amazon

April 25, 2024

1 Practice 2.2. Recurrent Neural Networks

- Alejandro Dopico Castro (alejandro.dopico2@udc.es).
- Ana Xiangning Pereira Ezquerro (ana.ezquerro@udc.es).

This notebook contains execution examples of the recurrent neural architectures proposed for the Amazon Reviews dataset. The Python scripts submitted include auxiliar code to simplify the readibility of the coding cells:

- data.py: Defines the `AmazonDataset` class to load, split, transform and stream the Amazon Reviews dataset.
- recurrent_models.py: Defines the `create_recurrent_model` function to instantiate a Keras model varying its architecture.
- utils.py: Defines auxiliary function to train and plot the performance of a Keras model.

[ ]:
```
from data import AmazonDataset
from model import AmazonReviewsModel
import plotly.io as pio
import plotly.graph_objects as go
from collections import OrderedDict
from keras.layers import LSTM, GRU, SimpleRNN
from keras.regularizers import Regularizer, L1, L2, L1L2
from keras.optimizers import Adam, RMSprop
import pandas as pd
from itertools import product

pio.renderers.default = "vscode"

# global parameters
MAX_FEATURES = 1000
MODEL_PATH = "results/"

# model default parameters
train_default = dict(epochs=30, batch_size=500, lr=1e-3, dev_patience=5)

# load data
path_dir = "AmazonDataset/"
```

```
2024-04-24 19:10:42.133173: I tensorflow/core/util/port.cc:113] oneDNN custom
```

```
operations are on. You may see slightly different numerical results due to
floating-point round-off errors from different computation orders. To turn them
off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.
2024-04-24 19:10:42.165187: E
external/local_xla/xla/stream_executor/cuda/cuda_dnn.cc:9261] Unable to register
cuDNN factory: Attempting to register factory for plugin cuDNN when one has
already been registered
2024-04-24 19:10:42.165214: E
external/local_xla/xla/stream_executor/cuda/cuda_fft.cc:607] Unable to register
cuFFT factory: Attempting to register factory for plugin cuFFT when one has
already been registered
2024-04-24 19:10:42.166077: E
external/local_xla/xla/stream_executor/cuda/cuda_blas.cc:1515] Unable to
register cuBLAS factory: Attempting to register factory for plugin cuBLAS when
one has already been registered
2024-04-24 19:10:42.171268: I tensorflow/core/platform/cpu_feature_guard.cc:182]
This TensorFlow binary is optimized to use available CPU instructions in
performance-critical operations.
To enable the following instructions: AVX2 AVX512F AVX512_VNNI FMA, in other
operations, rebuild TensorFlow with the appropriate compiler flags.
2024-04-24 19:10:42.814324: W
tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:38] TF-TRT Warning: Could not
find TensorRT

Using TensorFlow backend
```

1.1 Table of Contents

1.2 Simple Recurrent Baseline

1.2.1 Exploring the vocabulary size

We used a simple recurrent architecture to set our baseline performance. This model is conformed by two stacked modules: a recurrent encoder of 2-stacked RNN cells (Rumelhart et al., 1985) and a feed-forward layer with a sigmoidal activation to return the probability of a good review. We used an input embedding layer of dimension $d_x = 64$ and maintained the dimension of the decoder to

$d_h = 64$. In order to analyze the impact of the vocabulary size ($|\mathcal{V}|$) we repeated three experiments varying this value (200, 500 and 1000) maintaining the same architecture.

[ ]:
```python
dataset = AmazonDataset.load(
    train_path=path_dir + "train_small.txt",
    test_path=path_dir + "test_small.txt",
    max_features=200,
)
rnn_model_200 = AmazonReviewsModel(200, 64, SimpleRNN, name="SimpleRNN-200")
_, fig = rnn_model_200.train(
    dataset, f"{MODEL_PATH}/{rnn_model_200.name}.weights.h5", **train_default
)
print(rnn_model_200.evaluate(dataset.X_test, dataset.y_test))
fig
```

```
Epoch 1/30
40/40 [==============================] - 6s 118ms/step - loss: 0.6899 -
accuracy: 0.5311 - val_loss: 0.6742 - val_accuracy: 0.5920
Epoch 2/30
40/40 [==============================] - 5s 114ms/step - loss: 0.6158 -
accuracy: 0.6568 - val_loss: 0.5202 - val_accuracy: 0.7546
Epoch 3/30
40/40 [==============================] - 4s 112ms/step - loss: 0.4883 -
accuracy: 0.7683 - val_loss: 0.4642 - val_accuracy: 0.7854
Epoch 4/30
40/40 [==============================] - 4s 112ms/step - loss: 0.4690 -
accuracy: 0.7796 - val_loss: 0.4746 - val_accuracy: 0.7714
Epoch 5/30
40/40 [==============================] - 5s 114ms/step - loss: 0.4403 -
accuracy: 0.7953 - val_loss: 0.4714 - val_accuracy: 0.7708
Epoch 6/30
40/40 [==============================] - 4s 112ms/step - loss: 0.4447 -
accuracy: 0.7944 - val_loss: 0.4746 - val_accuracy: 0.7782
Epoch 7/30
40/40 [==============================] - 5s 113ms/step - loss: 0.4172 -
accuracy: 0.8105 - val_loss: 0.5014 - val_accuracy: 0.7762
Epoch 8/30
40/40 [==============================] - 4s 110ms/step - loss: 0.4061 -
accuracy: 0.8179 - val_loss: 0.4991 - val_accuracy: 0.7748
[0.48217952251434326, 0.7735999822616577]
```

[ ]:
```python
dataset = AmazonDataset.load(
    train_path=path_dir + "train_small.txt",
    test_path=path_dir + "test_small.txt",
    max_features=500,
)
rnn_model_500 = AmazonReviewsModel(500, 64, SimpleRNN, name="SimpleRNN-500")
_, fig = rnn_model_500.train(
```

```python
    dataset, f"{MODEL_PATH}/{rnn_model_500.name}.weights.h5", **train_default
)
print(rnn_model_500.evaluate(dataset.X_test, dataset.y_test))
fig
```

```
Epoch 1/30
40/40 [==============================] - 6s 119ms/step - loss: 0.6689 -
accuracy: 0.5717 - val_loss: 0.6603 - val_accuracy: 0.6734
Epoch 2/30
40/40 [==============================] - 4s 112ms/step - loss: 0.6506 -
accuracy: 0.6578 - val_loss: 0.6548 - val_accuracy: 0.6178
Epoch 3/30
40/40 [==============================] - 4s 112ms/step - loss: 0.6311 -
accuracy: 0.6474 - val_loss: 0.6222 - val_accuracy: 0.6550
Epoch 4/30
40/40 [==============================] - 4s 113ms/step - loss: 0.6069 -
accuracy: 0.6744 - val_loss: 0.6088 - val_accuracy: 0.6636
Epoch 5/30
40/40 [==============================] - 5s 114ms/step - loss: 0.5954 -
accuracy: 0.6978 - val_loss: 0.5939 - val_accuracy: 0.6870
Epoch 6/30
40/40 [==============================] - 4s 112ms/step - loss: 0.5589 -
accuracy: 0.7225 - val_loss: 0.5742 - val_accuracy: 0.6980
Epoch 7/30
40/40 [==============================] - 5s 113ms/step - loss: 0.5122 -
accuracy: 0.7548 - val_loss: 0.4867 - val_accuracy: 0.7754
Epoch 8/30
40/40 [==============================] - 4s 112ms/step - loss: 0.5314 -
accuracy: 0.7502 - val_loss: 0.5797 - val_accuracy: 0.7086
Epoch 9/30
40/40 [==============================] - 4s 111ms/step - loss: 0.5042 -
accuracy: 0.7591 - val_loss: 0.4855 - val_accuracy: 0.7780
Epoch 10/30
40/40 [==============================] - 4s 111ms/step - loss: 0.5151 -
accuracy: 0.7463 - val_loss: 0.5741 - val_accuracy: 0.6970
Epoch 11/30
40/40 [==============================] - 4s 113ms/step - loss: 0.4502 -
accuracy: 0.7919 - val_loss: 0.5263 - val_accuracy: 0.7556
Epoch 12/30
40/40 [==============================] - 4s 110ms/step - loss: 0.4622 -
accuracy: 0.7926 - val_loss: 0.5112 - val_accuracy: 0.7694
Epoch 13/30
40/40 [==============================] - 4s 112ms/step - loss: 0.4003 -
accuracy: 0.8293 - val_loss: 0.4909 - val_accuracy: 0.7754
Epoch 14/30
40/40 [==============================] - 4s 111ms/step - loss: 0.3560 -
accuracy: 0.8507 - val_loss: 0.4598 - val_accuracy: 0.8122
```

```
Epoch 15/30
40/40 [==============================] - 5s 113ms/step - loss: 0.3288 -
accuracy: 0.8641 - val_loss: 0.4374 - val_accuracy: 0.8068
Epoch 16/30
40/40 [==============================] - 4s 112ms/step - loss: 0.2969 -
accuracy: 0.8790 - val_loss: 0.4820 - val_accuracy: 0.8034
Epoch 17/30
40/40 [==============================] - 4s 112ms/step - loss: 0.2865 -
accuracy: 0.8835 - val_loss: 0.4981 - val_accuracy: 0.8060
Epoch 18/30
40/40 [==============================] - 4s 112ms/step - loss: 0.2603 -
accuracy: 0.8954 - val_loss: 0.5172 - val_accuracy: 0.7932
Epoch 19/30
40/40 [==============================] - 4s 112ms/step - loss: 0.2427 -
accuracy: 0.9044 - val_loss: 0.5969 - val_accuracy: 0.7484
Epoch 20/30
40/40 [==============================] - 5s 114ms/step - loss: 0.2275 -
accuracy: 0.9116 - val_loss: 0.6160 - val_accuracy: 0.7560
[0.43902644515037537, 0.8077200055122375]
```

[ ]:
```
dataset = AmazonDataset.load(
    train_path=path_dir + "train_small.txt",
    test_path=path_dir + "test_small.txt",
    max_features=1000,
)
rnn_model_1000 = AmazonReviewsModel(1000, 64, SimpleRNN, name="SimpleRNN-1000")
_, fig = rnn_model_1000.train(
    dataset, f"{MODEL_PATH}/{rnn_model_1000.name}.weights.h5", **train_default
)
print(rnn_model_1000.evaluate(dataset.X_test, dataset.y_test))
fig
```

```
Epoch 1/30
40/40 [==============================] - 7s 139ms/step - loss: 0.6900 -
accuracy: 0.5350 - val_loss: 0.6921 - val_accuracy: 0.5282
Epoch 2/30
40/40 [==============================] - 5s 121ms/step - loss: 0.6729 -
accuracy: 0.5847 - val_loss: 0.6896 - val_accuracy: 0.5198
Epoch 3/30
40/40 [==============================] - 5s 117ms/step - loss: 0.6638 -
accuracy: 0.6130 - val_loss: 0.6522 - val_accuracy: 0.6026
Epoch 4/30
40/40 [==============================] - 5s 113ms/step - loss: 0.5547 -
accuracy: 0.7263 - val_loss: 0.4817 - val_accuracy: 0.7944
Epoch 5/30
40/40 [==============================] - 5s 115ms/step - loss: 0.5013 -
accuracy: 0.7674 - val_loss: 0.4722 - val_accuracy: 0.7962
Epoch 6/30
```

```
40/40 [==============================] - 5s 116ms/step - loss: 0.3819 -
accuracy: 0.8366 - val_loss: 0.3675 - val_accuracy: 0.8376
Epoch 7/30
40/40 [==============================] - 5s 115ms/step - loss: 0.3061 -
accuracy: 0.8751 - val_loss: 0.3527 - val_accuracy: 0.8548
Epoch 8/30
40/40 [==============================] - 5s 118ms/step - loss: 0.2664 -
accuracy: 0.8927 - val_loss: 0.3515 - val_accuracy: 0.8504
Epoch 9/30
40/40 [==============================] - 5s 115ms/step - loss: 0.2352 -
accuracy: 0.9085 - val_loss: 0.3588 - val_accuracy: 0.8488
Epoch 10/30
40/40 [==============================] - 5s 115ms/step - loss: 0.1969 -
accuracy: 0.9273 - val_loss: 0.3843 - val_accuracy: 0.8442
Epoch 11/30
40/40 [==============================] - 5s 114ms/step - loss: 0.1649 -
accuracy: 0.9412 - val_loss: 0.4074 - val_accuracy: 0.8352
Epoch 12/30
40/40 [==============================] - 5s 114ms/step - loss: 0.1319 -
accuracy: 0.9560 - val_loss: 0.4710 - val_accuracy: 0.8318
Epoch 13/30
40/40 [==============================] - 5s 118ms/step - loss: 0.1199 -
accuracy: 0.9600 - val_loss: 0.4976 - val_accuracy: 0.8170
[0.37227746844291687, 0.8421199917793274]
```

The simple RNN cell with $|\mathcal{V}| = 1000$ achieves 84.21% of test accuracy and is capable of learning from 95% of the training data. It can be observed that the vocabulary size plays an important role in the performance of the model: when using a small vocabulary size (e.g. 200) the performance does not reach more than the 80% of the accuracy. By increasing its value up to $|\mathcal{V}| = 1000$ we see a five-points improvement in the evaluation set. However, this improvement comes with an aggravation in the test performance: the larger $|\mathcal{V}|$ value, the larger difference between the train and test accuracy. This phenomenon (overfitting) is likely due to the increased complexity and dimensionality of the input data, which can challenge the model's ability to generalize effectively.

[ ]:
```
# change name
rnn_model = rnn_model_1000
rnn_model._name = 'SimpleRNN-base'
```

1.2.2 Exploring the recurrent cell

In the next cells we maintain $|\mathcal{V}| = 1000$ and substitute the simple RNN by two different recurrent cells: the LSTM (Hochreiter et al., 1997) and the GRU (Chung et al., 2014). In the original papers, authors claimed to improve the performance of the simple RNN with a better inner representation of the temporal data flow by the introduction of different gates modeled with different learnable weights.

[ ]:
```
lstm_model = AmazonReviewsModel(1000, 64, LSTM, name="LSTM-base")
_, fig = lstm_model.train(
```

```
    dataset, f"{MODEL_PATH}/{lstm_model.name}.weights.h5", **train_default
)
print(lstm_model.evaluate(dataset.X_test, dataset.y_test))
fig
```

```
Epoch 1/30

2024-04-24 18:58:46.402658: W
tensorflow/core/common_runtime/type_inference.cc:339] Type inference failed.
This indicates an invalid graph that escaped type checking. Error message:
INVALID_ARGUMENT: expected compatible input types, but input 1:
type_id: TFT_OPTIONAL
args {
  type_id: TFT_PRODUCT
  args {
    type_id: TFT_TENSOR
    args {
      type_id: TFT_INT32
    }
  }
}
 is neither a subtype nor a supertype of the combined inputs preceding it:
type_id: TFT_OPTIONAL
args {
  type_id: TFT_PRODUCT
  args {
    type_id: TFT_TENSOR
    args {
      type_id: TFT_FLOAT
    }
  }
}

        for Tuple type infernce function 0
        while inferring type of node 'cond_19/output/_22'

40/40 [==============================] - 7s 79ms/step - loss: 0.5946 - accuracy:
0.6716 - val_loss: 0.4730 - val_accuracy: 0.7782
Epoch 2/30
40/40 [==============================] - 1s 30ms/step - loss: 0.3743 - accuracy:
0.8403 - val_loss: 0.3549 - val_accuracy: 0.8512
Epoch 3/30
40/40 [==============================] - 1s 27ms/step - loss: 0.3270 - accuracy:
0.8675 - val_loss: 0.3276 - val_accuracy: 0.8638
Epoch 4/30
40/40 [==============================] - 1s 27ms/step - loss: 0.3157 - accuracy:
0.8721 - val_loss: 0.3316 - val_accuracy: 0.8648
Epoch 5/30
```

```
40/40 [==============================] - 1s 25ms/step - loss: 0.3096 - accuracy:
0.8756 - val_loss: 0.3306 - val_accuracy: 0.8608
Epoch 6/30
40/40 [==============================] - 1s 25ms/step - loss: 0.3069 - accuracy:
0.8774 - val_loss: 0.3321 - val_accuracy: 0.8598
Epoch 7/30
40/40 [==============================] - 1s 25ms/step - loss: 0.2997 - accuracy:
0.8795 - val_loss: 0.3241 - val_accuracy: 0.8628
Epoch 8/30
40/40 [==============================] - 1s 26ms/step - loss: 0.2989 - accuracy:
0.8805 - val_loss: 0.3214 - val_accuracy: 0.8670
Epoch 9/30
40/40 [==============================] - 1s 24ms/step - loss: 0.2971 - accuracy:
0.8810 - val_loss: 0.3411 - val_accuracy: 0.8558
Epoch 10/30
40/40 [==============================] - 1s 24ms/step - loss: 0.3131 - accuracy:
0.8716 - val_loss: 0.3350 - val_accuracy: 0.8638
Epoch 11/30
40/40 [==============================] - 1s 25ms/step - loss: 0.2953 - accuracy:
0.8795 - val_loss: 0.3160 - val_accuracy: 0.8708
Epoch 12/30
40/40 [==============================] - 1s 23ms/step - loss: 0.2809 - accuracy:
0.8880 - val_loss: 0.3319 - val_accuracy: 0.8670
Epoch 13/30
40/40 [==============================] - 1s 23ms/step - loss: 0.2774 - accuracy:
0.8895 - val_loss: 0.3457 - val_accuracy: 0.8530
Epoch 14/30
40/40 [==============================] - 1s 24ms/step - loss: 0.3076 - accuracy:
0.8713 - val_loss: 0.3345 - val_accuracy: 0.8524
Epoch 15/30
40/40 [==============================] - 1s 24ms/step - loss: 0.2784 - accuracy:
0.8863 - val_loss: 0.3207 - val_accuracy: 0.8632
Epoch 16/30
40/40 [==============================] - 1s 27ms/step - loss: 0.2676 - accuracy:
0.8921 - val_loss: 0.3291 - val_accuracy: 0.8672
[0.3445417881011963, 0.8532000184059143]
```

[ ]:
```python
gru_model = AmazonReviewsModel(1000, 64, GRU, name="GRU-base")
_, fig = gru_model.train(
    dataset, f"{MODEL_PATH}/{gru_model.name}.weights.h5", **train_default
)
print(gru_model.evaluate(dataset.X_test, dataset.y_test))
fig
```

```
Epoch 1/30
40/40 [==============================] - 6s 77ms/step - loss: 0.6378 - accuracy:
0.6263 - val_loss: 0.4943 - val_accuracy: 0.7614
Epoch 2/30
```

```
40/40 [==============================] - 1s 25ms/step - loss: 0.3941 - accuracy:
0.8274 - val_loss: 0.3439 - val_accuracy: 0.8530
Epoch 3/30
40/40 [==============================] - 1s 23ms/step - loss: 0.3350 - accuracy:
0.8609 - val_loss: 0.3360 - val_accuracy: 0.8612
Epoch 4/30
40/40 [==============================] - 1s 22ms/step - loss: 0.3102 - accuracy:
0.8738 - val_loss: 0.3183 - val_accuracy: 0.8696
Epoch 5/30
40/40 [==============================] - 1s 24ms/step - loss: 0.3060 - accuracy:
0.8742 - val_loss: 0.3238 - val_accuracy: 0.8616
Epoch 6/30
40/40 [==============================] - 1s 21ms/step - loss: 0.2952 - accuracy:
0.8809 - val_loss: 0.3225 - val_accuracy: 0.8684
Epoch 7/30
40/40 [==============================] - 1s 25ms/step - loss: 0.2935 - accuracy:
0.8821 - val_loss: 0.3155 - val_accuracy: 0.8674
Epoch 8/30
40/40 [==============================] - 1s 22ms/step - loss: 0.2884 - accuracy:
0.8853 - val_loss: 0.3213 - val_accuracy: 0.8654
Epoch 9/30
40/40 [==============================] - 1s 22ms/step - loss: 0.2813 - accuracy:
0.8885 - val_loss: 0.3191 - val_accuracy: 0.8674
Epoch 10/30
40/40 [==============================] - 1s 21ms/step - loss: 0.2774 - accuracy:
0.8893 - val_loss: 0.3182 - val_accuracy: 0.8690
Epoch 11/30
40/40 [==============================] - 1s 22ms/step - loss: 0.2697 - accuracy:
0.8951 - val_loss: 0.3323 - val_accuracy: 0.8634
Epoch 12/30
40/40 [==============================] - 1s 24ms/step - loss: 0.2653 - accuracy:
0.8964 - val_loss: 0.3296 - val_accuracy: 0.8708
[0.33640894293785095, 0.8580800294876099]
```

Using the same architecture but only replacing the simple RNN layer by LSTMs or GRUs, we see that the performance reaches the 85.32% and 85.80% of accuracy, respectively, proving that the LSTM and GRU cells are better options for the baseline recurrent architecture than the simple RNN cell.

1.2.3 Exploring the dimension of the model (d_h)

In the next cells we maitain the vocabulary size ($|\mathcal{V}| = 1000$) and the type of recurrent cell (LSTM) to explore the impact of the model dimension $d_h \in \{64, 128, 256, 512\}$.

[ ]:
```
lstm_model_128 = AmazonReviewsModel(1000, 128, LSTM, name="LSTM-128")
_, fig = lstm_model_128.train(
    dataset, f"{MODEL_PATH}/{lstm_model_128.name}.weights.h5", **train_default
)
```

```python
print(lstm_model_128.evaluate(dataset.X_test, dataset.y_test))
fig
```

```
Epoch 1/30

2024-04-24 19:29:05.890379: W
tensorflow/core/common_runtime/type_inference.cc:339] Type inference failed.
This indicates an invalid graph that escaped type checking. Error message:
INVALID_ARGUMENT: expected compatible input types, but input 1:
type_id: TFT_OPTIONAL
args {
  type_id: TFT_PRODUCT
  args {
    type_id: TFT_TENSOR
    args {
      type_id: TFT_INT32
    }
  }
}
 is neither a subtype nor a supertype of the combined inputs preceding it:
type_id: TFT_OPTIONAL
args {
  type_id: TFT_PRODUCT
  args {
    type_id: TFT_TENSOR
    args {
      type_id: TFT_FLOAT
    }
  }
}

        for Tuple type infernce function 0
        while inferring type of node 'cond_19/output/_22'

40/40 [==============================] - 6s 69ms/step - loss: 0.5617 - accuracy:
0.7010 - val_loss: 0.4028 - val_accuracy: 0.8232
Epoch 2/30
40/40 [==============================] - 1s 24ms/step - loss: 0.3650 - accuracy:
0.8439 - val_loss: 0.4405 - val_accuracy: 0.7978
Epoch 3/30
40/40 [==============================] - 1s 27ms/step - loss: 0.3433 - accuracy:
0.8550 - val_loss: 0.3478 - val_accuracy: 0.8472
Epoch 4/30
40/40 [==============================] - 1s 22ms/step - loss: 0.3256 - accuracy:
0.8666 - val_loss: 0.3540 - val_accuracy: 0.8560
Epoch 5/30
40/40 [==============================] - 1s 25ms/step - loss: 0.3175 - accuracy:
0.8703 - val_loss: 0.3483 - val_accuracy: 0.8504
```

```
Epoch 6/30
40/40 [==============================] - 1s 23ms/step - loss: 0.3061 - accuracy:
0.8761 - val_loss: 0.3424 - val_accuracy: 0.8612
Epoch 7/30
40/40 [==============================] - 1s 23ms/step - loss: 0.2913 - accuracy:
0.8824 - val_loss: 0.3304 - val_accuracy: 0.8598
Epoch 8/30
40/40 [==============================] - 1s 22ms/step - loss: 0.2929 - accuracy:
0.8819 - val_loss: 0.3350 - val_accuracy: 0.8570
Epoch 9/30
40/40 [==============================] - 1s 24ms/step - loss: 0.3742 - accuracy:
0.8339 - val_loss: 0.5078 - val_accuracy: 0.7570
Epoch 10/30
40/40 [==============================] - 1s 22ms/step - loss: 0.3793 - accuracy:
0.8378 - val_loss: 0.3586 - val_accuracy: 0.8434
Epoch 11/30
40/40 [==============================] - 1s 25ms/step - loss: 0.3147 - accuracy:
0.8719 - val_loss: 0.3618 - val_accuracy: 0.8526
Epoch 12/30
40/40 [==============================] - 1s 23ms/step - loss: 0.3136 - accuracy:
0.8748 - val_loss: 0.3603 - val_accuracy: 0.8448
[0.34408462047576904, 0.8536400198936462]
```

[ ]:
```
lstm_model_256 = AmazonReviewsModel(1000, 256, LSTM, name="LSTM-256")
_, fig = lstm_model_256.train(
    dataset, f"{MODEL_PATH}/{lstm_model_256.name}.weights.h5", **train_default
)
print(lstm_model_256.evaluate(dataset.X_test, dataset.y_test))
fig
```

```
Epoch 1/30
40/40 [==============================] - 7s 79ms/step - loss: 0.5691 - accuracy:
0.6941 - val_loss: 0.3862 - val_accuracy: 0.8254
Epoch 2/30
40/40 [==============================] - 2s 43ms/step - loss: 0.3544 - accuracy:
0.8518 - val_loss: 0.3482 - val_accuracy: 0.8610
Epoch 3/30
40/40 [==============================] - 2s 38ms/step - loss: 0.3213 - accuracy:
0.8670 - val_loss: 0.3659 - val_accuracy: 0.8324
Epoch 4/30
40/40 [==============================] - 2s 40ms/step - loss: 0.3264 - accuracy:
0.8670 - val_loss: 0.3453 - val_accuracy: 0.8592
Epoch 5/30
40/40 [==============================] - 2s 40ms/step - loss: 0.3065 - accuracy:
0.8789 - val_loss: 0.3260 - val_accuracy: 0.8618
Epoch 6/30
40/40 [==============================] - 2s 39ms/step - loss: 0.2892 - accuracy:
0.8841 - val_loss: 0.3264 - val_accuracy: 0.8620
```

```
Epoch 7/30
40/40 [==============================] - 2s 39ms/step - loss: 0.2819 - accuracy:
0.8862 - val_loss: 0.3265 - val_accuracy: 0.8574
Epoch 8/30
40/40 [==============================] - 2s 38ms/step - loss: 0.2732 - accuracy:
0.8914 - val_loss: 0.3398 - val_accuracy: 0.8520
Epoch 9/30
40/40 [==============================] - 2s 38ms/step - loss: 0.2615 - accuracy:
0.8954 - val_loss: 0.3311 - val_accuracy: 0.8646
Epoch 10/30
40/40 [==============================] - 2s 38ms/step - loss: 0.2470 - accuracy:
0.9007 - val_loss: 0.3529 - val_accuracy: 0.8564
[0.3391265571117401, 0.8568800091743469]
```

[ ]:
```
lstm_model_512 = AmazonReviewsModel(1000, 512, LSTM, name="LSTM-512")
_, fig = lstm_model_512.train(
    dataset, f"{MODEL_PATH}/{lstm_model_512.name}.weights.h5", **train_default
)
print(lstm_model_512.evaluate(dataset.X_test, dataset.y_test))
fig
```

```
Epoch 1/30
40/40 [==============================] - 9s 130ms/step - loss: 0.6260 -
accuracy: 0.6507 - val_loss: 0.4816 - val_accuracy: 0.7676
Epoch 2/30
40/40 [==============================] - 4s 96ms/step - loss: 0.3704 - accuracy:
0.8418 - val_loss: 0.3370 - val_accuracy: 0.8572
Epoch 3/30
40/40 [==============================] - 3s 85ms/step - loss: 0.3877 - accuracy:
0.8385 - val_loss: 0.3795 - val_accuracy: 0.8504
Epoch 4/30
40/40 [==============================] - 4s 88ms/step - loss: 0.3294 - accuracy:
0.8608 - val_loss: 0.3414 - val_accuracy: 0.8528
Epoch 5/30
40/40 [==============================] - 3s 87ms/step - loss: 0.3105 - accuracy:
0.8724 - val_loss: 0.3559 - val_accuracy: 0.8568
Epoch 6/30
40/40 [==============================] - 3s 85ms/step - loss: 0.3070 - accuracy:
0.8723 - val_loss: 0.3410 - val_accuracy: 0.8522
Epoch 7/30
40/40 [==============================] - 3s 84ms/step - loss: 0.3132 - accuracy:
0.8714 - val_loss: 0.3417 - val_accuracy: 0.8542
[0.34632524847984314, 0.8533200025558472]
```

The next table shows the results with different hidden dimension (d_h) in the train, validation and test set:

d_h	train	val	test
64	89.21	86.72	85.32
128	87.48	84.48	85.36
256	90.07	85.64	85.68
512	87.14	85.42	85.33

Results show that there are not significative differences between the performance of different model dimensions, which might indicate that, in order to increase the model complexity (and hence the flexibility to learn the input data), instead of exploring the hyperparameter d_h, other hyperparameters should be tuned, such as the number of hidden layers.

1.3 Enhancing the architecture with hyperparameter tuning

Once we have a first estimation of the performance with small models we are going launch experiments with larger architectures. We increased the model dimension to $d_h = 128$ and the vocabulary size to $|\mathcal{V}| = 2000$. The encoder is now conformed by 3-stacked recurrent cells and the decoder adds a new extra feed-forward network between the last state of the encoder and the output layer. In order to balance this enhancement and avoid a possible overfitting, we included a dropout of the 10% in the latent space of the network (between the encoder and decoder).

[ ]:
```
# relaad the dataset
dataset = AmazonDataset.load(
    train_path=path_dir + "train_small.txt",
    test_path=path_dir + "test_small.txt",
    max_features=2000,
)
```

[ ]:
```
rnn_enhanced = AmazonReviewsModel(
    2000,
    256,
    SimpleRNN,
    num_recurrent_layers=3,
    dropout=0.1,
    ffn_dims=[64],
    name="SimpleRNN-enhanced",
)
_, fig = rnn_enhanced.train(
    dataset, f"{MODEL_PATH}/{rnn_enhanced.name}.weights.h5", **train_default
)
print(rnn_enhanced.evaluate(dataset.X_test, dataset.y_test))
fig
```

```
Epoch 1/30
40/40 [==============================] - 12s 227ms/step - loss: 0.7327 -
accuracy: 0.5100 - val_loss: 0.6983 - val_accuracy: 0.4852
Epoch 2/30
40/40 [==============================] - 8s 200ms/step - loss: 0.7181 -
```

```
accuracy: 0.5031 - val_loss: 0.6941 - val_accuracy: 0.4850
Epoch 3/30
40/40 [==============================] - 7s 183ms/step - loss: 0.7099 -
accuracy: 0.4997 - val_loss: 0.6934 - val_accuracy: 0.4850
Epoch 4/30
40/40 [==============================] - 7s 183ms/step - loss: 0.7115 -
accuracy: 0.4981 - val_loss: 0.7037 - val_accuracy: 0.4850
Epoch 5/30
40/40 [==============================] - 7s 175ms/step - loss: 0.7046 -
accuracy: 0.5084 - val_loss: 0.7043 - val_accuracy: 0.5152
Epoch 6/30
40/40 [==============================] - 7s 178ms/step - loss: 0.7054 -
accuracy: 0.5039 - val_loss: 0.6944 - val_accuracy: 0.5152
Epoch 7/30
40/40 [==============================] - 7s 172ms/step - loss: 0.7021 -
accuracy: 0.5038 - val_loss: 0.7071 - val_accuracy: 0.5152
Epoch 8/30
40/40 [==============================] - 7s 177ms/step - loss: 0.7010 -
accuracy: 0.5022 - val_loss: 0.6925 - val_accuracy: 0.5152
Epoch 9/30
40/40 [==============================] - 7s 174ms/step - loss: 0.7073 -
accuracy: 0.4942 - val_loss: 0.7006 - val_accuracy: 0.5006
Epoch 10/30
40/40 [==============================] - 7s 175ms/step - loss: 0.7039 -
accuracy: 0.5087 - val_loss: 0.7035 - val_accuracy: 0.5696
Epoch 11/30
40/40 [==============================] - 7s 176ms/step - loss: 0.6971 -
accuracy: 0.5375 - val_loss: 0.6981 - val_accuracy: 0.5152
Epoch 12/30
40/40 [==============================] - 7s 174ms/step - loss: 0.6962 -
accuracy: 0.5379 - val_loss: 0.6862 - val_accuracy: 0.5682
Epoch 13/30
40/40 [==============================] - 7s 172ms/step - loss: 0.6937 -
accuracy: 0.5397 - val_loss: 0.6941 - val_accuracy: 0.4848
Epoch 14/30
40/40 [==============================] - 7s 172ms/step - loss: 0.6938 -
accuracy: 0.5415 - val_loss: 0.6847 - val_accuracy: 0.5682
Epoch 15/30
40/40 [==============================] - 7s 173ms/step - loss: 0.6916 -
accuracy: 0.5432 - val_loss: 0.6867 - val_accuracy: 0.5682
Epoch 16/30
40/40 [==============================] - 7s 173ms/step - loss: 0.6904 -
accuracy: 0.5455 - val_loss: 0.6859 - val_accuracy: 0.5682
Epoch 17/30
40/40 [==============================] - 7s 174ms/step - loss: 0.6935 -
accuracy: 0.5350 - val_loss: 0.6843 - val_accuracy: 0.5682
Epoch 18/30
40/40 [==============================] - 7s 171ms/step - loss: 0.6995 -
```

```
accuracy: 0.5361 - val_loss: 0.6883 - val_accuracy: 0.5682
Epoch 19/30
40/40 [==============================] - 7s 173ms/step - loss: 0.6907 -
accuracy: 0.5480 - val_loss: 0.6856 - val_accuracy: 0.5682
Epoch 20/30
40/40 [==============================] - 7s 171ms/step - loss: 0.6904 -
accuracy: 0.5500 - val_loss: 0.6880 - val_accuracy: 0.5682
Epoch 21/30
40/40 [==============================] - 7s 171ms/step - loss: 0.6920 -
accuracy: 0.5397 - val_loss: 0.6924 - val_accuracy: 0.4848
Epoch 22/30
40/40 [==============================] - 7s 173ms/step - loss: 0.6919 -
accuracy: 0.5444 - val_loss: 0.6855 - val_accuracy: 0.5682
[0.6853593587875366, 0.5634400248527527]
```

[ ]:
```python
lstm_enhanced = AmazonReviewsModel(
    2000,
    256,
    LSTM,
    num_recurrent_layers=3,
    dropout=0.1,
    ffn_dims=[64],
    name="LSTM-enhanced",
)
_, fig = lstm_enhanced.train(
    dataset, f"{MODEL_PATH}/{lstm_enhanced.name}.weights.h5", **train_default
)
print(lstm_enhanced.evaluate(dataset.X_test, dataset.y_test))
fig
```

```
Epoch 1/30
40/40 [==============================] - 11s 135ms/step - loss: 0.5214 -
accuracy: 0.7262 - val_loss: 0.3757 - val_accuracy: 0.8596
Epoch 2/30
40/40 [==============================] - 3s 70ms/step - loss: 0.3160 - accuracy:
0.8762 - val_loss: 0.3212 - val_accuracy: 0.8722
Epoch 3/30
40/40 [==============================] - 2s 62ms/step - loss: 0.2797 - accuracy:
0.8921 - val_loss: 0.3305 - val_accuracy: 0.8696
Epoch 4/30
40/40 [==============================] - 2s 61ms/step - loss: 0.2541 - accuracy:
0.9047 - val_loss: 0.3130 - val_accuracy: 0.8774
Epoch 5/30
40/40 [==============================] - 2s 59ms/step - loss: 0.2315 - accuracy:
0.9155 - val_loss: 0.3047 - val_accuracy: 0.8744
Epoch 6/30
40/40 [==============================] - 2s 52ms/step - loss: 0.2217 - accuracy:
0.9189 - val_loss: 0.3307 - val_accuracy: 0.8644
```

```
Epoch 7/30
40/40 [==============================] - 2s 55ms/step - loss: 0.2023 - accuracy:
0.9265 - val_loss: 0.3419 - val_accuracy: 0.8686
Epoch 8/30
40/40 [==============================] - 2s 52ms/step - loss: 0.1860 - accuracy:
0.9317 - val_loss: 0.3592 - val_accuracy: 0.8732
Epoch 9/30
40/40 [==============================] - 2s 52ms/step - loss: 0.1743 - accuracy:
0.9369 - val_loss: 0.3671 - val_accuracy: 0.8690
Epoch 10/30
40/40 [==============================] - 2s 50ms/step - loss: 0.1612 - accuracy:
0.9426 - val_loss: 0.4251 - val_accuracy: 0.8384
[0.33539456129074097, 0.8589199781417847]
```

[ ]:
```python
gru_enhanced = AmazonReviewsModel(
    2000,
    256,
    GRU,
    num_recurrent_layers=3,
    dropout=0.1,
    ffn_dims=[64],
    name="GRU-enhanced",
)
_, fig = gru_enhanced.train(
    dataset, f"{MODEL_PATH}/{gru_enhanced.name}.weights.h5", **train_default
)
print(gru_enhanced.evaluate(dataset.X_test, dataset.y_test))
fig
```

```
Epoch 1/30
40/40 [==============================] - 10s 133ms/step - loss: 0.5410 -
accuracy: 0.7170 - val_loss: 0.3752 - val_accuracy: 0.8364
Epoch 2/30
40/40 [==============================] - 3s 66ms/step - loss: 0.3445 - accuracy:
0.8564 - val_loss: 0.3118 - val_accuracy: 0.8734
Epoch 3/30
40/40 [==============================] - 2s 58ms/step - loss: 0.2840 - accuracy:
0.8870 - val_loss: 0.3160 - val_accuracy: 0.8724
Epoch 4/30
40/40 [==============================] - 2s 61ms/step - loss: 0.2864 - accuracy:
0.8873 - val_loss: 0.3441 - val_accuracy: 0.8656
Epoch 5/30
40/40 [==============================] - 2s 50ms/step - loss: 0.2412 - accuracy:
0.9096 - val_loss: 0.3092 - val_accuracy: 0.8782
Epoch 6/30
40/40 [==============================] - 2s 50ms/step - loss: 0.2179 - accuracy:
0.9179 - val_loss: 0.3146 - val_accuracy: 0.8784
Epoch 7/30
```

```
40/40 [==============================] - 2s 54ms/step - loss: 0.2077 - accuracy:
0.9223 - val_loss: 0.3446 - val_accuracy: 0.8698
Epoch 8/30
40/40 [==============================] - 2s 49ms/step - loss: 0.1921 - accuracy:
0.9301 - val_loss: 0.3386 - val_accuracy: 0.8716
Epoch 9/30
40/40 [==============================] - 2s 48ms/step - loss: 0.1854 - accuracy:
0.9324 - val_loss: 0.3765 - val_accuracy: 0.8664
Epoch 10/30
40/40 [==============================] - 2s 49ms/step - loss: 0.1562 - accuracy:
0.9446 - val_loss: 0.3274 - val_accuracy: 0.8774
[0.33767375349998474, 0.8643199801445007]
```

We see a slight improvement with the LSTM (85.89%) and GRU-based (86.43%) architectures when increasing the number of learnable hyperparameters (both the train and the test set metrics are improved). However, the simple RNN only reaches 56.34% of accuracy. This drop in the performance evidences the clear superiority of the LSTM and GRU when modelling high-dimensional temporal data. The simple RNN is instead more useful for simpler problems (when the dimension of the model is small, e.g. $d_h = 64$ and $|\mathcal{V}| = 1000$) and we see that when the input increases its complexity the RNN lacks of a good representation to learn temporal relations.

1.4 Bidirectional Processing

In this section we tried to boost the performance of our model with the introduction of bidirectional processing. The Keras API has a Bidirectional Layer which accepts as input a recurrent cell (LSTM, GRU or SimpleRNN) and generates two different cells left-to-right a right-to-left contextualization. The final output is finally obtained via the concatenation of both representations.

[ ]:
```
birnn_model = AmazonReviewsModel(
    2000,
    256,
    SimpleRNN,
    num_recurrent_layers=4,
    dropout=0.15,
    ffn_dims=[128, 64],
    name="BiRNN",
    bidirectional=True,
)
_, fig = birnn_model.train(
    dataset, f"{MODEL_PATH}/{birnn_model.name}.weights.h5", **train_default
)
print(birnn_model.evaluate(dataset.X_test, dataset.y_test))
fig
```

```
Epoch 1/30
40/40 [==============================] - 26s 528ms/step - loss: 0.6121 -
accuracy: 0.6518 - val_loss: 0.4713 - val_accuracy: 0.7720
Epoch 2/30
40/40 [==============================] - 19s 471ms/step - loss: 0.4102 -
```

```
accuracy: 0.8181 - val_loss: 0.4085 - val_accuracy: 0.8166
Epoch 3/30
40/40 [==============================] - 18s 459ms/step - loss: 0.3109 -
accuracy: 0.8720 - val_loss: 0.3340 - val_accuracy: 0.8670
Epoch 4/30
40/40 [==============================] - 18s 456ms/step - loss: 0.3215 -
accuracy: 0.8647 - val_loss: 0.3759 - val_accuracy: 0.8402
Epoch 5/30
40/40 [==============================] - 18s 447ms/step - loss: 0.2792 -
accuracy: 0.8878 - val_loss: 0.3267 - val_accuracy: 0.8638
Epoch 6/30
40/40 [==============================] - 18s 446ms/step - loss: 0.3435 -
accuracy: 0.8586 - val_loss: 0.3409 - val_accuracy: 0.8522
Epoch 7/30
40/40 [==============================] - 18s 450ms/step - loss: 0.2930 -
accuracy: 0.8818 - val_loss: 0.3965 - val_accuracy: 0.8124
Epoch 8/30
40/40 [==============================] - 18s 450ms/step - loss: 0.2881 -
accuracy: 0.8849 - val_loss: 0.3466 - val_accuracy: 0.8544
Epoch 9/30
40/40 [==============================] - 18s 448ms/step - loss: 0.2919 -
accuracy: 0.8827 - val_loss: 0.3736 - val_accuracy: 0.8380
Epoch 10/30
40/40 [==============================] - 18s 445ms/step - loss: 0.2566 -
accuracy: 0.8982 - val_loss: 0.3362 - val_accuracy: 0.8642
[0.34082308411598206, 0.8606799840927124]
```

[ ]:
```
bilstm_model = AmazonReviewsModel(
    2000,
    256,
    LSTM,
    num_recurrent_layers=4,
    dropout=0.15,
    ffn_dims=[128, 64],
    name="BiLSTM",
    bidirectional=True,
)
_, fig = bilstm_model.train(
    dataset, f"{MODEL_PATH}/{bilstm_model.name}.weights.h5", **train_default
)
print(bilstm_model.evaluate(dataset.X_test, dataset.y_test))
fig
```

```
Epoch 1/30
40/40 [==============================] - 26s 323ms/step - loss: 0.4755 -
accuracy: 0.7538 - val_loss: 0.3033 - val_accuracy: 0.8764
Epoch 2/30
40/40 [==============================] - 7s 185ms/step - loss: 0.2709 -
```

```
accuracy: 0.8923 - val_loss: 0.3008 - val_accuracy: 0.8818
Epoch 3/30
40/40 [==============================] - 7s 180ms/step - loss: 0.2340 -
accuracy: 0.9077 - val_loss: 0.2958 - val_accuracy: 0.8756
Epoch 4/30
40/40 [==============================] - 6s 163ms/step - loss: 0.2143 -
accuracy: 0.9161 - val_loss: 0.3296 - val_accuracy: 0.8698
Epoch 5/30
40/40 [==============================] - 7s 166ms/step - loss: 0.1854 -
accuracy: 0.9286 - val_loss: 0.3100 - val_accuracy: 0.8724
Epoch 6/30
40/40 [==============================] - 7s 168ms/step - loss: 0.1583 -
accuracy: 0.9420 - val_loss: 0.3610 - val_accuracy: 0.8600
Epoch 7/30
40/40 [==============================] - 7s 167ms/step - loss: 0.1324 -
accuracy: 0.9506 - val_loss: 0.3799 - val_accuracy: 0.8686
Epoch 8/30
40/40 [==============================] - 7s 167ms/step - loss: 0.1085 -
accuracy: 0.9615 - val_loss: 0.4274 - val_accuracy: 0.8568
[0.30964571237564087, 0.8692399859428406]
```

[ ]:
```python
bigru_model = AmazonReviewsModel(
    2000,
    256,
    GRU,
    num_recurrent_layers=4,
    dropout=0.15,
    ffn_dims=[128, 64],
    name="BiGRU",
    bidirectional=True,
)
_, fig = bigru_model.train(
    dataset, f"{MODEL_PATH}/{bigru_model.name}.weights.h5", **train_default
)
print(bigru_model.evaluate(dataset.X_test, dataset.y_test))
fig
```

```
Epoch 1/30
40/40 [==============================] - 25s 300ms/step - loss: 0.4524 -
accuracy: 0.7727 - val_loss: 0.3071 - val_accuracy: 0.8714
Epoch 2/30
40/40 [==============================] - 7s 174ms/step - loss: 0.2682 -
accuracy: 0.8911 - val_loss: 0.2989 - val_accuracy: 0.8712
Epoch 3/30
40/40 [==============================] - 7s 169ms/step - loss: 0.2370 -
accuracy: 0.9065 - val_loss: 0.2957 - val_accuracy: 0.8766
Epoch 4/30
40/40 [==============================] - 6s 161ms/step - loss: 0.2045 -
```

```
accuracy: 0.9200 - val_loss: 0.3170 - val_accuracy: 0.8622
Epoch 5/30
40/40 [==============================] - 6s 158ms/step - loss: 0.1808 -
accuracy: 0.9298 - val_loss: 0.3256 - val_accuracy: 0.8756
Epoch 6/30
40/40 [==============================] - 6s 157ms/step - loss: 0.1525 -
accuracy: 0.9424 - val_loss: 0.3678 - val_accuracy: 0.8592
Epoch 7/30
40/40 [==============================] - 6s 153ms/step - loss: 0.1243 -
accuracy: 0.9555 - val_loss: 0.4217 - val_accuracy: 0.8694
Epoch 8/30
40/40 [==============================] - 6s 156ms/step - loss: 0.1094 -
accuracy: 0.9596 - val_loss: 0.4778 - val_accuracy: 0.8550
[0.31589433550834656, 0.871720016002655]
```

Althought the performance of the Bidirectional LSTM (86.92%) and Bidirectional GRU (87.17%) do not seem to strongly improve the unidirectional processing, we see that the simple RNN cell takes a great advantage of introducing right-to-left contextualization. While the unidirectional RNN cell barely reached ~60% of accuracy, the bidirecitonal RNN is able to reach 86.06 points, obtaining a similar performance to the other recurrent cells.

1.5 Transformer

In this section we introduce the Transformer block to reinforce the word contextualization between the input embedding layer and the recurrent layers. We conducted experiments adding three Transformer layers of 4 heads before the bidirectional recurrent block using a hidden size of $d_h = 128$. We see that the Transformer improves ~1% of the accuracy of all models.

[ ]:
```
birnn_transformer = AmazonReviewsModel(
    2000,
    128,
    SimpleRNN,
    num_recurrent_layers=4,
    dropout=0.2,
    ffn_dims=[128, 64],
    name="BiRNN-transformer",
    num_transformers=3,
    bidirectional=True,
)
_, fig = birnn_transformer.train(
    dataset, f"{MODEL_PATH}/{birnn_transformer.name}.weights.h5",␣
 ↪**train_default
)
print(birnn_transformer.evaluate(dataset.X_test, dataset.y_test))
fig
```

```
Epoch 1/30
40/40 [==============================] - 32s 540ms/step - loss: 0.5880 -
accuracy: 0.6709 - val_loss: 0.4147 - val_accuracy: 0.8198
```

```
Epoch 2/30
40/40 [==============================] - 20s 511ms/step - loss: 0.3187 -
accuracy: 0.8699 - val_loss: 0.2993 - val_accuracy: 0.8788
Epoch 3/30
40/40 [==============================] - 20s 513ms/step - loss: 0.2358 -
accuracy: 0.9075 - val_loss: 0.2937 - val_accuracy: 0.8810
Epoch 4/30
40/40 [==============================] - 20s 503ms/step - loss: 0.1816 -
accuracy: 0.9302 - val_loss: 0.3263 - val_accuracy: 0.8704
Epoch 5/30
40/40 [==============================] - 20s 503ms/step - loss: 0.1362 -
accuracy: 0.9486 - val_loss: 0.3674 - val_accuracy: 0.8756
Epoch 6/30
40/40 [==============================] - 20s 502ms/step - loss: 0.0927 -
accuracy: 0.9658 - val_loss: 0.4270 - val_accuracy: 0.8716
Epoch 7/30
40/40 [==============================] - 20s 503ms/step - loss: 0.0676 -
accuracy: 0.9764 - val_loss: 0.4949 - val_accuracy: 0.8570
Epoch 8/30
40/40 [==============================] - 20s 504ms/step - loss: 0.0682 -
accuracy: 0.9753 - val_loss: 0.5041 - val_accuracy: 0.8606
[0.3136318325996399, 0.8738399744033813]
```

[ ]:
```
bilstm_transformer = AmazonReviewsModel(
    2000,
    128,
    LSTM,
    num_recurrent_layers=4,
    dropout=0.2,
    ffn_dims=[128, 64],
    name="BiLSTM-transformer",
    num_transformers=3,
    bidirectional=True,
)
_, fig = bilstm_transformer.train(
    dataset, f"{MODEL_PATH}/{bilstm_transformer.name}.weights.h5",␣
 ↪**train_default
)
print(bilstm_transformer.evaluate(dataset.X_test, dataset.y_test))
fig
```

```
Epoch 1/30
40/40 [==============================] - 31s 354ms/step - loss: 0.5360 -
accuracy: 0.7034 - val_loss: 0.3169 - val_accuracy: 0.8694
Epoch 2/30
40/40 [==============================] - 7s 188ms/step - loss: 0.2819 -
accuracy: 0.8868 - val_loss: 0.2901 - val_accuracy: 0.8822
Epoch 3/30
```

```
40/40 [==============================] - 8s 189ms/step - loss: 0.2247 -
accuracy: 0.9142 - val_loss: 0.2758 - val_accuracy: 0.8882
Epoch 4/30
40/40 [==============================] - 7s 183ms/step - loss: 0.1895 -
accuracy: 0.9299 - val_loss: 0.2856 - val_accuracy: 0.8854
Epoch 5/30
40/40 [==============================] - 7s 184ms/step - loss: 0.1602 -
accuracy: 0.9416 - val_loss: 0.2955 - val_accuracy: 0.8784
Epoch 6/30
40/40 [==============================] - 7s 182ms/step - loss: 0.1269 -
accuracy: 0.9559 - val_loss: 0.3626 - val_accuracy: 0.8802
Epoch 7/30
40/40 [==============================] - 7s 181ms/step - loss: 0.1085 -
accuracy: 0.9608 - val_loss: 0.4091 - val_accuracy: 0.8688
Epoch 8/30
40/40 [==============================] - 7s 181ms/step - loss: 0.0844 -
accuracy: 0.9707 - val_loss: 0.4603 - val_accuracy: 0.8762
[0.2989642918109894, 0.8779600262641907]
```

[ ]:
```python
bigru_transformer = AmazonReviewsModel(
    2000,
    128,
    GRU,
    num_recurrent_layers=4,
    dropout=0.2,
    ffn_dims=[128, 64],
    name="BiGRU-transformer",
    num_transformers=3,
    bidirectional=True,
)
_, fig = bigru_transformer.train(
    dataset, f"{MODEL_PATH}/{bigru_transformer.name}.weights.h5",␣
 ↪**train_default
)
print(bigru_transformer.evaluate(dataset.X_test, dataset.y_test))
fig
```

```
Epoch 1/30
40/40 [==============================] - 30s 347ms/step - loss: 0.5078 -
accuracy: 0.7297 - val_loss: 0.3155 - val_accuracy: 0.8704
Epoch 2/30
40/40 [==============================] - 7s 186ms/step - loss: 0.2673 -
accuracy: 0.8927 - val_loss: 0.2799 - val_accuracy: 0.8860
Epoch 3/30
40/40 [==============================] - 7s 179ms/step - loss: 0.2053 -
accuracy: 0.9190 - val_loss: 0.3133 - val_accuracy: 0.8748
Epoch 4/30
40/40 [==============================] - 7s 178ms/step - loss: 0.1700 -
```

```
accuracy: 0.9347 - val_loss: 0.3381 - val_accuracy: 0.8726
Epoch 5/30
40/40 [==============================] - 7s 178ms/step - loss: 0.1338 -
accuracy: 0.9495 - val_loss: 0.3379 - val_accuracy: 0.8808
Epoch 6/30
40/40 [==============================] - 7s 175ms/step - loss: 0.1054 -
accuracy: 0.9628 - val_loss: 0.3902 - val_accuracy: 0.8776
Epoch 7/30
40/40 [==============================] - 7s 176ms/step - loss: 0.0774 -
accuracy: 0.9730 - val_loss: 0.4321 - val_accuracy: 0.8666
[0.299507200717926, 0.8766000270843506]
```

1.6 Optimal configuration of the recurrent architecture

In this section we experimented with the regularization options of the full network. We tested the performance of different weight regularizers, initializers and optimizers. The cell below shows the deployment of the hyperaparameter search and the final result:

[ ]:
```python
grid = OrderedDict(
    regularizer=[L1(1e-4), L2(1e-3), L1L2(1e-4)],
    initializer=["random_normal", "glorot_uniform", "he_normal", "orthogonal"],
    optimizer=[Adam, RMSprop],
)
Regularizer.__repr__ = lambda x: x.__class__.__name__
dataset = AmazonDataset.load(
    train_path=path_dir + "train_small.txt",
    test_path=path_dir + "test_small.txt",
    max_features=2000,
)


def tostring(x):
    if isinstance(x, type):
        return x.__name__
    else:
        return repr(x)


def applydeep(lists, func):
    result = []
    for item in lists:
        result.append(list(map(func, item)))
    return result


df = pd.DataFrame(
    columns=["train", "val", "test"],
    index=pd.MultiIndex.from_product(applydeep(grid.values(), tostring)),
```

```
)
df.index.names = ["regularizer", "initializer", "optimizer"]

for i, params in enumerate(product(*grid.values())):
    params = dict(zip(grid.keys(), params))
    optimizer = params.pop("optimizer")
    model = AmazonReviewsModel(
        2000,
        128,
        LSTM,
        num_recurrent_layers=4,
        dropout=0.2,
        ffn_dims=[128, 64],
        num_transformers=3,
        bidirectional=True,
        **params,
    )
    model.train(dataset, "results/amazon.weights.h5", opt=optimizer,␣
 ↪**train_default)
    _, train_acc = model.evaluate(dataset.X_train, dataset.y_train)
    _, val_acc = model.evaluate(dataset.X_val, dataset.y_val)
    _, test_acc = model.evaluate(dataset.X_test, dataset.y_test)
    df.loc[tuple(map(tostring, params.values()))] = [train_acc, val_acc,␣
 ↪test_acc]
    df.to_csv("grid.csv")
df = df.applymap(lambda x: round(x * 100, 2))
df
```

[ ]:

```
                                            train    val   test
regularizer initializer       optimizer
L1          'random_normal'   Adam         95.29  88.26  87.42
                              RMSprop      95.29  88.26  87.42
            'glorot_uniform'  Adam         94.03  88.14  87.57
                              RMSprop      94.03  88.14  87.57
            'he_normal'       Adam         96.73  85.02  84.52
                              RMSprop      96.73  85.02  84.52
            'orthogonal'      Adam         95.05  87.34  86.74
                              RMSprop      95.05  87.34  86.74
L2          'random_normal'   Adam         94.52  87.98  87.46
                              RMSprop      94.52  87.98  87.46
            'glorot_uniform'  Adam         93.98  87.98  87.18
                              RMSprop      93.98  87.98  87.18
            'he_normal'       Adam         97.61  88.02  87.27
                              RMSprop      97.61  88.02  87.27
            'orthogonal'      Adam         94.09  88.00  87.56
                              RMSprop      94.09  88.00  87.56
L1L2        'random_normal'   Adam         94.21  88.20  87.32
```

```
                    RMSprop    94.21  88.20  87.32
'glorot_uniform'    Adam       93.31  88.26  87.56
                    RMSprop    93.31  88.26  87.56
'he_normal'         Adam       99.39  86.48  85.75
                    RMSprop    99.39  86.48  85.75
'orthogonal'        Adam       93.52  87.28  86.89
                    RMSprop    93.52  87.28  86.89
```

[ ]:
```python
print(df.max())
print(df.idxmax())
```

```
train    99.39
val      88.26
test     87.57
dtype: float64
train       (L1L2, 'he_normal', Adam)
val        (L1, 'random_normal', Adam)
test      (L1, 'glorot_uniform', Adam)
dtype: object
```

The final outcome does not show a special improvement varying other the configuration of the network from the performance obtained with bidirectional cells and Transformers (near 87-88% of accuracy in the validation and test set). For this reason, we did not include weight regularization or special initializations for the final proposed architecture.

1.7 Final comparison

The next cell shows the final comparison of all the models evaluated in this work.

[ ]:
```python
base = [rnn_model, lstm_model, gru_model]
models = [
    lstm_enhanced,
    gru_enhanced,
    birnn_model,
    bilstm_model,
    bigru_model,
    birnn_transformer,
    bilstm_transformer,
    bigru_transformer,
]
dataset1 = AmazonDataset.load(
    train_path=path_dir + "train_small.txt",
    test_path=path_dir + "test_small.txt",
    max_features=1000,
)
dataset2 = AmazonDataset.load(
    train_path=path_dir + "train_small.txt",
    test_path=path_dir + "test_small.txt",
```

```
    max_features=2000,
)

names = [model.name for model in base + models]
train_accs = [
    model.evaluate(dataset1.X_train, dataset1.y_train)[1] for model in base
] + [model.evaluate(dataset2.X_train, dataset2.y_train)[1] for model in models]
test_accs = [model.evaluate(dataset1.X_test, dataset1.y_test)[1] for model in␣
 ↪base] + [
    model.evaluate(dataset2.X_test, dataset2.y_test)[1] for model in models
]
```

[ ]:
```
fig = go.Figure()
fig.add_trace(
    go.Scatter(
        x=train_accs,
        y=test_accs,
        text=names,
        mode="text+markers",
        textposition="top right",
    )
)
fig.update_layout(
    height=600,
    width=1000,
    margin=dict(t=50, b=10, r=10, l=10),
    title_text="Comparison of the Amazon Reviews models",
    xaxis_title="train",
    yaxis_title="test",
    template="seaborn",
)
fig.update_yaxes(range=[0.835, 0.88])
fig.update_xaxes(range=[0.875, 0.94])
fig.show()
```

In our analysis, base models exhibited the poorest performance on both the training and test sets. Enhanced architectures notably improved accuracy on the training set; however, their performance did not significantly surpass baseline models in the final evaluation, with the exception of the simple RNN-based architecture achieving only 56% accuracy.

Introducing bidirectionality notably enhanced the performance on the test set, while the integration of Transformer layers across architectures slightly improved evaluation metrics. The most effective architecture, BiLSTM-transformer, achieved an impressive 87.79% accuracy score. This architecture likely excelled due to its enhanced temporal representations in the BiLSTM cell (utilizing three gates compared to the two in GRU) and strengthened embedding contextualization through Transformer layers prior to the recurrent layers. Although Transformer-based architectures demonstrated some improvement, the associated computational costs may not justify the gains achieved. GRU-based models demonstrated comparable performance to LSTM-based models, with the exception of the Transformer-based architecture, where both achieved similar accuracy levels. Additionally, fine-tuned parameters in enhanced architectures consistently yielded superior results. Notably, selecting a vocabulary size that includes the maximum explored (2000 words) appears optimal, suggesting that richer information facilitates improved model performance.

In addition to investigating various architectural enhancements, several regularization techniques were employed to mitigate overfitting. However, these techniques did not result in significant improvements in performance, and thus, their impact was not reflected in the graphs or final evaluations. Despite their potential benefits in controlling model complexity and improving generalization, the specific configurations tested did not demonstrate superior results compared to the best-performing architectures highlighted in our analysis.

Given the limitations of our dataset size and the performance of our custom architectures, a exploration with pretrained models is a promising avenue for achieving improved results. Leveraging pretrained embeddings and transformer models, which encapsulate extensive prior knowledge from large-scale datasets, could offer significant advantages. By incorporating pretrained representations, our models may benefit from enhanced feature extraction and contextual understanding, potentially outperforming our base architectures. This avenue of investigation could potentially lead to significant performance gains, particularly when working with smaller datasets, by harnessing the wealth of information encoded within pretrained models.