new time series:

- Understand it. Plot it. Decompose it. What signals are present? What models might be suitable here?
- Split into train, validation, and test sets (or just train and test if you intend on doing cross-validation in the training set).
- Baseline and statistical models are the first stop. ETS and ARIMA are the workhorses of time series modelling, but don't discount the potential of basline methods!
- If you're looking for more performance, delve into machine learning. In particular, this allows us to easily and effectively incorporate explanatory variables and model highly complex functions.
- If you've got the time and the patience, say hello to CNNs, RNNs and LSTMs. They are complex, and take a long time to develop, but they show great potential. To me these

methods try to interpret the time series for you, they engineer the features and identify the seasonality. It may be difficult to build neural networks that consistently outperform the classic statistical methods. But in saying that, they offer the ability to get good predictive performance without expert knowledge of the field or data you are forecasting.

> Always remember. We are forecasting the future - nobody knows what's going to happen. In that spirit, I prefer to err on the side of parsimony (simplicity).