Data Mining Project

Master's degree in Data Science

UNIVERSITÀ DEGLI STUDI
DI NAPOLI FEDERICO II

FATER Business Game
Engine Optimization

Andrea Franzone - P37000025
Alessandro Santonicola - P37000039
Olesia Nitsovych – P37000044

2020/2021

Table of Contents

1. Introduction

The Campochiaro plant is powered by four internal motors plus some external energy source that is eventually purchased incurring in extra costs. Every day the surplus energy gets wasted leading us to an optimization problem: considering the four motors available at FATER's plant, their best usage configuration should be found in order to minimize the costs and the waste of energy , in three specific dates (19-11-2019, 16-01-2020 and 18-06-2020).To do so, different datasets were provided, most of them describe the power consumption over the days with a minute-by-minute granularity, while another dataset takes into account the scheduled productions for each type of item and their delivered quantity over the days. Other information regarding the cost of the motors, external energy, minimum and maximum energy produced by each motor are provided as well.

2. The Dataset

Specifically, FATER company's group provided us with 5 datasets. Four of them are pretty similar, in fact they share the same variables:

- Time (identifier variable)
- Power (quantitative variable)

These datasets are grouped by years, and they cover the years range that goes from 07-2016 to 06-2020. Below is reported an overview of such datasets:

	Time	Power
0	01/07/2016 00:00	0.0
1	01/07/2016 00:00	0.0
2	01/07/2016 00:01	0.0
3	01/07/2016 00:01	0.0
4	01/07/2016 00:02	0.0
...	...	...
1017331	30/06/2017 23:57	2568.0
1017332	30/06/2017 23:58	2737.0
1017333	30/06/2017 23:58	2524.0
1017334	30/06/2017 23:59	2479.0
1017335	30/06/2017 23:59	2530.0

Figure 2-4: 2016-2017 data

	Time	Power
0	01/07/2017 00:00	2727.0
1	01/07/2017 00:00	2588.0
2	01/07/2017 00:01	2555.0
3	01/07/2017 00:01	2699.0
4	01/07/2017 00:02	2471.0
...	...	...
1016759	30/06/2018 23:57	2703.0
1016760	30/06/2018 23:58	2602.0
1016761	30/06/2018 23:58	2594.0
1016762	30/06/2018 23:59	2644.0
1016763	30/06/2018 23:59	2583.0

Figure 2-4: 2017-2018 data

	Time	Power
0	01/07/2018 00:00	2551.0
1	01/07/2018 00:00	2562.0
2	01/07/2018 00:01	2641.0
3	01/07/2018 00:01	2597.0
4	01/07/2018 00:02	2580.0
...	...	...
1020445	30/06/2019 23:57	625.0
1020446	30/06/2019 23:58	589.0
1020447	30/06/2019 23:58	625.0
1020448	30/06/2019 23:59	601.0
1020449	30/06/2019 23:59	601.0

Figure 2-4: 2018-2019 data

	Time	Power
0	01/07/2019 00:00	601.0
1	01/07/2019 00:00	601.0
2	01/07/2019 00:01	625.0
3	01/07/2019 00:01	577.0
4	01/07/2019 00:02	577.0
...	...	...
998545	30/06/2020 23:57	2490.0
998546	30/06/2020 23:58	2490.0
998547	30/06/2020 23:58	2490.0
998548	30/06/2020 23:59	2490.0
998549	30/06/2020 23:59	2490.0

Figure 2-4: 2019-2020 data

As it can be seen by the above images, for each day of the related year two measures of the power consumption are granted by sensors for each minute. Given this data's structure, it can comfortably be said that we will be facing time series analysis, since we have observations of the same process that change over time. The last of those datasets concerns the produced goods:

Name of the variable	Description	Unit of measure
start date	Starting date of the production process	yy/mm/dd
start time	Starting time of the production process	hh:mm:ss
stop date	Ending date of the production process	yy/mm/dd
stop time	Ending time of the production process	hh:mm:ss
material Code	Item identifier	dimensionless
description	Item's description	dimensionless
scheduled quantity	Quantity of goods that need to be produced	dimensionless
delivered quantity	Quantity of goods that need to be delivered	dimensionless

Table 2.1: Production data

As anticipated before, there are two different types of motors with the following characteristics:

	Type of motor	
Variables	Caterpillar Motors	Jenbacher Motors
Max power	1656kWh	1501kWh
Min power	828kWh	747kWh
Cost	0.070744 € / kWh	0.071448 € / kWh
maintenance cost	15€ / h	0€ / h

Table 2.2: Motors specifics

* Outsourcing energy has a cost, and it amounts to 0.15 € / kWh.

3. Exploratory Data Analysis

In this section we will provide a detailed explanation of the given datasets, in order to gain some useful insight regarding the process at hand.

3.1 Data Preprocessing

To efficiently use the data granted by FATER, some preprocessing operations are needed.

Some data points have been previously transformed to provide an efficient loading since there could have been some type of mistakes in the acquiring of some measures by the sensors used (presence of double apex " " where not needed). Once data have been loaded successfully, the column "Time" has been cast into "datetime" type to have a feasible index for future references.

After all those operations the structure of the loaded data frames can be seen below :

Time	Power
1904-01-01 01:00:00	0.0
2016-07-01 00:00:00	0.0
2016-07-01 00:01:00	0.0
2016-07-01 00:02:00	0.0
2016-07-01 00:03:00	0.0
...	...
2017-06-30 23:55:00	2642.5
2017-06-30 23:56:00	2426.0
2017-06-30 23:57:00	2446.0
2017-06-30 23:58:00	2630.5
2017-06-30 23:59:00	2504.5

Figure 3-4: 2016-2017 data

Time	Power
1904-01-01 01:00:00	0.0
2017-07-01 00:00:00	2657.5
2017-07-01 00:01:00	2627.0
2017-07-01 00:02:00	2402.0
2017-07-01 00:03:00	2461.5
...	...
2018-06-30 23:55:00	2618.5
2018-06-30 23:56:00	2630.5
2018-06-30 23:57:00	2651.5
2018-06-30 23:58:00	2598.0
2018-06-30 23:59:00	2613.5

Figure 3-3: 2017-2018 data

Time	Power
2018-07-01 00:00:00	2556.5
2018-07-01 00:01:00	2619.0
2018-07-01 00:02:00	2609.0
2018-07-01 00:03:00	2606.0
2018-07-01 00:04:00	2611.0
...	...
2019-06-30 23:55:00	613.0
2019-06-30 23:56:00	607.0
2019-06-30 23:57:00	619.0
2019-06-30 23:58:00	607.0
2019-06-30 23:59:00	601.0

Figure 3-2: 2018-2019 data

Time	Power
1904-01-01 01:00:00	0.0
2019-07-01 00:00:00	601.0
2019-07-01 00:01:00	601.0
2019-07-01 00:02:00	577.0
2019-07-01 00:03:00	589.0
...	...
2020-06-30 23:55:00	2490.0
2020-06-30 23:56:00	2490.0
2020-06-30 23:57:00	2490.0
2020-06-30 23:58:00	2490.0
2020-06-30 23:59:00	2490.0

Figure 3-1: 2019-2020 data

There are some incongruencies regarding both the variables. Especially, there is a wrong date (1904-01-01) in 3 out of 4 data frames. Since this is a clear inconsistency, which can also be noticed by a null value for the "Power" variable, it has been dropped.
After that, data frames have been plotted:

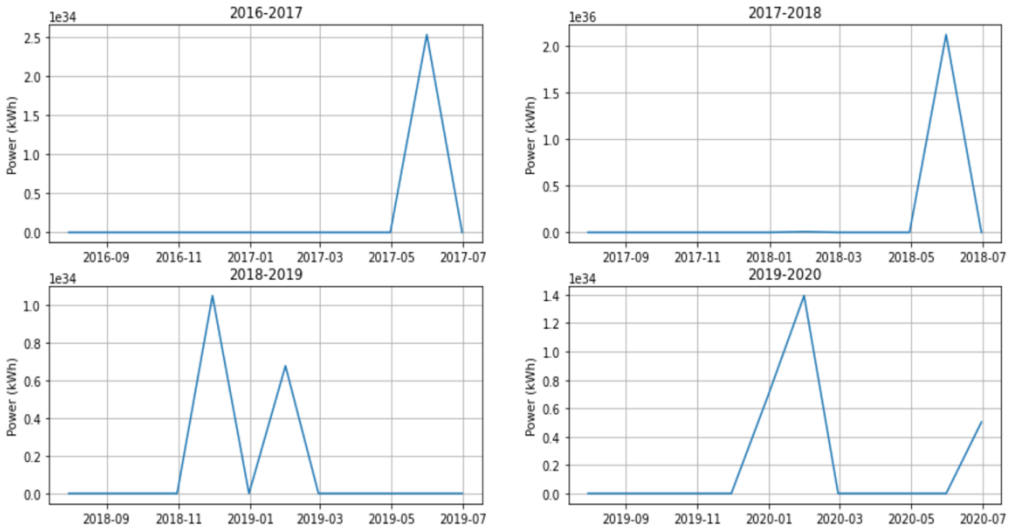


Figure 3-5: Power consumption Trend across 4 years

Outliers can be detected in the plots. The variable "Power" shows not physically feasible values in the order of 10^34 kWh. These observations have been imputed by considering the mean value given by their previous and following values, respectively. Once this process is completed, a more realistic behavior can be observed:

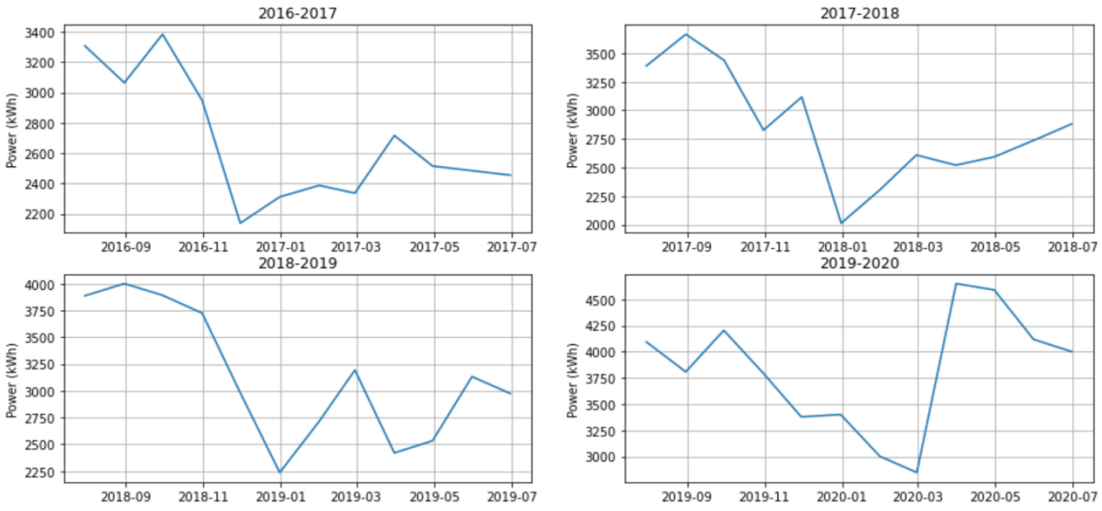


Figure 3-6: Power consumption "cleaned" over 4 years

From the above plots it can be seen a quite similar behavior across the years. As a matter of fact, there is always a downward trend towards December and beginning of January (this can be related to Italian holidays), as well as for the summer period (in which this downward trend is less emphasized). By a general overview, it can be noticed that the power consumption is increasing over the years. This could be due to an increase in the company's sales over time.

Now we can focus just on the year of interest for our challenge, that is 2019-2020. It can be seen a very high increment in term of power consumption around 2020-03, with respect to the other years. This can be since in that period, people who were constrained to be in their house by the pandemic, started buying products for personal care and house cleaning in a massive way. Furthermore, a high trend is held by the company for the second part of the working year (2020-05/06/07).

It has then deeply analyzed the time range for the years 2019-2020, with a minute-by-minute plot:

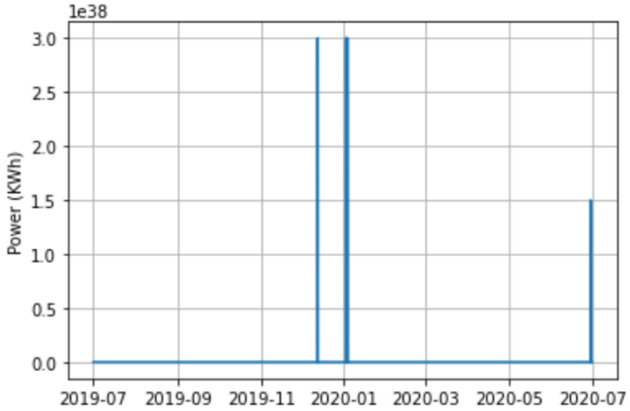


Figure 3-7: Power's measure inconsistency

As shown by the above plot, we can find the previous described outliers for the year 2019-2020 but with a different granularity. After we imputed them with a 1-KNN mean, we can finally get the actual plot for this data:

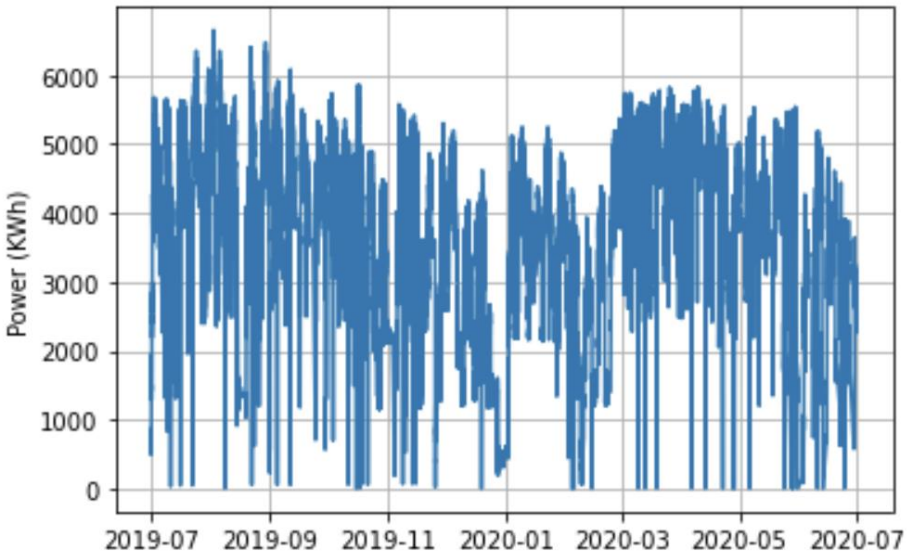


Figure 3-8: Power Consumption minute-by-minute 2019-2020

3.2 Time Series analysis (decomposition, seasonality, trend)

Once we were able to get a proper representation of the time series data of interest, we can now analyze it in detail, looking for trend, and the eventual presence of seasonal effects. Moreover, an in-depth discussion on time series analysis is reported at the end of the documentation in the theoretical appendix: Time Series Analysis

Before moving on with the practical issues of our process, it's important to know that most time series data can be described by three components: the trend, the seasonality, and the bias.
The trend is a general systematic linear or nonlinear component that changes over time and does not repeat, while the seasonality is a component that behaves in the same way as the trend, but it does repeat.
The bias (or noise) is, instead, a nonsystematic component and considers all the variations in the process that aren't explained by the trend or the seasonality.

Focusing now on our time series, *Figure 3-8*, the first thing we can do, is to check if it has a stationary behavior. This information can be gathered both by looking at the plot or, in a more statistical way, by performing the "Augmented Dickey-Fuller test (ADF)".

```
ADF Statistic: -8.954118
p-value: 0.000000
Critical Values:
        1%: -3.430
        5%: -2.862
        10%: -2.567
```

Figure 3-9: Augmented Dickey-Fuller Test Result

Given this result, we are quite confident in rejecting the null hypothesis, concluding that our time series is stationary.

The next step to perform to better analyze the time series, is to decompose it into its 3 main components which are, as described above, the trend, the seasonal and the residuals.

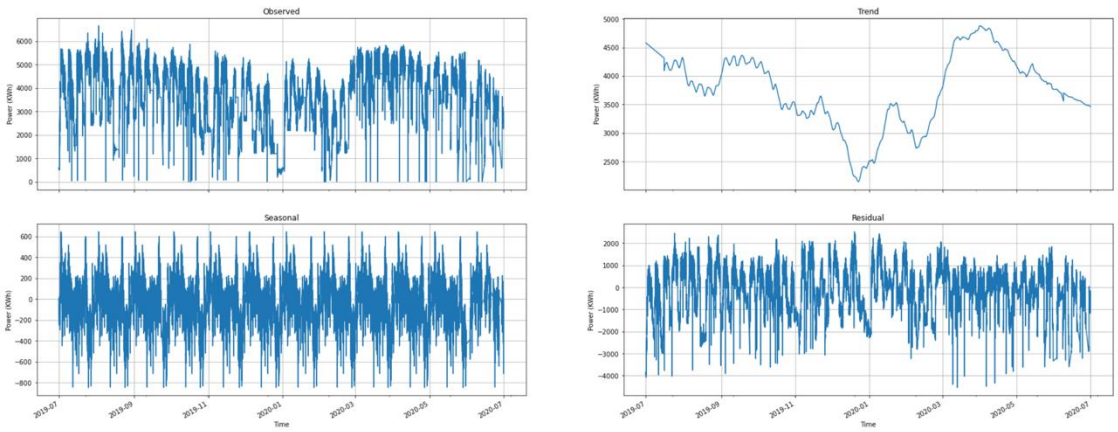


Figure 3-10: Process Decomposition. 2019-2020 minute-by-minute plot

Given the trend component, it's easy to spot a downward behavior in the first half of the time range which can be due to the holiday season. Moreover, an upward behavior in the second half can be spotted. In addition, it can be seen a certain seasonal effect by the corresponding component. Residuals have a very high variance, which could be due to the fact that we are dealing with a real process or to an improper choice of the unit measure (minute to observe a phenomenon over a year).
However, these plots are still affected by a notable amount of noise. In order to have a more "clear" and smoothed representation of our data, we can consider a more time-aggregate time series in particular, we can consider a weekly aggregation of the data.

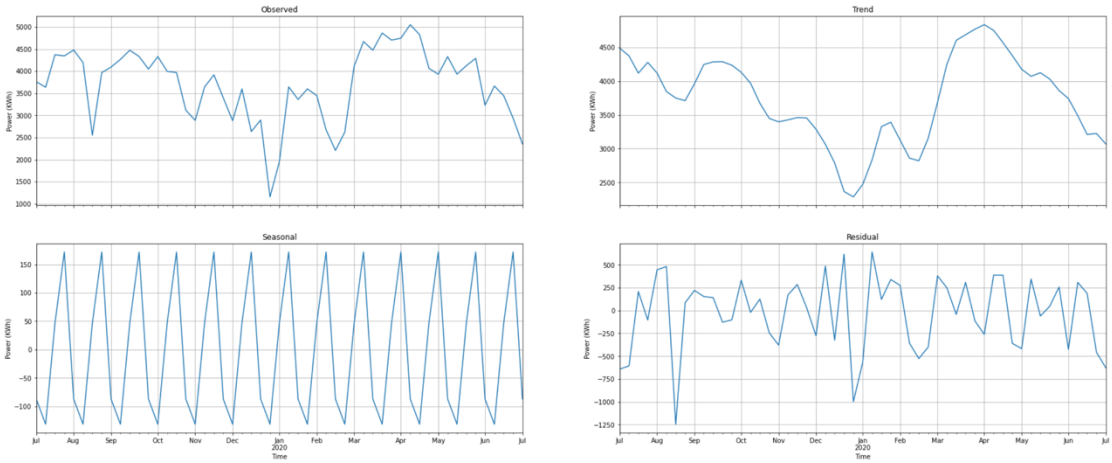


Figure 3-11: Process Decomposition. 2019-2020 weekly aggregation

From the above plots we can spot again a similar trend to the one discovered before, but smoother. In addition, we can detect a clearer seasonal effect for which Power consumption decrease at the beginning of each month and increase in the middle-end. Similar consideration on the residual can be applied to this plot.

For visualization purposes, a more concise view of the different components of the process can be observed in the plot below:

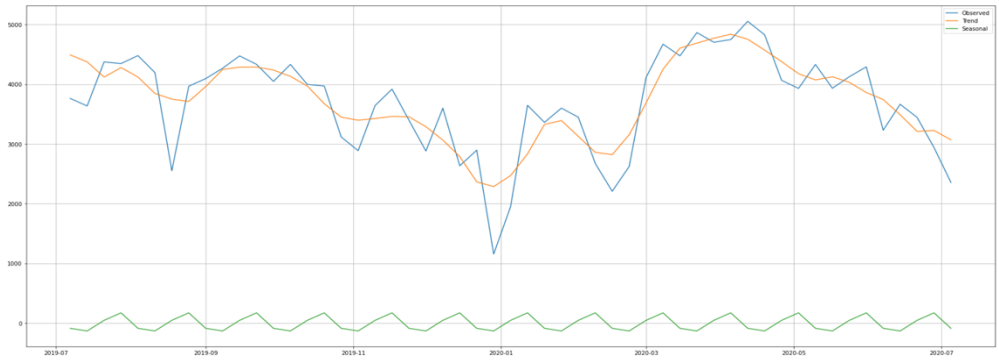


Figure 3-12: Process Decomposition, compact view (2019-2020)

For further investigation we can analyze the related autocorrelation plot. It represents the degree of similarity between a given time series and a lagged version of itself over successive time intervals. Moreover, it can help in identifying seasonality and selecting

the proper forecasting method. In particular, concerning the ARIMA class, it can be used to select the proper model for forecasting purposes.

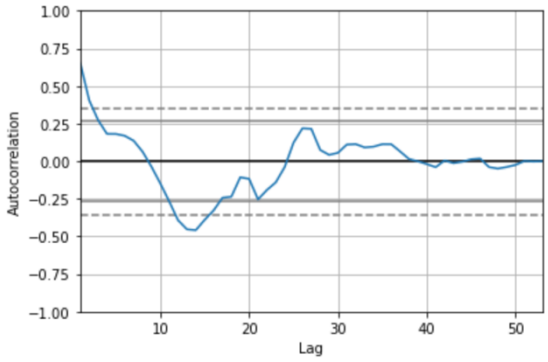


Figure 3-13: ACF on weekly aggregate data (2019-

From the plot, we have just few significant data points, (the ones that are outside the 95% confidence interval). By looking at the ACF plot, it is not apparent whether the process can be classified as an AR one or an MA one, even though there is no clear cut-off in the ACF, which leads us towards modeling our process as an AR one, there are still high values (outside the confidence interval) at lag 1, 2 that can lead us towards a MA process.

For what concerns seasonality, there does not seem to be spikes present at regular lags, which does not highlight a seasonal effect. However, since we are dealing with a real process, the seasonality cannot be easily detected from the general behavior of the process represented by the ACF plot. We could assume that, in this case, the trend and the bias components out-shadow the seasonal effect.

Since we are interested in forecasting the energy (kWh) consumption over three given days, we can now focus on a smaller timeframe, respectively the months in which these three dates fall (19-11-2019, 01-2020, 06-2020), observing the process on a day-by-day scale.

- Nov 2019

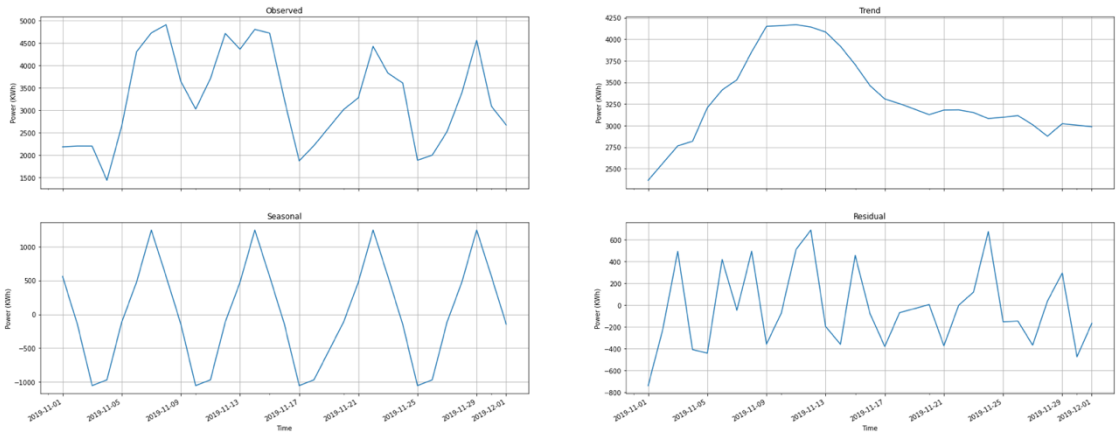


Figure 3-14: Process decomposition for time range November 2019

The above plots represent the process decomposition for November 2019. The trend keep a similar behavior with respect to the original time series data. In particular, a strong seasonal component can be noticed, characterized by a period of one week. Concerning the residual plot, it seems like it follows a stationary behavior.

For visualization purposes, a more concise view of the different components of the process can be observed in the plot below:

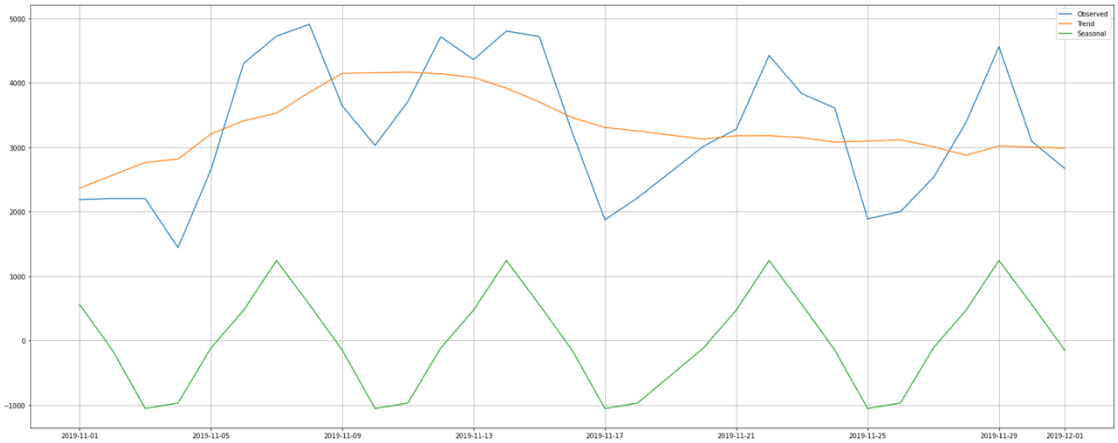


Figure 3-15: Process decomposition over November 2019 (Concise View)

For a deeper understanding of our process, we can analyze the related autocorrelation plot:

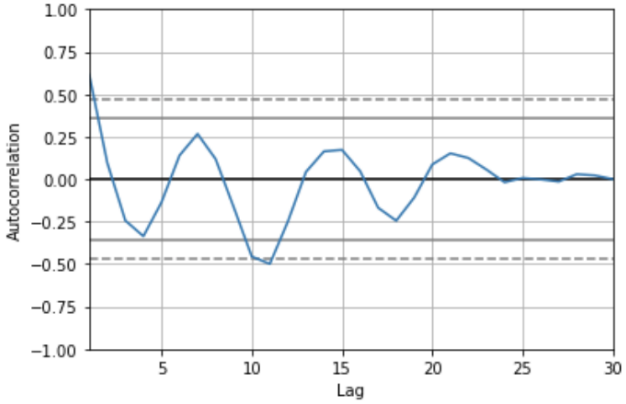


Figure 3-16: ACF plot over November 2019

There does not seem to be any significant data points (outside the confidence interval), indeed no correlation between data points at different lags can be observed. Since the process moves in a smooth way inside the confidence interval, we can assume that our process behaves like an AR process.

- Jan 2020

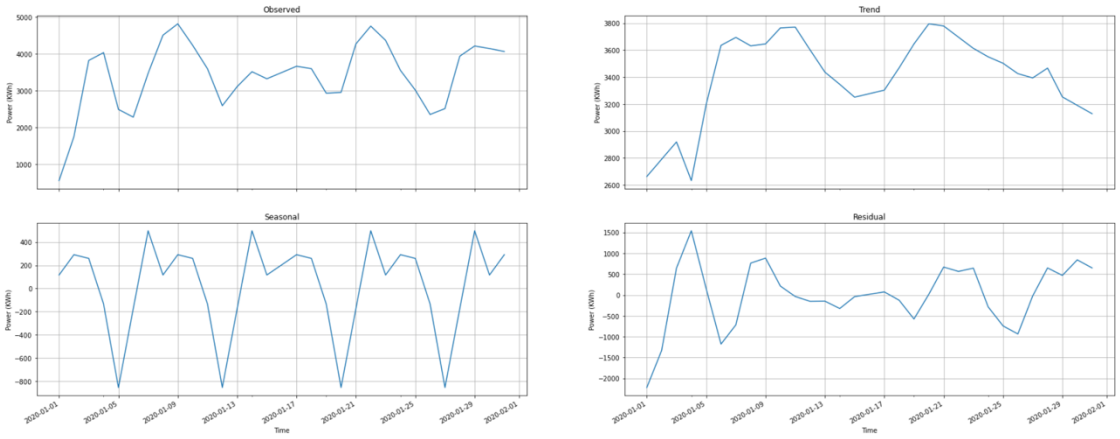


Figure 3-17: Process decomposition for time range Jan-2020

Here the process decomposition for January 2020 is illustrated. It can be spotted a similar trend with respect to the whole year analysis. In addition, it is present a strong seasonal component (it repeats itself over a span of 7 days). By observing the residual plot, a decrease in the total variation, with respect to the whole year analysis, can be noticed.

For visualization purposes, a more concise view of the different components of the process can be observed in the plot below:

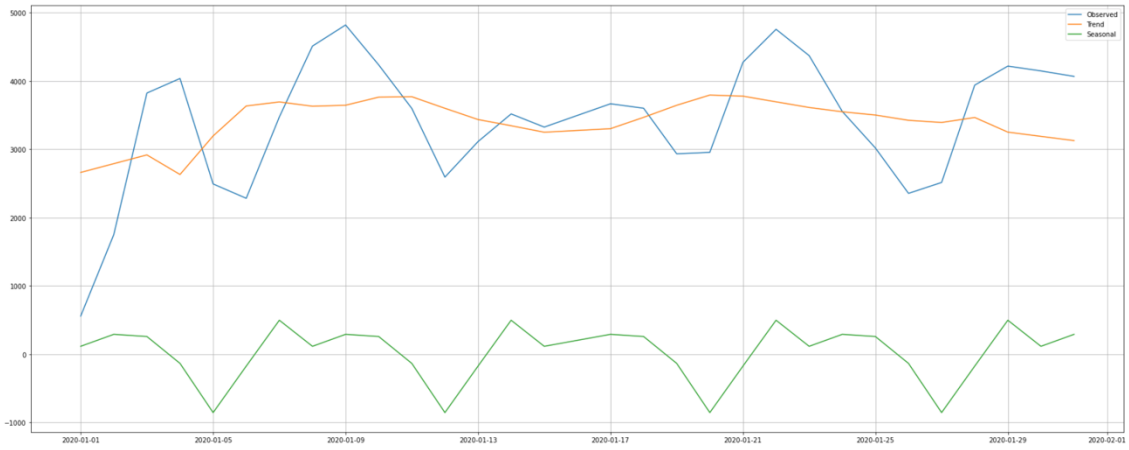


Figure 3-18: Process decomposition over Jen 2020 (concise view)

For further investigation we can analyze the related autocorrelation plot which is shown below:

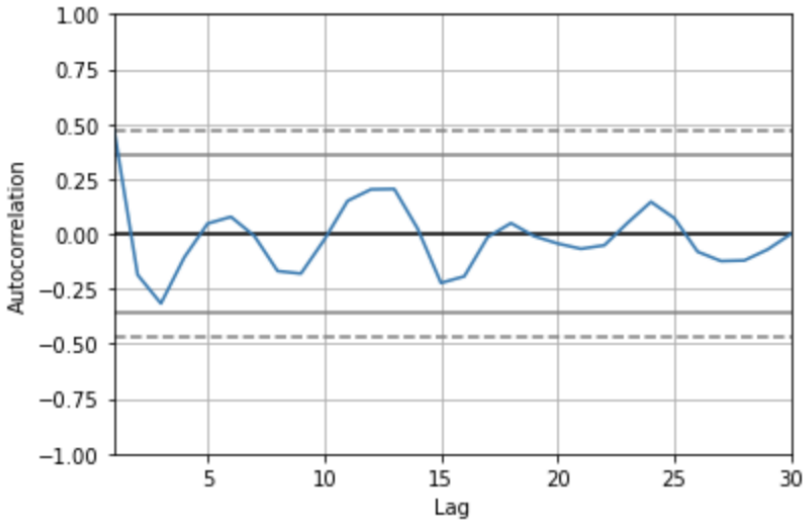


Figure 3-19: ACF plot over Jen 2020

There does not seem to be any significant data points (outside the confidence interval), indeed no correlation between data points at different lags can be observed. Since the process moves in a smooth way inside the confidence interval, we can assume that our process behaves like an AR process.

- June 2020

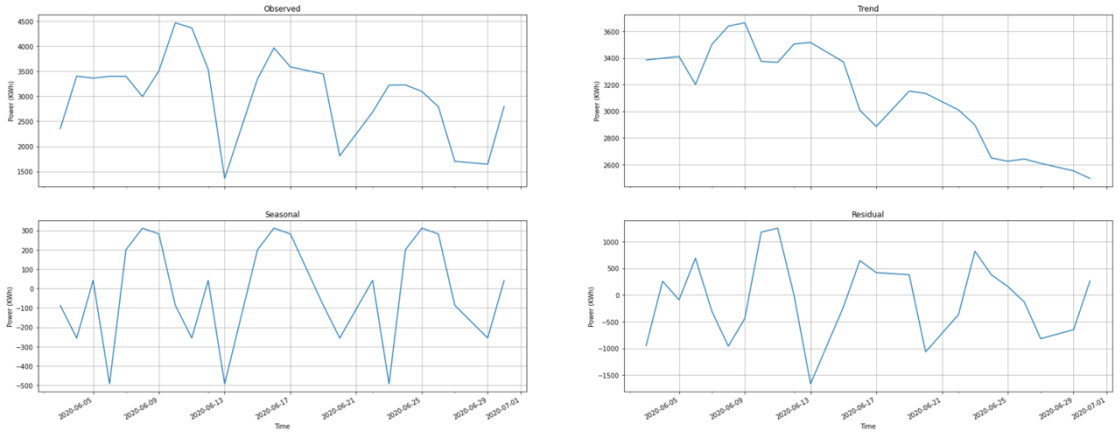


Figure 3-20: Process decomposition over June 2020

The trend in this case seems to change a bit, showing mainly a downward orientation which, on average, behaves like a linear decrease. Again, there is a strong seasonal component which repeats itself over a span of 7 days (notice for example the local minima points on 13-06-2020 and 20-06-2020, however their specific values can be due to bias).

For visualization purposes, a more concise view of the different components of the process can be observed in the plot below:

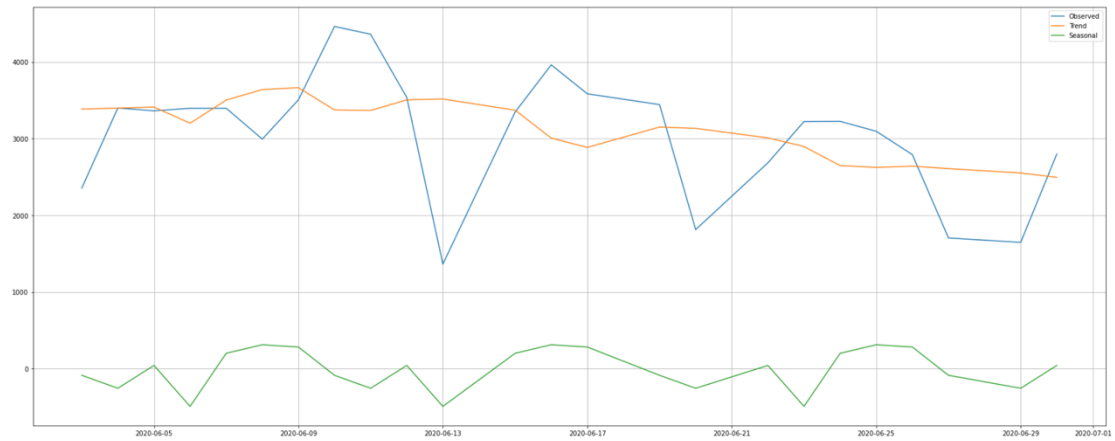


Figure 3-21: Process decomposition over June 2020 (concise View)

For further investigation we can again analyze the related autocorrelation plot:

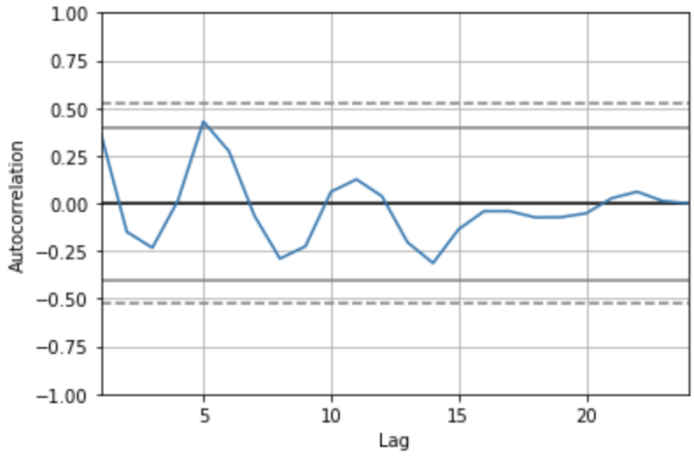


Figure 3-22: ACF Plot over June 2020

Similar consideration as the one for the previous month can be done, but here we can clearly spot some peaks. In this process, no correlation can be noticed by the ACF since all the values fall within the confidence interval. Moreover, by looking at the plot, this process could be thought to be an AR one given its resemblance to a sinusoidal curve.

4. Time Series forecasting

As required by FATER's company for this challenge, we are supposed to forecast the energy (kWh) consumption over three specific dates which are:

- 19-11-2019
- 16-01-2020
- 18-06-2020

All the predicted data should be visualized on an hourly basis. In order to forecast the required dates, the dataset regarding the year 2019-2020, will be split into three sub datasets each of those containing 3-months of observations taken until the date prior to the forecasting date. In particular the new datasets will be the following:

- Dataset 0: ['2019-08-18' ; '2019-11-18']
- Dataset 1: ['2019-10-15' ; '2020-01-15']
- Dataset 2: ['2020-03-17' ; '2020-06-17']

4.1 Data Forecasting

For what concerns this part, we are going to consider several models for data predictions, from the classic ARIMA models among which chose the proper one, to the more "complex" usage of a deep neural network like the LSTM approach. Theorical focus concerning these models can be read in the appendix: Models Overview.

4.1.1 (S)ARIMA

Since we can infer from the original dataset that there is a seasonal component, we can check if it is still present in the datasets 0, 1 and 2. We can proceed with the decomposition of the first process regarding dataset 0.

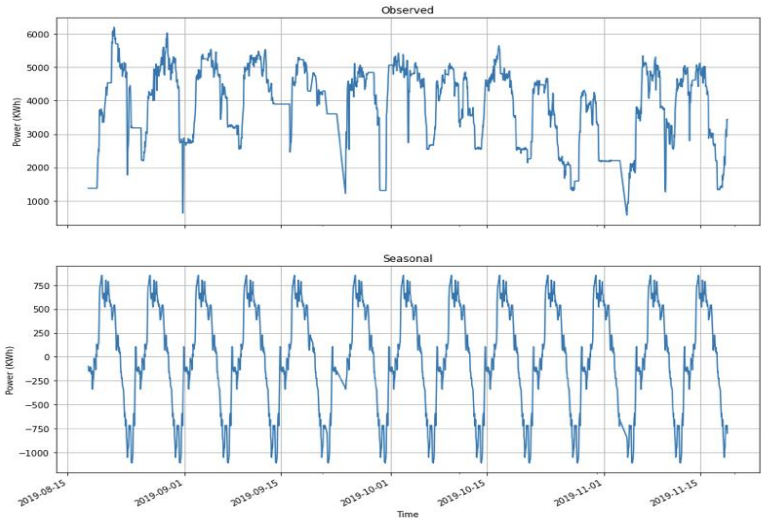


Figure 4-1: Observed process and Seasonal component of Dataset 0

From the above plot, we can confirm that the seasonal component is still present, and it has a weekly behavior. Moreover, by looking at the 'Observed' plot, we could infer that the time series is stationary, since it seems to spread around the average in the same way over time. We can check for this by performing an ADF test:

```
ADF Statistic: -5.201462
p-value: 0.000009
Critical Values:
        1%: -3.433
        5%: -2.863
        10%: -2.567
```

Figure 4-2: ADF test on dataset 0

Looking at the result of test, we can confidently (at 1% confidence level) reject the null hypothesis, that means, we can consider the process to be stationary.

Finally, given the observed seasonality, we can think of directly use as forecasting tool for this process, the SARIMA model. To do so, we need to infer the model's parameters, which can be done by analyzing the ACF and PACF plots of the process:

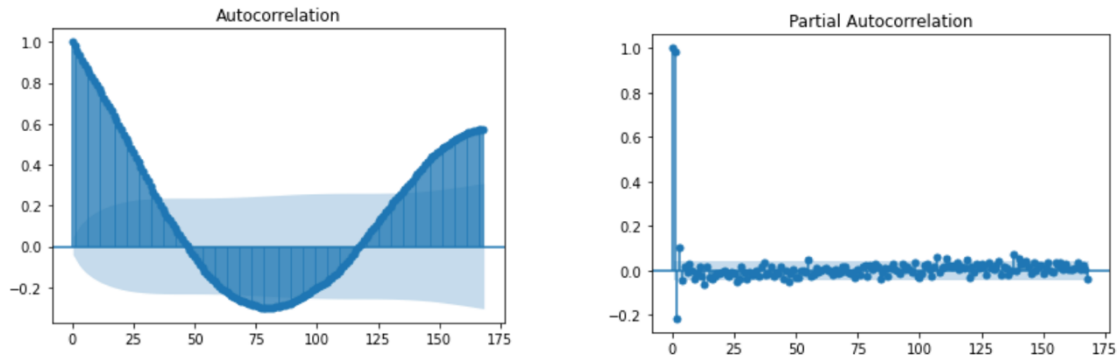


Figure 4-3: ACF and PACF of process of Dataset 0

168 lags have been considered since we are interested in studying the seasonal component within a weekly span (7 x 24).

By the behavior ACF and PACF plots we can see that there is no clear cut-off in the ACF plot while in the PACF it can be observed after the first lag. This can lead us to think that we have an AR process.

Regarding the d and D parameters, we set them to 0 since we can confidently say that the process is stationary.

As a rule of thumb, we can select the p,P,q,Q parameters as follows:

- p is equal to the first lag where the PACF value is above the significance level.
- q is equal to the first lag where the ACF value is above the significance level.

- P≥1 if the ACF is positive at lag S, else P=0.

- Q≥1 if the ACF is negative at lag S, else Q=0.
  Rule of thumb: P+Q≤2

- m: is equal to the ACF lag with the highest value (typically at a high lag).

For what concerns the frequency of the model, a value of 168 has been chosen since it corresponds to the seasonal cycle expressed in hours: (24h/day*7day).

Hence, we can fit the following SARIMA model to the process of dataset 0 after splitting it into train and test set (respectively the 80% and 20% of dataset 0):

$$SARIMA(1,0,1) \times (1,1,0)\ 168$$

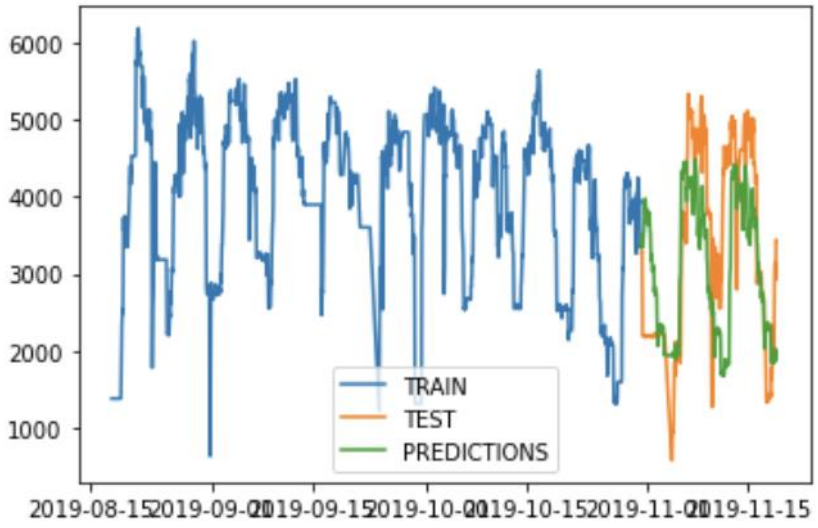


Figure 4-4: Forecasting on Dataset 0

This is an ARIMA process. We can check its performances by plotting the forecasted values over the test set values (above plot)

To quantify the performances, we can consider them in terms of RMSE on test set and a custom model fit measure (which is a kind of normalized RMSE) defined as follows:

$$ModelFit = \left(1 - \frac{RMSE}{max(dataset1) - min(dataset1)}\right) \times 100$$

RMSE: 1053.469
ModelFit: 81.0%

The result is quite good, still we are going to try other models before performing the required dates forecasting, to choose the best possible model. Said that, we can proceed by reiterate this process for dataset 1.
As we can see from the plot below, the seasonal component is still present in dataset 1, and its cycle has a weekly span. Again, we chose an hourly time unit.

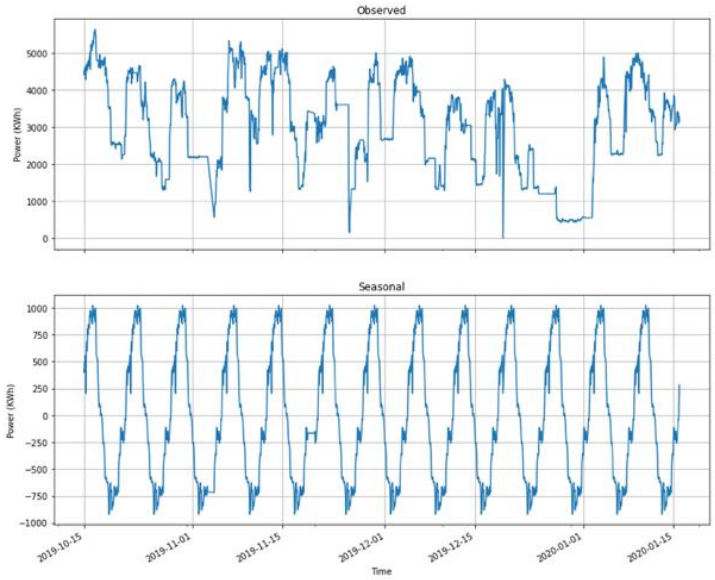


Figure 4-5: Observed process and Seasonal component for dataset 1

Even in this case, we can think of directly use as forecasting tool for this process, the SARIMA model. So, we can start to better analyze this process to choose the proper parameters for the model. Firstly, we can check for stationarity in the process by performing an ADF test:

```
ADF Statistic: -4.623589
p-value: 0.000117
Critical Values:
        1%: -3.433
        5%: -2.863
        10%: -2.567
```

As results from the statistic, we can reject the null hypothesis at a 1% confidence level that means, data are stationary.

Now, to infer the other parameters' value, we can analyze the autocorrelation plot of the process:

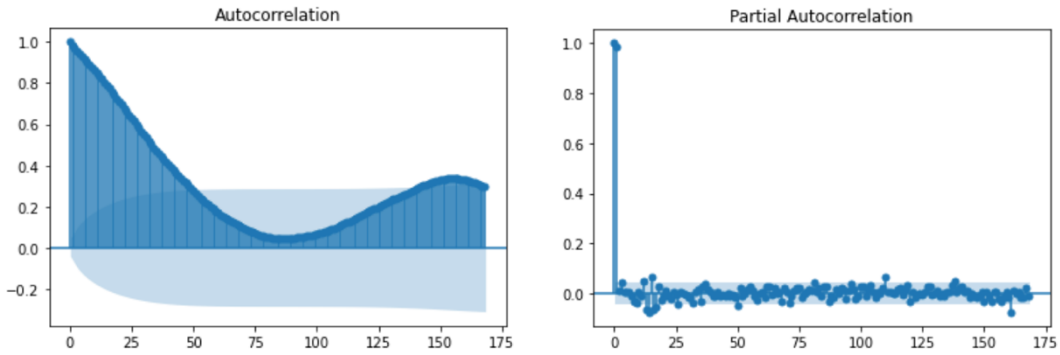


Figure 4-6: ACF and PACF of process of dataset 1

Also, in this case, by the behavior ACF and PACF plots we can see that there is no clear

cut-off in the ACF plot while in the PACF it can be observed after the first lag. This can lead us to think that we have an AR process.

Assuming the same rules of thumb defined for the previous process, we can fit the following model to dataset 1:

$$SARIMA(1,0,1) \times (1,0,0)\ 168$$

Which represents an ARMA process. We can check its performances by plotting the forecasted values over the test set values

.

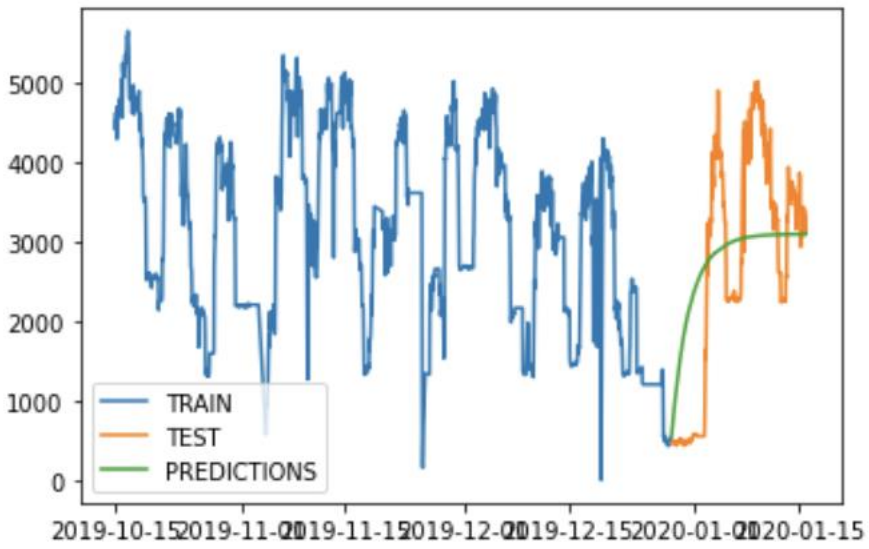


Figure 4-7: Forecasting on dataset1

RMSE: 1140.496
ModelFit: 80.0%

The result is quite good, anyway we can continue our analysis by analyzing dataset 2.

As before we start by decomposing the process to check for seasonality component:

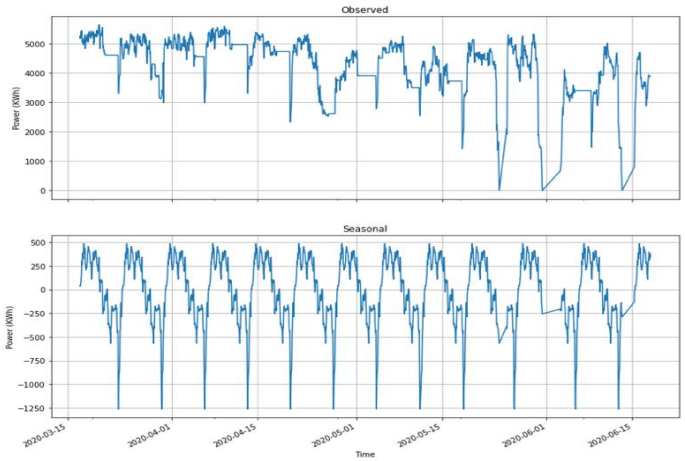


Figure 4-8: Observed process and seasonal component for dataset 2

In this case the same considerations regarding the seasonal component apply and we can use a SARIMA model for forecasting purposes. Again, we start by checking for stationarity in the process by using ADF test:

```
ADF Statistic: -6.357330
p-value: 0.000000
Critical Values:
        1%: -3.434
        5%: -2.863
        10%: -2.568
```

By looking at the result of the test, we can confidently (at 1% level) say that the process is stationary. Then we can analyze the ACF and PACF plots:

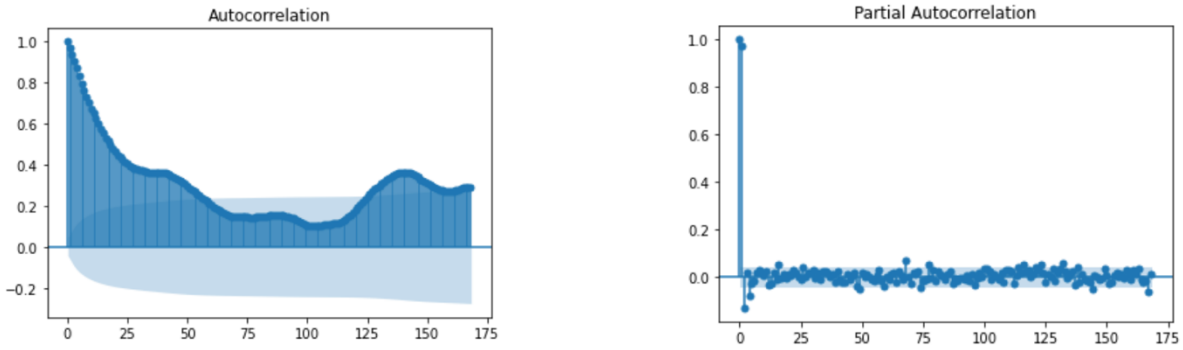
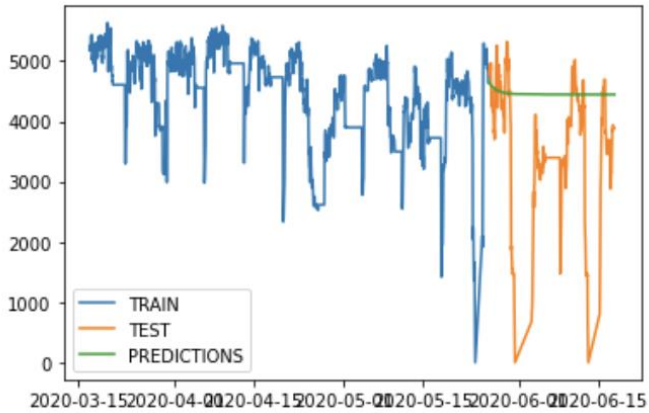


Figure 4-9: ACF and PACF plots for dataset2

By looking at the above plots, the considerations given for dataset 1 can be applied to dataset 2 for the parameter's selection. Moreover, like in the previous process, the plots seem to suggest an AR model. Therefore, given the above plots, we can fit the following model to the split dataset 2:

$$SARIMA(1,0,1) \times (2,0,0)\ 168$$

Which result in an ARMA process, with the following performances:



RMSE: 1261.700
ModelFit: 78.0%

Figure 4-10: Forecasting on dataset2

Finally, the performances results are a bit lower than the other datasets. For this we can now pass to another type of model which is the Holt-Winters one.

4.1.2 Holt-Winters

In this section, the Holt-Winters seasonal model has been used instead of the non-seasonal one (Exponential Smoothing), given the seasonality in our processes. We started by applying it to the dataset 0:

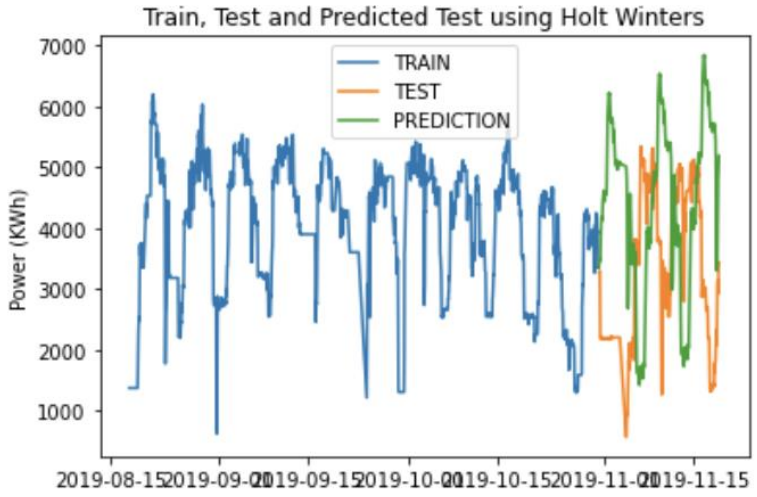
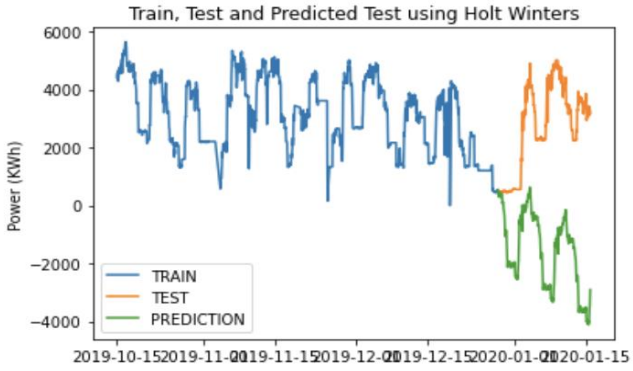


Figure 4-11: Forecast on Dataset 0 with Holt-Winters model

RMSE: 2473.686
ModelFit: 56.0%

The model takes as parameter a seasonal cycle of 7 days, which are measured in hours (168h). It's clear from the above plot, and confirmed by the results (RMSE, ModelFit), that this model is not appropriate for forecasting purposes in our case, this could be due to the fact that it is still a linear model, while the process seems to be non-linear.

We can then proceed with the dataset1:



RMSE: 4771.016
ModelFit: 15.0%

Figure 4-12: Forecast on dataset1 with Holt-Winters model

The model takes as parameter a seasonal cycle of 7 days, which are measured in hours (168h). From the above plot, and the related metrics, we can see that the model presents a poor fit, leading us to assume that it is not appropriate for forecasting purposes in our case. This could be due to the fact that it is still a linear model, while the process seems to be highly non-linear.

We can then proceed with the dataset 2:

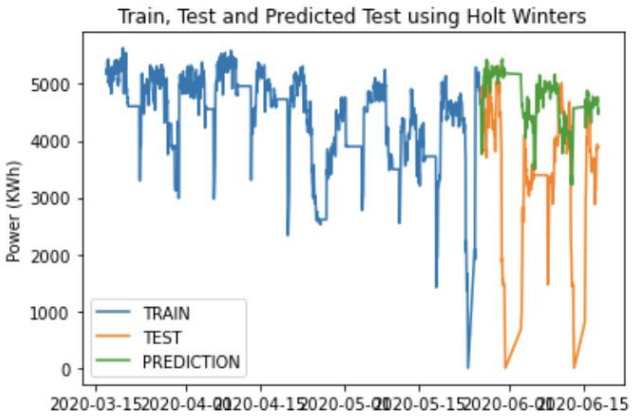


Figure 4-13: Forecast on dataset2 with Holt-Winters

RMSE: 1474.340
ModelFit: 74.0%

This model takes a seasonal cycle's parameter of 7 days as well (which correspond to 168h). In this case the model seems to follow closely the test-set process behavior. In particular, the performances are not that different from the ones obtained by applying the SARIMA model, however, they are still less accurate.

4.1.3 LSTM

In this section, we will use Neural Networks as a tool to learn hidden patterns/relationship within the time series data and provide a meaningful data forecasting. For what concerns the theory about Neural Networks, specific information can be found in the appendix Neural Networks.

Since we are dealing with time series, one of the best approaches (according to literature), is to use LSTM networks. To apply this model, we need to treat our data accordingly. Especially, the following steps need to be performed for all the datasets:

- data scaling
- train-test split
- input-output generation (loopback)
- choice of parameters and model fitting
- model evaluation
- forecasting

Concerning the first step, all the observations regarding the considered year (2019-2020) have been rescaled between 0 and 1, since this range makes it easier to do computations like finding the mean, the variance, perform gradient computation and in general data management. Once this step has been done, we can divide each dataset into a train set (80% of the observations) and a test set (20% of the observations). After that, since a recurrence in the input and in the output of the hidden state is needed, we define a loop_back factor of 1, so that in this case each state depends just on the previous step. Even though the loop_back factor is equal to one, LSTM networks can still keep information regarding long-term dependencies, thanks to their gates.

Once the data has been properly prepared, we have to set the right parameters for the model. In particular, after several empirical trials, a number of neurons of 100 has been chosen, along with the 'Adam' optimizer for computing the gradient, and since it is an adaptive learning rate optimization algorithm, starting with a learning rate of 0.01 does not pose any hindrance in selecting the best learning rate.

For what concerns the fitting of the model, a number of epochs of 25 has been selected. This value has been heuristically obtained in order to provide a good trade-off between computation time and model performances.

Finally, since the LSTM model result is affected by its initial state, in order to have a more robust result, we decided to introduce an additional parameter k, which represents the number of times that the model is fitted, and the data are forecasted (considering both a minute-by-minute time unit and a hour-by-hour basis). At the end of the k iterations, all the results are averaged to have the less biased one.

The results for the different datasets (0,1,2) can be observed in the following plots:

- Dataset 0

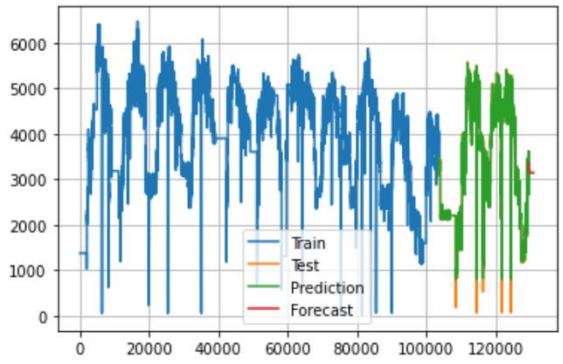


Figure 4-14: minute-by-minute energy (kWh) forecast

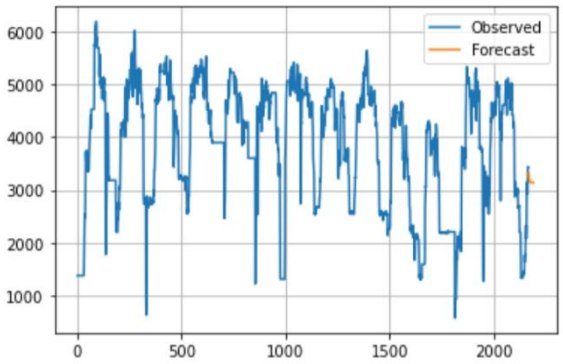


Figure 4-15: hour-by-hour energy (kWh) forecast

Train Model Fit %: 97.8269
Validation Model Fit %: 97.8759

- Dataset 1

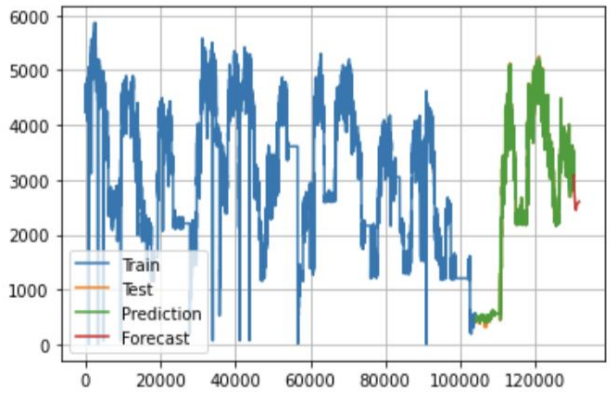


Figure 4-17: minute-by-minute energy (kWh) forecast

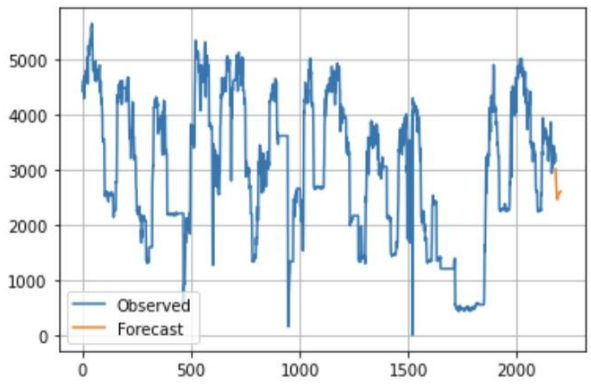


Figure 4-16: hour-by-hour energy (kWh) forecast

Train Model Fit %: 97.8863
Validation Model Fit %: 97.9287

- Dataset 2

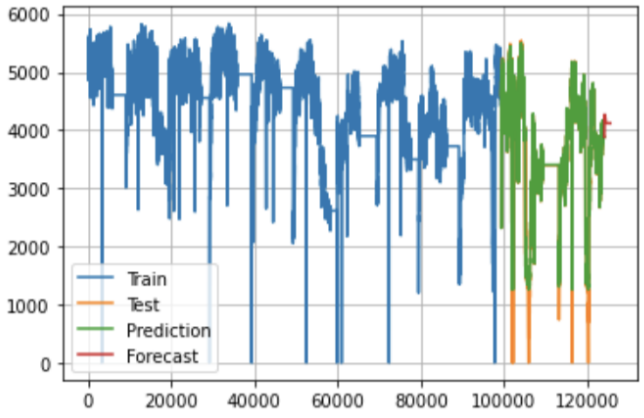


Figure 4-19: minute-by-minute energy (kWh) forecast

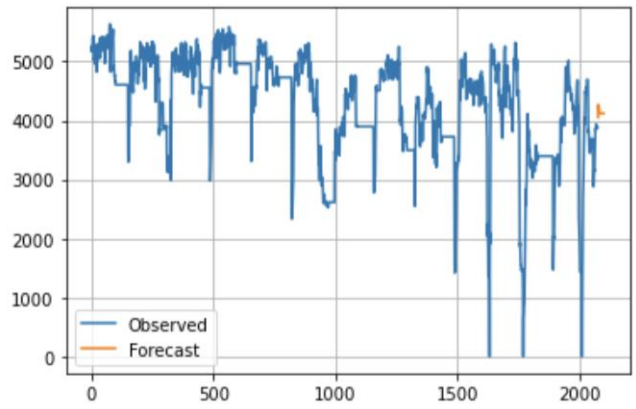


Figure 4-18: hour-by-hour energy (kWh) forecast

Train Model Fit %: 97.5053
Validation Model Fit %: 97.5549

After seeing the results given by the different datasets, we can clearly observe that the 'Validation Model Fit' is always higher than its 'Train' counterpart. This counter-intuitive behavior could be explained considering different aspects like:

- Regularization applied during training, but not during validation/testing.
- Training loss is measured *during* each epoch while validation loss is measured *after* each epoch.

- The validation set may be easier than the training set (or there may be leaks).

The performances of LSTM applied to each different dataset, are always better than the ones obtained by the "classical" statistical methods like SARIMA and Holt-Winters. This can be due to the fact that a data-driven method like LSTM does not have to follow a specific structure (model assumptions) as opposed to the previously used methods. Furthermore, since LSTMs are Neural Networks, they are universal approximators for any kind of function (the data generating function which is not known) we would like to study, in particular with respect to times series analysis, which is our case of study. In conclusion of this chapter, here is a table which summarizes and compares the obtained results.

Test Set Accuracy Comparison (%)			
Dataset/Model	SARIMA	Holt-Winters	LSTM
0	81.0	56.0	97.8
1	80.0	15.0	97.9
2	78.0	74.0	97.5

Table 4.1: Test set accuracy Comparison

5. Energy distribution and optimization, Cost minimization

Once the hourly energy consumption has been properly and efficiently estimated for the requested data, we can proceed by assigning the forecasted energy across the available motors and external energy. This procedure poses a constrained optimization problem in which the constraints are linked to the Table 2.2: Motors specifics (cost, minimum/maximum energy supply, maintenance) and to the cost of external energy (Enel).

Since the aim of this project is to minimize the cost linked to the energy supply, we need to define an objective function that considers the costs of each motor and the cost of external energy. Depending on the hourly energy forecasted, and given the motors' specifics which are known, we can define different intervals for which different combinations of motors and external energy are considered. This led us to define a different objective function in each of the intervals.

For defining such intervals and by considering the constraints of the problem, it is important to establish a cutoff for the external energy supply since we could have energy levels for which using the external energy is competitive with respect to the usage of the proposed motors. In this case, two cutoffs for the 'Enel' supply were identified: one with respect to the Jenbacher motors, and the other with respect to the Caterpillar ones (since they have different costs). The idea used to find those values is the following:

Considering the minimum energy supply provided by Jenbacher motors, and multiplying it by their cost, we find our comparison value. Then, we should find the highest integer value for the product of the energy supplied by 'Enel' and its cost, that is still lower than the comparison value. The same process has been applied to the Caterpillar motors as

well, but in this case, we had also to consider the maintenance cost. This process leads us to the following external energy values:

- Enel_cutoff_JEN: 355 kWh
- Enel_cutoff_CAT: 490 kWh

Now that these cutoffs are provided, we identified 10 different intervals along with their respective objective function:

1) $energy \in [0, Enel\_cutoff\_JEN]$
2) $energy \in (Enel\_cutoff\_JEN, min\_JEN]$
3) $energy \in (min\_JEN, max\_JEN]$
4) $energy \in (max\_JEN, max\_JEN + Enel\_cutoff\_JEN]$
5) $energy \in (max\_JEN + Enel\_cutoff\_JEN, max\_JEN \times 2]$
6) $energy \in (max\_JEN \times 2, (max\_JEN \times 2) + Enel\_cutoff\_CAT\ ]$
7) $energy \in ((max\_JEN \times 2) + Enel\_cutoff\_CAT\ , (max\_JEN \times 2) + max\_CAT]$
8) $energy \in ((max\_JEN \times 2) + max\_CAT, (max\_JEN \times 2) + max\_CAT + Enel\_cutoff\_CAT]$
9) $energy \in ((max\_JEN \times 2) + max\_CAT + Enel\_cutoff\_CAT, (max\_JEN \times 2) + (max\_CAT \times 2)]$
10) $energy \in (\ (max\_JEN \times 2) + (max\_CAT \times 2), +\infty)$

Where min_JEN/max_JEN are the minimum/maximum energy supply provided by Jenbacher motors and min_CAT/max_CAT are the minimum/maximum energy provided by Caterpillar motors.

In order to solve this problem in the python environment, which is the one used till now, we relied on the Google's library 'OR-tools'. This is an open-source software suite for optimization, tuned for tackling integer and linear programming, and constraint programming along with other problems. In particular, as specified at the beginning of the chapter, we modeled a constrained optimization problem (constraint programming problem). However, the library can handle only linear problems along with integer constraints, this also provides another reason for defining different energy ranges.

Below, are listed the different objective functions, according to their corresponding energy interval. In particular, the 4 motors and the external energy will be modeled as variables in the following way:

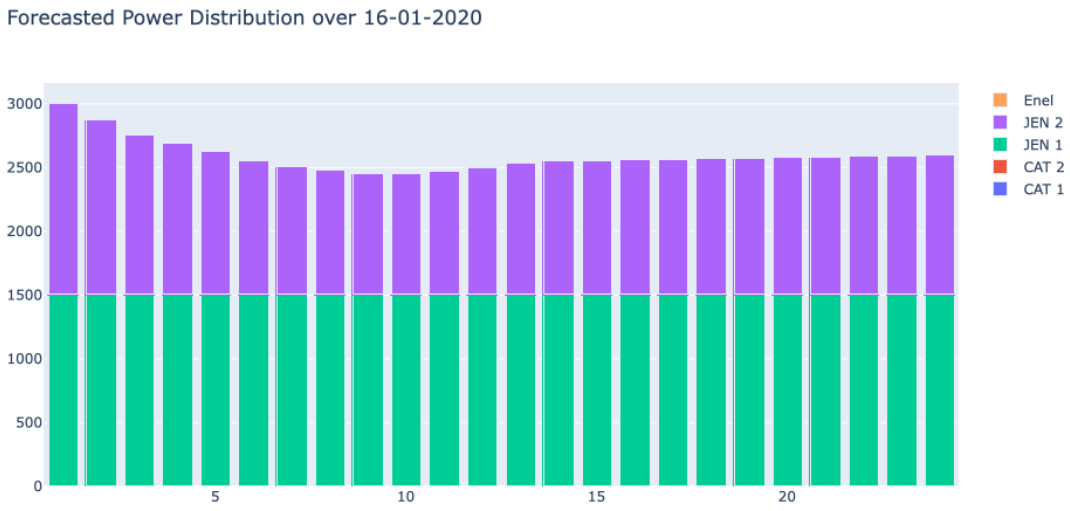
- X_1: Caterpillar 1
- X_2: Caterpillar 2
- Y_1: Jenbacher 1
- Y_2: Jenbacher 2
- Z: Enel
- a: Caterpillar usage cost, 0.070744 € / kWh
- b: Jenbacher usage cost, 0.071448 € / kWh
- c: Enel cost, 0.15 € / kWh
- d: Caterpillar maintenance cost, 15 € /h

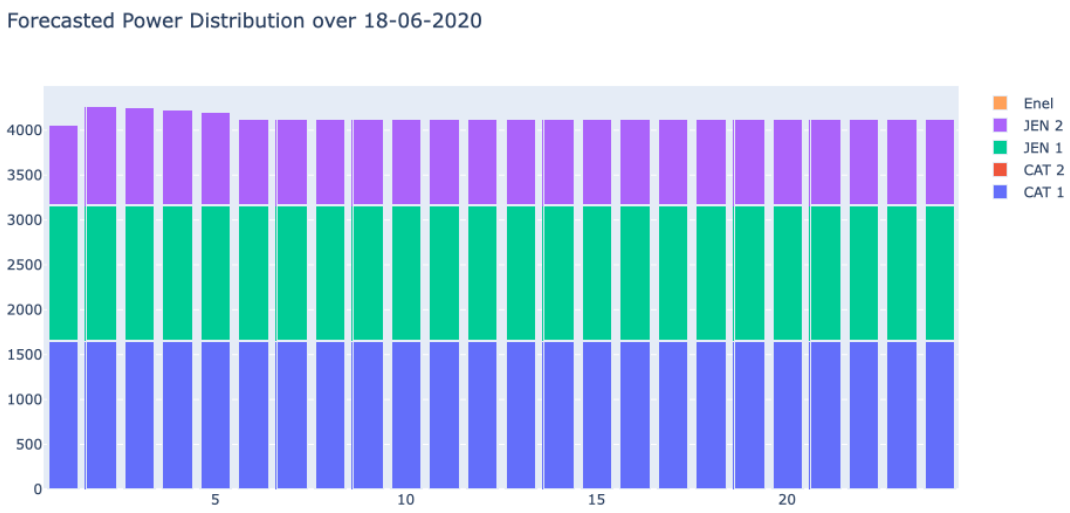
In order to meet the framework needs, the costs were rescaled accordingly to obtain integer values (this rescaling does not change the results).

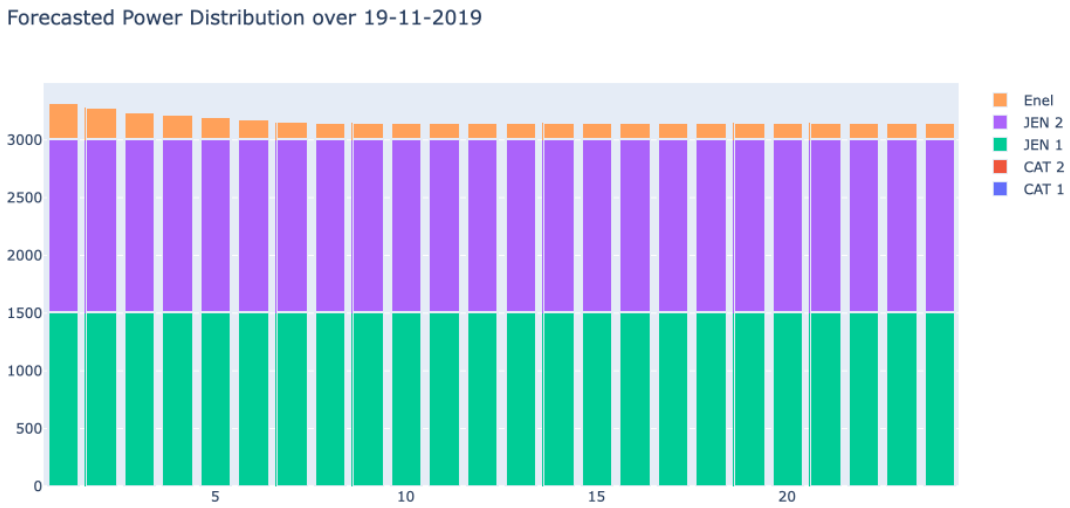
Here are the objective functions for the different power range:

1) $c\ Z$
2) b Y_1
3) b Y_1
4) b Y_1 + $c\ Z$
5) b Y_1 + b Y_2
6) b Y_1 + b Y_2 + $c\ Z$
7) a X_1 + b Y_1 + b Y_2 + d
8) a X_1+ b Y_1 + b Y_2 + $c\ Z$ + d
9) a X_1+ a X_2+b Y_1 + b Y_2+ 2 d
10) a X_1+ a X_2+b Y_1 + b Y_2 + $c\ Z$ + 2 d

Having modeled the problem in this way, we obtained the following results:







The above plots, represent the distribution of the variable Power across the available energy supply, over the required hours. In other words, they tell the way in which the forecasted energy for those dates, is split among the several motors (with the possible use of external energy), to minimize the costs of energy and its waste.

By looking at the above plots, the Jenbacher motors, which according to the specifics are the less expensive ones, are always used, since they minimize the costs. Furthermore, when Caterpillar motors are used, since they have also a maintenance cost, their usage is maximized by the model while the remaining energy supply is handled by the two Jenbacher motors. When both the Jenbacher motors are needed, the algorithm tries to keep one of them at full capacity while the other varies. This provides a good practice, because we do not have to reconfigure one of the motors over time, and this leads to a constant energy supply without any losses.

Moreover, as it can be seen, when the external energy is needed it is used in small quantities since, from its specifics, it is the most expensive energy provider for middle to high energy needs.

Conclusion

In this project we provided different forecasted methods for solving a time series analysis problem (energy distribution over time). Clearly, classical statistical methods have been overshadowed by modern machine learning techniques like Recurrent Neural Network and in particular LSTM. Then, costs were optimized according to different criteria which could be changed for considering the overcharging problem related to the provided motors, in order to not use Jenbacher motors at full capacity. Still, the results obtained were above the expectations, (regarding the accuracy of LSTMs), and probably by changing the hyperparameters of the Neural Networks we could even provide better results but at the expense of a higher computational load. However, we are confident that the results provided will be of great help in deciding the configuration of the engines in Fater's Campochiaro plant.

Theoretical Appendix

This part of the documentation contains a more in-depth focus on the topics covered.

Time Series Analysis

Since time series have been vastly studied in literature, it is useful to introduce some mathematical concepts that can help us to better understand and analyze the problem at hand. A **"stochastic process"** can be seen as a statistical phenomenon that evolves in time according to probabilistic laws. Mathematically speaking, a stochastic process may be defined as a collection of random variables ordered in time and defined at a set of time points which may be continuous ($X(t)\ with\ t\ \in (-\infty, \infty)$) or discrete ($X_t\ with\ t\ \in \mathbb{Z}$). According to time series analysis it is possible to vary the length of the observed time series (the sample), but it is usually impossible to make more than one observation at a given time (the processes are generally not repeatable).
Between the infinite "ensemble" of time series, we may observe only one "realization" of the stochastic process, in our case we will refer to this realization as $x_t\ with\ t \in \mathbb{N}$ since our process is discrete.

Thus, even though a time series is nothing more than a particular realization of the underlying stochastic process, with the term **"time series analysis"** we are referring to the causes that determined the observed time series and so to the stochastic process itself. A simple way to describe a stochastic process is by giving the **moments** of the process, especially the first and the second moments.

The first moment of a stochastic process is also known as the **mean** and it can be defined as follows:

$$\mu(t) = E[X_t]$$

The second moment is given by the **variance** and the **covariance** functions.
The variance function is defined in this way:

$$\sigma^2(t) = Var[X_t]$$

Since the variance alone is not enough to describe the second moments of a sequence of random variables the covariance function is needed. However, in this case, the covariance refers to different observations of the same process, so we can use the term **autocovariance.**
The autocovariance function between X_{t_1} and X_{t_2} is:

$$\gamma(t_1, t_2) = E\{[X_{t_1} - \mu_{t_1}][X_{t_2} - \mu_{t_2}]\}$$

A useful condition that makes the study of time series easier is the **"stationarity"**. A time series is said to be **strictly stationary** if the joint distribution of $X_{t_1}, \dots, X_{t_n}$ is the same as the joint distribution of $X_{t_1+\tau}, \dots, X_{t_{n+\tau}}$. In other words, if the process is stationary shifting the time origin by an arbitrary amount τ has no effect on the joint distributions, and this holds true for any value of n (the times in which we observe the process).
If $n = 2$ the joint distribution of X_{t_1} and X_{t_2} depends only on $\tau = (t_2 - t_1)$ which is called the **lag** of the process and the autocovariance function can be written as:

$$\gamma(\tau) = E\{[X_t - \mu][X_{t+\tau} - \mu]\} = Cov[X_t, X_{t+\tau}]$$

and is called the autocovariance coefficient at lag τ.
Since the size of the autocovariance coefficient depends on the units in which X_t is measured, it is useful to standardize it to produce a new function called the **autocorrelation** function, which is given by:

$$\rho(\tau) = \gamma(\tau) \Big/ \gamma(0)$$

and it measures the correlation between X_t and $X_{t+\tau}$.
Although is useful to have a rigorous definition of stationarity, in practice a laxer version of it is pursued, and it leads us to define a "second-order stationarity" or "weak stationarity".
A process is said to be weakly stationary if:

$$E[X_t] = \mu \text{ and } Cov[X_t, X_{t+\tau}] = \gamma(\tau)$$

As it can be noticed from this definition, no other assumptions are made about higher order moments. This weaker definition of stationarity is preferred to the stricter one since the structure of many processes depend only on their first two moments.

The simplest type of discrete random process is the "purely random process". A stochastic process is said to be **purely random** if it consists of a sequence of random variables $\{Z_t\}$ which are mutually independent and identically distributed. From this definition, it clearly follows that this kind of process has constant mean and variance. Furthermore, the autocovariance function is null for each value of k: $\gamma(k) = Cov(Z_t, Z_{t+k}) = 0 \; \forall k \in \mathbb{Z} - \{0\}$.

A process of this kind is stationary, sometimes is called **white noise** (prevalently in the engineering field) and has an autocorrelation function defined in this way:

$$\rho(k) = \begin{cases} 1 \; for \; k = 0 \\ 0 \; otherwise \end{cases}$$

Is useful to have a clear idea about what is a purely random process since it constitutes the building block, for many other processes like MA (Moving Average) processes and AR (Autoregressive) processes.

Let us assume to have a purely random process $\{Z_t\}$ with $\mu = 0$ and a finite variance σ_Z^2, then a process $\{X_t\}$ is said to be a MA process of order q if:

$$X_t = \beta_0 Z_t + \beta_1 Z_{t-1} + \cdots + \beta_q Z_{t-q}$$

where $\{\beta_i\}$ are constants. Usually, the Zs are scaled such that $\beta_0 = 1$.
It is easy to find that:

$$E[X_t] = 0 \text{ and } Var[X_t] = \sigma_Z^2 \sum_{i=0}^{q} \beta_i^2$$

since the Zs are independent. Then the autocovariance function takes the following form:

$$\gamma(k) = Cov(X_t, X_{t+k}) = Cov\left(\beta_0 Z_t + \cdots + \beta_q Z_{t-q}, \beta_0 Z_{t+k} + \cdots + \beta_q Z_{t+k-q}\right)$$

$$= \begin{cases} 0 \; for \; k > q \\ \sigma_Z^2 \sum_{i=0}^{q-k} \beta_i \beta_{i+k} \; for \; k = 0, 1, \dots, q \\ \gamma(-k) \; for \; k < 0 \end{cases}$$

(The value of $\gamma(k)$ for $k < 0$ is easily derived since the autocovariance function is even)
As $\gamma(k)$ does not depend on t and μ is constant, the process is second-order stationary for all values of the $\{\beta_i\}$.
The autocorrelation function of the MA(q) process is given by:

$$\rho(k) = \begin{cases} 1 \; for \; k = 0 \\ \sum_{i=0}^{q-k} \beta_i \beta_{i+k} / \sum_{i=0}^{q} \beta_i^2 \; for \; k = 1, \dots, q \\ 0 \; for \; k > q \\ \rho(-k) \; for \; k < 0 \end{cases}$$

We can notice that the autocorrelation function of a MA process "cuts off" at lag q.
An ulterior condition can be given to a stochastic process to grant the uniqueness of the autocorrelation function, this condition is called **invertibility**, but its description goes beyond the purpose of this project.

Another important class of processes is the AR processes. If we assume a $\{Z_t\}$ process to be a purely random one with mean zero and variance σ_Z^2, then a process $\{X_t\}$ is said to be an autoregressive one of order p if:

$$X_t = \alpha_1 X_{t-1} + \cdots + \alpha_p X_{t-p} + Z_t$$

Since this equation strongly resembles the one from the multiple linear regression, but with X_t regressed on previous values of itself, this kind of process is called autoregressive.
For simplicity of notation, to derive the first and second moments of an autoregressive process, we consider an AR process of order one ($p = 1$).
In this case:

$$X_t = \alpha X_{t-1} + Z_t$$

by successive substitutions the previous equation can be rewritten as:

$$X_t = \alpha(\alpha X_{t-2} + Z_{t-1}) + Z_t = \alpha^2(\alpha X_{t-3} + Z_{t-2}) + \alpha Z_{t-1} + Z_t$$

So eventually X_t can be expressed as an infinite order MA process (when $-1 < \alpha < 1$):

$$X_t = Z_t + \alpha Z_{t-1} + \alpha^2 Z_{t-2} + \cdots$$

Or by using the backward shift operator B:

$$(1 - \alpha B)X_t = Z_t \text{ where } B^j X_t = X_{t-j} \; for \; all \; j$$

By rewriting X_t with the use of this notation:

$$X_t = Z_t/(1 - \alpha B) =$$
$$(1 + \alpha B + \alpha^2 B^2 + \cdots)Z_t = Z_t + \alpha Z_{t-1} + \alpha^2 Z_{t-2} + \cdots$$

Now it is easier to see that:

$$E[X_t] = 0$$
$$Var[X_t] = \sigma_Z^2(1 + \alpha^2 + \alpha^4 + \cdots)$$

The variance is finite if $-1 < \alpha < 1$:

$$Var[X_t] = \sigma_X^2 = \sigma_Z^2/(1 - \alpha^2)$$

The autocovariance function is given by:

$$\gamma(k) = E[X_t X_{t+k}] = E\left[\left(\sum \alpha^i Z_{t-i}\right)\left(\sum \alpha^j Z_{t+k-j}\right)\right] = \sigma_Z^2 \sum_{i=0}^{\infty} \alpha^i \alpha^{k+i} \; for \; k \geq 0$$

Which converges for $|\alpha| < 1$ to:

$$\gamma(k) = \alpha^k \sigma_Z^2/(1 - \alpha^2) = \alpha^k \sigma_X^2$$

For $k < 0$, $\gamma(k) = \gamma(-k)$. Since $\gamma(k)$ does not depend on t, an AR process of order 1 is second-order stationary provided that $|\alpha| < 1$.

The autocorrelation function is given by:

$$\rho(k) = \alpha^{|k|} \; for \; k \epsilon \mathbb{Z}$$

Augmented Dickey-Fuller test (ADF)

An **augmented Dickey–Fuller test** (**ADF**) tests the null hypothesis that a unit root is present in a time series sample. The alternative hypothesis is different depending on which version of the test is used but is usually stationarity or trend-stationarity. We say that a linear stochastic process has a unit root if 1 is a root of the process's characteristic equation, such a process is indeed non-stationary. If there are *d* unit roots, the process will have to be differenced *d* times to make it stationary. Consider a discrete-time stochastic process $\{X_t\}$, and suppose that it can be written as an autoregressive process of order *p*:

$$X_t = \alpha_1 X_{t-1} + \cdots + \alpha_p X_{t-p} + Z_t$$

Here, $\{Z_t\}$ is a serially uncorrelated, zero-mean ($\mu = 0$) stochastic process with constant variance σ^2. For convenience, assume $X_0 = 0$. If $m = 1$ is a root of the characteristic equation, of multiplicity 1:

$$m^p - m^{p-1}\alpha_1 - m^{p-2}\alpha_2 - \cdots - \alpha_p = 0$$

then the stochastic process has a **unit root** or, alternatively, is integrated of order one, denoted $I(1)$. If $m = 1$ is a root of multiplicity r, then the stochastic process is integrated of order r, denoted $I(r)$.

The testing procedure for the ADF test is applied to the model:

$$\Delta X_t = \alpha + \beta t + \gamma X_{t-1} + \delta_1 \Delta X_{t-1} + \cdots + \delta_{p-1} \Delta X_{t-p+1} + Z_t$$

where α is a constant, β the coefficient on a time trend and p the lag order of the autoregressive process. There are three main versions of the test, one to check for the presence of a unit root, another for the presence of a unit root plus a constant and the last one for the presence of a unit root with a constant and a deterministic time trend.

By including lags of the order p the ADF formulation allows for higher-order autoregressive processes. This means that the lag length p needs to be determined when applying the test.

The unit root test is then carried out under the null hypothesis $\gamma = 0$ against the alternative hypothesis of $\gamma < 0$. Once a value for the test statistic $DF_\tau = \frac{\hat{\gamma}}{SE(\hat{\gamma})}$, where $\hat{\gamma}$ is an estimate of γ and $SE(\hat{\gamma})$ is its standard error, is computed it can be compared to the relevant critical value for the Dickey–Fuller test. As this test is asymmetrical, we are only concerned with negative values of our test statistic. If the calculated test statistic is less (more negative) than the critical value, then the null hypothesis is rejected and no unit root is present, so the series is stationary.

Models Overview

ARIMA

The *Autoregressive Integrated Moving Average* (ARIMA) model is a combination of the differenced *autoregressive* (AR) model with the *moving average* (MA) model. It is expressed as:

$$y'_t = I + \alpha_1 \times y'_{t-1} + \alpha_2 \times y'_{t-2} + \ldots + \alpha_p \times y'_{t-p} + e_t + \theta_1 \times y'_{t-1} + \theta_2 \times y'_{t-2} + \ldots + \theta_q \times y'_{t-q} \qquad (*)$$

The AR part of ARIMA shows that the time series is regressed on its own past data. The MA part of ARIMA indicates that the forecast error is a linear combination of past respective errors.
The ARIMA model is effective in fitting past data with this combination approach and help forecast future points in a time series.

Eq. (*) shows ***ARIMA (*****p,d,q*****)*** model, where

- The parameter p is the number of autoregressive terms or the number of "lag observations." It is also called the "lag order," and it determines the outcome of the model by providing lagged data points.
- The parameter d is known as the degree of differencing. It indicates the number of times the lagged indicators have been subtracted to make the data stationary.
- The parameter q is the number of forecast errors in the model and is also referred to as the size of the moving average window.

The parameters take the value of integers and must be defined for the model to work. They can also take a value of 0, implying that they will not be used in the model. Once the parameters (*p, d, q*) have been defined, the ARIMA model aims to estimate the coefficients **α** and **θ**, which is the result of using previous data points to forecast values.

SARIMA

Despite ARIMA potentiality, it has some limitation when seasonality comes into play. Therefore, to handle time series with seasonal component, its extension has been considered that means, the SARIMA models (Seasonal Autoregressive Integrated Moving Average). A seasonal ARIMA model takes into account the seasonal terms of the time series, by including them into the classic ARIMA models. These additional terms are then multiplied by the non-seasonal terms. A SARIMA model can be formalized as follow:

$$SARIMA(p,d,q) \times (P,D,Q)\, m$$

As anticipated, in maintains all the known parameters of an ARIMA model, plus some parameters regarding the seasonal component. In particular we have:

- P: Seasonal autoregressive order
- D: Seasonal difference order
- Q: Seasonal moving average order
- m: The number of time steps for a single seasonal period

In a similar way to ARIMA model, ACF and PACF plots can be analyzed to specify values for the seasonal model by looking at correlation at seasonal lag time steps. In conclusion, the modelling procedure is almost the same as for non-seasonal data, except that we need to set seasonal AR and MA parameters as well as the non-seasonal ones.

Exponential Smoothing & Holt-Winters

Exponential smoothing is a forecasting procedure invented by C.C. Holt around 1958 and is used for time series without a seasonal component and with no systematic trend. Even though most processes do not fall into this category, this method is still useful since seasonality and trend can be removed from a process to obtain a stationary process.
Given a non-seasonal time series with no systematic trend $x_1, x_2, \dots, x_N$, it is natural to take as an estimate of x_{N+1} a weighted sum of past observations:

$$\hat{x}(N,1) = c_0 x_N + c_1 x_{N-1} + c_2 x_{N-2} + \cdots$$

where the $\{c_i\}$ are the weights. By giving more weight to recent observations instead of older ones, geometric weights are chosen because they decrease by a constant ratio:

$$c_i = \alpha(1-\alpha)^i, i = 0,1, \dots and\ 0 < \alpha < 1$$

Then the previous equation can be rewritten as:
$\hat{x}(N,1) = \alpha x_N + (1-\alpha)[\alpha x_{N-1} + \alpha(1-\alpha)x_{N-2} + \cdots] = \alpha x_N + (1-\alpha)\hat{x}(N-1,1)$
This last equation can be used recursively to compute forecasts. Since the geometric weights lie on an exponential curve, hence the name of the method.

Exponential smoothing can be easily generalized to deal with time series containing trend and seasonal variation. This procedure is called Holt-Winters in honor to the pioneering work done by P.R. Winters in the time series forecasting field in 1960.
Let L_t, T_t, I_t denote the local level, trend, and seasonal index, respectively, at time t. Thus T_t is the expected increase or decrease per length of the seasonal cycle (days, weeks, months, years) in the current level. Let α, γ, δ denote the three smoothing parameters for

updating the level, the trend, and the seasonal index respectively (with $\alpha, \gamma, \delta \in (0,1)$). If the seasonal variation is additive, then the updating equations for this model are:

$$L_t = \alpha(x_t - I_{t-\tau}) + (1-\alpha)(L_{t-1} + T_{t-1})$$
$$T_t = \gamma(L_t - L_{t-1}) + (1-\gamma)T_{t-1}$$
$$I_t = \delta(x_t - L_{t-1}) + (1-\delta)I_{t-\tau}$$

Where x_t is a new observation of the time series at time t and τ is the length of the seasonal cycle (it would be equal to 7 for the days in a week, 4 for weeks in a month, 12 for the months in a year).
Then the forecasts from time t are given by:

$$\hat{x}(t,k) = L_t + kT_t + I_{t-\tau+k}, for\ k = 1, \dots, \tau$$

For the multiplicative seasonal variation, analogous formulas can be derived.

Neural Networks

Deep forward neural networks, also known as **feedforward neural networks**, or **multilayer perceptrons (MLPs)** define the building blocks for all the other neural networks. The goal of a feedforward network is to approximate some function f^*. For example, for a classifier, $y = f^*(\boldsymbol{x})$ maps an input $\boldsymbol{x}$ to a category y. A feedforward network defines a mapping $\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$ and learns the values of the parameters $\boldsymbol{\theta}$ that result in the best function approximation.
These models are called **feedforward** because information flows through the function being evaluated from $\boldsymbol{x}$, through the intermediate computations used to define f, and finally to the output $\boldsymbol{y}$. Feedforward neural networks are called **networks** because they are typically represented by composing together many different functions. The model is associated with a directed acyclic graph describing how the functions are composed together. For example, we could have three functions $f^{(1)}, f^{(2)}, f^{(3)}$ connected in a chain, to form $f(\boldsymbol{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\boldsymbol{x}))$. The overall length of the chain gives the **depth** of the model, hence the name **"deep-learning"**.
The training examples specify directly what the output layer must do at each point $\boldsymbol{x}$; it must produce a value that is close to y. The behavior of the other layers is not directly specified by the training data. Since the training data does not show the desired output for each of these layers, they are called **hidden layers**. Finally, these networks are called **neural** because they are loosely inspired by neuroscience. Each hidden layer of the network is typically vector valued. The dimensionality of these hidden layers determines the **width** of the model. Each element of the vector may be interpreted as playing a role analogous to a neuron. Rather than thinking of the layer as representing a single vector-to-vector function, we can also think of the layer as consisting of many units that act in parallel, each representing a vector-to-scalar function. Each unit resembles a neuron in the sense that it receives input from many other units and computes its own
activation value. To represent nonlinear functions of $\boldsymbol{x}$, we can apply a linear model not to $\boldsymbol{x}$ itself but to a transformed input $\varphi(\boldsymbol{x})$, where φ is a nonlinear transformation. We can think of φ as providing a set of features describing $\boldsymbol{x}$, or as providing a new representation for $\boldsymbol{x}$. An important question that arises naturally is "how do we choose φ?". There are different ways to choose φ, but according to the deep learning approach, this function needs to be learnt. Following this approach, we obtain this model:

$$y = f(\boldsymbol{x}; \boldsymbol{\theta}, \boldsymbol{w}) = \varphi(\boldsymbol{x}; \boldsymbol{\theta})^T \boldsymbol{w}$$

We now have parameters $\boldsymbol{\theta}$ that we use to learn φ from a broad class of functions, and parameters $\boldsymbol{w}$ that map from $\varphi(\boldsymbol{x})$ to the desired output. This is an example of a deep

feedforward network, with φ defining a hidden layer. In this approach, we parametrize the representation as $\varphi(\boldsymbol{x};\boldsymbol{\theta})$ and use an optimization algorithm to find the $\boldsymbol{\theta}$ that corresponds to a good representation.

Cost function:

An important aspect of the design of a deep neural network is the choice of the cost function. In most cases, our parametric model defines a distribution $p(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{\theta})$ and we simply use the principle of maximum likelihood. This means that we can use the cross-entropy between the training data and the model's predictions as the cost (or loss) function.
Most modern neural networks are trained using maximum likelihood. This means that the cost function is simply the negative log-likelihood. This cost function is given by:

$$J(\boldsymbol{\theta}) = -E_{\boldsymbol{x},\boldsymbol{y}\sim\hat{p}_{data}} log p_{model}(\boldsymbol{y}|\boldsymbol{x})$$

The specific form of the cost function changes from model to model, depending on the specific form of $log p_{model}$.

Output units:

The choice of cost function is tightly coupled with the choice of output unit. Suppose that the feedforward network provides a set of hidden features defined by $\boldsymbol{h} = f(\boldsymbol{x};\boldsymbol{\theta})$, then the role of the output layer is to provide some additional transformation from the features to complete the task that the network must perform.
One simple kind of output unit is based on an affine transformation with no nonlinearity, these are often just called **linear units**. Given features $\boldsymbol{h}$, a layer of linear output units produces a vector $\hat{\boldsymbol{y}} = \boldsymbol{W}^T\boldsymbol{h} + \boldsymbol{b}$. Linear output layers are often used to produce the mean of a conditional Gaussian distribution (i.e: $p(\boldsymbol{y}|\boldsymbol{x}) = \mathcal{N}(\boldsymbol{y};\hat{\boldsymbol{y}},\boldsymbol{I})$).
Maximizing the log-likelihood is then equivalent to minimizing the mean squared error. Since linear units do not saturate, they pose little difficulty for gradient-based optimization algorithms and may be used with a wide variety of optimization algorithms. Other kinds of frequently used output units are based on the use of sigmoid and softmax functions.
A sigmoid output unit is defined by:

$$\hat{y} = \sigma(\boldsymbol{w}^T\boldsymbol{h} + b)$$

Where σ is the logistic sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}}$. We can think of the sigmoid output unit as having two components. First, it uses a linear layer to compute $z = \boldsymbol{w}^T\boldsymbol{h} + b$. Next, it uses the sigmoid activation function to convert z into a probability. Having sigmoid outputs is useful when we want to deal with classification problems where the variable that we want to predict is binary.
Any time we wish to represent a probability distribution over a discrete variable with n possible values, we may use the softmax function. Softmax functions are most often used as the output of a classifier, to represent the probability distribution over n different classes. More rarely, softmax functions can be used inside the model itself, if we wish the model to choose between one of n different options for some internal variable.
To define a softmax unit we first need to apply a linear transformation (even though it is affine) to predict the unnormalized log probabilities:

$$\boldsymbol{z} = \boldsymbol{W}^T\boldsymbol{h} + \boldsymbol{b}$$

where $z_i = log\tilde{P}(y = i|\boldsymbol{x})$. The softmax function can then exponentiate and normalize $\boldsymbol{z}$ to obtain the desired $\hat{\boldsymbol{y}}$. Formally, the softmax function is given by the following formula:

$$softmax(\boldsymbol{z})_i = \frac{e^{(z_i)}}{\sum_j e^{(z_j)}}$$

As with the logistic sigmoid, the use of the exponential function works well when training the softmax to output a target value using maximum log-likelihood as the loss function.
Like the sigmoid, the softmax activation can saturate. The sigmoid function has a single output that saturates when its input is extremely negative or extremely positive. The softmax has multiple output values. These output values can saturate when the differences between input values become extreme. When the softmax saturates, many cost functions based on the softmax also saturate, unless they can invert the saturating activating function.

Hidden units:

A fundamental choice in neural networks' design regards the type of hidden unit to use in the hidden layers of the model. **Rectified linear units** (**ReLU**) are an excellent default choice of hidden unit.
Rectified linear units use the activation function $g(z) = \max\{0, z\}$.
These units are easy to optimize because they are like linear units. The only difference between a linear unit and a rectified linear unit is that a rectified
linear unit outputs zero across half its domain. This makes the derivatives through a rectified linear unit remain large whenever the unit is active. The gradients are not only large but also consistent. The second derivative of the rectifying operation is 0 almost everywhere, and the derivative of the rectifying operation is 1 everywhere that the unit is active. This means that the gradient direction is far more useful for learning than it would be with activation functions that introduce second-order effects.
Rectified linear units are typically used on top of an affine transformation:

$$\boldsymbol{h} = g(\boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b})$$

When initializing the parameters of the affine transformation, it can be a good practice to set all elements of $\boldsymbol{b}$ to a small positive value, such as 0.1. Doing so makes it very likely that the rectified linear units will be initially active for most inputs in the training set and allow the derivatives to pass through. Several generalizations of rectified linear units exist (**Absolute value rectification**, **Leaky ReLU**, **parametric ReLU**). However, since most of these generalizations perform comparably to rectified linear units, those will not be taken in consideration.
One drawback to rectified linear units is that they cannot learn via gradient-
based methods on examples for which their activation is zero, but this obstacle can be overcome by using some of the generalization mentioned above.

Neural networks architecture:

Another key design consideration for neural networks is determining the **architecture**. The word architecture refers to the overall structure of the network: how many units it should have and how these units should be connected to each other.
Most neural networks are organized into groups of units called **layers**. Most
neural network architectures arrange these layers in a chain structure, with each layer being a function of the layer that preceded it. In this structure, the first layer is given by:

$$\boldsymbol{h}^{(1)} = g^{(1)}(\boldsymbol{W}^{(1)T}\boldsymbol{x} + \boldsymbol{b}^{(1)})$$

the second layer is given by:

$$\boldsymbol{h}^{(2)} = g^{(2)}(\boldsymbol{W}^{(2)T}\boldsymbol{x} + \boldsymbol{b}^{(2)})$$

and so on.
In these chain-based architectures, the main architectural considerations are choosing the depth of the network and the width of each layer. Deeper networks are often able to use far fewer units per layer and far fewer parameters, as well as frequently generalizing to the test set, but they also tend to be harder to optimize.
As mentioned before, feedforward networks with hidden layers provide a universal approximation framework. Especially, the **universal approximation theorem** states that a feedforward network with a linear output layer and at least one hidden layer with any "squashing" activation function (such as the logistic sigmoid activation function) can approximate any Borel measurable function from one finite-dimensional space to another with any desired nonzero amount of error, provided that the network is given enough hidden units. The derivatives of the feedforward network can also approximate the derivatives of the function arbitrarily well.
So according to this theorem, there exists a network large enough to achieve any degree of accuracy we desire, but the theorem does not say how large this network will be. In summary, a feedforward network with a single layer is sufficient to represent any function, but the layer may be infeasibly large and may fail to learn and generalize correctly. In many circumstances, using deeper models can reduce the number of units required to represent the desired function and can reduce the amount of generalization error.

Back-propagation algorithm:

When we use a feedforward neural network to accept an input $\boldsymbol{x}$ and produce an output $\hat{\boldsymbol{y}}$, information flows forward through the network. The input $\boldsymbol{x}$ provides the initial information that then propagates up to the hidden units at each layer and eventually produces $\hat{\boldsymbol{y}}$. This is called **forward propagation**. During training, forward propagation can continue onward until it produces a scalar cost $J(\boldsymbol{\theta})$.
The **back-propagation algorithm** allows the information from the cost to then flow backward through the network to compute the gradient.
The term back-propagation refers only to the method for computing the gradient, while another algorithm, such as stochastic gradient descent, is used to perform learning using this gradient.
Here we will describe how to compute the gradient $\nabla_{\boldsymbol{x}} f(\boldsymbol{x}, \boldsymbol{y})$ for an arbitrary function f, where $\boldsymbol{x}$ is a set of variables whose derivatives are desired, and $\boldsymbol{y}$ is an additional set of variables that are inputs to the function
but whose derivatives are not required. In learning algorithms, the gradient we most often require is the gradient of the cost function with respect to the parameters, $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$. The idea of computing derivatives by propagating information through a network is very general and can be used to compute values such as the Jacobian of a function f with multiple outputs.
Back-propagation, at its core, is an algorithm that computes the chain rule of calculus, with a specific order of operations that is highly efficient. Let x be a real number and let f and g both be functions mapping from a real number to a real number.
Suppose that $y = g(x)$ and $z = f\big(g(x)\big) = f(y)$. Then according to the chain rule:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

We can generalize this beyond the scalar case. Suppose that $\boldsymbol{x} \epsilon \mathbb{R}^m$, $\boldsymbol{y} \in \mathbb{R}^n$, g maps from $\mathbb{R}^m$ to $\mathbb{R}^n$, and f maps from $\mathbb{R}^n$ to $\mathbb{R}$. If $\boldsymbol{y} = g(\boldsymbol{x})$ and $\boldsymbol{z} = f(\boldsymbol{y})$, then:

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}$$

or equivalently in vector notation:

$$\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^T \nabla_{\boldsymbol{y}} z$$

where $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ is the $n\ x\ m$ Jacobian matrix of g.

From this we see that the gradient of a variable $\boldsymbol{x}$ can be obtained by multiplying a Jacobian matrix $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ by a gradient $\nabla_{\boldsymbol{y}} z$. The back-propagation algorithm consists of performing such a Jacobian-gradient product for each operation in the graph.

The back-propagation algorithm can be applied to tensors of arbitrary dimensionality, not merely to vectors. Conceptually, this is the same as back-propagation with vectors. The only difference is how the numbers are arranged in a grid to form a tensor.

Using the chain rule, it is straightforward to write down an algebraic expression for the gradient of a scalar with respect to any node in the computational graph that produced that scalar. But, evaluating that expression in a computer, introduces some extra considerations.

In fact, many subexpressions may be repeated several times within the overall expression for the gradient. Any procedure that computes the gradient will need to choose whether to store these subexpressions or to recompute them several times. In some cases, computing the same subexpression twice would simply be wasteful, while in other cases, computing the same subexpression twice could be a valid way to reduce memory consumption at the cost of higher runtime.

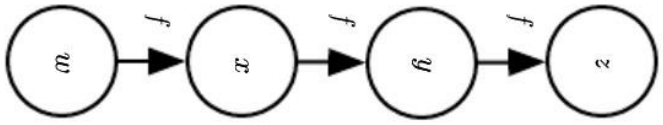


Figure 0-1: Example of a computational graph that results in repeated subexpressions when computing the gradient

In the above figure we can observe how the computation of the gradient happens. Let $w \in \mathbb{R}$ be the input to the graph. We use the same function $f: \mathbb{R} \to \mathbb{R}$ as the operation that we apply at every step of a chain: $x = f(w)$, $y = f(x)$, $z = f(y)$. To compute $\frac{\partial z}{\partial w}$, we apply equation the chain rule and obtain:

$$\frac{\partial z}{\partial w} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w} = f'(y)f'(x)f'(w)$$

Or analogously:

$$\frac{\partial z}{\partial w} = f'(f(f(w)))f'(f(w))f'(w)$$

The first of those two equations suggests an implementation in which we compute the value of $f(w)$ only once and store it in the variable x. This is the approach taken by the back-propagation algorithm. An alternative approach is suggested by the other equation, where the subexpression $f(w)$ appears more than once. In the alternative approach, $f(w)$ is recomputed each time it is needed. When the memory required to store the value of these expressions is low, the back-propagation approach is clearly preferable because of its reduced runtime. However, the other approach provides also a valid implementation of the chain rule and is useful when memory is limited.

Consider a computational graph describing how to compute a single scalar $u^{(n)}$ (for example the loss on a training data). This scalar is the quantity whose gradient we want to obtain, with respect to the n_i input nodes $u^{(1)}$ to $u^{(n_i)}$. In other words, we wish to compute $\frac{\partial u^{(n)}}{\partial u^{(i)}}$ for all $i \in \{1,2,\dots,n_i\}$. In the application of back-propagation to computing gradients for gradient descent over parameters, $u^{(n)}$ will be the cost associated with an example or a minibatch, while $u^{(1)}$ to $u^{(n_i)}$ correspond to the parameters of the model. We can assume that the nodes of the graph have been ordered in such a way that their output can be computed one after the other, starting at $u^{(n_i+1)}$ and going up to $u^{(n)}$. Each node $u^{(i)}$ is associated with an operation $f^{(i)}$ and is computed by evaluating the function $u^{(i)} = f(\mathbb{A}^{(i)})$ where $\mathbb{A}^{(i)}$ is the set of all nodes that are parents of $u^{(i)}$.
So, a procedure that performs the computations mapping n_i inputs $u^{(1)}$ to $u^{(n_i)}$ to an output $u^{(n)}$ can be outlined. We can define a computational graph where each node computes numerical value $u^{(i)}$ by applying a function $f^{(i)}$ to the set of arguments $A^{(i)}$ that comprises the values of previous nodes $u^{(j)}, j < i$, with $j \in Pa\left(u^{(i)}\right)$ (where $Pa\left(u^{(i)}\right)$ denotes the power set of $(u^{(i)})$). The input to the computational graph is the vector $\boldsymbol{x}$, and is set into the first n_i nodes $u^{(1)}$ to $u^{(n_i)}$. The output of the computational graph is read off the last (output) node $u^{(n)}$.
In algorithm notation we could define the forward propagation like this:

for $i = 1, \dots, n_i$ **do**
  $u^{(i)} \leftarrow x_i$
end for
for $i = n_i + 1, \dots, n$ **do**
  $\mathbb{A}^{(i)} \leftarrow \{u^{(j)} \mid j \in Pa(u^{(i)})\}$
  $u^{(i)} \leftarrow f^{(i)}(\mathbb{A}^{(i)})$
end for
return $u^{(n)}$

After those computations are done, we could put them in a graph $\boldsymbol{\mathcal{G}}$. To perform back-propagation, we can construct a computational graph that depends on $\boldsymbol{\mathcal{G}}$ and adds to it an extra set of nodes. These form a subgraph $\boldsymbol{\mathcal{B}}$ with one node per node of $\boldsymbol{\mathcal{G}}$. Computation in $\boldsymbol{\mathcal{B}}$ proceeds in exactly the reverse of the order of computation in $\boldsymbol{\mathcal{G}}$, and each node of $\boldsymbol{\mathcal{B}}$ computes the derivative $\frac{\partial u^{(n)}}{\partial u^{(i)}}$ associated with the forward graph node $u^{(i)}$. This is done using the chain rule with respect to scalar output $u^{(n)}$:

$$\frac{\partial u^{(n)}}{\partial u^{(i)}} = \sum_{i:j \in Pa(u^{(i)})} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}}$$

The subgraph $\boldsymbol{\mathcal{B}}$ contains exactly one edge for each edge from node $u^{(j)}$ to node $u^{(i)}$ of $\boldsymbol{\mathcal{G}}$. The edge from $u^{(j)}$ to $u^{(i)}$ is associated with the computation of $\frac{\partial u^{(i)}}{\partial u^{(j)}}$. In addition, a dot product is performed for each node, between the gradient already computed with respect to nodes $u^{(i)}$ that are children of $u^{(j)}$ and the vector containing the partial derivatives $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ for the same children nodes $u^{(i)}$. To summarize, the amount of computation required for performing the back-propagation scales linearly with the number of edges in $\boldsymbol{\mathcal{G}}$, where the computation for each edge corresponds to computing a partial derivative (of one node with respect to one of its parents) as well as performing one multiplication and one addition.

To generalize to some extent (without the use of tensor notation, but just by using matrices) what has already been said, by considering a fully connected multi-layer MLP we will redefine the forward propagation and then provide an algorithm for back-propagation.

Forward propagation through a typical deep neural network and the computation of the cost function:

With forward propagation we map parameters to the supervised loss $L(\hat{\boldsymbol{y}}, \boldsymbol{y})$ associated with a single (input, target) training example $(\boldsymbol{x}, \boldsymbol{y})$, with $\hat{\boldsymbol{y}}$ the output of the neural network when $\boldsymbol{x}$ is provided in input.
The loss $L(\hat{\boldsymbol{y}}, \boldsymbol{y})$ depends on the output $\hat{\boldsymbol{y}}$ and on the target $\boldsymbol{y}$. To obtain the total cost J, the loss may be added to a regularizer $\Omega(\theta)$, where θ contains all the parameters (weights and biases). For simplicity we will consider only a single input example $\boldsymbol{x}$. Practical applications should use a minibatch.

The algorithm proceeds as follows:

Require: Network depth, l
Require: $\boldsymbol{W}^{(i)}, i \in \{1, \ldots, l\}$, the weight matrices of the model
Require: $\boldsymbol{b}^{(i)}, i \in \{1, \ldots, l\}$, the bias parameters of the model
Require: $\boldsymbol{x}$, the input to process
Require: $\boldsymbol{y}$, the target output

$\boldsymbol{h}^{(0)} = \boldsymbol{x}$
for $k = 1, \ldots, l$ **do**
  $\boldsymbol{a}^{(k)} = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)} \boldsymbol{h}^{(k-1)}$
  $\boldsymbol{h}^{(k)} = f(\boldsymbol{a}^{(k)})$
end for
$\hat{\boldsymbol{y}} = \boldsymbol{h}^{(l)}$
$J = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \lambda \Omega(\theta)$

The backward computation for the deep neural network after the first step defined by the previous algorithm, which uses, in addition to the input $\boldsymbol{x}$, a target $\boldsymbol{y}$, can be defined as follows:

After the forward computation, compute the gradient on the output layer:
$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, \boldsymbol{y})$
for $k = l, l-1, \ldots, 1$ **do**
Convert the gradient on the layer's output into a gradient on the pre-nonlinearity activation (element-wise multiplication if f is element-wise)

$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'(\boldsymbol{a}^{(k)})$
Compute gradients on weights and biases (including the regularization term where needed):
$\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^{(k)}} \Omega(\theta)$
$\nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g}\, \boldsymbol{h}^{(k-1)\top} + \lambda \nabla_{\boldsymbol{W}^{(k)}} \Omega(\theta)$
Propagate the gradients w.r.t. the next lower-level hidden layer's activations:
$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)\top} \boldsymbol{g}$
end for

This computation yields the gradients on the activations $\boldsymbol{a}^{(k)}$ for each layer k, starting from the output layer and going backwards to the first hidden layer. From these gradients, which can be interpreted as an indication of how each layer's output should change to reduce error, one can obtain the gradient on the parameters of each layer. The gradients on weights and biases can be immediately used as part of a stochastic gradient update (performing the update right after the gradients have been computed) or used with other gradient-based optimization methods.

Some approaches to back-propagation take a computational graph and a set of numerical values for the inputs to the graph, then return a set of numerical values describing the gradient at those input values. We call this approach **symbol-to-number differentiation**. This is the approach used by libraries such as Torch and Caffe. Another approach is to take a computational graph and add additional nodes to the graph that provide a symbolic description of the desired derivatives. This is the approach taken by Theano and TensorFlow (**symbol-to-symbol** approach).
An example of this last approach is given below:

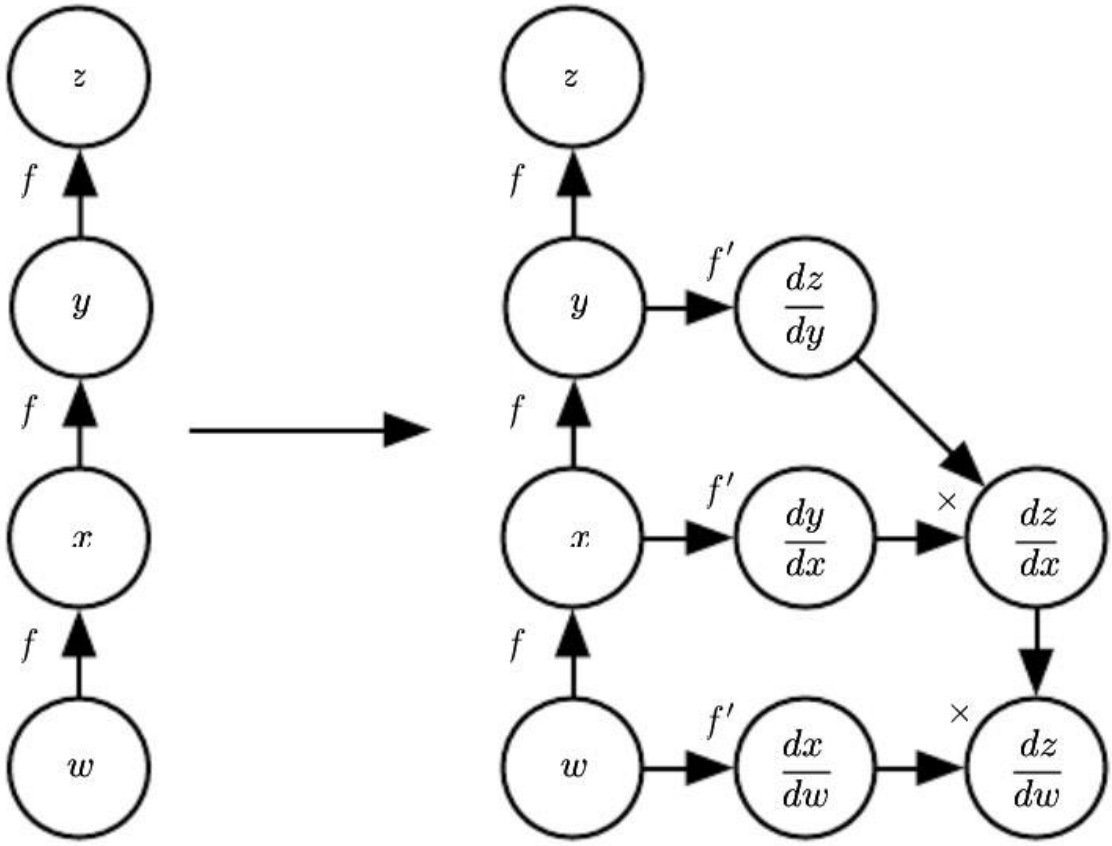


Figure 0-2: An example of the symbol-to-symbol approach to computing derivatives

By looking at the above figure, we can notice on the left a graph representing $z = f(f(f(w)))$. On the right, we run the back-propagation algorithm, instructing it to construct the graph for the expression corresponding to $\frac{dz}{dw}$. The figure clearly illustrates what the desired result is: a computational graph with a symbolic description of the derivative.

The primary advantage of this approach is that the derivatives are described in the same language as the original expression. Because the derivatives are just another computational graph, it is possible to run back-propagation again, differentiating the derivatives to obtain higher derivatives.
The symbol-to-number approach can be understood as performing the same computations as are done in the graph built by the symbol-to-symbol approach. The key difference is that the symbol-to-number approach does not expose the graph.

Regularization for deep learning

Regularization can be defined as "any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error."
There are many regularization strategies. Some put extra constraints on a machine learning model, such as adding restrictions on the parameter values. Some add extra terms in the objective function that can be thought of as corresponding to a soft constraint on the parameter values.
In the context of deep learning, most regularization strategies are based on

regularizing estimators. Regularization of an estimator works by trading increased bias for reduced variance, to move from an overfitting regime to one that matches the true generating process. An effective regularizer is one that makes a profitable trade, reducing variance significantly while not overly increasing the bias. Controlling the complexity of the model is not a
simple matter of finding the model of the right size, with the right number of parameters. Often times the best fitting model (in the sense of minimizing generalization error) is a large model that has been regularized appropriately.

Many regularization approaches are based on limiting the capacity of models by adding a parameter norm penalty $\Omega(\boldsymbol{\theta})$ to the objective function
J. We denote the regularized objective function by $\tilde{J}$:

$$\tilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha\Omega(\boldsymbol{\theta})$$

where $\alpha \in [0, \infty)$ is a hyperparameter that weights the relative contribution of the norm penalty term, Ω, relative to the standard objective function J. Setting α to 0 results in no regularization. Larger values of α correspond to more regularization.
For neural networks, we typically choose to use a parameter norm penalty Ω that penalizes only the weights of the affine transformation at each layer and leaves the biases unregularized. The biases typically require less data than the weights to fit accurately. Each weight specifies how two variables interact. Fitting the weight well requires observing both variables in a variety of conditions. Each bias controls only a single variable. This means that we do not induce too much variance by leaving the biases unregularized. Also, regularizing the bias parameters can introduce a significant amount of underfitting. We therefore use the vector $\boldsymbol{w}$ to indicate all the weights that should be affected by a norm penalty, while the vector $\boldsymbol{\theta}$ denotes all the parameters, including both $\boldsymbol{w}$ and the unregularized parameters.

Norm penalties as constrained optimization:

Generally, we can minimize a function subject to constraints by constructing a generalized Lagrange function, consisting of the original objective function plus a set of penalties. Each penalty is a product between a coefficient, called a Karush–Kuhn–Tucker (KKT) multiplier, and a function representing whether the constraint is satisfied. If we wanted to constrain $\Omega(\boldsymbol{\theta})$ to be less than some constant k, we could construct a generalized Lagrange function:

$$\mathcal{L}(\boldsymbol{\theta}, \alpha; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha(\Omega(\boldsymbol{\theta}) - k)$$

The solution to the constrained problem is given by:

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \max_{\alpha, \alpha \geq 0} \mathcal{L}(\boldsymbol{\theta}, \alpha)$$

Solving this problem requires modifying both $\boldsymbol{\theta}$ and α. Many different procedures are possible to solve this problem, some may use gradient descent, while others may use analytical solutions for where the gradient is zero, but in all procedures α must increase whenever $\Omega(\boldsymbol{\theta}) > k$ and decrease whenever $\Omega(\boldsymbol{\theta}) < k$. All positive α encourage $\Omega(\boldsymbol{\theta})$ to shrink. The optimal value α^*will encourage $\Omega(\boldsymbol{\theta})$ to shrink, but not so strongly to make $\Omega(\boldsymbol{\theta})$ become less than k. Then if we fix α^* and view the problem as just a function of $\boldsymbol{\theta}$:

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}, \alpha^*) = \arg\min_{\boldsymbol{\theta}} J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha^* \Omega(\boldsymbol{\theta})$$

This is last equation coincides with the regularized training problem of minimizing $\tilde{J}$. We can thus think of a parameter norm penalty as imposing a constraint on the weights. If Ω is the L^2 norm, then the weights are constrained to lie in an L^2 ball. If Ω is the L^1 norm, then the weights are constrained to lie in a region of limited L^1 norm. Usually, we do not

know the size of the constraint region that we impose by using weight decay with coefficient α^*because the value of α^*does not directly tell us the value of k. In principle, one can solve for k, but the relationship between k and α^* depends on the form of J. While we do not know the exact size of the constraint region, we can control it roughly by increasing or decreasing α to grow or shrink the constraint region. Larger α will result in a smaller constraint region. Smaller α will result in a larger constraint region.

Early stopping:

When training large models with sufficient representational capacity to overfit the task, we often observe that training error decreases steadily over time, but validation set error begins to rise again.
This behavior is shown in the following image:

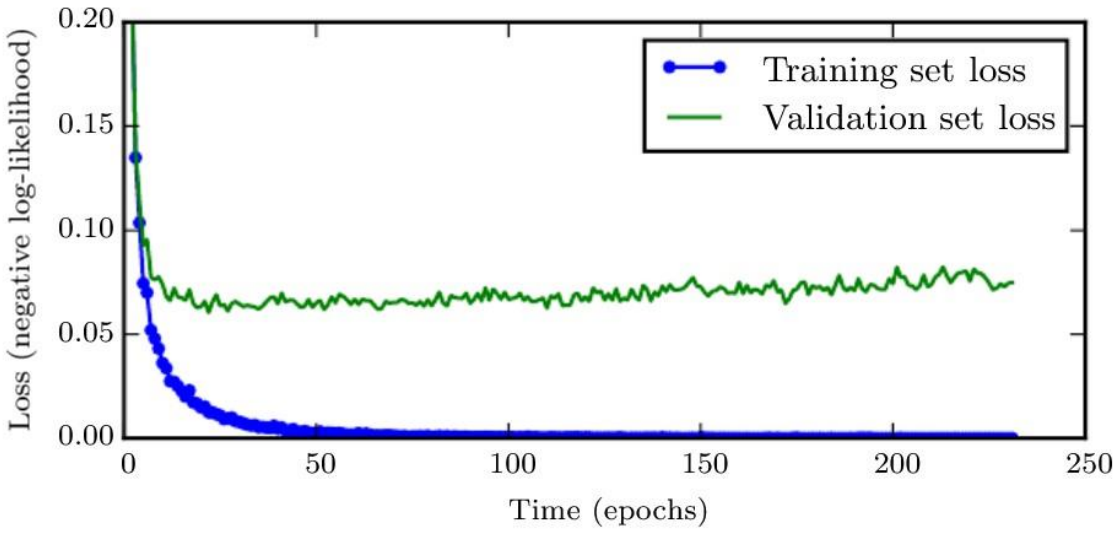


Figure 0-3: Example of how the training and validation set losses behave over time (epochs)

This means that we can obtain a model with better validation set error (and thus, hopefully better test set error) by returning to the parameter setting at the point in time with the lowest validation set error. Every time the error on the validation set improves, we store a copy of the model parameters. When the training algorithm terminates, we return these parameters, rather than the latest parameters. The algorithm terminates when no parameters have improved over the best recorded validation error for some pre-specified number of iterations. This strategy is known as **early stopping**. It is probably the most used form of regularization in deep learning. Its popularity is due to both its effectiveness and its simplicity. With early stopping we are controlling the effective capacity of the model by determining how many steps it can take to fit the training set.
In algorithm terms we can write:

Let n be the number of steps between evaluations.
Let p be the "patience," the number of times to observe worsening validation set error before giving up.
Let $\boldsymbol{\theta}_o$ be the initial parameters.
$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}_o$
$i \leftarrow 0$
$j \leftarrow 0$
$v \leftarrow \infty$
$\boldsymbol{\theta}^* \leftarrow \boldsymbol{\theta}$
$i^* \leftarrow i$
while $j < p$ **do**
  Update $\boldsymbol{\theta}$ by running the training algorithm for n steps.
  $i \leftarrow i + n$
  $v' \leftarrow \text{ValidationSetError}(\boldsymbol{\theta})$
  if $v' < v$ **then**
    $j \leftarrow 0$
    $\boldsymbol{\theta}^* \leftarrow \boldsymbol{\theta}$
    $i^* \leftarrow i$
    $v \leftarrow v'$
  else
    $j \leftarrow j + 1$
  end if
end while
Best parameters are $\boldsymbol{\theta}^*$, best number of training steps is i^*.

The most significant cost for choosing the "training time" hyperparameter automatically via early stopping is running the validation set evaluation periodically during training. An additional cost to early stopping is the need to maintain a copy of the best parameters. This cost is generally negligible, because it is acceptable to store these parameters in a slower and larger form of memory (for example, training in GPU memory, but storing the optimal parameters in host memory or on a disk drive). Early stopping is an unobtrusive form of regularization, in that it requires almost no change in the underlying training procedure, the objective function, or the set of allowable parameter values. This means that it is easy to use early stopping without damaging the learning dynamics.

Optimization in neural network training

Optimization algorithms used for training of deep models differ from traditional optimization algorithms in several ways. Machine learning usually acts indirectly. In most machine learning scenarios, we care about some performance measure P, that is defined with respect to the test set and may also be intractable. We therefore optimize P only indirectly. We reduce a different cost function $J(\boldsymbol{\theta})$ in the hope that doing so will improve P. This contrasts with pure optimization, where minimizing J is a goal in and of itself. Typically, the cost function can be written as an average over the training set, such as:

$$J(\boldsymbol{\theta}) = E_{(\boldsymbol{x},y)\sim\hat{p}_{data}} L(f(\boldsymbol{x};\boldsymbol{\theta}), y)$$

where L is the per-example loss function $f(\boldsymbol{x};\boldsymbol{\theta})$ is the predicted output when the input is $\boldsymbol{x}$, and $\hat{p}_{data}$ is the empirical distribution. In the supervised learning case, y is the target

output. Throughout this section, we develop the unregularized supervised case, where the arguments to L are $f(\boldsymbol{x};\boldsymbol{\theta})$ and y. This development could be extended, for example, to include $\boldsymbol{\theta}$ or $\boldsymbol{x}$ as arguments, or to exclude y as arguments, to develop various forms of regularization or unsupervised learning.
Instead of the previous equation, we would usually prefer to minimize the corresponding objective function where the expectation is taken across the **data-generating distribution** p_{data} rather than just over the finite training set:

$$J^*(\boldsymbol{\theta}) = E_{(\boldsymbol{x},y)\sim p_{data}} L(f(\boldsymbol{x};\boldsymbol{\theta}), y)$$

Empirical risk minimization:

The goal of a machine learning algorithm is to reduce the expected generalization error given by the previous equation. This quantity is known as the **risk**. If we knew the true distribution $p_{data}(\boldsymbol{x}, y)$, risk minimization would be an optimization task solvable by an optimization algorithm. When we do not know $p_{data}(\boldsymbol{x}, y)$, but only have a training set of samples, we have a machine learning problem. The simplest way to convert a machine learning problem back into an optimization problem is to minimize the expected loss on the training set. This means replacing the true distribution $p(\boldsymbol{x}, y)$ with the empirical distribution $\hat{p}(\boldsymbol{x}, y)$ defined by the training set. By following this way of thinking, we can minimize the **empirical risk**:

$$E_{(\boldsymbol{x},y)\sim \hat{p}_{data(x,y)}}[L(f(\boldsymbol{x};\boldsymbol{\theta}), y)] = \frac{1}{m}\sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), y^{(i)})$$

where m is the number of training examples.
The training process based on minimizing this average training error is known as **empirical risk minimization**. In this setting, machine learning is still very similar to straightforward optimization. Rather than optimizing the risk directly, we optimize the empirical risk and hope that the risk decreases significantly as well. As one could suggest, empirical risk minimization is prone to overfitting. The most effective modern optimization algorithms are based on gradient descent, but many useful loss functions, such as *0-1 loss*, have no useful derivatives (the derivative is either zero or undefined everywhere). These two problems mean that, in the context of deep learning, we rarely use empirical risk minimization. Instead, we must use a slightly different approach, in which the quantity that we optimize is even more different from the quantity that we truly want to optimize. In those cases, we will define a **surrogate loss function**.

Batch and minibatch algorithms:

Optimization algorithms that use the entire training set are called **batch** or **deterministic** gradient methods, because they process all the training examples simultaneously in a large batch. Optimization algorithms that use only a single example at a time are called **stochastic**. Most algorithms used for deep learning fall somewhere in between, using more than one but fewer than all the training examples. These were traditionally called **minibatch** or **minibatch stochastic methods**, and it is now common to call them simply **stochastic** methods.

Stochastic gradient descent:

Stochastic gradient descent (**SGD**) and its variants are probably the most used optimization algorithms for deep learning. It is possible to obtain an unbiased estimate of

the gradient by taking the average gradient on a minibatch of m examples drawn independent identically distributed (i.i.d) from the data-generating distribution.
The algorithm is defined as follows:

Require: Learning rate schedule $\epsilon_1, \epsilon_2, \ldots$
Require: Initial parameter $\boldsymbol{\theta}$
$k \leftarrow 1$
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon_k \hat{\boldsymbol{g}}$
$k \leftarrow k + 1$
end while

A crucial parameter for the SGD algorithm is the learning rate. It is necessary to gradually decrease the learning rate over time, so we denote the learning rate at iteration k as ϵ_k. This is because the SGD gradient estimator introduces a source of noise (the random sampling of m training examples) that does not vanish even when we arrive at a minimum. By comparison, the true gradient of the total cost function becomes small and then **0** when we approach and reach a minimum using batch gradient descent, so batch gradient descent can use a fixed learning rate. A sufficient condition to guarantee convergence of SGD is that:

$$\sum_{k=1}^{\infty} \epsilon_k = \infty$$

and

$$\sum_{k=1}^{\infty} \epsilon_k^2 < \infty$$

In practice is common to decay the learning rate linearly until iteration τ:

$$\epsilon_k = (1 - \alpha)\epsilon_0 + \alpha\epsilon_\tau$$

with $\alpha = \frac{k}{\tau}$. After the iteration τ, it is common to leave ϵ constant. The learning rate may be chosen by trial and error, but it is usually best to choose it by monitoring learning curves that plot the objective function as a function of time. Typically, the optimal initial learning rate, in terms of total training time and the final cost value, is higher than the learning rate that yields the best performance after the first 100 iterations or so. Therefore, it is usually best to monitor the first several iterations and use a learning rate that is higher than the best-performing learning rate at this time, but not so high that it causes severe instability.

Momentum:

The method of **momentum** is designed to accelerate learning, especially in the face of high curvature, small but consistent gradients, or noisy gradients. The momentum algorithm accumulates an exponentially decaying moving average of past gradients and continues to move in their direction. Formally, the momentum algorithm introduces a variable $\boldsymbol{v}$ that plays the role of velocity; indeed, it is the direction and speed at which the parameters move through parameter space. The velocity is set to an exponentially decaying average of the negative gradient. The name momentum derives from a physical analogy, in which the negative gradient is a force moving a particle through parameter

space, according to Newton's laws of motion. Momentum in physics is mass times velocity. In the momentum learning algorithm, we assume unit mass, so the velocity vector $\boldsymbol{v}$ may also be regarded as the momentum of the particle. A hyperparameter $\alpha \in [0,1)$ determines how quickly the contributions of previous gradients exponentially decay.
The update rule is given by:

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} L(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}) \right)$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$$

The velocity $\boldsymbol{v}$ accumulates the gradient elements $\nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} L(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}) \right)$. The larger α is relative to ϵ, the more previous gradients affect the current direction.
The SGD algorithm with momentum can be described as follows:

Require: Learning rate ϵ, momentum parameter α
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient estimate: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$.
Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$.
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$.
end while

With SCG the size of the step was simply the norm of the gradient multiplied by the learning rate. Now, the size of the step depends on how large and how aligned a sequence of gradients are. The step size is largest when many successive gradients point in the same direction. If the momentum algorithm always observes gradient $\boldsymbol{g}$, then it will accelerate in the direction of $-\boldsymbol{g}$, until reaching a terminal velocity where the size of each step is:

$$\frac{\epsilon ||\boldsymbol{g}||}{1 - \alpha}$$

It is thus helpful to think of the momentum hyperparameter in terms of $\frac{1}{1-\alpha}$. For example, $\alpha = 0.9$ corresponds to multiplying the maximum speed by 10 relative to the gradient descent algorithm. Common values of α used in practice include 0.5, 0.9, and 0.99. Like the learning rate, α may also be adapted over time. Typically, it begins with a small value and is later raised. However, adapting α over time is less important than shrinking ϵ over time.
An example of how SGD with momentum works is given in the image below:

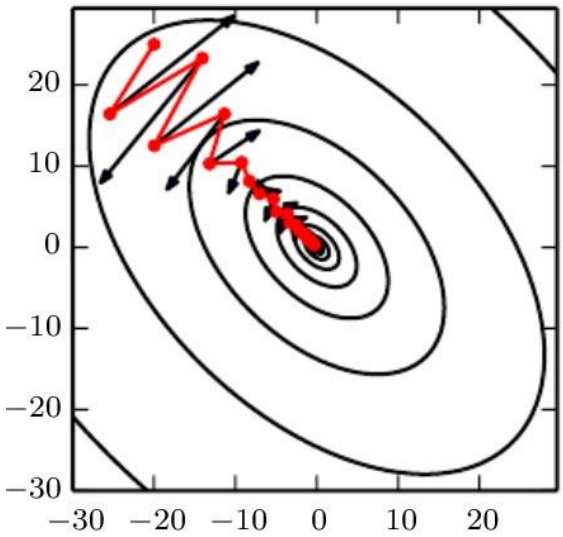


Figure 0-4: Example of SGD with momentum applied to a problem where there is a quadratic loss function with a poorly conditioned hessian matrix

The red path cutting across the contours indicates the path followed by the momentum learning rule as it minimizes this function. At each step along the way, we draw an arrow indicating the step that gradient descent would take at that point. We can see that a poorly conditioned quadratic objective function looks like a long, narrow valley or canyon with steep sides. Momentum correctly traverses the canyon lengthwise, while gradient steps waste time moving back and forth across the narrow axis of the canyon.

A popular variant of the momentum is the **Nesterov momentum** method, however, it will not be discussed here.

RMSProp:

RMSProp is an adaptive learning rate optimization algorithm, and it has been shown to be an effective and practical optimization algorithm for deep neural networks. It is currently one of the go-to optimization methods being employed routinely by deep learning practitioners.
It can be described as follows:

Require: Global learning rate ϵ, decay rate ρ
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant δ, usually 10^{-6}, used to stabilize division by small numbers
Initialize accumulation variables $\boldsymbol{r} = 0$
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$.
Accumulate squared gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$.
Compute parameter update: $\Delta \boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$. $(\frac{1}{\sqrt{\delta + \boldsymbol{r}}}$ applied element-wise)
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$.
end while

Adam:

Adam is another adaptive learning rate optimization algorithm. The name "Adam" derives from the phrase "adaptive moments." It can be seen as a variant on the combination of RMSProp and momentum with a few important distinctions. First, in Adam, momentum is incorporated directly as an estimate of the first-order moment (with exponential weighting) of the gradient. The most straightforward way to add momentum to RMSProp is to apply momentum to the rescaled gradients. Second, Adam includes bias corrections to the estimates of both the first-order moments (the momentum term) and the (uncentered) second-order moments to account for their initialization at the origin. RMSProp also incorporates an estimate of the (uncentered) second-order moment; however, it lacks the correction factor. Thus, unlike in Adam, the RMSProp second-order moment estimate may have high bias early in training. Adam is generally regarded as being fairly robust to the choice of hyperparameters, though the learning rate sometimes needs to be changed from the suggested default

Algorithmically speaking, Adam works as follows:

Require: Step size ϵ (Suggested default: 0.001)

Require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$. (Suggested defaults: 0.9 and 0.999 respectively)

Require: Small constant δ used for numerical stabilization (Suggested default: 10^{-8})

Require: Initial parameters $\boldsymbol{\theta}$

Initialize 1st and 2nd moment variables $\boldsymbol{s} = \mathbf{0}$, $\boldsymbol{r} = \mathbf{0}$

Initialize time step $t = 0$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

$t \leftarrow t + 1$

Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$

Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$

Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1 - \rho_1^t}$

Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1 - \rho_2^t}$

Compute update: $\Delta\boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$ (operations applied element-wise)

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

end while

Recurrent neural networks (RNN)

Recurrent neural networks, or **RNNs**, are a family of neural networks for processing sequential data. These neural networks are specialized for processing a sequence of values $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(\tau)}$. Parameter sharing makes it possible to extend and apply the model to examples of different forms (in this case different lengths) and generalize across them. If we had separate parameters for each value of the time index, we could not generalize to sequence lengths not seen during training, nor share statistical strength across different sequence lengths and across different positions in time.

Such sharing is particularly important when a specific piece of information can occur at multiple positions within the sequence.
Recurrent networks share parameters in this way: each member of the output is a function of the previous members of the output (it is produced using the same update rule applied to the previous outputs). This recurrent formulation results in the sharing of parameters through a very deep computational graph. For the simplicity of exposition, we refer to RNNs as operating on a sequence that contains vectors $\boldsymbol{x}^{(t)}$ with the time step index t ranging from 1 to τ. In practice, recurrent networks usually operate on minibatches of such sequences, with a different sequence length τ for each member of the minibatch. This section extends the idea of a computational graph to include cycles. These cycles represent the influence of the present value of a variable on its own value at a future time step. Such computational graphs allow us to define recurrent neural networks.

Unfolding computational graphs:

In this section we explain the idea of unfolding a recursive or recurrent computation into a computational graph that has a repetitive structure, typically corresponding to a chain of events. Unfolding this graph results in the sharing of parameters across a deep network structure.
If we consider the classical form of a dynamical system:

$$\boldsymbol{s}^{t} = f(\boldsymbol{s}^{(t-1)}; \boldsymbol{\theta})$$

where $\boldsymbol{s}^{(t)}$ is called the state of the system.
The above in equation is recurrent because the definition of $\boldsymbol{s}$ at time t refers back to the same definition at time $t-1$. For a finite number of time steps τ, the graph can be unfolded by applying the definition $\tau - 1$ times. For example, if we unfold the previous equation for $\tau = 3$ time steps, we obtain:

$$\boldsymbol{s}^{(3)} = f\big(\boldsymbol{s}^{(2)}; \boldsymbol{\theta}\big) = f(f\big(\boldsymbol{s}^{(1)}; \boldsymbol{\theta}\big); \boldsymbol{\theta})$$

Unfolding the equation by repeatedly applying the definition in this way has yielded an expression that does not involve recurrence. Such an expression can now be represented by a traditional directed acyclic computational graph. An example of this representation is given in the following image:

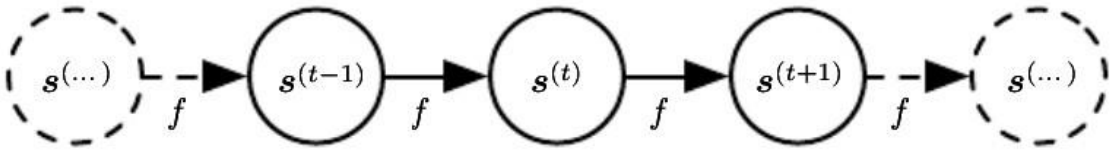


Figure 0-5: Example of a dynamical system illustrated as an unfolded computational graph

Many recurrent neural networks use the following equation or a similar equation to define the values of their hidden units:

$$\boldsymbol{h}^{(t)} = f(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta})$$

To indicate that the state is the hidden units of the network, we now use the variable $\boldsymbol{h}$ to represent the state, while $\boldsymbol{x}$ represents the information obtained from the input. An example of a recurrent network defined just by the above equation (with no outputs) can be seen below:

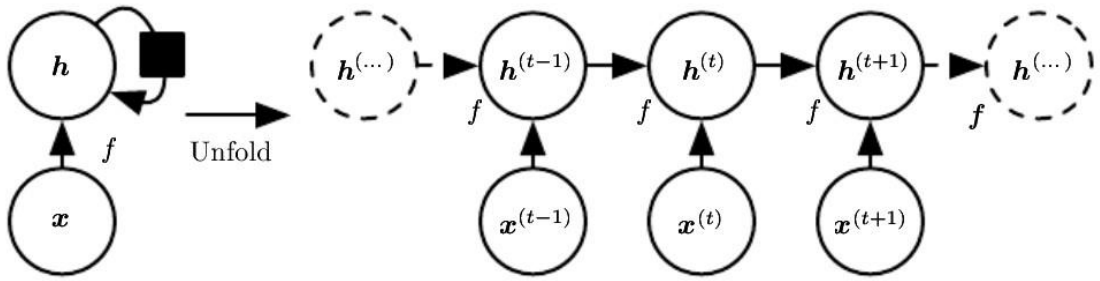


Figure 0-6: Example of a recurrent network with no outputs

In the above picture on the left, in the circuit diagram, the black square indicates a delay of a single time step. The same network can be seen on the right as an unfolded computational graph, where each node is now associated with one particular time instance.

When the recurrent network is trained to perform a task that requires predicting the future from the past, the network typically learns to use $\boldsymbol{h}^{(t)}$ as a kind of lossy summary of the task-relevant aspects of the past sequence of inputs up to t. This summary is in general necessarily lossy since it maps an arbitrary length sequence $(\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, \boldsymbol{x}^{(t-2)}, \dots, \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(1)})$ to a fixed length vector $\boldsymbol{h}^{(t)}$. Depending on the training criterion, this summary might selectively keep some aspects of the past sequence with more precision than other aspects.

We can represent the unfolded recurrence after t steps with a function $g^{(t)}$:

$$\boldsymbol{h}^{(t)} = g^{(t)}(\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, \boldsymbol{x}^{(t-2)}, \dots, \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(1)}) = f(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta})$$

The function $g^{(t)}$ takes the whole past sequence $(\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, \boldsymbol{x}^{(t-2)}, \dots, \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(1)})$ as input and produces the current state, but the unfolded recurrent structure allows us to factorize $g^{(t)}$ into repeated application of a function f. The unfolding process thus introduces two major advantages:

1. Regardless of the sequence length, the learned model always has the same input size, because it is specified in terms of transition from one state to another state, rather than specified in terms of a variable length history of states.
2. It is possible to use the same transition function f with the same parameters at every time step.

These two factors make it possible to learn a single model f that operates on all time steps and all sequence lengths, rather than needing to learn a separate model $g^{(t)}$ for all possible time steps. Learning a single shared model allows generalization to sequence lengths that did not appear in the training set and enables the model to be estimated with far fewer training examples than would be required without parameter sharing.

Given the graph unrolling and parameter sharing defined before, we can design a wide variety of recurrent neural networks, among them the most important ones are:

- Recurrent networks that produce an output at each time step and have recurrent connections between hidden units, illustrated in the following picture:

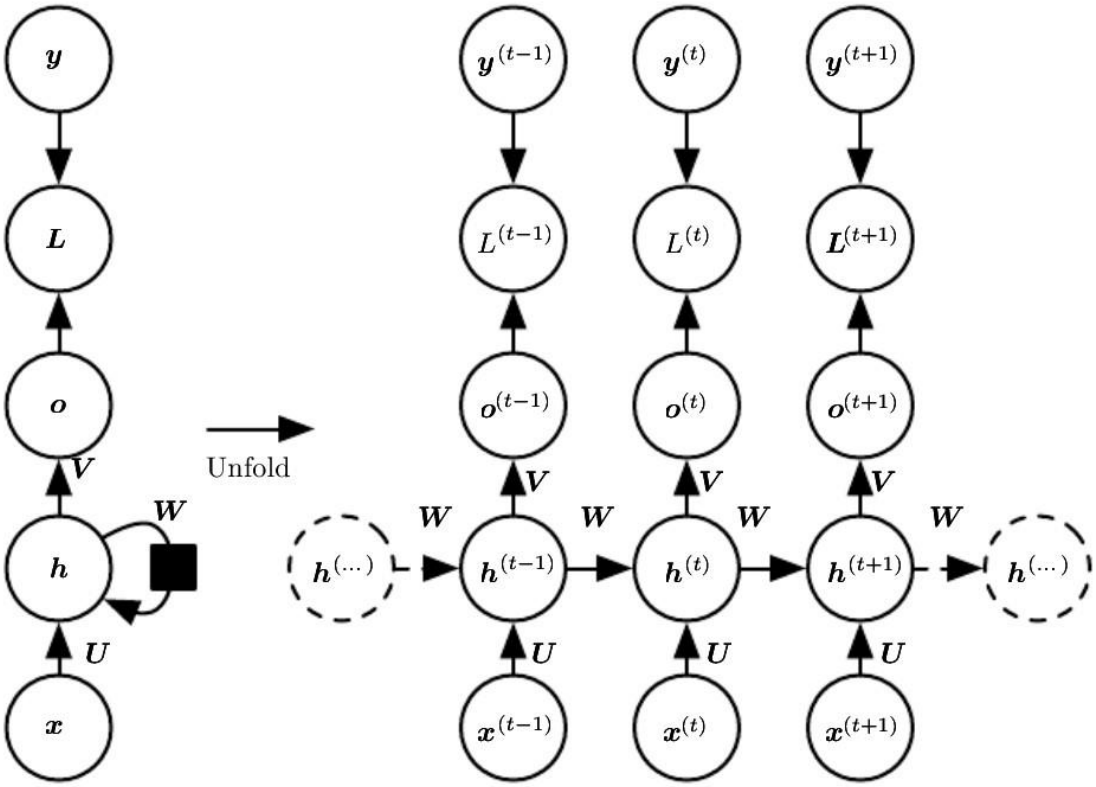


*Figure 0-7:The computational graph to compute the training loss of a recurrent network that maps an input sequence of **x** values to a corresponding sequence of output **o** values*

In the image above, a loss L measures how far each $\boldsymbol{o}$ is from the corresponding training target $\boldsymbol{y}$. When using softmax outputs, we assume $\boldsymbol{o}$ is the unnormalized log probabilities. The loss L internally computes:

$$\hat{\boldsymbol{y}} = softmax(\boldsymbol{o})$$

and compares this to the target $\boldsymbol{y}$. The RNN has input to hidden connections parametrized by a weight matrix $\boldsymbol{U}$, hidden-to-hidden recurrent connections parametrized by a weight matrix $\boldsymbol{W}$, and hidden-to-output connections parametrized by a weight matrix $\boldsymbol{V}$. The forward propagation in this model is defined by the following equations (for each time step from $t = 1$ to $t = \tau$):

$$\boldsymbol{a}^{(t)} = \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{t}$$
$$\boldsymbol{h}^{(t)} = \tanh(\boldsymbol{a}^{(t)})$$
$$\boldsymbol{o}^{(t)} = \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)}$$
$$\hat{\boldsymbol{y}}^{(t)} = softmax(\boldsymbol{o}^{(t)})$$

where the parameters are the bias vectors $\boldsymbol{b}$ and $\boldsymbol{c}$ along with the weight matrices $\boldsymbol{U}, \boldsymbol{V}$ and $\boldsymbol{W}$, respectively, for input-to-hidden, hidden-to-output, and hidden-to-hidden connections. Notice that in the previous equations the choice regarding the activation function for the hidden units, the form taken by the output and the loss function were made arbitrarily, however, the previous model can be generalized to different scenarios as well.

On the left of the image, the RNN and its loss drawn with recurrent connections can be seen. On the right, the same RNN can be seen as a time-unfolded computational graph, where each node is now associated with one particular time instance.

- Recurrent networks that produce an output at each time step and have recurrent connections only from the output at one time step to the hidden units at the next time step, illustrated in the following picture:

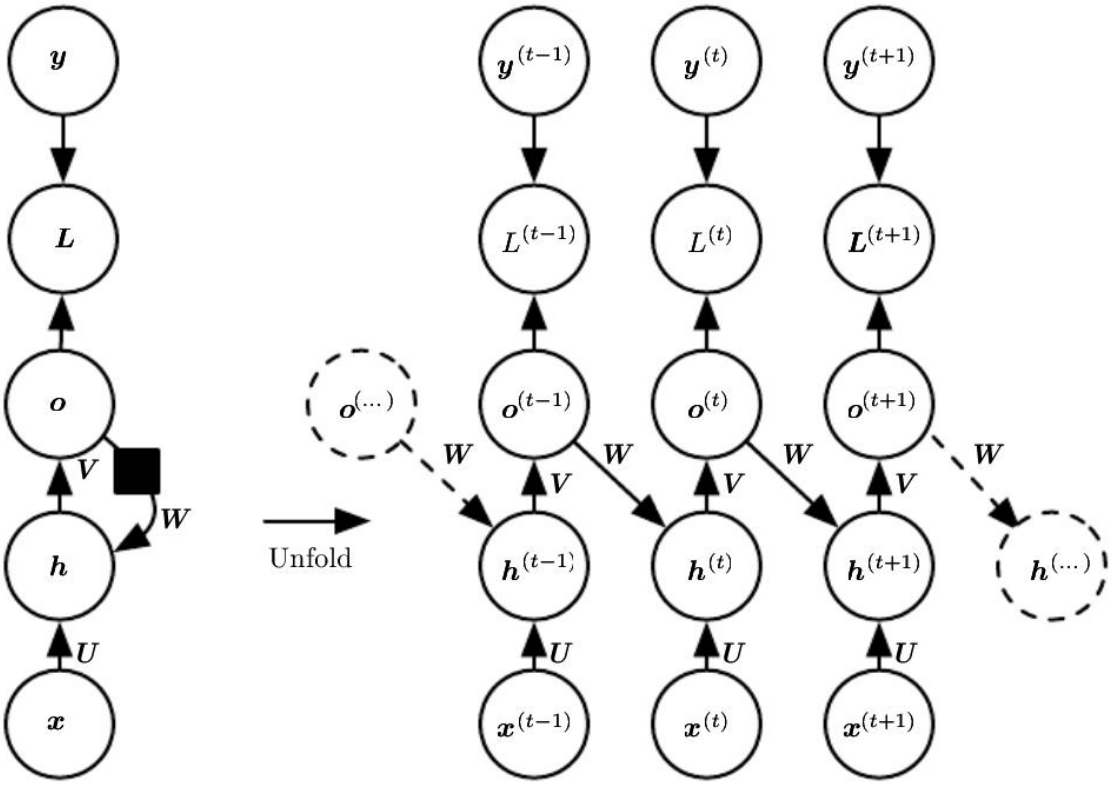


Figure 0-8:An RNN whose only recurrence is the feedback connection from the output to the hidden layer

As it can be seen from the above picture, at each time step t, the input is $\boldsymbol{x}_t$, the hidden layer activations are $\boldsymbol{h}^{(t)}$, the outputs are $\boldsymbol{o}^{(t)}$, the targets are $\boldsymbol{y}^{(t)}$, and the loss is $L^{(t)}$. On the left we have the circuit diagram. On the right, the unfolded computational graph is represented. The RNN in this figure is trained to put a specific output value into $\boldsymbol{o}$, and $\boldsymbol{o}$ is the only information it is allowed to send to the future. There are no direct connections from $\boldsymbol{h}$ going forward. The previous $\boldsymbol{h}$ is connected to the present only indirectly, via the predictions it was used to produce. Unless $\boldsymbol{o}$
is very high-dimensional and rich, it will usually lack important information
from the past. This makes this type of RNN less powerful with respect to the first type discussed before, but it may be easier to train because each time step can be trained in isolation from the others, allowing greater parallelization during training.

- Recurrent networks with recurrent connections between hidden units, that read an entire sequence and then produce a single output, an example of these types of networks can be seen in the following figure:

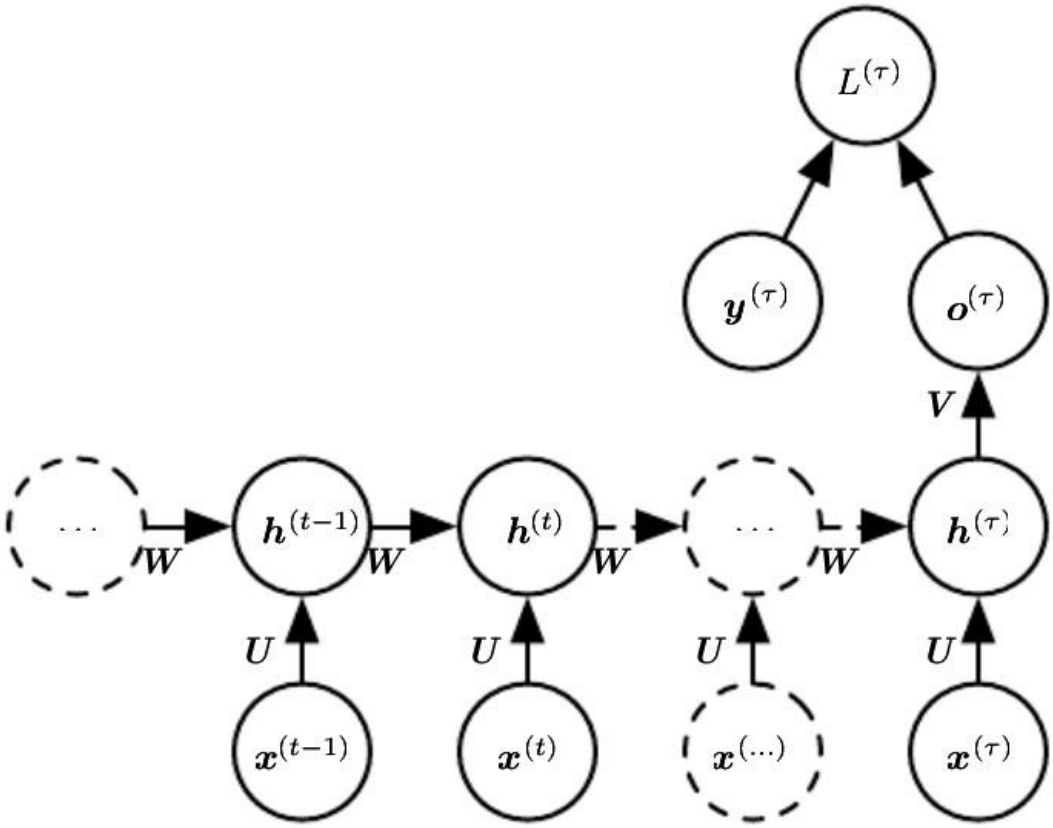


Figure 0-9: Time-unfolded recurrent neural network with a single output at the end of the sequence

In the above picture, it can be observed how such a network can be used to summarize a sequence and produce a fixed-size representation used as input for further processing. There might be a target right at the end (as depicted here), or the gradient on the output $\boldsymbol{o}^{(t)}$ can be obtained by back-propagating from further downstream modules.

Computing the gradient in a recurrent neural network:

To compute the gradient through a RNN we simply apply the back-propagation algorithm to the unrolled computational graph, which is called **BPTT** (**back-propagation through time**). Gradients obtained by back-propagation may then be used with any general-purpose
gradient-based techniques to train an RNN. To gain some intuition for how the BPTT algorithm behaves, we provide an example of how to compute gradients by BPTT for the RNN equations.
The nodes of our computational graph include the parameters $\boldsymbol{U}, \boldsymbol{V}, \boldsymbol{W}, \boldsymbol{b}$ and $\boldsymbol{c}$ as well as the sequence of nodes indexed by t for $\boldsymbol{x}^{(t)}, \boldsymbol{h}^{(t)}, \boldsymbol{o}^{(t)}$ and $L^{(t)}$. For each node $\mathbf{N}$ we need to compute the gradient $\nabla_{\mathbf{N}} L$ recursively, based on the gradient computed at nodes that follow it in the graph. We start the recursion with the nodes immediately preceding the final loss:

$$\frac{\partial L}{\partial L^{(t)}} = 1$$

In this derivation we assume that the outputs $\boldsymbol{o}^{(t)}$ are used as the argument to the softmax function to obtain the vector $\hat{\boldsymbol{y}}$ of probabilities over the output. We also assume that the

loss is the negative log-likelihood of the true target $y^{(t)}$ given the input so far. The gradient $\nabla_{\boldsymbol{o}^{(t)}} L$ on the outputs at time step t, for all i, t, is as follows:

$$(\nabla_{\boldsymbol{o}^{(t)}} L)_i = \frac{\partial L}{\partial o_i^{(t)}} = \frac{\partial L}{\partial L^{(t)}} \frac{\partial L^{(t)}}{\partial o_i^{(t)}} = \hat{y}_i^{(t)} - \mathbf{1}_{i=y^{(t)}}$$

We work our way backward, starting from the end of the sequence. At the final time step τ, $\boldsymbol{h}^{(\tau)}$ only has $\boldsymbol{o}^{(\tau)}$as a descendent, so its gradient is simple:

$$\nabla_{\boldsymbol{h}^{(\tau)}} L = \boldsymbol{V}^T \nabla_{\boldsymbol{o}^{(\tau)}} L$$

We can then iterate backward in time to back-propagate gradients through time, from $t = \tau - 1$ down to $t = 1$, noting that $\boldsymbol{h}^{(t)}$(for $t < \tau$) has as descendants both $\boldsymbol{o}^{(t)}$ and $\boldsymbol{h}^{(t+1)}$. Its gradient is thus given by:

$$\nabla_{\boldsymbol{h}^{(t)}} L = \left(\frac{\partial \boldsymbol{h}^{(t+1)}}{\partial \boldsymbol{h}^{(t)}}\right)^T \left(\nabla_{\boldsymbol{h}^{(t+1)}} L\right) + \left(\frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{h}^{(t)}}\right)^T \left(\nabla_{\boldsymbol{o}^{(t)}} L\right)$$

$$= \boldsymbol{W}^T diag\left(1 - \left(\boldsymbol{h}^{(t+1)}\right)^2\right)\left(\nabla_{\boldsymbol{h}^{(t+1)}} L\right) + \boldsymbol{V}^T\left(\nabla_{\boldsymbol{o}^{(t)}} L\right)$$

where $diag\left(1 - \left(\boldsymbol{h}^{(t+1)}\right)^2\right)$ indicates the diagonal matrix containing the elements $\left(1 - \left(\boldsymbol{h}^{(t+1)}\right)^2\right)$. This is the Jacobian of the hyperbolic tangent associated with the hidden unit i at a time $t + 1$. Once the gradients on the internal nodes of the computational graph are obtained, we can obtain the gradients on the parameter nodes. Because the parameters are shared across many time steps, we must take some care when denoting calculus operations involving these variables. The equations we wish to implement use the back-propagation method, which computes the contribution of a single edge in the computational graph to the gradient. The $\nabla_{\boldsymbol{W}} f$ operator used in calculus, however, considers the contribution of $\boldsymbol{W}$ to the value of f due to all edges in the computational graph. To resolve this ambiguity, we introduce dummy variables $\boldsymbol{W}^{(t)}$ that are defined to be copies of $\boldsymbol{W}$ but with each $\boldsymbol{W}^{(t)}$ used only at time step t. We may then use $\nabla_{\boldsymbol{W}^{(t)}}$ to denote the contribution of the weights at time step t to the gradient.

Using this notation, the gradient on the remaining parameters is given by:

$$\nabla_{\boldsymbol{c}} L = \sum_t \left(\frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{c}}\right)^T \nabla_{\boldsymbol{o}^{(t)}} L = \sum_t \nabla_{\boldsymbol{o}^{(t)}} L$$

$$\nabla_{\boldsymbol{b}} L = \left(\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{b}^{(t)}}\right)^T \nabla_{\boldsymbol{h}^{(t)}} L = \sum_t diag\left(1 - \left(\boldsymbol{h}^{(t)}\right)^2\right) \nabla_{\boldsymbol{h}^{(t)}} L$$

$$\nabla_{\boldsymbol{V}} L = \sum_t \sum_i \left(\frac{\partial L}{\partial o_i^{(t)}}\right) \nabla_{\boldsymbol{V}^{(t)}} o_i^{(t)} = \sum_t \left(\nabla_{\boldsymbol{o}^{(t)}} L\right) \boldsymbol{h}^{(t)^T}$$

$$\nabla_{\boldsymbol{W}} L = \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^{(t)}}\right) \nabla_{\boldsymbol{W}^{(t)}} h_i^{(t)} = \sum_t diag\left(1 - \left(\boldsymbol{h}^{(t)}\right)^2\right) \left(\nabla_{\boldsymbol{h}^{(t)}} L\right) \boldsymbol{h}^{(t-1)^T}$$

$$\nabla_{\boldsymbol{U}} L = \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^{(t)}}\right) \nabla_{\boldsymbol{U}^{(t)}} h_i^{(t)} = \sum_t diag\left(1 - \left(\boldsymbol{h}^{(t)}\right)^2\right) \left(\nabla_{\boldsymbol{h}^{(t)}} L\right) \boldsymbol{x}^{(t)^T}$$

Notice that we do not need to compute the gradient with respect to $\boldsymbol{x}^{(t)}$ for training because it does not have any parameters as ancestors in the computational graph defining the loss.

Even though in the recurrent networks we have developed so far, the losses $L^{(t)}$ were cross-entropies between training targets $\boldsymbol{y}^{(t)}$ and outputs $\boldsymbol{o}^{(t)}$, it is possible to use almost any loss with a recurrent network.

Deep recurrent networks:

The computation in most RNNs can be decomposed into three blocks of parameters and associated transformations:

1. From the input to hidden state
2. From the previous hidden state to the next hidden state
3. From the hidden state to the output

A recurrent neural network can be made deep in many ways:

a) The hidden recurrent state can be broken down into groups organized hierarchically.
b) Deeper computation (e.g., an MLP) can be introduced in the input-to-hidden, hidden-to-hidden, and hidden-to-output parts. This may lengthen the shortest path linking different time steps.
c) The path-lengthening effect can be mitigated by introducing skip connections.

Examples of those RNNs are shown below:

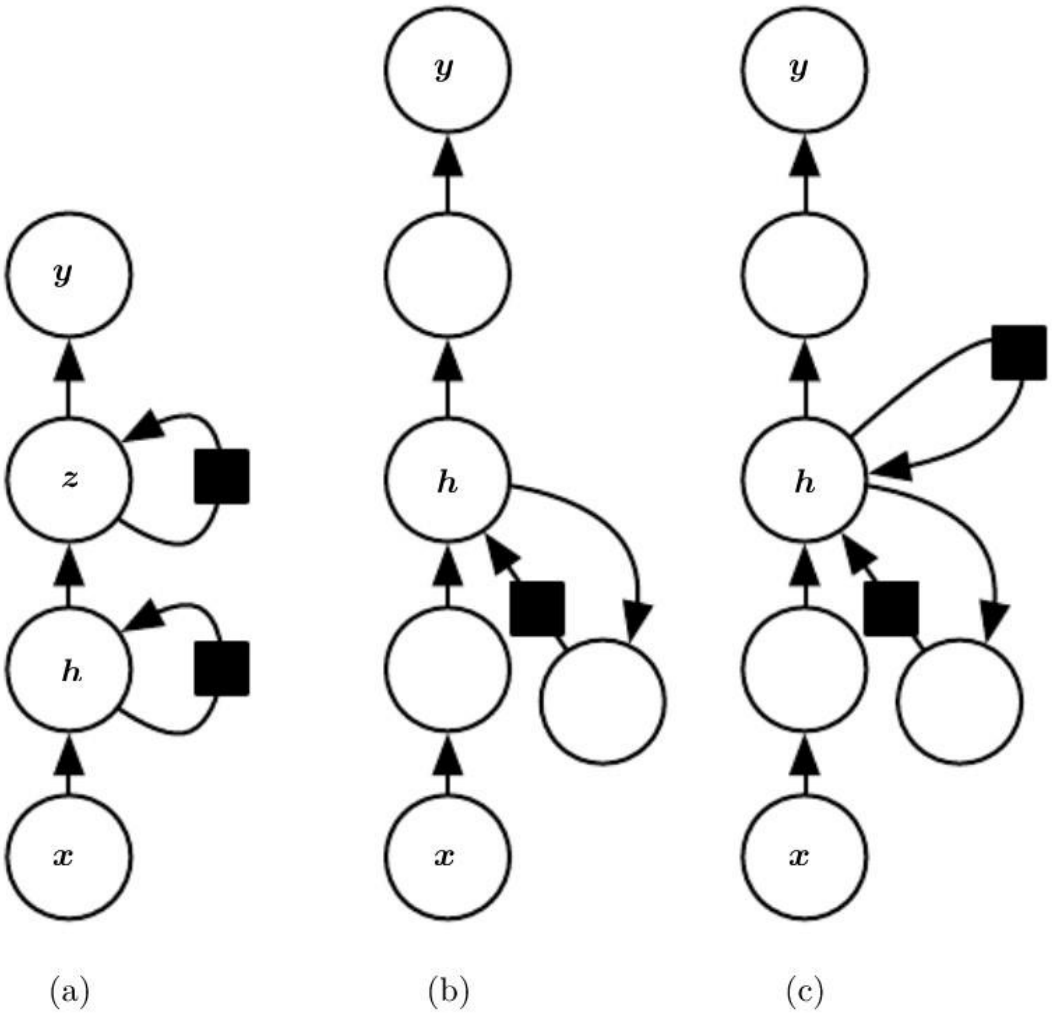


Figure 0-10: Three different examples of deep RNNs

Recursive neural networks:

Recursive neural networks represent yet another generalization of recurrent networks, with a different kind of computational graph, which is structured as a deep tree, rather

than the chain-like structure of RNNs. The typical computational graph for a recursive network is illustrated in the figure below:

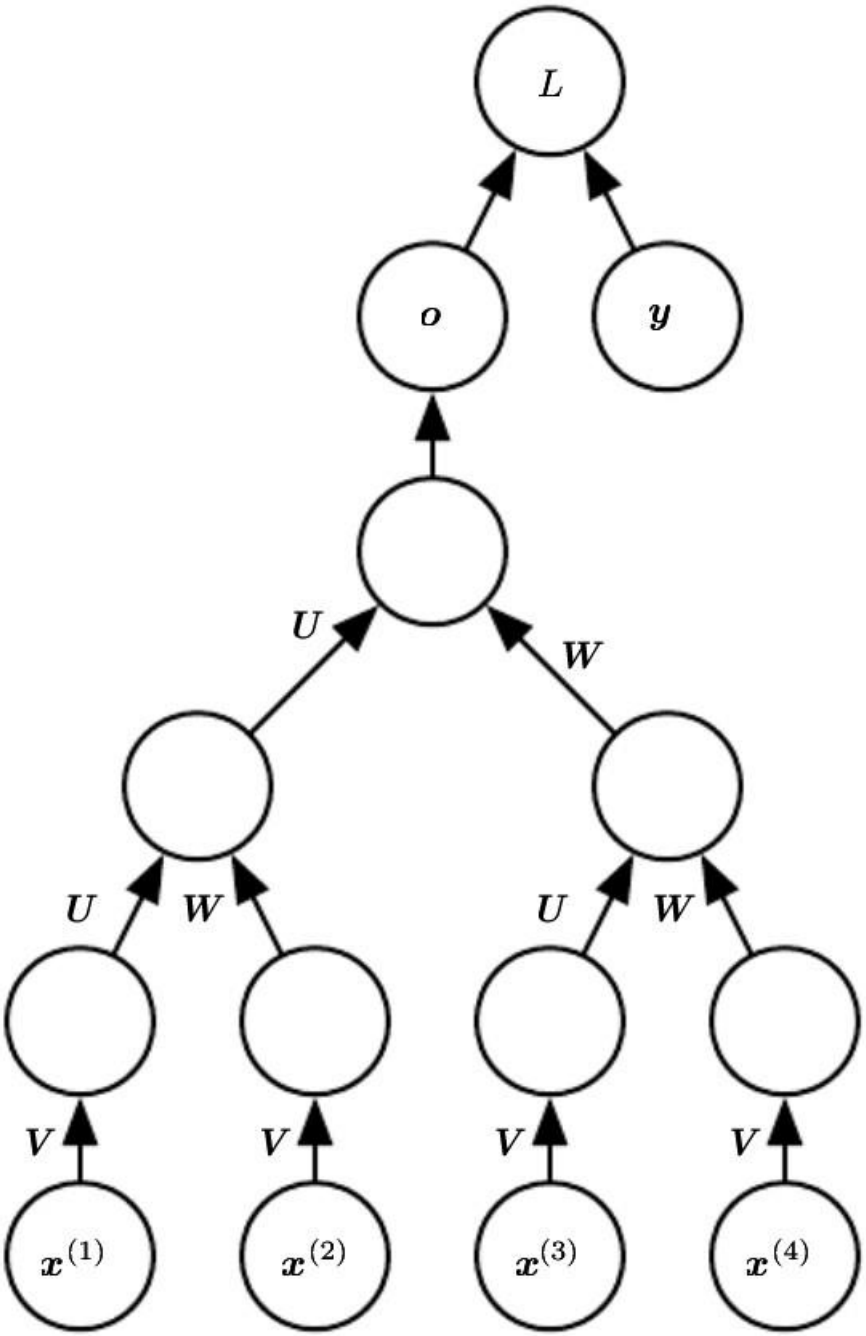


Figure 0-11: Example of a computational graph for a recursive neural network

A variable-size sequence $\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(t)}$ can be mapped to a fixed-size representation (the output $\boldsymbol{o}$), with a fixed set of parameters (the weight matrices $\boldsymbol{U}, \boldsymbol{V}, \boldsymbol{W}$). The above figure illustrates a supervised learning case in which some target $\boldsymbol{y}$ is provided that is associated with the whole sequence.
One clear advantage of recursive nets over recurrent nets is that for a sequence of the same length τ, the depth (measured as the number of compositions of nonlinear operations) can be drastically reduced from τ to
$O(\log \tau)$, which might help deal with long-term dependencies.

The challenge of long-term dependencies:

The mathematical challenge of learning long-term dependencies in recurrent networks relies in the fact that gradients propagated over many stages tend to either vanish (most of the time) or explode (rarely, but with much damage to the optimization). Even if we assume that the parameters are such that the recurrent network is stable (can store memories, with gradients not exploding), the difficulty with long-term dependencies arises from the exponentially smaller weights given to long-term interactions (involving the multiplication of many Jacobians) compared to short-term ones. Recurrent networks involve the composition of the same function multiple times, once per time step. These compositions can result in extremely nonlinear behavior, as illustrated in the figure below:

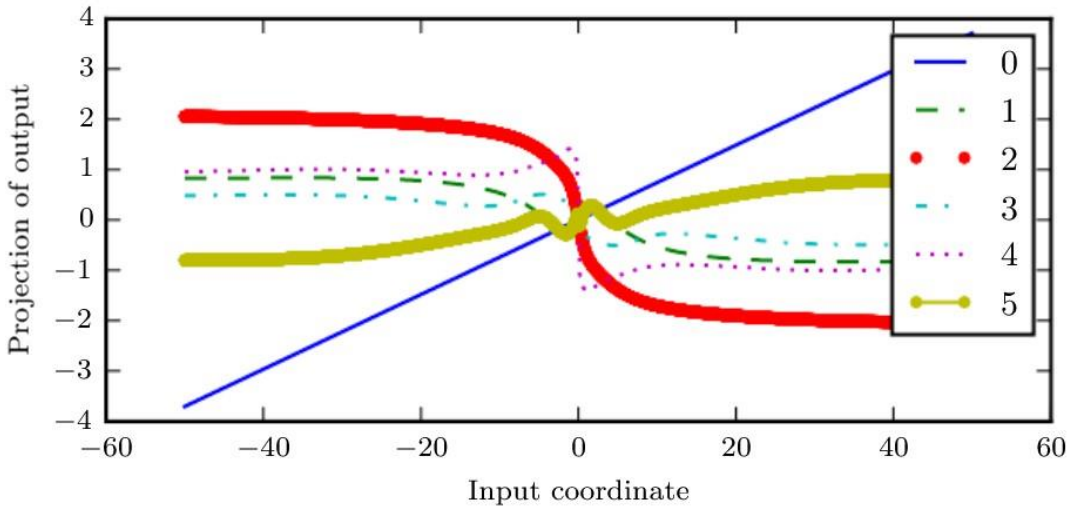


Figure 0-12: Example of repeated nonlinear function composition

In the above plot a linear projection of a 100-dimensional hidden state is projected down to a single dimension, plotted on the y axis. The x axis is the coordinate of the initial state along a random direction in the 100-dimensional space. We can thus view this plot as a linear cross-section of a high-dimensional function. The plots show the function after each time step, or equivalently, after each number of times the transition function has been composed. The labels stand for the composition of linear-tanh layer, and we can clearly see that the more this computation is repeated, the more the result becomes nonlinear.

The function composition employed by recurrent neural networks somewhat resembles matrix multiplication. We can think of the recurrence relation:

$$\boldsymbol{h}^{(t)} = \boldsymbol{W}^T \boldsymbol{h}^{(t-1)}$$

as a very simple recurrent neural network lacking a nonlinear activation function, and lacking inputs $\boldsymbol{x}$. This recurrence relation essentially describes the power method. It may be simplified to:

$$\boldsymbol{h}^{(t)} = (\boldsymbol{W}^t)^T \boldsymbol{h}^{(0)}$$

and if $\boldsymbol{W}$ admits an eigendecomposition of the form:

$$\boldsymbol{W} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^T$$

with orthogonal $\boldsymbol{Q}$, the recurrence may be simplified further to:

$$\boldsymbol{h}^{(t)} = \boldsymbol{Q}^T \boldsymbol{\Lambda}^t \boldsymbol{Q} \boldsymbol{h}^{(0)}$$

This problem is particular to recurrent networks. In the scalar case, imagine multiplying a weight w by itself many times. The product w^t will either vanish or explode depending on the magnitude of w. If we make a nonrecurrent network that has a different weight $w^{(t)}$ at each time step, the situation is different. If the initial state is given by 1,

then the state at time t is given by $\prod_t w^{(t)}$. Suppose that the $w^{(t)}$ values are generated randomly, independently from one another, with zero mean and variance v. The variance of the product is $O(v^n)$. To obtain some desired variance v^* we may choose the individual weights with variance $v = \sqrt[n]{v^*}$. Very deep feedforward networks with carefully chosen scaling can thus avoid the vanishing and exploding gradient problem.
The vanishing and exploding gradient problem for RNNs was independently discovered by separate researchers. One may hope that the problem can be avoided simply by staying in a region of the parameter space where the gradients do not vanish or explode. Unfortunately, to store memories in a way that is robust to small perturbations, the RNN must enter a region of the parameter space where gradients vanish. Specifically, whenever the model can represent long-term dependencies, the gradient of a long-term interaction has exponentially smaller magnitude than the gradient of a short-term interaction. This means not that it is impossible to learn, but that it might take a very long time to learn long-term dependencies, because the signal about these dependencies will tend to be hidden by the smallest fluctuations arising from short-term dependencies. In practice, the experiments have shown that as we increase the span of the dependencies that need to be captured, gradient-based optimization becomes increasingly difficult, with the probability of successful training a traditional RNN via SGD rapidly reaching 0 for sequences of only length 10 or 20.

Leaky units and other strategies for multiple time scales:

One way to deal with long-term dependencies is to design a model that operates at multiple time scales, so that some parts of the model operate at fine-grained time scales and can handle small details, while other parts operate at coarse time scales and transfer information from the distant past to the present more efficiently. Various strategies for building both fine and coarse time scales are possible. These include the addition of skip connections across time, "leaky units" that integrate signals with different time constants, and the removal of some of the connections used to model fine-grained time scales.

Adding skip connections through time:

One way to obtain coarse time scales is to add direct connections from variables in the distant past to variables in the present. In an ordinary recurrent network, a recurrent connection goes from a unit at time t to a unit at time $t + 1$. However, it is possible to construct recurrent networks with longer delays. As we have previously seen, gradients may vanish or explode exponentially with respect to the number of time steps. We can construct recurrent connections with a time delay of d to mitigate this problem. Gradients now diminish exponentially as a function of $\frac{\tau}{d}$ rather than τ. Since there are both delayed and single step connections, gradients may still explode exponentially in τ. This allows the learning algorithm to capture longer dependencies, although not all long-term dependencies may be represented well in this way.

Leaky units and a spectrum of different time scales:

Another way to obtain paths on which the product of derivatives is close to one is to have units with linear self-connections and a weight near one on these connections. When we accumulate a running average $\mu^{(t)}$ of some value $v^{(t)}$ by applying the update $\mu^{(t)} \leftarrow \alpha\mu^{(t-1)} + (1 - \alpha)v^{(t)}$, the α parameter is an example of a linear self-connection from

$\mu^{(t-1)}$ to $\mu^{(t)}$. When α is near one, the running average remembers information about the past for a long time, and when α is near zero, information about the past is rapidly discarded. Hidden units with linear self-connections can behave similarly to such running averages. Such hidden units are called **leaky units**.
Skip connections through d time steps are a way of ensuring that a unit can
always learn to be influenced by a value from d time steps earlier. The use of a linear self-connection with a weight near one is a different way of ensuring that the unit can access values from the past. The linear self-connection approach allows this effect to be adapted more smoothly and flexibly by adjusting the real valued α rather than by adjusting the integer-valued skip length.

Removing Connections:

Another approach to handling long-term dependencies is the idea of organizing the state of the RNN at multiple time scales, with information flowing more easily through long distances at the slower time scales. This idea differs from the skip connections through time discussed earlier because it involves actively removing length-one connections and replacing them with longer connections. Units modified in such a way are forced to operate on a long-time scale, while skip connections add edges (and we can learn to operate on a long-time scale, but also choose to focus on short-term connections).

The Long Short-Term Memory and other gated RNNs:

One of the most effective sequence models (if not the most at all) used in practical applications are called **gated RNNs**. These include the **long short-term memory** and networks based on the **gated recurrent unit**.
Like leaky units, gated RNNs are based on the idea of creating paths through
time that have derivatives that neither vanish nor explode. Leaky units did this with connection weights that were either manually chosen constants or were parameters. Gated RNNs generalize this to connection weights that may change at each time step. Leaky units allow the network to accumulate information (such as evidence for a particular feature or category) over a long duration. Once that information has been used, however, it might be useful for the neural network to forget the old state. For example, if a sequence is made of subsequences and we want a leaky unit to accumulate evidence inside each sub-subsequence, we need a mechanism to forget the old state by setting it to zero. Instead of manually deciding when to clear the state, we want the neural network to learn to decide when to do it. This
is what gated RNNs do.

LSTM

The clever idea of introducing self-loops to produce paths where the gradient can flow for long durations is a core contribution of the initial **long short-term memory** (**LSTM**) model. A crucial addition has been to make the weight on this self-loop conditioned on the context, rather than fixed. By making the weight of this self-loop gated (controlled by another hidden unit), the time scale of integration can be changed dynamically. In this case, we mean that even for an LSTM with fixed parameters, the time scale of integration can change based on the input sequence, because the time constants are output by the model itself.

Instead of a unit that simply applies an element-wise nonlinearity to the affine transformation of inputs and recurrent units, LSTM recurrent networks have "LSTM cells" that have an internal recurrence (a self-loop), in addition to the outer recurrence of the RNN. Each cell has the same inputs and outputs as an ordinary recurrent network, but also has more parameters and a system of gating units that controls the flow of information. The most important component is the state unit $s_i^{(t)}$, which has a linear self-loop like the leaky units described in the previous section. Here, however, the self-loop weight (or the associated time constant) is controlled by a **forget gate** unit $f_i^{(t)}$(for time step t and cell i), which sets this weight to a value between 0 and 1 via a sigmoid unit:

$$f_i^{(t)} = \sigma\left(b_i^f + \sum_j U_{i,j}^f x_j^{(t)} + \sum_j W_{i,j}^f h_j^{(t-1)}\right)$$

where $\boldsymbol{x}^{(t)}$ is the current input vector and $\boldsymbol{h}^{(t)}$ is the current hidden layer vector, containing the outputs of all the LSTM cells, and $\boldsymbol{b}^f, \boldsymbol{U}^f, \boldsymbol{W}^f$ are respectively biases, input weights, and recurrent weights for the forget gates. The LSTM cell internal state is thus updated as follows, but with a conditional self-loop weight $f_i^{(t)}$:

$$s_i^{(t)} = f_i^{(t)} s_i^{(t-1)} + g_i^{(t)} \sigma\left(b_i + \sum_j U_{i,j} x_j^{(t)} + \sum_j W_{i,j} h_j^{(t-1)}\right)$$

where $\boldsymbol{b}, \boldsymbol{U}$ and $\boldsymbol{W}$ respectively denote the biases, input weights, and recurrent weights into the LSTM cell. The **external input gate** unit $g_i^{(t)}$ is computed similarly to the forget gate (with a sigmoid unit to obtain a gating value between 0 and 1), but with its own parameters:

$$g_i^{(t)} = \sigma\left(b_i^g + \sum_j U_{i,j}^g x_j^{(t)} + \sum_j W_{i,j}^g h_j^{(t-1)}\right)$$

The output $h_i^{(t)}$ of the LSTM cell can also be shut off, via the **output gate** $q_i^{(t)}$, which also uses a sigmoid unit for gating:

$$h_i^{(t)} = \tanh\left(s_i^{(t)}\right) q_i^{(t)}$$

$$q_i^{(t)} = \sigma\left(b_i^0 + \sum_j U_{i,j}^0 x_j^{(t)} + \sum_j W_{i,j}^0 h_j^{(t-1)}\right)$$

which has parameters $\boldsymbol{b}^0, \boldsymbol{U}^0, \boldsymbol{W}^0$ for its biases, input weights and recurrent weights, respectively. Among the variants, one can choose to use the cell state $s_i^{(t)}$ as an extra input (with its weight) into the three gates of the i^{th} unit, doing so would require three additional parameters (this approach can be seen in the figure on this page).
The LSTM block diagram is illustrated in the figure below:

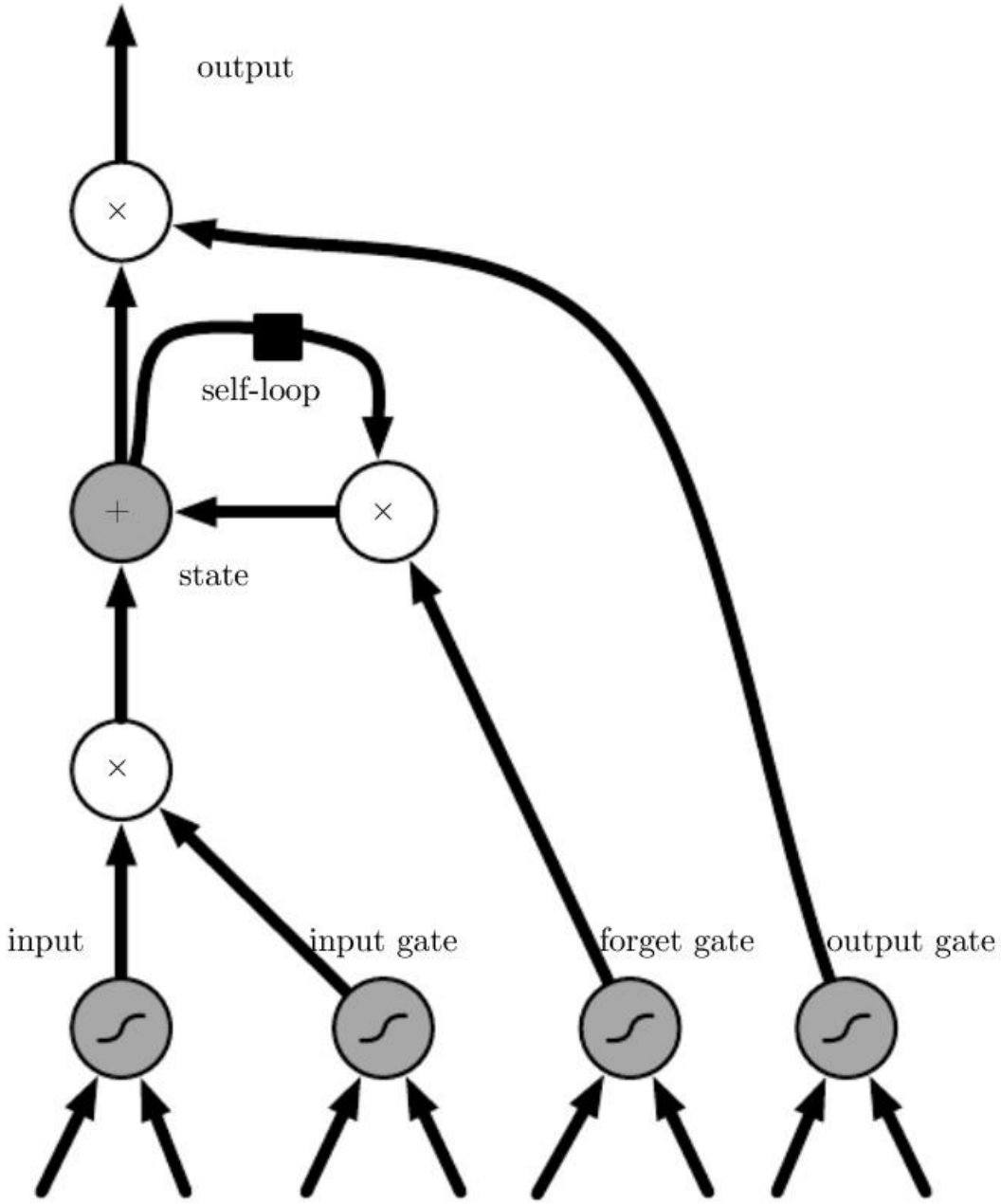


Figure 0-13: Block diagram of the LSTM recurrent network "cell"

As it can be seen from the figure above, cells are connected recurrently to each other, replacing the usual hidden units of ordinary recurrent networks.
An input feature is computed with a regular artificial neuron unit. Its value can be accumulated into the state if the sigmoidal input gate allows it. The state unit has a linear self-loop whose weight is controlled by the forget gate. The output of the cell can be shut off by the output gate. All the gating units have a sigmoid nonlinearity, while the input unit can have any squashing nonlinearity. The state unit can also be used as an extra input to the gating units. The black square indicates a delay of a single time step.
LSTM networks have been shown to learn long-term dependencies more easily than the simple recurrent architectures, first on artificial datasets designed for testing the ability to learn long-term dependencies, then on challenging sequence processing tasks where state-of-the-art performance was obtained. Variants and alternatives to the LSTM are beyond the scope of this work, so they will not be discussed here.

References

For this project the following material has been consulted:

- Data Mining, The Textbook - Aggarwal, Charu C – Springer 2015
- Neural Networks and Deep Learning - Michael Nielsen
- Deep Learning - Ian Goodfellow and Yoshua Bengio and Aaron Courville - An MIT Press book
- Chris Chatfield - The Analysis of Time Series_ An Introduction (1996, Chapman and Hall_CRC)