Titanic Survival Prediction

Introduction

The following work is based on the **Kaggle** competition named "Titanc - Machine Learning from Disaster".

A minimal primer on the subject: The Titanic was a ship built to not sink. A novum. But things did not go according to plan, a disaster happend and 1,500 from the 2,224 passengers died. For more background please look at this Wikipedia article.

The idea of the **Kaggle** competion is to train a machine to predict if a passenger will survive the diaster on the Titanic or not using only a hand full of details.

This challenge has been running since 2012 and the leaderboard is full of people reaching 100% accuracy. As I'm at the beginning of my ML journey I do not aim that high so I'm not doomed to sink prematurely.

Many people have written excellent papers on their machine learning approach and data analysis and preparation. I conciously did not read any attempts that solved the problem with a neural network but I took ideas and code from the data preparation steps of others. Code from other sources is clearly marked in the particular cells.

An important remark about code generated by machines: All code that is not marked as sourced from another paper/tutrial is written by myself using my brain and my own fingers typing on the keyboard. I did not use any assisted programming technique.

Defining the Task

Frame the Problem

The problem at hand is to predict wether someone survived the disaster on the Titanic or not. This is a binary classification problem - survived or died - determined by features like social rank, age, etc. the algorithm needs to learn.

Since this happend a long time ago the whole dataset is known and it will not change. All predictions can be fully evaluated. The model trained for this will also only work for this particular incident and can not be generalized to boat accidents without further research (, modelling and training).

Collect the Dataset

In this case the collection is easy as the dataset comes readily available and already split in training and testing sets from the **Kaggle** competion site.

The real world scenario would be to unearth the archives to find all Titanic passengers, gather their details and cross reference that with surviver information.

Understand the Data

Conveniently for the competition to have comparable results the data is already split in a training and testing set. Both downloadable as CSV file.

To see what the dataset looks like the first step is to load the CSV files and perform some exploratory analysis.

In [1]:
```
import pandas as pd

# load train and test data
train_df = pd.read_csv('titanic_train.csv')
test_df = pd.read_csv('titanic_test.csv')

# print the first 30 entries of the training data set
train_df.head(30)
```

Out[1]:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ti
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 2
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON 310
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373
5	6	0	3	Moran, Mr. James	male	NaN	0	0	33
6	7	0	1	McCarthy, Mr. Timothy J	male	54.0	0	0	17
7	8	0	3	Palsson, Master. Gosta Leonard	male	2.0	3	1	349
8	9	1	3	Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)	female	27.0	0	2	34
9	10	1	2	Nasser, Mrs. Nicholas (Adele Achem)	female	14.0	1	0	23
10	11	1	3	Sandstrom, Miss. Marguerite Rut	female	4.0	1	1	PP 9
11	12	1	1	Bonnell, Miss. Elizabeth	female	58.0	0	0	113
12	13	0	3	Saundercock, Mr. William Henry	male	20.0	0	0	A/5.
13	14	0	3	Andersson, Mr. Anders Johan	male	39.0	1	5	347
14	15	0	3	Vestrom, Miss. Hulda Amanda Adolfina	female	14.0	0	0	350

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ti
15	16	1	2	Hewlett, Mrs. (Mary D Kingcome)	female	55.0	0	0	24
16	17	0	3	Rice, Master. Eugene	male	2.0	4	1	38
17	18	1	2	Williams, Mr. Charles Eugene	male	NaN	0	0	24
18	19	0	3	Vander Planke, Mrs. Julius (Emelia Maria Vande...	female	31.0	1	0	34
19	20	1	3	Masselmani, Mrs. Fatima	female	NaN	0	0	
20	21	0	2	Fynney, Mr. Joseph J	male	35.0	0	0	23
21	22	1	2	Beesley, Mr. Lawrence	male	34.0	0	0	24
22	23	1	3	McGowan, Miss. Anna "Annie"	female	15.0	0	0	33
23	24	1	1	Sloper, Mr. William Thompson	male	28.0	0	0	11
24	25	0	3	Palsson, Miss. Torborg Danira	female	8.0	3	1	34
25	26	1	3	Asplund, Mrs. Carl Oscar (Selma Augusta Emilia...	female	38.0	1	5	34
26	27	0	3	Emir, Mr. Farred Chehab	male	NaN	0	0	
27	28	0	1	Fortune, Mr. Charles Alexander	male	19.0	3	2	1
28	29	1	3	O'Dwyer, Miss. Ellen "Nellie"	female	NaN	0	0	33
29	30	0	3	Todoroff, Mr. Lalio	male	NaN	0	0	34

Description of the column content (in part taken from the Kaggle page):

- *PassengerId*, int, unique ID for each passenger

- *Survived*, int, 0 (no) or 1 (yes), is the value to be predicted.
- *Pclass*, int, 1-3, the ticket class but also the social rank
- *Name*, string, full name and title of the traveller
- *Sex*, string, male/female, gender of the traveller
- *Age*, float, age in faction if it's below 1
- *SibSp*, int, sum of siblings and spouse
- *Parch*, int, sum or parents and children
- *Ticket*, string, ticket ID
- *Fare*, float, passenger fare
- *Cabin*, string, cabin ID
- *Embarked*, string, C, S or Q, port where the passenger came on board

The previous output already shows some missing values so checking for missing values in general and counting them is the next step [1]. This information will help to determine wether it's feasable to fill the missing information in or discard a feature.

In [2]:
```
# checking for missing values - code from [1]
total_train = train_df.isnull().sum().sort_values(ascending=False)
percent_train = (round(train_df.isnull().sum() / train_df.isnull().count(

total_test = test_df.isnull().sum().sort_values(ascending=False)
percent_test = (round(test_df.isnull().sum() / test_df.isnull().count() *

missing_data = pd.concat([total_train, percent_train, total_test, percent
missing_data
```

Out[2]:

	Training Total	%	Testing Total	%
Cabin	687	77.1	327.0	78.2
Age	177	19.9	86.0	20.6
Embarked	2	0.2	0.0	0.0
PassengerId	0	0.0	0.0	0.0
Survived	0	0.0	NaN	NaN
Pclass	0	0.0	0.0	0.0
Name	0	0.0	0.0	0.0
Sex	0	0.0	0.0	0.0
SibSp	0	0.0	0.0	0.0
Parch	0	0.0	0.0	0.0
Ticket	0	0.0	0.0	0.0
Fare	0	0.0	1.0	0.2

A couple of things stand aleady out:

- *PassengerId* is clearly irrelevant as it's just an increasing number used as identifier. This column can be discarded.

- *Survived* is the value that should be learned and later predicted. This needs to be split off.
- *Pclass* needs normalization.
- *Name* will not be used for this model but could be used to extract the title to further support the social rank [1].
- *Sex* needs conversion to int.
- *Age* has a couple of missing values but it may be possible to make good guesses from the rest of the data [2].
- *SibSp* and *Parch* need normalization but may be summed up to weather the passenger travelled alone or not [1].
- *Ticket* as random string that can be dropped.
- *Fare* is another support of the social rank, needs normalization.
- *Cabin* has a lot of missing values that can not be guessed and should therefore be discarded (see later for details).
- *Embarked* needs conversion to float.

Now to check if the datasets contain duplicate entries.

In [3]:
```
# find the number of duplicate record
print('Duplicate records')
print('- in the training dataset:', train_df.duplicated().sum())
print('- in the testing dataset:', test_df.duplicated().sum())
```

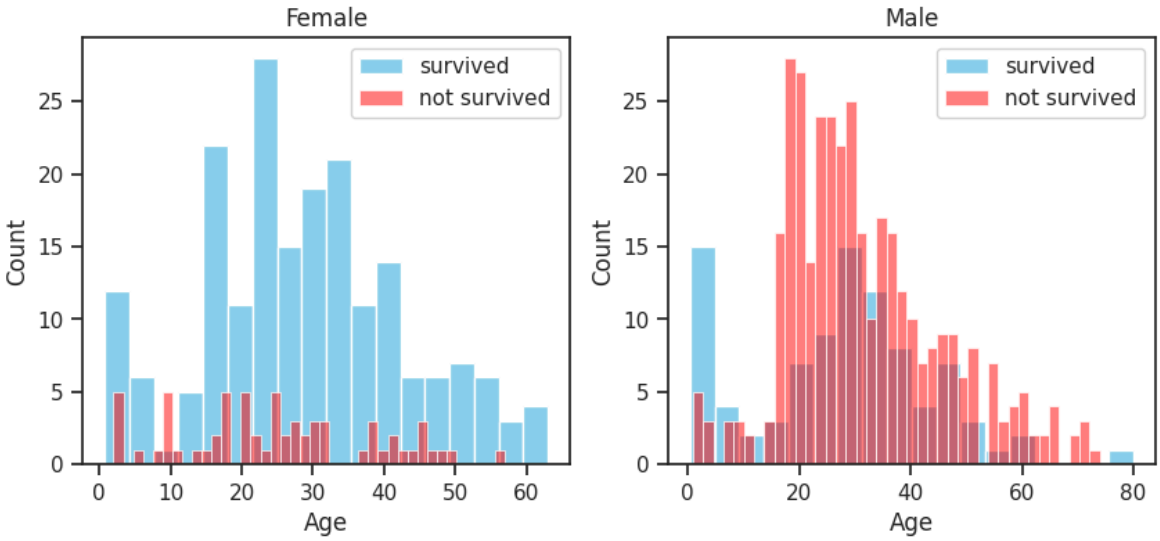
```
Duplicate records
- in the training dataset: 0
- in the testing dataset: 0
```

To get a better feel for the distribution of the datapoints I'm plotting a graph for female and male passengers to see in which age range survival is most common [1].

Female survivers have no single distinct peak but come from all ages whereas mal survivers clearly peak at around 30 years old.

In [10]:
```
# code adapted to work with current seaborn version from [1]

import seaborn as sns
%matplotlib inline
from matplotlib import pyplot as plt

sns.set_theme(style="ticks")

survived = 'survived'
not_survived = 'not survived'

women = train_df[train_df['Sex']=='female']
men = train_df[train_df['Sex']=='male']

fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(10, 4))
ax = sns.histplot(data=women[women['Survived']==1].Age.dropna(), color="s
ax = sns.histplot(data=women[women['Survived']==0].Age.dropna(), color="r
ax.legend()
ax.set_title('Female')
ax = sns.histplot(data=men[men['Survived']==1].Age.dropna(), color="skybl
ax = sns.histplot(data=men[men['Survived']==0].Age.dropna(), color="red",
```

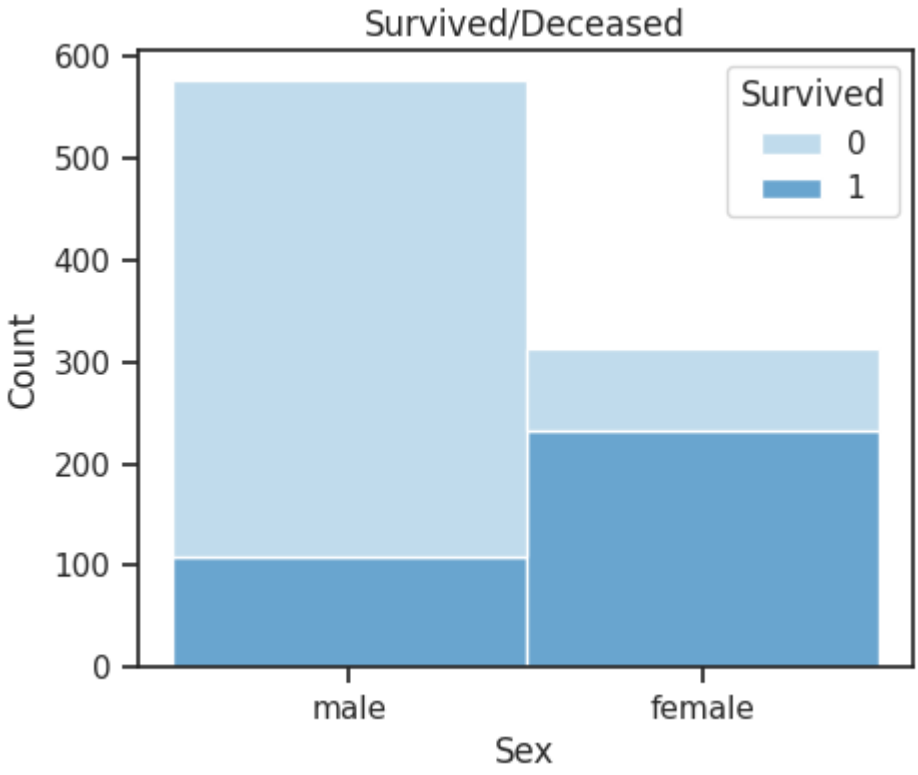
```
ax.legend()
_ = ax.set_title('Male')
```



The following graph sums up male and female passengers and shows how many of each group survived the disaster. Female travelers had a clear survival advantage.

In [11]:
```
# building a temporary dataframe only containing Sex and Survival informa
mf_data = pd.DataFrame({
    'Sex': train_df['Sex'],
    'Survived': train_df['Survived']
})

# prepare the figure
fig, axes = plt.subplots(nrows=1, ncols=1, figsize=(5, 4))

# plot the stacked histogram
sns.histplot(data=mf_data, ax=axes,
             stat='count', multiple='stack',
             x='Sex', palette='Blues',
             kde=False, hue='Survived',
             element='bars', legend=True
             )
axes.set_title('Survived/Deceased')
```

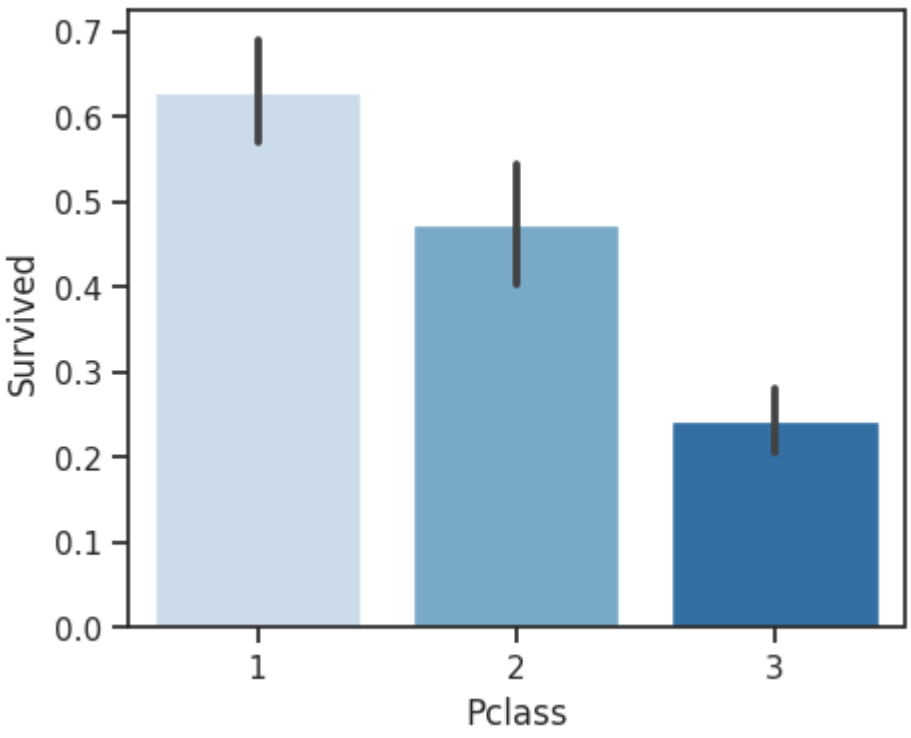
Out[11]:
```
Text(0.5, 1.0, 'Survived/Deceased')
```



The next plot compares survivers and their ticket (social) class. It shows the "elite" has a better chance for survival than the others.

In [12]:
```
# code adapted from [1]

fig, ax = plt.subplots(figsize=(5, 4))
sns.barplot(x='Pclass', y='Survived', data=train_df, palette="Blues")
```

Out[12]:
```
<Axes: xlabel='Pclass', ylabel='Survived'>
```



Choose a Measure of Success

As Chollet and the lecture teaches a measure for success for a balanced classification problem is accuracy. So this will be used in the first model attempt.

Develop a Model

Prepare the Data

I have read about differnt approaches that do no only drop columns, fill missing elements and normalize but add columns by ie. combination of the information in *SibSp* and *Parch* into one feature that only tells if someone travelled alone or not. There has also been an attempt at recreating the missing information in the *Cabin* column but they concluded that this column is yet another feature describing the social status and as this is already present in the *Pclass* column and the recreation of that many missing values is inaccurate anyway it should be discarded.

For this exploration I'll only guess the missing age and port values as there are only a few and they can be guessed with high probability of viability.

Since the preprocessing loops over training and test dataset and some steps are based on data transformations of others this will be done in one somewhat bigger (well commented) code cell.

The following processing steps are performed on the whole dataset (train and test):

- the most frequent port is determined to fill the two empty port entries
- features *PassengerId*, *Name*, *Ticket* and *Cabin* are dropped as they do not provide valuable information for the training
- the 177 missing age values are predicted in relation to gender and social class
- numeric values in *Age*, *Fare*, *SibSp*, *Parch*, *Pclass* are normalized (range -1 - 1)
- string values in *Embarked* are converted to integers (range -1 - 1)
- the *PassengerId* is dropped from the training data as it's unnecessary information for training (the test dataset needs it so the prediction results can be sent to Kaggle)

In [4]:
```
import numpy as np

# updates should happen on both datasets
data = [train_df, test_df]

# which labels to drop
drop_labels = ['Name', 'Ticket', 'Cabin']

# which labels to normalize
norm_labels = ['Age', 'Fare', 'SibSp', 'Parch']

# matrixes holding age and fare guess values
guess_ages = np.zeros((2,3))
guess_fare = np.zeros((1,3))

for dataset in data:
```

```
  # find the mode of port [2]
  freq_port = dataset.Embarked.dropna().mode()[0]

  # fill missing Embarked values
  dataset['Embarked'] = dataset['Embarked'].fillna(freq_port)

  # drop irrelevant features PassengerId, Name, Ticket, Cabin
  dataset.drop(labels=drop_labels, axis=1, inplace=True)

  # convert gender to integers
  dataset['Sex'] = dataset['Sex'].map({'male': 0, 'female': 1}).astype(in

  # filling missing age values - this code comes from [2]
  for i in range(0, 2):
    for j in range(0, 3):
      guess_df = dataset[(dataset['Sex'] == i) & (dataset['Pclass'] == j+
      # take the median
      age_guess = guess_df.median()
      # Convert random age float to nearest .5 age
      guess_ages[i,j] = int( age_guess/0.5 + 0.5 ) * 0.5

      # same routine as above but for the fair
      guess_df = dataset[(dataset['Pclass'] == j+1)]['Fare'].dropna()
      fare_guess = guess_df.median()
      guess_fare[0,j] = int( fare_guess/0.5 + 0.5) * 0.5

  # put the guesed values in the dataframe
  for i in range(0, 2):
    for j in range(0, 3):
        dataset.loc[ (dataset.Age.isnull()) & (dataset.Sex == i) & (datas
        dataset.loc[ (dataset.Fare.isnull()) & (dataset.Pclass == j+1), '

  # normalization - based on Chollet 4.24
  for n in norm_labels:
    mean = dataset[n].mean(axis=0)
    dataset[n] -= mean
    std = dataset[n].std(axis=0)
    dataset[n] /= std

  # convert port and class to normalized values
  dataset['Embarked'] = dataset['Embarked'].map({'S': 1, 'C': 0, 'Q': -1}
  dataset['Pclass'] = dataset['Pclass'].map({1: 1, 2: 0, 3: -1}).astype(i

train_df.drop(labels='PassengerId', axis=1, inplace=True)
```

Checking again what the training dataset looks like after processing.

In [5]:
```
train_df
```

Out[5]:

	Survived	Pclass	Sex	Age	SibSp	Parch	Fare	Embarke
0	0	-1	0	-0.534591	0.432550	-0.473408	-0.502163	
1	1	1	1	0.668017	0.432550	-0.473408	0.786404	
2	1	-1	1	-0.233939	-0.474279	-0.473408	-0.488580	
3	1	1	1	0.442528	0.432550	-0.473408	0.420494	
4	0	-1	0	0.442528	-0.474279	-0.473408	-0.486064	
...	...	...	...	...	...	...	...	
886	0	0	0	-0.158776	-0.474279	-0.473408	-0.386454	
887	1	1	1	-0.760080	-0.474279	-0.473408	-0.044356	
888	0	-1	1	-0.572172	0.432550	2.007806	-0.176164	
889	1	1	0	-0.233939	-0.474279	-0.473408	-0.044356	
890	0	-1	0	0.217039	-0.474279	-0.473408	-0.492101	

891 rows × 8 columns

And verifying that all missing values have been replaced. (*PassengerId* in the testing set has been dropped and is therefore NaN).

In [6]:
```
# checking for missing values - code from [1]
total_train = train_df.isnull().sum().sort_values(ascending=False)
percent_train = (round(train_df.isnull().sum() / train_df.isnull().count(

total_test = test_df.isnull().sum().sort_values(ascending=False)
percent_test = (round(test_df.isnull().sum() / test_df.isnull().count() *

missing_data = pd.concat([total_train, percent_train, total_test, percent
missing_data
```

Out[6]:

	Training Total	%	Testing Total	%
Survived	0.0	0.0	NaN	NaN
Pclass	0.0	0.0	0.0	0.0
Sex	0.0	0.0	0.0	0.0
Age	0.0	0.0	0.0	0.0
SibSp	0.0	0.0	0.0	0.0
Parch	0.0	0.0	0.0	0.0
Fare	0.0	0.0	0.0	0.0
Embarked	0.0	0.0	0.0	0.0
PassengerId	NaN	NaN	0.0	0.0

Select an Evaluation Protocol

The dataset holds 891 entries which is not a lot to train a neural network. This lends itself to implementing the training using k-fold so the data is optimally evaluated. But since it's not a lot of data and processing it will be fairly quickly I'll also try holdout as this is described (in Chollet) as a good starting point too (and I'm curious).

Choose a Baseline

Searching for probabilities of surviving a shipwreck it turns out that the average survival rate for passengers is about 31.9 percent.

So the NN needs to predict about this percentage of survivers to be fit.

The article actually mentions that on the Titanic passengers had a 38% survial rate which is a statistical outlier.

Train the First Model

For the first model the hyperparameters (layer count, unit size, epochs, batch size) are chosen around the values taught in the Chollet examples with the goal to get the model to overfit.

Overfitting is when the training continous after the validation set loss reached a minimum and begins to rise again. This means that the neural network learns to fit the training data much better and looses capablilities to accurately predict the validation set.

In [7]:
```
# importing the necessary packages
from tensorflow import keras
from tensorflow.keras import layers
import matplotlib.pyplot as plt
```

The following cell defines two methods that plot training and validation loss and accuracy. Both plot two graphs but the second method is tweaked to build small sub plots and place them accordingly (used in the hyperparameter exploration).

In [8]:
```
# based on plot generation in Chollet chapter 4

# plot loss and accuracy side by side
def plot_loss_acc(hist_dict):
  fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(12, 4))

  loss_values = hist_dict["loss"]
  val_loss_values = hist_dict["val_loss"]
  epochs = range(1, len(loss_values) + 1)
  ax[0].plot(epochs, loss_values, label="Training loss", color="skyblue")
  ax[0].plot(epochs, val_loss_values, label="Validation loss", color="red
  ax[0].set_title("Training and validation loss")
  ax[0].set_xlabel("Epochs")
  ax[0].set_ylabel("Loss")
  ax[0].legend()

  acc = hist_dict["accuracy"]
  val_acc = hist_dict["val_accuracy"]
```

```
  ax[1].plot(epochs, acc, label="Training acc", color="skyblue")
  ax[1].plot(epochs, val_acc, label="Validation acc", color="red")
  ax[1].set_title("Training and validation accuracy")
  ax[1].set_xlabel("Epochs")
  ax[1].set_ylabel("Accuracy")
  ax[1].legend()

  fig.show()

# same plot but adapted to build a huge plot for the param testing below
def plot_loss_acc2(hist_dict, title, ax, plot_row, plot_col):
  loss_values = hist_dict["loss"]
  val_loss_values = hist_dict["val_loss"]
  epochs = range(1, len(loss_values) + 1)
  ax[plot_row, plot_col].plot(epochs, loss_values, label="train loss", co
  ax[plot_row, plot_col].plot(epochs, val_loss_values, label="val loss",
  ax[plot_row, plot_col].set_title("Loss " + title)
  ax[plot_row, plot_col].legend()

  acc = hist_dict["accuracy"]
  val_acc = hist_dict["val_accuracy"]
  ax[plot_row, plot_col+1].plot(epochs, acc, label="train acc", color="sk
  ax[plot_row, plot_col+1].plot(epochs, val_acc, label="val acc", color="
  ax[plot_row, plot_col+1].set_title("Accuracy " + title)
  ax[plot_row, plot_col+1].legend()
```

In the upcoming cell a k-fold cross validation approach is used to evaluate the accuracy of an initial model with two dense layers that have 32 units and use a rectified linear unit function for activation. The third layer only has one unit and a sigmoid activation as the outcome of the network should be a probability between 0 and 1 (dead or alive).

Optimizer, loss and metrics are chosen according to the best practice for a binary classification problem (tabulated in Chollet).

In [99]:
```
# code adapted from Chollet 4.3

# sets up a model and returns it
def build_model():
  # model setup
  model = keras.Sequential([
      layers.Dense(32, activation="relu"),
      layers.Dense(32, activation="relu"),
      layers.Dense(1, activation='sigmoid')
  ])

  # compile the model
  model.compile(
      optimizer='rmsprop',
      loss='binary_crossentropy',
      metrics=['accuracy']
  )

  # return the model
  return model

# set up training data
train_data = train_df.drop(labels='Survived', axis=1)
```

```
# and training targets
train_targets = train_df['Survived']

# how many folds there should bd
k = 4
# samples calculated as number training data available divided by number
num_val_samples = len(train_data) // k
# how many epochs to train for
num_epochs = 300
# history collection array
all_acc_histories = []
for i in range(k):
    print(f'Processing fold #{i}')
    # splitting training and validation dataset according to fold
    # the validation set is only one part
    val_data = train_data[i * num_val_samples: (i + 1) * num_val_samples]
    val_targets = train_targets[i * num_val_samples: (i + 1) * num_val_sa
    # the training data needs to be assembled
    partial_train_data = np.concatenate(
        [train_data[:i * num_val_samples],
         train_data[(i + 1) * num_val_samples:]],
        axis=0)
    partial_train_targets = np.concatenate(
        [train_targets[:i * num_val_samples],
         train_targets[(i + 1) * num_val_samples:]],
        axis=0)
    # build the model for this run
    model = build_model()
    # train the NN and store the history
    history = model.fit(partial_train_data, partial_train_targets,
                        validation_data=(val_data, val_targets),
                        epochs=num_epochs, batch_size=16, verbose=0)
    # extract validation accuracy and store globally
    acc_history = history.history['val_accuracy']
    all_acc_histories.append(acc_history)
```

```
Processing fold #0
Processing fold #1
Processing fold #2
Processing fold #3
```

In [103…

```
# code adapted from Chollet 4.3

# calculates the mean accuracy for all folds
average_acc_history = [np.mean([x[i] for x in all_acc_histories]) for i i
# output the mean accuracy
np.mean(average_acc_history)
```

Out[103…

```
0.8109947382907072
```

The mean accuracy using a k-fold validation approach reached about 81%. That is not a bad start. Out of curiosity the next cells will try holdout to see how this approach works.

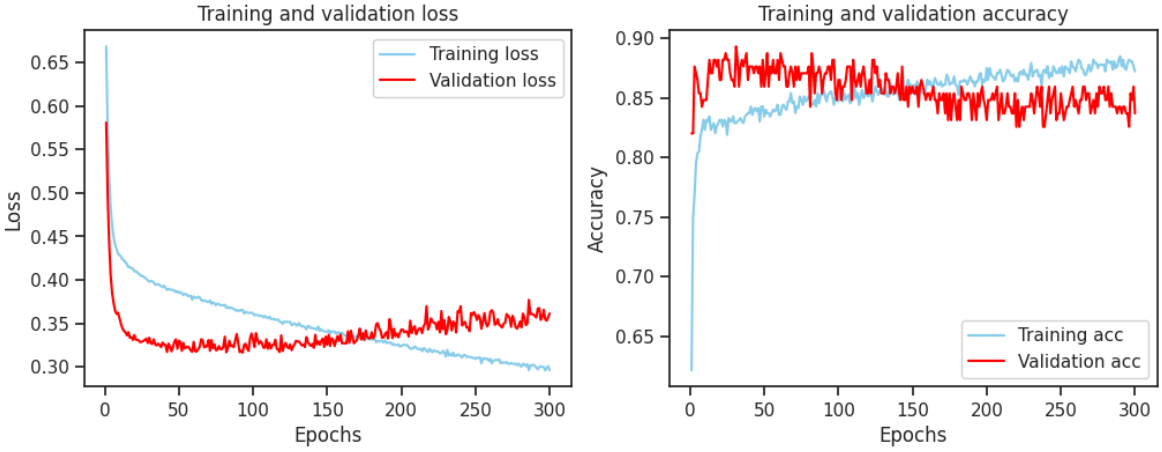
As the data rows in the training set are already randomized the set is split in one part training and another part for validation. In both cases the x prefixed dataset drops the *Survived* feature as this should be the outcome and not the input.

In [9]:
```
# split the training data into train and validation 80%:20% sub sets

# split and remove Survived feature which should not be learned but predi
partial_x_train = train_df[:713].drop(labels='Survived', axis=1)
# split raw data again but only store the Survived feature
partial_y_train = train_df['Survived'][:713]

# validation set split
x_val = train_df[713:].drop(labels='Survived', axis=1)
y_val = train_df['Survived'][713:]
```

The model setup for the holdout method is the same as it was in the k-fold implementation.

In [105…
```
# prepare the model
model = keras.Sequential([
    layers.Dense(32, activation='relu'),
    layers.Dense(32, activation='relu'),
    layers.Dense(1, activation='sigmoid')
])

# compile the mode
model.compile(
    optimizer='rmsprop',
    loss='binary_crossentropy',
    metrics=['accuracy']
)

# train the network and store the history
history = model.fit(
    partial_x_train,
    partial_y_train,
    epochs=300,
    batch_size=16,
    validation_data=(x_val, y_val),
    verbose=0
)

# print some info and plot graphs
print('mean accuracy:', np.mean(history.history['accuracy']))
print('optimal number of epochs:', np.argmin(history.history["val_loss"]))
plot_loss_acc(history.history)
```

```
mean accuracy: 0.85542309542497
optimal number of epochs: 92
```



The mean accuracy using holdout is a little bit higher than during the k-fold run. The higher accuracy can simply be due to the specific train and validation set selection for holdout. But both accuracy values are close enough together so that I'll proceed with holdout as the following step exploring different hyperparameters already takes ~40min processing time without having split the data set k times in each run.

In general it can be observed that overfitting already occurs using this simple model with "default" parameters albeit it takes almost ~125 epochs until this happens.

Let's expore different hyperparamerter settings to see how training and validation evolves. For this a method (*test_run*) is created that takes all necessary data and settings as parameters for one paricular neural network build, fit and evaluation. It also calls the previously customized plot function to visualize the output.

In [107…

```
# method to build, compile and fit a model with parameters
# given as method parameters
def test_run(x_train, y_train, x_val, y_val,
             ax, plot_row, plot_col,
             title='foo',
             num_layers=2, layer_units=32,
             optimizer='rmsprop', loss='binary_crossentropy',
             metric='accuracy', epochs=500,
             batch_size=64):
  # list of model layers
  layer_list = []

  # assemble the layers
  # it's always the same layer type and setting
  for i in range(num_layers):
    layer_list.append(layers.Dense(layer_units, activation='relu'))

  # add the last layer for probability output (0-1)
  layer_list.append(layers.Dense(1, activation='sigmoid'))

  # set up the model
  model = keras.Sequential(layer_list)

  # compile the model
  model.compile(
    optimizer=optimizer,
    loss=loss,
    metrics=[metric]
  )

  # train the network and store the history
  history = model.fit(
    x_train,
    y_train,
    epochs=epochs,
    batch_size=batch_size,
    validation_data=(x_val, y_val),
    verbose=0
  )

  # print max accuracy/validation accuracy, min loss/validation loss
  print(title, 'max accuracy:', np.max(history.history['accuracy']),
```

```
        np.max(history.history['val_accuracy']), 'min loss:',
        np.min(history.history['loss']), np.min(history.history['val_loss

  # plot the two sub plots
  plot_loss_acc2(history.history, title, ax, plot_row, plot_col)
```

The next cell uses nested for loops to explore different parameter settings. In particular the following is checked:

- number of layers : 2 or 3
- unit size : 16, 32, 64, 128
- training epoch count : 200, 300, 400
- batch size : 32, 64, 128

This will produce 72 different neural networks, summary text output and visualization of accuracy and loss for training and validation. Since the training data set is not particularely big this kind of (brute force) exploration is possible and took about ~40min runtime in a Google Colab notebook.

In [110…

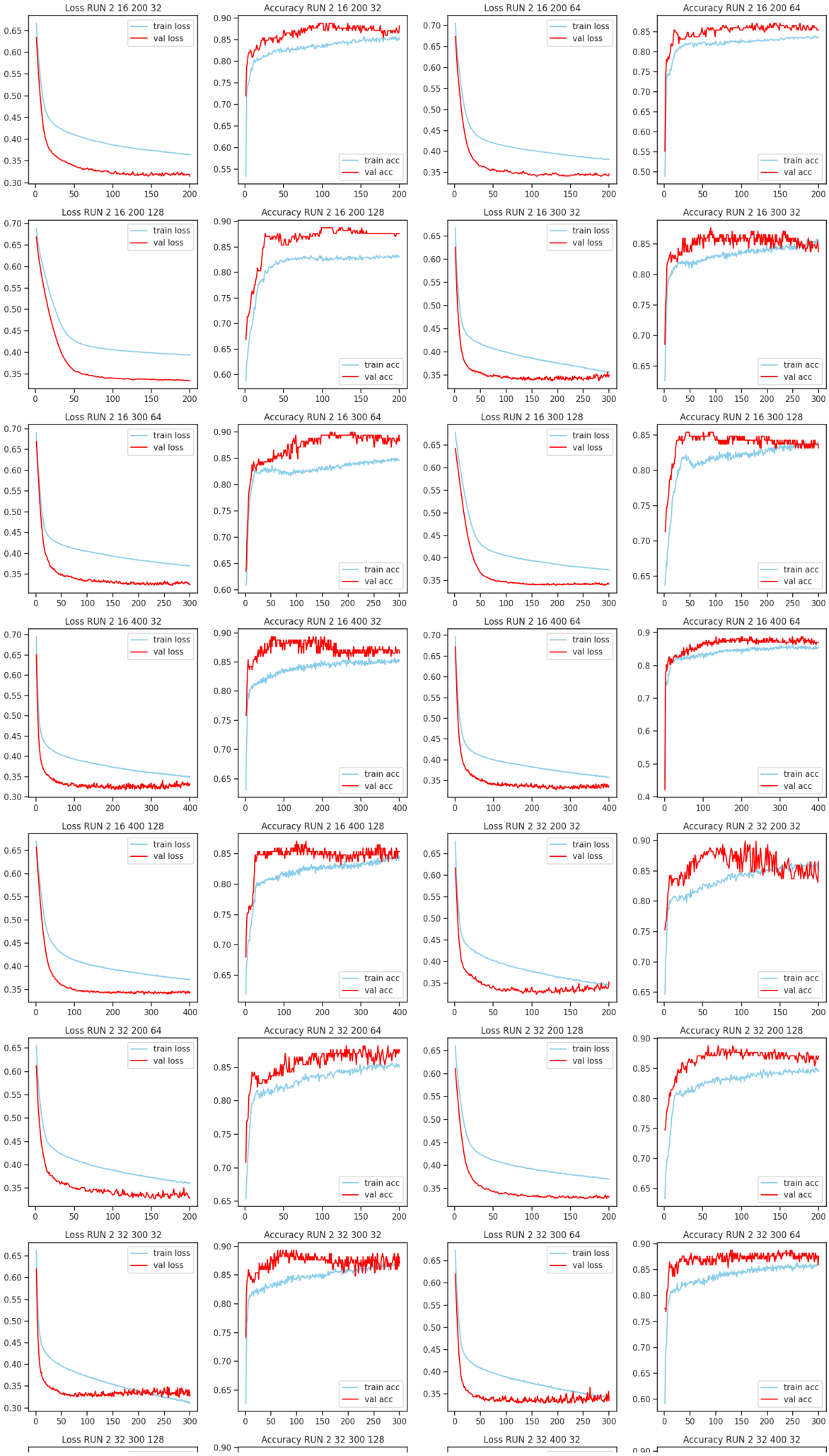
```
# plot row and column
row = 0
col = 0

# create a huge plot with 72 sub plots
fig, axs = plt.subplots(nrows=36, ncols=4, figsize=(16, 140))
# fig.suptitle('NN Hyperparameter Tuning')

# loop over how many layers the NN should have
for l in range(2, 4):
  # loop over the unit size of each layer
  for u in [16, 32, 64, 128]:
    # loop over the number of epochs the training should run
    for e in [200, 300, 400]:
      # loop over the batch size for training
      for b in [32, 64, 128]:
        # assemble a simple title
        title = 'RUN ' + str(l) + ' ' + str(u) + ' ' + str(e) + ' ' + str
        # run the NN test
        test_run(
            partial_x_train, partial_y_train,
            x_val, y_val,
            axs, row, col,
            title=title,
            num_layers=l,
            layer_units=u,
            epochs=e,
            batch_size=b,
        )

        # increase the plot column
        col += 2
        # reset the column and increase row count
        if (col == 4):
          col = 0
          row += 1
```

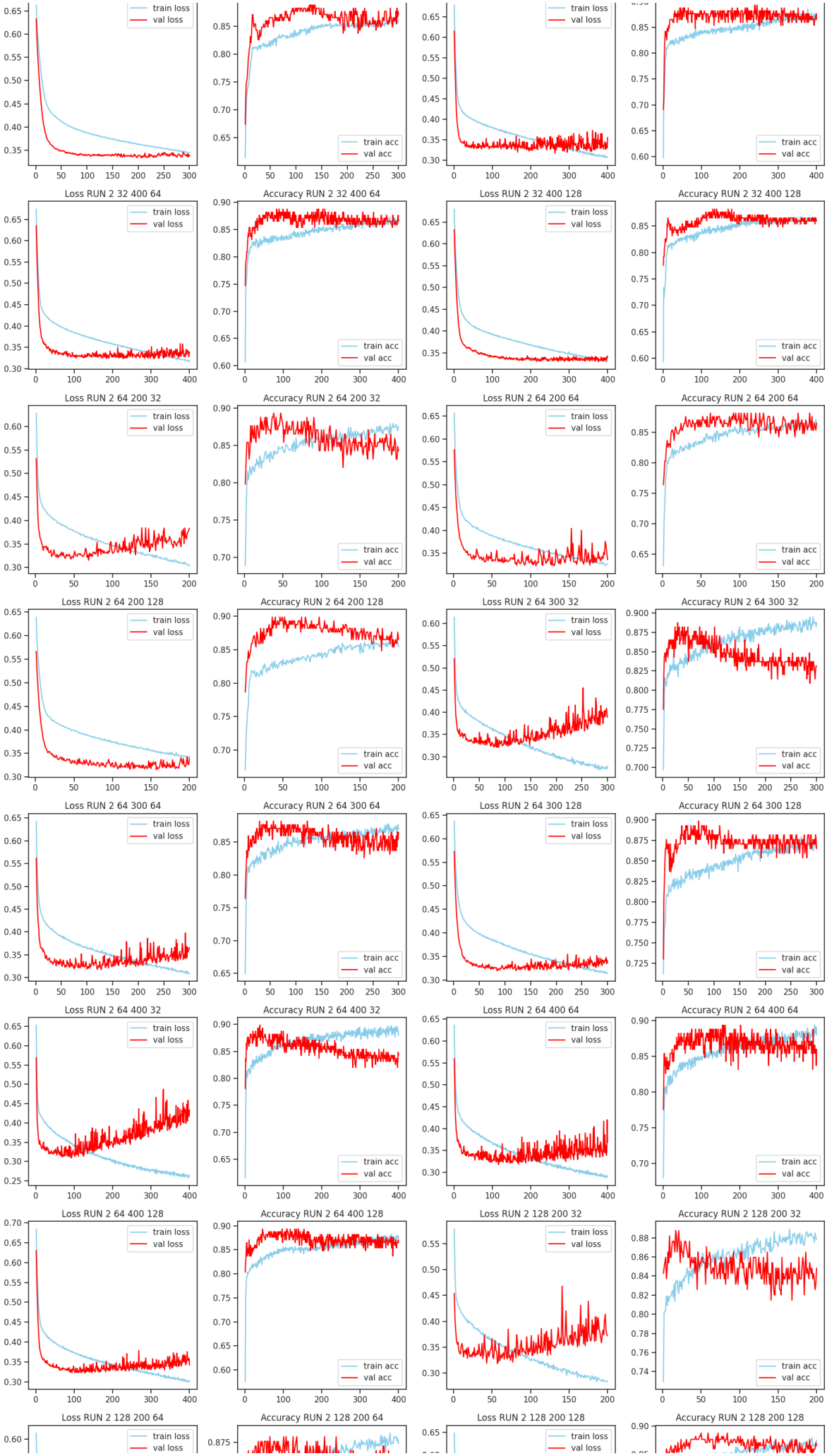
```
fig.tight_layout()
```

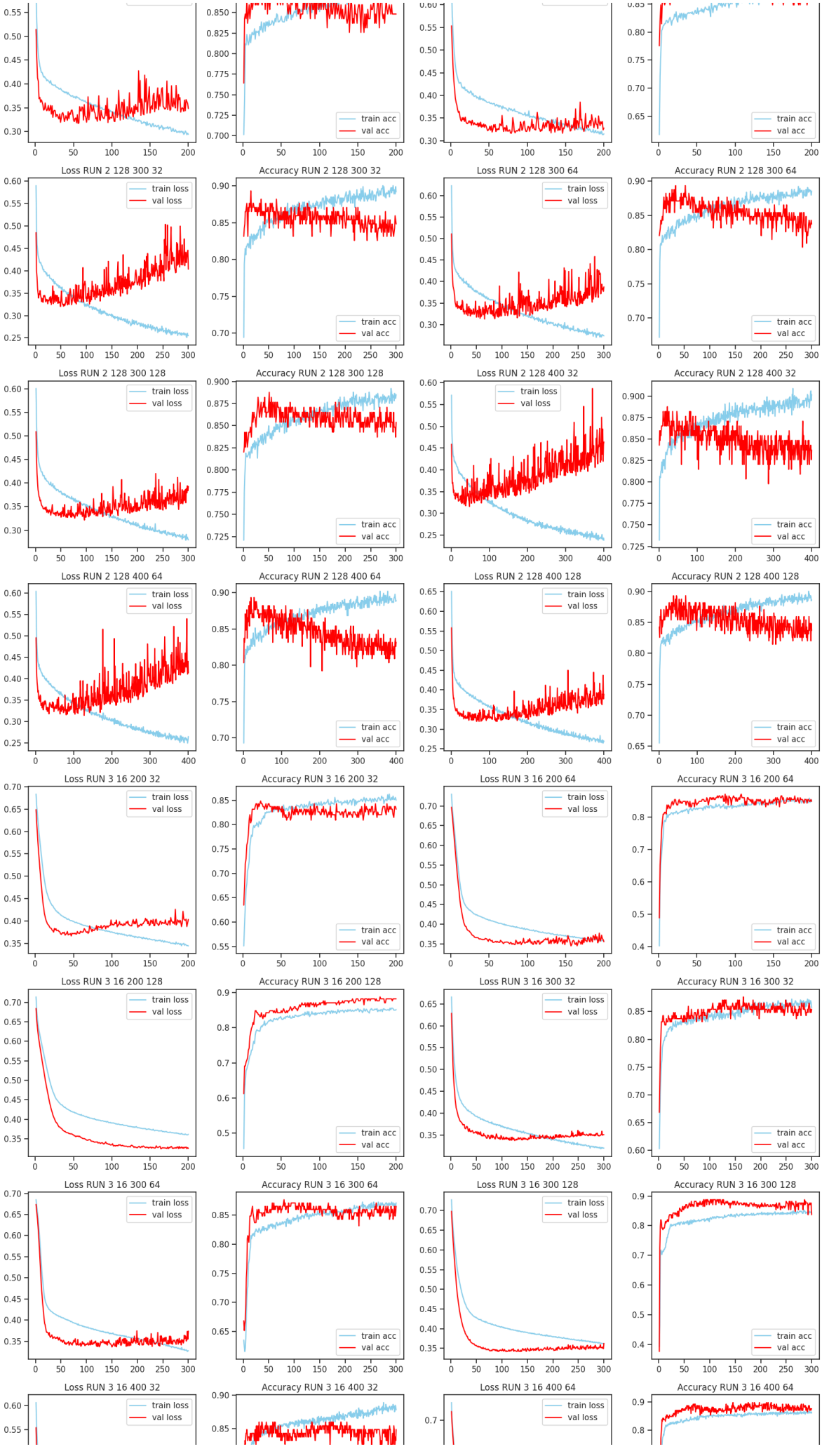
```
RUN 2 16 200 32 max accuracy: 0.8569424748420715 0.8876404762268066 min lo
ss: 0.36457231640815735 0.3145369291305542
RUN 2 16 200 64 max accuracy: 0.8401122093200684 0.8707864880561829 min lo
ss: 0.3809678256511688 0.3408249318599701
RUN 2 16 200 128 max accuracy: 0.834502100944519 0.8876404762268066 min lo
ss: 0.39365750551223755 0.33387434482574463
RUN 2 16 300 32 max accuracy: 0.8583450317382812 0.8764045238494873 min lo
ss: 0.3554205000400543 0.3378413915634155
RUN 2 16 300 64 max accuracy: 0.8499298691749573 0.898876428604126 min los
s: 0.36866283416748047 0.32269856333732605
RUN 2 16 300 128 max accuracy: 0.8415147066116333 0.8539325594902039 min l
oss: 0.3736020624637604 0.3393014967441559
RUN 2 16 400 32 max accuracy: 0.8569424748420715 0.8932584524154663 min lo
ss: 0.3487137258052826 0.31736916303634644
RUN 2 16 400 64 max accuracy: 0.8625525832176208 0.8876404762268066 min lo
ss: 0.35745182633399963 0.32766595482826233
RUN 2 16 400 128 max accuracy: 0.8485273718833923 0.8707864880561829 min l
oss: 0.37105312943458557 0.33907803893089294
RUN 2 32 200 32 max accuracy: 0.8653576374053955 0.898876428604126 min los
s: 0.34468525648117065 0.3235231339931488
RUN 2 32 200 64 max accuracy: 0.8569424748420715 0.882022500038147 min los
s: 0.3603476583957672 0.3269679844379425
RUN 2 32 200 128 max accuracy: 0.851332426071167 0.8876404762268066 min lo
ss: 0.3701736330986023 0.32762008905410767
RUN 2 32 300 32 max accuracy: 0.8723702430725098 0.8932584524154663 min lo
ss: 0.3106086254119873 0.3242550790309906
RUN 2 32 300 64 max accuracy: 0.8695651888847351 0.8876404762268066 min lo
ss: 0.3414069414138794 0.3288508355617523
RUN 2 32 300 128 max accuracy: 0.8625525832176208 0.8876404762268066 min l
oss: 0.34458988904953003 0.3335905969142914
RUN 2 32 400 32 max accuracy: 0.8849930167198181 0.8932584524154663 min lo
ss: 0.305798202753067 0.3168828785419464
RUN 2 32 400 64 max accuracy: 0.8723702430725098 0.8876404762268066 min lo
ss: 0.31583163142204285 0.3228941559791565
RUN 2 32 400 128 max accuracy: 0.8681626915931702 0.882022500038147 min lo
ss: 0.3292945325374603 0.32950639724731445
RUN 2 64 200 32 max accuracy: 0.8793829083442688 0.8932584524154663 min lo
ss: 0.3034588694572449 0.315172016620636
RUN 2 64 200 64 max accuracy: 0.8709677457809448 0.882022500038147 min los
s: 0.32335373759269714 0.32266947627067566
RUN 2 64 200 128 max accuracy: 0.8653576374053955 0.898876428604126 min lo
ss: 0.34007060527801514 0.31544923782348633
RUN 2 64 300 32 max accuracy: 0.894810676574707 0.8876404762268066 min los
s: 0.27045851945877075 0.32094937562942505
RUN 2 64 300 64 max accuracy: 0.8779803514480591 0.882022500038147 min los
s: 0.30860933661460876 0.3179112374782562
RUN 2 64 300 128 max accuracy: 0.8793829083442688 0.898876428604126 min lo
ss: 0.3130434453487396 0.3194584250450134
RUN 2 64 400 32 max accuracy: 0.8976157307624817 0.898876428604126 min los
s: 0.2580525279045105 0.3093944787979126
RUN 2 64 400 64 max accuracy: 0.8934081196784973 0.8932584524154663 min lo
ss: 0.2876657247543335 0.3179338276386261
RUN 2 64 400 128 max accuracy: 0.8821879625320435 0.8932584524154663 min l
oss: 0.2997235953807831 0.3219010531902313
RUN 2 128 200 32 max accuracy: 0.8892005681991577 0.8876404762268066 min l
oss: 0.28282901644706726 0.3181363344192505
RUN 2 128 200 64 max accuracy: 0.8863955140113831 0.882022500038147 min lo
ss: 0.2933553457260132 0.3168126344680786
RUN 2 128 200 128 max accuracy: 0.8793829083442688 0.8876404762268066 min
loss: 0.31359773874282837 0.3163864314556122
```

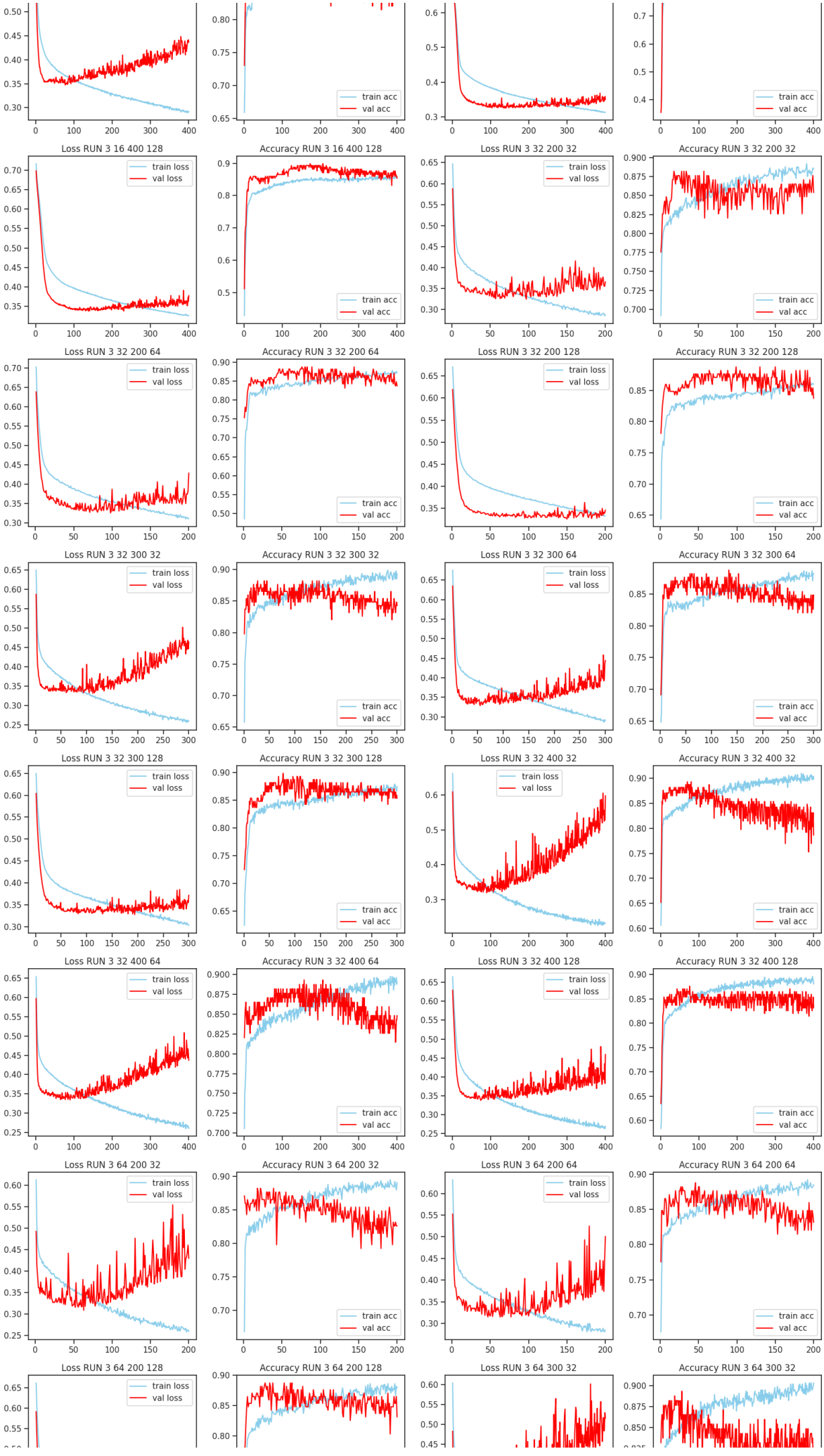
```
RUN 2 128 300 32 max accuracy: 0.9004207849502563 0.8932584524154663 min l
oss: 0.2510685622692108 0.3201621174812317
RUN 2 128 300 64 max accuracy: 0.8920056223869324 0.8932584524154663 min l
oss: 0.27012547850608826 0.31288278102874756
RUN 2 128 300 128 max accuracy: 0.8920056223869324 0.8876404762268066 min
loss: 0.278602659702301 0.3215205669403076
RUN 2 128 400 32 max accuracy: 0.9088358879089355 0.8876404762268066 min l
oss: 0.2378692477941513 0.3155094087123871
RUN 2 128 400 64 max accuracy: 0.9018232822418213 0.8932584524154663 min l
oss: 0.24962854385375977 0.31432193517684937
RUN 2 128 400 128 max accuracy: 0.9004207849502563 0.8932584524154663 min
loss: 0.2646080255508423 0.3195216655731201
RUN 3 16 200 32 max accuracy: 0.8625525832176208 0.8483145833015442 min lo
ss: 0.344379723072052 0.3665779232978821
RUN 3 16 200 64 max accuracy: 0.8583450317382812 0.8707864880561829 min lo
ss: 0.35740330815315247 0.34518420696258545
RUN 3 16 200 128 max accuracy: 0.8555399775505066 0.8876404762268066 min l
oss: 0.3598886728286743 0.32430461049079895
RUN 3 16 300 32 max accuracy: 0.8723702430725098 0.8764045238494873 min lo
ss: 0.3185182213783264 0.33721286058425903
RUN 3 16 300 64 max accuracy: 0.8723702430725098 0.8764045238494873 min lo
ss: 0.32580098509788513 0.33661484718322754
RUN 3 16 300 128 max accuracy: 0.8527349233627319 0.8876404762268066 min l
oss: 0.36139997839927673 0.34182286262512207
RUN 3 16 400 32 max accuracy: 0.8892005681991577 0.8595505356788635 min lo
ss: 0.2898155450820923 0.3472716808319092
RUN 3 16 400 64 max accuracy: 0.8667601943016052 0.898876428604126 min los
s: 0.31284981966018677 0.32575762271881104
RUN 3 16 400 128 max accuracy: 0.8625525832176208 0.898876428604126 min lo
ss: 0.3259150981903076 0.33803051710128784
RUN 3 32 200 32 max accuracy: 0.8920056223869324 0.882022500038147 min los
s: 0.28538209199905396 0.32507026195526123
RUN 3 32 200 64 max accuracy: 0.8779803514480591 0.8876404762268066 min lo
ss: 0.3103374242782593 0.3261335492134094
RUN 3 32 200 128 max accuracy: 0.8625525832176208 0.8876404762268066 min l
oss: 0.33260536193847656 0.32586953043937683
RUN 3 32 300 32 max accuracy: 0.8990182280540466 0.882022500038147 min los
s: 0.2564300298690796 0.33138927817344666
RUN 3 32 300 64 max accuracy: 0.8863955140113831 0.8876404762268066 min lo
ss: 0.28643059730529785 0.3290224075317383
RUN 3 32 300 128 max accuracy: 0.8793829083442688 0.898876428604126 min lo
ss: 0.30345064401626587 0.32816600799560547
RUN 3 32 400 32 max accuracy: 0.9102384448051453 0.8932584524154663 min lo
ss: 0.22602880001068115 0.31988832354545593
RUN 3 32 400 64 max accuracy: 0.8976157307624817 0.8932584524154663 min lo
ss: 0.26003894209861755 0.33458003401756287
RUN 3 32 400 128 max accuracy: 0.896213173866272 0.8764045238494873 min lo
ss: 0.2629985809326172 0.3379009962081909
RUN 3 64 200 32 max accuracy: 0.894810676574707 0.882022500038147 min los
s: 0.25821778178215027 0.3086432218551636
RUN 3 64 200 64 max accuracy: 0.8920056223869324 0.8876404762268066 min lo
ss: 0.28078198432922363 0.31496956944465637
RUN 3 64 200 128 max accuracy: 0.8863955140113831 0.8876404762268066 min l
oss: 0.28953880071640015 0.3251194357872009
RUN 3 64 300 32 max accuracy: 0.9032257795333862 0.8932584524154663 min lo
ss: 0.22095142304897308 0.3152983486652374
RUN 3 64 300 64 max accuracy: 0.9018232822418213 0.8876404762268066 min lo
ss: 0.24726611375808716 0.3266204595565796
RUN 3 64 300 128 max accuracy: 0.894810676574707 0.8932584524154663 min lo
ss: 0.25751546025276184 0.3221261501312256
```

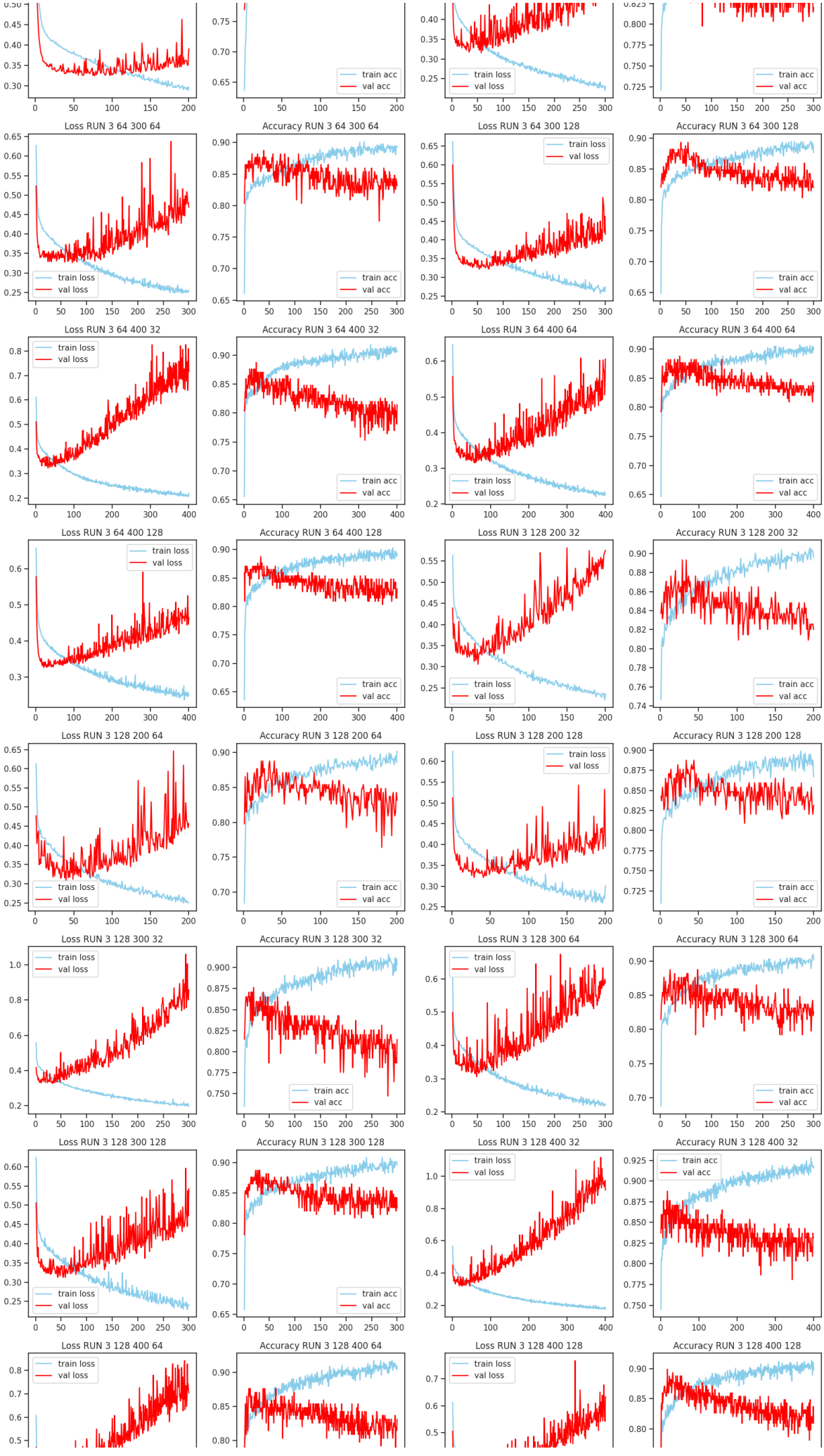
```
RUN 3 64 400 32 max accuracy: 0.9186535477638245 0.8876404762268066 min lo
ss: 0.20603908598423004 0.3233202397823334
RUN 3 64 400 64 max accuracy: 0.9074333906173706 0.8876404762268066 min lo
ss: 0.2207263708114624 0.31559914350509644
RUN 3 64 400 128 max accuracy: 0.9032257795333862 0.8876404762268066 min l
oss: 0.2369167059659958 0.3267776072025299
RUN 3 128 200 32 max accuracy: 0.9060308337211609 0.8932584524154663 min l
oss: 0.2215294986963272 0.3059001564979553
RUN 3 128 200 64 max accuracy: 0.9018232822418213 0.8876404762268066 min l
oss: 0.2493429034948349 0.308757483959198
RUN 3 128 200 128 max accuracy: 0.8990182280540466 0.8876404762268066 min
loss: 0.25927650928497314 0.3206077218055725
RUN 3 128 300 32 max accuracy: 0.9158485531806946 0.8764045238494873 min l
oss: 0.19624163210391998 0.3260044753551483
RUN 3 128 300 64 max accuracy: 0.9102384448051453 0.8876404762268066 min l
oss: 0.21679040789604187 0.3061066269874573
RUN 3 128 300 128 max accuracy: 0.9088358879089355 0.8876404762268066 min
loss: 0.22904732823371887 0.31266239285469055
RUN 3 128 400 32 max accuracy: 0.9284712672233582 0.8876404762268066 min l
oss: 0.1738879680633545 0.31896090507507324
RUN 3 128 400 64 max accuracy: 0.9172510504722595 0.8764045238494873 min l
oss: 0.19125644862651825 0.3246479332447052
RUN 3 128 400 128 max accuracy: 0.9130434989929199 0.898876428604126 min l
oss: 0.20498627424240112 0.31369301676750183
```

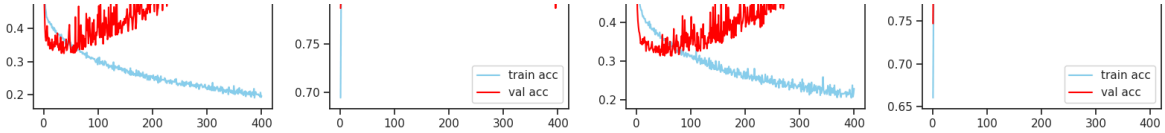

Loss RUN 2 16 200 32
Accuracy RUN 2 16 200 32
Loss RUN 2 16 200 64
Accuracy RUN 2 16 200 64
Loss RUN 2 16 200 128
Accuracy RUN 2 16 200 128
Loss RUN 2 16 300 32
Accuracy RUN 2 16 300 32
Loss RUN 2 16 300 64
Accuracy RUN 2 16 300 64
Loss RUN 2 16 300 128
Accuracy RUN 2 16 300 128
Loss RUN 2 16 400 32
Accuracy RUN 2 16 400 32
Loss RUN 2 16 400 64
Accuracy RUN 2 16 400 64
Loss RUN 2 16 400 128
Accuracy RUN 2 16 400 128
Loss RUN 2 32 200 32
Accuracy RUN 2 32 200 32
Loss RUN 2 32 200 64
Accuracy RUN 2 32 200 64
Loss RUN 2 32 200 128
Accuracy RUN 2 32 200 128
Loss RUN 2 32 300 32
Accuracy RUN 2 32 300 32
Loss RUN 2 32 300 64
Accuracy RUN 2 32 300 64
Loss RUN 2 32 300 128
Accuracy RUN 2 32 300 128
Loss RUN 2 32 400 32
Accuracy RUN 2 32 400 32
train loss
val loss
train acc
val acc


train loss
val loss
train acc
val acc
Loss RUN 2 32 400 64
Accuracy RUN 2 32 400 64
Loss RUN 2 32 400 128
Accuracy RUN 2 32 400 128
Loss RUN 2 64 200 32
Accuracy RUN 2 64 200 32
Loss RUN 2 64 200 64
Accuracy RUN 2 64 200 64
Loss RUN 2 64 200 128
Accuracy RUN 2 64 200 128
Loss RUN 2 64 300 32
Accuracy RUN 2 64 300 32
Loss RUN 2 64 300 64
Accuracy RUN 2 64 300 64
Loss RUN 2 64 300 128
Accuracy RUN 2 64 300 128
Loss RUN 2 64 400 32
Accuracy RUN 2 64 400 32
Loss RUN 2 64 400 64
Accuracy RUN 2 64 400 64
Loss RUN 2 64 400 128
Accuracy RUN 2 64 400 128
Loss RUN 2 128 200 32
Accuracy RUN 2 128 200 32
Loss RUN 2 128 200 64
Accuracy RUN 2 128 200 64
Loss RUN 2 128 200 128
Accuracy RUN 2 128 200 128


Loss RUN 2 128 300 32
Accuracy RUN 2 128 300 32
Loss RUN 2 128 300 64
Accuracy RUN 2 128 300 64
Loss RUN 2 128 300 128
Accuracy RUN 2 128 300 128
Loss RUN 2 128 400 32
Accuracy RUN 2 128 400 32
Loss RUN 2 128 400 64
Accuracy RUN 2 128 400 64
Loss RUN 2 128 400 128
Accuracy RUN 2 128 400 128
Loss RUN 3 16 200 32
Accuracy RUN 3 16 200 32
Loss RUN 3 16 200 64
Accuracy RUN 3 16 200 64
Loss RUN 3 16 200 128
Accuracy RUN 3 16 200 128
Loss RUN 3 16 300 32
Accuracy RUN 3 16 300 32
Loss RUN 3 16 300 64
Accuracy RUN 3 16 300 64
Loss RUN 3 16 300 128
Accuracy RUN 3 16 300 128
Loss RUN 3 16 400 32
Accuracy RUN 3 16 400 32
Loss RUN 3 16 400 64
Accuracy RUN 3 16 400 64
train loss
val loss
train acc
val acc


Loss RUN 3 16 400 128
Accuracy RUN 3 16 400 128
Loss RUN 3 32 200 32
Accuracy RUN 3 32 200 32
Loss RUN 3 32 200 64
Accuracy RUN 3 32 200 64
Loss RUN 3 32 200 128
Accuracy RUN 3 32 200 128
Loss RUN 3 32 300 32
Accuracy RUN 3 32 300 32
Loss RUN 3 32 300 64
Accuracy RUN 3 32 300 64
Loss RUN 3 32 300 128
Accuracy RUN 3 32 300 128
Loss RUN 3 32 400 32
Accuracy RUN 3 32 400 32
Loss RUN 3 32 400 64
Accuracy RUN 3 32 400 64
Loss RUN 3 32 400 128
Accuracy RUN 3 32 400 128
Loss RUN 3 64 200 32
Accuracy RUN 3 64 200 32
Loss RUN 3 64 200 64
Accuracy RUN 3 64 200 64
Loss RUN 3 64 200 128
Accuracy RUN 3 64 200 128
Loss RUN 3 64 300 32
Accuracy RUN 3 64 300 32
train loss
val loss
train acc
val acc


Loss RUN 3 64 300 64
Accuracy RUN 3 64 300 64
Loss RUN 3 64 300 128
Accuracy RUN 3 64 300 128
Loss RUN 3 64 400 32
Accuracy RUN 3 64 400 32
Loss RUN 3 64 400 64
Accuracy RUN 3 64 400 64
Loss RUN 3 64 400 128
Accuracy RUN 3 64 400 128
Loss RUN 3 128 200 32
Accuracy RUN 3 128 200 32
Loss RUN 3 128 200 64
Accuracy RUN 3 128 200 64
Loss RUN 3 128 200 128
Accuracy RUN 3 128 200 128
Loss RUN 3 128 300 32
Accuracy RUN 3 128 300 32
Loss RUN 3 128 300 64
Accuracy RUN 3 128 300 64
Loss RUN 3 128 300 128
Accuracy RUN 3 128 300 128
Loss RUN 3 128 400 32
Accuracy RUN 3 128 400 32
Loss RUN 3 128 400 64
Accuracy RUN 3 128 400 64
Loss RUN 3 128 400 128
Accuracy RUN 3 128 400 128
train loss
val loss
train acc
val acc



Conclusion of the parameter exploration is

- most of the two layer options do not overfit (or at least very late)
- three layers perform over all better
- at that point increasing epochs is not an issue any more to get overfitting
- overfitting can be achieved with many different unit size and batch size combinations
- the fluctuations of accuracy and loss increase with higher unit and batch values

There are many parameter combinations to start with. I've chosen a run that doesn't show too much fluctuation and resonable fast overfitting : **RUN 3 32 400 32**, meaning 3 layers, 32 units, 400 epochs, batch size 32.

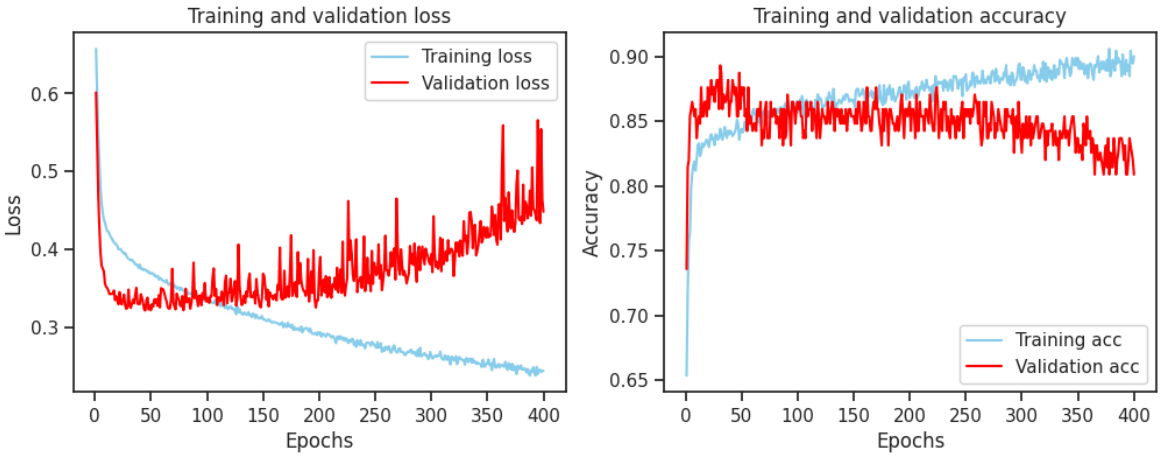
These setting will be used for further exploration.

From here on out the NN model build, compile, fit and output in the code cells is no longer commented as this has been done several times before already and the changes are explained in the text cell next to the code.

In [122…

```
model = keras.Sequential([
    layers.Dense(32, activation='relu'),
    layers.Dense(32, activation='relu'),
    layers.Dense(32, activation='relu'),
    layers.Dense(1, activation='sigmoid')
])

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

history = model.fit(
    partial_x_train,
    partial_y_train,
    epochs=400,
    batch_size=32,
    validation_data=(x_val, y_val),
    verbose=0
)

print(np.max(history.history['accuracy']))
print(np.argmin(history.history["val_loss"]), 'epochs')
plot_loss_acc(history.history)
```

```
0.9060308337211609
44 epochs
```



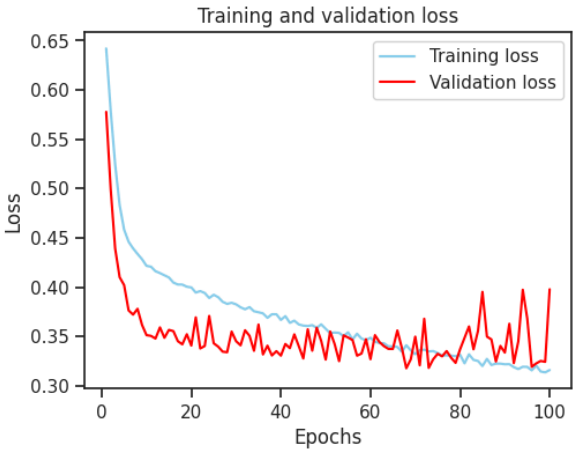
Regularize and Tune

Following the universal workflow of Chollet I begin by tuning the model. The previous analysis showed that three layers performed better than two. So the first step is to add a fourth and fifth layer and see how it performs then.

In [127…

```
model = keras.Sequential([
    layers.Dense(32, activation='relu'),
    layers.Dense(32, activation='relu'),
    layers.Dense(32, activation='relu'),
    layers.Dense(32, activation='relu'),
    layers.Dense(1, activation='sigmoid')
])

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

history = model.fit(
    partial_x_train,
    partial_y_train,
    epochs=100,
    batch_size=32,
    validation_data=(x_val, y_val),
    verbose=0
)

print(np.max(history.history['accuracy']))
print(np.argmin(history.history["val_loss"]), 'epochs')
plot_loss_acc(history.history)
```

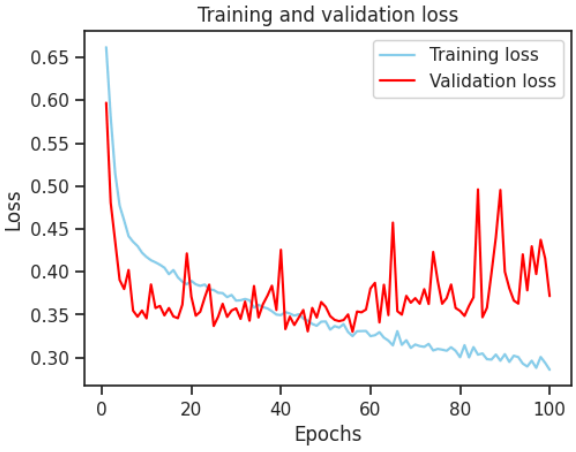
```
0.8765778541564941
67 epochs
```

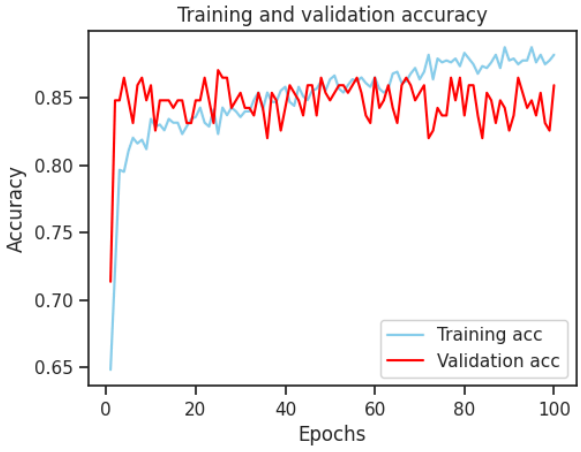


In [129…

```
model = keras.Sequential([
    layers.Dense(32, activation='relu'),
    layers.Dense(32, activation='relu'),
    layers.Dense(32, activation='relu'),
    layers.Dense(32, activation='relu'),
    layers.Dense(32, activation='relu'),
    layers.Dense(1, activation='sigmoid')
])

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

history = model.fit(
    partial_x_train,
    partial_y_train,
    epochs=100,
    batch_size=32,
    validation_data=(x_val, y_val),
    verbose=0
)

print(np.max(history.history['accuracy']))
print(np.argmin(history.history["val_loss"]), 'epochs')
plot_loss_acc(history.history)
```

```
0.887798011302948
55 epochs
```





Running the fit for less epochs obviously reduces the mean accuracy a little bit. Over all it seems more layers perform better than less. Having a fifth layer seems to be an advantage. This will be the baesline layer setup from now on.

Chollet suggeststst regularization for small models. This might be helpful with this data set so an L2 regularization is set up.

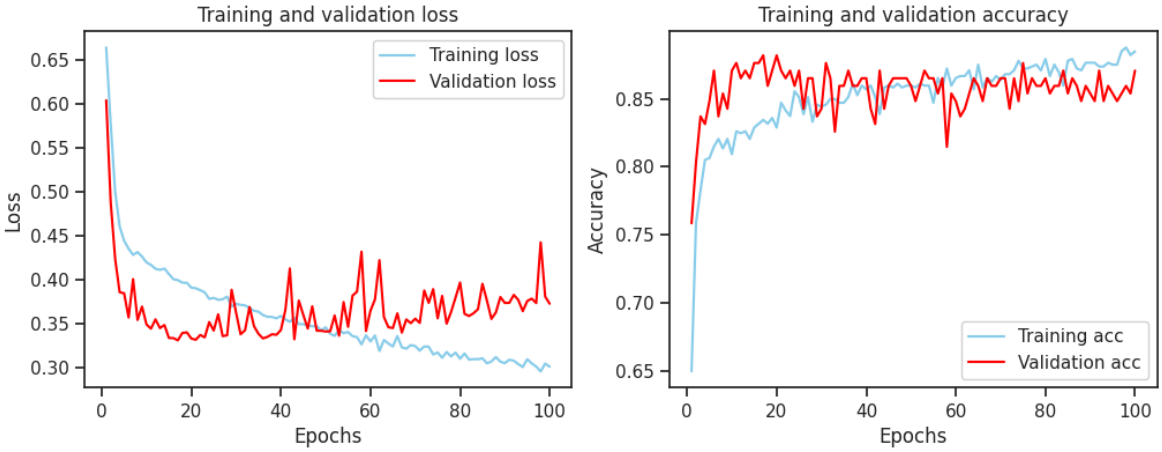
In [131…
```
from tensorflow.keras import regularizers

model = keras.Sequential([
    layers.Dense(32, activation='relu', kernel_regularizer=regularizers.l
    layers.Dense(32, activation='relu', kernel_regularizer=regularizers.l
    layers.Dense(32, activation='relu', kernel_regularizer=regularizers.l
    layers.Dense(32, activation='relu', kernel_regularizer=regularizers.l
    layers.Dense(32, activation='relu', kernel_regularizer=regularizers.l
    layers.Dense(1, activation='sigmoid')
])

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

history = model.fit(
    partial_x_train,
    partial_y_train,
    epochs=100,
    batch_size=32,
    validation_data=(x_val, y_val),
    verbose=0
)

print(np.max(history.history['accuracy']))
print(np.argmin(history.history["val_loss"]), 'epochs')
plot_loss_acc(history.history)
```

```
0.887798011302948
34 epochs
```

In [135…
```
from tensorflow.keras import regularizers

model = keras.Sequential([
    layers.Dense(32, activation='relu', kernel_regularizer=regularizers.l
```

```
    layers.Dense(32, activation='relu', kernel_regularizer=regularizers.l
    layers.Dense(32, activation='relu', kernel_regularizer=regularizers.l
    layers.Dense(32, activation='relu', kernel_regularizer=regularizers.l
    layers.Dense(32, activation='relu', kernel_regularizer=regularizers.l
    layers.Dense(1, activation='sigmoid')
])

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

history = model.fit(
    partial_x_train,
    partial_y_train,
    epochs=100,
    batch_size=32,
    validation_data=(x_val, y_val),
    verbose=0
)

print(np.max(history.history['accuracy']))
print(np.argmin(history.history["val_loss"]), 'epochs')
plot_loss_acc(history.history)
```

```
0.887798011302948
16 epochs
```



Different weights for the regularizer have been tested. Only the last two successfull tests are included in the report. Weights higher than 0.0006 will result in reduced accuracy. 0.0002 seems to be a good value to reduce the number of epochs necessary until overfitting occurs. Alternating between 0.0002 and 0.0001 needs even less training epochs but at that point the accuracy could be better. So I'll stick with 0.0002 for all weights.

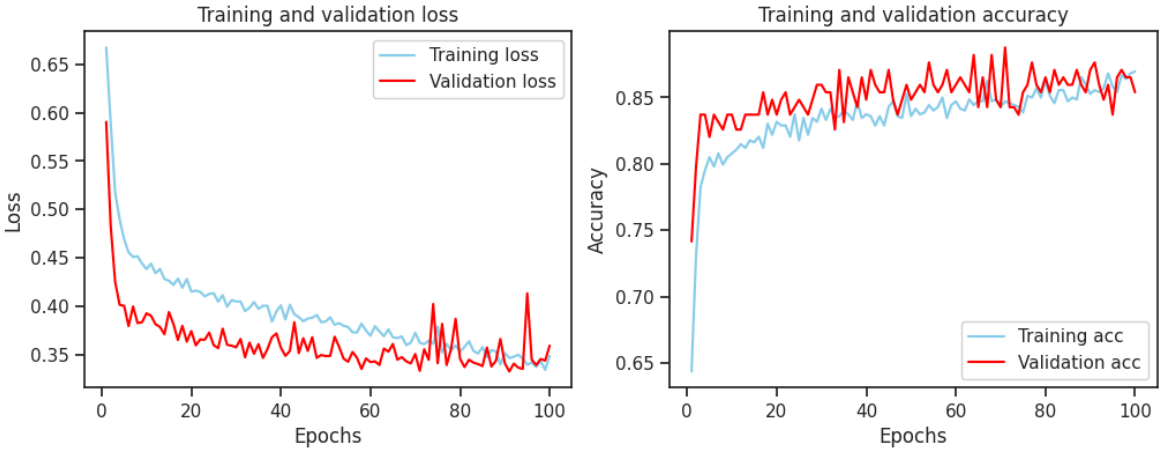
The next step is to add a drop out layer and see if that has a positive effect.

In [134…

```
from tensorflow.keras import regularizers

d = 0.01

model = keras.Sequential([
    layers.Dense(32, activation='relu', kernel_regularizer=regularizers.l
```

```
    layers.Dropout(d),
    layers.Dense(32, activation='relu', kernel_regularizer=regularizers.l
    layers.Dropout(d),
    layers.Dense(32, activation='relu', kernel_regularizer=regularizers.l
    layers.Dropout(d),
    layers.Dense(32, activation='relu', kernel_regularizer=regularizers.l
    layers.Dropout(d),
    layers.Dense(32, activation='relu', kernel_regularizer=regularizers.l
    layers.Dropout(d),
    layers.Dense(1, activation='sigmoid')
])

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

history = model.fit(
    partial_x_train,
    partial_y_train,
    epochs=100,
    batch_size=32,
    validation_data=(x_val, y_val),
    verbose=0
)

print(np.max(history.history['accuracy']))
print(np.argmin(history.history["val_loss"]), 'epochs')
plot_loss_acc(history.history)
```

```
0.8695651888847351
90 epochs
```



Multiple drop out values have been tested. Only one is shown. In all cases the number of epochs until overfitting occurs is raised significantly. Instead of using dropout I'll continue trying to advance the existing setup with the regularizer using a different optimizer.

In [145…

```
from tensorflow.keras import regularizers

model = keras.Sequential([
    layers.Dense(32, activation='relu', kernel_regularizer=regularizers.l
    layers.Dense(32, activation='relu', kernel_regularizer=regularizers.l
    layers.Dense(32, activation='relu', kernel_regularizer=regularizers.l
```

```
    layers.Dense(32, activation='relu', kernel_regularizer=regularizers.l
    layers.Dense(32, activation='relu', kernel_regularizer=regularizers.l
    layers.Dense(1, activation='sigmoid')
])

opt = keras.optimizers.Adam(learning_rate=0.001)

model.compile(
    optimizer=opt,
    loss='binary_crossentropy',
    metrics=['accuracy']
)

history = model.fit(
    partial_x_train,
    partial_y_train,
    epochs=100,
    batch_size=32,
    validation_data=(x_val, y_val),
    verbose=0
)

print(np.max(history.history['accuracy']))
print(np.argmin(history.history["val_loss"]), 'epochs')
plot_loss_acc(history.history)
```

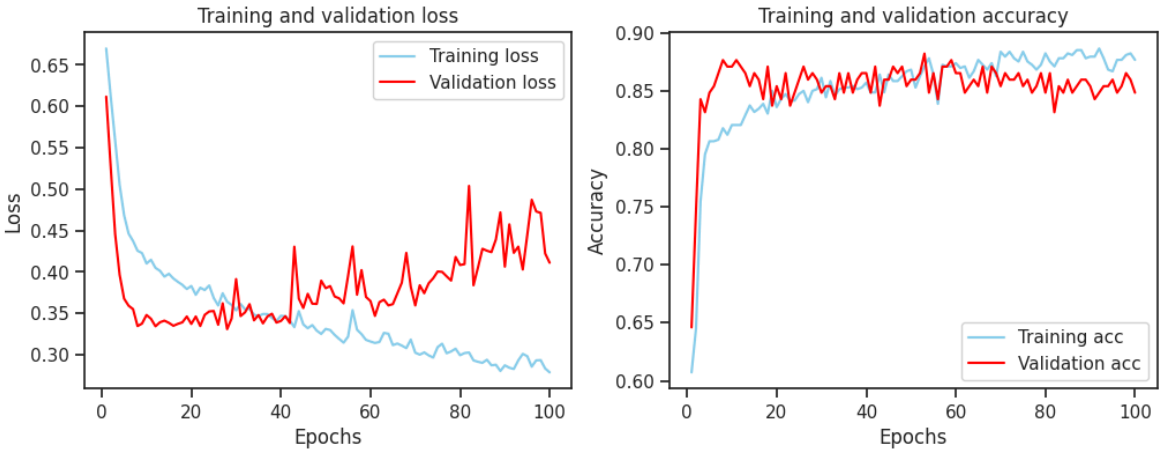
```
0.8920056223869324
46 epochs
```

In [160…

```
model = keras.Sequential([
    layers.Dense(32, activation='relu'),
    layers.Dense(32, activation='relu'),
    layers.Dense(32, activation='relu'),
    layers.Dense(32, activation='relu'),
    layers.Dense(32, activation='relu'),
    layers.Dense(1, activation='sigmoid')
])

opt = keras.optimizers.Adam(learning_rate=0.001)

model.compile(
    optimizer=opt,
    loss='binary_crossentropy',
    metrics=['accuracy']
)
```

```
history = model.fit(
    partial_x_train,
    partial_y_train,
    epochs=100,
    batch_size=32,
    validation_data=(x_val, y_val),
    verbose=0
)

print(np.max(history.history['accuracy']))
print(np.argmin(history.history["val_loss"]), 'epochs')
plot_loss_acc(history.history)
```

```
0.8863955140113831
27 epochs
```



Testing different optimizers (only Adam is shown) with different learning rates did not reveal significant improvements. Using Adam and L2 regularization overfitting occurs around epoch 46. Removing the L2 parameter reduces the runtime till overfitting to 27 epochs. So this easier model using Adam as it performs equal to RMSprop is used for the final fit.

Train the Final Model

The final model is trained on the whole training set without splitting a validation set off.

This is then used to predict the testing set surviver feature. The resultig array is used to create a new dataframe containing the *PassengerId* and the *Survived* data which is then submitted to the **Kaggle** competition for evaluation.

Usually the test set would contain the necessary target information for accuracy calculation but since this is a competition this evaluation step is done online using the **Kaggle** portal.

In my first submission to **Kaggle** I trained the model for 30 epoch as the last overfitting occured around that value. It scored 0.74.

Increasing the epochs raised the score first slightly and after setting the training periode to 300 epochs to 0.755.

In [17]:
```
# remove Survived feature which should not be learned but predicted
x_train = train_df.drop(labels='Survived', axis=1)
# only store the Survived feature
y_train = train_df['Survived']

model = keras.Sequential([
    layers.Dense(32, activation='relu'),
    layers.Dense(32, activation='relu'),
    layers.Dense(32, activation='relu'),
    layers.Dense(32, activation='relu'),
    layers.Dense(32, activation='relu'),
    layers.Dense(1, activation='sigmoid')
])

opt = keras.optimizers.Adam(learning_rate=0.001)

model.compile(
    optimizer=opt,
    loss='binary_crossentropy',
    metrics=['accuracy']
)

history = model.fit(
    x_train,
    y_train,
    epochs=300,
    batch_size=32,
)

fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(12, 4))

acc = history.history["accuracy"]
loss_values = history.history["loss"]
epochs = range(1, len(loss_values) + 1)
ax.plot(epochs, loss_values, label="Training loss", color="skyblue")
ax.plot(epochs, acc, label="Training acc", color="red")
ax.set_title("Training and validation loss")
ax.set_xlabel("Epochs")
ax.set_ylabel("Loss")
ax.legend()

fig.show()
```

Deploy the Model

The fully trained model is used to predict the survivers and the results of the prediction are sent to **Kaggle** for evaluation.

The input for evaluation needs to be in a CSV file with certain format that is created in the following steps.

In [14]:
```
# drop the PassengerId column for the prediction
x_test = test_df.drop(labels='PassengerId', axis=1)

# predict the survivor chance
prediction = model.predict(x_test)
```

```
14/14 [==============================] - 0s 2ms/step
```

In [15]:
```
# create a new dataframe containing the passengerid and survivor chance
# the prediction is rounded (0 or 1) and converted to an integer
survivers = pd.concat([test_df['PassengerId'], pd.DataFrame(prediction, c
survivers
```

Out[15]:

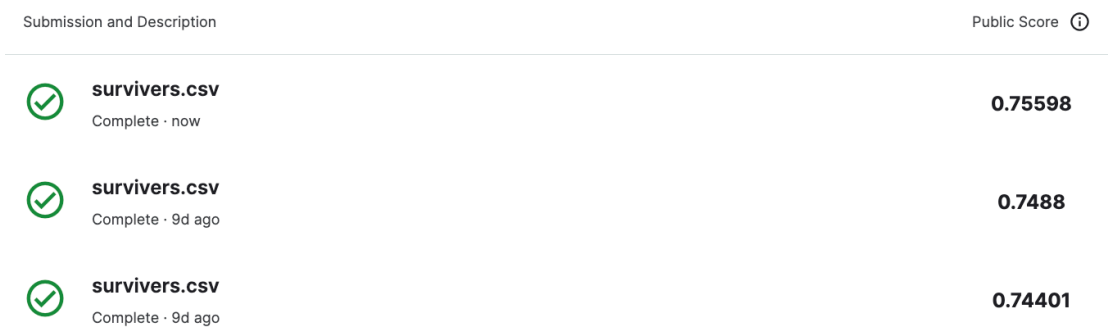
	PassengerId	Survived
0	892	0
1	893	0
2	894	0
3	895	0
4	896	0
...	...	...
413	1305	0
414	1306	1
415	1307	0
416	1308	0
417	1309	1

418 rows × 2 columns

In [16]:
```
# export the dataframe as CSV for submission
survivers.to_csv('survivers.csv', index=False)
```

Results

The trained model reached a prediction score of **0.76** on **Kaggle**. Certainly well below the top submissions but as the competition documentation states everything above 0.7 is an achievement worth mentioning.



The training score was obviously higher. Reason for this is that the network learned the training set well. Maybe too well considering the test set should not be signifficantly different from the training set. The lower accuracy is probably a combination of using holdout instead of k-fold and tuning the parameters better. Maybe even using layers we're not supposed to use for this project.

For another attempt I would use a k-fold crossvalidation approach but for my first ML project with a previously unknown and not pre prepared dataset I'm happy to be above the **Kaggle** threshold of success.

As final step let's check if the predicated survial rate is in the range of the baseline:

In [34]:
```
# count all test set entries
total = survivers['Survived'].count()
# count all predicted survivors in that set
num_survivers = survivers['Survived'][survivers['Survived'] > 0].count()

# calculate the survival rate in %
survival_rate = int(round(num_survivers / total * 100, 0))
print('Survial Rate: ' + str(survival_rate) + '%')
```

```
Survial Rate: 33%
```

The answer is yes. The baseline established through statistial analysis of actual boat accidents estimates 32% survival rate. The analysis of the output of the trained model shows a rate of 33% which matches pretty well. Overall I would call that a success.

References

Code and ideas for data interpretation and processing came in part from Chollet and

[1]: Predicting the Survival of Titanic Passengers

[2]: Titanic Survival Prediction Using Machine Learning