This is a companion notebook for the book Deep Learning with Python, Second Edition. For readability, it only contains runnable code blocks and section titles, and omits everything else in the book: text paragraphs, figures, and pseudocode.

**If you want to be able to follow what's going on, I recommend reading the notebook side by side with your copy of the book.**

This notebook was generated for TensorFlow 2.6.

# Getting started with neural networks: Classification and regression

## Classifying movie reviews: A binary classification example

### The IMDB dataset

**Loading the IMDB dataset**

```
from tensorflow.keras.datasets import imdb
(train_data, train_labels), (test_data, test_labels) = imdb.load_data(
    num_words=10000)
```

```
train_data[0]
```

```
[1,
 14,
 22,
 16,
 43,
 530,
 973,
 1622,
 1385,
 65,
 458,
 4468,
 66,
 3941,
 4,
 173,
 36,
 256,
```

```
     5,
     25,
     100,
     43,
     838,
     112,
     50,
     670,
     2,
     9,
     35,
     480,
     284,
     5,
     150,
     4,
     172,
     112,
     167,
     2,
     336,
     385,
     39,
     4,
     172,
     4536,
     1111,
     17,
     546,
     38,
     13,
     447,
     4,
     192,
     50,
     16,
     6,
     147,
     2025,
     19.
```

```
train_labels[0]
```

```
1
```

```
max([max(sequence) for sequence in train_data])
```

```
9999
```

### Decoding reviews back to text

```
word_index = imdb.get_word_index()
reverse_word_index = dict(
```

```
    [(value, key) for (key, value) in word_index.items()])
decoded_review = " ".join(
    [reverse_word_index.get(i - 3, "?") for i in train_data[0]])
```

## Preparing the data

### Encoding the integer sequences via multi-hot encoding

```
import numpy as np
def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        for j in sequence:
            results[i, j] = 1.
    return results
x_train = vectorize_sequences(train_data)
x_test = vectorize_sequences(test_data)
```

```
x_train[0]
```

```
array([0., 1., 1., ..., 0., 0., 0.])
```

```
y_train = np.asarray(train_labels).astype("float32")
y_test = np.asarray(test_labels).astype("float32")
```

## Building your model

### Model definition

```
from tensorflow import keras
from tensorflow.keras import layers

model = keras.Sequential([
    layers.Dense(16, activation="relu"),
    layers.Dense(16, activation="relu"),
    layers.Dense(1, activation="sigmoid")
])
```

### Compiling the model

```
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
```

## Validating your approach

### Setting aside a validation set

```
x_val = x_train[:10000]
partial_x_train = x_train[10000:]
y_val = y_train[:10000]
partial_y_train = y_train[10000:]
```

### Training your model

```
history = model.fit(partial_x_train,
                    partial_y_train,
                    epochs=4,
                    batch_size=512,
                    validation_data=(x_val, y_val))
```

```
Epoch 1/4
30/30 ━━━━━━━━━━━━━━━━━━━━ 3s 67ms/step - accuracy: 0.6808 - loss: 0.6103 - val_accuracy
Epoch 2/4
30/30 ━━━━━━━━━━━━━━━━━━━━ 1s 47ms/step - accuracy: 0.8787 - loss: 0.3659 - val_accuracy
Epoch 3/4
30/30 ━━━━━━━━━━━━━━━━━━━━ 2s 37ms/step - accuracy: 0.9166 - loss: 0.2607 - val_accuracy
Epoch 4/4
30/30 ━━━━━━━━━━━━━━━━━━━━ 1s 34ms/step - accuracy: 0.9265 - loss: 0.2167 - val_accuracy
```

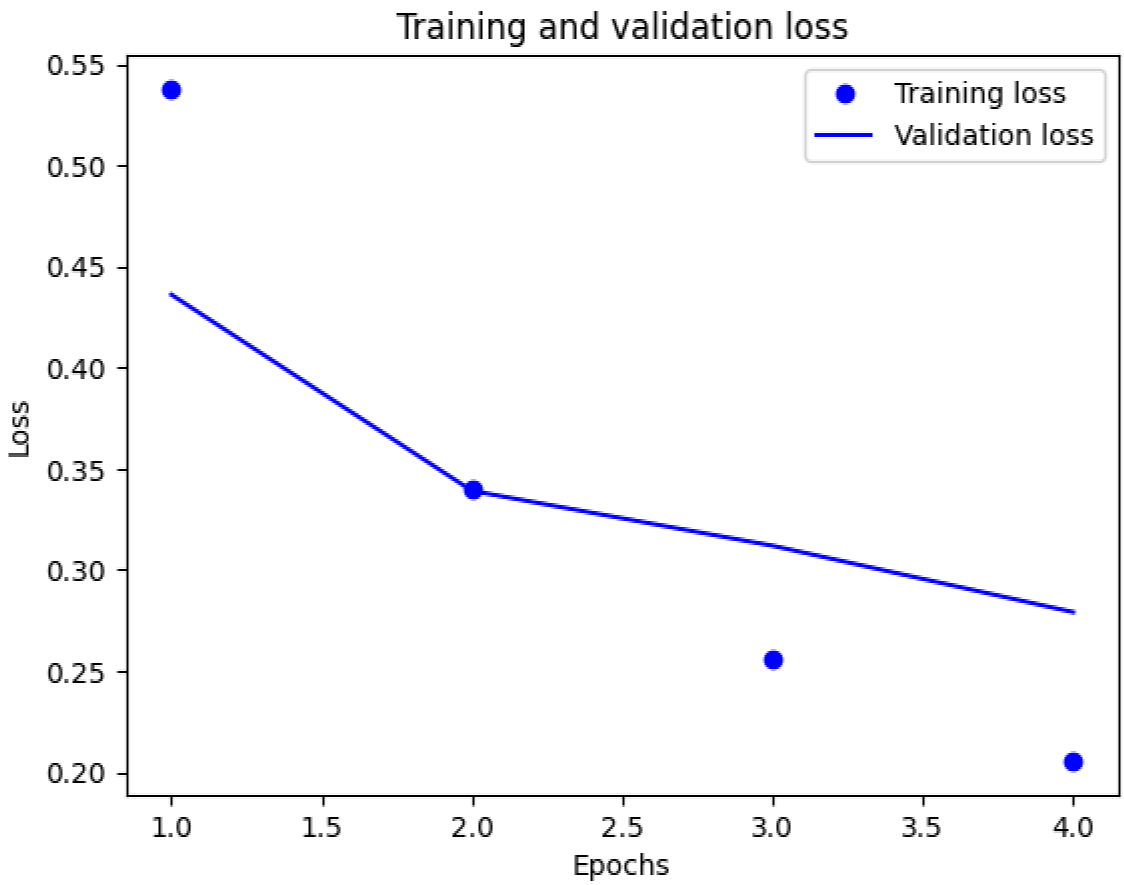
```
history_dict = history.history
history_dict.keys()
```

```
dict_keys(['accuracy', 'loss', 'val_accuracy', 'val_loss'])
```

### Plotting the training and validation loss

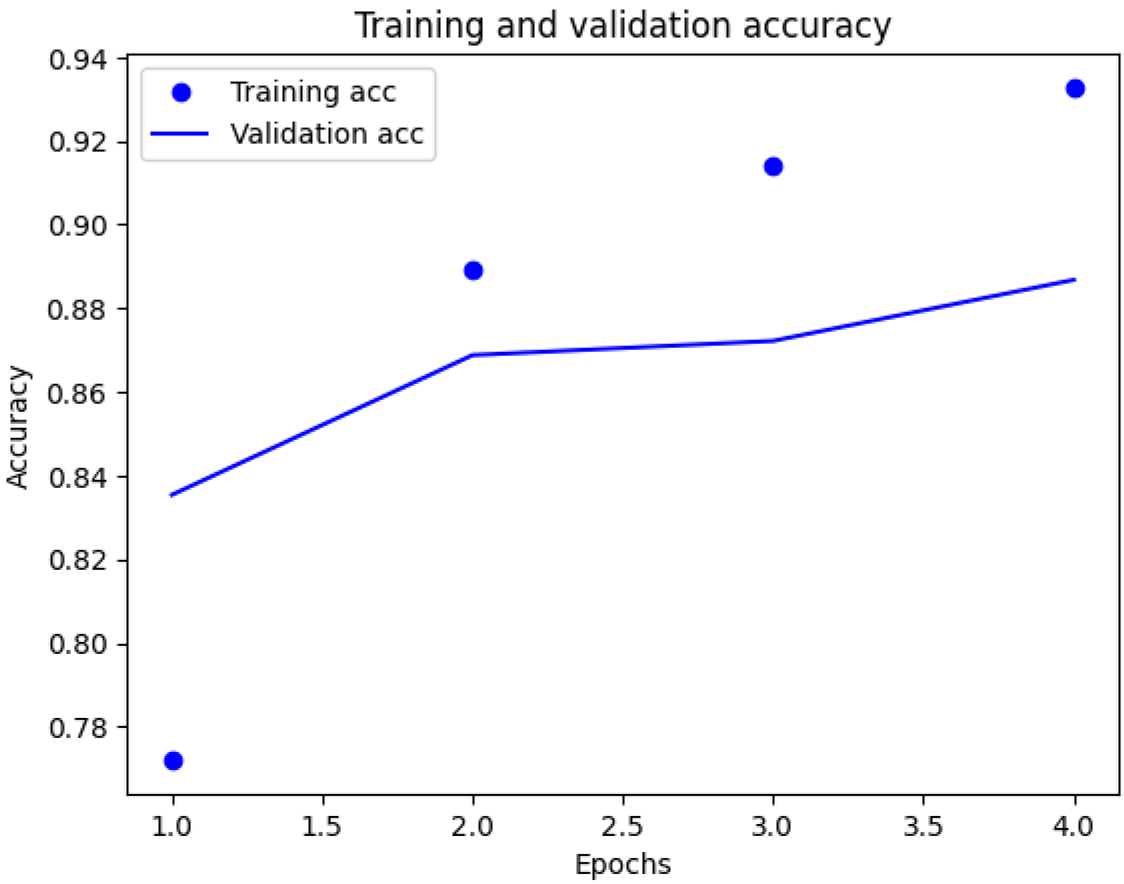
```
import matplotlib.pyplot as plt
history_dict = history.history
loss_values = history_dict["loss"]
val_loss_values = history_dict["val_loss"]
epochs = range(1, len(loss_values) + 1)
plt.plot(epochs, loss_values, "bo", label="Training loss")
plt.plot(epochs, val_loss_values, "b", label="Validation loss")
plt.title("Training and validation loss")
```

```
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.show()
```




### Plotting the training and validation accuracy

```
plt.clf()
acc = history_dict["accuracy"]
val_acc = history_dict["val_accuracy"]
plt.plot(epochs, acc, "bo", label="Training acc")
plt.plot(epochs, val_acc, "b", label="Validation acc")
plt.title("Training and validation accuracy")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend()
plt.show()
```




Training Results:

```
results = model.evaluate(x_test, y_test)
```

```
782/782 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.8782 - loss: 0.2941
```

Test Results

```
results

#creating a results table

results_history = {}
results_history["base_model"] = {
    "loss": results[0],
    "accuracy": results[1],
    "history": history.history
}
```

The training and test results from the base model are relatively close, at an accuracy of .86, and a loss of .32. For this 'baseline' model, I kept the 4 epochs as accuracy seemed to peak there and taper off after 4. In the next few iterations, I will use the suggestions from the assignment to see which changes affect the model for the better.

## Question 1: Add a hidden layer to the model to see effects on test and training accuracy.

```
model = keras.Sequential([
    layers.Dense(16, activation="relu"),
    layers.Dense(16, activation="relu"),
    layers.Dense(16, activation="relu"), # added hidden layer
    layers.Dense(1, activation="sigmoid")
])

model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])

x_val = x_train[:10000]
partial_x_train = x_train[10000:]
y_val = y_train[:10000]
partial_y_train = y_train[10000:]

history = model.fit(partial_x_train,
                    partial_y_train,
                    epochs=4,
                    batch_size=512,
                    validation_data=(x_val, y_val))


history_dict = history.history
history_dict.keys()
```

```
Epoch 1/4
30/30 ━━━━━━━━━━━━━━━━━━━━ 4s 83ms/step - accuracy: 0.6827 - loss: 0.6214 - val_accuracy
Epoch 2/4
30/30 ━━━━━━━━━━━━━━━━━━━━ 1s 36ms/step - accuracy: 0.8912 - loss: 0.3391 - val_accuracy
Epoch 3/4
30/30 ━━━━━━━━━━━━━━━━━━━━ 1s 38ms/step - accuracy: 0.9198 - loss: 0.2353 - val_accuracy
Epoch 4/4
30/30 ━━━━━━━━━━━━━━━━━━━━ 1s 36ms/step - accuracy: 0.9459 - loss: 0.1759 - val_accuracy
dict_keys(['accuracy', 'loss', 'val_accuracy', 'val_loss'])
```

Training Results:

```
results = model.evaluate(x_test, y_test)
```

```
782/782 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.8791 - loss: 0.2977
```

Test Results:

```
results

results_history["model_with_added_layer"] = {
    "loss": results[0],
    "accuracy": results[1],
    "history": history.history
}
```

By adding a hidden layer, the model gets marginally better, at .87 for accuracy, and .31 loss for the training data set, and .3 for the test data set. I don't know that adding the extra layer, time, and computational capacity is worth it for these slightly better results. Next, I'll change the number of hidden units in each layer.

## Question 2: More or fewer hidden units per layer.

```
model = keras.Sequential([
    layers.Dense(8, activation="relu"),
    layers.Dense(8, activation="relu"),
    layers.Dense(1, activation="sigmoid")
])

model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])

x_val = x_train[:10000]
partial_x_train = x_train[10000:]
y_val = y_train[:10000]
partial_y_train = y_train[10000:]

history = model.fit(partial_x_train,
                    partial_y_train,
                    epochs=4,
                    batch_size=512,
                    validation_data=(x_val, y_val))
```

```
history_dict = history.history
history_dict.keys()
```

```
Epoch 1/4
30/30 ━━━━━━━━━━━━━━━━━━━━ 4s 82ms/step - accuracy: 0.6769 - loss: 0.6307 - val_accuracy
Epoch 2/4
30/30 ━━━━━━━━━━━━━━━━━━━━ 1s 34ms/step - accuracy: 0.8871 - loss: 0.4084 - val_accuracy
Epoch 3/4
30/30 ━━━━━━━━━━━━━━━━━━━━ 1s 32ms/step - accuracy: 0.9169 - loss: 0.3047 - val_accuracy
Epoch 4/4
30/30 ━━━━━━━━━━━━━━━━━━━━ 1s 34ms/step - accuracy: 0.9329 - loss: 0.2362 - val_accuracy
dict_keys(['accuracy', 'loss', 'val_accuracy', 'val_loss'])
```

```
results = model.evaluate(x_test, y_test)
```

```
782/782 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.8831 - loss: 0.2998
```

```
results

results_history["model_with_changed_hidden_units"] = {
    "loss": results[0],
    "accuracy": results[1],
    "history": history.history
}
```

I landed on 8 hidden units per layer, because the results had the best balance of high accuracy and lower loss function vs 32, 64, or 128.

## Question 3: Using the MSE Loss function instead of binary_crossentropy

```
model = keras.Sequential([
    layers.Dense(16, activation="relu"),
    layers.Dense(16, activation="relu"),
    layers.Dense(1, activation="sigmoid")
])

model.compile(optimizer="rmsprop",
              loss="mse",
              metrics=["accuracy"])

x_val = x_train[:10000]
partial_x_train = x_train[10000:]
y_val = y_train[:10000]
partial_y_train = y_train[10000:]
```

```
history = model.fit(partial_x_train,
                    partial_y_train,
                    epochs=4,
                    batch_size=512,
                    validation_data=(x_val, y_val))
```

```
history_dict = history.history
history_dict.keys()
```

```
Epoch 1/4
30/30 ━━━━━━━━━━━━━━━━━━━━ 4s 78ms/step - accuracy: 0.6856 - loss: 0.2189 - val_accuracy
Epoch 2/4
30/30 ━━━━━━━━━━━━━━━━━━━━ 1s 36ms/step - accuracy: 0.8762 - loss: 0.1213 - val_accuracy
Epoch 3/4
30/30 ━━━━━━━━━━━━━━━━━━━━ 1s 37ms/step - accuracy: 0.9096 - loss: 0.0848 - val_accuracy
Epoch 4/4
30/30 ━━━━━━━━━━━━━━━━━━━━ 1s 36ms/step - accuracy: 0.9223 - loss: 0.0700 - val_accuracy
dict_keys(['accuracy', 'loss', 'val_accuracy', 'val_loss'])
```

```
results = model.evaluate(x_test, y_test)
```

```
782/782 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.8832 - loss: 0.0909
```

```
results

results_history["model_with_mse"] = {
    "loss": results[0],
    "accuracy": results[1],
    "history": history.history
}
```

## Question 4: Use tanh activation instead of relu.

```
model = keras.Sequential([
    layers.Dense(16, activation="tanh"),
    layers.Dense(16, activation="tanh"),
    layers.Dense(1, activation="sigmoid")
])

model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
```

```
x_val = x_train[:10000]
partial_x_train = x_train[10000:]
y_val = y_train[:10000]
partial_y_train = y_train[10000:]

history = model.fit(partial_x_train,
                    partial_y_train,
                    epochs=4,
                    batch_size=512,
                    validation_data=(x_val, y_val))


history_dict = history.history
history_dict.keys()
```

```
Epoch 1/4
30/30 ━━━━━━━━━━━━━━━━━━━━ 4s 93ms/step - accuracy: 0.7017 - loss: 0.5772 - val_accuracy
Epoch 2/4
30/30 ━━━━━━━━━━━━━━━━━━━━ 3s 37ms/step - accuracy: 0.8998 - loss: 0.3152 - val_accuracy
Epoch 3/4
30/30 ━━━━━━━━━━━━━━━━━━━━ 1s 33ms/step - accuracy: 0.9254 - loss: 0.2255 - val_accuracy
Epoch 4/4
30/30 ━━━━━━━━━━━━━━━━━━━━ 1s 36ms/step - accuracy: 0.9442 - loss: 0.1724 - val_accuracy
dict_keys(['accuracy', 'loss', 'val_accuracy', 'val_loss'])
```

```
results = model.evaluate(x_test, y_test)
```

```
782/782 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.8656 - loss: 0.3309
```

```
results

results_history["model_with_tanh"] = {
    "loss": results[0],
    "accuracy": results[1],
    "history": history.history
}
```

Using the tahn function instead of relu slightly improved accuracy and loss, but still had issues with overfitting to the training data.

## Question 5: Adding someting we learned in class. (I'm adding Dropout to reduce overfitting)

```
from tensorflow.keras.layers import Dense, Dropout

model = keras.Sequential([
    layers.Dense(16, activation="relu"),
    Dropout(0.3),
    layers.Dense(16, activation="relu"),
    Dropout(0.3),
    layers.Dense(1, activation="sigmoid")
])

model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])

x_val = x_train[:10000]
partial_x_train = x_train[10000:]
y_val = y_train[:10000]
partial_y_train = y_train[10000:]

history = model.fit(partial_x_train,
                    partial_y_train,
                    epochs=4,
                    batch_size=512,
                    validation_data=(x_val, y_val))


history_dict = history.history
history_dict.keys()
```

```
Epoch 1/4
30/30 ━━━━━━━━━━━━━━━━━━━━ 3s 66ms/step - accuracy: 0.6444 - loss: 0.6284 - val_accuracy
Epoch 2/4
30/30 ━━━━━━━━━━━━━━━━━━━━ 2s 37ms/step - accuracy: 0.8429 - loss: 0.4137 - val_accuracy
Epoch 3/4
30/30 ━━━━━━━━━━━━━━━━━━━━ 1s 37ms/step - accuracy: 0.8822 - loss: 0.3229 - val_accuracy
Epoch 4/4
30/30 ━━━━━━━━━━━━━━━━━━━━ 1s 38ms/step - accuracy: 0.9066 - loss: 0.2671 - val_accuracy
dict_keys(['accuracy', 'loss', 'val_accuracy', 'val_loss'])
```

```
results = model.evaluate(x_test, y_test)
```

```
782/782 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.8828 - loss: 0.2886
```

```
results
```

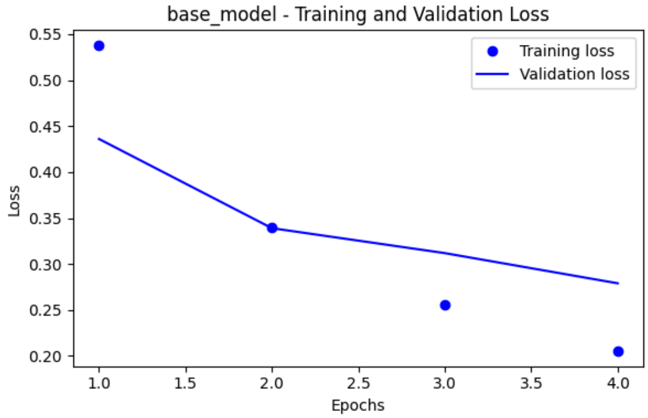
```
results_history["model_with_dropout"] = {
    "loss": results[0],
    "accuracy": results[1],
```

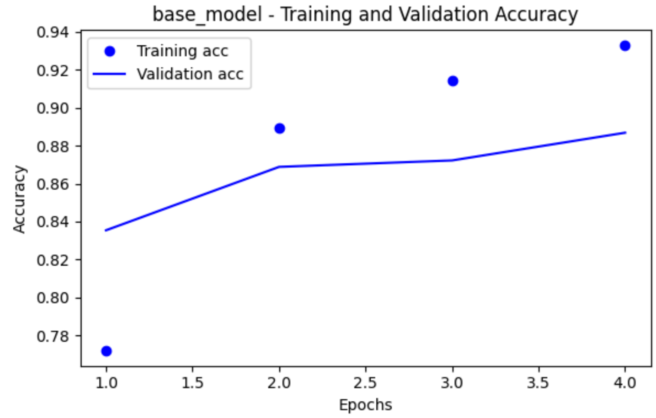
```
    "history": history.history
}
```

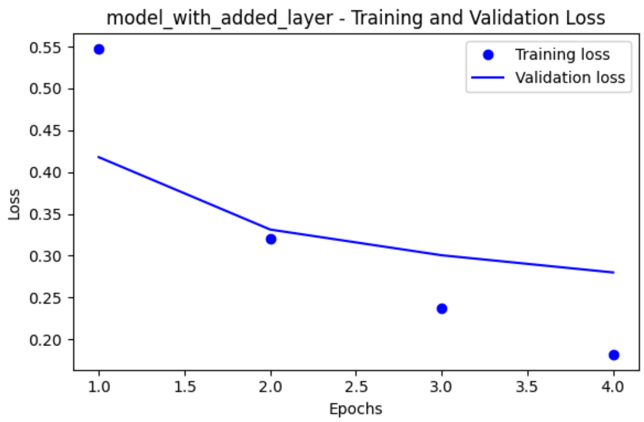
Adding the dropout didn't quite work how I expected. Although it did improve overfitting from previous versions of the model, it didn't improve the overall accuracy or loss function of the model.

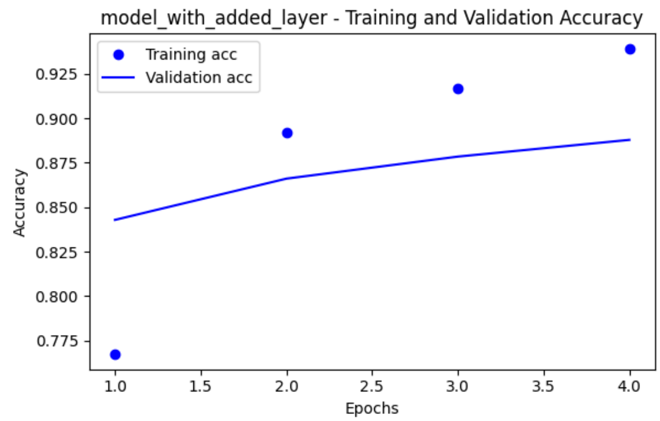
## Plotting the results in graphs as well as a table

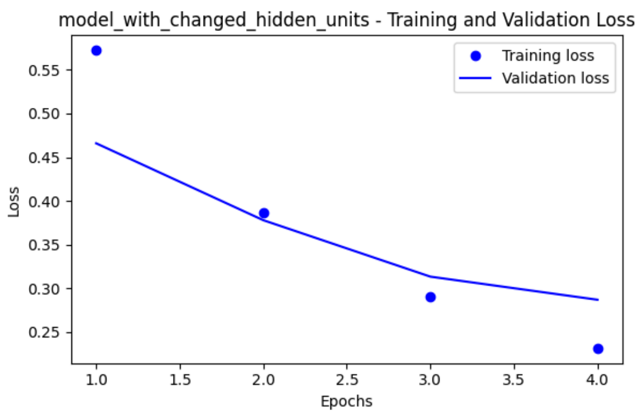
```
# Plot accuracy and loss for all the different models
def plot_model_results(model_name, history):
    history_dict = history
    loss_values = history_dict["loss"]
    val_loss_values = history_dict["val_loss"]
    epochs = range(1, len(loss_values) + 1)

    plt.figure(figsize=(12, 4))

    plt.subplot(1, 2, 1)
    plt.plot(epochs, loss_values, "bo", label="Training loss")
    plt.plot(epochs, val_loss_values, "b", label="Validation loss")
    plt.title(f"{model_name} - Training and Validation Loss")
    plt.xlabel("Epochs")
    plt.ylabel("Loss")
    plt.legend()

    plt.subplot(1, 2, 2)
    acc = history_dict["accuracy"]
    val_acc = history_dict["val_accuracy"]
    plt.plot(epochs, acc, "bo", label="Training acc")
    plt.plot(epochs, val_acc, "b", label="Validation acc")
    plt.title(f"{model_name} - Training and Validation Accuracy")
    plt.xlabel("Epochs")
    plt.ylabel("Accuracy")
    plt.legend()

    plt.tight_layout()
    plt.show()

# Plot results for each model
for model_name, results in results_history.items():
    plot_model_results(model_name, results['history'])
```

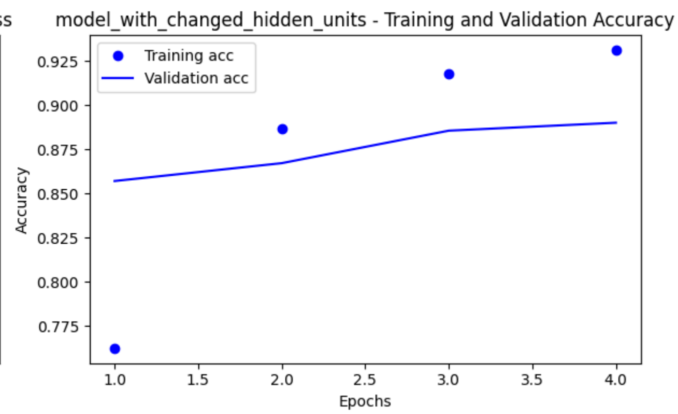

base_model - Training and Validation Loss
Training loss
Validation loss
Loss
Epochs

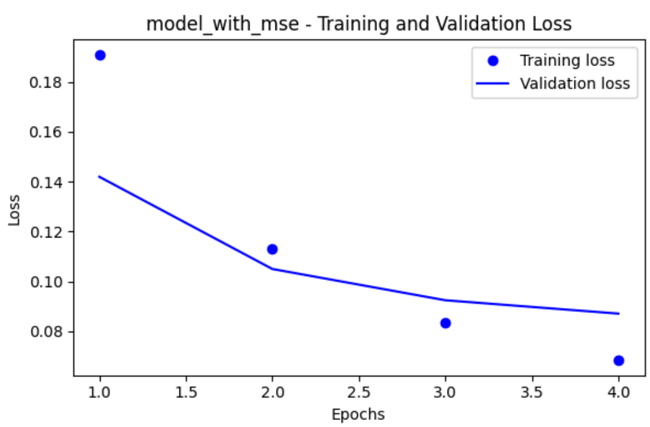

base_model - Training and Validation Accuracy
Training acc
Validation acc
Accuracy
Epochs

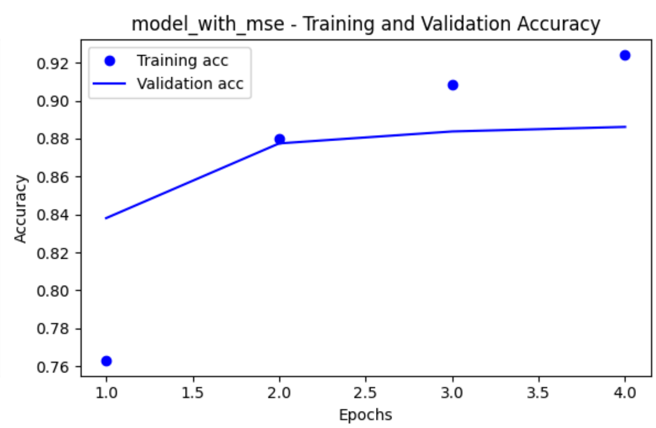

model_with_added_layer - Training and Validation Loss
Training loss
Validation loss
Loss
Epochs

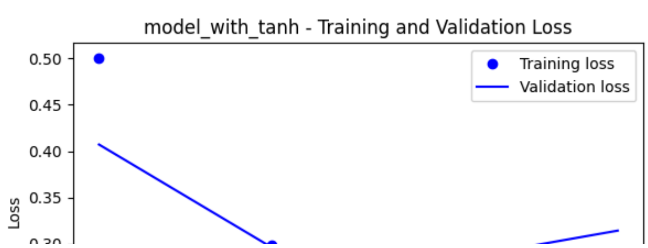

model_with_added_layer - Training and Validation Accuracy
Training acc
Validation acc
Accuracy
Epochs

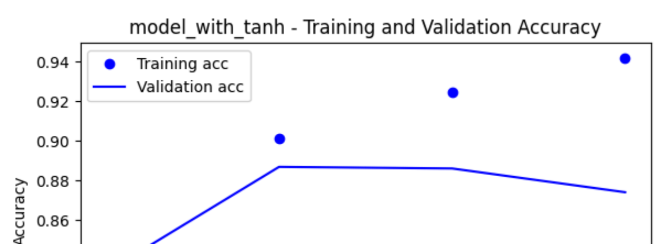

model_with_changed_hidden_units - Training and Validation Loss
Training loss
Validation loss
Loss
Epochs

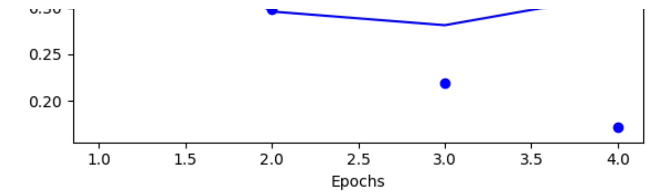

model_with_changed_hidden_units - Training and Validation Accuracy
Training acc
Validation acc
Accuracy
Epochs

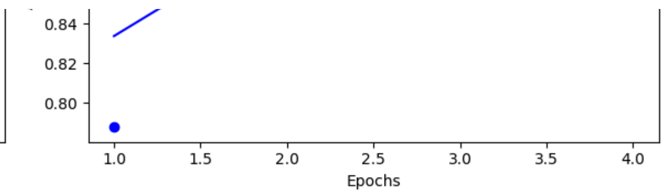

model_with_mse - Training and Validation Loss
Training loss
Validation loss
Loss
Epochs

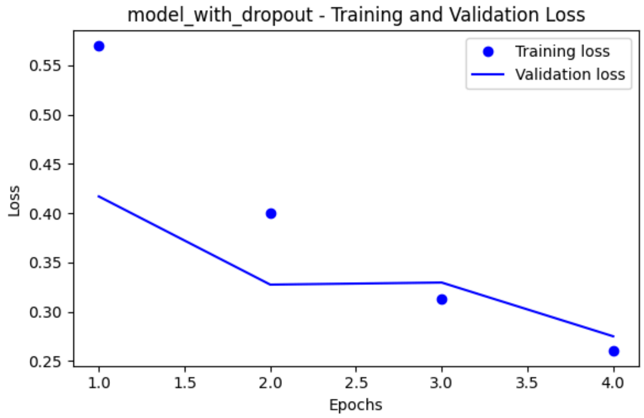

model_with_mse - Training and Validation Accuracy
Training acc
Validation acc
Accuracy
Epochs

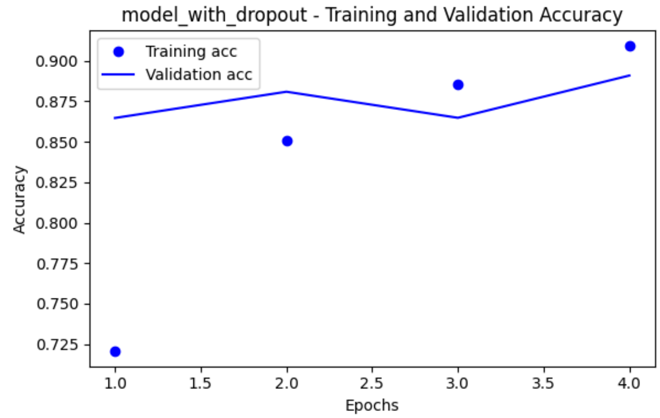

model_with_tanh - Training and Validation Loss
Training loss
Validation loss
Loss


model_with_tanh - Training and Validation Accuracy
Training acc
Validation acc
Accuracy


0.25
0.20
1.0 1.5 2.0 2.5 3.0 3.5 4.0
Epochs


0.84
0.82
0.80
1.0 1.5 2.0 2.5 3.0 3.5 4.0
Epochs


model_with_dropout - Training and Validation Loss
Training loss
Validation loss
Loss
Epochs


model_with_dropout - Training and Validation Accuracy
Training acc
Validation acc
Accuracy
Epochs

In conclusion, the model I would pick overall would be the model with MSE instead of binary_crossentropy as the loss function. This model improved accuracy over all other models, and had a much lower loss function in both the training and validation sets.

## Outputting a table of all results together

```
import pandas as pd

# Convert the dictionary to a DataFrame
results_df = pd.DataFrame.from_dict(results_history, orient="index")

# Print the DataFrame as a table
print(results_df)

from tabulate import tabulate

# Print the results using tabulate
print(tabulate(results_df, headers='keys', tablefmt='pretty'))

# Extract accuracy and loss for each model from the results_history
accuracy = []
loss = []
models = list(results_history.keys())

for model_name, results in results_history.items():
    history_dict = results['history']
    accuracy.append(history_dict['accuracy'][-1])  # Use the last accuracy value
    loss.append(history_dict['loss'][-1])  # Use the last loss value
```

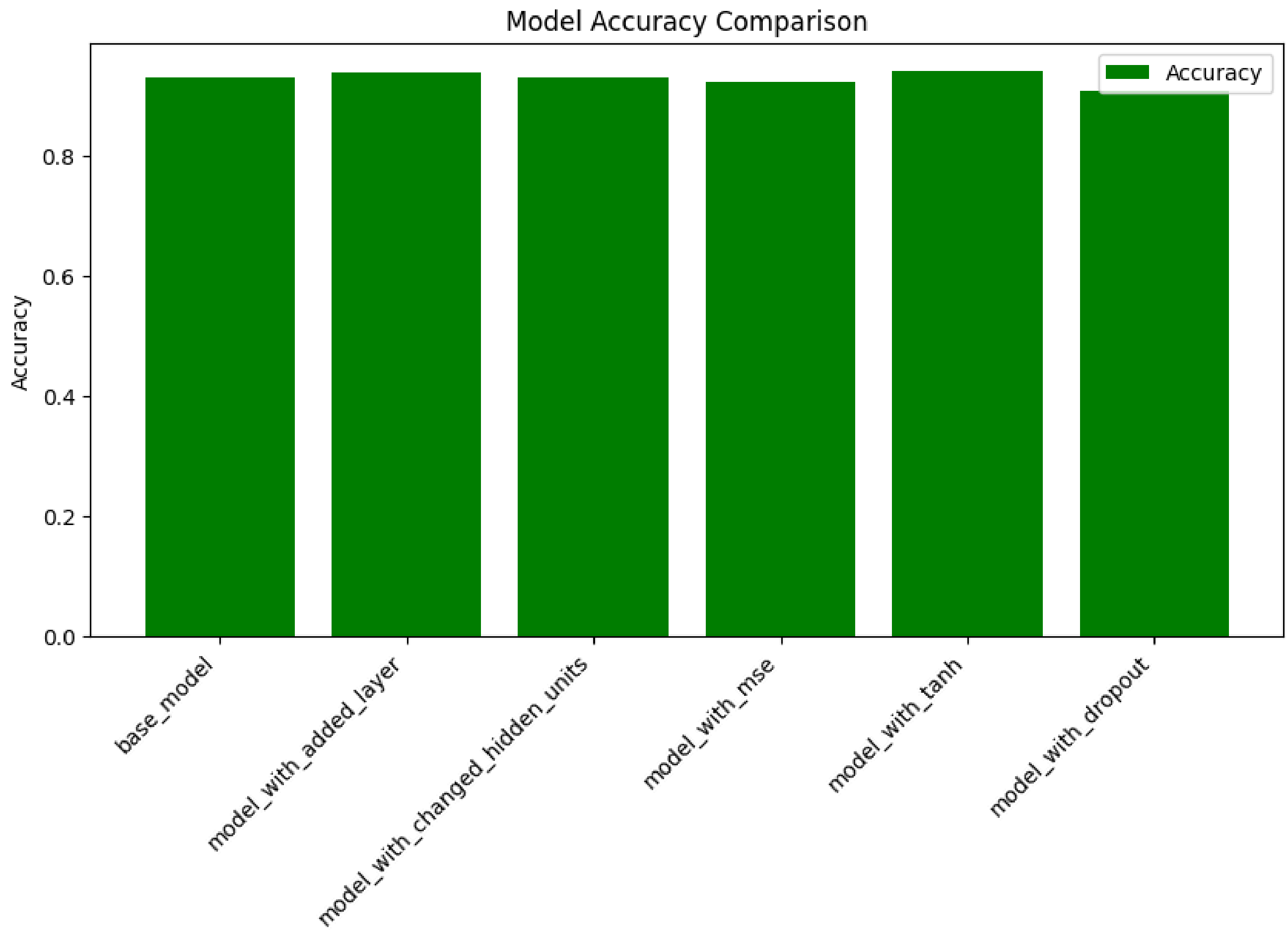
```
                                  loss  accuracy  \
base_model                    0.294816   0.87980
model_with_added_layer        0.297208   0.88068
model_with_changed_hidden_units 0.298604   0.88480
model_with_mse                0.090495   0.88300
model_with_tanh               0.326985   0.86728
model_with_dropout            0.287226   0.88420

                                                                            history
base_model                      {'accuracy': [0.7720666527748108, 0.8892666697...
model_with_added_layer          {'accuracy': [0.76746666431427, 0.891866683959...
model_with_changed_hidden_units {'accuracy': [0.7624666690826416, 0.8868666887...
model_with_mse                  {'accuracy': [0.7630666494369507, 0.8801333308...
model_with_tanh                 {'accuracy': [0.787933349609375, 0.90113335847...
model_with_dropout              {'accuracy': [0.72079998254776, 0.850933313369...
+---------------------------------+---------------------+---------------------+----------
|                                 |        loss         |      accuracy       |
+---------------------------------+---------------------+---------------------+----------
```

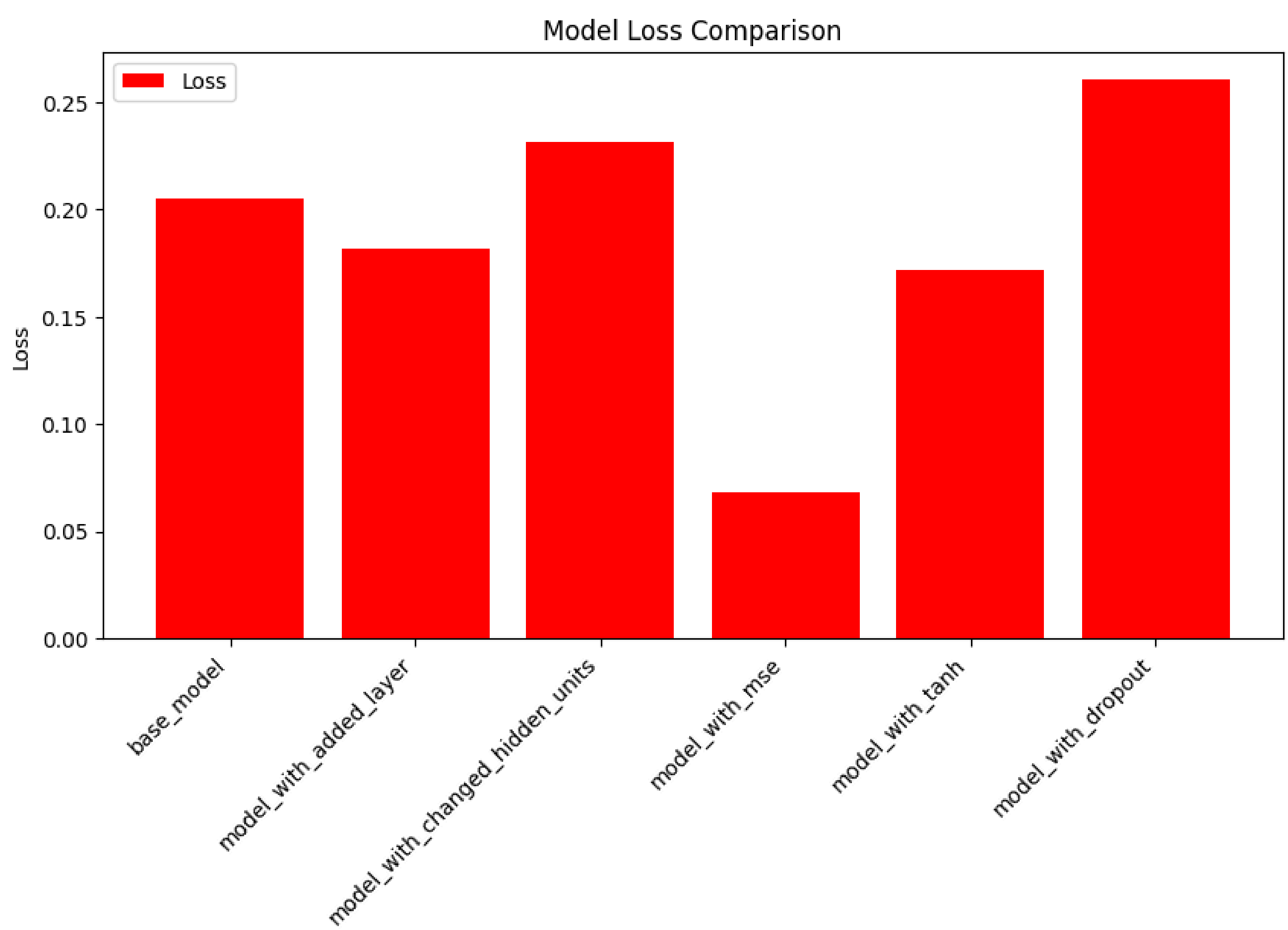
```
|           base_model            | 0.29481571912765503 | 0.879800021648407  |  {'accura
|     model_with_added_layer      | 0.2972082197666168  | 0.8806800246238708 |     {'acc
| model_with_changed_hidden_units | 0.29860374331474304 | 0.8848000168800354 |  {'accura
|         model_with_mse          | 0.09049452841281891 | 0.8830000162124634 | {'accurac
|         model_with_tanh         | 0.32698461413383484 | 0.8672800064086914 |    {'accu
|       model_with_dropout        | 0.28722602128982544 | 0.8841999769210815 |     {'acc
+---------------------------------+---------------------+--------------------+----------
```

In conclusion, the model I would pick overall would be the model with MSE instead of binary_crossentropy as the loss function. This model improved accuracy over all other models, and had a much lower loss function in both the training and validation sets.

## Plotting all results in charts to visually compare

```
# Plot accuracy comparison
plt.figure(figsize=(10, 5))
plt.bar(models, accuracy, color="green", label="Accuracy")
plt.ylabel("Accuracy")
plt.title("Model Accuracy Comparison")
plt.legend()
plt.xticks(rotation=45, ha="right")  # Rotate model names for better readability
plt.show()

# Plot loss comparison
plt.figure(figsize=(10, 5))
plt.bar(models, loss, color="red", label="Loss")
plt.ylabel("Loss")
plt.title("Model Loss Comparison")
plt.legend()
plt.xticks(rotation=45, ha="right")  # Rotate model names for better readability
plt.show()
```


Model Accuracy Comparison
Accuracy
Accuracy
0.0
0.2
0.4
0.6
0.8
base_model
model_with_added_layer
model_with_changed_hidden_units
model_with_mse
model_with_tanh
model_with_dropout


Model Loss Comparison
Loss
Loss
0.00
0.05
0.10
0.15
0.20
0.25
base_model
model_with_added_layer
model_with_changed_hidden_units
model_with_mse
model_with_tanh
model_with_dropout