Beginning Application Development using TensorFlow and Keras

LESSON - 01

Model Architecture

Lesson Objectives

- Prepare data for a deep learning model
- Choose the right model architecture
- Use Keras, a TensorFlow abstraction library
- Make predictions with a trained model

Topic A:

Choosing a Model Architecture for Your Model

- There are many possible architectures
- Researchers frequently create new architectures when working on new problems

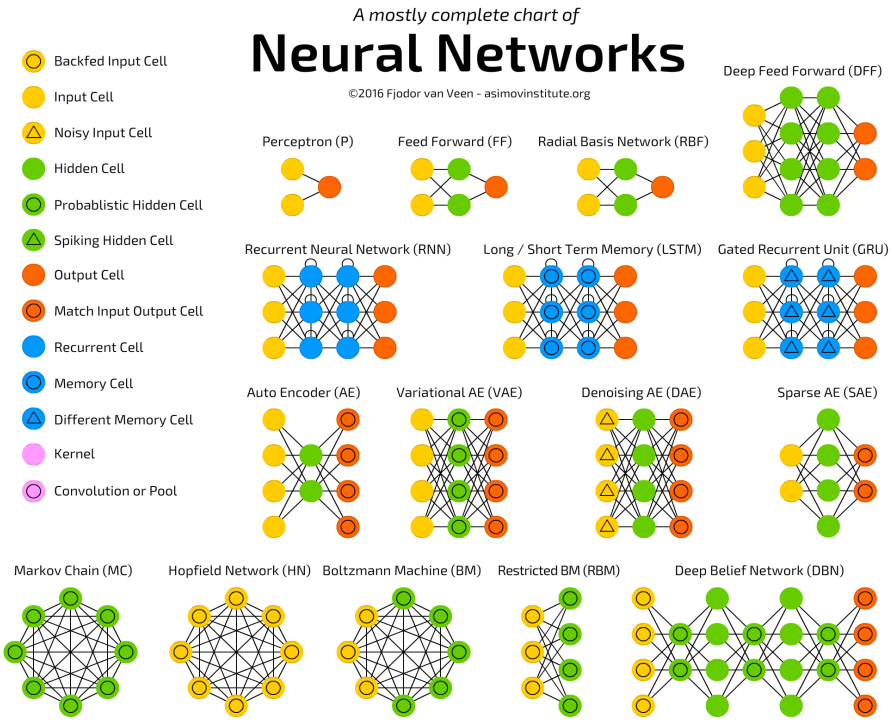


Illustration of different neural network architectures.
Original illustration available at http://www.asimovinstitute.org/neural-network-zoo/

Convolutional Neural Networks

- Convolutional Neural Networks (CNNs) essential insight is to use closely related data as an element of the training process, instead of individual data inputs only.
- For instance, instead of individual pixels a CNN uses a block of pixels as data input.
- The mathematical representation of that process is called "convolution".

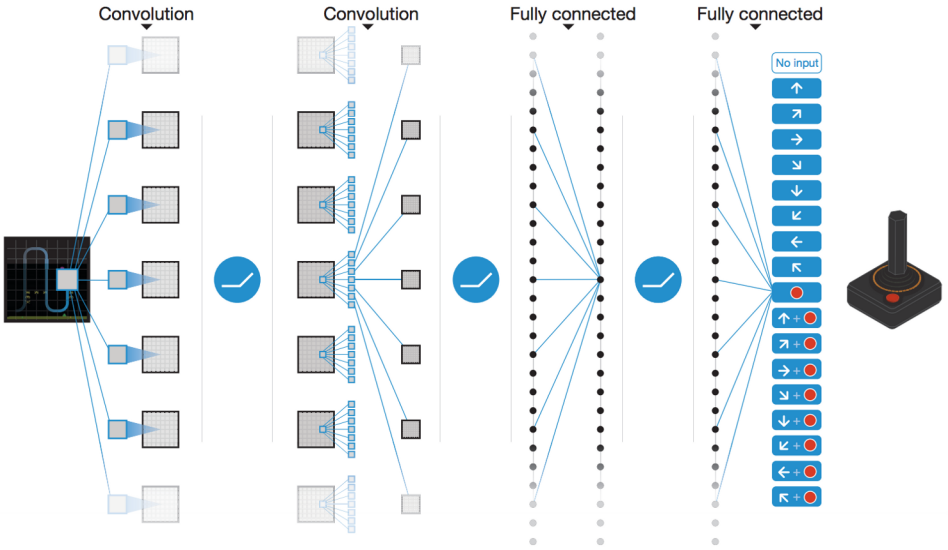


Illustration of the different steps in a convolutional neural network.
Image source: Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". February 2015, Nature.
Available at:
https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf

Recurrent Neural Networks

- Recurrent Neural Networks (RNNs) are designed to work with sequential data.
- This means that, at every epoch, layers can be influenced by the output of previous layers and the output of the memory of previous steps of the sequence.
- Specifically when working with Long-short Term Memory (LSTMs) networks, the previous step can either be close to the data or far apart.

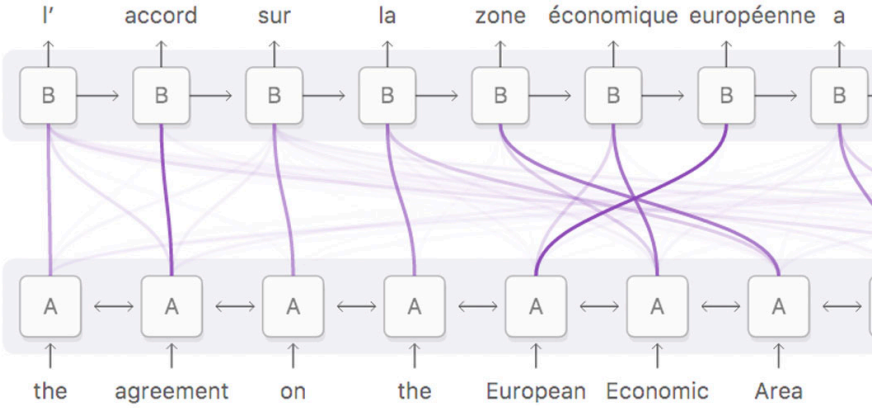


Illustration from distill.pub (https://distill.pub/2016/augmented-rnns/) that show words in English are related to words in French based on where they appear in a sentence. RNNs are very popular in language translation problems.

Generative Adversarial Networks (GANs)

- GANs use concepts from game theory to have many networks "compete" with each other to find a solution for a problem.
- GANs have a network that generates new data ("fake" data) and a network that evaluates the likelihood of the data generated by the first network being real or generated.
- They are very popular for creating new images of objects, people's faces, animals, and so on.

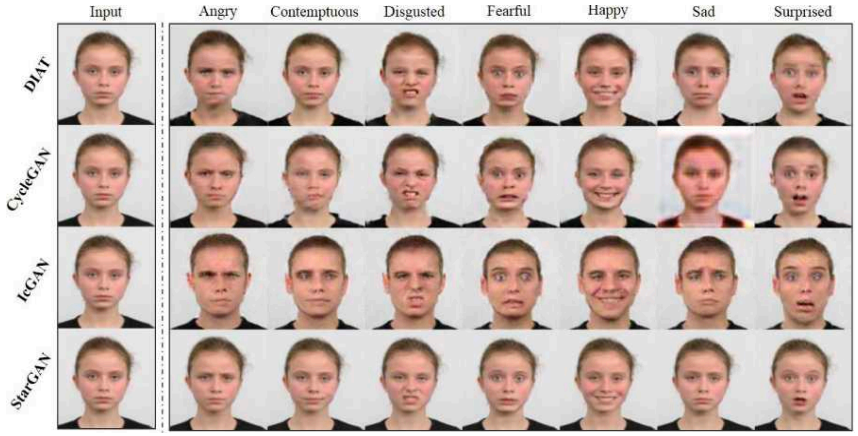


Image that shows the result of different GAN algorithms in changing people's faces based on a given emotion. Source: StarGAN Project. Available at: https://github.com/yunjey/StarGAN

Architecture Summary

Different neural network architectures have shown success in different fields. The networks' architecture is typically related to the problem and the structure of data used to solve it:

Architecture	Data Structure	Successful Applications
Convolutional Neural Networks (CNNs)	Grid-like topological structure (images)	Image recognition and classification
Recurrent Neural Network (RNN) and Long-short Term Memory (LSTM) networks	Sequential data (time-series data)	Speech recognition, text generation, and translation
Generative Adversarial Networks (GANs)	Grid-like topological structure (images)	Image generation

Bitcoin Price Prediction: The Right Architecture for Our Problem

- Given that Bitcoin prices are organized as a sequence of values in time, we will be using a RNN as our architecture.
- Moreover, we will use its Long-short Term Memory (LSTM) variant precisely because it can use patterns from observations early in the series to observations that appear later on to influence the predictions of the system.
- We will be using that architecture to predict future Bitcoin prices.

Data Normalization

- Data normalization is a common practice in machine learning systems.
- In neural networks, normalizations can:
  - Decrease the network's training time
  - Increase its overall performance
- We will be exploring three common normalization techniques:
  - Z-score
  - Point-relative normalization
  - Maximum-minimum normalization

Z-score

- **Z-score:**
  - When data is normally distributed (Gaussian), one can compute the distance between each observation as a standard deviation from its mean:

$$z_i = \frac{x_i - \mu}{\sigma}$$

Point-Relative Normalization

- **Point-relative normalization** *
  - This normalization computes the difference of a given observation in relation to the first observation of the series:

$$n_i = (o_i / o_0) - 1$$

Source: Siraj Raval in his video "How to Predict Stock Prices Easily - Intro to Deep Learning #7" available on YouTube at: https://www.youtube.com/watch?v=ftMq5ps503w

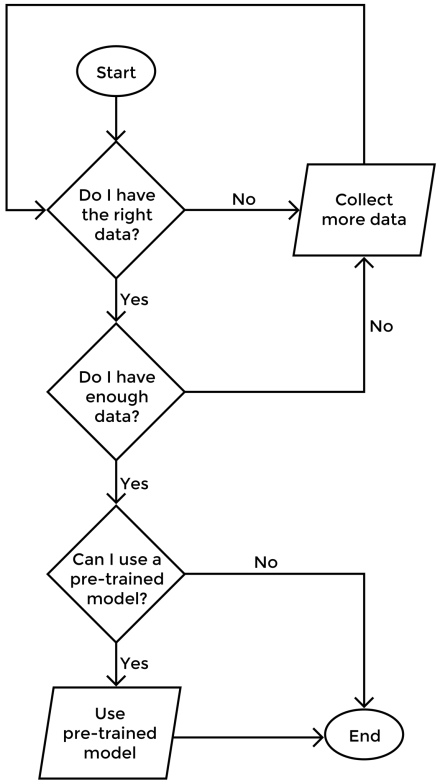
Maximum and Minimum Normalization

- **Maximum and minimum normalization**
  - This normalization computes the distance between a given observation to the maximum and minimum value of the series:

$$n_i = o_i - \min(O) / \max(O) - \min(O)$$

Decision-Tree of Key Reflection Questions

- Follow the decision tree to explore the key reflection questions addressed in this lesson
- The tree starts at the beginning of a project
- And ends with the training of a model



Activity 3 :

Exploring the Bitcoin Dataset and Preparing Data for Model

- We will be using data original retrieved from CoinMarketCap (https://coinmarketcap.com/)
- The dataset has been provided alongside this lesson and will be used throughout the rest of this course

Activity 3 :

Exploring the Bitcoin Dataset and Preparing Data for Model

- Using your terminal, navigate to the directory **lesson_2/activity_3** and execute the following code to start a Jupyter Notebook instance:
  - **$ jupyter notebook**
- Open the URL that appears in a browser.

Activity 3 :

Exploring the Bitcoin Dataset and Preparing Data for Model

- Expected output after executing the command:
  - $ **jupyter notebook**

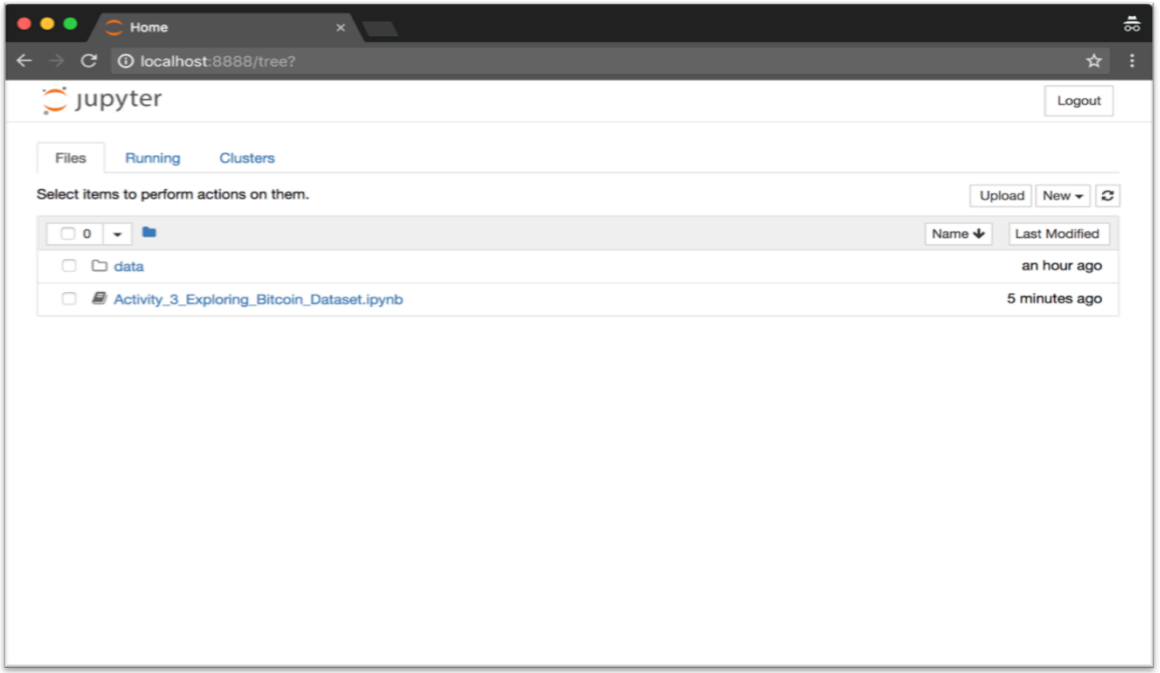
```
(venv)  ✘  ~/Programs/book/lesson_2/activity_3  jupyter notebook
[I 00:25:49.252 NotebookApp] Serving notebooks from local directory: /Users/lcapelo/Programs/book/lesson_2/activity_3
[I 00:25:49.252 NotebookApp] 0 active kernels
[I 00:25:49.252 NotebookApp] The Jupyter Notebook is running at:
[I 00:25:49.252 NotebookApp] http://localhost:8888/?token=2a5f88a7e4eedadd758a342ba1310013610353560df95a32
[I 00:25:49.252 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[C 00:25:49.253 NotebookApp]

    Copy/paste this URL into your browser when you connect for the first time,
    to login with a token:
        http://localhost:8888/?token=2a5f88a7e4eedadd758a342ba1310013610353560df95a32
[I 00:25:49.451 NotebookApp] Accepting one-time-token-authenticated connection from ::1
```

Activity 3 :

Exploring the Bitcoin Dataset and Preparing Data for Model

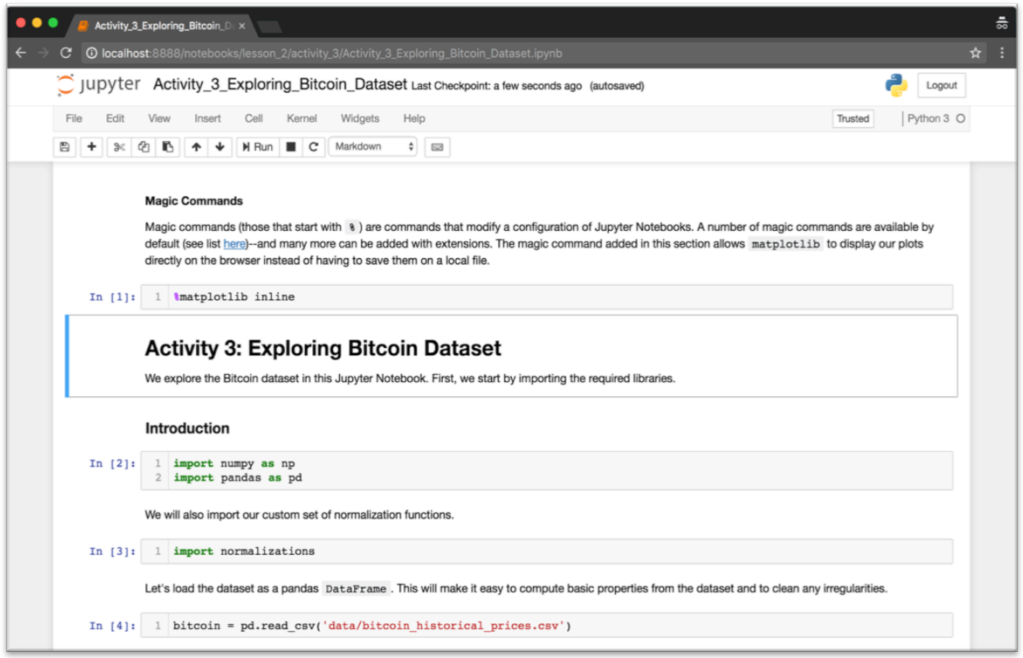
- Now, click on the file **Activity_3_Exploring_Bitcoin_Dataset.ipynb**:



Activity 3 :

Exploring the Bitcoin Dataset and Preparing Data for Model

- Now, click on the file **Activity_3_Exploring_Bitcoin_Dataset.ipynb**:



Activity 3 :

Exploring the Bitcoin Dataset and Preparing Data for Model

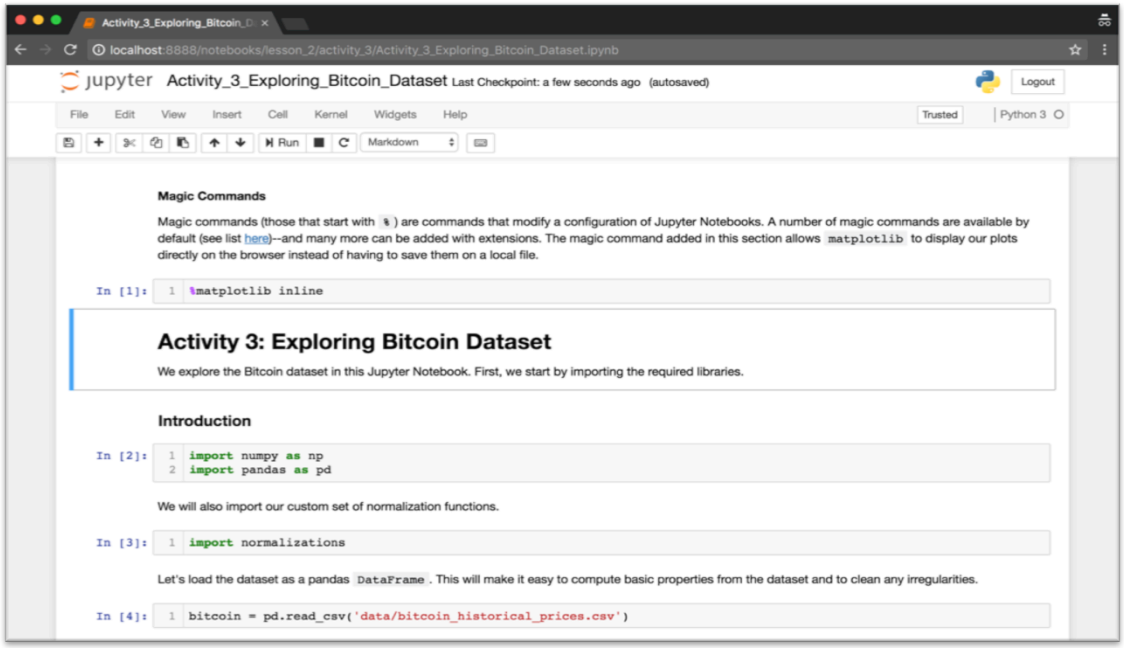
- Let's now load the dataset **data/bitcoin_historical_prices.csv** into a Pandas dataframe.
- The dataset contains the following variables:

Variable	Description
`date`	Date of the observation.
`iso_week`	Week number for a given year.
`open`	Open value for a single Bitcoin coin.
`high`	Highest value achieved during a given day period.
`low`	Lowest value achieved during a given day period.
`close`	Value at the close of the transaction day.
`volume`	What is the total volume of Bitcoin that was exchanged during that day
`market_capitalization`	Market capitalization, which is explained by: Market Cap = Price * Circulating Supply.

Activity 3 :

Exploring the Bitcoin Dataset and Preparing Data for Model

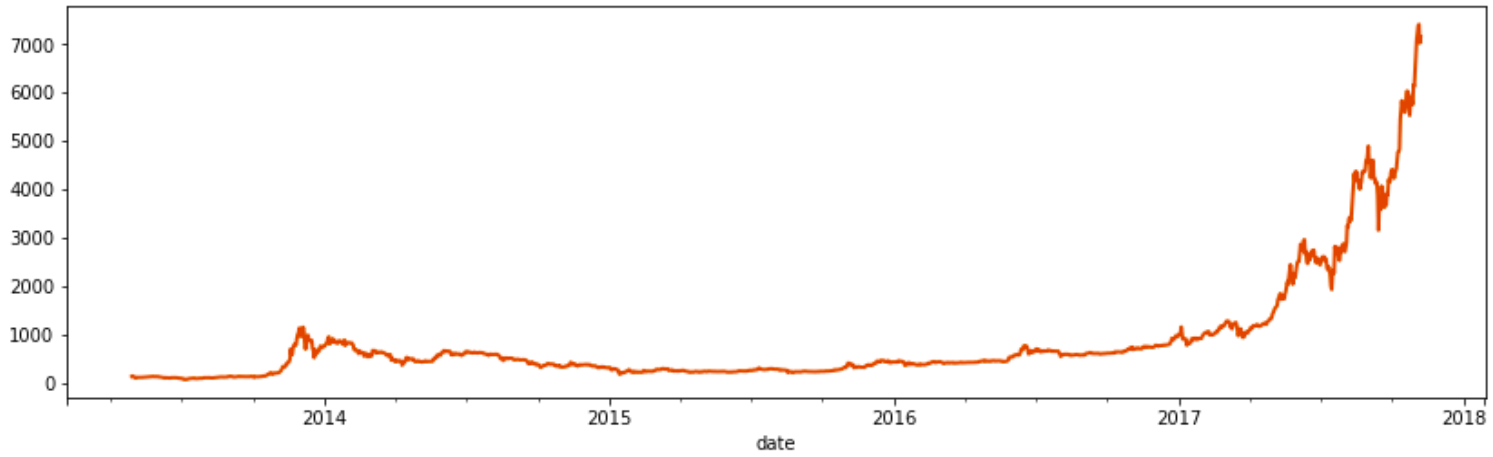
- Execute all cells under the header **Introduction**:



Activity 3 :

Exploring the Bitcoin Dataset and Preparing Data for Model

- After the dataset has been imported into memory, move to the **Exploration** section. You will find a snippet of code that generates a time-series plot for the **close** variable. Can you generate the same plot for the **volume** variable?



Closing price of Bitcoin coins in USD. Notice an early spike by later 2013 ant early 2014. Also, notice how the recent prices have skyrocketed since the beginning of 2017.

Activity 3 :

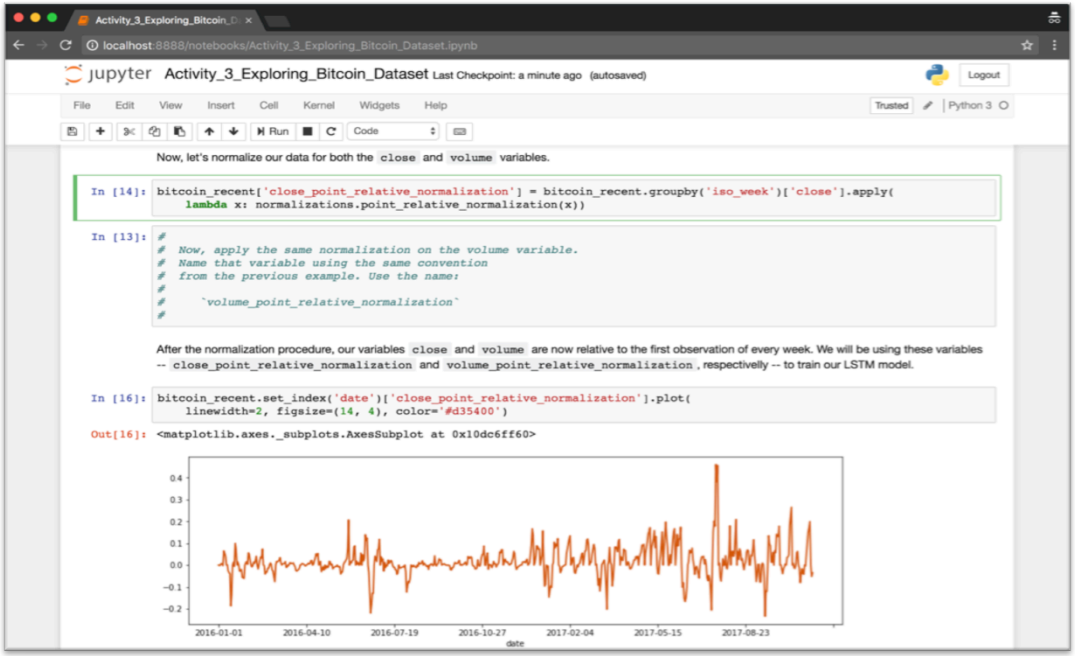
Exploring the Bitcoin Dataset and Preparing Data for Model

- Now, navigate to the **Preparing Dataset for Model** section. You will need to execute two cells:
  - Filter the dates to only include days from 2016 and 2017**:**
    - `bitcoin_rec`
    - `ent = bitcoin[bitcoin['date'] >= '2016-01-01']`
  - Normalize the dataset using the point-relative normalization technique:
    - `bitcoin_recent['close_point_relative_normalization'] =`
    - `bitcoin_recent.groupby('iso_week')['close'].apply(`
    - `lambda x: normalizations.point_relative_normalization(x))`

Activity 3 :

Exploring the Bitcoin Dataset and Preparing Data for Model

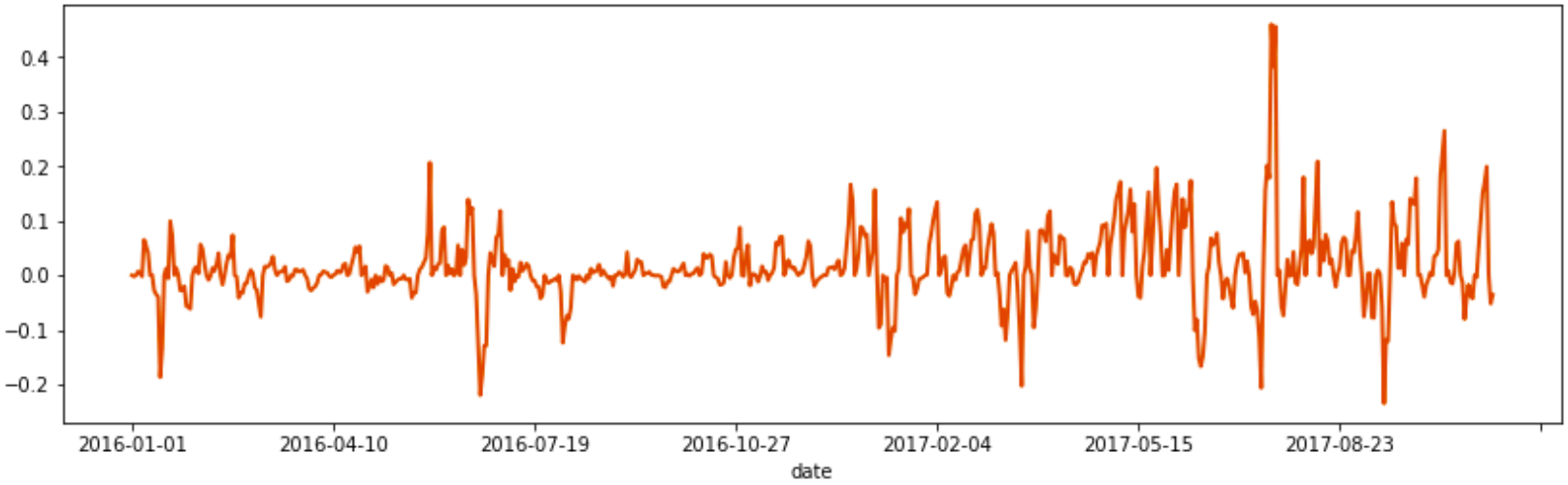
- Output after executing normalization code:



Activity 3 :

Exploring the Bitcoin Dataset and Preparing Data for Model

- After normalizing the close variable, generate the following time-series plot:

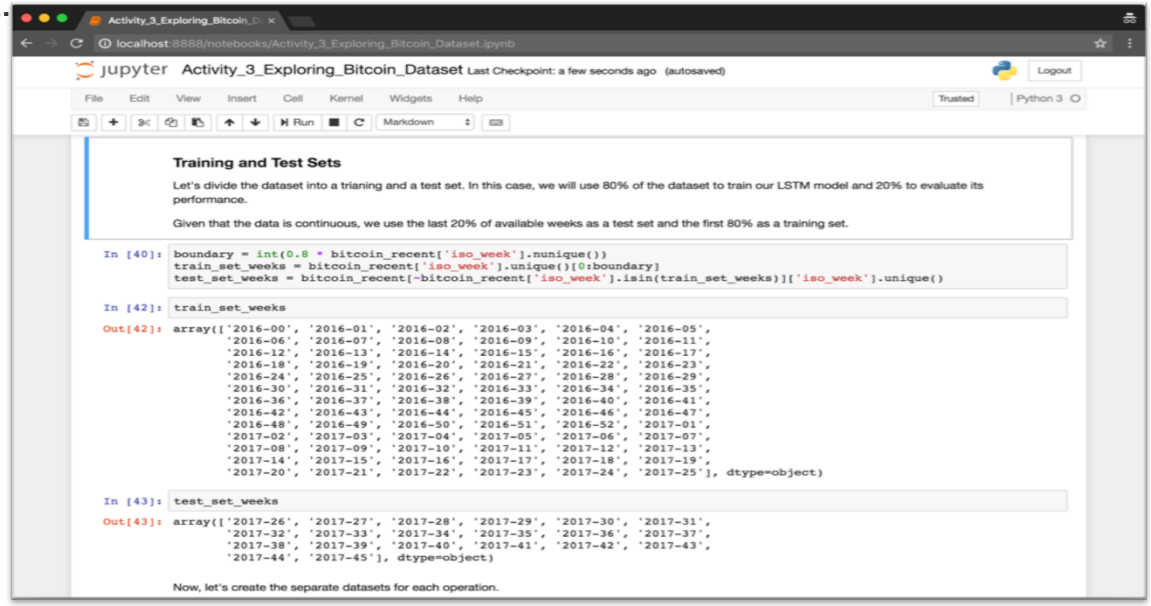


Plot that displays the series from the normalized variable **close_point_relative_normalization**. Notice how this normalization technique highlights the variance of the data in relation to the beginning of a week's period.

Activity 3 :

Exploring the Bitcoin Dataset and Preparing Data for Model

- Separate the dataset between a training and a validation set. In order to do that, chose 80% of the weeks between 2016 and 2017 to be in the training set and 20% to be in the validation set:



Activity 3 :

Exploring the Bitcoin Dataset and Preparing Data for Model

- Finally, store the prepared dataset for later use:

```
test_dataset.to_csv('data/test_dataset.csv', index=False)

train_dataset.to_csv('data/train_dataset.csv', index=False)

bitcoin_recent.to_csv('data/bitcoin_recent.csv', index=False)
```

Topic A: Summary

- During this section we explored:
  - The Bitcoin dataset, learning that the prices of the crypto-currency have changed dramatically over 2017.
  - Filtered and normalized the dataset using the point-relative normalization technique
- The next topic we will explore:
  - How to train a Deep Learning model using Keras
  - Make predictions for a week into the future of Bitcoin prices

Topic B: Using Keras as a TensorFlow Interface

- We are using Keras because it simplifies the TensorFlow interface into general abstractions
- Keras makes it easy to experiment with different architectures and hyperparameters, moving quicker towards a performant solution
- As of TensorFlow 1.4.0 (November 2017), Keras is now officially distributed with TensorFlow as **`tf.keras`**

Model Components

- Keras implements components in a similar to the neural network components explored in *Lesson 1*, *Introduction to Neural Networks and Deep Learning*.
- We will be using an LSTM network for building our prediction model. Notice how that is only one of the many components available on Keras:

Component	Keras Class
High-level abstraction of a complete sequential neural network.	`keras.models.Sequential()`
Dense, fully-connected layer.	`keras.layers.core.Dense()`
Activation function.	`keras.layers.core.Activation()`
LSTM recurrent neural network. This class contains components that are exclusive to this architecture, most of which are abstracted by Keras.	`keras.layers.recurrent.LSTM()`

Designing a Model

- Keras components can be layered together using on of the base classes from the keras.models module.
  - We will be using Sequential()
- We add each class from the previous table, instantiating that class with a series of default and configurable parameters.

```
1   from keras.models import Sequential
2   from keras.layers.recurrent import LSTM
3   from keras.layers.core import Dense, Activation
4
5   model = Sequential()
6
7   model.add(LSTM(
8      units=number_of_periods,
9      input_shape=(period_length, number_of_periods),
10     return_sequences=False))
11
12  model.add(Dense(units=period_length))
13
14  model.add(Activation("linear"))
15
16  model.compile(loss="mse", optimizer="rmsprop")
```

Training a Model

- After a model has been designed (and "compiled") we can now train it using model.fit():

```
1  model.fit(
2    X_train, Y_train,
3    batch_size=32, epochs=epochs)
```

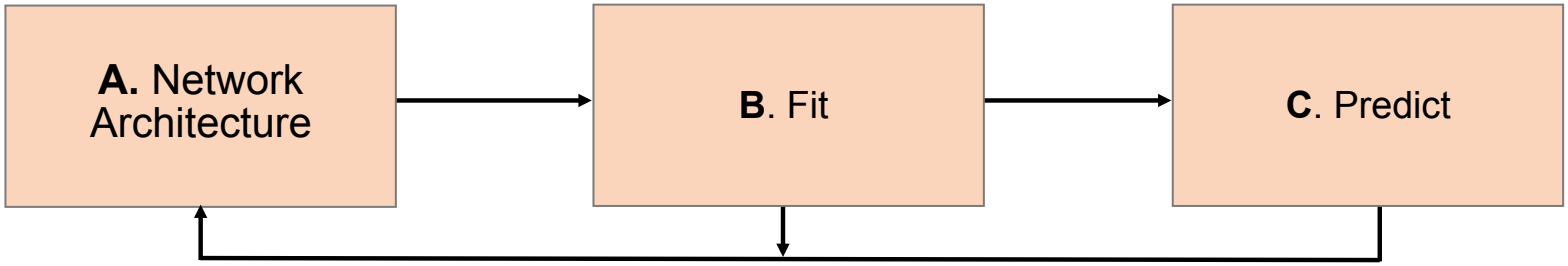
Making Predictions

- After a model has been trained we can now use it for making predictions with model.predict():

```
1   model.predict(x=X_train)
```

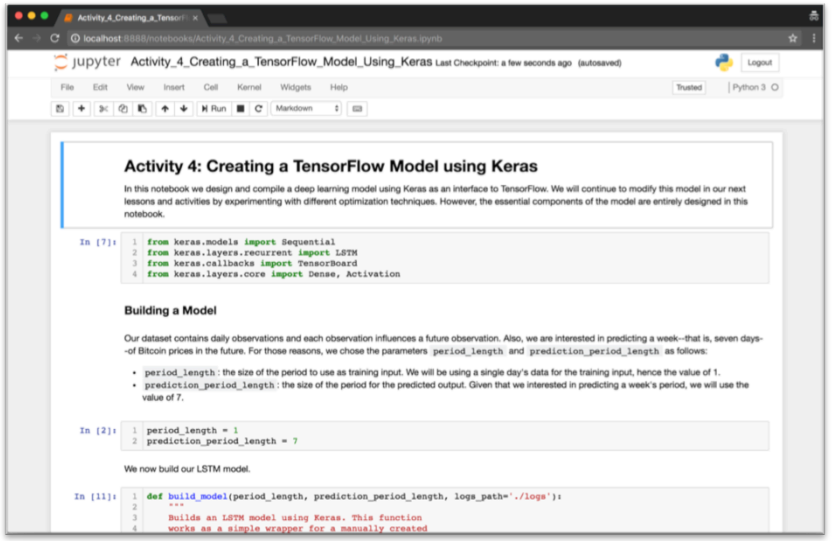
The Keras Paradigm

- The previous three steps outline the Keras paradigm: network architecture, fit (train), and predict. Most deep learning models designed with Keras follow that paradigm:



Activity 4: Creating a TensorFlow Model Using Keras

- In this activity we will train an LSTM model to predict the price of Bitcoin.
- First, navigate to the open instance of the Jupyter Notebook **Activity_4_Creating_a_TensorFlow_Model_Using_Keras.ipynb**. Now, execute all cells under the header **Building a Model:**



Activity 4: Creating a TensorFlow Model Using Keras

- We will use this Jupyter Notebook **`Activity_4_Creating_a_TensorFlow_Model_Using_Keras.ipynb`** to build the same model from *Subtopic B*, *Model Components*, parametrizing the period length of input and of output to allow for experimentation
- Go through the notebook and study the function `build_model()`
- Execute it to build an LSTM model

Activity 4: Creating a TensorFlow Model Using Keras

- After the model is built, save it to disk so we can use it in the future:

```
model.save('bitcoin_lstm_v0.h5')
```

Topic B: Summary

- This topic covered:
  - An introduction to the Keras API (as interface to TensorFlow)
  - Built the first version of a deep learning model for predicting Bitcoin prices and stored it on disk (without training)
- The next topic builds a (nearly complete) deep earning system.

Topic C: From Data Preparation to Modeling

- Neural networks can take a long time to train.
- Three factors are important to consider:
  - The network's architecture
  - How many layers and neurons the network has, and
  - How much data is there to be used in the training process
- Each one of those factors can affect the training time by orders of magnitude.

Bitcoin Dataset

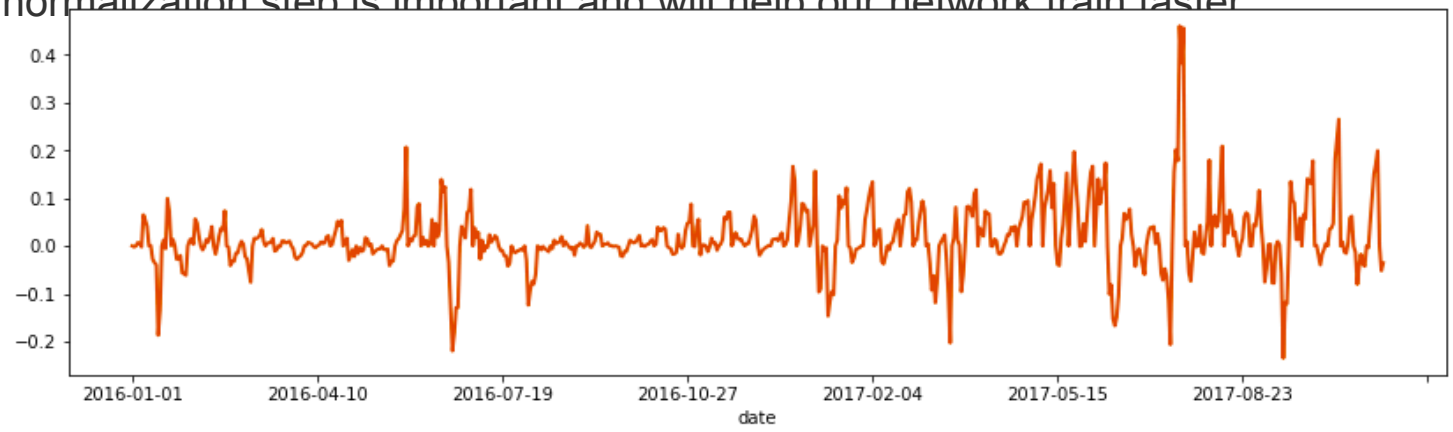
- We will be using our prepared Bitcoin dataset for training our model:

```
1   import pandas as pd
2   train = pd.read_csv('data/train_dataset.csv')
```

	date	iso_week	close	volume	close_point_relative_normalization	volume_point_relative_normalization
0	2016-01-01	2016-00	434.33	36278900.0	0.000000	0.000000
1	2016-01-02	2016-00	433.44	30096600.0	-0.002049	-0.170410
2	2016-01-03	2016-01	430.01	39633800.0	0.000000	0.000000
3	2016-01-04	2016-01	433.09	38477500.0	0.007163	-0.029175
4	2016-01-05	2016-01	431.96	34522600.0	0.004535	-0.128961

Bitcoin Dataset

- The variable **`close_point_relative_normalization`** has been normalized on a weekly basis
- Each observation from the week's period is made relative to the difference from the closing prices on the first day of the period
- This normalization step is important and will help our network train faster



Reshaping Time-Series Data

- Neural networks typically work with vectors and tensors, both mathematical objects that organize data in a number of dimensions.
- Each neural network implemented in Keras will have as input either a vector or a tensor that is organized according to a specification.
- LSTMs take tensors as input with the following three dimensions:
  - **Period length (or Window)**: The period length, that is how many observations are there on a period
  - **Number of periods**: How many periods are available in the dataset
  - **Number of features**: Number of features available in the dataset

Reshaping Time-Series Data

- We will first organize data on a weekly basis (that is, consecutive 7-days period):

```
1   group_size = 7
2   samples = list()
3   for i in range(0, len(data), group_size):
4       sample = list(data[i:i + group_size])
5       if len(sample) == group_size:
6           samples.append(np.array(sample).reshape(group_size, 1).tolist())
7
8   data = np.array(samples)
```

Reshaping Time-Series Data

- Then, we will use the numpy.reshape() method to reshape data into that shape.
- Notice how we separate the last week of the dataset tensor to use as a testing set (that is, `Y_validation`). This is effectively what the model uses to verify how it is performing:

```
1  X_train = data[:-1].reshape(1, 76, 7)
2  Y_validation = data[-1].reshape(1, 7)
```

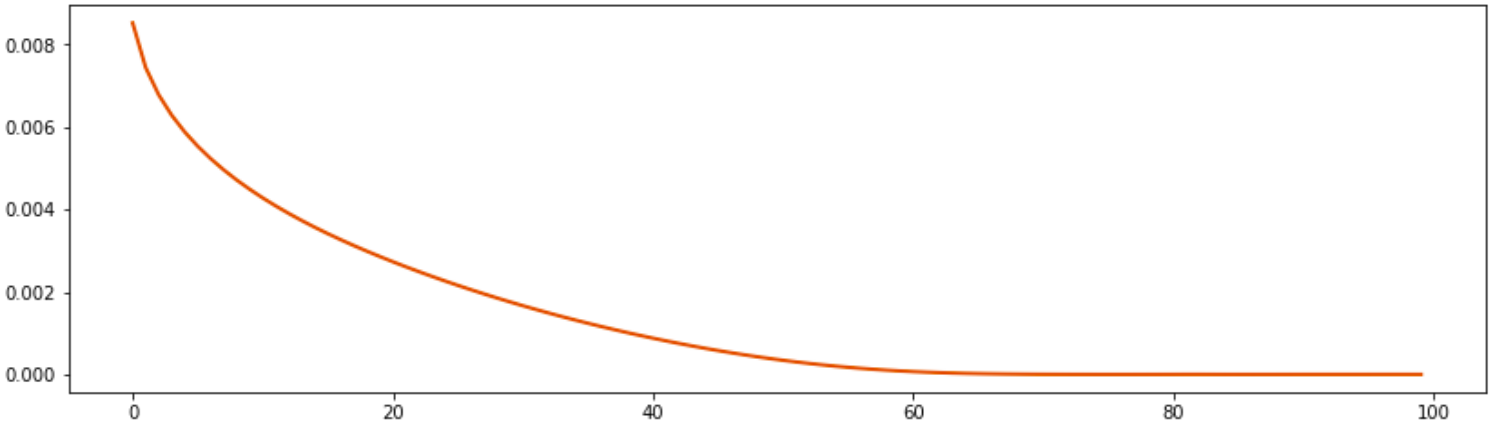
Training a Model

- Let's now train a model using model.`fit()`. We will train this model for 100 epochs:

```
1   model.fit(x=X_train, y=Y_validation, epochs=100)
```

Training a Model

- If we quickly evaluate how the model performs, we notice that its loss function decays very quickly
- We will explore how to evaluate the efficiency of a model in *Lesson 3*, *Model Evaluation and Optimization*:



Graph that shows the results of the loss function evaluated at each epoch. This compares what the model predicted at each epoch then compares with the real data using a technique called mean-squared error. This plot shows those results.

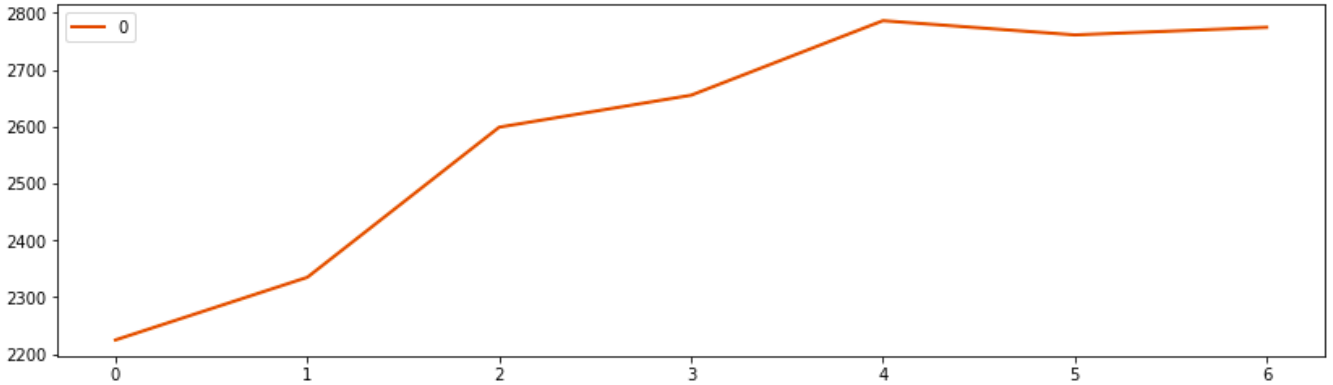
Making Predictions

- Once we have trained our model with `model.fit()` making predictions is trivial:

```
1   model.predict(x=X_train)
```

Overfitting

- Our LSTM network is predicting a price increase of nearly 30%—from about $2,200 to $2,800—during a period of seven days. Is this correct? Can such a simple network predict a week of Bitcoin prices with any precision? We will explore answers to that question in our next lesson:



After de-normalization, our LSTM model has predicted that in late July 2017 the prices of Bitcoin would increase from $2,200 to roughly $2,800, a 30% increase a single week. We will evaluate if that is correct during our next lesson.

Activity 5: Assembling a Deep Learning System

- In this activity, we bring together all the essential pieces for building a basic deep learning system: data, model, and prediction.
- Navigate to the Notebook called **`Activity_5_Assembling_a_Deep_Learning_System.ipynb`** and open it using a Jupyter Notebook instance.
- Execute the cells from the header to load the required components and then navigate to the header **Shaping Data**.

Activity 5: Assembling a Deep Learning System

- Load data into the Notebook using:

```
train = pd.read_csv('data/train_dataset.csv')
```

- Now, use the function create_groups() to create groups with N days (in our case, seven days):

```
create_groups(data=train, group_size=7)
```

Activity 5: Assembling a Deep Learning System

- Now, make sure to both split our data into two sets: training and validation.
- Also, reshape the data to match LSTM's three dimensions (period length, number of periods, and number of features):

  - `X_train = data[:-1].reshape(1, 76, 7)`
  - `Y_validation = data[-1].reshape(1, 7)`

Activity 5: Assembling a Deep Learning System

- Our data is now ready to be used in training. Now we load our previously saved model and train it with a given number of epochs:

```
model = load_model('bitcoin_lstm_v0.h5')
history = model.fit(
    x=X_train, y=Y_validation,
    batch_size=32, epochs=100)
```

- Notice that we store the logs of the model in a variable called **`history`**. We will use that variable to inspect how the model performs.

Activity 5: Assembling a Deep Learning System

- Finally, let's now make a prediction with our trained model. We use the same data used in training (`X_train`):

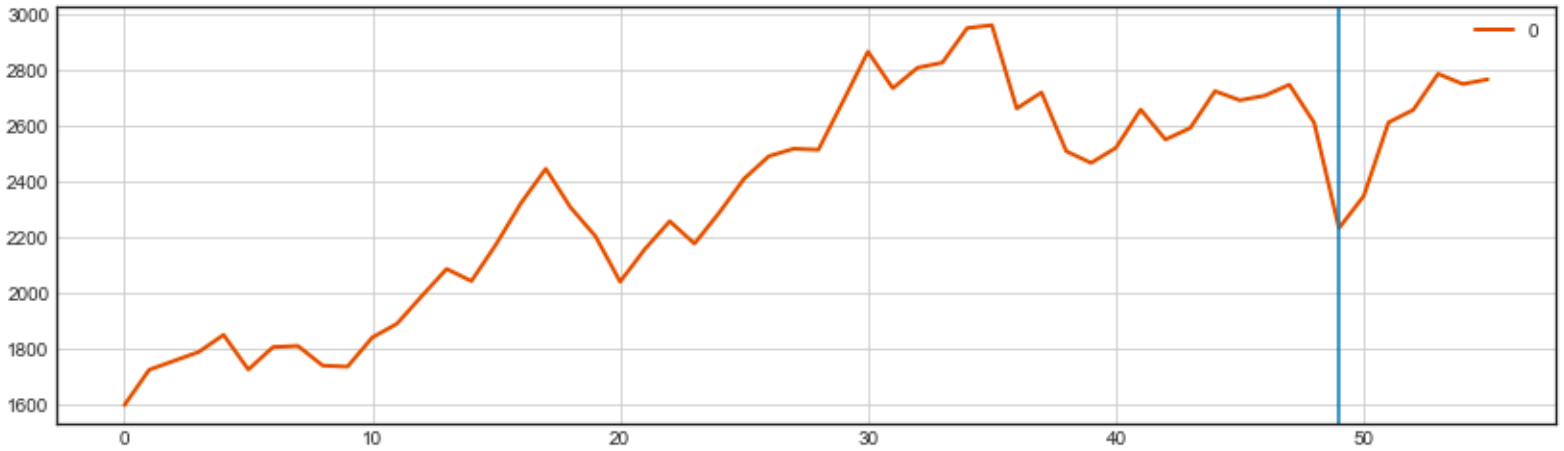
```
predictions = model.predict(x=X_train)[0]
```

- The predictions set returns a normalized series. We can denormalize it with the `denormalize()` function:

```
denormalized_prediction = denormalize(predictions,
last_weeks_value)
```

Activity 5: Assembling a Deep Learning System

- As you can now inspect in the resulting data, our LSTM model predicts a price gain of about $600 dollars during the next seven days—an enormous price increase:



Projection of Bitcoin prices for 7 days in the future using the LSTM model we just built. We combine both time-series in this graph: the real data (before the line) and the predicted data (after the line). The model shows variance similar to the patterns seen before and it suggests a price increase during the following 7-day period.

Activity 5: Assembling a Deep Learning System

- Do you think that the predictions are correct? How can you modify this network to make it more or less robust? Feel free to attempt ideas you may have in your Jupyter Notebook instance.
- After you are done, store the trained model for later use:

```
model.save('bitcoin_lstm_v0_trained.h5')
```

Summary

- We developed a complete deep learning system: from data to prediction

THANK YOU!

Beginning Application Development using TensorFlow and Keras

LESSON - 02

Model Evaluation and Optimization

Lesson Objectives

- Model evaluation
  - Types of problems addressed by neural networks
  - Loss Functions, accuracy, and error rates
  - Using TensorBoard
  - Evaluation metrics and techniques
- Hyperparameter optimization
  - Layers and Nodes
  - Epochs
  - Activation Functions
  - Regularization Strategies

Topic A: Model Evaluation

There are two types of problems in machine learning:

Classification:

Classification problems regard the prediction of the right categories from data, for instance if the temperature is "hot" or "cold"

Regression:

Regression problems are about the prediction of values in a continuous scalar, for instance what is the actual temperature value

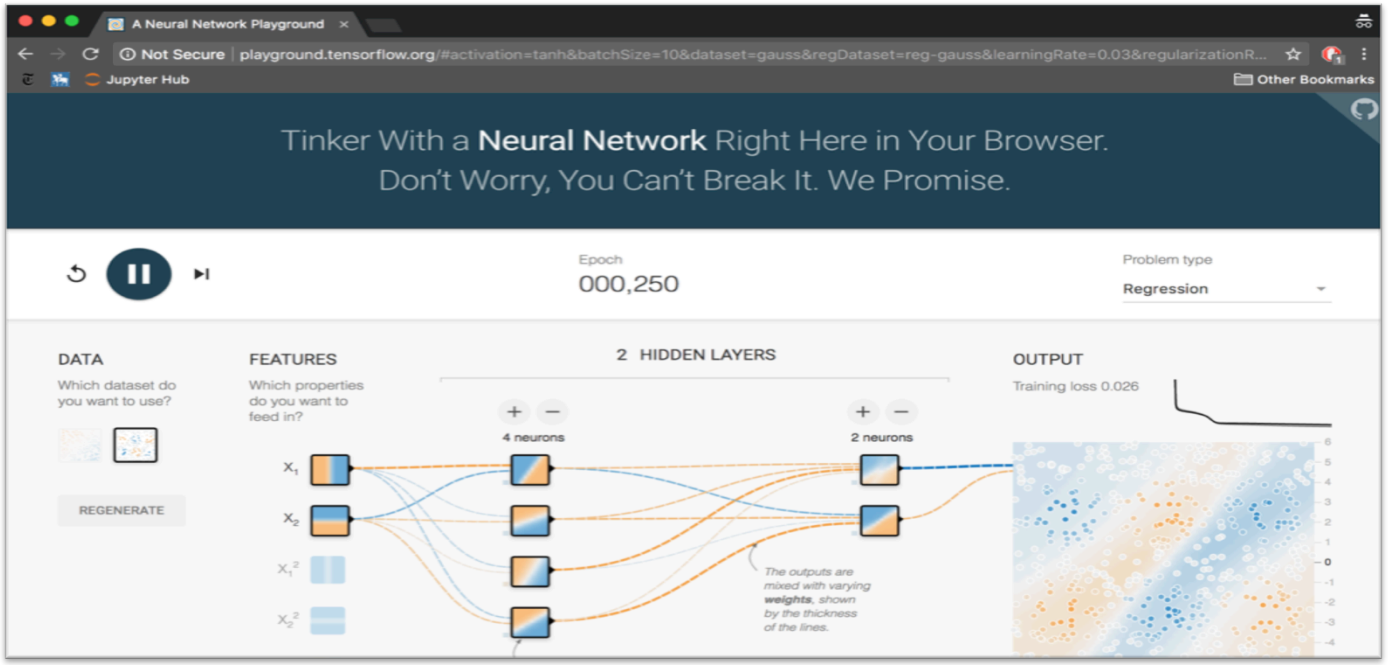
Loss Functions, Accuracy, and Error Rates

- Loss functions will evaluate how "wrong" the network is in making a prediction
- Then, the resulting values are propagated to the network, modifying the weights of the network's connections which ultimately lead to an improved model
- Loss functions are integral parts on how neural networks learn

Exercise 1: Different Loss Functions, Same Architecture

- Follow along this demonstration on your browser by opening the URL: https://goo.gl/NYKhj8
- You will see open the **TensorFlow Playground** application, which has been made available by the TensorFlow team to help us understand how neural networks work.

Exercise 1: Different Loss Functions, Same Architecture



Exercise 1: Different Loss Functions, Same Architecture

- After clicking on the play button, the numbers in the "Training loss" area keep going down as the network continuously trains
- The numbers are very similar in each problem category (Classification or Regression) because the loss functions play the same role both neural networks
- However, the actual loss functions used for each category is different and is chosen depending on the problem type
- Explore the TensorFlow Playground application on your own time—it's a great resource

Implementing Model Evaluation Metrics

```
1 model.evaluate(x=X_test, y=Y_test)
```

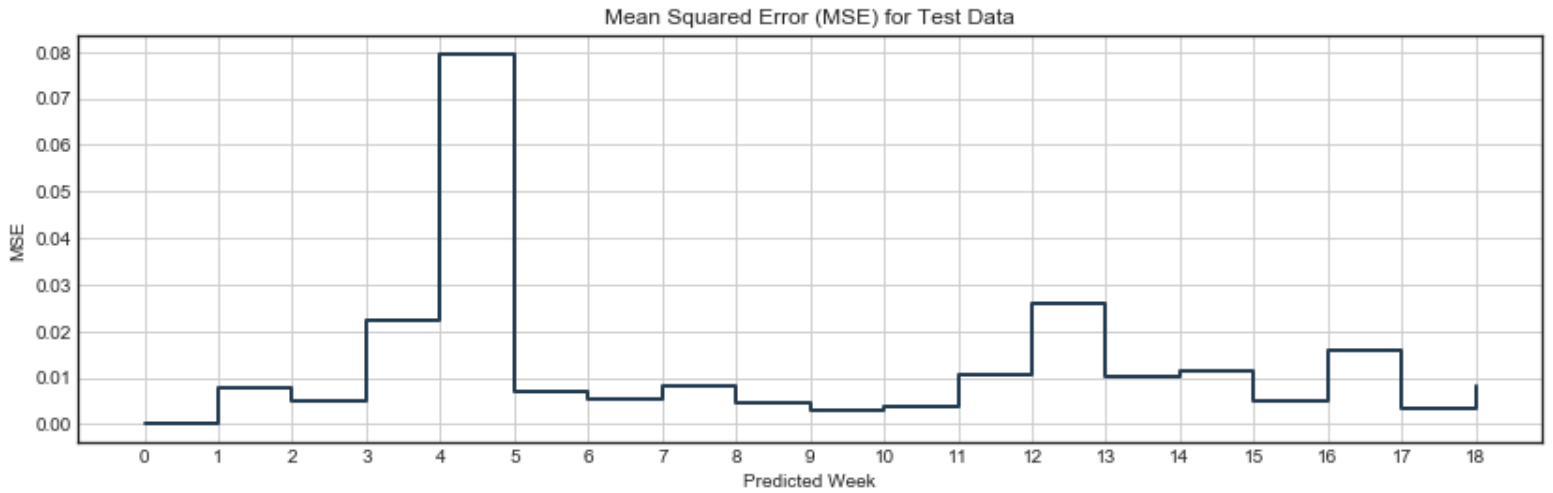
Evaluating the Bitcoin Model

- We do this by using the **`model.evaluate()`** method on each week of the test data:

```
1  combined_set = np.concatenate((train_data, test_data), axis=1)
2  evaluated_weeks = []
3  for i in range(0, validation_data.shape[1]):
4      input_series = combined_set[0:,i:i+77]
5
6      X_test = input_series[0:,:-1].reshape(1, input_series.shape[1] - 1, 7)
7      Y_test = input_series[0:,-1:][0]
8
9      result = B.model.evaluate(x=X_test, y=Y_test, verbose=0)
10     evaluated_weeks.append(result)
```

Evaluating the Bitcoin Model

- We can see that our model has a constantly low MSE, with a few exceptions:



Overfitting

- Our first trained network (**`bitcoin_lstm_v0`**) may be suffering from a phenomenon known as ***overfitting***
- Overfitting is when a model is trained to optimize a validation set, but it does so at the expense of more generalizable patterns from the phenomenon we are interested in predicting
- The main issue with overfitting is that a model learns how to predict the validation set but fails to predict new data
- Further in this lesson we employ hyperparameter optimization techniques that address overfitting

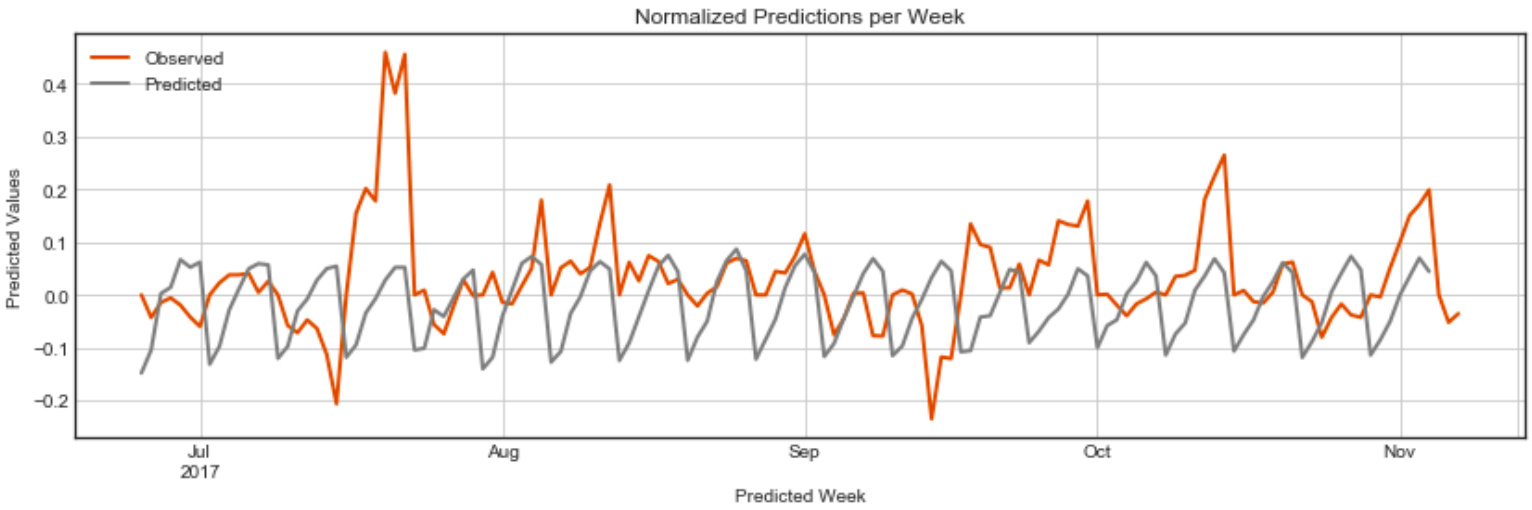
Model Predictions

- We use the same technique to evaluate how well the model predictions perform:

```
1  combined_set = np.concatenate((train_data, test_data), axis=1)
2  predicted_weeks = []
3  for i in range(0, validation_data.shape[1] + 1):
4      input_series = combined_set[0:,i:i+76]
5      predicted_weeks.append(B.predict(input_series))
```

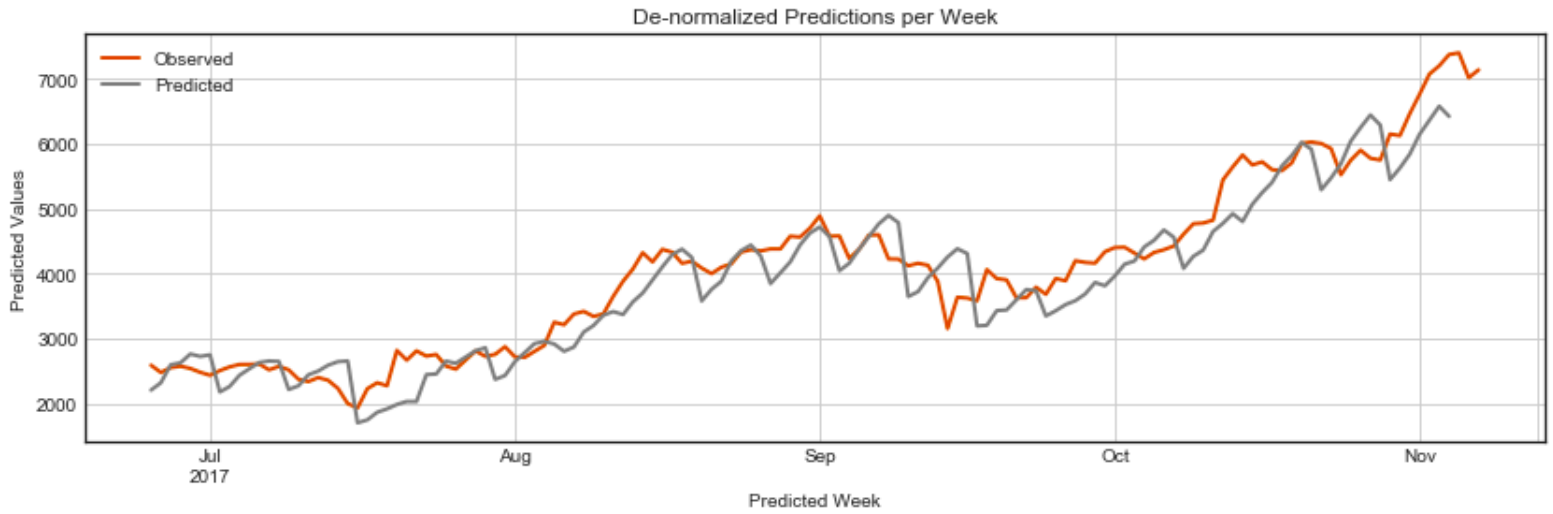
Model Predictions

- The results show that our model is able to match certain patterns from the test data, but not others:



Model Predictions

- The results from our de-normalized are easier to interpret:

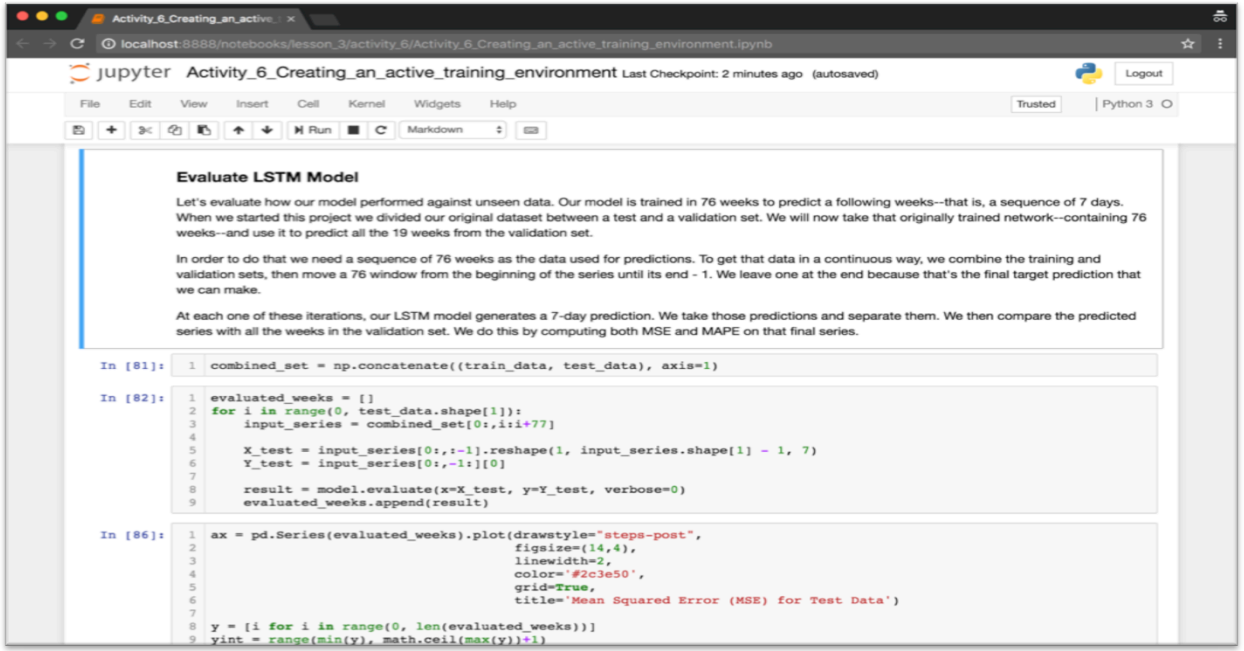


Activity 6: Creating an Active Training Environment

- Using your terminal, navigate to the directory **`lesson_3/activity_6`** and execute the following code to start a Jupyter Notebook instance:
  - **$ jupyter notebook**
- Open the URL provided by the application in your browser and open the Jupyter notebook named **`Activity_6_Creating_an_active_training_environment.ipynb.`**

Activity 6: Creating an Active Training Environment

- You should be able to see the Notebook open in your browser:



Activity 6: Creating an Active Training Environment

- Also using your terminal, start a TensorBoard instance by executing the following command:
  - **$ cd ./lesson_3/activity_6/**
  - **$ tensorboard --logdir=logs/**
- Open the URL that appears on the screen and leave that browser tab open as well.

Activity 6: Creating an Active Training Environment

- Load the train and test datasets in the Jupyter Notebook instance using:
- **$ train = pd.read_csv('data/train_dataset.csv')**
- **$ test = pd.read_csv('data/test_dataset.csv')**
- Also load our previously compiled model using the command:
- **$ model = load_model('bitcoin_lstm_v0.h5')**

Activity 6: Creating an Active Training Environment

- Execute the cells under the header **Evaluate LSTM Model**. The key concept of these cells it to call the **model.evaluate()** method for each of the weeks in the test set. This line is the most important:

  $ result = model.evaluate(x=X_test, y=Y_test, verbose=0)

Activity 6: Creating an Active Training Environment

- Navigate to the section **Interpreting Model Results** and execute the code cells under the sub-header **Make Predictions**.
- Notice that we are calling the **`model.predict()`** method, but a slightly different combination of parameters. Instead of using both **X** and **Y** values, we only use **X**:

```
predicted_weeks = []
for i in range(0, test_data.shape[1]):
  input_series = combined_set[0:,i:i+76]
  predicted_weeks.append(model.predict(input_series))
```

Activity 6: Creating an Active Training Environment

- Navigate to the header **De-normalizing Predictions** and execute all cells under that header.
- In this section we defined the function **`denormalize()`** that performs the complete de-normalization process. Different than other functions, this function takes in a Pandas DataFrame instead of a NumPy array.
- We do so for using dates as an index. This is the most relevant cell block from that header:

```
predicted_close = predicted.groupby('iso_week').apply(
                        lambda x: denormalize(observed, x))
```

Activity 6: Creating an Active Training Environment

- Navigate to the header **De-normalizing Predictions** and load two functions from the **`utilities.py`** script:

```
from scripts.utilities import rmse, mape
```

- The functions from the script are actually really simple:

```
def mape(A, B):
    return np.mean(np.abs((A - B) / A)) * 100


def rmse(A, B):
    return np.sqrt(np.square(np.subtract(A, B)).mean())
```

- Each function is implemented using NumPy's vector-wise operations. They work well in vectors with the same length. They are designed to be applied on a complete set of results.

Activity 6: Creating an Active Training Environment

- Before moving on to the next topic, go back into the Notebook and find the header **Re-train Model with TensorBoard**.
- You may have noticed that we created a helper function called **`train_model().`**
- This function is a wrapper around our model that trains (using **`model.fit()`**) our model, storing its respective results under a new directory. Those results are then used by TensorBoard as a discriminator in order to display statistics for different models.
- Go ahead and modify some of the values for the parameters passed to the **`model.fit()`** function (try epochs, for instance).

Topic A: Summary

- In this topic, we learned about how to evaluate a network using loss function
- We learned that loss functions evaluate the performance of a network and are the starting point for the propagation of adjustments back into layers and nodes
- Most importantly, this lesson concludes with an active training environment
- We now have a system that can train a deep learning model and evaluate its results continuously

Topic B: Hyperparameter Optimization

- In this topic we study different techniques for optimizing our mode. These include:
  - Adding or removing layers and changing the number of nodes
  - Increasing or decreasing the number of training epochs
  - Experimenting with different activation functions
  - Using different regularization strategies

Layers and Nodes - Adding More Layers

- Adding more layers can help the model make higher-level representations that build on low-level ones
- However, many layers may cause overfitting
- And also may significantly increase training and prediction time
- Start with one layer, then increase as needed

Adding More Nodes

- The number of neurons that your layer requires is related to both the how the input and the output data is structured
- If input data has four variables, then four neurons are used
- If the output data has two categories, then two output neurons are used
- Multiples of the input and output data can also be used
- However, starting with a number of neurons that reflect the shape of the input and output is common practice

Layers and Nodes - Implementation

- In LSTM models, LSTM layers work in sequences.
- In our case, the new LSTM layer has the same number of neurons as the original layer, so we don't have to configure that parameter:

```
1  period_length = 7
2  number_of_periods = 76
3  batch_size = 1
4
5  model = Sequential()
6  model.add(LSTM(
7      units=period_length,
8      batch_input_shape=(batch_size, number_of_periods, period_length),
9      input_shape=(number_of_periods, period_length),
10     return_sequences=True, stateful=False))
```

Layers and Nodes - Implementation

- We will bump our model version to **`bitcoin_lstm_v1`** after this modification:

```
11
12  model.add(LSTM(
13      units=period_length,
14      batch_input_shape=(batch_size, number_of_periods, period_length),
15      input_shape=(number_of_periods, period_length),
16      return_sequences=False, stateful=False))
17
18  model.add(Dense(units=period_length))
19  model.add(Activation("linear"))
20
21  model.compile(loss="mse", optimizer="rmsprop")
```

Epochs

- Epochs are the number of cycles the model uses for training
- Increasing the number of epochs has an impact on training time, but may also cause overfitting
- TensorBoard is really useful when exploring what may be a good number of epochs
- When the loss function starts to plateau, that may be a good place to stop training

Epochs - Implementation

- In order to have the model train for more epochs, one only has to change the epochs parameter in **`model.fit()`.**
- This bumps our model to **`bitcoin_lstm_v2`:**

```
number_of_epochs = 10**3
model.fit(x=X, y=Y, batch_size=1,
          epochs=number_of_epochs,
          verbose=0,
          callbacks=[tensorboard])
```

Activation Functions

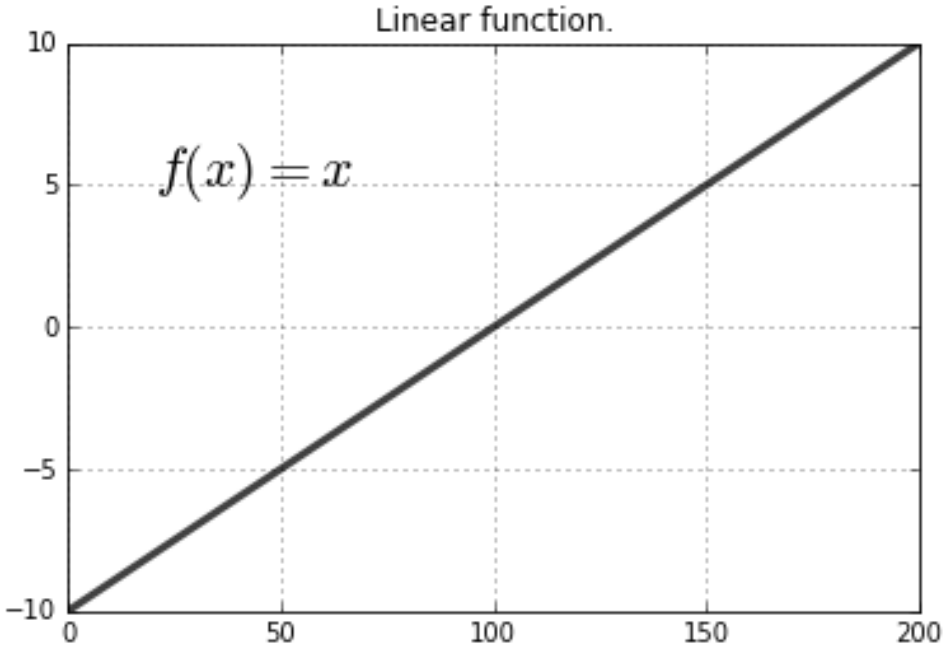
- Activation functions evaluate how much to "activate" individual neurons
- They determine the value that each neuron will pass to the next element of the network, using both the input from the previous layer and the results from the loss function—or if a neuron should pass any values at all

Activation Functions

- Three are relevant for this course:
  - Linear
  - **Hyperbolic Tangent** (**Tanh**)
  - **Rectified Linear Unit** (**ReLU**)

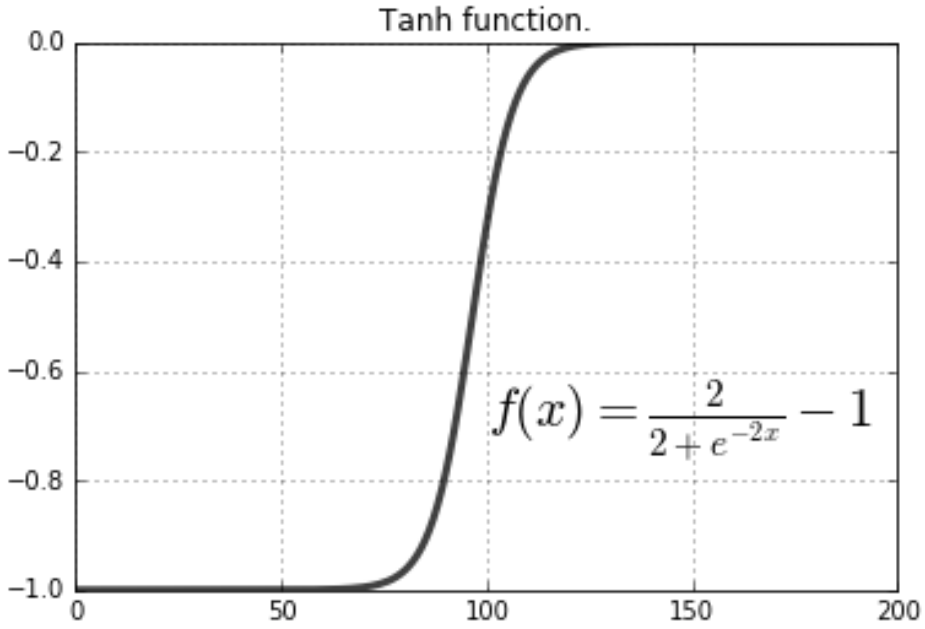
Linear

- Linear functions only activate a neuron based on a constant value
- They essentially don't "activate" neurons as training takes place



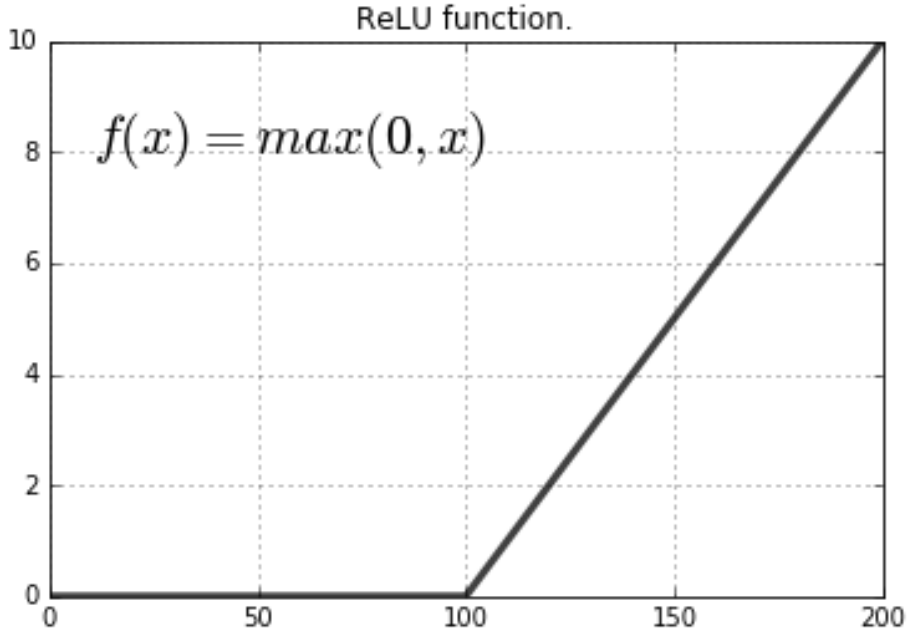
Hyperbolic Tangent (Tanh)

- Non-linear functions that activate nodes as training takes place
- May suffer from the vanishing gradients problem



Rectified Linear Unit (ReLU)

- ReLUs have non-linear properties, working similar to *Tanhs*
- ReLU functions are often recommended as great starting points before trying other functions



Activation Functions - Implementation

- Simply add an **`Activation()`** step to your neural network.
- This bumps our model to **`bitcoin_lstm_v3`**:

```
8  model.add(LSTM(
9      units=period_length,
10      batch_input_shape=(batch_size, number_of_periods, period_length),
11      input_shape=(number_of_periods, period_length),
12      return_sequences=False, stateful=False))
13
14  model.add(Dense(units=period_length))
15  model.add(Activation("tanh"))
16
17  model.compile(loss="mse", optimizer="rmsprop")
```

Dropout

- One randomly takes a way a proportion of nodes from layers
- The other nodes now have to adapt and learn the patterns that were being learned by other nodes
- Very useful and efficient against overfitting

Regularization Strategies – Implementation

- One can add a regularization step after each layer
- In this example, we add a Dropout(0.2) step after each LSTM layer
- This bumps our model to **`bitcoin_lstm_v4`**

```
2  model.add(LSTM(
3      units=period_length,
4      batch_input_shape=(batch_size, number_of_periods, period_length),
5      input_shape=(number_of_periods, period_length),
6     return_sequences=True, stateful=False))
7
8  model.add(Dropout(0.2))
9
10  model.add(LSTM(
11      units=period_length,
12      batch_input_shape=(batch_size, number_of_periods, period_length),
13      input_shape=(number_of_periods, period_length),
14      return_sequences=False, stateful=False))
15
16  model.add(Dropout(0.2))
```

Optimization Results

Our first model **bitcoin_lstm_v0** performed the best in nearly all defined metrics. That model effectively only contained a single LSTM layer:

Model	MSE (last epoch)	RMSE (whole series)	MAPE (whole series)	Training Time
bitcoin_lstm_v0	-	399.6	8.4%	-
bitcoin_lstm_v1	7.15*10-6	419.3	8.8%	49.3 s
bitcoin_lstm_v2	3.55*10-6	425.4	9.0%	1min 13s
bitcoin_lstm_v3	2.8*10-4	423.9	8.8%	1min 19s
bitcoin_lstm_v4	4.8*10-7	442.4	9.4%	1min 20s

Activity 7: Optimizing a Deep Learning Model

- In this activity, we implement different optimization strategies to the model created in *Lesson 2*, *Model Architecture* (**`bitcoin_lstm_v0`**).
- That model achieves a MAPE performance on the complete de-normalization test set of 8.4%. We will try to reduce that gap.

Activity 7: Optimizing a Deep Learning Model

- Using your terminal, start a TensorBoard instance by executing the following command:

```
$ cd ./lesson_3/activity_7/
$ tensorboard --logdir=logs/
```

- Open the URL that appears on the screen and leave that browser tab open as well. Also, start a jupyter notebook instance with:

```
$ jupyter notebook
```

Activity 7: Optimizing a Deep Learning Model

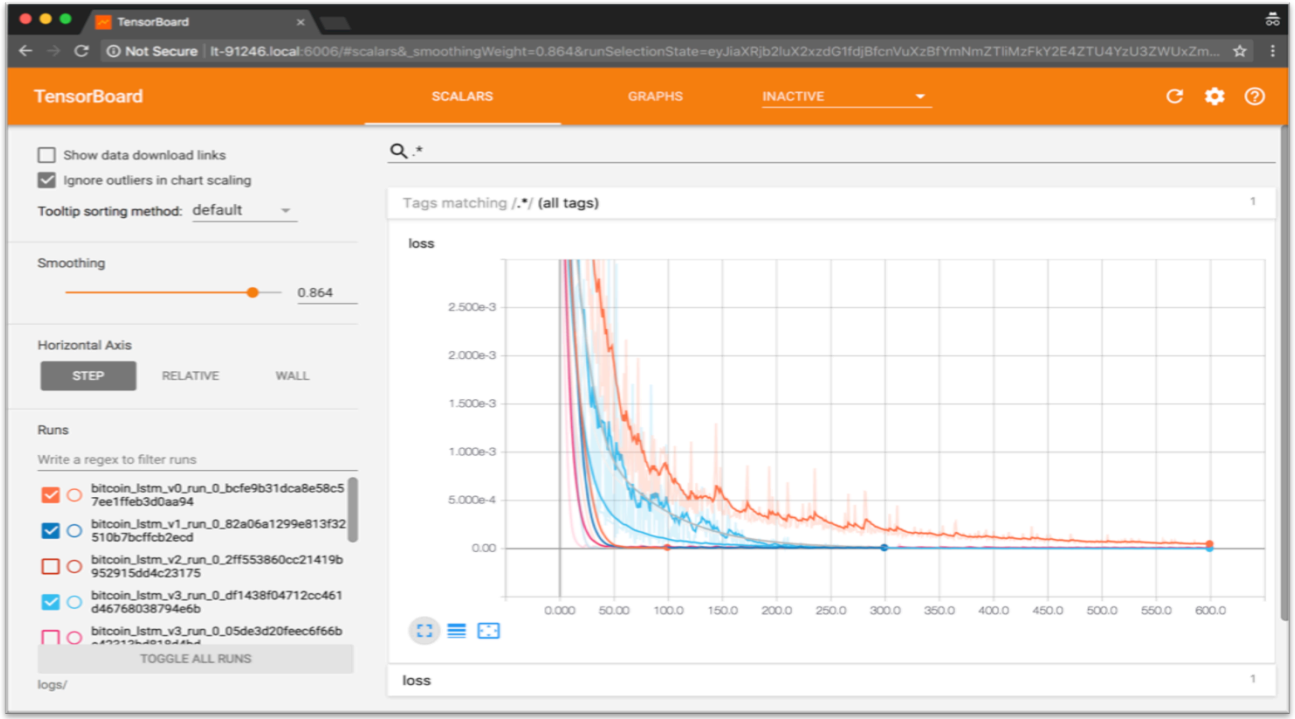
- Now, open the Jupyter Notebook called **`Activity_7_Optimizing_a_deep_learning_model.ipynb`** and navigate to the title of the notebook and import all required libraries.

Activity 7: Optimizing a Deep Learning Model

- Now, navigate to the header **Adding Layers and Nodes**. You will recognize our first model in the next cell. This is the basic LSTM network that we built on *Lesson 2*, *Model Architecture*. Now, we have to add a new LSTM layer to this network.
- Go ahead and add that layer, compile, and train the model.

Activity 7: Optimizing a Deep Learning Model

Open TensorBoard and keep the tab open to watch different model configurations run in real time:



Activity 7: Optimizing a Deep Learning Model

- Now, navigate to the header **Epochs**. In this section, we are interested in exploring different magnitudes of epochs. Use the utility function **`train_model()`** to name different model version and runs:

```
train_model(model=model_v0, X=X_train, Y=Y_train, epochs=100, version=0,
run_number=0)
```

- Train the model with a few different epoch parameters.

Activity 7: Optimizing a Deep Learning Model

- Now, navigate to the header **Activation Functions** in the Notebook. In this section, you only need to change the variable **`activation_function = "tanh"`**
- We have used the **`tanh`** function in this section, but feel free to try other activation functions. Review the list available at https://keras.io/activations/ and try other possibilities.

Activity 7: Optimizing a Deep Learning Model

- Navigate to the header **Regularization Strategies** in the notebook. In this section, you need to implement the `Dropout()` regularization strategy.
- Find the right place to place that step and implement it in our model.
- You can also try the L2 regularization here as well (or combine both).
- Do the same as with `Dropout()`, but now using `ActivityRegularization(l2=0.001)`.

Activity 7: Optimizing a Deep Learning Model

- Now, navigate to the header **Evaluate Models** in the notebook. In this section, we will evaluate the model predictions for the next 19 weeks of data in the test set.
- Then we will compute the RMSE and MAPE of the predicted series versus the test series.
- We have implemented the same evaluation techniques from *Activity 6, Creating an Active Training Environment* all wrapped in utility functions.
- Simply run all the cells from this section until the end of the notebook to see the results.

Summary

- We learned model evaluation and optimization techniques.
- We learned how to evaluate our model using the metrics mean squared error (MSE), squared mean. squared error (RMSE), and mean averaged percentage error (MAPE).
- We looked at optimization techniques typically used to increase the performance of neural networks:
  - Modifying layers and nodes
  - Increasing epochs
  - Changing activation functions
  - And using regularization strategies

THANK YOU!

Beginning Application Development using TensorFlow and Keras

LESSON - 03

Productization

Lesson Objectives

- Handle new data
- Deploy a model as a web application

Dealing with New Data

- The core idea of machine learning models—neural networks included—is that they can learn patterns from data.
- We discuss two strategies on how to deal with new data:
  - Re-training an old model
  - Training a new model

Dealing with New Data - Re-training an Old Model

- One strategy is to re-train an existing model with new data
- However, data used in later training periods may be significantly different to other earlier data
- Such differences may cause significant changes to the model. parameters, making it learn new patterns and "forget" old patterns
- This is called **catastrophic forgetting**

Dealing with New Data - Re-training an Old Model

- The familiar **`model.fit()`** is used to re-train a model without re-creating it:

```
2   X_train_new, Y_train_new = load_new_data()
3
4   model.fit(x=X_train_new, y=Y_train_new,
5             batch_size=1, epochs=100,
6             verbose=0)
```

Dealing with New Data - Re-training an Old Model

- One can create a simple iterator that retrains the model using overlapping windows to every new data available

```
1   M = Model(data=model_data[0*7:7*40 + 7],
2             variable='close',
3             predicted_period_size=7)
4
5   M.build()
6   M.train()
7
8   for i in range(1, 10 + 1):
9       M.train(model_data[i*7:7*(40 + i) + 7])
```

Dealing with New Data – Train New Model

- Another strategy is to create and train a new model every time new data is available
- This approach tends to reduce catastrophic forgetting, but training time increases as data increases

Dealing with New Data - Re-training an Old Model

- Again, the implementation is quite simple

```
1   old_data = model_data[0*7:7*48 + 7]
2   new_data = model_data[0*7:7*49 + 7]
3
4   M = Model(data=old_data,
5             variable='close',
6             predicted_period_size=7)
7
8   M.build()
9   M.train()
10
11  M = Model(data=new_data,
12            variable='close',
13            predicted_period_size=7)
14
15   M.build()
16   M.train()
```

Deploying a Model as a Web-application

- In this lesson, we deploy our model as a web application
- We use a set of modern tools and technologies to do so, including Docker, Flask, and Redis
- The web application is called cryptonic
- A public version of this application is available at https://cryptonic.market

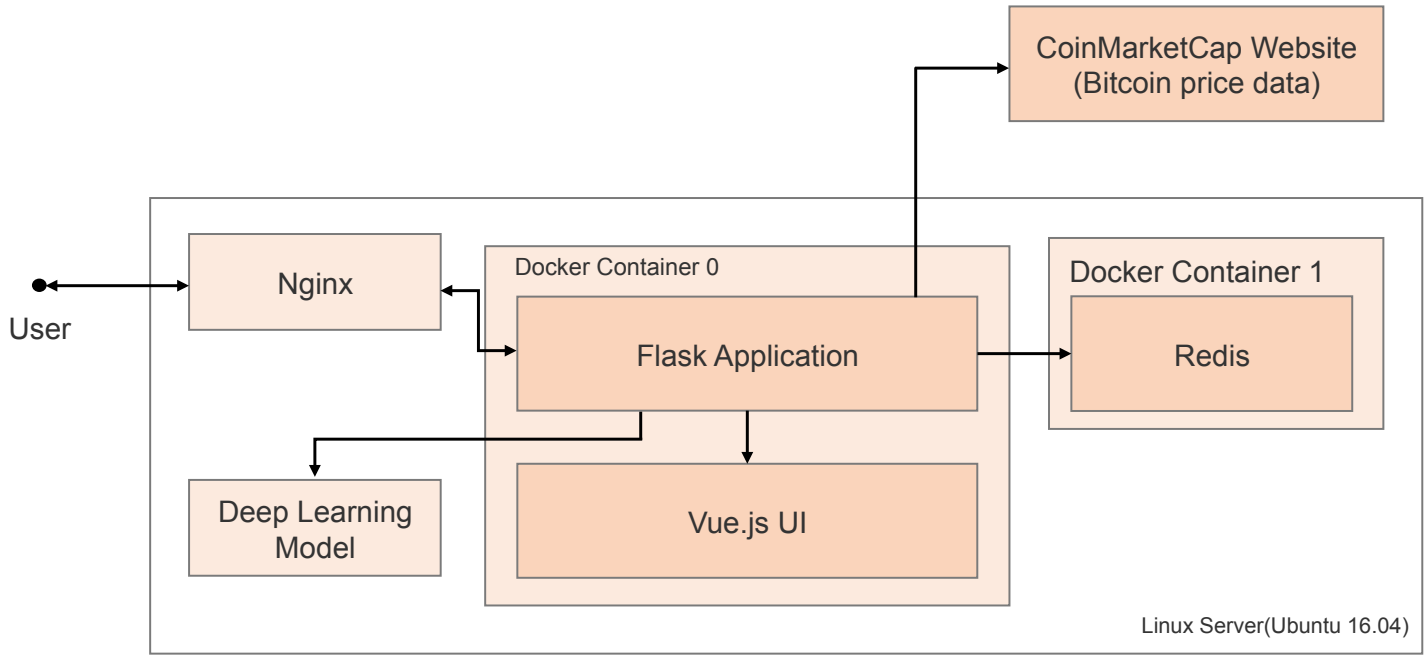
Application Architecture and Technologies

- In order to deploy our web-application, we will use the following tools and technologies as follows:

Tool or Technology	Role
Docker	Packages Python application and UI.
Flask	Creates application routes.
Vue.js	Renders a user interface.
Nginx	Routes traffic between user and Flask application.
Redis	Cache API requests.

Application Architecture and Technologies

- Web application architecture diagram



Application Architecture and Technologies

- Our diagram shows that:
  - A user visits the web-application using her browser
  - She goes to Nginx, then Flask
  - The Flask application either will train a model on start-up or will use a trained model
- After having a model ready:
  - The application verifies if the request has been cached on Redis
  - If yes, it returns the cached data
  - If no cache exists, then it will go ahead and issue predictions which are render in the UI

Deployment and Using Cryptonic

- Our web-application is developed using Docker
- We will use **docker-compose** to deploy it
- Docker needs to be installed—as well as **docker-compose**—in order for the application to be deployed successfully

Deployment and Using Cryptonic

- Docker uses files called Dockerfile to determine how to build images.
- Cryptonic's Dockerfile copies the **`cryptonic`** Python module into a Docker image:

```
1  FROM python:3.6
2  COPY . /cryptonic
3  WORKDIR "/cryptonic"
4  RUN pip install -r requirements.txt
5  EXPOSE 5000
6  CMD ["python", "run.py"]
```

Deployment and Using Cryptonic

- One can then build a Docker image with the docker build command.
- That command will make the image **`cryptonic:latest`** available to be deployed as a container:

```
1  $ docker build --tag cryptonic:latest .
```

Deployment and Using Cryptonic

- After building an image, one can run the Docker container using **docker run**:

```
1  $ docker run --publish 5000:5000 \
2               --detach cryptonic:latest
```

Deployment and Using Cryptonic

- These variables can be used in the terminal or into the **`docker-compose.yml`** (truncated) file as follows:
- (Open the provided file in Visual Studio Code)

```
12    cryptonic:
13        image: cryptonic:latest
14        ports:
15            - "5000:5000"
16        environment:
17            - MODEL_NAME=bitcoin_lstm_v0_trained.h5
17            - BITCOIN_START_DATE=2017-01-01
18            - EPOCH=300
19            - PERIOD_SIZE=7
```

Deployment and Using Cryptonic

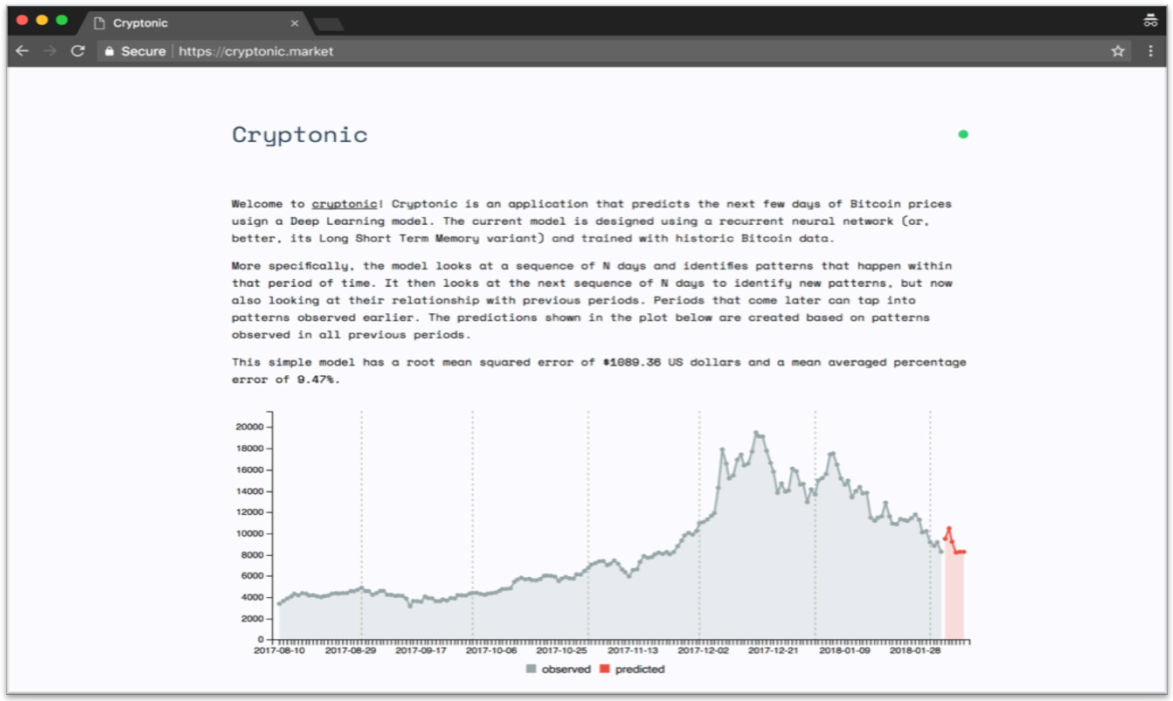
- After editing the **`docker-compose.yml`** file, start the application with:

```
$  docker-compose up
```

Deployment and Using Cryptonic

- Follow the log messages
- At some point you will see an URL (most likely http://localhost:5000)
- Visit that URL and your web-application is now up and running
- You can also use the API via http://localhost:5000/predict

Deployment and Using Cryptonicl



Activity 9: Deploying Deep Learning Application

This activity has the following requirements:

- Docker (Community Edition) 17.12.0-ce or later
- Docker Compose (docker-compose) 1.18.0 or later

Activity 9: Deploying Deep Learning Application

- Using your terminal, navigate to the **cryptonic** directory and build the docker images:

  $ docker build --tag cryptonic:latest .

  $ docker build --tag cryptonic-cache:latest ./ cryptonic-cache/

- These build the two images that we will use in this application **cryptonic** (contain the Flask application) and **cryptonic-cache** (containing the Redis cache).

Activity 9: Deploying Deep Learning Application

- After building the images, identify and open the **`docker-compose.yml`** file and open it in Visual Studio Code.
- Change the parameter **`BITCOIN_START_DATE`** to another date than 2017-01-01:

```
BITCOIN_START_DATE = # Use other date here
```

Activity 9: Deploying Deep Learning Application

- As a final step, deploy your web-application locally using **`docker-compose`** as follows:

  docker-compose up

- You should see a log of activity on your terminal, including training epochs from your model.

Summary

- This lesson concludes our journey into creating a deep learning model and deploying it as a web application.
- We concluded this lesson by deploying a web application
- The challenge now is two fold:
    1. How can you make that model perform better as time passes?
    2. And what features can you add to your web application to make your model more accessible?

Good luck and keep learning!

THANK YOU!