RSM8421: AI and Deep Learning Tools - Tutorial 2

- explain how preprocessing choices (cleaning, vocabulary size, sequence length) shape downstream models,
- implement and train a convolutional neural network (CNN),
- implement and train a bidirectional recurrent neural network (RNN), and
- compare model behavior with shared metrics, plots, and qualitative predictions.

Learning roadmap

1. **Part 1 – Data preparation, word embeddings, and vectorization:** download the raw IMDB dataset, clean the text, adapt a `TextVectorization` layer, and visualize the resulting vocabulary.
2. **Part 2 – CNN sentiment classifier:** build a fast 1-D convolutional model that looks for n-gram level evidence of positive or negative tone.
3. **Part 3 – RNN sentiment classifier:** replace convolutions with a bidirectional LSTM that reads the review sequentially and captures order-sensitive cues.
4. **Part 4 – Comparisons:** evaluate both models on the same splits, plot their training curves, and inspect side-by-side predictions.

Part 1 – Data preparation, word embeddings, and vectorization

---

1.1 Configure the environment and reproducibility switches

Check https://www.tensorflow.org/install for guidelines on installing TensorFlow. We begin by importing TensorFlow along with a couple of helper libraries for numerical computing and visualization. TensorFlow ships with the high-level Keras API that we will rely on to define layers quickly.

Do some data reprocessing→ transformable -> classification task:

Download , clean, embedding, noisy,

CNN Classifier

RNN Classifer

1.2 Download IMDB reviews dataset

We revisit the classic Stanford IMDB dataset that ships as raw text files. There are ***25,000 labeled reviews for training and 25,000 for testing***, each tagged as positive (1) or negative (0).

We first download and extract the archive (skip the download if you already have it locally). We then remove the unlabeled reviews so that we focus entirely on supervised learning.

1.3 Turn directories into batched tf.data datasets

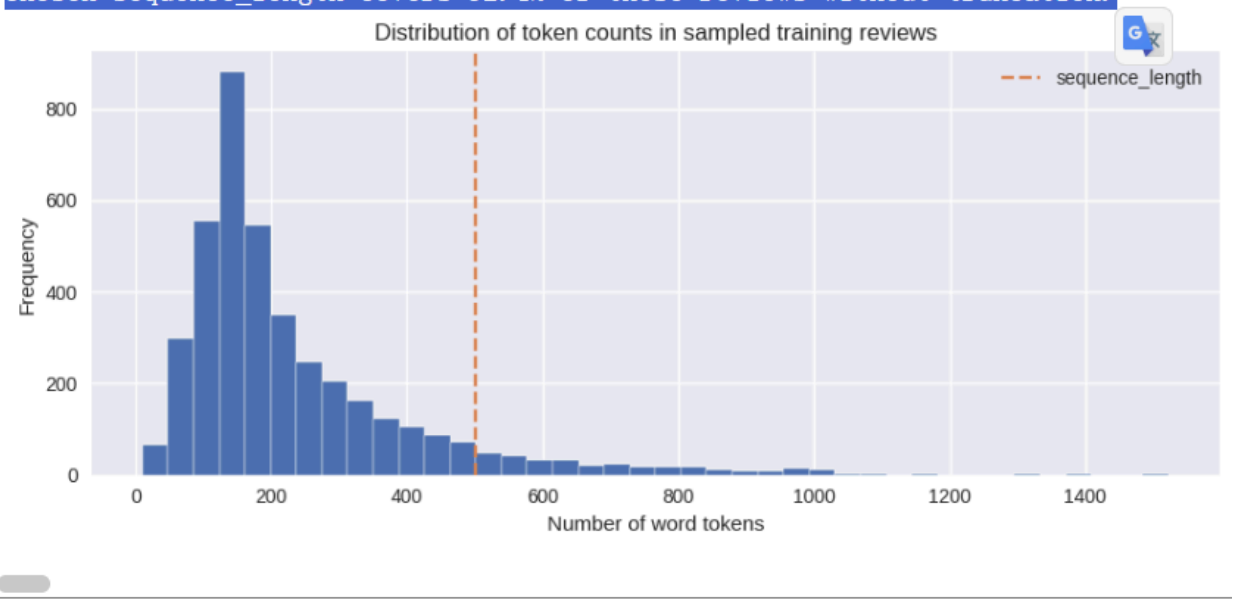
We now build training, validation, and test datasets with tf.data and examine their class balance. Keras ships with text_dataset_from_directory, which reads the folder structure (`neg/` and `pos/`) and emits tf.data. dataset objects of (text, label) pairs. We keep 20% of the training split for validation so that the CNN and RNN later on see identical splits.

1.4 Inspect label balance and review lengths

We now visualize review lengths to select the sequence_length .

Before vectorizing anything, it helps to know whether the dataset is balanced and how long the reviews are. The histogram below uses 4,000 training reviews to visualize word counts and overlays the chosen sequence_length to show how much truncation/padding we will perform.

```
I have one word to someup this movie, WOW! I saw ″Darius Goes West″ at the Tribe
tf.Tensor(b'i guess its possible that I\'ve seen worse movies, but this one is a
i guess its possible that I've seen worse movies, but this one is a real stinker
tf.Tensor(b″Imagine you're a high-school boy, in the back of a dark, uncrowded t
Imagine you're a high-school boy, in the back of a dark, uncrowded theater with
tf.Tensor(b″I can remember this movie from when i was a small child, i loved it
I can remember this movie from when i was a small child, i loved it then and I s
Label distribution in first 4000 training reviews:
  neg: 2032
  pos: 1968
Mean tokens: 232.4 | Median: 173.0 | 90th percentile: 450.2
Chosen sequence_length covers 92.4% of these reviews without truncation.
```



Can change the length randomly, positive or negatively.

1.5 Clean raw text with a custom standardization function

We now clean the text, adapt a TextVectorization layer, and look at the learned vocabulary. The IMDB files contain HTML break tags (
) and punctuation that we do not want the model to treat as standalone tokens. We therefore lowercase everything, strip HTML, and remove punctuation before tokenization.

1.6 See what vectorization produces for a sample review

To keep the process tangible, take a short sentence, vectorize it, and decode the first few non-zero IDs back to their tokens.

1.7 Prepare fast input pipelines for later models

We wrap the vectorization logic inside `tf.data` pipelines so future models receive ready-to-use tensors. Caching and prefetching overlap CPU preprocessing with GPU training.

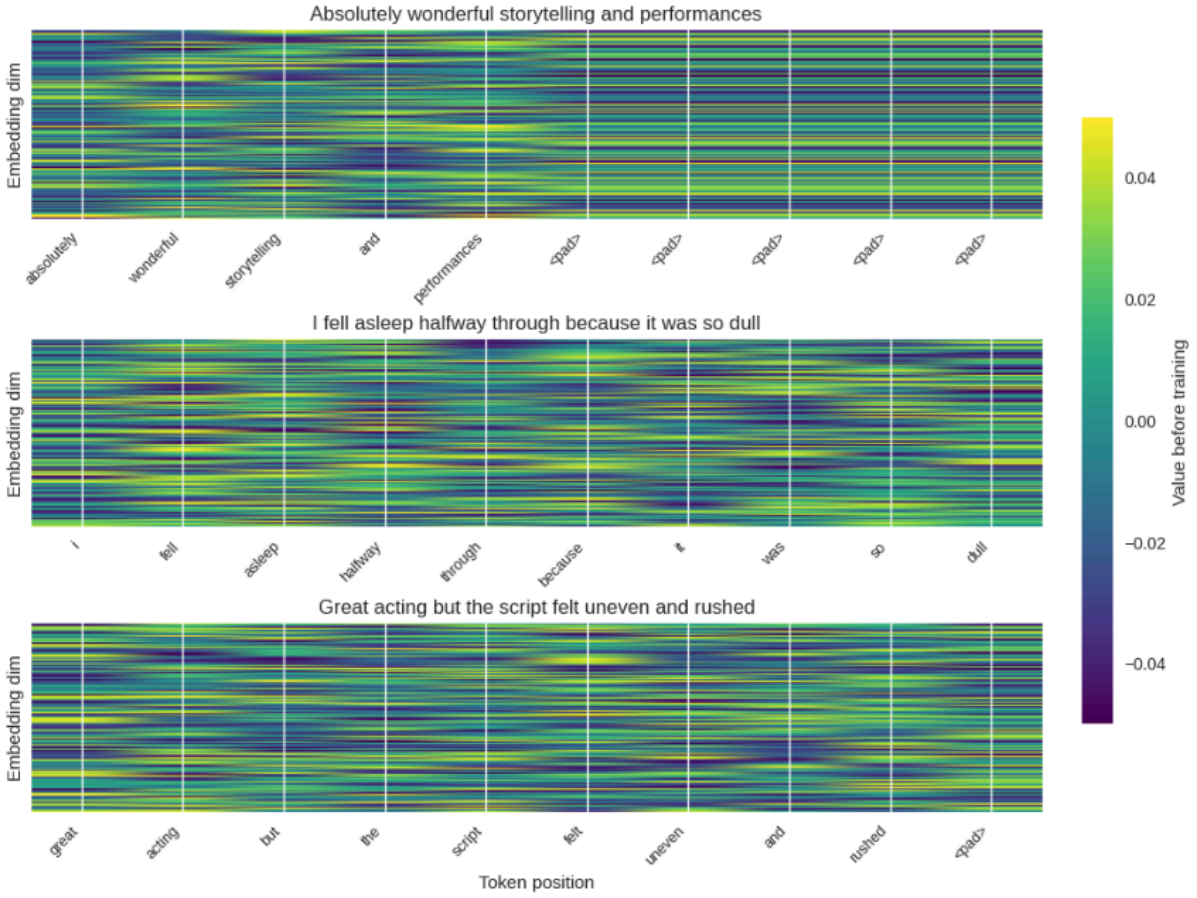
1.8 Embedding intuition: from sparse tokens to dense geometry

An embedding layer learns a dense vector for every token ID so that similar reviews receive similar representations. The heatmaps below show 12 token positions from three sample sentences mapped into a 128-dimensional space (colors represent the initial random values that future training will adjust).

```
cbar = fig.colorbar(im, ax=axes, fraction=0.03, pad=0.04)
cbar.set_label("Value before training")
plt.xlabel("Token position")

# Zero padding: make the legthen of sentence into same size 500 euqualized and exactly
# Add fitted number of zeros into the input two sides
```

```
Text(0.5, 0, 'Token position')
```



1.10 Function to visualize model training later on

We finish Part 1 by defining a small plotting helper that both the CNN and RNN will reuse. Keeping it here highlights that tooling for evaluation is part of the data-preparation story.

RNN is better , as RNN includes meaning, Cons: RNN computational onerous

CNN- convolution capture complex features, indeep and advanced exploration and explanation on the categorization

Concaptually RNN capture the full content, CNN is simple classification,here, our classification task is relative five , batchized five words capture all the unground is enough for the classification with five to three categories,

Situation: more comprehensive understanding on the syntax, vectors, needs more advance analysis

L2 is smooth , smooth out derivatives;

What is convolution fitler ? why 256? What is proper value?

Part 2 – Convolutional neural network for sentiment

With vectorized reviews in hand we can feed them into a convolutional neural network. A 1-D CNN slides filters across the embedded tokens to detect short phrases such as “not good” or “absolutely loved.” We follow the following:

1. Embed tokens into dense vectors that are learned jointly with the classifier.
2. Apply dropout and stacked Conv1D layers to capture *n-gram evidence.*
3. Use global max pooling to convert variable-length sequences into fixed-size representations.
4. Add dense layers and a sigmoid output for binary sentiment.

We train for a few epochs, inspect the learning curves, and wrap the model with the TextVectorization layer so it can accept raw strings just like any production API.

2.1 Connect vectorized reviews to convolution-friendly tensors

`train_ds` now yields padded sequences of token IDs with shape (`batch`, `sequence_length`). Before building a full CNN it helps to confirm those shapes and see how an embedding layer turns each integer into a dense vector.

Layer (type)	Output Shape	#
token_ids (InputLayer)	(None, None)	0
embedding (Embedding)	(None, None, 128)	2,560,000
dropout_emb (SpatialDropout1D)	(None, None, 128)	0
conv1 (Conv1D)	(None, None, 256)	164,096
bn1 (BatchNormalization)	(None, None, 256)	1,024
conv2 (Conv1D)	(None, None, 256)	327,936
bn2 (BatchNormalization)	(None, None, 256)	1,024
global_max_pool (GlobalMaxPooling1D)	(None, 256)	0
dense_hidden (Dense)	(None, 256)	65,792
dropout_hidden (Dropout)	(None, 256)	0
output (Dense)	(None, 1)	257

Total params: 3,120,129 (11.90 MB)
Trainable params: 3,119,105 (11.90 MB)
Non-trainable params: 1,024 (4.00 KB)

Global maximum pool : change the input features into **one dimension**, flatten, check all of the matrix value to the max , desen_hidden: last two layers, dense function capture all the features, cov neuron, make the feature complicated and multiple, pervasive, the dense connected with the previous retain all clear information features from con neus to the output,

Layer (type)	Output Shape	#
token_ids (InputLayer)	(None, None)	0
embedding (Embedding)	(None, None, 128)	2,560,000
dropout_emb (SpatialDropout1D)	(None, None, 128)	0
conv1 (Conv1D)	(None, None, 256)	164,096
bn1 (BatchNormalization)	(None, None, 256)	1,024
conv2 (Conv1D)	(None, None, 256)	327,936
bn2 (BatchNormalization)	(None, None, 256)	1,024
global_max_pool (GlobalMaxPooling1D)	(None, 256)	0
dense_hidden (Dense)	(None, 256)	65,792
dropout_hidden (Dropout)	(None, 256)	0
output (Dense)	(None, 1)	257

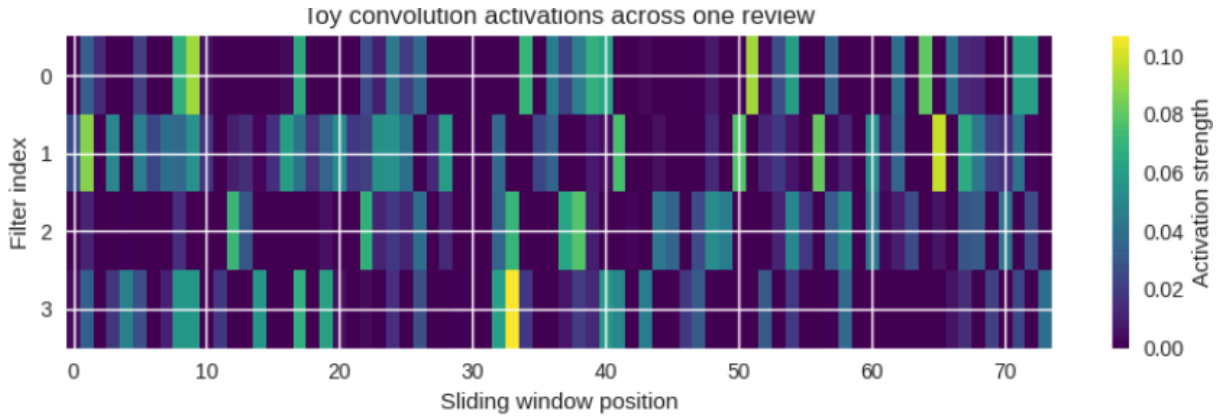
Total params: 3,120,129 (11.90 MB)
Trainable params: 3,119,105 (11.90 MB)
Non-trainable params: 1,024 (4.00 KB)

2.2 Build the CNN block by block

A 1-D CNN treats the sequence dimension like a spatial axis. We embed tokens, drop out some activations for regularization, stack two convolutional layers to detect meaningful n-grams, pool across time, and finish with dense layers. The `build_cnn_model` helper below keeps these decisions encapsulated so we can reuse the architecture later.

2.3 Visualize how convolution windows slide across embeddings

Each `Conv1D` kernel looks at `kernel_size` consecutive tokens at a time. The heatmap below feeds one sample review through a toy convolutional layer (random weights) to illustrate how activations are produced across the sequence



The activation strength, the maxim layer, the higher heat, the meaning impact and corelation of the input mix text features,

2.4 Compile the CNN with binary cross-entropy and Adam

Sentiment analysis is a binary classification problem. We pair the sigmoid output with binary_crossentropy and monitor accuracy.

2.5 Train the CNN

We reuse the cached datasets from Part 1, train a bit longer with a reduce-on-plateau schedule, and call plot_history so the curves are adjacent to the training output.

```
[]
```

2.6 Peek at trained CNN embeddings

Replot the same sample sentences after CNN training to see how the embedding layer reshapes token vectors once it has learned from the data.

2.7 Evaluate on the held-out test set

After training, we compute the final test loss/accuracy.

3.2 Build the bidirectional LSTM sentiment model

We embed the input sequence, apply dropout, pass it through a bidirectional LSTM (64 units in each direction), and finish with dense layers. The forward/backward combination lets the network reason about both preceding and following context for every token.

The LSTM stands for the importance of the evaluation on the sequence, and the character learning sequences, on explaining and categorizing the final result

No kernel size , no windowing , the whole sequence matters on the total length horizon.

3.2 Build the bidirectional LSTM sentiment model

We embed the input sequence, apply dropout, pass it through a bidirectional LSTM (64 units in each direction), and finish with dense layers. The forward/backward combination lets the network reason about both preceding and following context for every token.

```
def build_rnn_model(
    vocab_size=max_features,
    embed_dim=embedding_dim,
    lstm_units=128,
    dense_units=256,
    dropout_rate=0.35,
    recurrent_dropout=0.2,
    l2_reg=1e-4,
):
    inputs = tf.keras.Input(shape=(None,), dtype="int64", name="token_ids")
    x = tf.keras.layers.Embedding(vocab_size, embed_dim, name="embedding")(inputs)
    x = tf.keras.layers.Dropout(dropout_rate, name="dropout_emb")(x)
    reg = tf.keras.regularizers.l2(l2_reg)
    x = tf.keras.layers.Bidirectional(
        tf.keras.layers.LSTM(
            lstm_units,
            return_sequences=False,
            dropout=recurrent_dropout,
            kernel_regularizer=reg,
            recurrent_regularizer=reg,
        ),
        name="bilstm",
    )(x)
```

```
    inputs = tf.keras.Input(shape=(None,), dtype="int64", name="token
    x = tf.keras.layers.Embedding(vocab_size, embed_dim, name="embedding")(inputs)
    x = tf.keras.layers.Dropout(dropout_rate, name="dropout_emb")(x)
    reg = tf.keras.regularizers.l2(l2_reg)
    x = tf.keras.layers.Bidirectional(
        tf.keras.layers.LSTM(
            lstm_units,
            return_sequences=False,
            dropout=recurrent_dropout,
            kernel_regularizer=reg,
            recurrent_regularizer=reg,
        ),
        name="bilstm",
    )(x)
    x = tf.keras.layers.Dense(
        dense_units,
        activation="relu",
        kernel_regularizer=reg,
        name="dense_hidden",
    )(x)
    x = tf.keras.layers.Dropout(dropout_rate, name="dropout_hidden")(x)
    outputs = tf.keras.layers.Dense(
        1,
        activation="sigmoid",
        kernel_regularizer=reg,
        name="output",
    )(x)
    return tf.keras.Model(inputs, outputs, name="rnn_model")

rnn_model = build_rnn_model()
rnn_model.summary()
```

It is not fair to let the RNN compare with CNN, the RNN analysis is more on sequential correlations, CNN is more straight in the simple individual classification task;