index

May 6, 2022

1 Deeper Neural Networks - Lab

1.1 Introduction

In this lesson, we'll dig deeper into the work horse of deep learning, ***Multi-Layer Perceptrons***! We'll build and train a couple of different MLPs with Keras and explore the tradeoffs that come with adding extra hidden layers. We'll also try switching between some of the activation functions we learned about in the previous lesson to see how they affect training and performance.

1.2 Objectives

- Build a deep neural network using Keras

1.3 Getting Started

Run the cell below to import everything we'll need for this lab.

[1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import keras
from keras.models import Sequential
from keras.layers import Dense
from sklearn.datasets import load_breast_cancer
from sklearn.preprocessing import StandardScaler, LabelBinarizer
```

For this lab, we'll be working with the Boston Breast Cancer Dataset. Although we're importing this dataset directly from scikit-learn, the Kaggle link above contains a detailed explanation of the dataset, in case you're interested. We recommend you take a minute to familiarize yourself with the dataset before digging in.

In the cell below:

- Call `load_breast_cancer()` to store the dataset
- Access the `.data`, `.target`, and `.feature_names` attributes and store them in the appropriate variables below

[2]:
```
bc_dataset = load_breast_cancer()
data = bc_dataset.data
target = bc_dataset.target
col_names = bc_dataset.feature_names
```

Now, let's create a DataFrame so that we can see the data and explore it a bit more easily with the column names attached.

- In the cell below, create a pandas DataFrame from `data` (use `col_names` for column names)
- Print the `.head()` of the DataFrame

[3]:
```
df = pd.DataFrame(data, columns = col_names)
df.head()
```

[3]:
```
   mean radius  mean texture  mean perimeter  mean area  mean smoothness  \
0        17.99         10.38          122.80     1001.0          0.11840
1        20.57         17.77          132.90     1326.0          0.08474
2        19.69         21.25          130.00     1203.0          0.10960
3        11.42         20.38           77.58      386.1          0.14250
4        20.29         14.34          135.10     1297.0          0.10030

   mean compactness  mean concavity  mean concave points  mean symmetry  \
0           0.27760          0.3001              0.14710         0.2419
1           0.07864          0.0869              0.07017         0.1812
2           0.15990          0.1974              0.12790         0.2069
3           0.28390          0.2414              0.10520         0.2597
4           0.13280          0.1980              0.10430         0.1809

   mean fractal dimension  …  worst radius  worst texture  worst perimeter  \
0                 0.07871  …         25.38          17.33           184.60
1                 0.05667  …         24.99          23.41           158.80
2                 0.05999  …         23.57          25.53           152.50
3                 0.09744  …         14.91          26.50            98.87
4                 0.05883  …         22.54          16.67           152.20

   worst area  worst smoothness  worst compactness  worst concavity  \
0      2019.0            0.1622             0.6656           0.7119
1      1956.0            0.1238             0.1866           0.2416
2      1709.0            0.1444             0.4245           0.4504
3       567.7            0.2098             0.8663           0.6869
4      1575.0            0.1374             0.2050           0.4000

   worst concave points  worst symmetry  worst fractal dimension
0                0.2654          0.4601                  0.11890
1                0.1860          0.2750                  0.08902
2                0.2430          0.3613                  0.08758
3                0.2575          0.6638                  0.17300
```

```
4                0.1625          0.2364                   0.07678

[5 rows x 30 columns]
```

1.4 Getting the Data Ready for Deep Learning

In order to pass this data into a neural network, we'll need to make sure that the data:

- is purely numerical
- contains no missing values
- is normalized

Let's begin by calling the DataFrame's `.info()` method to check the datatype of each feature.

[4]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 569 entries, 0 to 568
Data columns (total 30 columns):
 #   Column                   Non-Null Count  Dtype
---  ------                   --------------  -----
 0   mean radius              569 non-null    float64
 1   mean texture             569 non-null    float64
 2   mean perimeter           569 non-null    float64
 3   mean area                569 non-null    float64
 4   mean smoothness          569 non-null    float64
 5   mean compactness         569 non-null    float64
 6   mean concavity           569 non-null    float64
 7   mean concave points      569 non-null    float64
 8   mean symmetry            569 non-null    float64
 9   mean fractal dimension   569 non-null    float64
 10  radius error             569 non-null    float64
 11  texture error            569 non-null    float64
 12  perimeter error          569 non-null    float64
 13  area error               569 non-null    float64
 14  smoothness error         569 non-null    float64
 15  compactness error        569 non-null    float64
 16  concavity error          569 non-null    float64
 17  concave points error     569 non-null    float64
 18  symmetry error           569 non-null    float64
 19  fractal dimension error  569 non-null    float64
 20  worst radius             569 non-null    float64
 21  worst texture            569 non-null    float64
 22  worst perimeter          569 non-null    float64
 23  worst area               569 non-null    float64
 24  worst smoothness         569 non-null    float64
 25  worst compactness        569 non-null    float64
 26  worst concavity          569 non-null    float64
 27  worst concave points     569 non-null    float64
```

```
 28  worst symmetry           569 non-null    float64
 29  worst fractal dimension  569 non-null    float64
dtypes: float64(30)
memory usage: 133.5 KB
```

From the output above, we can see that the entire dataset is already in numerical format. We can also see from the counts that each feature has the same number of entries as the number of rows in the DataFrame – that means that no feature contains any missing values. Great!

Now, let's check to see if our data needs to be normalized. Instead of doing statistical tests here, let's just take a quick look at the `.head()` of the DataFrame again. Do this in the cell below.

[5]:
```python
df.head()
```

[5]:
```
   mean radius  mean texture  mean perimeter  mean area  mean smoothness  \
0        17.99         10.38          122.80     1001.0          0.11840
1        20.57         17.77          132.90     1326.0          0.08474
2        19.69         21.25          130.00     1203.0          0.10960
3        11.42         20.38           77.58      386.1          0.14250
4        20.29         14.34          135.10     1297.0          0.10030

   mean compactness  mean concavity  mean concave points  mean symmetry  \
0           0.27760          0.3001              0.14710         0.2419
1           0.07864          0.0869              0.07017         0.1812
2           0.15990          0.1974              0.12790         0.2069
3           0.28390          0.2414              0.10520         0.2597
4           0.13280          0.1980              0.10430         0.1809

   mean fractal dimension  …  worst radius  worst texture  worst perimeter  \
0                 0.07871  …         25.38          17.33           184.60
1                 0.05667  …         24.99          23.41           158.80
2                 0.05999  …         23.57          25.53           152.50
3                 0.09744  …         14.91          26.50            98.87
4                 0.05883  …         22.54          16.67           152.20

   worst area  worst smoothness  worst compactness  worst concavity  \
0      2019.0            0.1622             0.6656           0.7119
1      1956.0            0.1238             0.1866           0.2416
2      1709.0            0.1444             0.4245           0.4504
3       567.7            0.2098             0.8663           0.6869
4      1575.0            0.1374             0.2050           0.4000

   worst concave points  worst symmetry  worst fractal dimension
0                0.2654          0.4601                  0.11890
1                0.1860          0.2750                  0.08902
2                0.2430          0.3613                  0.08758
3                0.2575          0.6638                  0.17300
4                0.1625          0.2364                  0.07678
```

```
[5 rows x 30 columns]
```

As we can see from comparing `mean radius` and `mean area`, columns are clearly on different scales, which means that we need to normalize our dataset. To do this, we'll make use of scikit-learn's `StandardScaler()` class.

In the cell below, instantiate a `StandardScaler` and use it to create a normalized version of our dataset.

[6]:
```
scaler = StandardScaler()
scaled_data = scaler.fit_transform(df)
```

1.5 Binarizing our Labels

If you took a look at the data dictionary on Kaggle, then you probably noticed the target for this dataset is to predict if the sample is "M" (Malignant) or "B" (Benign). This means that this is a ***Binary Classification*** task, so we'll need to binarize our labels.

In the cell below, make use of scikit-learn's `LabelBinarizer()` class to create a binarized version of our labels.

[7]:
```
binarizer = LabelBinarizer()
labels = binarizer.fit_transform(target)
```

1.6 Building our MLP

Now, we'll build a small ***Multi-Layer Perceptron*** using Keras in the cell below. Our first model will act as a baseline, and then we'll make it bigger to see what happens to model performance.

In the cell below:

- Instantiate a `Sequential()` Keras model
- Use the model's `.add()` method to add a `Dense` layer with 10 neurons and a `'tanh'` activation function. Also set the `input_shape` attribute to `(30,)`, since we have 30 features
- Since this is a binary classification task, the output layer should be a `Dense` layer with a single neuron, and the activation set to `'sigmoid'`

[8]:
```
model_1 = Sequential()
model_1.add(Dense(10, activation = "tanh", input_shape = (30,)))
model_1.add(Dense(1, activation = "sigmoid"))
```

1.6.1 Compiling the Model

Now that we've created the model, the next step is to compile it.

In the cell below, compile the model. Set the following hyperparameters:

- `loss='binary_crossentropy'`

- optimizer='sgd'
- metrics=['acc']

[31]:
```
model_1.compile(loss = "binary_crossentropy",
                optimizer = "sgd",
                metrics = ["acc"] )
```

1.6.2 Fitting the Model

Now, let's fit the model. Set the following hyperparameters:

- epochs=25
- batch_size=1
- validation_split=0.2

[11]:
```
results_1 = model_1.fit(scaled_data, labels, epochs = 25,
                       batch_size = 1, validation_split = 0.2)
```

```
Epoch 1/25
455/455 [==============================] - 0s 950us/step - loss: 1.9093 - acc:
0.1538 - val_loss: 1.4120 - val_acc: 0.0351
Epoch 2/25
455/455 [==============================] - 0s 720us/step - loss: 0.7988 - acc:
0.0505 - val_loss: 0.9609 - val_acc: 0.0439
Epoch 3/25
455/455 [==============================] - 0s 715us/step - loss: 0.9302 - acc:
0.0945 - val_loss: 1.5727 - val_acc: 0.1842
Epoch 4/25
455/455 [==============================] - 0s 731us/step - loss: 1.3028 - acc:
0.1495 - val_loss: 1.7584 - val_acc: 0.1754
Epoch 5/25
455/455 [==============================] - 0s 756us/step - loss: 1.2815 - acc:
0.1560 - val_loss: 2.1827 - val_acc: 0.2018
Epoch 6/25
455/455 [==============================] - 0s 702us/step - loss: 1.6050 - acc:
0.1516 - val_loss: 1.9713 - val_acc: 0.2105
Epoch 7/25
455/455 [==============================] - 0s 876us/step - loss: 1.4640 - acc:
0.1582 - val_loss: 1.8094 - val_acc: 0.2281
Epoch 8/25
455/455 [==============================] - 0s 750us/step - loss: 1.3337 - acc:
0.1912 - val_loss: 1.6996 - val_acc: 0.2368
Epoch 9/25
455/455 [==============================] - 0s 724us/step - loss: 1.1979 - acc:
0.2000 - val_loss: 1.5583 - val_acc: 0.2281
Epoch 10/25
455/455 [==============================] - 0s 716us/step - loss: 1.0616 - acc:
0.2044 - val_loss: 1.4458 - val_acc: 0.2544
Epoch 11/25
```

```
455/455 [==============================] - 0s 742us/step - loss: 0.9871 - acc:
0.2132 - val_loss: 1.3008 - val_acc: 0.2544
Epoch 12/25
455/455 [==============================] - 0s 731us/step - loss: 0.8857 - acc:
0.2088 - val_loss: 1.1783 - val_acc: 0.2719
Epoch 13/25
455/455 [==============================] - 0s 737us/step - loss: 0.7515 - acc:
0.2374 - val_loss: 1.0375 - val_acc: 0.2719
Epoch 14/25
455/455 [==============================] - 0s 701us/step - loss: 0.6218 - acc:
0.2484 - val_loss: 0.8627 - val_acc: 0.2982
Epoch 15/25
455/455 [==============================] - 0s 730us/step - loss: 0.5102 - acc:
0.2725 - val_loss: 0.7502 - val_acc: 0.3333
Epoch 16/25
455/455 [==============================] - 0s 715us/step - loss: 0.3837 - acc:
0.3209 - val_loss: 0.6953 - val_acc: 0.3509
Epoch 17/25
455/455 [==============================] - 0s 710us/step - loss: 0.3063 - acc:
0.3253 - val_loss: 0.6293 - val_acc: 0.3684
Epoch 18/25
455/455 [==============================] - 0s 754us/step - loss: 0.2491 - acc:
0.3451 - val_loss: 0.5620 - val_acc: 0.3860
Epoch 19/25
455/455 [==============================] - 0s 745us/step - loss: 0.2117 - acc:
0.3780 - val_loss: 0.4927 - val_acc: 0.4298
Epoch 20/25
455/455 [==============================] - 0s 720us/step - loss: 0.1818 - acc:
0.4110 - val_loss: 0.4763 - val_acc: 0.4298
Epoch 21/25
455/455 [==============================] - 0s 758us/step - loss: 0.1700 - acc:
0.4220 - val_loss: 0.4508 - val_acc: 0.4561
Epoch 22/25
455/455 [==============================] - 0s 706us/step - loss: 0.1594 - acc:
0.4549 - val_loss: 0.3848 - val_acc: 0.4737
Epoch 23/25
455/455 [==============================] - 0s 713us/step - loss: 0.1529 - acc:
0.4681 - val_loss: 0.3597 - val_acc: 0.4649
Epoch 24/25
455/455 [==============================] - 0s 734us/step - loss: 0.1511 - acc:
0.4857 - val_loss: 0.3548 - val_acc: 0.4825
Epoch 25/25
455/455 [==============================] - 0s 714us/step - loss: 0.1491 - acc:
0.4791 - val_loss: 0.3316 - val_acc: 0.4825
```

[17]:
```
results_1.history.keys()
```

[17]: dict_keys(['loss', 'acc', 'val_loss', 'val_acc'])

Note that when you call a Keras model's `.fit()` method, it returns a Keras callback containing information on the training process of the model. If you examine the callback's `.history` attribute, you'll find a dictionary containing both the training and validation loss, as well as any metrics we specified when compiling the model (in this case, just accuracy).

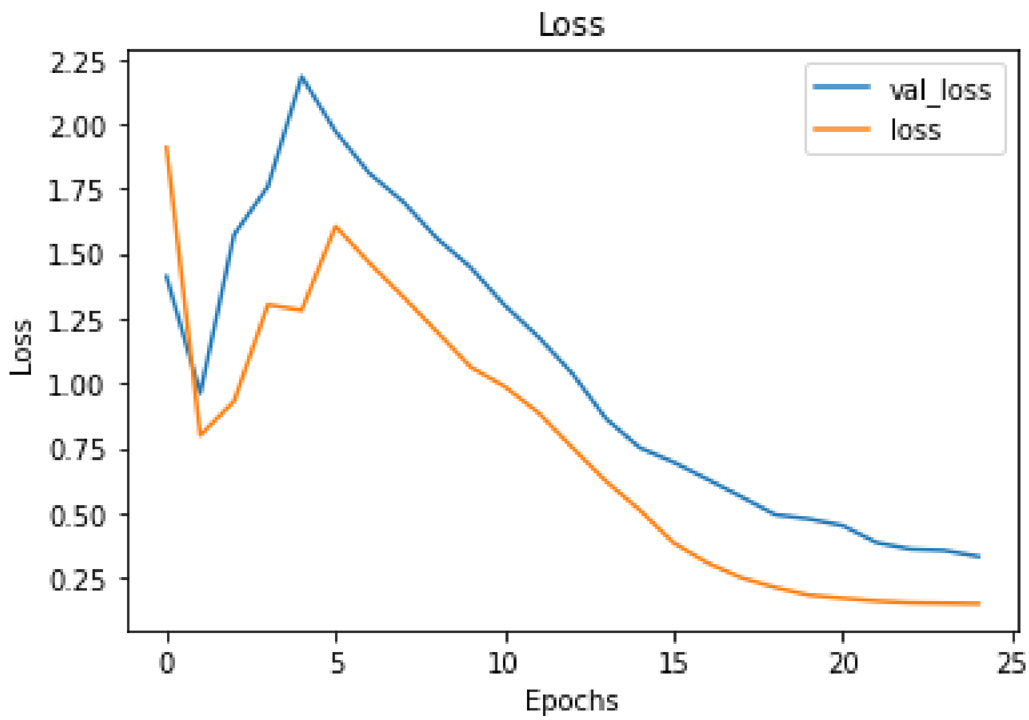
Let's quickly plot our validation and accuracy curves and see if we notice anything. Since we'll want to do this anytime we train an MLP, its worth wrapping this code in a function so that we can easily reuse it.

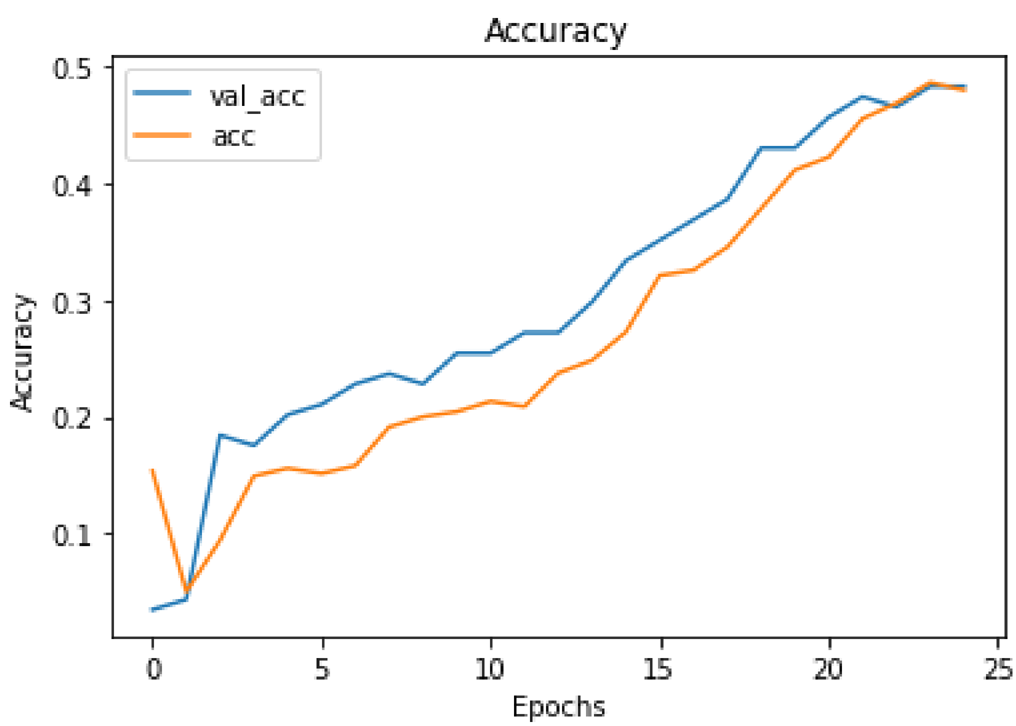
In the cell below, we created a function for visualizing the loss and accuracy metrics.

[18]:
```
def visualize_training_results(results):
    history = results.history
    plt.figure()
    plt.plot(history['val_loss'])
    plt.plot(history['loss'])
    plt.legend(['val_loss', 'loss'])
    plt.title('Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.show()

    plt.figure()
    plt.plot(history['val_acc'])
    plt.plot(history['acc'])
    plt.legend(['val_acc', 'acc'])
    plt.title('Accuracy')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.show()
```

[19]:
```
visualize_training_results(results_1)
```


Loss
val_loss
loss
2.25
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
Loss
0
5
10
15
20
25
Epochs


Accuracy
val_acc
acc
0.5
0.4
0.3
0.2
0.1
Accuracy
0
5
10
15
20
25
Epochs

1.7 Detecting Overfitting

You'll probably notice that the model did pretty well! It's always recommended to visualize your training and validation metrics against each other after training a model. By plotting them like this, we can easily detect when the model is starting to overfit. We can tell that this is happening by seeing the model's training performance steadily improve long after the validation performance plateaus. We can see that in the plots above as the training loss continues to decrease and the training accuracy continues to increase, and the distance between the two lines gets greater as the epochs gets higher.

1.8 Iterating on the Model

By adding another hidden layer, we can a given the model the ability to capture more high-level abstraction in the data. However, increasing the depth of the model also increases the amount of data the model needs to converge to answer, because with a more complex model comes the "Curse of Dimensionality", thanks to all the extra trainable parameters that come from adding more size to our network.

If there is complexity in the data that our smaller model was not big enough to catch, then a larger model may improve performance. However, if our dataset isn't big enough for the new, larger model, then we may see performance decrease as then model "thrashes" about a bit, failing to converge. Let's try and see what happens.

In the cell below, recreate the model that you created above, with one exception. In the model below, add a second `Dense` layer with `'tanh'` activation function and 5 neurons after the first. The network's output layer should still be a `Dense` layer with a single neuron and a `'sigmoid'` activation function, since this is still a binary classification task.

Create, compile, and fit the model in the cells below, and then visualize the results to compare the history.

[23]:
```
model_2 = Sequential()
model_2.add(Dense(10, activation = "tanh", input_shape = (30,)))
model_2.add(Dense(5 , activation = "tanh"))
model_2.add(Dense(1 , activation = "sigmoid"))
```

[24]:
```
model_2.compile(loss = "binary_crossentropy",
                optimizer = "sgd",
                metrics = ["acc"] )
```

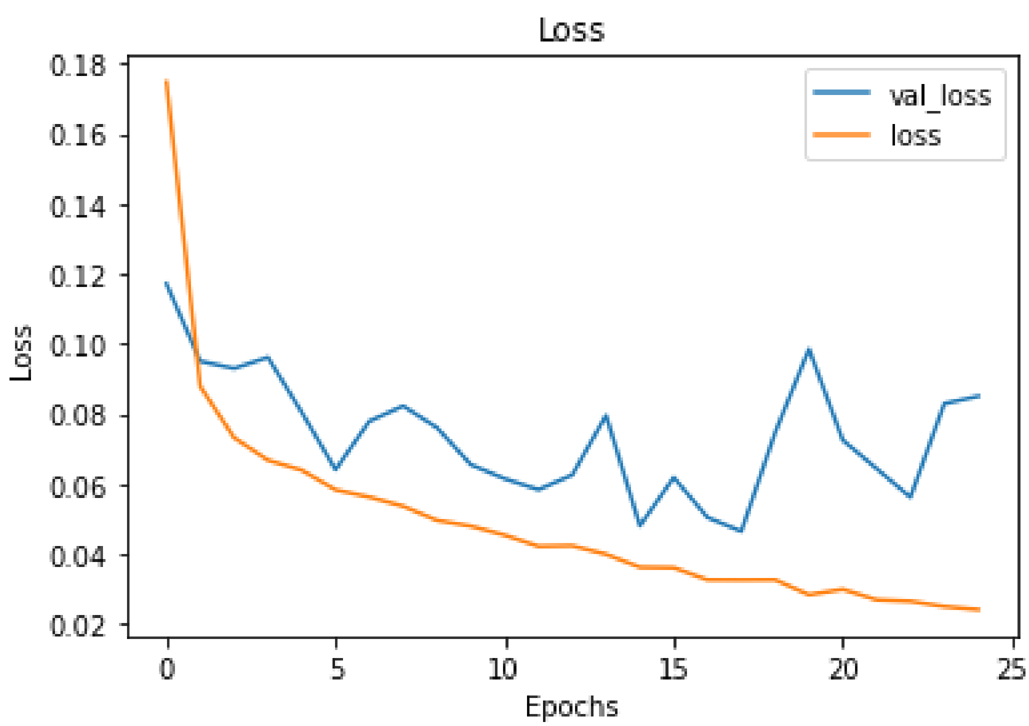
[25]:
```
results_2 = model_2.fit(scaled_data, labels, epochs = 25,
                       batch_size = 1, validation_split = 0.2)
```

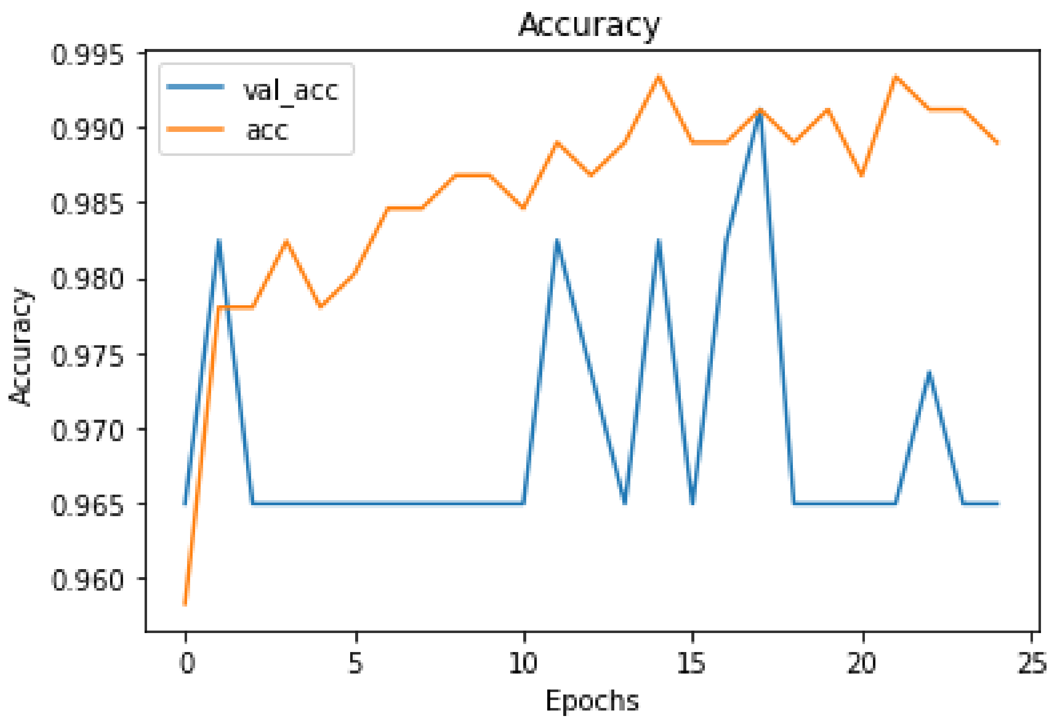
```
Epoch 1/25
455/455 [==============================] - 0s 973us/step - loss: 0.1748 - acc:
0.9582 - val_loss: 0.1172 - val_acc: 0.9649
Epoch 2/25
455/455 [==============================] - 0s 954us/step - loss: 0.0878 - acc:
0.9780 - val_loss: 0.0950 - val_acc: 0.9825
Epoch 3/25
```

```
455/455 [==============================] - 0s 887us/step - loss: 0.0732 - acc:
0.9780 - val_loss: 0.0930 - val_acc: 0.9649
Epoch 4/25
455/455 [==============================] - 0s 894us/step - loss: 0.0667 - acc:
0.9824 - val_loss: 0.0961 - val_acc: 0.9649
Epoch 5/25
455/455 [==============================] - 0s 798us/step - loss: 0.0639 - acc:
0.9780 - val_loss: 0.0804 - val_acc: 0.9649
Epoch 6/25
455/455 [==============================] - 0s 811us/step - loss: 0.0583 - acc:
0.9802 - val_loss: 0.0640 - val_acc: 0.9649
Epoch 7/25
455/455 [==============================] - 0s 753us/step - loss: 0.0562 - acc:
0.9846 - val_loss: 0.0779 - val_acc: 0.9649
Epoch 8/25
455/455 [==============================] - 0s 801us/step - loss: 0.0536 - acc:
0.9846 - val_loss: 0.0822 - val_acc: 0.9649
Epoch 9/25
455/455 [==============================] - 0s 779us/step - loss: 0.0496 - acc:
0.9868 - val_loss: 0.0760 - val_acc: 0.9649
Epoch 10/25
455/455 [==============================] - 0s 721us/step - loss: 0.0479 - acc:
0.9868 - val_loss: 0.0655 - val_acc: 0.9649
Epoch 11/25
455/455 [==============================] - 0s 780us/step - loss: 0.0454 - acc:
0.9846 - val_loss: 0.0615 - val_acc: 0.9649
Epoch 12/25
455/455 [==============================] - 0s 731us/step - loss: 0.0421 - acc:
0.9890 - val_loss: 0.0583 - val_acc: 0.9825
Epoch 13/25
455/455 [==============================] - 0s 736us/step - loss: 0.0423 - acc:
0.9868 - val_loss: 0.0626 - val_acc: 0.9737
Epoch 14/25
455/455 [==============================] - 0s 784us/step - loss: 0.0399 - acc:
0.9890 - val_loss: 0.0795 - val_acc: 0.9649
Epoch 15/25
455/455 [==============================] - 0s 714us/step - loss: 0.0362 - acc:
0.9934 - val_loss: 0.0480 - val_acc: 0.9825
Epoch 16/25
455/455 [==============================] - 0s 780us/step - loss: 0.0360 - acc:
0.9890 - val_loss: 0.0618 - val_acc: 0.9649
Epoch 17/25
455/455 [==============================] - 0s 777us/step - loss: 0.0325 - acc:
0.9890 - val_loss: 0.0504 - val_acc: 0.9825
Epoch 18/25
455/455 [==============================] - 0s 763us/step - loss: 0.0325 - acc:
0.9912 - val_loss: 0.0464 - val_acc: 0.9912
Epoch 19/25
```

```
455/455 [==============================] - 0s 766us/step - loss: 0.0326 - acc:
0.9890 - val_loss: 0.0746 - val_acc: 0.9649
Epoch 20/25
455/455 [==============================] - 0s 742us/step - loss: 0.0283 - acc:
0.9912 - val_loss: 0.0985 - val_acc: 0.9649
Epoch 21/25
455/455 [==============================] - 0s 769us/step - loss: 0.0298 - acc:
0.9868 - val_loss: 0.0725 - val_acc: 0.9649
Epoch 22/25
455/455 [==============================] - 0s 773us/step - loss: 0.0268 - acc:
0.9934 - val_loss: 0.0644 - val_acc: 0.9649
Epoch 23/25
455/455 [==============================] - 0s 742us/step - loss: 0.0264 - acc:
0.9912 - val_loss: 0.0562 - val_acc: 0.9737
Epoch 24/25
455/455 [==============================] - 0s 831us/step - loss: 0.0249 - acc:
0.9912 - val_loss: 0.0829 - val_acc: 0.9649
Epoch 25/25
455/455 [==============================] - 0s 784us/step - loss: 0.0240 - acc:
0.9890 - val_loss: 0.0850 - val_acc: 0.9649
```

[26]:
```
visualize_training_results(results_2)
```





1.9 What Happened?

Although the final validation score for both models is the same, this model is clearly worse because it hasn't converged yet. We can tell because of the greater variance in the movement of the `val_loss` and `val_acc` lines. This suggests that we can remedy this by either:

- Decreasing the size of the network, or
- Increasing the size of our training data

1.10 Visualizing why we Normalize our Data

As a final exercise, let's create a third model that is the same as the first model we created earlier. The only difference is that we will train it on our raw dataset, not the normalized version. This way, we can see how much of a difference normalizing our input data makes.

Create, compile, and fit a model in the cell below. The only change in parameters will be using `data` instead of `scaled_data` during the `.fit()` step.

[32]:
```
model_3 = Sequential()
model_3.add(Dense(10, activation = "tanh", input_shape = (30,)))
model_3.add(Dense(1 , activation = "sigmoid"))
```

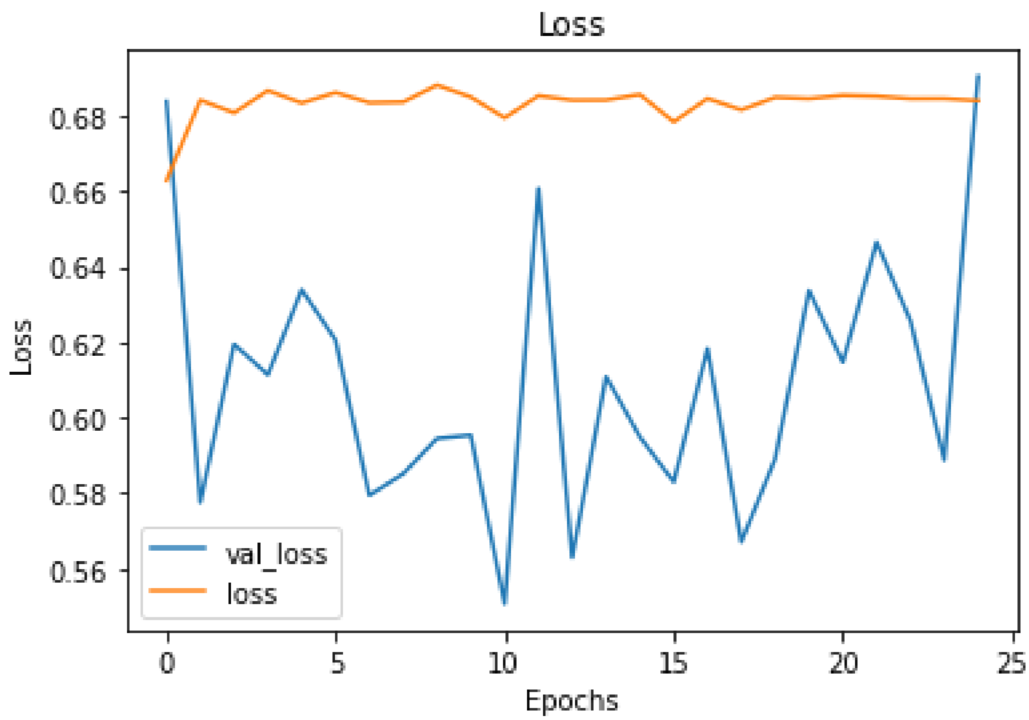
[33]:
```
model_3.compile(loss = "binary_crossentropy",
                optimizer = "sgd",
                metrics = ["acc"] )
```

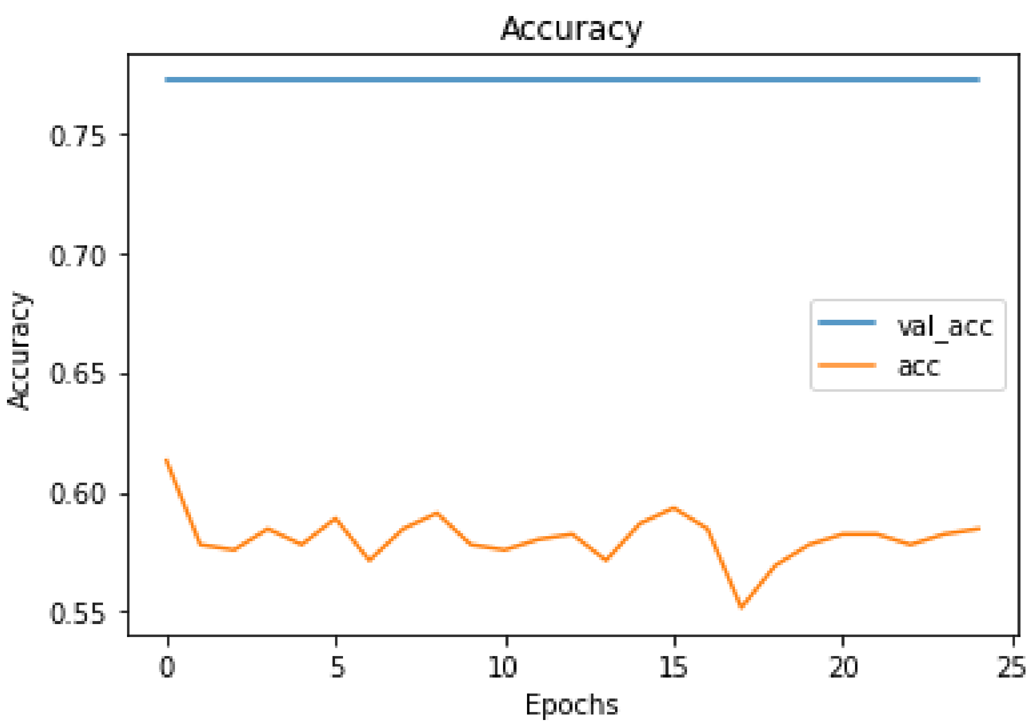
[34]:
```python
results_3 = model_3.fit(data, labels, epochs = 25,
                        batch_size = 1, validation_split = 0.2);
```

```
Epoch 1/25
455/455 [==============================] - 0s 907us/step - loss: 0.6629 - acc:
0.6132 - val_loss: 0.6839 - val_acc: 0.7719
Epoch 2/25
455/455 [==============================] - 0s 740us/step - loss: 0.6843 - acc:
0.5780 - val_loss: 0.5776 - val_acc: 0.7719
Epoch 3/25
455/455 [==============================] - 0s 745us/step - loss: 0.6809 - acc:
0.5758 - val_loss: 0.6195 - val_acc: 0.7719
Epoch 4/25
455/455 [==============================] - 0s 723us/step - loss: 0.6868 - acc:
0.5846 - val_loss: 0.6114 - val_acc: 0.7719
Epoch 5/25
455/455 [==============================] - 0s 718us/step - loss: 0.6835 - acc:
0.5780 - val_loss: 0.6340 - val_acc: 0.7719
Epoch 6/25
455/455 [==============================] - 0s 723us/step - loss: 0.6863 - acc:
0.5890 - val_loss: 0.6206 - val_acc: 0.7719
Epoch 7/25
455/455 [==============================] - 0s 735us/step - loss: 0.6836 - acc:
0.5714 - val_loss: 0.5794 - val_acc: 0.7719
Epoch 8/25
455/455 [==============================] - 0s 718us/step - loss: 0.6838 - acc:
0.5846 - val_loss: 0.5852 - val_acc: 0.7719
Epoch 9/25
455/455 [==============================] - 0s 737us/step - loss: 0.6882 - acc:
0.5912 - val_loss: 0.5945 - val_acc: 0.7719
Epoch 10/25
455/455 [==============================] - 0s 733us/step - loss: 0.6851 - acc:
0.5780 - val_loss: 0.5954 - val_acc: 0.7719
Epoch 11/25
455/455 [==============================] - 0s 715us/step - loss: 0.6796 - acc:
0.5758 - val_loss: 0.5507 - val_acc: 0.7719
Epoch 12/25
455/455 [==============================] - 0s 732us/step - loss: 0.6854 - acc:
0.5802 - val_loss: 0.6608 - val_acc: 0.7719
Epoch 13/25
455/455 [==============================] - 0s 748us/step - loss: 0.6843 - acc:
0.5824 - val_loss: 0.5630 - val_acc: 0.7719
Epoch 14/25
455/455 [==============================] - 0s 780us/step - loss: 0.6843 - acc:
0.5714 - val_loss: 0.6109 - val_acc: 0.7719
Epoch 15/25
455/455 [==============================] - 0s 774us/step - loss: 0.6858 - acc:
```

```
0.5868 - val_loss: 0.5949 - val_acc: 0.7719
Epoch 16/25
455/455 [==============================] - 0s 885us/step - loss: 0.6785 - acc:
0.5934 - val_loss: 0.5830 - val_acc: 0.7719
Epoch 17/25
455/455 [==============================] - 0s 723us/step - loss: 0.6847 - acc:
0.5846 - val_loss: 0.6183 - val_acc: 0.7719
Epoch 18/25
455/455 [==============================] - 0s 782us/step - loss: 0.6817 - acc:
0.5516 - val_loss: 0.5671 - val_acc: 0.7719
Epoch 19/25
455/455 [==============================] - 0s 724us/step - loss: 0.6850 - acc:
0.5692 - val_loss: 0.5892 - val_acc: 0.7719
Epoch 20/25
455/455 [==============================] - 0s 749us/step - loss: 0.6847 - acc:
0.5780 - val_loss: 0.6337 - val_acc: 0.7719
Epoch 21/25
455/455 [==============================] - 0s 726us/step - loss: 0.6855 - acc:
0.5824 - val_loss: 0.6148 - val_acc: 0.7719
Epoch 22/25
455/455 [==============================] - 0s 717us/step - loss: 0.6853 - acc:
0.5824 - val_loss: 0.6466 - val_acc: 0.7719
Epoch 23/25
455/455 [==============================] - 0s 760us/step - loss: 0.6846 - acc:
0.5780 - val_loss: 0.6258 - val_acc: 0.7719
Epoch 24/25
455/455 [==============================] - 0s 724us/step - loss: 0.6847 - acc:
0.5824 - val_loss: 0.5888 - val_acc: 0.7719
Epoch 25/25
455/455 [==============================] - 0s 739us/step - loss: 0.6841 - acc:
0.5846 - val_loss: 0.6905 - val_acc: 0.7719
```

[35]:
```
visualize_training_results(results_3)
```


Loss
0.68
0.66
0.64
0.62
0.60
0.58
0.56
Loss
val_loss
loss
0
5
10
15
20
25
Epochs


Accuracy
0.75
0.70
0.65
0.60
0.55
Accuracy
val_acc
acc
0
5
10
15
20
25
Epochs

Wow! Our results were much worse – over 20% poorer performance when working with non-normalized input data!

1.11 Summary

In this lab, we got some practice creating ***Multi-Layer Perceptrons***, and explored how things like the number of layers in a model and data normalization affect our overall training results!