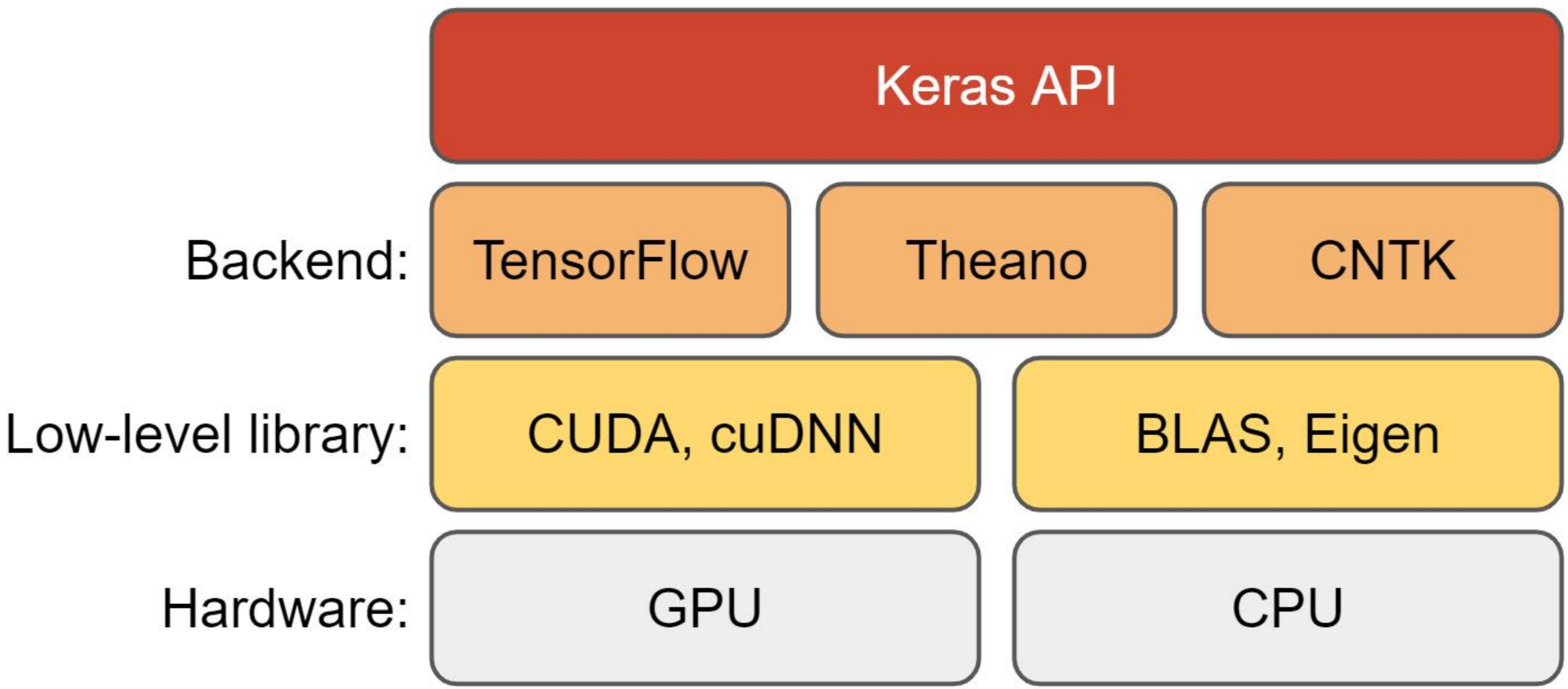
Hands-on with Neural Networks

The Programming Stack



Note

You can import Keras as

- Standalone module:

  ```
  import keras
  ```

  which used to rely on different backends. From v2.4 only TensorFlow is supported

- As part of TensorFlow:

  ```
  from tensorflow import keras
  ```

  in this case the only backend being used is TensorFlow

Discussion - I

- High-level library we will use to program: Keras
- Keras may work on different backends, e.g. Tensorflow (most popular), Theano (almost abandoned), or CNTK
- There is an implementation of Keras part of TensorFlow, only working with TensorFlow
- Low-level libraries mainly used to create customized models
- We will mainly use TensorFlow directly for handling some data structures

Discussion - II

- The low-level libraries handle optimized parallel computing and linear algebra on specific pieces of hardware

- Among other, CUDA (Compute Unified Device Architecture) is the infrastructure used to program GPUs

- The entire stack, depending on the low-level library, may work on CPUs or GPUs, and even on mobile devices.

Sentiment Analysis with ANN

A first example

Simple Example

- In this first example we will use ANN with a very simple representation, i.e., the bag-of-words we have used until now

- However, this is not the best representation, and we can do much better with ANN if we use embeddings (later)

Data Preparation

```
import pandas as pd
from sklearn.preprocessing import LabelEncoder
dataset=pd.read_csv("IMDB Dataset.csv")
print(dataset.describe())
print(dataset['sentiment'].value_counts())

dataset['review'] = dataset['review'].map(transformText)

le = LabelEncoder()
le.fit(dataset['sentiment'])
dataset['sentiment']=le.transform(dataset['sentiment'])

dataset.to_csv("IMDB_Table.csv",index=False)
```

Discussion

- We use the preprocessText function written before
- First problem: with ANN, we can't keep the dependent variable as string
- We have different ways to encode it
  - Integer Encoding (suitable for linear outputs or when there are two categories)
  - One-Hot Encoding (for multiple categories)
- We store the preprocessed data to save time and then continue from here

Integer vs. One-Hot Encoding

- a=[“black”, “white”, “black”]

- Integer encoding: each value is represented as an integer

  [0, 1, 0]

- One-hot encoding: we have a [0,1] output for each category, indicating whether the datum belongs to that category

  [[1,0], [0, 1], [1, 0])

Encoding - Example

```
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
import numpy as np

a=np.array(["black","white","black"])

le=LabelEncoder()
a_le=le.fit_transform(np.array(a))
print(a_le)

#We need to reshape a to apply one-hot encoding
a_reshaped = a.reshape(len(a), 1)
oh=OneHotEncoder()
a_oh=oh.fit_transform(a_reshaped)
print(a_oh)
```

Result:

```
[0 1 0]
```

```
[[1. 0.]
 [0. 1.]
 [1. 0.]]
```

Data preparation

```
import pandas as pd

dataset=pd.read_csv("IMDB_Table.csv")

from sklearn.model_selection import train_test_split
X_trainAll, X_test, y_trainAll, y_test = train_test_split(dataset['review'], dataset['sentiment'],
                                                          test_size=0.10, random_state=10)

X_train, X_valid, y_train, y_valid = train_test_split(X_trainAll, y_trainAll,
                                                                test_size=0.20, random_state=10)

#Build the counting corpus
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfTransformer
count_vect = CountVectorizer(min_df=30)
tfidf_transformer = TfidfTransformer()

X_train = count_vect.fit_transform(X_train)
X_train = tfidf_transformer.fit_transform(X_train).toarray()

X_valid=count_vect.transform(X_valid)
X_valid=tfidf_transformer.transform(X_valid).toarray()

X_test=count_vect.transform(X_test)
X_test=tfidf_transformer.transform(X_test).toarray()
```

Discussion

- We use CountVectorizer (taking only words appearing in at least 30 documents, but you can change that) and TfIdfTransformer
  - You could also try to apply other preprocessing, e.g., feature selection
- Then (**IMPORTANT FOR NEURAL NETWORKS!**) we need to split the data into Train, Validation and Test
  - We do this by using the train_test_split function twice (first we get the test set out, then the validation set out)
  - We use the following (typical) percentages: 70%, 20%, 10%
- Finally, we transform data as usual
  - However, ANN do not accept the sparse representation produced by the TfIdfTransformer, therefore we need to **convert it into an array**

Creating the model

```
input_dim = X_train.shape[1]  # Number of features

model = Sequential()
model.add(layers.Input(shape=(input_dim,)))
model.add(layers.Dense(100, activation='relu'))
model.add(layers.Dense(100, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

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

model.summary()
```

Discussion - I

- The model is a Sequential model (each layer follows the previous one)
- We first instantiate the model, then we add layers to it
- The network has three layers:
  - Two dense layers (each note connected to each receiving input) composed of 100 nodes, and with a relu activation
  - An output node (single node, sigmoid activation)

Discussion - II

- Relu is chosen as an activation for the internal layers as it works much better during the training (for following the gradient)
- The output uses a sigmoid as it serves for the classification
- After creating the model, we compile it specifying
  - The loss function: binary_crossentropy
  - The optimizer: adam in this case, but we could have used a stochastic gradient descent (`optimizer="SGD"`)
  - The metric to optimize: accuracy, but it could be precision, recall,…

Parameters

- We have seen the cross-entropy as loss measure and stochastic gradient descent as optimizer

- However, there are alternative measures and optimizers (in particular some optimizers are found to go much better than gradient descent)

- As for the metrics, as usual you could optimize accuracy, but also other metrics

- For details, see:

  - https://keras.io/losses

  - https://keras.io/optimizers

  - https://keras.io/metrics

Model Topology

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 100)	835,500
dense_1 (Dense)	(None, 100)	10,100
dense_2 (Dense)	(None, 1)	101

Total params: 845,701 (3.23 MB)

Trainable params: 845,701 (3.23 MB)

Non-trainable params: 0 (0.00 B)

Weights to evaluate

The total number of weights is given, in this case, by:

- The number of inputs ((vocabulary size) + 1 (bias)) * number of nodes first layer = (8350+1)*100

- The number of outputs of the 1st layer + 1 (bias) multiplied by the nodes of the second layer: (100+1)*100

- The number of outputs of the 2nd layer +1 (bias): 100+1

- Total: (8350+1)*100+101*100+101=845301

Training and testing the model

```
history = model.fit(X_train, y_train, epochs=10, verbose=True,
                    validation_data=(X_valid, y_valid), batch_size=10)


loss, accuracy = model.evaluate(X_train, y_train, verbose=False)
print("Training Accuracy: {:.4f}".format(accuracy))

loss, accuracy = model.evaluate(X_test, y_test, verbose=True)
print("Testing Accuracy:  {:.4f}".format(accuracy))
```

Parameters

- Batch size (number of samples used to train the network at each step)
    - If you have 1000 samples, the algorithm takes the first 100 and trains the network, then other 100, etc.
    - Mini batches require less memory
    - Too fast might easily produce overfit
- Number of epochs (number of passes through the training set): you can reduce when you see an early convergence

Training result…

Epoch 1/10

3600/3600 [==============================] - 10s 3ms/step - loss: 0.3065 - accuracy: 0.8691 - val_loss: 0.2774 - val_accuracy: 0.8821

Epoch 2/10

3600/3600 [==============================] - 10s 3ms/step - loss: 0.1988 - accuracy: 0.9193 - val_loss: 0.2926 - val_accuracy: 0.8801

Epoch 3/10

3600/3600 [==============================] - 10s 3ms/step - loss: 0.1097 - accuracy: 0.9586 - val_loss: 0.4147 - val_accuracy: 0.8746

Epoch 4/10

3600/3600 [==============================] - 9s 3ms/step - loss: 0.0308 - accuracy: 0.9896 - val_loss: 0.6421 - val_accuracy: 0.8680

Epoch 5/10

3600/3600 [==============================] - 9s 3ms/step - loss: 0.0073 - accuracy: 0.9977 - val_loss: 0.9989 - val_accuracy: 0.8699

Epoch 6/10

3600/3600 [==============================] - 9s 3ms/step - loss: 0.0056 - accuracy: 0.9984 - val_loss: 1.1988 - val_accuracy: 0.8724

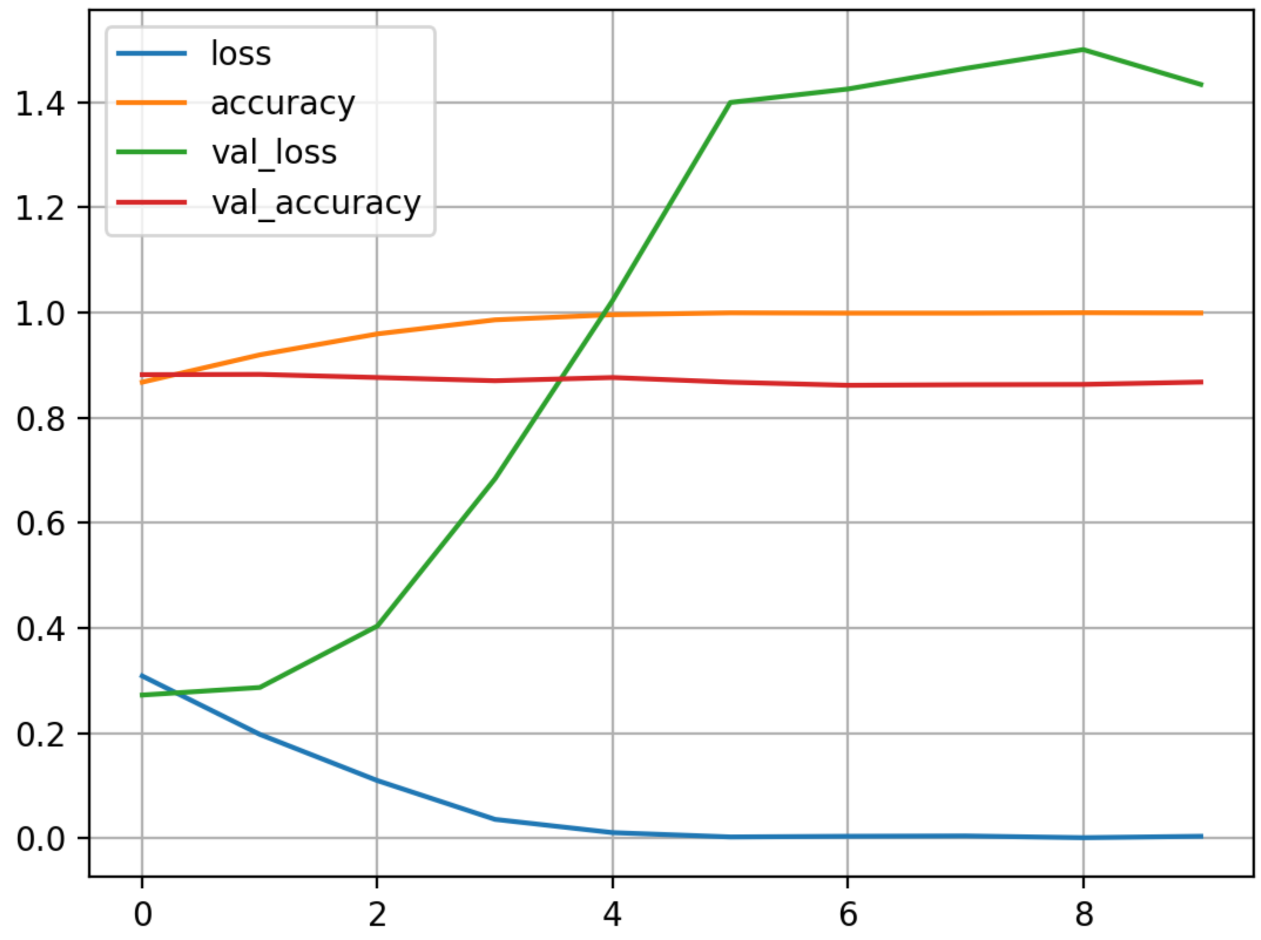
Epoch 7/10

3600/3600 [==============================] - 9s 3ms/step - loss: 0.0020 - accuracy: 0.9993 - val_loss: 1.3140 - val_accuracy: 0.8700

Epoch 8/10

Plot

```
import matplotlib.pyplot as plt
pd.DataFrame(history.history).plot()
plt.grid(True)
plt.show()
```



Metrics on the test set

- Keras does not have features for metric reporting
- Therefore, we need to convert the output back to the classification (in our case is a simple Boolean, we will then see how to do it with multiple classes) using a threshold

Printing metrics

```
#Prediction metrics
from sklearn.metrics import classification_report

y_pred = model.predict(X_test, verbose=1)
pred_threshold=0.5
print("Y pred",y_pred)
print("Y test",y_test)
y_pred_bool = [int(x+0.5) for [x] in y_pred]


print(y_pred_bool)
print(classification_report(y_test, y_pred_bool))
```

From here, you can also produce the AUC, the confusion matrix, etc.

What happens?

- loss for validation set increase over time
- accuracy tops for training set → overfit
- Not good!

How to avoid overfitting?

Early stopping:

- keeping the peak accuracy value for the validation set
- patience: stopping iterations over epochs if the model does not improve for a given number of generations

Dropout:

- Avoiding to train some random nodes during training steps

Saving the best model for the validation set

```
from tensorflow.keras import callbacks
from tensorflow.keras import models
checkpoint_cb = callbacks.ModelCheckpoint("my_keras_model.keras", save_best_only=True)

history = model.fit(X_train, y_train, epochs=10,
                    validation_data=(X_valid, y_valid),
                    callbacks=[checkpoint_cb])
model = models.load_model("my_keras_model.keras") # rollback to best model)
```

- We use keras callbacks, i.e., functions that are invoked during the fitting phase

- Using the ModelCheckpoint callback, over the epochs, best models for the validation are saved in a file

- Then, the best model is retrieved to be used on the test set

Note

- Saving trained models, and then loading them to use on test sets (e.g., in production is a common practice when using machine learning in real world)

- To save a model, given a variable storing it (`model`) you can use:
```
model.save('path/to/location')
```

Early stopping

- The previous solution still requires to run over all epochs
- In our example we considered very few epochs, but this is not true in most of the real cases, and for deep neural networks
- In addition, we can use the “patience” callback
- If the performance on the validation does not improve for a given number of epochs, then the process stops

Early stopping: source code

```
checkpoint_cb = callbacks.ModelCheckpoint("my_keras_model.keras", save_best_only=True)

early_stopping_cb = callbacks.EarlyStopping(patience=5,
                                                  restore_best_weights=True)
history = model.fit(X_train, y_train, epochs=100,
                    validation_data=(X_valid, y_valid),
                    callbacks=[checkpoint_cb, early_stopping_cb])
```

- The fitting stops if there is no improvement in the validation for 10 epochs

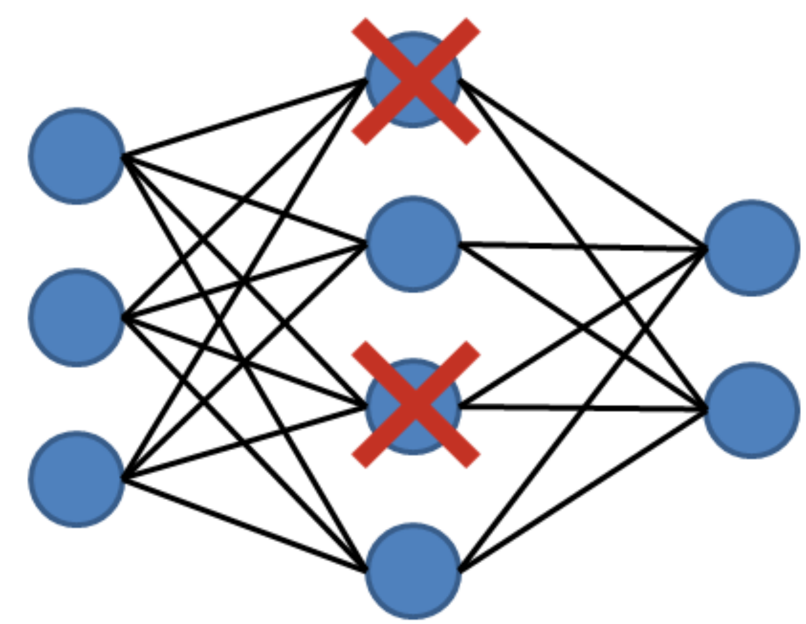
- In any case, the best value is retained

Another regularization technique: dropout

- Every neuron, during a training step, will have a probability p of being ignored, and hence its weight not adjusted

- Typically, this probability is between 20%-50% (20%-30% for deep neural networks)

- Important: of course, dropout layers are not used during the evaluation phase



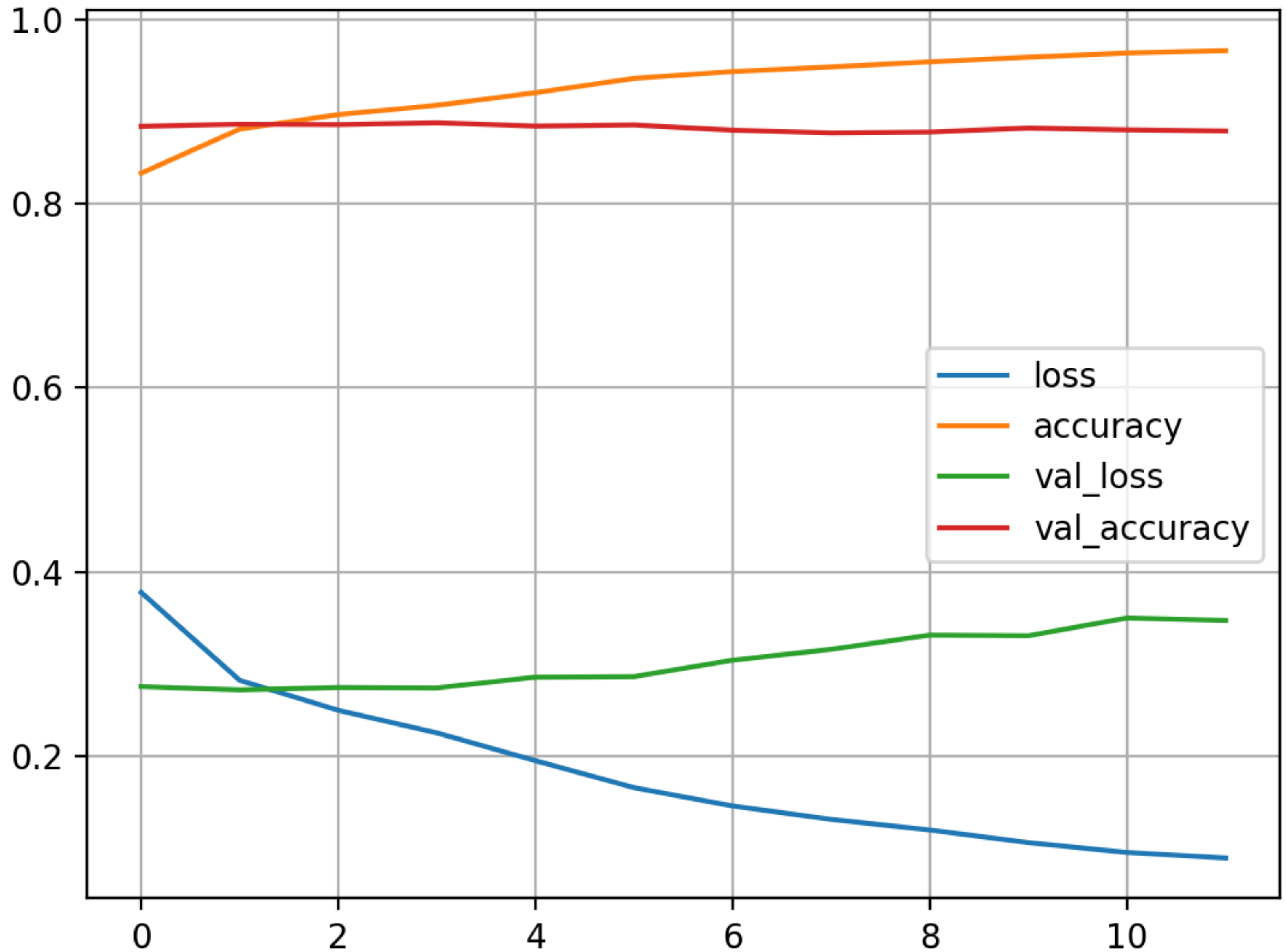
Dropout in Keras

```
model = Sequential()
model.add(layers.Input(shape=(input_dim,)))
model.add(layers.Dropout(0.3))
model.add(layers.Dense(100, activation='relu'))
model.add(layers.Dropout(0.3))
model.add(layers.Dense(100, activation='relu'))
model.add(layers.Dropout(0.3))
model.add(layers.Dense(1, activation='sigmoid'))

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

model.summary()
```

Discussion

- In this case we are adding a dropout:
  - To the inputs (hence, the dropout layer becomes the first layer)
  - After each layer, except the last one producing the output

Running again…



We are still not very satisfied…

Many things we can change

- Loss function
- Optimizer
- Number **and type** of the hidden layers
- Number of nodes for each hidden layer
- Dropout rate
- Number of epochs, patience, batch size…
- A completely different representation of the inputs

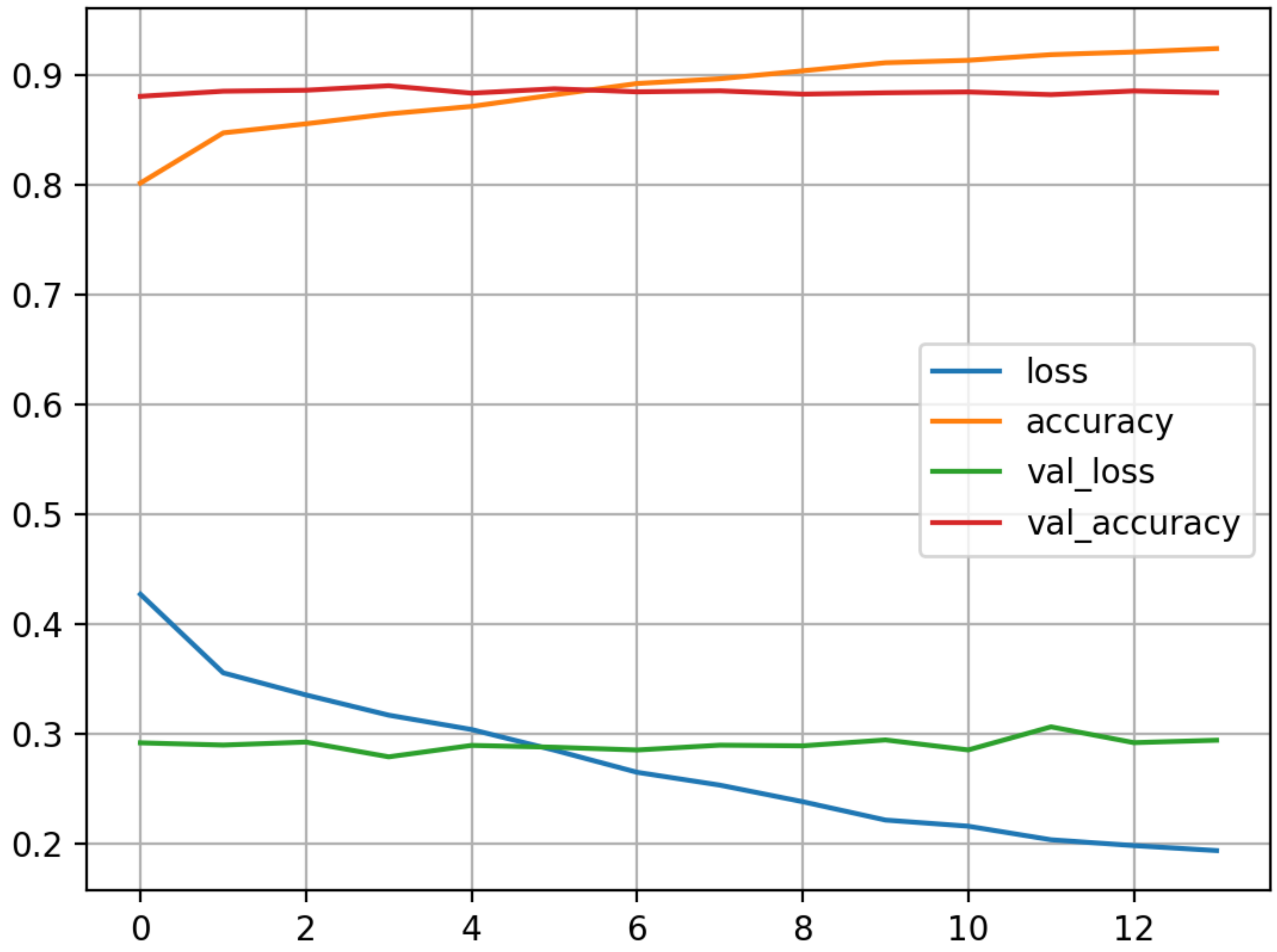
What we do?

- For now let's just add more and larger hidden layers, hence creating a deep neural network
- Later, we will see
  - How to optimize hyperparameters
  - How to change the representation and reuse pretrained layers
  - How to add different types of layers, e.g., recurrent layers

Towards
a deep neural network

```
model = Sequential()
model.add(layers.Input(shape=(input_dim,)))
model.add(layers.Dropout(0.5))
numHiddenLayers=3
numNodes=500
for i in range(0,numHiddenLayers):
    model.add(layers.Dense(numNodes, activation='relu'))
    model.add(layers.Dropout(0.5))
model.add(layers.Dense(1, activation='sigmoid'))

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

model.summary()

checkpoint_cb = callbacks.ModelCheckpoint("my_keras_model.keras", save_best_only=True)

early_stopping_cb = callbacks.EarlyStopping(patience=5,
                                                  restore_best_weights=True)
history = model.fit(X_train, y_train, epochs=100,
                    validation_data=(X_valid, y_valid),
                    callbacks=[checkpoint_cb, early_stopping_cb],batch_size=20)
```

Results...

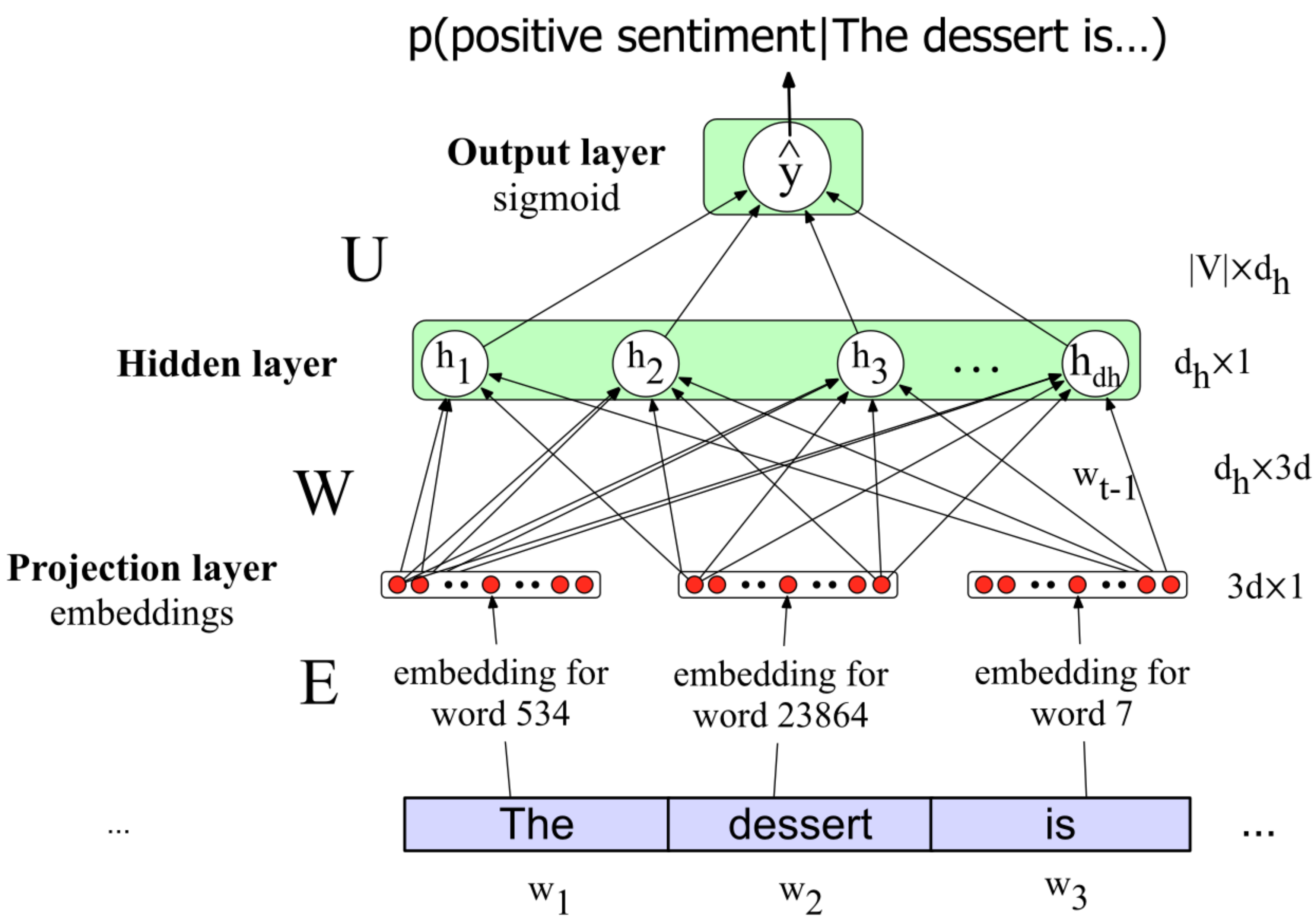

0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0
2
4
6
8
10
12
loss
accuracy
val_loss
val_accuracy

Lessons

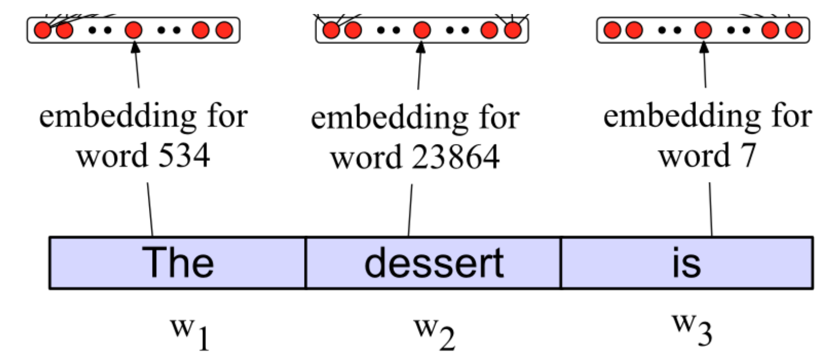
- Hard to get too far
- We need:
    - Better ways to represent inputs
    - Automated calibration of the network hyperparameters
    - Possibly, more complex network architectures

Embeddings as inputs

Neural Net Classification with embeddings as input features



Issue: texts comes in different sizes



This assumes a fixed size length (3)!

Kind of unrealistic.

Some simple solutions:

1. Make the input the length of the longest review
   - If shorter then pad with zero embeddings
   - Truncate if you get longer reviews at test time

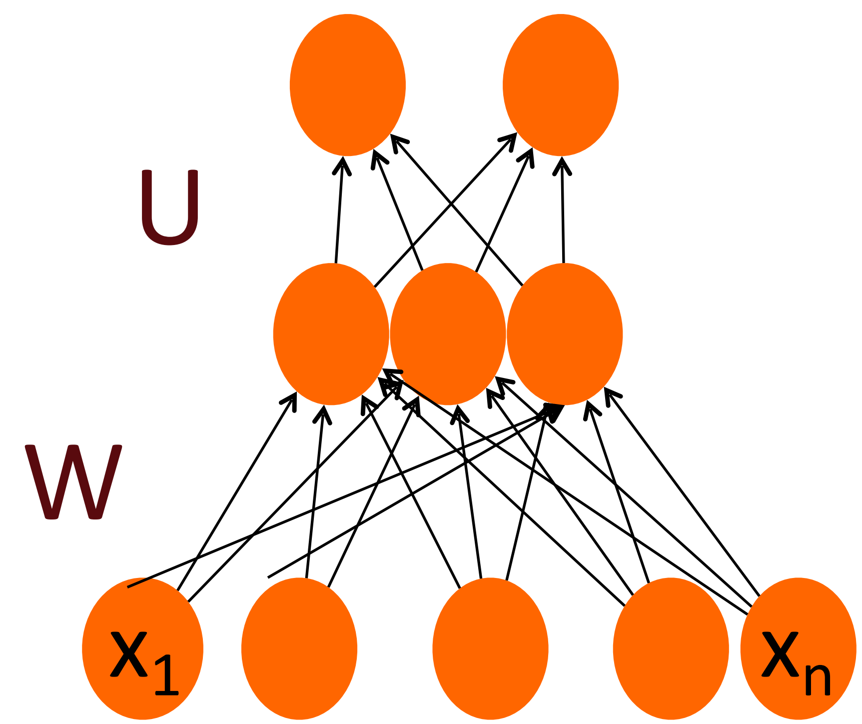
2. Create a single "sentence embedding" (the same dimensionality as a word) to represent all the words
   - Take the mean of all the word embeddings
   - Take the element-wise max of all the word embeddings
     - For each dimension, pick the max value from all words

Reminder: Multiclass Outputs

What if you have more than two output classes?

- Add more output units (one for each class)
- Use a "softmax layer"

$$\mathrm{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}} \quad 1 \le i \le D$$



First step:
representing document

How to represent a document?

- Each word in the vocabulary is encoded as an integer
- Therefore, each sentence is represented as a list of integers

Example

1 2 3 4 5 6 7

- John is going to the bus stop

5 8 2 9 5 10

- The technician is repairing the F512

Padding

- For the neural network models we will use, sentences must all have the same size

- Other models (e.g., for text completion) an pass subsequent windows of the same sentence to the Neural Network

Padding

- Let us assume a maximum length equal to 10

1 2 3 4 5 6 7 0 0 0

- John is going to the bus stop

5 8 2 9 5 10 0 0 0 0

- The technician is repairing the F512

- Shorter sentences will be “padded” with special IDs (typically zero)

- Longer sentences will be truncated

Out of value (OOV) tokens

- We code all words from the training set
- However, what if a word in the validation or test set never appeared in the training?
- How will this word be coded?

Moreover…

- To avoid having a too rich vocabulary, we may decide to limit the number of words, taking only the most frequent ones

- If we use (as it will be clearer later) a pre-trained embedding, some words may not belong to the pretrained embedding

  - E.g., F512 in our case, and maybe even John

- What will happen to those words?

OOV Dropped

- As a first possibility, OOV could be dropped
- Therefore, our sentences will become:
    - “is going to the bus stop”
    - “The technician is repairing the”
- Fine, but we may lose semantics…

OOV special token

- We may use a special token to represent OOV values
- Coded with an integer greater than the vocabulary size
- E.g. if the vocabulary is of 10000 words, it will be coded as 10001
- In this case the sentences become:
    - “<OOV> is going to the bus stop”
    - “The technician is repairing the <OOV>”
- In this case, the model will at least learn that there is a word there

Other approaches

There exist more advanced transformers capable to learn embeddings for OOV tokens based on the context in which they appear

Sentence tokenizer in Keras

- We use the Tokenizer class from tensorflow.keras.preprocessing.text
- It automatically:
  - Splits the sentences
  - Performs lowercasing
  - Skips special characters
  - Encodes words into integers
  - Handles a bounded vocabulary size
  - Handles OOV tokens

Example

```
from tensorflow.keras.preprocessing.text import Tokenizer

NB_WORDS = 40000  # Parameter indicating the number of words we'll put in the dictionary
MAX_LEN = 20  # Maximum number of words in a sequence
FILTER_STRING='!"#$%&()*+,-./:;<=>?@[\]^_`{"}~\t\n'

sentences=["John is going to the bus stop",
           "The technician is repairing the F512"]

tokenizer = Tokenizer(num_words=NB_WORDS,filters=FILTER_STRING,lower=True,
                      split=" ",oov_token="<OOV>")
tokenizer.fit_on_texts(sentences) #fits the sentences, creating the dictionary
print("Word index:",tokenizer.word_index)

t=tokenizer.texts_to_sequences(sentences)
print("Sequences:",t)

print("Reconstructed sentences",tokenizer.sequences_to_texts(t))
```

Discussion - I

- The constructor Tokenizer takes, among others:
    - num_words: the maximum dictionary size
    - filters: The list of characters to be filtered
    - lower: whether words should be lowercased
    - split: the splitting character
    - oov_token: the token used to encode the OOV. If omitted, OOV will be skipped
- Note: you can (and should!) always (better) preprocess the sentence before applying this tokenizer

Discussion - II

- fit_on_texts learns the vocabulary from the sentences
- word_index contains the dictionary (tuples containing the word and the corresponding ID)
- texts_to_sequences converts sentences into list of integers
- sequences_to_texts reconstructs the sentence (OOV will be skipped if no OOV token is used)

Result

```
Word index: {'<OOV>': 1, 'the': 2, 'is': 3, 'john': 4, 'going': 5,
'to': 6, 'bus': 7, 'stop': 8, 'technician': 9, 'repairing': 10,
'f512': 11}

[[4, 3, 5, 6, 2, 7, 8], [2, 9, 3, 10, 2, 11]]

['john is going to the bus stop', 'the technician is repairing the
f512']
```

Sentence padding

```
from tensorflow.keras.preprocessing.sequence import pad_sequences
padded_sentences = pad_sequences(t, maxlen=MAX_LEN,padding='post', truncating='post')
print(padded_sentences)
```

```
[[ 4  3  5  6  2  7  8  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 2  9  3 10  2 11  0  0  0  0  0  0  0  0  0  0  0  0  0  0]]
```

Note

- padding adds padding zeroes to maxlen before or after the sentence (depending on whether you use padding=‘pre’ or padding=‘post’)

- also, it truncates sentences longer than maxlen, performing the truncation before or after depending on the value of the truncating parameter

Complete Example: Word Embeddings in Keras

Imports, reading data

```
import pandas as pd
from bs4 import BeautifulSoup
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras import models
from tensorflow.keras import layers
from tensorflow.keras import callbacks


def strip_html(text):
    soup = BeautifulSoup(text, "html.parser")
    return soup.get_text()

dataset=pd.read_csv("IMDB Dataset.csv")
```

Setting some model hyper parameters...

```
NB_WORDS = 30000  # Parameter indicating the number of words we'll put in the dictionary
NB_EPOCHS = 10  # Number of epochs we usually start to train with
BATCH_SIZE = 5  # Size of the batches used in the mini-batch gradient descent
MAX_LEN = 100  # Maximum number of words in a sequence
FILTER_STRING='!"#$%&()*+,-./:;<=>?@[\]^_`{"}~\t\n'
EMBEDDING_SIZE=100 # Size of the word embedding
PATIENCE=10 # Patience level
DROP_RATE=0.4 # Dropout rate
```

Note

- In future we will have more parameters, including the number of layers, the number of nodes per layer, etc..
- Also, we will use a GridSearch or a RandomSearch to optimize the hyperparameters

Creating train, validation and test set

```
dataset['review']=dataset['review'].map(strip_html)

X_trainAll, X_test, y_trainAll, y_test = train_test_split(dataset['review'], dataset['sentiment'],
                                                           test_size=0.10, random_state=10)

X_train, X_valid, y_train, y_valid = train_test_split(X_trainAll, y_trainAll,
                                                           test_size=0.20, random_state=10)
```

Creating sequences and encoding labels

```
tokenizer = Tokenizer(num_words=NB_WORDS,filters=FILTER_STRING,
                      lower=True, split=" ",oov_token="<OOV>")

tokenizer.fit_on_texts(X_train) #fits the sentences, creating the dictionary
X_train_seq = tokenizer.texts_to_sequences(X_train)
X_valid_seq = tokenizer.texts_to_sequences(X_valid)
X_test_seq = tokenizer.texts_to_sequences(X_test)

X_train_seq_trunc = pad_sequences(X_train_seq, maxlen=MAX_LEN, padding='post')
X_valid_seq_trunc = pad_sequences(X_valid_seq, maxlen=MAX_LEN, padding='post')
X_test_seq_trunc = pad_sequences(X_test_seq, maxlen=MAX_LEN, padding='post')

le = LabelEncoder()
y_train_le=le.fit_transform(y_train)
y_valid_le=le.transform(y_valid)
y_test_le=le.transform(y_test)
```

Important Note!

- The Tokenizer must be fit on the training set

```
tokenizer.fit_on_texts(X_train)
```

- Because in principle you don’t have the test set
- Also, you don’t want to bias the validation set either

Creating the model

```
voc_len=len(tokenizer.word_index)

model = models.Sequential()
model.add(layers.Input(shape=(MAX_LEN,)))
model.add(layers.Embedding(voc_len+1,EMBEDDING_SIZE))
model.add(layers.Dropout(DROP_RATE))
model.add(layers.Flatten())
model.add(layers.Dense(128, activation='relu'))
model.add(layers.Dropout(DROP_RATE))
model.add(layers.Dense(128, activation='relu'))
model.add(layers.Dropout(DROP_RATE))
model.add(layers.Dense(1, activation='sigmoid'))

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

Network Topology

- The network is composed of four layers:
  - An embedding layer (see next slide)
  - Two Dense hidden layers
  - An output Dense layer with sigmoid activation
- Also, there are dropouts after each layer

The Embedding layer

- Takes as input the encoded sentences (i.e., lists of integers, including the OOV coding and the padding zeroes)
- During the training, the layer trains, for each word in the vocabulary, an embedding vector, as we previously explained when we introduced the concept of word embedding
- The layer sizes are:
  - Number of words in the vocabulary (this includes the OOV token) plus the zero padding
  - Size of the embedding
- Also, we need to specify the input size, which is the sentence (fixed) length

Training the model...

```
checkpoint_cb = callbacks.ModelCheckpoint("my_keras_model.keras", save_best_only=True)


early_stopping_cb = callbacks.EarlyStopping(patience=PATIENCE,
                                                       restore_best_weights=True)
history = model.fit(X_train_seq_trunc, y_train_le, epochs=NB_EPOCHS,
                    validation_data=(X_valid_seq_trunc, y_valid_le),
                    callbacks=[checkpoint_cb, early_stopping_cb],batch_size=BATCH_SIZE)

model = models.load_model("my_keras_model.keras") # rollback to best model
```

...and evaluating it...

```
loss, accuracy = model.evaluate(X_train_seq_trunc, y_train_le, verbose=False)
print("Training Accuracy: {:.4f}".format(accuracy))

loss, accuracy = model.evaluate(X_test_seq_trunc, y_test_le, verbose=True)
print("Testing Accuracy:  {:.4f}".format(accuracy))

import matplotlib.pyplot as plt
pd.DataFrame(history.history).plot()
plt.grid(True)
plt.show()
```

Sentiment analysis with multiple levels

Dataset

- We use Reddit sentiment analysis data from here:
- https://www.kaggle.com/cosmos98/twitter-and-reddit-sentimental-analysis-dataset?select=Reddit_Data.csv

The dataset

```
dataset=pd.read_csv("Reddit_Data.csv")
print(dataset.head())
dataset['clean_comment']=dataset['clean_comment'].map(str)
```

```
                                       clean_comment  category
0   family mormon have never tried explain them t...         1
1  buddhism has very much lot compatible with chr...         1
2  seriously don say thing first all they won get...        -1
3  what you have learned yours and only yours wha...         0
4  for your own benefit you may want read living ...         1
```

Note: the text is already clean, we may not need pruning it

Train, test, validation, and sentence processing

```
X_trainAll, X_test, y_trainAll, y_test = train_test_split(dataset['clean_comment'], dataset['category'],
                                                    test_size=0.10, random_state=10)

X_train, X_valid, y_train, y_valid = train_test_split(X_trainAll, y_trainAll,
                                                    test_size=0.20, random_state=10)


tokenizer = Tokenizer(num_words=NB_WORDS,filters=FILTER_STRING,
                      lower=True, split=" ",oov_token="<OOV>")

tokenizer.fit_on_texts(X_train) #fits the sentences, creating the dictionary
X_train_seq = tokenizer.texts_to_sequences(X_train)
X_valid_seq = tokenizer.texts_to_sequences(X_valid)
X_test_seq = tokenizer.texts_to_sequences(X_test)

X_train_seq_trunc = pad_sequences(X_train_seq, maxlen=MAX_LEN, padding='post')
X_valid_seq_trunc = pad_sequences(X_valid_seq, maxlen=MAX_LEN, padding='post')
X_test_seq_trunc = pad_sequences(X_test_seq, maxlen=MAX_LEN, padding='post')
```

One-hot encoding of labels

```
from sklearn.preprocessing import OneHotEncoder

oh=OneHotEncoder()
y_train_oh=oh.fit_transform([[x] for x in y_train]).toarray()
y_valid_oh=oh.transform([[x] for x in y_valid]).toarray()
y_test_oh=oh.transform([[x] for x in y_test]).toarray()

print(y_train_oh)
```

Note

- In order to apply the one-hot encoding the array must be reshaped
  - e.g. [1 0 1] → [[1],[0],[1]]
- This is done through the list comprehension

Creating the model

```
oc_len=len(tokenizer.word_index)

model = models.Sequential()
model.add(layers.Input(shape=(MAX_LEN,)))
model.add(layers.Embedding(voc_len,EMBEDDING_SIZE))
model.add(layers.Dropout(DROP_RATE))
model.add(layers.Flatten())
model.add(layers.Dense(128, activation='relu'))
model.add(layers.Dropout(DROP_RATE))
model.add(layers.Dense(128, activation='relu'))
model.add(layers.Dropout(DROP_RATE))
model.add(layers.Dense(3, activation='softmax'))

model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
model.summary()
```

Note

- The last layer has 3 nodes (one for each category) with a ‘softmax’ activation

- The loss function is ‘categorical_crossentropy’ to account for categorical variables instead of binary variables

Model training (nothing changes)

```
checkpoint_cb = callbacks.ModelCheckpoint(“my_keras_model.keras", save_best_only=True)


early_stopping_cb = callbacks.EarlyStopping(patience=PATIENCE,
                                                       restore_best_weights=True)
history = model.fit(X_train_seq_trunc, y_train_oh, epochs=NB_EPOCHS,
                    validation_data=(X_valid_seq_trunc, y_valid_oh),
                    callbacks=[checkpoint_cb, early_stopping_cb],batch_size=BATCH_SIZE)
```

Model evaluating - I (nothing changes)

```
model = models.load_model("my_keras_model.keras") # rollback to best model

loss, accuracy = model.evaluate(X_train_seq_trunc, y_train_oh, verbose=False)
print("Training Accuracy: {:.4f}".format(accuracy))

loss, accuracy = model.evaluate(X_test_seq_trunc, y_test_oh, verbose=True)
print("Testing Accuracy:  {:.4f}".format(accuracy))

import matplotlib.pyplot as plt
pd.DataFrame(history.history).plot()
plt.grid(True)
plt.show()
```

Model evaluating - II

```
#Prediction metrics
from sklearn.metrics import classification_report
import numpy as np

y_pred = model.predict(X_test_seq_trunc, verbose=1)

y_pred_cat=np.argmax(y_pred,axis=1)-1
y_test_cat=np.argmax(y_test_oh,axis=1)-1
print("Confusion matrix: ",pd.crosstab(y_test_cat,y_pred_cat))

print(classification_report(y_test_cat,y_pred_cat))
```

Notes

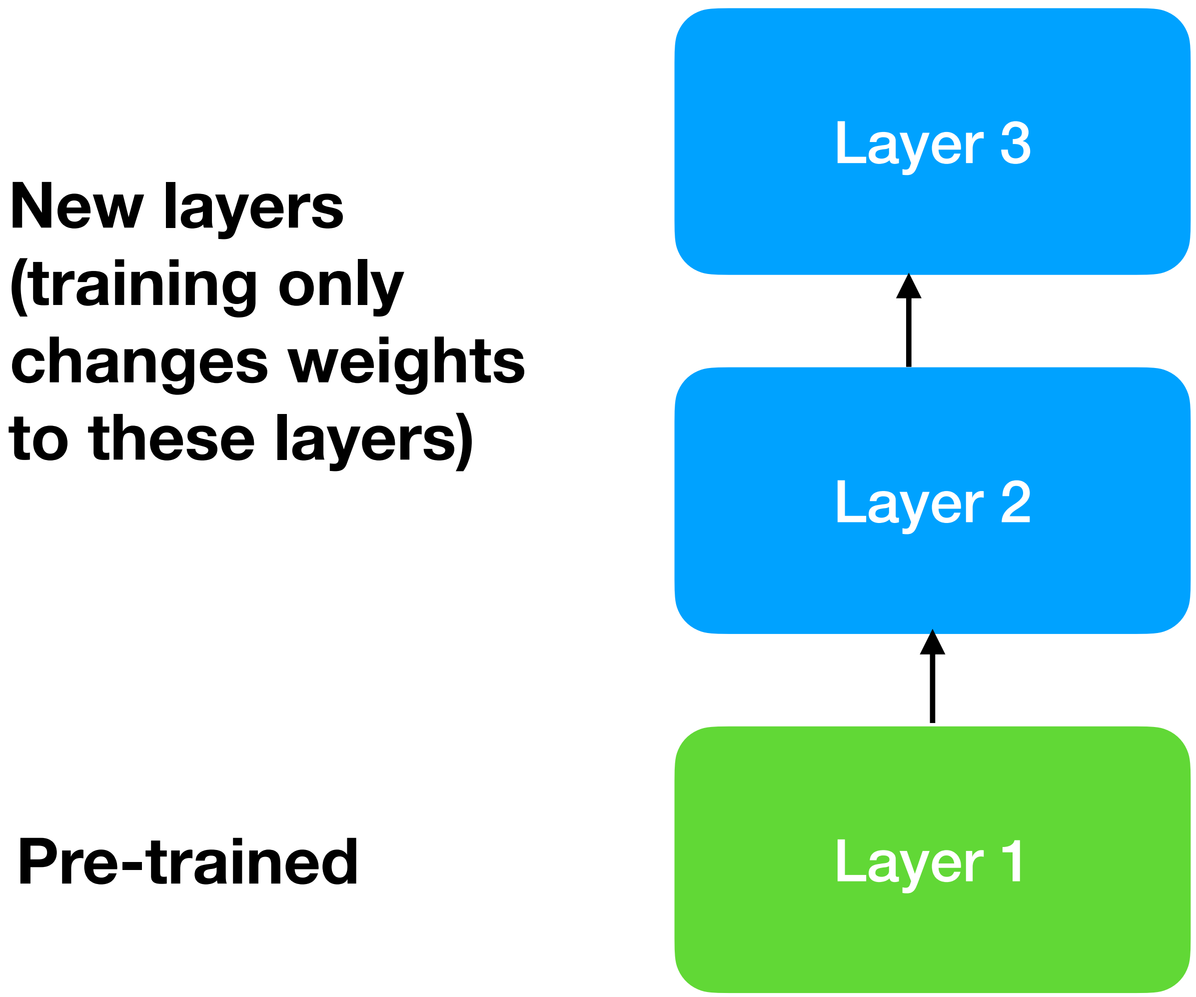
- If we print y_pred we obtain:
```
[[1.9410513e-01 3.2797144e-04 8.0556691e-01]
 [5.8923888e-01 1.7557168e-01 2.3518944e-01]
 [4.2120762e-02 9.2255133e-01 3.5327874e-02]
 ...
 [3.5630044e-01 3.4800732e-01 2.9569224e-01]
 [8.0389720e-01 6.6665724e-02 1.2943704e-01]
 [3.4622315e-01 4.1423094e-01 2.3954593e-01]]
```
- Therefore, to get back categories, we use the np.argmax function (on axis 1, so that each sub-list is converted into a value)
- It returns the column of the y_pred output with the highest value
- For comparison purpose, we do the same on the one-hot encoded converted labels, i.e., y_test_oh
- Then, since this will produces values in the interval [0, 2], we subtract 1 to get back {-1, 0, 1} (but it's not strictly necessary)
- After that we can print the confusion matrix or the classification report

Using a pre-trained embedding

Loading pretrained embedding

- Instead of training the weights of the Embedding layer, we could load them from a pretrained embedding (those we used in previous examples)
- Do it when:
    - Your dataset is not big enough to train an embedding
    - Your dataset contains plain English or text similar (for the domain) to the domain on which the embedding has been trained
- Don't do it when
    - Your dataset is from a very specific domain, where terms assume a very specific meaning
    - Also, the dataset contains many terms that may not be in the pretrained embedding (e.g., when classifying technical documents)

About reusing pretrained layers



How

- Each word w in our training has a numerical ID i
- If the word w exists in the embedding, we associate to the i-th row of the Embedding weights its vector
- Otherwise, we set that row to zero

Dataset creation

- Up to the training set construction and the labels' encoding, the source code is exactly the same as before

Loading pre-trained embedding

```
import gensim.downloader
glove_vectors = gensim.downloader.load('glove-wiki-gigaword-100')

EMBEDDING_SIZE=glove_vectors.vector_size
voc_len=len(glove_vectors.key_to_index)


embedding_matrix = np.zeros((voc_len, EMBEDDING_SIZE))
for word, i in tokenizer.word_index.items():
    if word in glove_vectors:
        embedding_vector = glove_vectors[word]
        embedding_matrix[i] = embedding_vector
```

Creating the model

```python
from tensorflow.keras import initializers
model = models.Sequential()
model.add(layers.Input(shape=(MAX_LEN,)))
model.add(layers.Embedding(
    voc_len,
    EMBEDDING_SIZE,
    embeddings_initializer=initializers.Constant(embedding_matrix),
    trainable=False))
model.add(layers.Dropout(DROP_RATE))
model.add(layers.Flatten())
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dropout(DROP_RATE))
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dropout(DROP_RATE))
model.add(layers.Dense(3, activation='softmax'))

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

Notes

- We pass the weights to the Embedding through the parameter:

  embeddings_initializer=keras.initializers.Constant(embedding_matrix)

- Also, we set trainable=False so that the layer will not be trained (this will be faster)

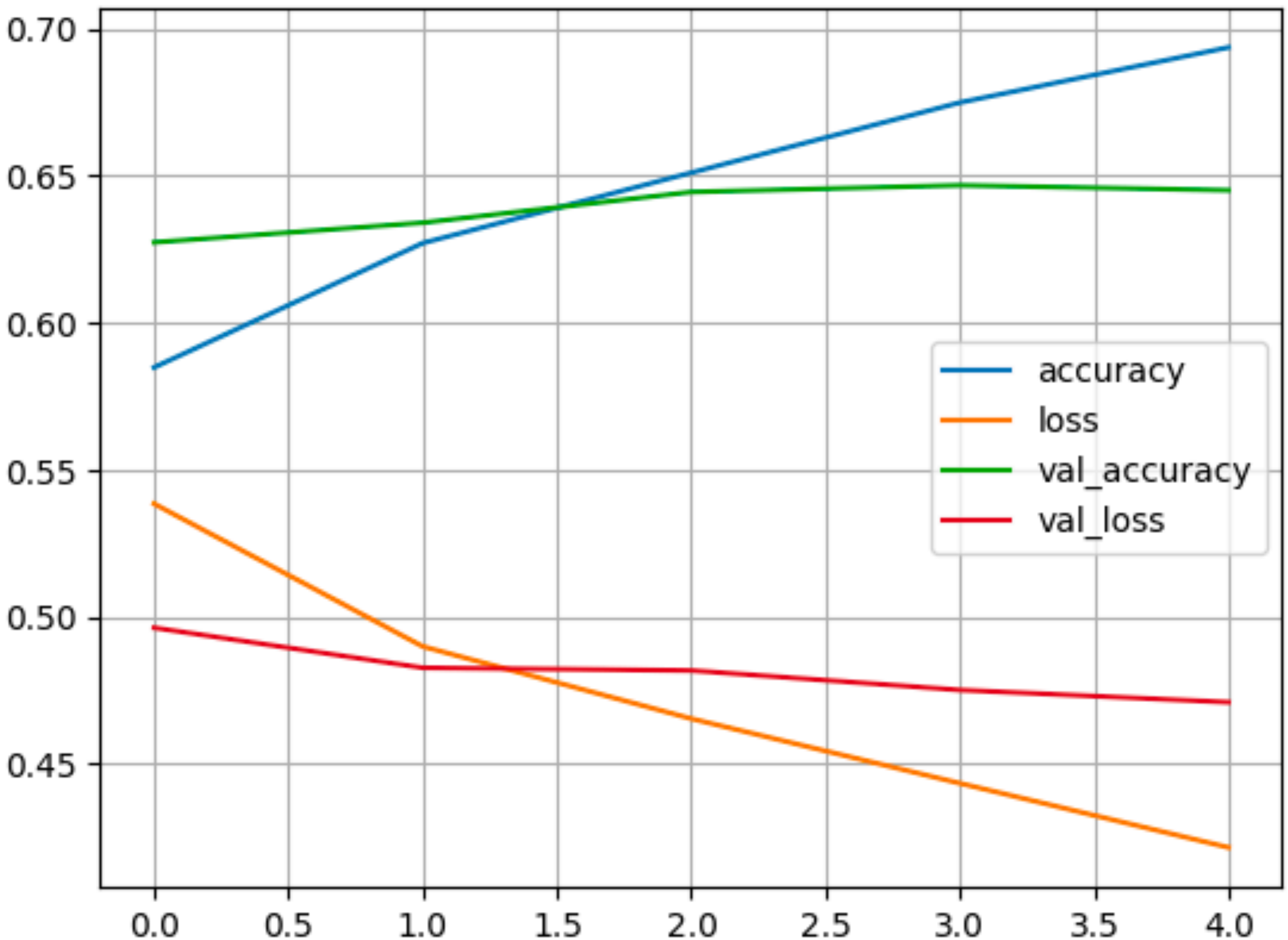
Rest of the code…

Same as before…

Notes

- We load a Gensim pretrained embedding as done before
- Then, we set the EMBEDDING_SIZE as the size of any of its row
- Then, we create a matrix of zeroes having a size = voc_len X EMBEDDING_SIZE
- We traverse the words and IDs from the vocabulary
    - If the word is in the embedding we add its vector to the matrix to the i-th row

Results…not so good…

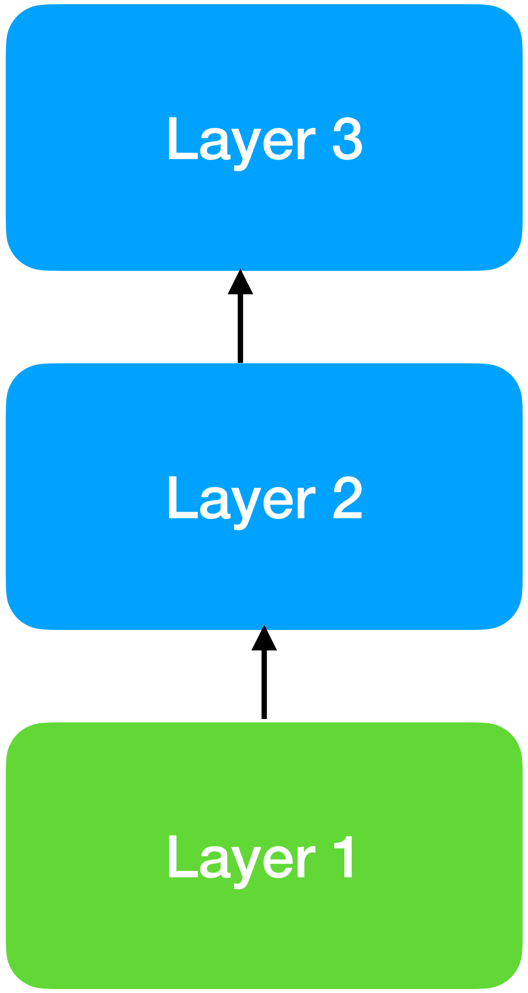


Fine tuning

- If the pretrained embedding (as it is) does not produce very good results, you may consider to still load it, but then fine-tune its weights

- All you need to do is to set
  `trainable=True`

Fine tuning

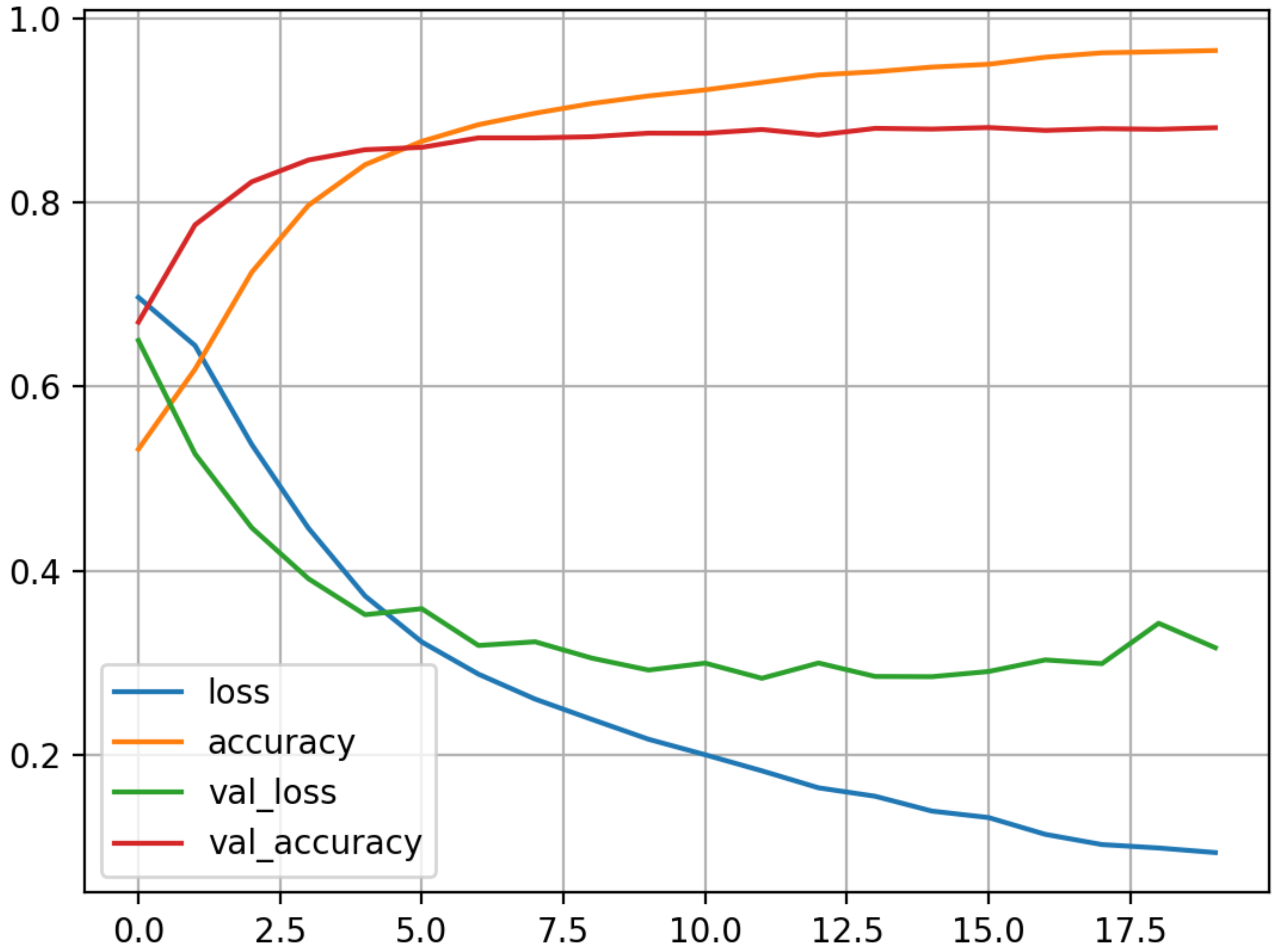


New layers

Pre-trained

Training determines the weights of Layer 2 and Layer 3, and tunes the weights of Layer 1 as well

Results


1.0
0.8
0.6
0.4
0.2
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
loss
accuracy
val_loss
val_accuracy

General discussion

- When the model is fitting, see how the accuracy on the training set and on the validation set improves
- If it is improving too fast on the training set and it is kept flat on the validation set, this is an indication of overfitting
- What to do?
  - Use regularization approaches, e.g. dropout
  - Early stopping/patience
  - Use less complex models
  - Tune the hyperparameters (next part of the lecture)

Hyeperparameter Optimization

On Hyperparameter Tuning

- The most difficult task when using neural networks is to define its hyperparameters

- In the following, we will see a possible procedure (several available) to achieve this goal

Keras-Tuner package

- Install it as
  `pip3 install keras-tuner`

Steps

- Create a class inheriting from HyperModel
- The class constructor gets all the needed information to generate the parameters
    - For example, lists of values, or ranges
- The method build has a parameter hp of type HyperParameter
    - Then, it uses methods of this class to generate values

The class - I

```
from tensorflow.keras import layers
from tensorflow.keras import models
from tensorflow.keras import optimizers
from bs4 import BeautifulSoup
import keras_tuner as kt
from keras_tuner import HyperModel


def strip_html(text):
    soup = BeautifulSoup(text, "html.parser")
    return soup.get_text()

class EmbeddingHyperModel(HyperModel):
    def __init__(self,voc_size,emb_size,max_len,numNodes,minLayers,
                 maxLayers,minDrop,maxDrop,minLearning,maxLearning):
        self.voc_size=voc_size
        self.emb_size=emb_size
        self.max_len=max_len
        self.numNodes=numNodes
        self.minLayers=minLayers
        self.maxLayers=maxLayers
        self.minDrop=minDrop
        self.maxDrop=maxDrop
        self.minLearning=minLearning
        self.maxLearning=maxLearning
```

The class - II

```
def build(self,hp):
    drop_rate=hp.Float(name="dropout",min_value=self.minDrop,
                       max_value=self.maxDrop,sampling='linear')
    nodes_hidden=hp.Choice("units",self.numNodes)
    model = models.Sequential()
    model.add(layers.Input(shape=(MAX_LEN,)))
    model.add(layers.Embedding(self.voc_size+1,100,input_length=100))
    model.add(layers.Dropout(drop_rate))
    model.add(layers.Flatten())
    for i in range(hp.Int(name="layers",min_value=self.minLayers,
                           max_value=self.maxLayers,sampling='linear')):
        model.add(layers.Dense(nodes_hidden, activation='relu'))
        model.add(layers.Dropout(drop_rate))
    model.add(layers.Dense(1, activation='sigmoid'))
    model.compile(loss='binary_crossentropy',
                  optimizer=optimizers.Adam(
                      hp.Float(
                          "learning_rate",
                          min_value=self.minLearning,
                          max_value=self.maxLearning,
                          sampling="LOG"
                      )),
                  metrics=['accuracy'])
    return model
```

Some methods of the class HyperParameter

Each method requires to specify a label for the hyperparameter and then settings to generate them

- `Choice:` draws values from a list
- `Int:` generates integer values in a range
- `Float:` generates float values in a range

Initial settings and dataset loading

```
NB_WORDS = 30000  # Parameter indicating the number of words we'll put in the dictionary
NB_EPOCHS = 20  # Number of epochs we usually start to train with
BATCH_SIZE = 50  # Size of the batches used in the mini-batch gradient descent
MAX_LEN = 100  # Maximum number of words in a sequence
FILTER_STRING='!"#$%&()*+,-./:;<=>?@[\]^_`{"}~\t\n'
EMBEDDING_SIZE=100 # Size of the word embedding
PATIENCE=10 # Patience level

import pandas as pd
dataset=pd.read_csv("IMDB_Dataset.csv")
dataset['review']=dataset['review'].map(strip_html)
```


```
dataset['review']=dataset['review'].map(strip_html)
```

Dataset preparation

```
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras import models
from tensorflow.keras import layers
from tensorflow.keras import callbacks


X_trainAll, X_test, y_trainAll, y_test = train_test_split(dataset['review'], dataset['sentiment'],
                                                          test_size=0.10, random_state=10)

tokenizer = Tokenizer(num_words=NB_WORDS,filters=FILTER_STRING,
                      lower=True, split=" ",oov_token="<OOV>")

tokenizer.fit_on_texts(X_trainAll) #fits the sentences, creating the dictionary
X_train_seq = tokenizer.texts_to_sequences(X_trainAll)
X_test_seq = tokenizer.texts_to_sequences(X_test)

X_train_seq_trunc = pad_sequences(X_train_seq, maxlen=MAX_LEN, padding='post')
X_test_seq_trunc = pad_sequences(X_test_seq, maxlen=MAX_LEN, padding='post')

le = LabelEncoder()
y_train_le=le.fit_transform(y_trainAll)
y_test_le=le.transform(y_test)

voc_len=len(tokenizer.word_index)
```

Note

- Validation set not required (the tuner creates it for you)
- However, once the tuning is complete, you need to create yourself a new split of the training into training and validation, to properly fit the model

Running the Optimization

```
from tensorflow.keras import callbacks
hm=EmbeddingHyperModel(voc_len,EMBEDDING_SIZE,MAX_LEN,[128,256,512],1,5,0,0.4,1e-2,1e-4)

tuner = kt.Hyperband(hm,
                     objective='val_accuracy',
                     max_epochs=10,
                     factor=3,
                     directory='my_dir')

stop_early = callbacks.EarlyStopping(monitor='val_loss', patience=5)

tuner.search(X_train_seq_trunc, y_train_le, epochs=50, validation_split=0.2, callbacks=[stop_early])

best_hps=tuner.get_best_hyperparameters(num_trials=1)[0]

print(f"""
The hyperparameter search is complete.
The optimal number of layers is {best_hps.get('layers')}
The units in the densely-connected layers are {best_hps.get('units')}
The optimal dropout rate is {best_hps.get('dropout')}
""")

model = tuner.hypermodel.build(best_hps)

# From here, you can just use the model to fit it and use it
```

Discussion - I

- First, we instantiate the EmbeddingHyperModel class, passing all parameters
  - ranges or lists for hyperparameters to tune
  - fixed parameters (e.g., vocabulary and sentence length, embedding size)
- Second we create the tuning by instantiate the HyperBand tuner
  - Note, alternative optimizers are available:
    - BayesianOptimization
    - RandomSearch

HyperBand syntax

- See https://keras.io/api/keras_tuner/tuners/hyperband/ for a full set of parameters
- We pass, among others:
    - The model
    - The objective to optimize (accuracy)
    - The maximum number of epochs
    - A reduction factor which at each iteration reduces the number of epochs and of hyperparameters
    - A directory where to save optimization results

Discussion - II

- Third, we set a callback for early stopping (patience)
```
stop_early = keras.callbacks.EarlyStopping(monitor='val_loss', patience=5)
```

- Fourth, we run the search, also specifying how the validation set should be split

```
tuner.search(X_train_seq_trunc, y_train_le, epochs=50,
validation_split=0.2, callbacks=[stop_early])
```

- Fifth, we can get the hyperparameters from the results (there is a ranked list, and we only take the first one

```
best_hps=tuner.get_best_hyperparameters(num_trials=1)[0]
```

- Finally, we can either use `best_hps` to fit a model, or simply print them

Example of result

```
The hyperparameter search is complete.

The optimal number of layers is 4

The units in the densely-connected layers are 256

The optimal dropout rate is 0.08324500256196288
```

Note

- The optimization may take several hours, so leave it alone while running

- Once it is complete, I suggest to save the hyperparameters found, so you could reuse them

Using pre-trained models from Tensorflow Hub

Reusing pretrained layers

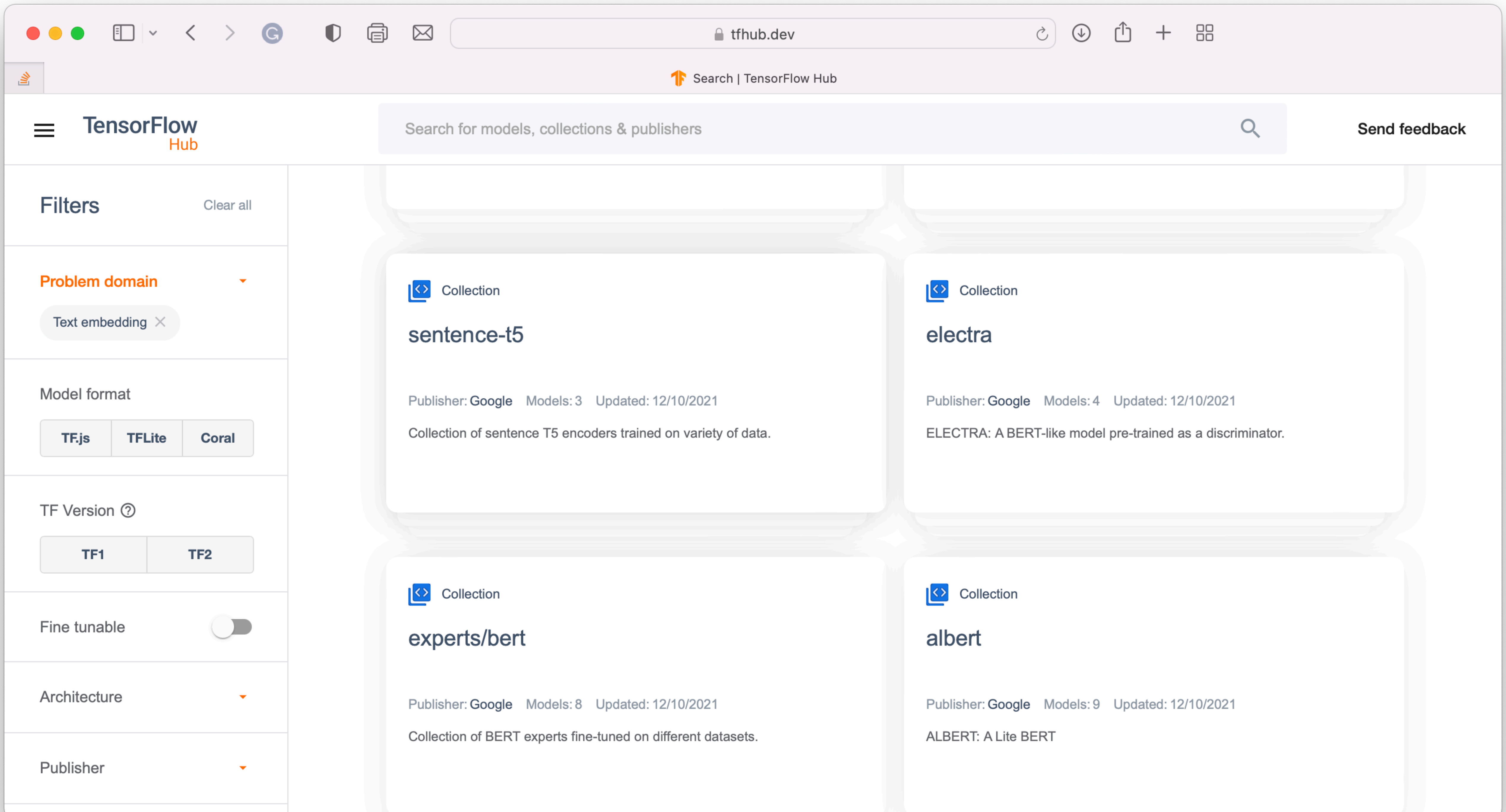
- Tensorflow makes available pre-trained architectures that you could reuse, fine-tune, and integrate with your models

  https://www.tensorflow.org/hub

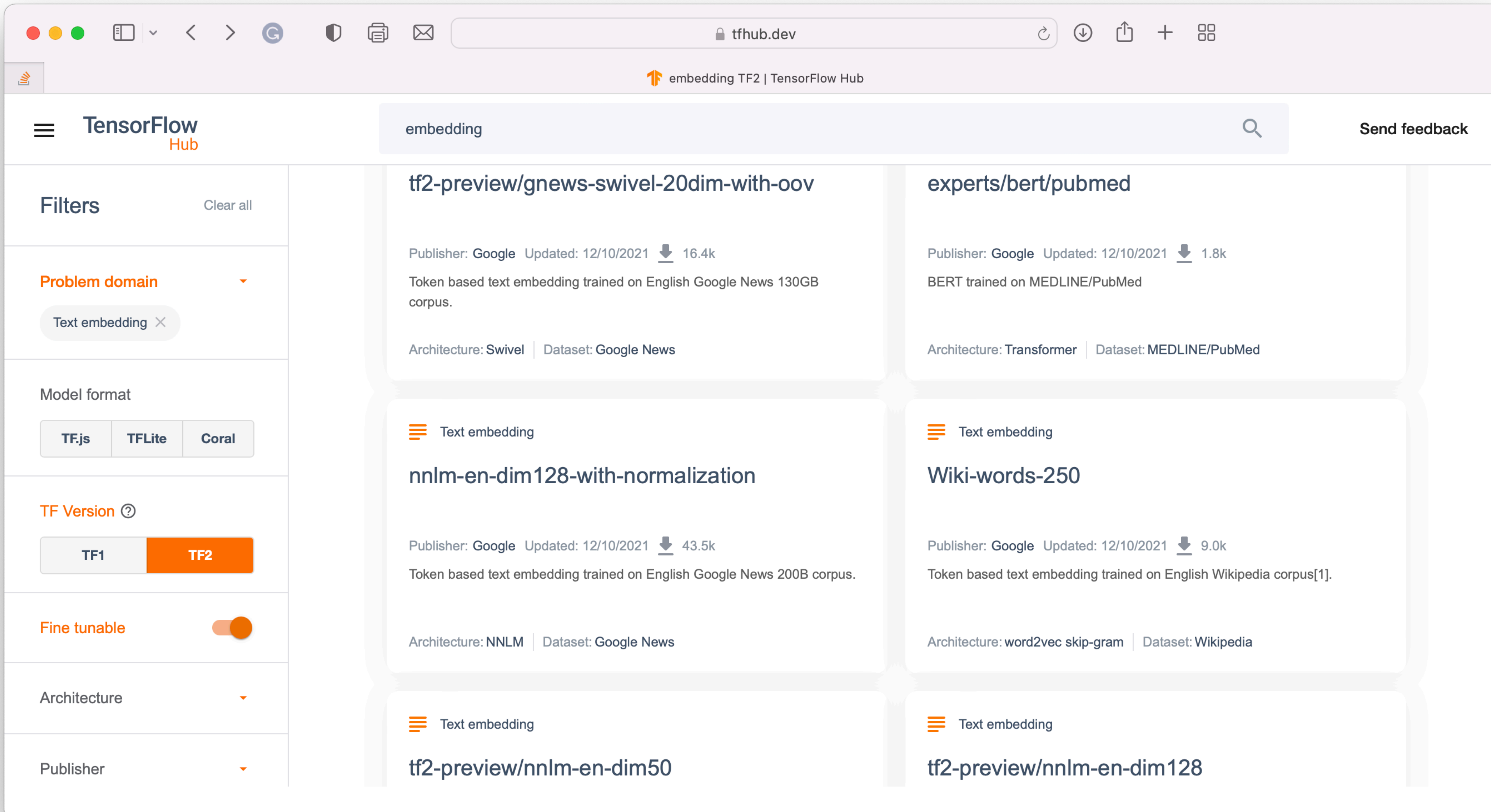
- Installation

```
pip install --upgrade tensorflow_hub
```

The interface



Searching for a model

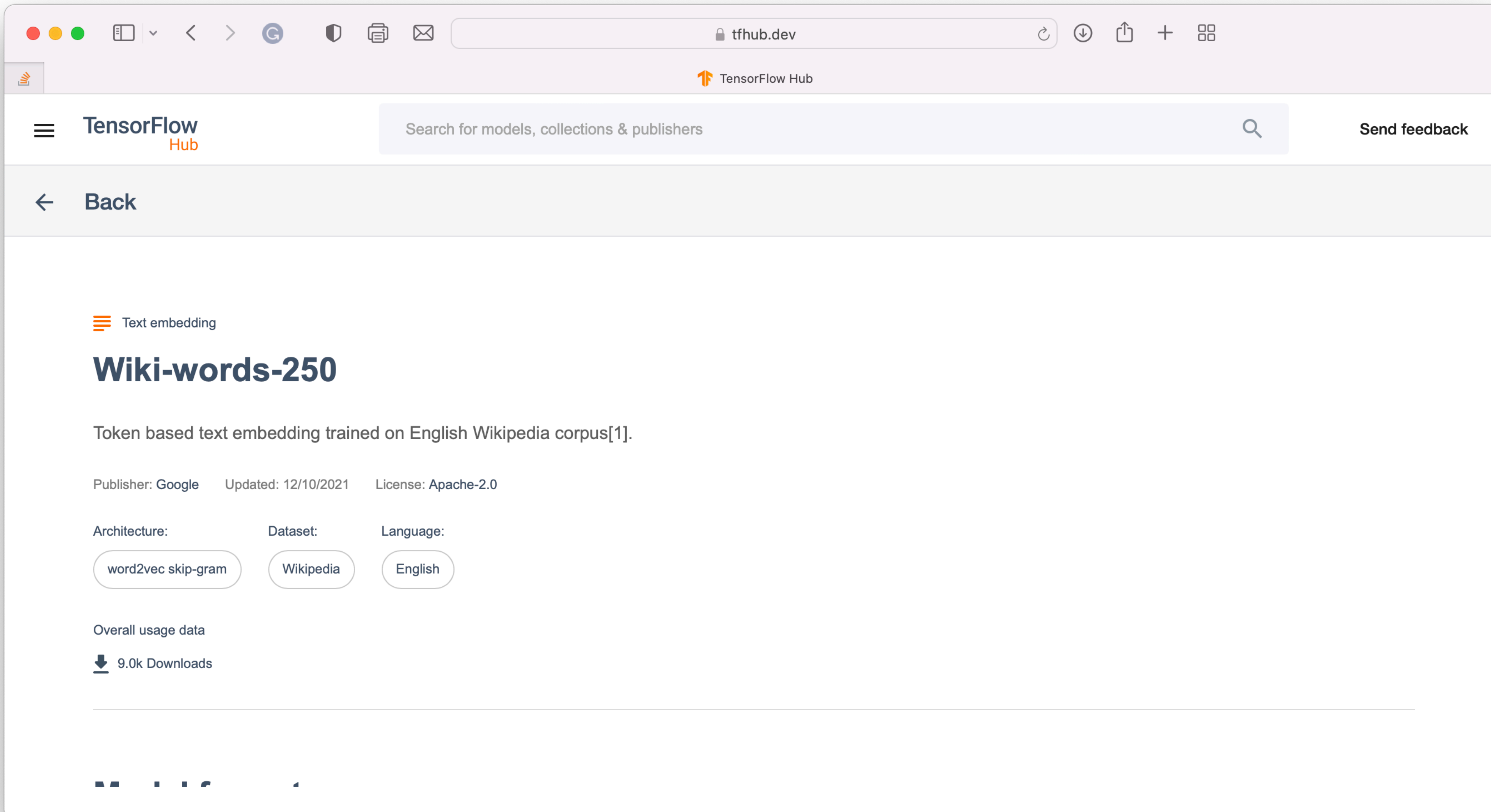


Discussion

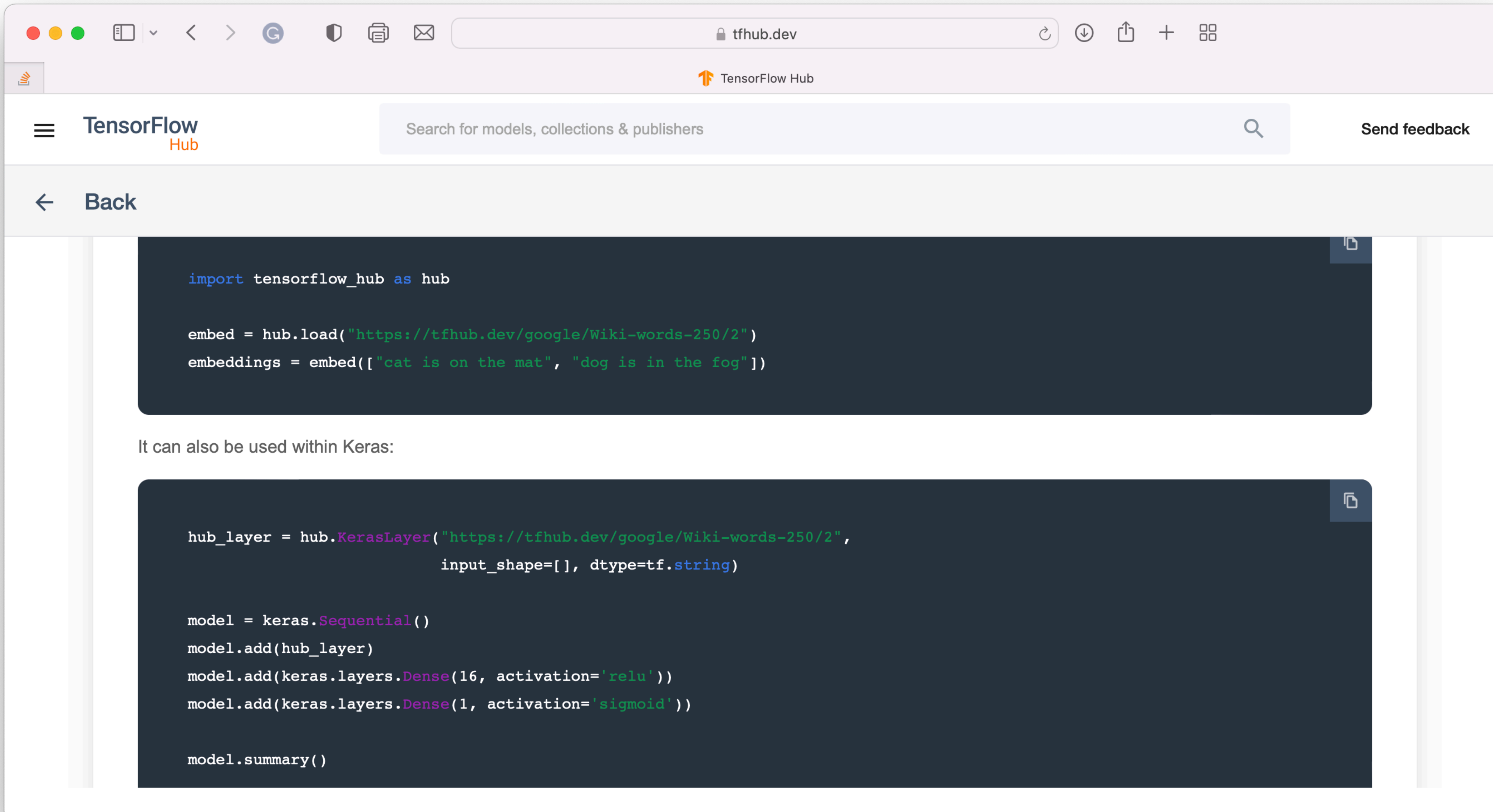
I selected:

- Embedding models for text processing
- Models supporting TensorFlow 2
- Models that can be fine-tuned

Selected model



How to import it...



Just using the embeddings...

```
import tensorflow_hub as hub

embed = hub.load("https://tfhub.dev/google/Wiki-words-250/2")
embeddings = embed(["cat is on the mat", "dog is in the fog"])

print(embeddings)
```

Result (partial)

```
tf.Tensor(
[[-5.14805540e-02 -1.92053974e-01  4.53008525e-02 -9.96370390e-02
   5.38923480e-02  8.34979564e-02  7.27996677e-02 -1.27169073e-01
   6.24356270e-02  9.81895104e-02 -5.33969402e-02  1.61190659e-01
  -1.36027522e-02 -2.72708107e-03  1.71537384e-01  1.24906197e-01
   1.15381563e-02 -2.77321301e-02  6.66442439e-02 -1.28565924e-02
   3.95655632e-02  1.61706526e-02  3.44905234e-03 -3.30653414e-02
   1.13467865e-01 -3.23929265e-02  6.64588250e-03  5.34387156e-02
   1.19479060e-01  4.63577174e-02  8.30192715e-02 -5.91111630e-02
   8.59290361e-02 -1.01532824e-01  7.54378317e-03  4.15412569e-03
   5.89248538e-03 -2.51556151e-02  1.13079183e-01 -4.36960533e-02
  -1.68391705e-01  2.92641334e-02 -1.40178025e-01 -8.40619281e-02
   1.48394153e-01  9.07467008e-02 -5.67608029e-02 -1.04004763e-01
  -8.44553933e-02  8.50597844e-02  7.93245584e-02  2.39145532e-02
  -1.19153991e-01  2.17635736e-01 -2.18595695e-02 -4.27431203e-02
  -1.75292030e-01 -5.83514608e-02  1.50858937e-02 -9.78629012e-03
  -8.74623880e-02 -1.32550955e-01  2.55552512e-02  1.07006066e-01
   1.12839080e-01 -1.16539821e-01 -1.15803346e-01 -9.23949555e-02
  -1.55965701e-01 -1.25600128e-02  1.27804086e-01 -8.76564384e-02
   1.16007529e-01  6.24112086e-03  1.30231380e-01 -1.41361311e-01
  -3.13660502e-02  3.92044894e-02  7.72149563e-02  3.57291549e-02
  -1.36040106e-01  1.53127965e-02  2.75261067e-02 -2.27289833e-02
   5.95712326e-02  5.23981899e-02 -1.40063182e-01  1.72593407e-02
  -2.35673296e-03 -7.02513158e-02 -7.38211796e-02  6.78406060e-02
   5.90462275e-02 -2.17459753e-01  1.00375796e-02 -6.83509782e-02
  -1.23797124e-02  7.79274926e-02  1.04184868e-02  1.41857594e-01
   7.24928230e-02  2.46292517e-01 -3.59483659e-02  1.62022024e-01
  -1.50000408e-01 -9.25377309e-02 -4.19961512e-02 -9.46031958e-02
```

Note

- A TensorFlow Tensor is a data structure very similar to Numpy array
- Optimizes the array handling in neural networks

Using for classification…

- The reusable layer already accepts arrays of strings, so no need for tokenizing them
- However, you may want to clean up the strings…
- Note: you need to install `tf_keras` for this, due to an incompatibility of Tensorflow Hub layers with `tensorflow.keras`

```
pip install tf_keras
```

Example (without cleanup)

```python
import tensorflow_hub as hub
import tensorflow as tf
import tensorflow.keras as keras
import pandas as pd
from bs4 import BeautifulSoup
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras import models
from tensorflow.keras import callbacks


def strip_html(text):
    soup = BeautifulSoup(text, "html.parser")
    return soup.get_text()

dataset=pd.read_csv(“IMDB_Dataset.csv")

NB_EPOCHS = 20  # Number of epochs we usually start to train with
BATCH_SIZE = 50  # Size of the batches used in the mini-batch gradient descent
PATIENCE=10 # Patience level
DROP_RATE=0.4 # Dropout rate


dataset['review']=dataset['review'].map(strip_html)
```

Creating the sets

```
X_trainAll, X_test, y_trainAll, y_test = train_test_split(dataset['review'], dataset['sentiment'],
                                                           test_size=0.10, random_state=10)

X_train, X_valid, y_train, y_valid = train_test_split(X_trainAll, y_trainAll,
                                                       test_size=0.20, random_state=10)

le = LabelEncoder()
y_train_le=le.fit_transform(y_train)
y_valid_le=le.transform(y_valid)
y_test_le=le.transform(y_test)
```

Creating the network

```
import tf_keras as keras
import tensorflow_hub as hub
import tensorflow as tf

hub_layer = hub.KerasLayer("https://tfhub.dev/google/Wiki-words-250/2",
                           input_shape=[], dtype=tf.string,trainable=False)

model = keras.Sequential()
model.add(hub_layer)
model.add(keras.layers.Dropout(DROP_RATE))
model.add(keras.layers.Dense(128, activation='relu'))
model.add(keras.layers.Dropout(DROP_RATE))
model.add(keras.layers.Dense(128, activation='relu'))
model.add(keras.layers.Dropout(DROP_RATE))
model.add(keras.layers.Dense(1, activation='sigmoid'))

model.summary()

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

Fitting...

```
checkpoint_cb = keras.callbacks.ModelCheckpoint("my_keras_model.keras", save_best_only=True)

early_stopping_cb = keras.callbacks.EarlyStopping(patience=PATIENCE,
                                                  restore_best_weights=True)
history = model.fit(X_train, y_train_le, epochs=NB_EPOCHS,\
                    validation_data=(X_valid, y_valid_le), \
                        callbacks=[early_stopping_cb, checkpoint_cb],batch_size=BATCH_SIZE)
```

Evaluating...

```
loss, accuracy = model.evaluate(X_train, y_train_le, verbose=False)
print("Training Accuracy: {:.4f}".format(accuracy))

loss, accuracy = model.evaluate(X_test, y_test_le, verbose=True)
print("Testing Accuracy:  {:.4f}".format(accuracy))

import matplotlib.pyplot as plt
pd.DataFrame(history.history).plot()
plt.grid(True)
plt.show()
```

Fine tuning…

As explained before, just setting `trainable=True` does the job

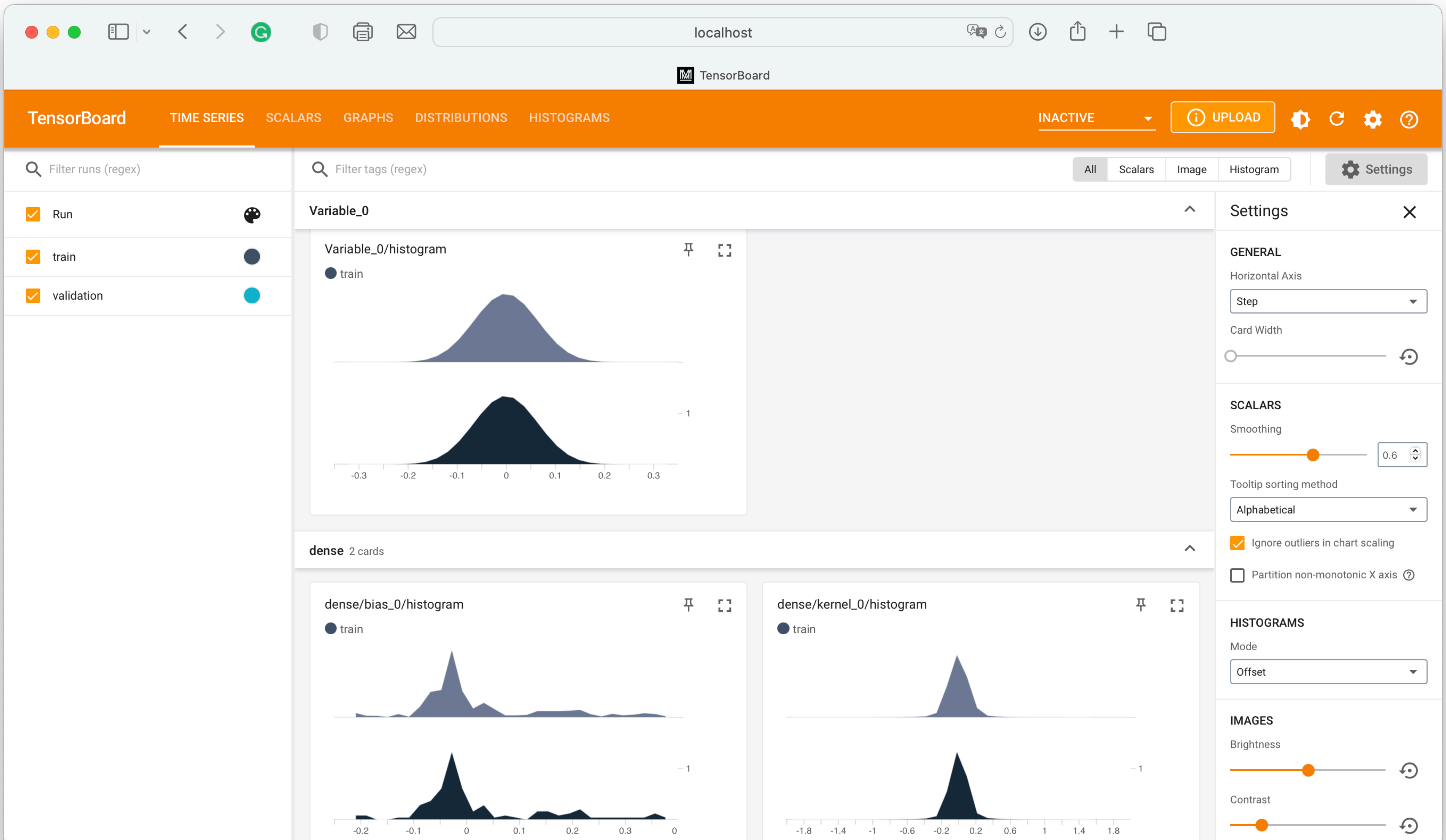
Tensorboard

- Dashboard to visualize TensorFlow evolution
- Enable Callbacks by adding the following callback to the list of your model callbacks:

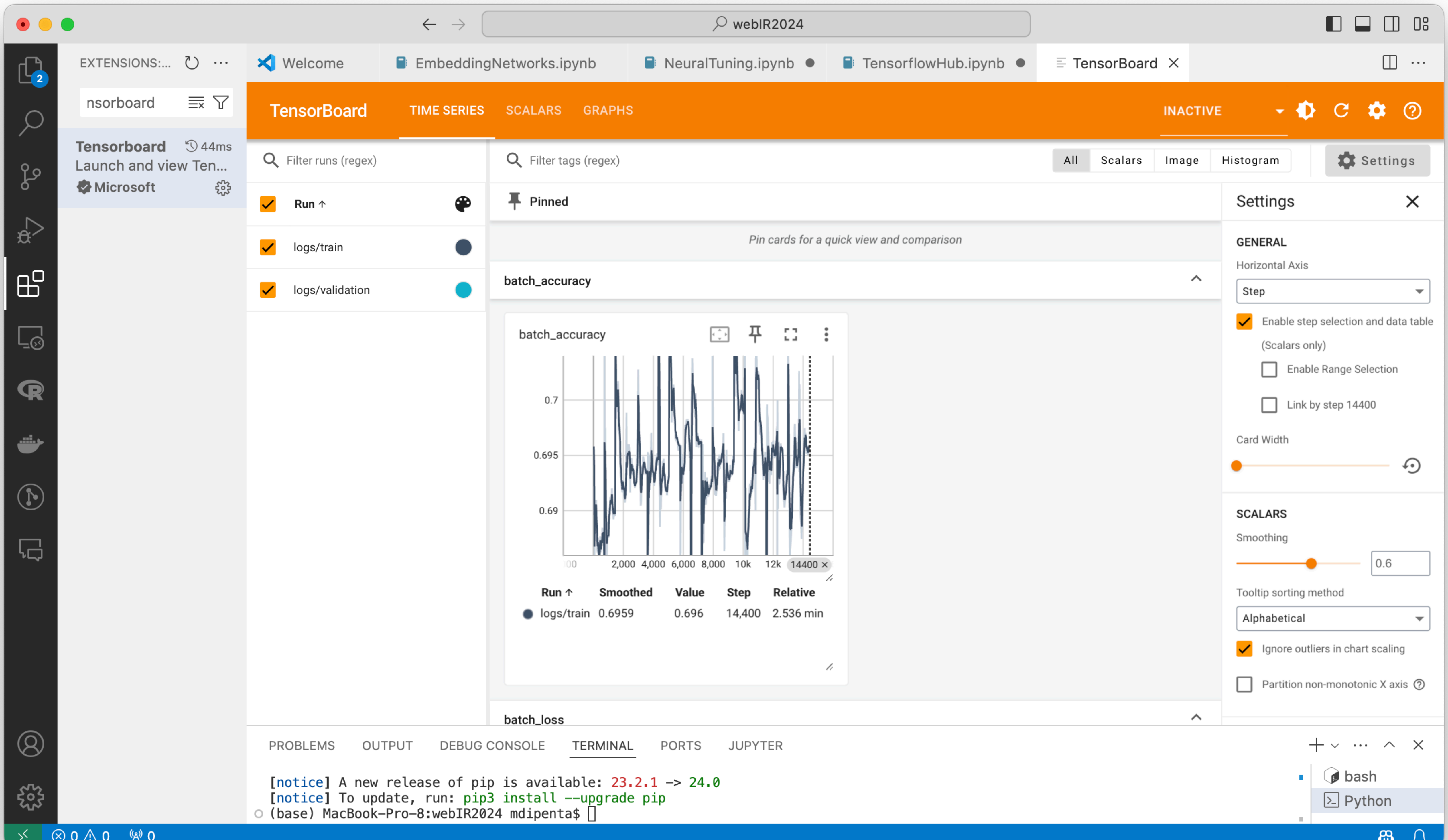
```
tb_callback = keras.callbacks.TensorBoard('./logs',
update_freq=1)
```

- Where the first parameter specifies the directory where Tensorboard Logs are saved (same as below) and the update_freq indicates that the logs will be updated every N batches (every batch in this case)
- Running it:
  - `tensorboard serve --logdir tb_dir`
- Accessing it:
  - `http://localhost:6006/`

The board


localhost
TensorBoard
TensorBoard
TIME SERIES
SCALARS
GRAPHS
DISTRIBUTIONS
HISTOGRAMS
INACTIVE
UPLOAD
Filter runs (regex)
Run
train
validation
Filter tags (regex)
All
Scalars
Image
Histogram
Settings
Variable_0
Variable_0/histogram
train
dense 2 cards
dense/bias_0/histogram
train
dense/kernel_0/histogram
train
Settings
GENERAL
Horizontal Axis
Step
Card Width
SCALARS
Smoothing
0.6
Tooltip sorting method
Alphabetical
Ignore outliers in chart scaling
Partition non-monotonic X axis
HISTOGRAMS
Mode
Offset
IMAGES
Brightness
Contrast

Also available as VSCode Extension



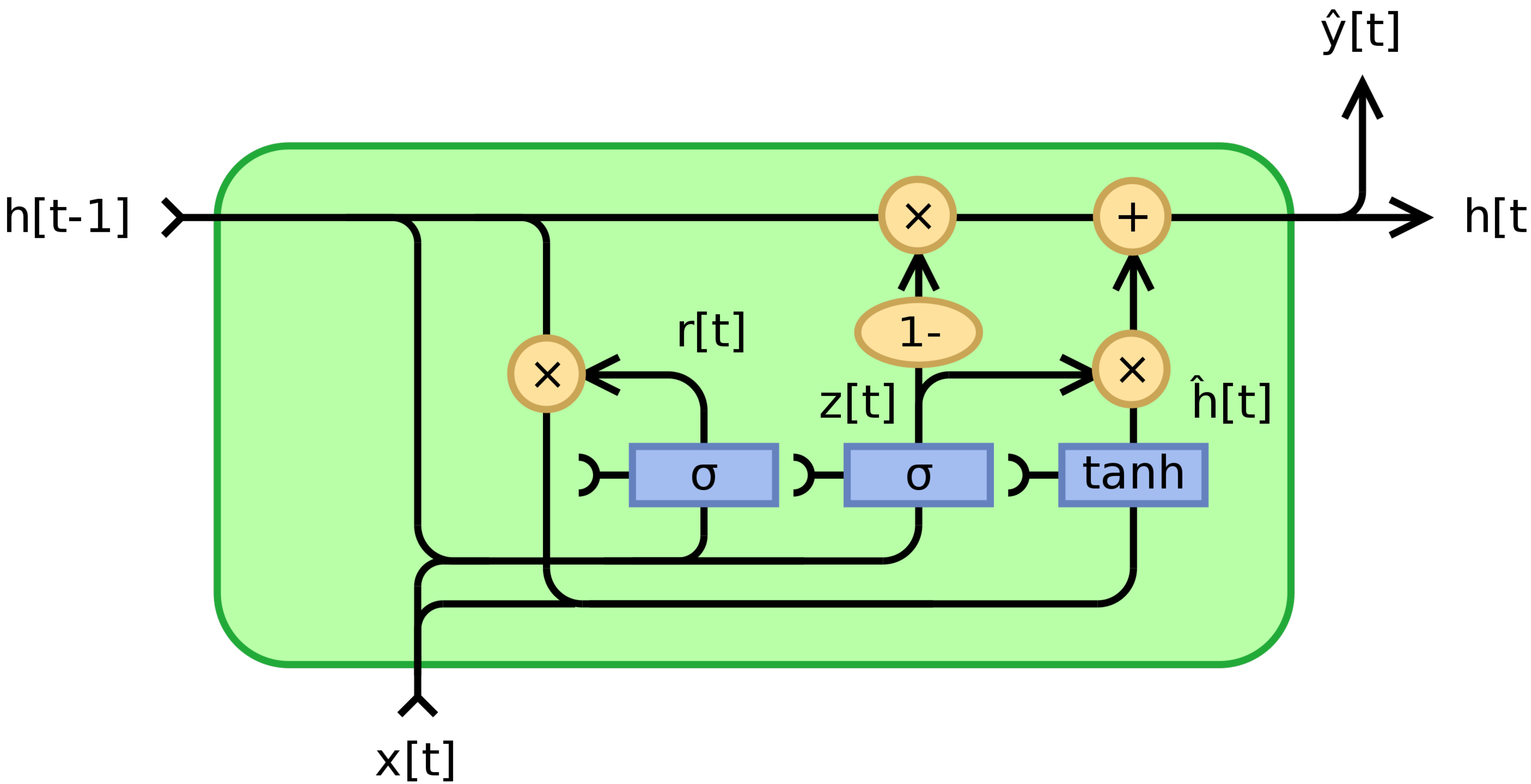
Adding different node types

Adding different node types…

- So far we have simply used
    - Dense nodes
    - Embeddings
- Deep learning architectures can comprise many more types of nodes, and, in general, configuration
    - Convolutional, recurrent
- Since for text processing we want to recognize sequences, we may try the use of recurrent nodes

Gated Recurrent Units (GRU)

Type of recorrent node able to memorize relatively long sequences



Notes

- The node takes as input:
  - The previous state h[t-1]
  - The input x[t]
- It produces as output
  - The current state h[t]
  - The output $\hat{y}$[t]
- We won’t analyze its internal structure in detail

Network with GRU nodes

```
model = models.Sequential()
model.add(layers.Embedding(
    voc_len,
    EMBEDDING_SIZE,
    embeddings_initializer=keras.initializers.Constant(embedding_matrix),
    trainable=True))
model.add(layers.Dropout(DROP_RATE))
model.add(layers.GRU(128,return_sequences=True,dropout=DROP_RATE,recurrent_dropout=DROP_RATE))
model.add(layers.GRU(128,dropout=DROP_RATE,recurrent_dropout=DROP_RATE))
model.add(layers.Dense(1, activation='sigmoid'))
```

Notes

- No flattening after embedding necessary
- The node has two dropouts, a regular dropout applied on the input x[t] weight, and a recurrent_dropout applied on the h[t-1] weight
- The rest of the code is the same as usual