



TENSORFLOW 2.0 PRACTITIONER CHEATSHEET

Scale data using TF2. Divide dataset into training and testing.
Build ANN & CNN



1. WHAT'S NEW IN TENSORFLOW 2.0 COMPARED TO TF 1.0?

- TensorFlow now enables eager execution by default which means that operations can be evaluated immediately.
- Eager execution means that we can now interact with TF 2.0 line by line in Google Colab or Jupyter notebook without the need to define a graph and run sessions and all the complexity that came with TensorFlow 1.0.

1.1 EAGER EXECUTION IS ENABLED BY DEFAULT IN TF 2.0

A. Adding two variables in TF 1.0 was a headache (luckily, not anymore)!

Install and import TensorFlow 1.0

```
!pip install tensorflow-gpu==1.13.01
import tensorflow as tf
```

First we have a “construction phase” where we build a graph


```
>> x = tf.Variable(3)
>> y = tf.Variable(5)
```

Tensorflow created a graph but did not execute the graph yet so a session is needed to run the graph

```
>> z = tf.add(x,y)
```

“Execution phase” where we run a session and this makes it super difficult to debug and develop models

```
>> with tf.Session() as sess:
>> sess.run(tf.global_variables_initializer())
# initialize all variables
>> z = sess.run(z) # run the session
```



```
>> print("The sum of x and y is:", z) # we now get the expected
answer, i.e: 8
```

B. Adding two variables in TF 2.0 is easier than ever!


Install and import TensorFlow 2.0

```
>> !pip install tensorflow-gpu==2.0.0.alpha0
>> import tensorflow as tf
```

TensorFlow 2.0 still works with graphs but enable eager execution by default

Let's add the same variables together

```
>> x = tf.Variable(3)
>> y = tf.Variable(5)
>> z = tf.add(x,y) # immediate answer!
>> print("The sum of x and y is:", z)
# we get the answer immediately!
```




1.2 KERAS IS THE DEFAULT API (NOW EASIER THAN EVER TO TRAIN AND DEBUG MODELS)

- The second important feature in TF 2.0 is the use of keras as the high level API by default.
- Keras is extremely easy to work with since Keras syntax is very pythonic.
- Let's build a mini artificial neural network that can classify fashion images using keras API using couple of lines of code.

Import TensorFlow and load dataset

```
>> import tensorflow as tf
>> fashion_mnist = tf.keras.datasets.fashion_mnist
>> (train_images, train_labels), (test_images, test_labels) =
fashion_mnist.load_data()
```



Build, compile and fit the model to training data

```
>> model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation=tf.nn.relu),
    tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])

>> model.compile(optimizer='adam',
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])

>> model.fit(train_images, train_labels, epochs=5)
```


1.3 EASILY LAUNCH TENSORBOARD

- Tensorboard enable us to track the network progress such as accuracy and loss throughout various epochs along with the graph showing various layers of the network.
- TensorBoard provides a built-in performance dashboard that can be used to track device placement and help minimize bottlenecks during model execution and training.
- Here's how to launch Tensorboard:

```
>> %load_ext tensorboard
>> fashion_mnist = tf.keras.datasets.fashion_mnist

>> (train_images, train_labels), (test_images, test_labels) =
fashion_mnist.load_data()

>> model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation=tf.nn.relu),
    tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])
```




```
>> model.compile(optimizer='adam',
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])

>> log_dir="logs/fit/" +
datetime.datetime.now().strftime("%Y%m%d-%H%M%S")

>> tensorboard_callback =
tf.keras.callbacks.TensorBoard(log_dir=log_dir,
histogram_freq=1)

>> model.fit(train_images, train_labels, epochs=5, callbacks =
[tensorboard_callback])
>> %tensorboard --logdir logs/fit
```




1.4 DISTRIBUTED STRATEGY

- Tensorflow enables distributed strategy which allows developers to develop the model once and then decide how they want to run it later; over multiple GPUs or TPUs.
- This will dramatically improve the computational efficiency with just two additional lines of code.

```
>> !pip install tensorflow-gpu==2.0.0.alpha0
>> fashion_mnist = tf.keras.datasets.fashion_mnist

>> (train_images, train_labels), (test_images, test_labels) =
fashion_mnist.load_data()
>> strategy = tf.distribute.MirroredStrategy()

>> with strategy.scope():
>> model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation=tf.nn.relu),
    tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])
```



```
>> model.compile(optimizer='adam',  
                  loss='sparse_categorical_crossentropy',  
                  metrics=['accuracy'])
```

2. CLASSIFICATION METRICS

2.1 CONFUSION MATRIX CONCEPT

A confusion matrix is used to describe the performance of classification model:

- **True positives (TP):** cases when classifier predicted TRUE (has a disease), and correct class was TRUE (patient has disease).
- **True negatives (TN):** cases when model predicted FALSE (no disease), and correct class was FALSE (patient does not have disease).
- **False positives (FP) (Type I error):** classifier predicted TRUE, but correct class was FALSE (patient does not have disease).
- **False negatives (FN) (Type II error):** classifier predicted FALSE (patient do not have disease), but they actually do have the disease.

| | | TRUE CLASS | |
|-------------|---|----------------------------|---------------------------|
| | | + | - |
| PREDICTIONS | + | TRUE + | FALSE + → Type I error |
| | - | FALSE - ← Type II error | TRUE - |

- **Classification Accuracy** = $(TP+TN) / (TP + TN + FP + FN)$
- **Misclassification rate (Error Rate)** = $(FP + FN) / (TP + TN + FP + FN)$
- **Precision** = $TP / \text{Total TRUE Predictions} = TP / (TP+FP)$ (When model predicted TRUE class, how often did it get it right?)
- **Recall** = $TP / \text{Actual TRUE} = TP / (TP+FN)$ (when the class was actually TRUE, how often did the classifier get it right?)

2.2 CONFUSION MATRIX IN SKLEARN

```
>> from sklearn.metrics import classification_report,
confusion_matrix
```

```
>> y_predict_test = classifier.predict(X_test)
>> cm = confusion_matrix(y_test, y_predict_test)
```

```
>> sns.heatmap(cm, annot=True)
```

2.3 CLASSIFICATION REPORT

```
>> from sklearn.metrics import classification_report
```

```
>> print(classification_report(y_test, y_pred))
```



3. REGRESSION METRICS

3.1 MEAN ABSOLUTE ERROR (MAE)

- Mean Absolute Error (MAE) is obtained by calculating the absolute difference between the model predictions and the true (actual) values.


- MAE is a measure of the average magnitude of error generated by the regression model.
- The mean absolute error (MAE) is calculated as follows:

$$\text{MAE} = \frac{1}{n} \sum_{i=1} |y_i - \hat{y}_i|$$

- MAE is calculated by following these steps:
 1. Calculate the residual for every data point.
 2. Calculate the absolute value (to get rid of the sign).
 3. Calculate the average of all residuals.
- If MAE is zero, this indicates that the model predictions are perfect.

3.2 MEAN SQUARE ERROR (MSE)

- Mean Square Error (MSE) is very similar to the Mean Absolute Error (MAE) but instead of using absolute values, squares of the difference between the model predictions and the training dataset (true values) is being calculated.
- MSE values are generally large compared to the MAE since the residuals are being squared.
- In case of data outliers, MSE will become much larger compared to MAE.
- In MSE, error increases in a quadratic fashion while the error increases in proportional fashion in MAE.
- The MSE is calculated as follows:


$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$


- MAE is calculated by following these steps:
 1. Calculate the residual for every data point.
 2. Calculate the squared value of the residuals.
 3. Calculate the average of all residuals.




3.3 ROOT MEAN SQUARE ERROR (RMSE)

- Root Mean Square Error (RMSE) represents the standard deviation of the residuals (i.e.: differences between the model predictions and the true values (training data)).
- RMSE can be easily interpreted compared to MSE because RMSE units match the units of the output.
- RMSE provides an estimate of how large the residuals are being dispersed.
- The MSE is calculated as follows:

$$\text{RMSE} = \sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2}$$

- RMSE is calculated by following these steps:
 1. Calculate the residual for every data point.
 2. Calculate the squared value of the residuals.
- 

- 
3. Calculate the average of the squared residuals.
 4. Obtain the square root of the result.


3.4 MEAN ABSOLUTE PERCENTAGE ERROR (MAPE)

- MAE values can range from 0 to infinity which makes it difficult to interpret the result as compared to the training data.
- Mean Absolute Percentage Error (MAPE) is the equivalent to MAE but provides the error in a percentage form and therefore overcomes MAE limitations.
- MAPE might exhibit some limitations if the data point value is zero (since there is division operation involved).
- The MAPE is calculated as follows:

$$\text{MAPE} = \frac{100\%}{n} \sum_{i=1}^n |(y_i - \hat{y}_i)/y_i|$$

3.5 MEAN PERCENTAGE ERROR (MPE)

- MPE is similar to MAPE but without the absolute operation.
- MPE is useful to provide an insight of how many positive errors as compared to negative ones.
- The MPE is calculated as follows:

$$\text{MPE} = \frac{100\%}{n} \sum_{i=1}^n (y_i - \hat{y}_i)/y_i$$


4. HOW TO SCALE DATA USING SCIKIT-LEARN?

4.1 CONCEPT

Objective is to scale the training data to (0, 1) or (-1, 1). This step is critical before training Artificial Neural Networks to ensure that all inputs fed to the network are within similar range (i.e.: “treated fairly”). If normalization is ignored, large inputs will dominate smaller ones.

4.2 SCALING DATA USING SCIKIT LEARN

```
>> from sklearn.preprocessing import MinMaxScaler  
>> scaler = MinMaxScaler()  
>> X_scaled = scaler.fit_transform(X)
```

5. HOW TO DIVIDE DATASETS INTO TRAINING AND TESTING?

5.1 CONCEPT

Data set is generally divided into 75% for training and 25% for testing.

- **Training set:** used for model training.
- **Testing set:** used for testing trained model.

Make sure that testing dataset has never been seen by the trained model before.

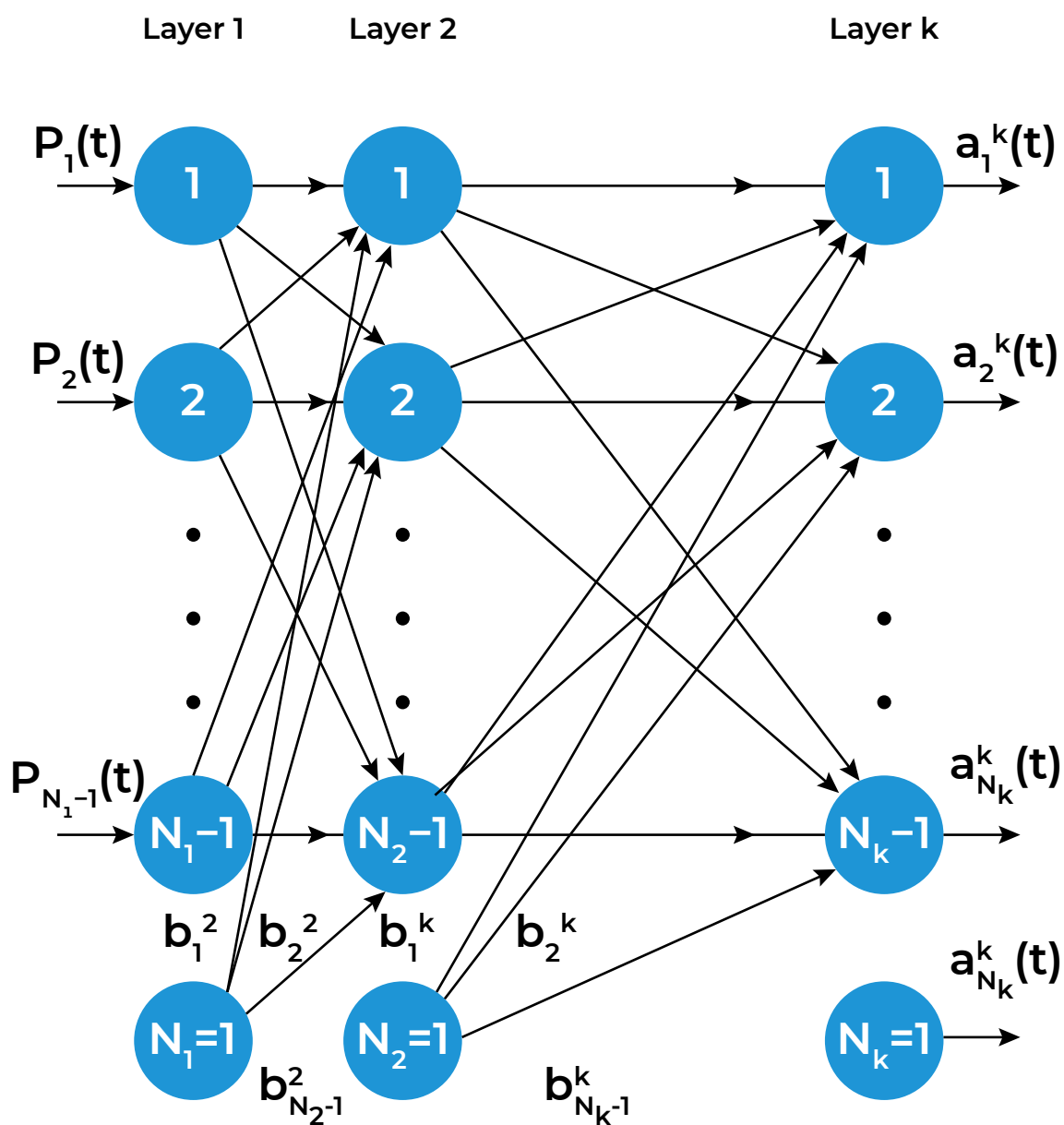
5.2 DIVIDING DATASET USING SCIKIT-LEARN

```
>> from import  sklearn.model_selection train_test_split  
>> X_train, X_test, y_train, y_test = train_test_split(X, y,  
test_size=0.25, random_state=0)
```

6. HOW TO BUILD A BASIC FEED-FORWARD ARTIFICIAL NEURAL NETWORKS (ANNS) IN TF 2.0?

6.1 CONCEPT

Artificial Neural Networks are information processing models inspired by the human brain. ANNs are built in a layered fashion where inputs are propagated starting from the input layer through the hidden layers and finally to the output.



Networks training is performed by optimizing the matrix of weights outlined below:

$$\begin{bmatrix} W_{11} & W_{12} & \cdots & W_{1,N_1} \\ W_{21} & W_{22} & \cdots & W_{2,N_1} \\ \vdots & \vdots & \ddots & \vdots \\ W_{m-1,1} & W_{m-1,2} & \cdots & W_{m-1,N_1} \\ H_{m,1} & W_{m,2} & \cdots & W_{m,N_1} \end{bmatrix}$$

6.2 BUILD AN ANNS USING KERAS

```
>> !pip install tensorflow-gpu==2.0.0.alpha0
>> import tensorflow as tf

>> model = tf.keras.models.Sequential()
>> model.add(tf.keras.layers.Dense(units = 100, activation = 'relu',
input_shape = (35,)))
>> model.add(tf.keras.layers.Dense(units = 100, activation =
'relu'))
>> model.add(tf.keras.layers.Dense(units = 100, activation =
'relu'))
>> model.add(tf.keras.layers.Dense(units = 1, activation = 'linear'))
```

6.3 TRAIN AN ANN USING KERAS

```
>> model.compile(optimizer='adam', loss='mean_squared_error')
>> epochs_hist = model.fit(X_train, y_train, epochs=20,
batch_size=25, validation_split=0.2)
```

6.4 EVALUATE THE TRAINED ANN MODEL

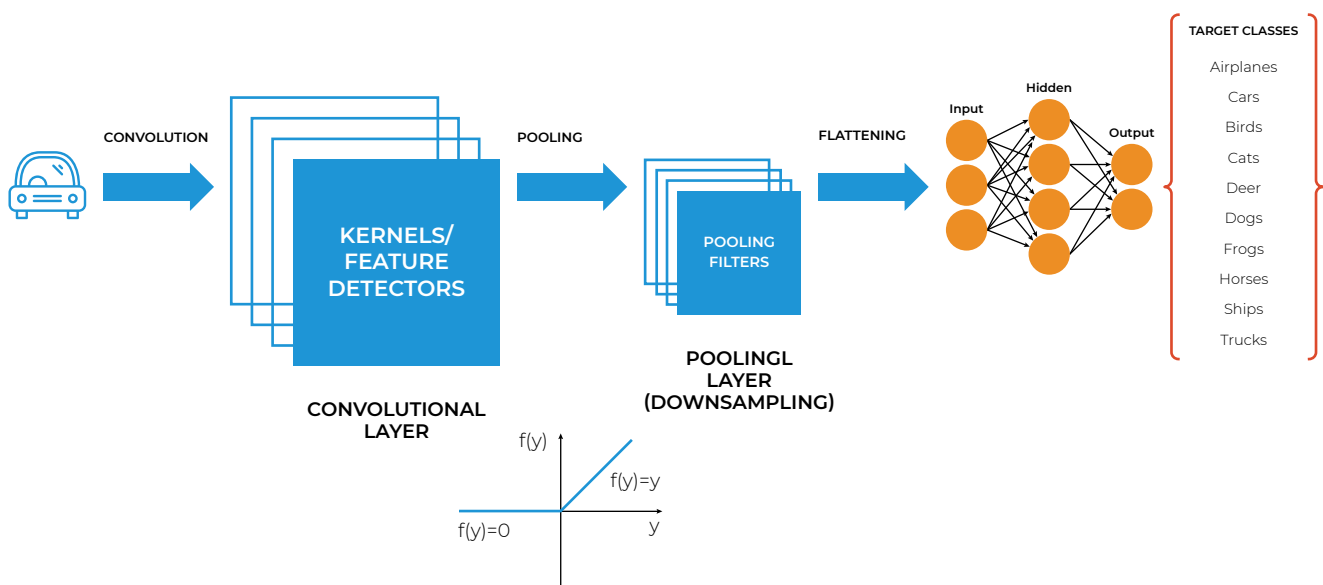
```
>> X_Testing = np.array([[input #1, input #2, input #3,..., input
#n]])
>> y_predict = model.predict(X_Testing)
```

7. HOW TO BUILD A CONVOLUTIONAL NEURAL NETWORK (CNN) IN TF 2.0?

7.1 CONCEPT

CNNs are a type of deep neural networks that are commonly used for image classification.

CNNs are formed of (1) Convolutional Layers (Kernels and feature detectors), (2) Activation Functions (RELU), (3) Pooling Layers (Max Pooling or Average Pooling), and (4) Fully Connected Layers (Multi-layer Perceptron Network).



7.2 BUILD A DEEP CONVOLUTIONAL NEURAL NETWORKS IN TF 2.0:


```
>> cnn = tf.keras.Sequential()
```

```
>> cnn.add(tf.keras.layers.Conv2D(32, (3,3), activation = 'relu',  
input_shape = (32,32,3)))
```

```
>> cnn.add(tf.keras.layers.Conv2D(32, (3,3), activation = 'relu'))
```

```
>> cnn.add(tf.keras.layers.MaxPooling2D(2,2))
```

```
>> cnn.add(tf.keras.layers.Dropout(0.3))
```



```
>> cnn.add(tf.keras.layers.Conv2D(64, (3,3), activation = 'relu'))
>> cnn.add(tf.keras.layers.Conv2D(64, (3,3), activation = 'relu'))


>> cnn.add(tf.keras.layers.MaxPooling2D(2,2))
>> cnn.add(tf.keras.layers.Dropout(0.3))

>> cnn.add(tf.keras.layers.Flatten())

>> cnn.add(tf.keras.layers.Dense(1024, activation = 'relu'))
>> cnn.add(tf.keras.layers.Dropout(0.3))

>> cnn.add(tf.keras.layers.Dense(1024, activation = 'relu'))

>> cnn.add(tf.keras.layers.Dense(10, activation = 'softmax'))
>> cnn.summary()
```






8. HOW TO PERFORM TOKENIZATION?

8.1 CONCEPT

Tokenization is a common procedure in natural language processing. Tokenization works by dividing a sentence into a set of words. These words are then used to train a machine learning model to perform a certain task.

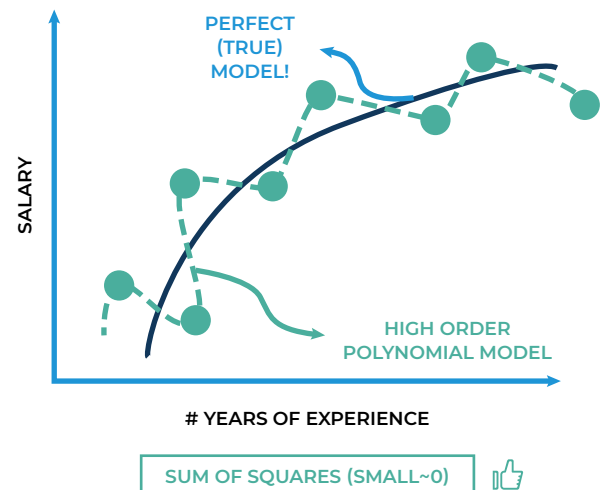
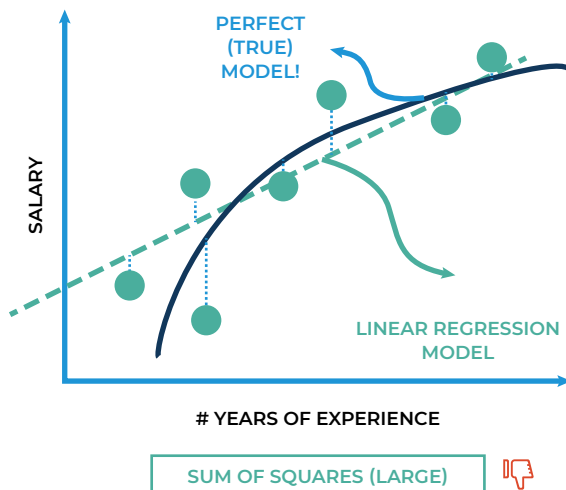
8.2 TOKENIZATION USING SCIKIT-LEARN

```
>> from sklearn.feature_extraction.text import CountVectorizer
>> vectorizer = CountVectorizer()
>> output = vectorizer.fit_transform(data_df)
```

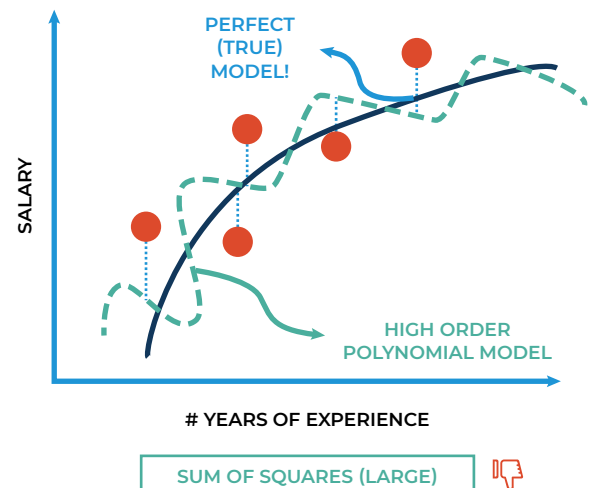
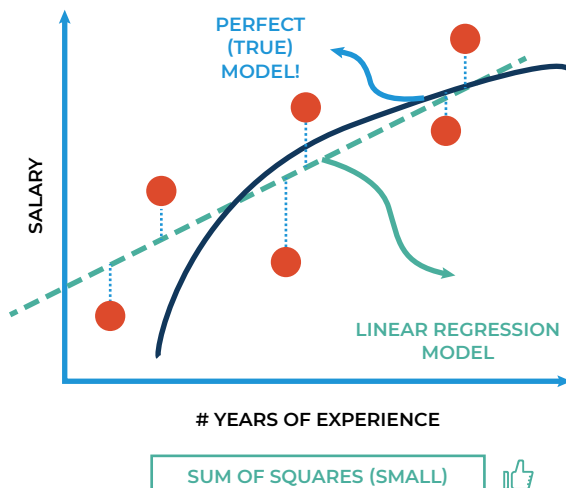


9. BIAS-VARIANCE TRADE-OFF

BIAS AND VARIANCE: MODEL #1 VS. MODEL #2 DURING TRAINING



BIAS AND VARIANCE: MODEL #1 VS. MODEL #2 DURING TESTING



The polynomial model performs poorly on the testing dataset and therefore it has large variance.



MODEL #1 (Linear regression) (Simple)

Model #1 has High bias because it is very rigid (not flexible) and cannot fit the training dataset well.

Model #1 has small variance (variability) because it can fit the training data and the testing data with similar level (the model is able to generalize better) and avoids overfitting.

Model #1 Performance is consistent between the training dataset and the testing dataset.

Good generalization.

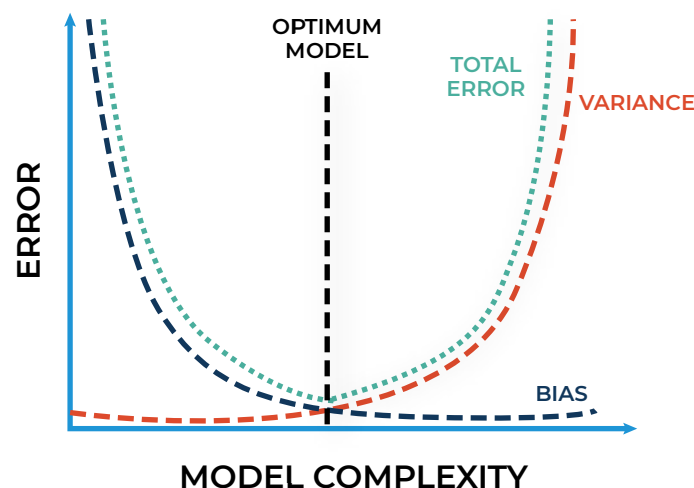
MODEL #2 (High order polynormal) (Complex)

Model #2 has small bias because it is flexible and can fit the training dataset very well.

Model #2 Has large variance (variability) because the model over fitted the training dataset and it performed poorly on the testing dataset.

Model #2 performance varies greatly between the training dataset and the testing dataset (high variability).

Over fitted.



10. TENSORFLOW SERVING

10.1 SAVE THE TRAINED MODEL



```
>> import tempfile # Obtain a temporary storage directory
>> MODEL_DIR = tempfile.gettempdir()
>> version = 1 # specify the model version, choose #1 for now
```

Let's join the temp model directory with our chosen version number

```
>> export_path = os.path.join(MODEL_DIR, str(version))
>> print('export_path = {}'.format(export_path))
```

Save the model using simple_save

```
>> if os.path.isdir(export_path):
    print('\nAlready saved a model, cleaning up\n')
    !rm -r {export_path}

>> tf.saved_model.simple_save(
    keras.backend.get_session(),
    export_path,
    inputs={'input_image': model.input},
    outputs={t.name:t for t in model.outputs})
```

10.2 ADD TENSORFLOW-MODEL-SERVER PACKAGE TO OUR LIST OF PACKAGES

```
>> !echo "deb
http://storage.googleapis.com/tensorflow-serving-apt stable
tensorflow-model-server >> tensorflow-model-server-universal"
| tee /etc/apt/sources.list.d/tensorflow-serving.list && \
curl
https://storage.googleapis.com/tensorflow-serving-apt/tensorflo
w-serving.release.pub.gpg | apt-key add -
>> !apt update
```



10.3 INSTALL TENSORFLOW MODEL SERVER:

```
>> !apt-get install tensorflow-model-server
```

10.4 RUN TENSORFLOW SERVING

```
>> os.environ["MODEL_DIR"] = MODEL_DIR
>> %%bash --bg
>> nohup tensorflow_model_server \
  --rest_api_port=8501 \
  --model_name=fashion_model \
  --model_base_path="${MODEL_DIR}" >server.log 2>&1

>> !tail server.log
```

10.5 START MAKING REQUESTS IN TENSORFLOW SERVING

Let's create a JSON object and make 3 inference requests

```
>> data = json.dumps({"signature_name": "serving_default",
"instances": test_images[0:10].tolist()})
>> print('Data: {} ... {}'.format(data[:50], data[len(data)-52:]))
>> !pip install -q requests
>> import requests
>> headers = {"content-type": "application/json"}
>> json_response =
requests.post('http://localhost:8501/v1/models/fashion_model:pr
edict', data=data, headers=headers)
>> predictions = json.loads(json_response.text)['predictions']
>> show(0, 'The model thought this was a {} (class {}), and it was
actually a {} (class {})'
      class_names[np.argmax(predictions[0])], test_labels[0],
class_names[np.argmax(predictions[0])], test_labels[0]))
```