**Convolutional neural networks for text classification**

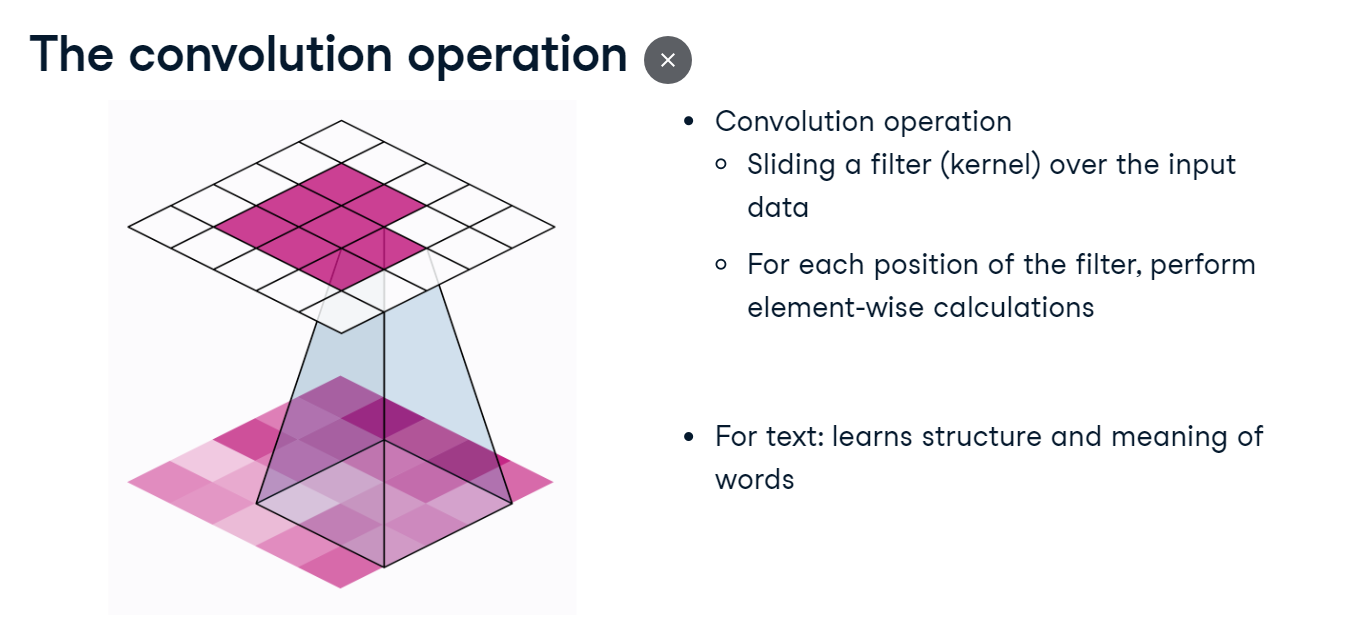
**CNNs for text classification**

We have seen CNNs used for classifying images, but they can also apply to text, for example, for classifying tweets as positive, negative, or neutral.

**The convolution operation**

The key operation in CNNs is convolution, where a filter or kernel slides over the input, performing element-wise calculations. This helps the model learn detailed word and sentence structure and meaning in text data.

1. 1 Animation from Vincent Dumoulin, Francesco Visin



**Filter and stride in CNNs**

In the convolution operation, we use a filter, a small matrix that slides over the input data, a matrix of tensors. We use a parameter called stride, which determines how many positions the filter moves each time it slides. In the image, we move a two by two filter with a stride of two.

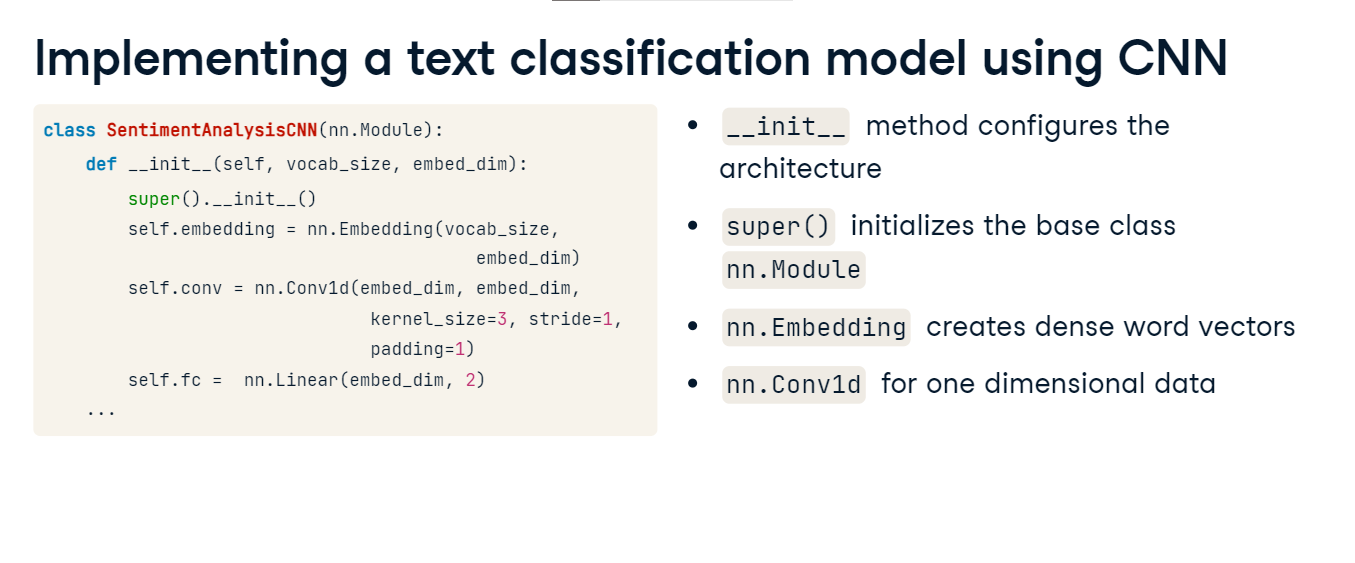
1. 1 Animation from Vincent Dumoulin, Francesco Visin

**CNN architecture for text**

A typical CNN architecture for text classification consists of three layers. The convolutional layer applies filters to the input data to detect patterns. The pooling layer reduces the size of the data while preserving important information. Finally, the fully connected layer uses the previous layer outputs for final predictions.

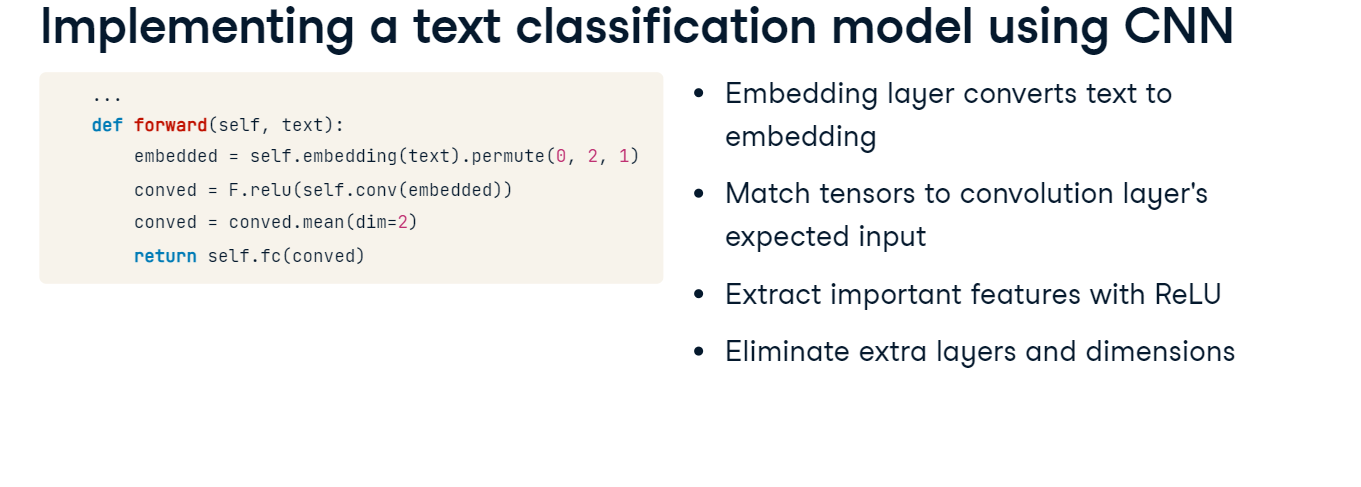
**Implementing a text classification model using CNN**

Let's build a sentiment analysis model, starting with the SentimentAnalysisCNN class. Much of the code will look familiar, and we'll prepare the dataset in later steps. The init method accepts vocabulary size and embedding dimension to configure the network architecture. The super method initializes the base class of nn-dot-Module to leverage the PyTorch framework properly. We initialize an embedding layer using nn-dot-Embedding, which creates dense vectors with the specified vocabulary size and embed\_dim. In our case, self-dot-conv directly initializes a single convolutional layer, while in other models, convolutional layers are grouped and applied sequentially. Convolutions apply nn-dot-Conv1d, with uniform input-output channels, kernel size, stride, and padding, which ensures uniform text sequence lengths. Conv1d is preferred over Conv2d as our text data is one-dimensional. Lastly, the nn-dot-Linear layer transforms the combined outputs of all convolutional layers into the desired target output size. We omit a pooling layer in this model because our data in the exercises will be small.



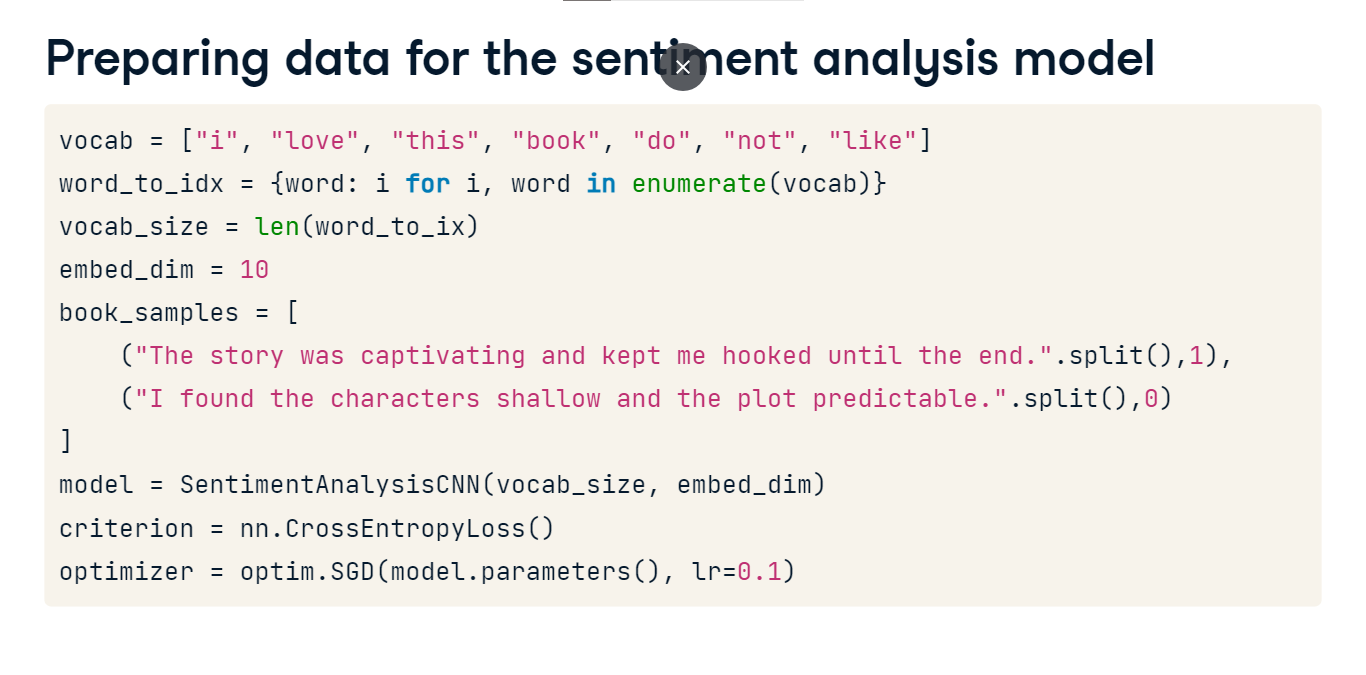
**Implementing a text classification model using CNN**

In the forward method, we pass the input text through an embedding layer, which converts each word to its embedding. The tensor's dimensions are permuted to match the convolutional layer's expected input format defined with batch size, embedding size, and sequence length, in our case zero, two, and one, respectively. We use the convolutional layer with a ReLU activation function to extract important features from the embeddings. Applying the activation function in forward allows dynamic computation that saves memory compared to defining in init. conved-dot-mean calculates the average across the sequence length to reduce the feature dimension and capture the essential information, simplifying the information in each sentence to a single average value for easier analysis by the model.



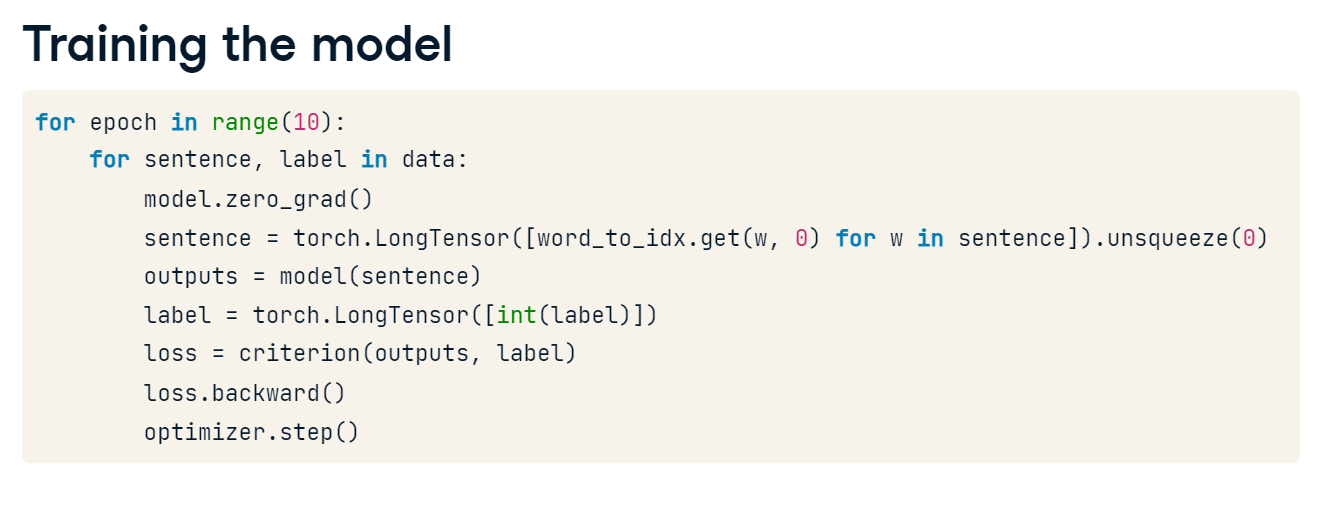
**Preparing data for the sentiment analysis model**

To prepare our data, we create a vocabulary and use word to index mapping. While One-Hot or TF-IDF encoding methods are alternatives, they are less efficient as they do not capture contextual word relationships and result in high dimensional input vectors mostly filled with zeros. Hence, we opt for embeddings. We set vocab-size to the length of word to index and embed-dim to ten. We have two book review samples for demonstration. We then initialize our SentimentAnalysisCNN model with vocab\_size for vocabulary size and embed\_dim for word embedding dimension. For training, we use Cross-Entropy loss with Stochastic Gradient Descent as our optimizer, setting a learning rate of zero-point-one.



**Training the model**

During ten training epochs, we iterate over each sentence-label pair in the data, clearing previous gradients at the model level for clean computation. Words in sentences are mapped to indexes using word\_to\_idx and converted to a long tensor. We use unsqueeze zero to add an extra dimension to the start of the tensor, creating a batch containing a single sequence to fit the model's input expectations. The model then predicts sentiments, and we turn the label into a long tensor. We compute the loss between predictions and actual labels, calculate gradients via backpropagation, and adjust the model parameters using the optimizer.



**Running the Sentiment Analysis Model**

With our model and data ready, we can start making predictions. We iterate over book\_samples, transforming the words to tensors, and feed it to the model. The output provides sentiment scores for our classification labels. Using torch-dot-max, we identify the component with the highest score. One corresponds to positive sentiment, while zero corresponds to negative sentiment. We then print the review alongside its predicted sentiment.

