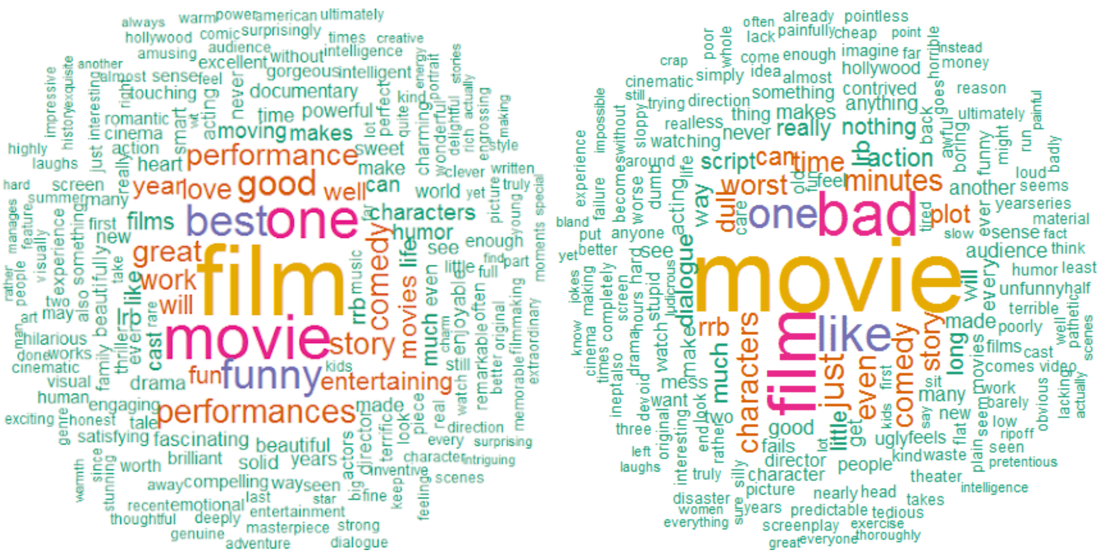
Using keras to classify movie reviews
Mathematical Programming with Python

MATH 2604: Advanced Scientific Computing 4
Spring 2025
Monday/Wednesday/Friday, 1:00-1:50pm
Room A202 Langley Hall

https://people.sc.fsu.edu/∼jburkardt/classes/python_2025/keras/keras.pdf



Words used in positive and negative movie reviews.

Classifying with keras

- We start with a set of movie reviews, each one classified as posiive or negative;
- We want to create a neural network able to examine new movie reviews and guess whether they are also positive or negative;
- We train it on some reviews, and then test it on the remainder.
- This is supervised learning - we want the network to "learn" how to classify reviews like we did.

1 What do we want?

When NetFlix recommends movies to you, it is using an automated system that has classified the movies in its library, and has also analyzed your own likes and dislikes. It then lists movies that you have not yet watched, but which most closely fit your apparent preferences.

We are going to experiment with a similar, but simpler task. We have collected the text of many movie reviews, and labeled them as *positive* or *negative*. How we did this is something of a mystery that would be very hard to express as a computer program. Some (but not all!) of our judgment could be explained simply by the choice of words in each review. If we encounter the words "miserable", or "boring" or "stupid", we are likely to be reading a negative review.

Now suppose we have a pile of movie reviews, and we have read each of them, and assigned them a rating of "positive" or "negative". Now imagine we hand the reviews and their ratings to a very intelligent Martian, who can see the words, but doesn't know what they mean. The Martian might notice that there is a rough relationship between the words and the classification: words like *great* and *thilling* may indicate a positive review, while *awful* would suggest the opposite. However, a bad movie can be called *a great disappointment*, so if we decided that *great* by itself was a sure indication of a positive review, we would soon notice some problems. To build a good classifier, we would have to review the data, make some tentative rules, test them, and then adjust our model to try to reduce the errors.

Finding patterns in data is something that the keras package does well. To deal with the movie review classification problem, we can use keras to build a neural network, apply it to our data to get a model, and then test the model to see how well it performs on new data.

The movie review dataset is a built-in feature of keras, and so we can take advantage of a large set of data which has already been labeled. This will allow us to create models and see how effective they are.

2 A reference for keras

You can find an introduction to keras in the book:

```
Francois Chollet,
Deep Learning with Python,
Second Edition,
Manning, 2021,
ISBN: 9781617296864
https://www.manning.com/books/deep-learning-with-python-second-edition
```

The IMDB movie review exercise is covered in section 4.1, *Classifying movie reviews: a binary classification example.*

3 What is the IMDB data and how is it used?

The Internet Movie Database (IMDB) dataset consists of 50,000 movie reviews and 50,000 labels. Each review is labeled "0" if it was judged to be negative, or "1" if positive. These labels were supplied by humans who read the reviews.

Our goal is to come up with a procedure that can automatically produce a label for a movie review, and which will closely match the behavior of human readers. We are to do this by using a computer procedure which can see, but not understand, the words of the reviews. In other words, as far as we are concerned, these reviews could have been written in Polish, or Chinese, or Egyptian hieroglyphics.

A dictionary was created from all the words in all the reviews, and each word in that dictionary was given a numerical index. Then each review was used to generate a corresponding file of numbers, where each word was replaced by its index. Now, instead of English, each review can be transformed into a sequence of numbers, which refer to the words in the dictionary.

Thus the text of a movie review is now numeric. For technical reasons, we wish to consider only the 10,000 most common words, so each numeric file is modified to eliminate unusual words. When we actually process a movie review, we do one last step: we replace the file of numbers by a vector of length 10,000, where entry i of the vector is set to 1 if word[i] appeared at least once in the review. The reason for doing this is that the neural network needs to process vectors of a uniform size. We keep the neural network happy by making every movie review a 10,000 entry vector of 0's and 1's. We say that we have *vectorized* a review in this way.

We divide the data into three sets: training, validation, and testing data. We will build a model with the training data, and then use it to predict the labels on the validation data. The prediction failures will be used to adjust the model. We will do this adjustment a fixed number of times (perhaps 10 or 20 "epochs") and then declare the model ready for testing.

We now hold the model fixed, and try it out on the testing data. If the training procedure was done well, then the model should have good accuracy in predicting the labels for the testing data. If the model does poorly, then we must go back and adjust our model and repeat the entire process.

4 Peeking at the reviews

The movie reviews in the IMDB dataset are no longer humanly readable; they are just lists of numbers. However, it is possible to use the dataset to decode the numeric review back into a semi-readable version by replacing each number by the corresponding word. The file *imdb_decode.py* can be used this way, where the input argument specifies the index of the movie review you want to decode.

```
python3
from imdb_decode import imdb_decode
imdb_decode(7)

? lavish production values and solid performances in this straightforward adaption of jane ?
    satirical classic about the marriage game within and between the classes in ? 18th
   century england northam and paltrow are a ? mixture as friends who must pass through ?
   and lies to discover that they love each other good humor is a ? virtue which goes a
   long way towards explaining the ? of the aged source material which has been toned down
   a bit in its harsh ? i liked the look of the film and how shots were set up and i
   thought it didn't rely too much on ? of head shots like most other films of the 80s and
   90s do very good results
```

The question marks in the listing indicate unusual words that were not in the top 10,000 most common. You should be able to guess that this review is labeled 1 **positive**. We hope that our movie classifier will also be able to correctly label it.

5 Running the movie review script

We have several options for running the movie review example with keras:

1. on your laptop, but you must install keras and some other libraries, see `https://keras.io/getting_started/`;
2. using Google Colab, which has the necessary libraries already, see `https://colab.research.google.com/`;

You may find it tricky to install keras on your laptop. However, if you can get it set up, you may prefer to do your work there. The movie review is large, but not enormous, so it should run fairly quickly for you.

Installation information can be found on the keras website `keras.io`.

Mac and Linux users can try these install commands:

```
sudo pip install tensorflow
sudo pip install keras
```

while Windows users may try to install with:

```
pip install tensorflow
pip install keras
```

Whether you want to work directly on your laptop, or through Google Colab, the next step is to download the file *movie_review.py* from the class website. You can execute it with a command like:

```
python3 movie_review.py
```

On Mac and Linux, you can save the “interesting” output to a separate text file:

```
python3 movie_review.py > movie_review.txt
```

6 Understanding the code

The algorithm implemented in `movie_review.py` uses these steps:

1. Access and Load the data;
2. Prepare it for use;
3. Describe the model;
4. Create the model;
5. Use the model on training and validation data;
6. Evaluate the model on new test data;

We will not worry about how the data is loaded and prepared, except to note that keras will need to download the data from the Internet if you are running on your laptop. It will then rearrange and split the data so that it has the right shape for the neural network, and is divided into training, validation, and test sets.

```
import keras
import numpy as np
import tensorflow

from tensorflow.keras.datasets import imdb

( train_data , train_labels ), ( test_data , test_labels ) = \
  imdb.load_data ( num_words = 5000 )
```

Our data is long lists in which the words of a review have been replaced by index values. But we need to replace each such list by a vector that simply records whether each word in our dictionary has been used in this review. We create a function to do this. We also have to make sure our review labels look like floating point numbers.

```
def vectorize_sequences ( sequences , dimension = 5000 ):
  results = np.zeros((len(sequences), dimension))
  for i , sequence in enumerate(sequences):
    results[i , sequence] = 1.0
  return results

x_train = vectorize_sequences ( train_data )
x_test = vectorize_sequences ( test_data )

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

The model is a sequence of layers, with two hidden layers, each with 16 hidden units, and a `relu` activation. The first layer expects an input vector of length 5,000; in other words, one of our movie reviews, stored as a vector of 0’s and 1’s. Our output layer uses the `sigmoid` function, which returns a value between 0 and 1, the probability that the movie review is negative or positive.

```
model = keras.Sequential ( [ \
  layers.Dense ( 16, activation = 'relu' ), \
  layers.Dense ( 16, activation = 'relu' ), \
```

```
    layers.Dense ( 1, activation = 'sigmoid') \
  ] )
```

The model is created by choosing an optimizer, loss function, and a metric. The `rmsprop` optimizer seeks to minimize the root mean square of the error. The `binary_crossentropy` loss function is a way of measuring the difference between two probability distributions.

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

We split out the first 5,000 entries of our training data for validation.

```
  x_val = x_train[:5000]
  partial_x_train = x_train[5000:]

  y_val = y_train[:5000]
  partial_y_train = y_train[5000:]
```

Now we train the model through 20 iterations. We save the intermediate results as `history` so we can look at our progress:

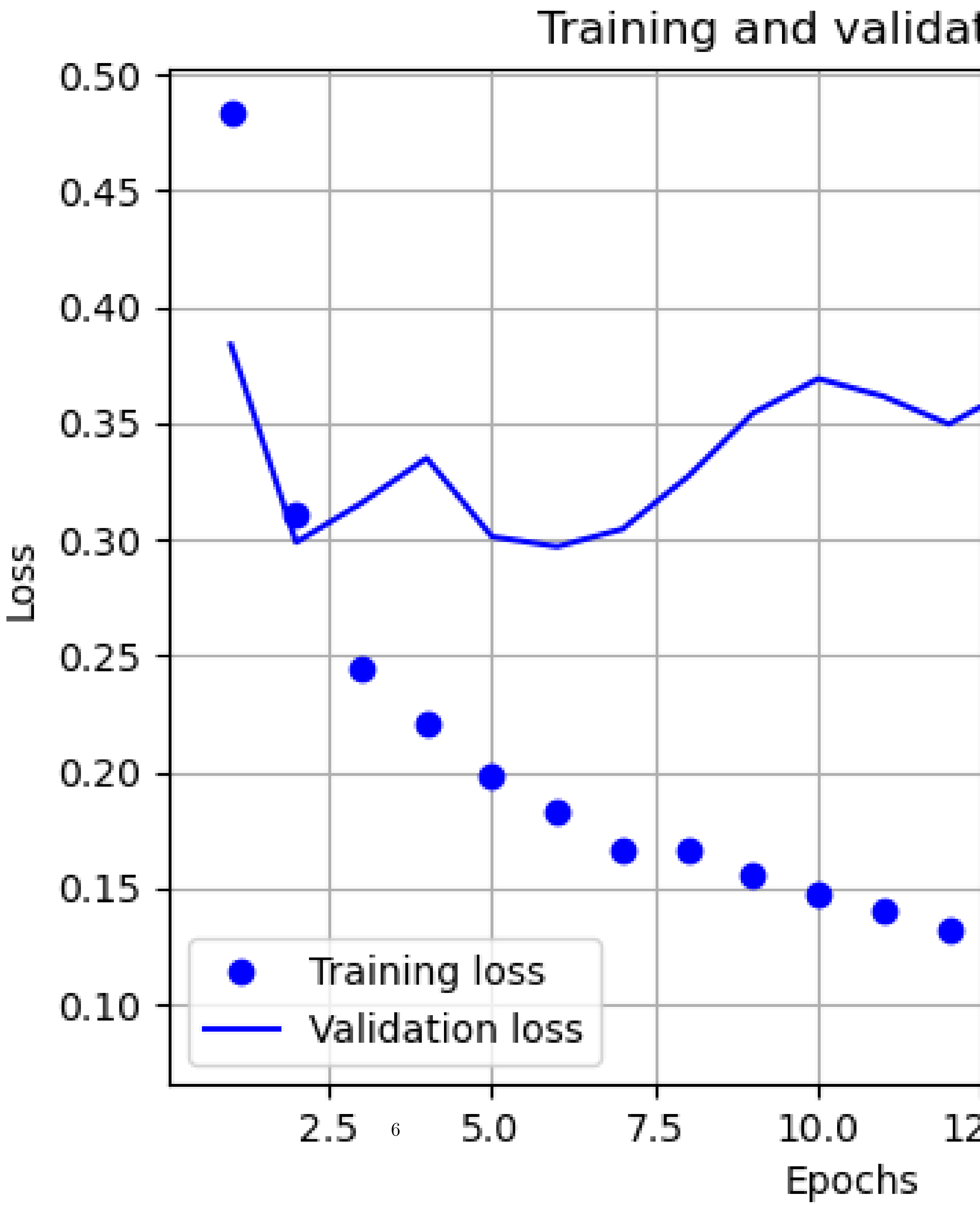
```
  history = model.fit ( \
    partial_x_train,
    partial_y_train,
    epochs = 20,
    batch_size = 512,
    validation_data = ( x_val, y_val ) )
```

Let's look at how well the model was able to handle the validation data:

```
  val_acc = history_dict['val_accuracy']
  val_loss = history_dict['val_loss']
  print ( '    Final validation loss', val_loss[-1] )
  print ( '    Final validation accuracy', val_acc[-1] )
```

Because we saved the history, we can plot the behavior of the loss and accuracy functions on our training and validation data. Here is how we plot the loss information:

```
  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' )
```


Training and validat
0.50
0.45
0.40
0.35
0.30
0.25
0.20
0.15
0.10
Loss
Training loss
Validation loss
2.5
5.0
7.5
10.0
12
Epochs

From the plots, it looks we would be better off stopping after 4 iterations. We can rerun the model. This time, we use the entire training set, and we don't ask for validation:

```
  model.fit ( x_train , y_train , epochs = 4, batch_size = 512 )
```

And now we can apply our model to our test data:

```
  results = model.evaluate ( x_test , y_test )
  print ( '' )
  print ( '  Model loss and accuracy on test data:' )
  for i in range ( len ( model.metrics_names ) ):
    print ( model.metrics_names[i], results[i] )
```

The test loss and test accuracy are printed out something like this:

```
  Model loss and accuracy on test data:
loss 0.3238627934074402
accuracy 0.87308
```

This result suggest that our model has 87% accuracy in judging whether a movie review is positive or negative, based on word usage alone, and without understanding what any of the words mean.