index

May 10, 2022

1 Image Classification with MLPs - Lab

1.1 Introduction

For the final lab in this section, we'll build a more advanced ***Multi-Layer Perceptron*** to solve image classification for a classic dataset, MNIST! This dataset consists of thousands of labeled images of handwritten digits, and it has a special place in the history of Deep Learning.

1.2 Objectives

- Build a multi-layer neural network image classifier using Keras

1.3 Packages

First, let's import all the classes and packages you'll need for this lab.

[1]:
```python
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 keras.datasets import mnist
import os
os.environ['KMP_DUPLICATE_LIB_OK']='True' #This prevents kernel shut down due␣
↪to xgboost conflict
```

1.4 Data

Before we get into building the model, let's load our data and take a look at a sample image and label.

The MNIST dataset is often used for benchmarking model performance in the world of AI/Deep Learning research. Because it's commonly used, Keras actually includes a helper function to load the data and labels from MNIST – it even loads the data in a format already split into training and test sets!

Run the cell below to load the MNIST dataset. Note that if this is the first time you are working with MNIST through Keras, this will take a few minutes while Keras downloads the data.

```
[2]: (X_train, y_train), (X_test, y_test) = mnist.load_data()
```

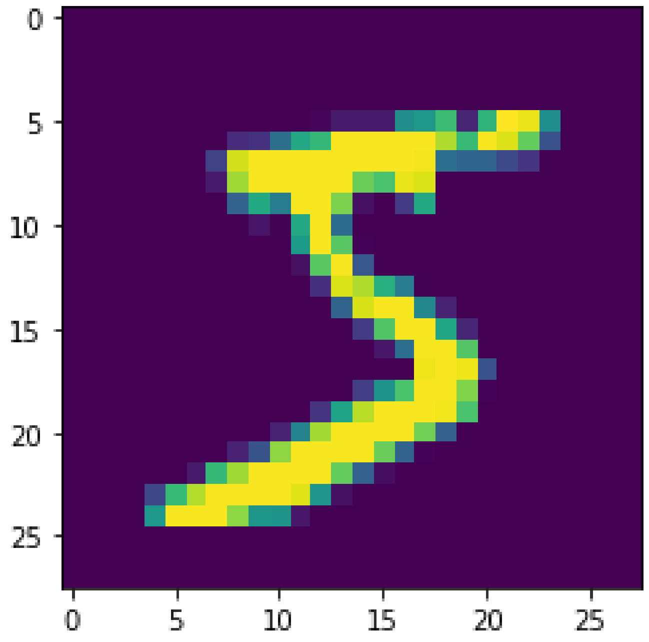
Great!

Now, let's quickly take a look at an image from the MNIST dataset – we can visualize it using Matplotlib. Run the cell below to visualize the first image and its corresponding label.

```
[7]: sample_image = X_train[0]
     sample_label = y_train[0]
     display(plt.imshow(sample_image));
     print('Label: {}'.format(sample_label));
```

```
<matplotlib.image.AxesImage at 0x7fd49edbfc10>
```

```
Label: 5
```



Great! That was easy. Now, we'll see that preprocessing image data has a few extra steps in order to get it into a shape where an MLP can work with it.

1.5 Preprocessing Images For Use With MLPs

By definition, images are matrices – they are a spreadsheet of pixel values between 0 and 255. We can see this easily enough by just looking at a raw image:

```
[8]: sample_image
```

```
[8]: array([[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
              0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
    0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
    0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
    0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
    0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
    0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
    0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
    0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
    0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
    0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   3,
   18,  18,  18, 126, 136, 175,  26, 166, 255, 247, 127,   0,   0,
    0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,  30,  36,  94, 154, 170,
  253, 253, 253, 253, 253, 225, 172, 253, 242, 195,  64,   0,   0,
    0,   0],
[  0,   0,   0,   0,   0,   0,   0,  49, 238, 253, 253, 253, 253,
  253, 253, 253, 253, 251,  93,  82,  82,  56,  39,   0,   0,   0,
    0,   0],
[  0,   0,   0,   0,   0,   0,   0,  18, 219, 253, 253, 253, 253,
  253, 198, 182, 247, 241,   0,   0,   0,   0,   0,   0,   0,   0,
    0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,  80, 156, 107, 253, 253,
  205,  11,   0,  43, 154,   0,   0,   0,   0,   0,   0,   0,   0,
    0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,   0,  14,   1, 154, 253,
   90,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
    0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0, 139, 253,
  190,   2,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
    0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  11, 190,
  253,  70,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
    0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  35,
  241, 225, 160, 108,   1,   0,   0,   0,   0,   0,   0,   0,   0,
    0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   81, 240, 253, 253, 119,  25,   0,   0,   0,   0,   0,   0,   0,
    0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
    0,  45, 186, 253, 253, 150,  27,   0,   0,   0,   0,   0,   0,
    0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,  16,  93, 252, 253, 187,   0,   0,   0,   0,   0,   0,
         0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0, 249, 253, 249,  64,   0,   0,   0,   0,   0,
         0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,  46, 130, 183, 253, 253, 207,   2,   0,   0,   0,   0,   0,
         0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  39,
       148, 229, 253, 253, 253, 250, 182,   0,   0,   0,   0,   0,   0,
         0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  24, 114, 221,
       253, 253, 253, 253, 201,  78,   0,   0,   0,   0,   0,   0,   0,
         0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,  23,  66, 213, 253, 253,
       253, 253, 198,  81,   2,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0],
       [  0,   0,   0,   0,   0,   0,  18, 171, 219, 253, 253, 253, 253,
       195,  80,   9,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0],
       [  0,   0,   0,   0,  55, 172, 226, 253, 253, 253, 253, 244, 133,
        11,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0],
       [  0,   0,   0,   0, 136, 253, 253, 253, 212, 135, 132,  16,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0]], dtype=uint8)
```

This is a problem in its current format, because MLPs take their input as vectors, not matrices or tensors. If all of the images were different sizes, then we would have a more significant problem on our hands, because we'd have challenges getting each image reshaped into a vector the exact same size as our input layer. However, this isn't a problem with MNIST, because all images are black white 28x28 pixel images. This means that we can just concatenate each row (or column) into a single 784-dimensional vector! Since each image will be concatenated in the exact same way, positional information is still preserved (e.g. the pixel value for the second pixel in the second row of an image will always be element number 29 in the vector).

Let's get started. In the cell below, print the `.shape` of both `X_train` and `X_test`

[11]:
```
print(X_train.shape)
print(X_test.shape)
```

```
(60000, 28, 28)
(10000, 28, 28)
```

We can interpret these numbers as saying "`X_train` consists of 60,000 images that are 28x28". We'll need to reshape them from `(28, 28)`, a 28x28 matrix, to `(784,)`, a 784-element vector. However, we need to make sure that the first number in our reshape call for both `X_train` and `X_test` still correspond to the number of observations we have in each.

In the cell below:

- Use the `.reshape()` method to reshape `X_train`. The first parameter should be `60000`, and the second parameter should be `784`
- Similarly, reshape `X_test` to `10000` and `784`

- Also, chain both `.reshape()` calls with an `.astype('float32')`, so that we convert our data from type `uint8` to `float32`

[12]:
```
X_train = X_train.reshape(X_train.shape[0], X_train.shape[1]**2).
  ↪astype('float32')
X_test  = X_test.reshape(X_test.shape[0], X_test.shape[1]**2).astype('float32')
```

Now, let's check the shape of our training and test data again to see if it worked.

[13]:
```
print(X_train.shape)
print(X_test.shape)
```

```
(60000, 784)
(10000, 784)
```

Great! Now, we just need to normalize our data!

1.6 Normalizing Image Data

Since all pixel values will always be between 0 and 255, we can just scale our data by dividing every element by 255! Run the cell below to do so now.

[14]:
```
X_train /= 255.
X_test /= 255.
```

Great! We've now finished preprocessing our image data. However, we still need to deal with our labels.

1.7 Preprocessing our Labels

Let's take a quick look at the first 10 labels in our training data:

[17]:
```
print(y_train)
print(y_test)
```

```
[5 0 4 … 5 6 8]
[7 2 1 … 4 5 6]
```

As we can see, the labels for each digit image in the training set are stored as the corresponding integer value – if the image is of a 5, then the corresponding label will be `5`. This means that this is a ***Multiclass Classification*** problem, which means that we need to ***One-Hot Encode*** our labels before we can use them for training.

Luckily, Keras provides a really easy utility function to handle this for us.

In the cell below:

- Use the function `to_categorical()` to one-hot encode our labels. This function can be found in the `keras.utils` sub-module. Pass in the following parameters:
  - The object we want to one-hot encode, which will be `y_train`/`y_test`
  - The number of classes contained in the labels, `10`

[18]:
```
y_train = keras.utils.to_categorical(y_train)
y_test  = keras.utils.to_categorical(y_test)
```

Great. Now, let's examine the label for the first data point, which we saw was `5` before.

[21]:
```
y_train[0]
```

[21]:
```
array([0., 0., 0., 0., 0., 1., 0., 0., 0., 0.], dtype=float32)
```

Perfect! As we can see, the fifth index is set to `1`, while everything else is set to `0`. That was easy! Now, let's get to the fun part – building our model!

1.8 Building our Model

For the remainder of this lab, we won't hold your hand as much – flex your newfound Keras muscles and build an MLP with the following specifications:

- A `Dense` hidden layer with `64` neurons, and a `'tanh'` activation function. Also, since this is the first hidden layer, be sure to pass in `input_shape=(784,)` in order to create a correctly-sized input layer!
- Since this is a multiclass classification problem, our output layer will need to be a `Dense` layer where the number of neurons is the same as the number of classes in the labels. Also, be sure to set the activation function to `'softmax'`

[22]:
```
model_1  = Sequential()
model_1.add(Dense(64, activation = "tanh", input_shape = (784,)))
model_1.add(Dense(y_train.shape[1], activation = "softmax"))
```

Now, compile your model with the following parameters:

- `loss='categorical_crossentropy'`
- `optimizer='sgd'`
- `metrics = ['acc']`

[23]:
```
model_1.compile(loss = 'categorical_crossentropy',
                optimizer='sgd', metrics = ['acc'])
```

Let's quickly inspect the shape of our model before training it and see how many training parameters we have. In the cell below, call the model's `.summary()` method.

[25]:
```
model_1.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 64)                50240
_________________________________________________________________
dense_1 (Dense)              (None, 10)                650
=================================================================
Total params: 50,890
Trainable params: 50,890
Non-trainable params: 0
_________________________________________________________________
```

50,890 trainable parameters! Note that while this may seem large, deep neural networks in production may have hundreds or thousands of layers and many millions of trainable parameters!

Let's get on to training. In the cell below, fit the model. Use the following parameters:

- Our training data and labels
- `epochs=5`
- `batch_size=64`
- `validation_data=(X_test, y_test)`

[27]:
```
results_1 = model_1.fit(X_train, y_train, epochs = 5,
                        batch_size = 64, validation_data = (X_test, y_test))
```

```
Epoch 1/5
938/938 [==============================] - 1s 1ms/step - loss: 0.8867 - acc:
0.7851 - val_loss: 0.5049 - val_acc: 0.8799
Epoch 2/5
938/938 [==============================] - 1s 984us/step - loss: 0.4552 - acc:
0.8839 - val_loss: 0.3919 - val_acc: 0.8998
Epoch 3/5
938/938 [==============================] - 1s 942us/step - loss: 0.3827 - acc:
0.8970 - val_loss: 0.3476 - val_acc: 0.9065
Epoch 4/5
938/938 [==============================] - 1s 966us/step - loss: 0.3473 - acc:
0.9044 - val_loss: 0.3218 - val_acc: 0.9128
Epoch 5/5
938/938 [==============================] - 1s 916us/step - loss: 0.3246 - acc:
0.9095 - val_loss: 0.3046 - val_acc: 0.9152
```

1.9 Visualizing our Loss and Accuracy Curves

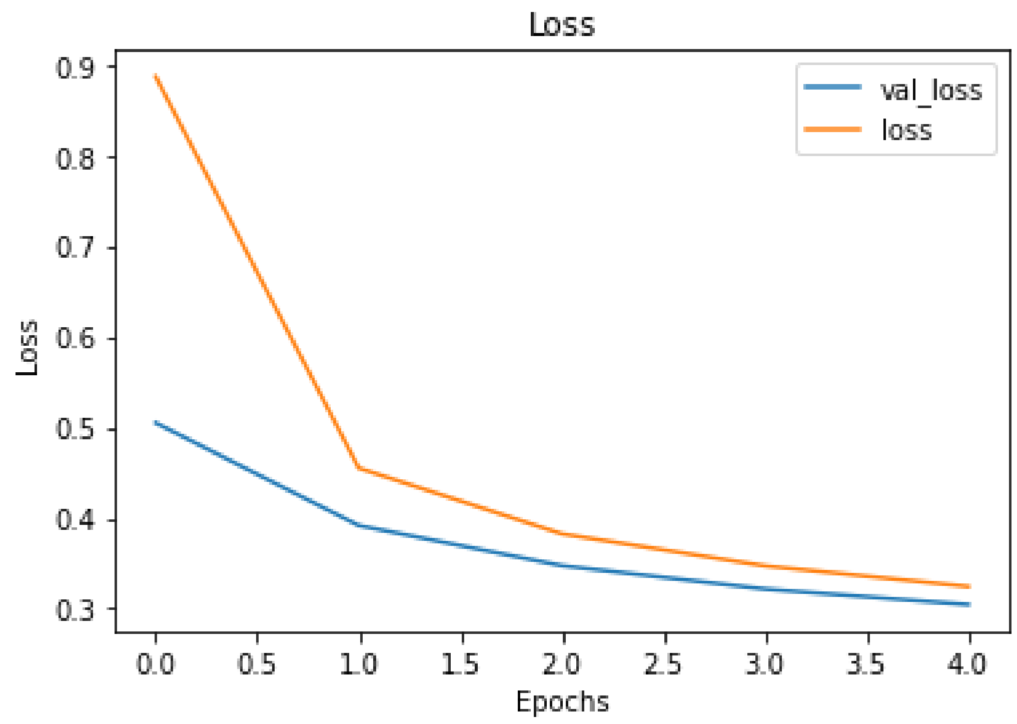
Now, let's inspect the model's performance and see if we detect any overfitting or other issues. In the cell below, create two plots:

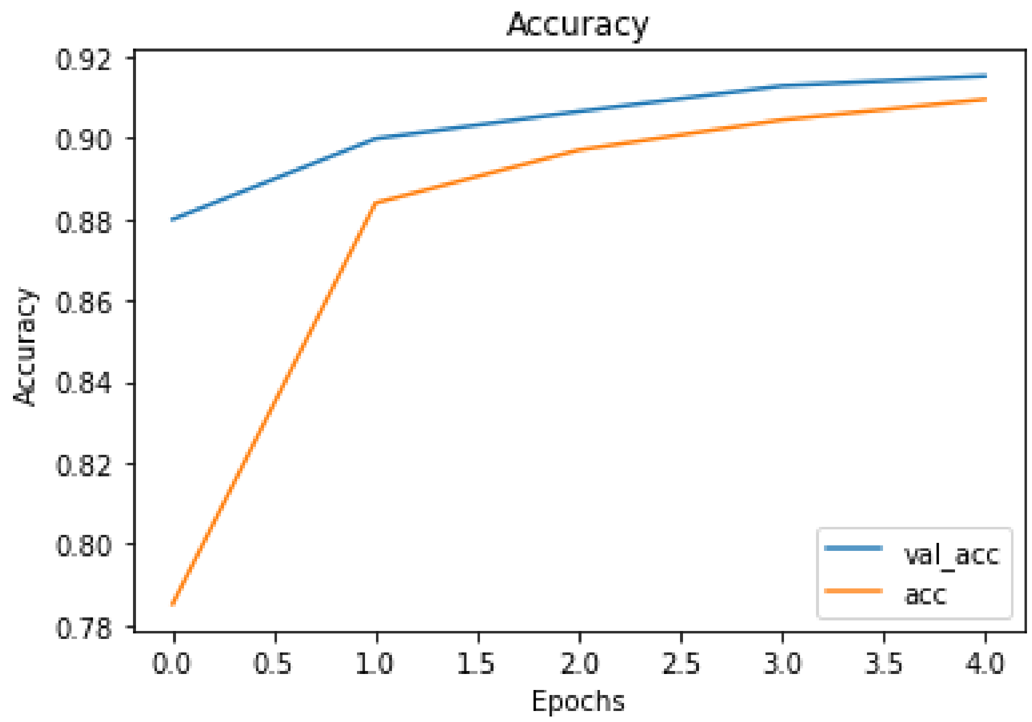
- The `loss` and `val_loss` over the training epochs
- The `acc` and `val_acc` over the training epochs

HINT: Consider copying over the visualization function from the previous lab in order to save time!

[28]:
```python
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()
```

[29]:
```python
visualize_training_results(results_1)
```


Loss
Loss
Epochs
val_loss
loss
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0


Accuracy
Accuracy
Epochs
val_acc
acc
0.92
0.90
0.88
0.86
0.84
0.82
0.80
0.78
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

Pretty good! Note that since our validation scores are currently higher than our training scores, its extremely unlikely that our model is overfitting to the training data. This is a good sign – that means that we can probably trust the results that our model is ~91.7% accurate at classifying handwritten digits!

1.10 Building a Bigger Model

Now, let's add another hidden layer and see how this changes things. In the cells below, create a second model. This model should have the following architecture:

- Input layer and first hidden layer same as `model_1`
- Another `Dense` hidden layer, this time with `32` neurons and a `'tanh'` activation function
- An output layer same as `model_1`

[31]:
```
model_2 = Sequential()
model_2.add(Dense(64, activation = "tanh", input_shape = (784,)))
model_2.add(Dense(32, activation = "tanh"))
model_2.add(Dense(y_train.shape[1], activation = "softmax"))
```

Let's quickly inspect the `.summary()` of the model again, to see how many new trainable parameters this extra hidden layer has introduced.

[32]:
```
model_2.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_5 (Dense)              (None, 64)                50240
_________________________________________________________________
dense_6 (Dense)              (None, 32)                2080
_________________________________________________________________
dense_7 (Dense)              (None, 10)                330
=================================================================
Total params: 52,650
Trainable params: 52,650
Non-trainable params: 0
_________________________________________________________________
```

This model isn't much bigger, but the layout means that the 2080 parameters in the new hidden layer will be focused on higher layers of abstraction than the first hidden layer. Let's see how it compares after training.

In the cells below, compile and fit the model using the same parameters you did for `model_1`.

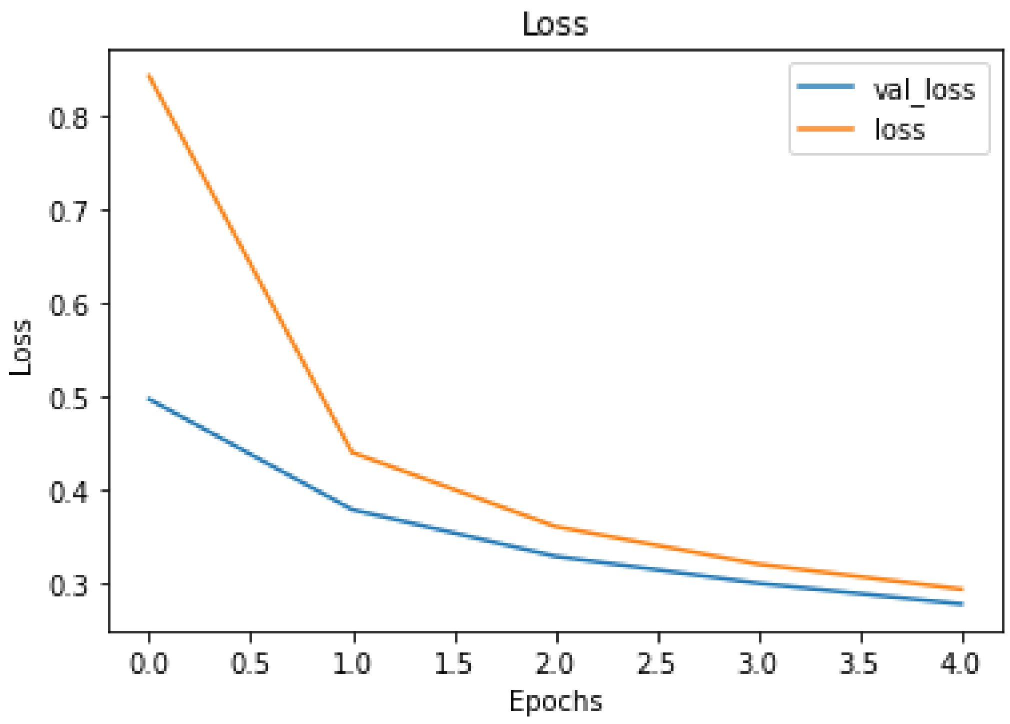
[37]:
```
model_2.compile(loss = 'categorical_crossentropy',
               optimizer='sgd', metrics = ['acc'])
```

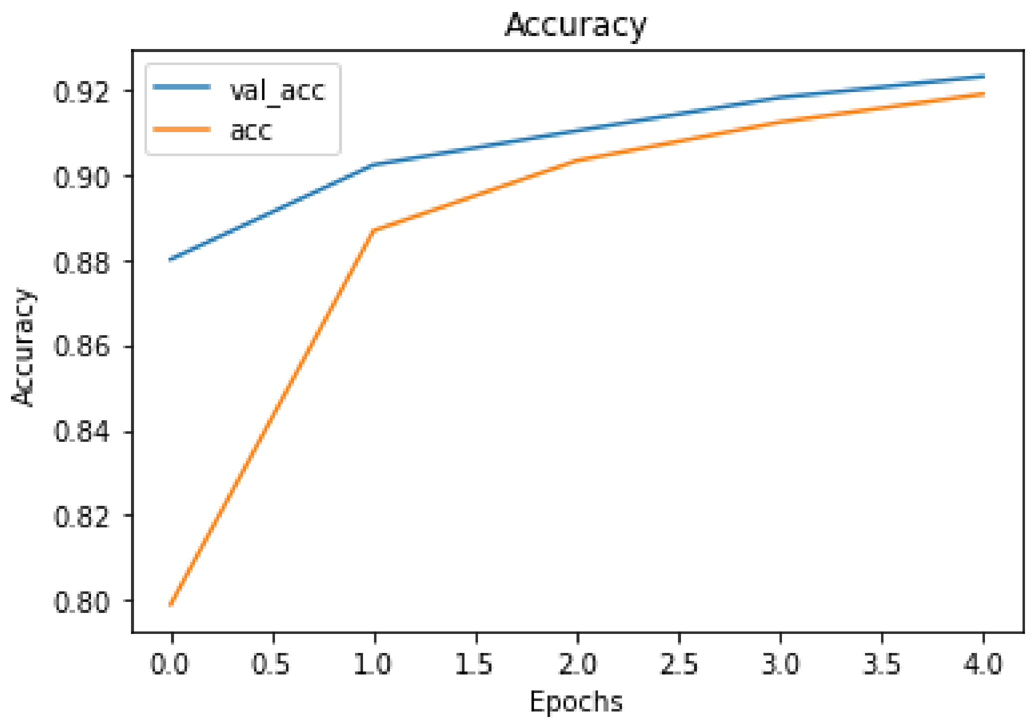
[38]:
```
results_2 = model_2.fit(X_train, y_train, epochs = 5, batch_size = 64,
                       validation_data = (X_test, y_test))
```

```
Epoch 1/5
938/938 [==============================] - 1s 1ms/step - loss: 0.8412 - acc:
0.7990 - val_loss: 0.4964 - val_acc: 0.8800
Epoch 2/5
938/938 [==============================] - 1s 1ms/step - loss: 0.4396 - acc:
0.8867 - val_loss: 0.3785 - val_acc: 0.9022
Epoch 3/5
938/938 [==============================] - 1s 1ms/step - loss: 0.3604 - acc:
0.9032 - val_loss: 0.3285 - val_acc: 0.9102
Epoch 4/5
938/938 [==============================] - 1s 1ms/step - loss: 0.3201 - acc:
0.9122 - val_loss: 0.2997 - val_acc: 0.9180
Epoch 5/5
938/938 [==============================] - 1s 1ms/step - loss: 0.2934 - acc:
0.9188 - val_loss: 0.2775 - val_acc: 0.9229
```

Now, visualize the plots again.

[39]:
```
visualize_training_results(results_2)
```





Slightly better validation accuracy, with no evidence of overfitting – great! If you run the model for more epochs, you'll see the model's performance continues to improve until the validation metrics plateau and the model begins to overfit to training data.

1.11 A Bit of Tuning

As a final exercise, let's see what happens to the model's performance if we switch activation functions from `'tanh'` to `'relu'`. In the cell below, recreate `model_2`, but replace all `'tanh'` activations with `'relu'`. Then, compile, train, and plot the results using the same parameters as the other two.

[40]:
```
model_3 = Sequential()
model_3.add(Dense(64, activation = "relu", input_shape = (784,) ))
model_3.add(Dense(32, activation = "relu"))
model_3.add(Dense(y_train.shape[1], activation = "softmax"))
```

[41]:
```
model_3.summary()
```

```
Model: "sequential_3"

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_8 (Dense)              (None, 64)                50240

_________________________________________________________________
dense_9 (Dense)              (None, 32)                2080
```

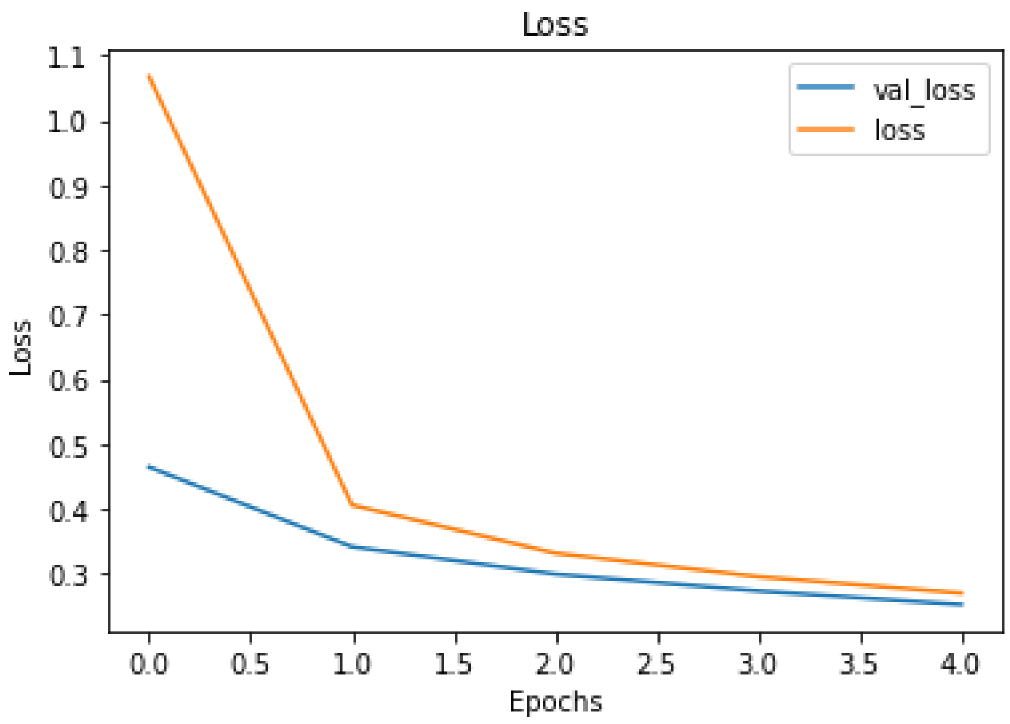
```
_________________________________________________________________
dense_10 (Dense)             (None, 10)                330
=================================================================
Total params: 52,650
Trainable params: 52,650
Non-trainable params: 0
_________________________________________________________________
```

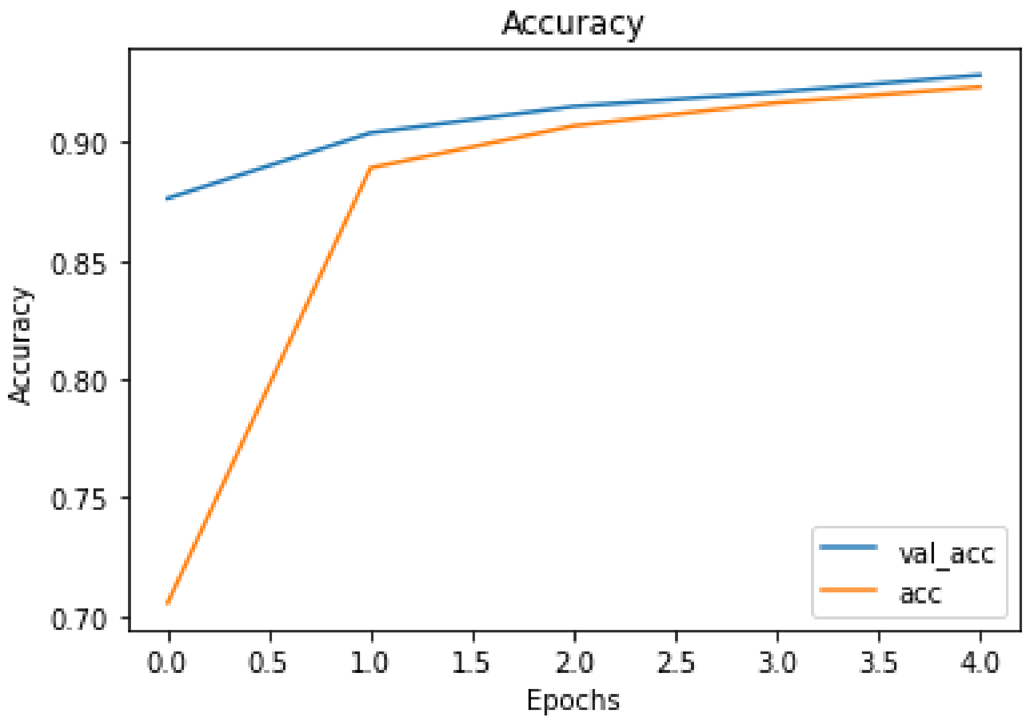
[42]:
```
model_3.compile(loss = 'categorical_crossentropy', optimizer = "sgd", metrics =␣
  ↪["acc"])
```

[43]:
```
results_3 = model_3.fit(X_train, y_train, epochs = 5, batch_size = 64,
                       validation_data = (X_test, y_test))
```

```
Epoch 1/5
938/938 [==============================] - 1s 1ms/step - loss: 1.0675 - acc:
0.7054 - val_loss: 0.4648 - val_acc: 0.8764
Epoch 2/5
938/938 [==============================] - 1s 1ms/step - loss: 0.4060 - acc:
0.8893 - val_loss: 0.3414 - val_acc: 0.9041
Epoch 3/5
938/938 [==============================] - 1s 991us/step - loss: 0.3316 - acc:
0.9071 - val_loss: 0.2992 - val_acc: 0.9153
Epoch 4/5
938/938 [==============================] - 1s 966us/step - loss: 0.2952 - acc:
0.9169 - val_loss: 0.2732 - val_acc: 0.9213
Epoch 5/5
938/938 [==============================] - 1s 1ms/step - loss: 0.2699 - acc:
0.9235 - val_loss: 0.2521 - val_acc: 0.9285
```

[44]:
```
visualize_training_results(results_3)
```


Loss
1.1
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
Loss
val_loss
loss
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
Epochs


Accuracy
0.90
0.85
0.80
0.75
0.70
Accuracy
val_acc
acc
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
Epochs

Performance improved even further! ReLU is one of the most commonly used activation functions around right now – it's especially useful in computer vision problems like image classification, as we've just seen.

1.12 Summary

In this lab, you once again practiced and reviewed the process of building a neural network. This time, you built a more complex network with additional layers which improved the performance of your model on the MNIST dataset!