Text classifier for Sentiment analysis using a Bi-directional Recurrent Neural Network

Introduction

Recurrent neural networks (RNNs) were introduced in the 1990s in response to a variety of challenging problems, including motion detection, temporal sequences of events, and ordered data, such as characters in words [1]. As part of ongoing research, Schuster and Kuldip (1997) introduced the idea of bidirectional communication between the hidden layers of an RNN. Such a network can be trained simultaneously in both positive and negative time directions, removing the constraint of using input information only up to a predefined future frame [2].

While large financial and research institutions publish their insights on market movements in the form of discrete datasets, the feelings and opinions of retail investors are often difficult to assess on a large scale due to their distributed nature. However, research has shown that sentiment analysis (SA) can be used to extract important metrics for retail investors, such as systematic correlation (individuals buy or sell assets in a coordinated manner) [3].

Financial social media have been a phenomenon of the last 15 years and offer an exciting opportunity to apply SA at scale. SA is a method commonly used to assess users' sentiments in social media applications such as Twitter [4] and use them to build machine learning models and classifiers that can later be used to further analyse various financial market datasets.

Objectives of the project

The first goal is to find social media sentiment datasets suitable for neural network training and ensure that they contain texts that are correctly labelled (as positive or negative sentiment). The resulting text corpora must then be processed to improve the model's results by applying techniques such as word stemming, contraction substitution, and tokenization.

The next major challenge is to design the bidirectional recurrent neural network (BRNN) itself by selecting appropriate hidden layers, placing them in the correct order, and choosing appropriate hyperparameters. The output of this network should be a floating point number indicating that the input corpus belongs to either a positive or negative sentiment class, and a measure of the certainty of the finding (a higher or lower number would indicate higher certainty).

The algorithm should also be able to store the model locally during the training process to ensure that the best scoring model and hyperparameters are preserved.

Finally, I will also compare the performance of this model, as measured by its accuracy, to the existing Native Bayes text classifier from the NLTK library.

Setup

Import modules

In [1]:
```
import numpy as np
import pandas as pd
import tensorflow as tf
import plotly.express as px
from IPython.display import display
from pathlib import Path
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer
from nltk.stem import *
from sklearn.model_selection import train_test_split
from google.colab import drive
from psutil import virtual_memory
import re
import string
import random
import chardet
import warnings
```

Environment

In [2]:
```
# Mount Google drive as a persistent storage.
# This is to overcome ephemeral nature of Google colab
drive.mount('/content/drive')
```

```
# Make sure we utilize GPU acceleration
gpu_info = !nvidia-smi
gpu_info = '\n'.join(gpu_info)
if gpu_info.find('failed') >= 0:
  print('Not connected to a GPU')
else:
  print('Connected to a GPU:')
  print(gpu_info)

ram = virtual_memory().total / 1e9
print('This VM has {:.1f}GB of RAM available'.format(ram))

warnings.filterwarnings('ignore')
```

```
Mounted at /content/drive
Connected to a GPU:
Wed Jan  4 18:39:20 2023
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 460.32.03    Driver Version: 460.32.03    CUDA Version: 11.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  A100-SXM4-40GB      Off  | 00000000:00:04.0 Off |                    0 |
| N/A   32C    P0    51W / 400W |      0MiB / 40536MiB |      0%      Default |
|                               |                      |             Disabled |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
This VM has 89.6GB of RAM available
```

Download required NLTK data sets

In [3]:
```
nltk.download('stopwords')
nltk.download('punkt')
nltk.download('wordnet')
nltk.download('omw-1.4')
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt.zip.
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data] Downloading package omw-1.4 to /root/nltk_data...
```

Out[3]: True

Ensure required directories exist

In [4]:
```
for dir in ['data', 'models/checkpoint']:
  Path(dir).mkdir(parents=True, exist_ok=True)
```

Classes

In [5]:
```
# A simple wrapper class to apply various text cleanup steps
class RegExReplacer(object):
    def __init__(self):
        patterns = [
            # Replace contractions with extended forms
            (r"won\'t", "will not"),
            (r"can\'t", "can not"),
            (r"i\'m", "i am"),
            (r"(\w+)\'ve", "\g<1> have"),
            (r"(\w+)\'ll", "\g<1> will"),
            (r"(\w+)n\'t", "\g<1> not"),
            (r"(\w+)\'re", "\g<1> are"),
            (r"(\w+)\'s", "\g<1> is"),
            # Dataset-specific cleanup
            (r"\'s", ""),
            (r"user", ""),
        ]
        self.patterns = [(re.compile(regex), repl) for (regex, repl) in patterns]

    def replace(self, text):
```

```
        for (pattern, repl) in self.patterns:
            text = re.sub(pattern, repl, text)

        return text
```

Variables

In [6]:
```
validation_set_size = 0.4
vocab_size = 4000

rep_word = RegExReplacer()
wnl = WordNetLemmatizer()
stemmer = PorterStemmer()
stop_words = stopwords.words('english')
```

Description of the selected dataset

I originally planned to train the NN with one of the available general sentiment analysis datasets. For example, Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank dataset by Richard Socher et al. (2014) [5] or the Large Movie Review Dataset by Andrew Maas et al [6], which is directly available in Tensorflow. However, models trained on such datasets showed rather poor accuracy when applied to financial corpora. For this reason, I used two datasets specifically compiled for sentiment analysis in finance.

The Financial Phrase Bank was created by Malo et al. (2014) contains financial news headlines and their sentiments from the perspective of retail investors. This dataset contains 4,837 in CSV format.

The second dataset, compiled by Yash Chaudhary (2020), focuses on retail investors' sentiments about U.S. stock market tickers. Both datasets contain text in English language.

These datasets were processed into a data frame with two main columns:

1. Text (corpus), which contains the processed corpora, and
2. Label, which indicates the positive or negative sentiment.

To achieve the best training results, the following processing steps were applied to the corpora.

1. Invalid records were removed (e.g., if they contained only a number or a ticker symbol).
2. Punctuation was removed.
3. Text was lowercased and trimmed.
4. URLs were removed.
5. English abbreviations were expanded.
6. Stop words were removed.
7. Word stemming was applied.
8. The text was tokenized (word-based).

In total, the processed dataset contains 7,758 rows.

Fetch data sets

In [7]:
```
# Initialize first column names and order
cols = ['text', 'label']
```

Financial Phrase Bank [6]

In [8]:
```
# Downlaod the dataset CSV into a Pandas DataFrame
phrasebank_df = pd.read_csv('https://s3.eu-west-2.amazonaws.com/pavdev.io/nlp/all-data.csv', encoding='Windows-

# Add column names
phrasebank_df.columns = ['label', 'text']

# Only pick rows with positive or negative sentiment
phrasebank_df = phrasebank_df[phrasebank_df.label != 'neutral']

# Convert string values into numerical labels
mapping = {'positive': 1, 'negative': 0}
phrasebank_df.label = [mapping[item] for item in phrasebank_df.label]

# Reorder columns to be consistent with other data sets
phrasebank_df = phrasebank_df[cols]

# Preview the data
phrasebank_df.head()
```

Out[8]:

	text	label
1	The international electronic industry company ...	0
2	With the new production plant the company woul...	1
3	According to the company 's updated strategy f...	1
4	FINANCING OF ASPOCOMP 'S GROWTH Aspocomp is ag...	1
5	For the last quarter of 2010 , Componenta 's n...	1

Stock-Market Sentiment Dataset [7]

In [9]:
```
# Downlaod the dataset CSV into a Pandas DataFrame
stock_df = pd.read_csv('https://s3.eu-west-2.amazonaws.com/pavdev.io/nlp/stock_data.csv')

# Convert label values to be consistent with other data sets
stock_df.replace(to_replace=-1, value=0, inplace=True)

# Add column names, columns are already ordered correctly
stock_df.columns = ['text', 'label']

# Preview the data
stock_df.head()
```

Out[9]:

	text	label
0	Kickers on my watchlist XIDE TIT SOQ PNK CPW B...	1
1	user: AAP MOVIE. 55% return for the FEA/GEED i...	1
2	user I'd be afraid to short AMZN - they are lo...	1
3	MNTA Over 12.00	1
4	OI Over 21.37	1

Combine data sets

In [10]:
```
# Combine data sets
df = pd.concat([phrasebank_df, stock_df])

# Randomly shuffle the rows
df = df.sample(frac=1).reset_index(drop=True)

# Cache the resulting DF locally
df.to_csv('data/set.csv')

df.head()
```

Out[10]:

	text	label
0	APO Q1 2013 operational cash flow 210 million....	0
1	NKD looking distressed. Anticipating a gap dow...	0
2	Productional situation has now improved .	1
3	I remain convinced you're either in Apple stoc...	1
4	Kesko 's car import and retailing business , V...	1

Pre-processing

In [11]:
```
# Note: each operation is done separately here for readability
# The dataset is not very large (~8000 rows), so performance is not of major concern
def custom_tokenize(text):
    # The dataset unfortunately contains some invalid values
    if not isinstance(text, str):
        return ""

    # Remove punctuation
    punct_table = dict((ord(c), None) for c in string.punctuation)
    text = text.translate(punct_table)

    text = text.lower() # Lower case the text
    text = text.strip()  # Trim the text
    text = re.sub(r"http\S+", "", text) # Remove URLs
    text = rep_word.replace(text) # Expand english contractions (e.g.: aren't => are not) and conduct dataset sp
    words = word_tokenize(text) # Actual word-based tokenization
    words = [w for w in words if not w in stop_words] # Remove stop words
```

```
    # Stemming, somewhat surprisingly, gives better results than lemmatization
    # words = [wnl.lemmatize(word.lower()) for word in words]
    words = [stemmer.stem(word) for word in words]

    return words
```

In [12]:
```
df['tokenized'] = df['text'].apply(custom_tokenize)
df['processed_text'] = df['tokenized'].apply(lambda word_list : ' '.join(word_list))

# Filter out rows where only empty string was left after the cleanup
df = df[df['processed_text'] != ""]
```

Preview original and processed texts

In [13]:
```
df[['text', 'processed_text']].head(10)
```

Out[13]:

	text	processed_text
0	APO Q1 2013 operational cash flow 210 million....	apo q1 2013 oper cash flow 210 million 90milli...
1	NKD looking distressed. Anticipating a gap dow...	nkd look distress anticip gap come day close t...
2	Productional situation has now improved .	product situat improv
3	I remain convinced you're either in Apple stoc...	remain convinc your either appl stock aap anot...
4	Kesko 's car import and retailing business , V...	kesko car import retail busi vvauto saw sale g...
5	For financial year 2019-20 if the independent ...	financi year 201920 independ director compani ...
6	Post: How ong Can You Hold Your Breath? SHD DE	post ong hold breath shd de
7	However , the total orders received will still...	howev total order receiv still last year level
8	AAP a classic move to 435 testing on no cause....	aap classic move 435 test caus webinar follow ...
9	Finnish meat company Atria can no longer promi...	finnish meat compani atria longer promis suffi...

The dataset contains ~7800 rows with positive and negative sentiment distributed roughly evenly.

In [14]:
```
df.describe().transpose()
```

Out[14]:

	count	mean	std	min	25%	50%	75%	max
label	7758.0	0.650683	0.476785	0.0	0.0	1.0	1.0	1.0

Generate volabulary

In [15]:
```
# Concatenate and flatten tokens
words = np.concatenate(df['tokenized']).ravel()
unique_words = np.unique(words)

# Print out number of words and unique words
print('All words: {}'.format(len(words)))
print('Unique words: {}'.format(len(unique_words)))
```

```
All words: 81449
Unique words: 12047
```

In [16]:
```
# Display most and least common words
pd.value_counts(np.array(words))
```

Out[16]:
```
aap          925
eur          774
short        503
mn           465
profit       463
            ...
endang         1
212150         1
companâ€¦      1
anadarko       1
17201710       1
Length: 12047, dtype: int64
```

Evaluation methodology

There are many ways to measure the performance of the BRNN. I have chosen to use model accuracy and loss, which are commonly used in benchmarks published by other authors and are readily available in both TensorFlow and NLTK (which was used as the baseline measurement). However, I recognise that opinions vary on this topic and that there are a number of other metrics that could also be used. For example, an F1 score or mean squared error.

I used NLTK's existing Naive Bayes classifier as the basis for evaluating our RNN model. The input features were identical to those shown in class: simple Python dict in the format `contains(feature): boolean`. The reasons for choosing this classifier are that it is a standard and well-tested algorithm and that it provides suitable performance metrics, as mentioned above.

To ensure valid readings, both classifiers were given an identical input data frame.

The dataset was split into training and validation datasets. The validation dataset was quite large (40%), but this was necessary, bacause otherwise there would not have been sufficient amount of data for each validation step during the training of the NN model.

In the case of the BRNN, I tracked both loss and accuracy across all epochs. This is to ensure that both metrics are trending in the right direction (up for accuracy and down for loss). For example, it is common for the loss to trend upward after a certain point because the hyperparameter for the training rate is too high.

Baseline performance (Naive Bayes classifier)

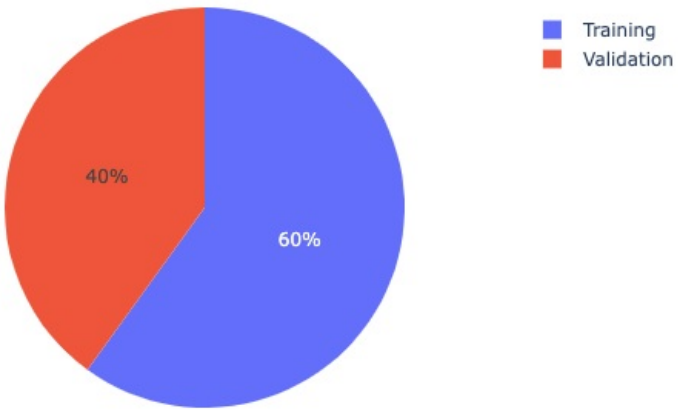
Extract Features (Words)

In [17]:
```
# Only use top <vocab_size> words to limit dataset dimensionality (and make the model train in a reasonable tim
word_features = list(unique_words)[:vocab_size]


# The following snippet is an adapted version of code shown in the Coursera learning environment for this modul
# Week 10: Sentiment analysis using supervised learning (Video)
def extract_features(word_list):
    word_set = set(word_list)  # Take only unique words in the text

    # Create a dict of format 'contains(<feature>)': bool
    features = {}

    for word in word_features:
        features["contains({})".format(word)] = (word in word_set)

    return features


featuresets = [
    (extract_features(row["tokenized"]), row["label"]) for index, row in df.iterrows()
]
```

Split the dataset into training and validation subsets

In [18]:
```
train_set, test_set = train_test_split(featuresets, test_size=validation_set_size, random_state=42)
```

In [19]:
```
split_df = pd.DataFrame({
    "Subset": ['Training', 'Validation'],
    "Words": [len(train_set), len(test_set)]
})

fig = px.pie(split_df, values='Words', names='Subset', width=600, height=400)
fig.show()
```



Create and evaluate classifiers

In [20]:
```
# Initialize lists for the results DataFrame
models = []
results = []
```

Naive Bayes

In [21]:
```
# Train the classifier
nb_classifier = nltk.NaiveBayesClassifier.train(train_set)

# Calculate acuracy on the validation dataset
nb_accuracy = nltk.classify.accuracy(nb_classifier, test_set)

models.append('Naive Bayes')
results.append(nb_accuracy)

print('Naive Bayes accuracy: {:.0%}'.format(nb_accuracy))
```

```
Naive Bayes accuracy: 67%
```

Classification approach - Bi-directional RNN text classifier with TensorFlow

Input pipeline

In [22]:
```
# Generate TF dataset
ds = tf.data.Dataset.from_tensor_slices(
    (
        df['processed_text'],
        df['label'],
    )
)
```

Train-test split

In [23]:
```
test_rows_no = round(len(list(ds)) * validation_set_size)

# Random shuffle and batching
# This generates (text, label) tuples
buffer = 10000
batch = 64

train_dataset = ds.skip(test_rows_no).shuffle(buffer).batch(buffer).prefetch(tf.data.AUTOTUNE)
test_dataset = ds.take(test_rows_no).batch(batch).prefetch(tf.data.AUTOTUNE)
```

Create a vectorizer (first hidden layer in RNN)

Keras conveniently provides a number of layers, which makes creating the NN considerably easier. The TextVectorization layer in maps text features to integer sequences, which the subsequent layers can "understand".

In [24]:
```
vectorizer = tf.keras.layers.TextVectorization(max_tokens=40000)
vectorizer.adapt(train_dataset.map(lambda text, label: text))
```

The vocabulary is unsurprisingly almost the same as our custom list generated for NLTK, although it also adds `[UNK]` word (by default) as a fallback for when evaluated words are not present in the vocabulary.

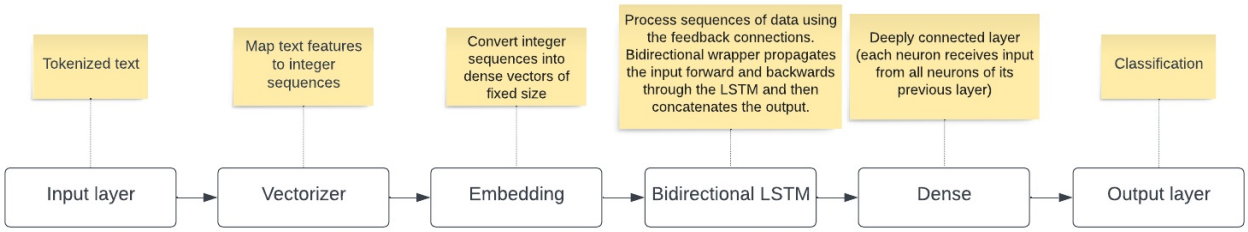
In [25]:
```
np.array(vectorizer.get_vocabulary())[:20]
```

Out[25]:
```
array(['', '[UNK]', 'aap', 'eur', 'short', 'profit', 'compani', 'stock',
       'mn', 'today', 'market', 'year', 'sale', 'oper', 'like', 'look',
       'day', 'volum', 'net', 'million'], dtype='<U30')
```

Creating the model

The structure of the BRNN is visualised in the diagram below along with the description of each layer. I used pre-built layers available in Keras module due to them being well-tested and proven in production settings. I have put a substantial amount of effort into fine-tuning model's and optimizers's parameters and hyperparameters, especially selecting the correct:

1. Learning rates to prevent "overshooting" during the gradient descent.
2. Activation function. This was largely a trial and error process, but ended up using Rectified linear unit (ReLU), bacause it produced the best results.
3. Number of neurons, to prevent over-fitting.



In [26]:
```
# Create the model
model = tf.keras.Sequential([
    vectorizer,
    tf.keras.layers.Embedding(input_dim=len(vectorizer.get_vocabulary()), output_dim=64, mask_zero=True),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(128)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(1)
])

# Compile it
model.compile(
    loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
    optimizer=tf.keras.optimizers.Adam(0.001), # 0.00001, 0.001, 0.0001
    metrics=['accuracy']
)
```

Training the RNN

In [27]:
```
train_epochs = 23

# Save the model if accuracy increased after an epoch
model_checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(
    filepath='models/checkpoint',
    save_weights_only=False,
    monitor='val_accuracy',
    mode='max',
    save_best_only=True
)

# Fit the model
history = model.fit(train_dataset, epochs=train_epochs, validation_data=test_dataset, validation_steps=30, callb
```

```
Epoch 1/23
1/1 [==============================] - ETA: 0s - loss: 0.6939 - accuracy: 0.3564
1/1 [==============================] - 39s 39s/step - loss: 0.6939 - accuracy: 0.3564 - val_loss: 0.6904 - val_
accuracy: 0.3417
Epoch 2/23
1/1 [==============================] - 0s 256ms/step - loss: 0.6907 - accuracy: 0.3564 - val_loss: 0.6870 - val
_accuracy: 0.3417
Epoch 3/23
1/1 [==============================] - 0s 250ms/step - loss: 0.6874 - accuracy: 0.3564 - val_loss: 0.6827 - val
_accuracy: 0.3417
Epoch 4/23
1/1 [==============================] - 0s 241ms/step - loss: 0.6835 - accuracy: 0.3564 - val_loss: 0.6776 - val
_accuracy: 0.3417
Epoch 5/23
1/1 [==============================] - 0s 238ms/step - loss: 0.6786 - accuracy: 0.3564 - val_loss: 0.6712 - val
_accuracy: 0.3417
Epoch 6/23
1/1 [==============================] - 0s 239ms/step - loss: 0.6725 - accuracy: 0.3564 - val_loss: 0.6635 - val
_accuracy: 0.3417
Epoch 7/23
1/1 [==============================] - 0s 243ms/step - loss: 0.6650 - accuracy: 0.3564 - val_loss: 0.6544 - val
_accuracy: 0.3417
Epoch 8/23
1/1 [==============================] - ETA: 0s - loss: 0.6559 - accuracy: 0.3564
1/1 [==============================] - 22s 22s/step - loss: 0.6559 - accuracy: 0.3564 - val_loss: 0.6442 - val_
accuracy: 0.4057
Epoch 9/23
1/1 [==============================] - ETA: 0s - loss: 0.6453 - accuracy: 0.4159
1/1 [==============================] - 23s 23s/step - loss: 0.6453 - accuracy: 0.4159 - val_loss: 0.6341 - val_
accuracy: 0.5682
Epoch 10/23
1/1 [==============================] - ETA: 0s - loss: 0.6344 - accuracy: 0.5968
```

```
1/1 [==============================] - 23s 23s/step - loss: 0.6344 - accuracy: 0.5968 - val_loss: 0.6278 - val_
accuracy: 0.6151

Epoch 11/23
1/1 [==============================] - ETA: 0s - loss: 0.6262 - accuracy: 0.6438

1/1 [==============================] - 22s 22s/step - loss: 0.6262 - accuracy: 0.6438 - val_loss: 0.6286 - val_
accuracy: 0.6286
Epoch 12/23
1/1 [==============================] - ETA: 0s - loss: 0.6232 - accuracy: 0.6561

1/1 [==============================] - 22s 22s/step - loss: 0.6232 - accuracy: 0.6561 - val_loss: 0.6264 - val_
accuracy: 0.6479
Epoch 13/23
1/1 [==============================] - ETA: 0s - loss: 0.6149 - accuracy: 0.6711

1/1 [==============================] - 23s 23s/step - loss: 0.6149 - accuracy: 0.6711 - val_loss: 0.6184 - val_
accuracy: 0.6536
Epoch 14/23
1/1 [==============================] - ETA: 0s - loss: 0.6001 - accuracy: 0.6930

1/1 [==============================] - 22s 22s/step - loss: 0.6001 - accuracy: 0.6930 - val_loss: 0.6088 - val_
accuracy: 0.6635
Epoch 15/23
1/1 [==============================] - ETA: 0s - loss: 0.5835 - accuracy: 0.7255

1/1 [==============================] - 23s 23s/step - loss: 0.5835 - accuracy: 0.7255 - val_loss: 0.5996 - val_
accuracy: 0.6812
Epoch 16/23
1/1 [==============================] - ETA: 0s - loss: 0.5668 - accuracy: 0.7665

1/1 [==============================] - 22s 22s/step - loss: 0.5668 - accuracy: 0.7665 - val_loss: 0.5904 - val_
accuracy: 0.7021
Epoch 17/23
1/1 [==============================] - ETA: 0s - loss: 0.5482 - accuracy: 0.8095

1/1 [==============================] - 22s 22s/step - loss: 0.5482 - accuracy: 0.8095 - val_loss: 0.5799 - val_
accuracy: 0.7193
Epoch 18/23
1/1 [==============================] - ETA: 0s - loss: 0.5258 - accuracy: 0.8402

1/1 [==============================] - 23s 23s/step - loss: 0.5258 - accuracy: 0.8402 - val_loss: 0.5678 - val_
accuracy: 0.7297
Epoch 19/23
1/1 [==============================] - ETA: 0s - loss: 0.4992 - accuracy: 0.8623

1/1 [==============================] - 23s 23s/step - loss: 0.4992 - accuracy: 0.8623 - val_loss: 0.5553 - val_
accuracy: 0.7307
Epoch 20/23
1/1 [==============================] - ETA: 0s - loss: 0.4688 - accuracy: 0.8767

1/1 [==============================] - 23s 23s/step - loss: 0.4688 - accuracy: 0.8767 - val_loss: 0.5443 - val_
accuracy: 0.7349
Epoch 21/23
1/1 [==============================] - ETA: 0s - loss: 0.4352 - accuracy: 0.8881

1/1 [==============================] - 23s 23s/step - loss: 0.4352 - accuracy: 0.8881 - val_loss: 0.5385 - val_
accuracy: 0.7401
Epoch 22/23
1/1 [==============================] - ETA: 0s - loss: 0.3989 - accuracy: 0.8969

1/1 [==============================] - 23s 23s/step - loss: 0.3989 - accuracy: 0.8969 - val_loss: 0.5421 - val_
accuracy: 0.7417
Epoch 23/23
1/1 [==============================] - ETA: 0s - loss: 0.3613 - accuracy: 0.9050

1/1 [==============================] - 23s 23s/step - loss: 0.3613 - accuracy: 0.9050 - val_loss: 0.5579 - val_
accuracy: 0.7495
```

Evaluation

BRNN performance

In [28]:
```
test_loss, test_acc = model.evaluate(test_dataset)

print('Validation loss:', test_loss)
print('Validation accuracy:', test_acc)
```

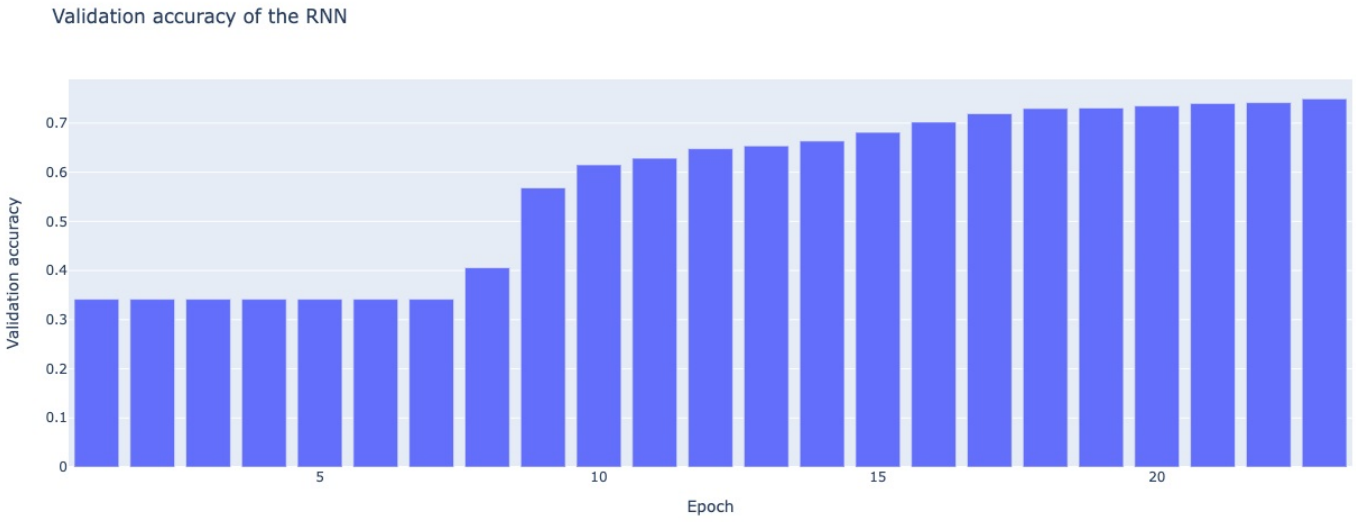
```
49/49 [==============================] - 0s 5ms/step - loss: 0.5504 - accuracy: 0.7483
Validation loss: 0.5504173040390015
Validation accuracy: 0.7483080625534058
```

In [29]:
```
res_df = pd.DataFrame({
    'Epoch': np.arange(1, len(history.history['val_accuracy']) + 1),
    'Validation accuracy': history.history['val_accuracy'],
    'Validation loss': history.history['val_loss'],
```

```
})

display(res_df)
```

	Epoch	Validation accuracy	Validation loss
0	1	0.341667	0.690427
1	2	0.341667	0.686952
2	3	0.341667	0.682746
3	4	0.341667	0.677552
4	5	0.341667	0.671194
5	6	0.341667	0.663497
6	7	0.341667	0.654403
7	8	0.405729	0.644194
8	9	0.568229	0.634127
9	10	0.615104	0.627788
10	11	0.628646	0.628631
11	12	0.647917	0.626408
12	13	0.653646	0.618403
13	14	0.663542	0.608793
14	15	0.681250	0.599642
15	16	0.702083	0.590376
16	17	0.719271	0.579870
17	18	0.729688	0.567843
18	19	0.730729	0.555312
19	20	0.734896	0.544275
20	21	0.740104	0.538541
21	22	0.741667	0.542086
22	23	0.749479	0.557946

In [30]:
```
fig = px.bar(res_df, x='Epoch', y='Validation accuracy', title="Validation accuracy of the RNN")
fig.show()
```



Comparing the accuracy to the baseline (Naive Bayes classifier)

The model was able to achieve accuracy of 75% on the validation dataset, beating the Naive Bayes classifier in this metric by roughly 11%. We can see that the accuracy improvements are uneven, which can be explained by the still not perfectly optimized learning rate. The model was able to achieve even higher accuracy in validation data when trained on a larger number of epochs (> 85% for 40 epochs), but only at the cost of increased loss (> 70%). Despite considerable effort in this area and fine-tuning the learning rate of the model and the number of neurons in each layer, the loss began to increase roughly after the 22nd epoch.

Conclusions

I successfully implemented a bidirectional recurrent neural network specialised to evaluate the sentiments of retail investors with respect

to the financial market conditions. The model achieved a satisfactory accuracy of 75% with a relatively high loss of 55%. This means that the model made a relatively small number of large errors, which is makes sense given the discrete nature of the results. The BRNN was able to outperform the prebuilt Naive Bayes text classifier by a significant margin of 11%.

I found that the main limiting factor is the quality of the input data and not the performance of the neural network itself, with a possible exception described below. While there are a large number of datasets dealing with financial sentiment, many contain corpora that are clearly mislabeled or contain other errors, such as off-topic text.

Up to roughly 22nd epoch, the model's loss descreases almost linearly, which is a good indication. However I did notice it starts to increase afterwards. Due to the limited time available, I was not able to determine the root cause of this behaviour and this area could be a good starting point for subsequent research.

References

[1] Medsker, Larry R., and L. C. Jain. "Recurrent neural networks." Design and Applications 5 (2001): 64-67.

[2] Schuster, Mike, and Kuldip K. Paliwal. "Bidirectional recurrent neural networks." IEEE transactions on Signal Processing 45.11 (1997): 2673-2681.

[3] Kumar, Alok, and Charles MC Lee. "Retail investor sentiment and return comovements." The Journal of Finance 61.5 (2006): 2451-2486.

[4] Sohangir, Sahar, et al. "Big Data: Deep Learning for financial sentiment analysis." Journal of Big Data 5.1 (2018): 1-25.

[5] Socher, Richard, et al. "Recursive deep models for semantic compositionality over a sentiment treebank." Proceedings of the 2013 conference on empirical methods in natural language processing. 2013.

[6] Maas, Andrew, et al. "Learning word vectors for sentiment analysis." Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies. 2011.

[7] Yash Chaudhary, 2020, *Stock-Market Sentiment Dataset*, https://www.kaggle.com/dsv/1217821

In [1]:
```
!export PATH=/Library/TeX/texbin:$PATH
```

In [ ]:
```

```