Keras

[Keras w/ Tensorflow Course]

→ We'll be working w/ a Sequential model from the Keras API.

⇒ model = Sequential([
Dense(units=16, input_shape=(1,), activation='relu'),
⋮
])

→ The sequential model receives data during training whenever we call the fit function on it.

⇒ fit(x=None, y=None, batch_size=None, epochs=1, verbose=1, ...)

Arguments:-

x → • A numpy array (or array-like), or a list of arrays (in case the model has multiple inputs).
• A TensorFlow tensor, or a list of tensors.

- A dict mapping i/p names to the corresponding array/tensors.
- A tf.data dataset. Should return a tuple of either (inputs, targets) or (i/ps, targets, sample_weights).
- A generator or keras.utils.Sequence returning the above.

Y → - Needs to be in the same format as 'x'.

- We should shuffle the data to get rid of any imposed order.

→ from sklearn.utils import shuffle

train_labels, train_samples = shuffle(train_labels, train_samples)

- Also for scaling b/w 0 to 1:-

→ from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler(feature_range=(0,1))

↑ MinMaxScaler Object

```
scaled_train_samples = scaler.fit_transform(train_samples.reshape(-1,1))
```

→ reshape(-1,1) returns a 2D array w/ 1 col, and as many rows necessary to accomodate the data.

The MinMaxScaler doesn't take 1D array by default, so we pass it through this to transform it.

** Calling .shape on a numpy array returns its shape.

Ex:- Array w/ 100 rows, & 1 col. ⇒ 2 rows, 1 cols :- [[1],[2]]

⇒ arr.shape ⇒ (2,1)

> (100,1)

Building the model:-

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Activation, Dense
from tensorflow.keras.optimizers import Adam
```

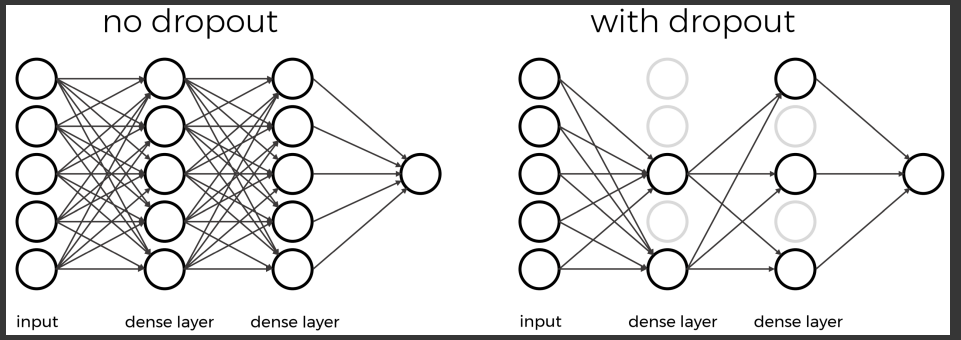
importing the Sequential model from sequential api / Contrast this to the functional api

Dense Layer

```
model = Sequential([
  Dense(units=16, input_shape=(1,), activation='relu'),
  Dense(units=32, activation='relu'),
  Dense(units=2, activation='softmax')
])
```

16 nodes (arbitrary) — relu follows this layer — arb. — for percentages (probs.)

The ① Dense layer is actually not the first layer.
→ It is the first hidden layer.
→ We're not explicitly defining the i/p layer, it is specified by the input_shape parameter.



→ Dense layer refers to a fully connected layer.
→ It may have dropout
→ Called so b/c they're densely connected

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 16)                32
_________________________________________________________________
dense_1 (Dense)              (None, 32)                544
_________________________________________________________________
dense_2 (Dense)              (None, 2)                 66
=================================================================
Total params: 642
Trainable params: 642
Non-trainable params: 0
_________________________________________________________________
```

16+16 ← bias

16*32+32 ← bias

32*2+2 ← bias

```
> model.compile(optimizer = Adam(learning_rate=0.001), loss='sparse_categorical_
                                                          crossentropy',
              metrics = ['accuracy'])
```

↳ Common way to evaluate performance

① → optimizers are used to minimize the loss fxn

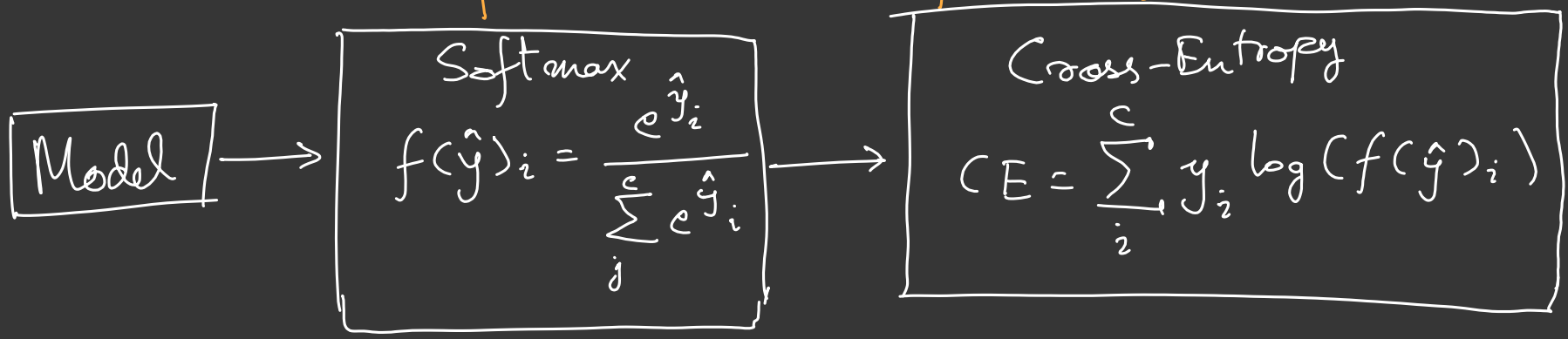
→ Adam :- Adaptive Moment Estimation

↳ can be thought of as a combination of RMS prop and Stochastic Gradient Descent

② Sparse Categorical Cross-Entropy

→ Extension of Categorical Cross Entropy, whose o/p labels are represented in Sparse matrix format.

- Categorical Cross-Entropy :- Softmax loss
  - → Softmax Activation + Cross Entropy loss
  - → Used for Multiclass Classification. (for CNNs)



```
> model.fit(x=scaled_train_samples, y=train_labels, validation_split=0.1,
            batch_size=10, epochs=30, shuffle=True, verbose=2)
```

x → already done

y → target_data, done

batch size → how many data points to be included in one pass

```
Epoch 1/30
2100/2100
```

↑ this number is total data divided by batch size

epochs → ✓

shuffle → By default already True, make False for Time Series

verbose = [0, 1, 2] → most verbose

↓

no verbose

Validation Set

→ The model is still learning on or 'training on' the training data

→ It allows us to see how general our model is/ how well it is able to generalize on data. This is done for each epoch.

→ So, if we see the model is giving very good result for the training set, but sub-par results for the validation set, then the model is overfit.

→ There are 2 ways of creating a validation set :-

```
model.fit ( x = scaled_train_samples, y = train_labels,
        ① validation_data = valid_set, ② validation_split = 0.1,
          batch_size = 10,
          epochs = 30,
        * shuffle = True,
          verbose = 2)
```

→ The shuffle function is called after splitting the data, so let's say we have a dataset where it is arranged by age, so the last 10% will have one parameter not shuffled throughout, so it is important to pre-shuffle the data before calling split_validation.

Testing (Inference)

→ Make a test set.

→ Make it into a numpy array.

→ Then shuffle it

→ Then reshape it and scale it accordingly

Predict:-

```
predictions = model.predict(x=scaled_test_samples, batch_size=10, verbose=0)
```

→ should be the same batch size

→ This gives us a number b/w 0 & 1

returns an array [0.92, 0.077] → no side eff. → side eff.

→ Take the index of the higher index for only the prediction or the higher prob.

→ 0
1
1
...

Confusion Matrix

→ It is in sklearn

```
%matplotlib inline
from sklearn.metrics import confusion_matrix
import itertools
import matplotlib.pyplot as plt

cm = confusion_matrix(y_true = test_labels, y_pred = rounded_predictions)
```

→ def plot_confusion_matrix(...)

ou scikit learn website

```
cm_plot_labels = ['no_side_effects', 'had_side_effects']
plot_confusion_matrix(cm=cm, classes = cm_plot_labels, title = 'Confusion Matrix')
```

Saving & Loading a Model

Saving:-

```
import os.path
if os.path.isfile('models/model_name.h5') is False:
    model.save('models/model_name.h5')
```

This save function saves:-

- The architecture of the model
- The weights of the model
- The training configeration
- The state of the optimizer ; allowing to resume training exactly where it was left off.

Loading:-

```
from tensorflow.keras.models import load_model
new_modd = load_model('models/name.h5')
```

model.get_weights() → gives the weights of the model

From & to JSON or YAML

```
json_string = model.to_json()

yaml_string = model.to_yaml()

from tensorflow.keras.models import model_from_json
model_arch = model_from_json(json_string)
```

→ This does not save the weights for the model.
→ We need model.save_weights() for that.

```
if os.path.isfile('models/model_name.h5') is False:
    model.save_weights('models/model_name.h5')
```

```
model2 = load model...

model2.load_weights('models/model_name.h5')
```

Convolutional Neural Networks