Traditionally, CNNs are used to analyse images and are made up of one or more convolutional layers, followed by one or more linear layers. The convolutional layers use filters (also called kernels) which scan across an image and produce a processed version of the image. This processed version of the image can be fed into another convolutional layer or a linear layer. Each filter has a shape, e.g. a 3x3 filter covers a 3 pixel wide and 3 pixel high area of the image, and each element of the filter has a weight associated with it, the 3x3 filter would have 9 weights. In traditional image processing these weights were specified by hand by engineers, however the main advantage of the convolutional layers in neural networks is that these weights are learned via backpropagation. The intuitive idea behind learning the weights is that your convolutional layers act like feature extractors, extracting parts of the image that are most important for your CNN's goal, e.g. if using a CNN to detect faces in an image, the CNN may be looking for features such as the existence of a nose, mouth or a pair of eyes in the image [3, 4].

1 CNNs Warmup exercises

In this section, you can find some warmup exercises before actually building your own CNN network. The code for this exercise is at the beginning of the file *CNN.ipynb*. For more details take a look at the Conv2d [1], MaxPool2d [2] modules from Pytorch. Note that we have added TODO comments for all the lines that input is required from your side.

- Assuming that the input data are illustrated in Fig. 1. What the output would be in the case that you apply max pooling with kernel size (i.e., dimension of the filter) equals 3 and stride equals 2? You can either calculate it manually by hand or via Pytorch functions. Note that in this case we are talking about a square filter of size 3 x 3 and for a convolutional or a pooling operation, the stride denotes the number of pixels by which the window moves after each operation. In this case, the stride is also square of dimensions 2 x 2. Zero-padding denotes the process of adding zeroes to each side of the boundaries of the input. In our case, we assume no padding thus, we drop the last convolution if the dimensions do not match.

```
tensor([[[[ 2.6987, 82.5899, 76.2929, 26.5976, 13.1088],
          [23.0795, 97.8725, 76.6898, 88.7104, 51.1192],
          [95.8796, 85.6584, 43.7259,  2.2553, 97.0426],
          [99.6418, 38.8225, 80.7951, 33.3520, 83.8901],
          [20.7309, 81.9728, 61.7567, 87.1467, 10.1192]]]])
```

Figure 1: Input tensor.

- The shape of the input to the maxpool2d Pytorch module is:
  $-\text{Input} = (N, C, H_{in}, W_{in})$
  $-\text{Output} = (N, C, H_{out}, W_{out})$, where
  N is the batch size, C denotes a number of channels, H is a height of input in pixels, and W is width in pixels.

$$H_{out} = \left\lfloor \frac{H_{in} + 2 * \text{padding}[0] - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} + 1 \right\rfloor \tag{1}$$

$$W_{out} = \left\lfloor \frac{W_{in} + 2 * \text{padding}[1] - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \right\rfloor \tag{2}$$

  Here we assume no padding, dilation equals 1 that are default parameters. The kernel_size has indices 0 and 1. This has to do with the size of the filter. For simplicity, we assume square filters and strides and max kernel and stride sizes smaller than five. Given a random input as in the example, find the kernel size and the stride in order to have an output of a specific size (see the .ipynb file).

- Follow the same process for calculating the correct parameters for the conv2d block. For the maxpool block, the formulas were available at this document and the Pytorch documentation. Follow the corresponding formulas for the conv2d pytorch module from [1]. Again given a random input as in the example, find the kernel size and the stride in order to have an output of a specific size (see the .ipynb file).

2 Convolutional Neural Networks with Pytorch

In this exercise, we will implement an exercise similar to the one from the previous labs. In particular, we will again classify digits. The code for this

exercise is in the file *CNN.ipynb*. Note that we have added TODO comments for all the lines that input is required from your side. Most of the parts of this exercise are already filled in for you since we rely on the same dataset as in the previous lab.

Step 1: Reshape the data Reshape the data in such a way that is appropriate for your Pytorch CNN module.

Step 2: Build a CNN model using the Pytorch interface Fill in the Net class in order to build a model that consists of 2 convolutional layers, one max pooling layer, and three linear layers. Note that the last linear layer should transform your output to the number of classes. Train the model and evaluate its performance. What do you observe compared to the previous model built with only fully connected layers?

Step 3: Run the model with different learning rates. Plot the validation accuracy using different learning rates of [1e-1, 1e-2, 1e-3, 1e-4, 1e-5] by keeping the rest of the hyperparameters fixed. Note that the e-suffix represents times ten raised to the power. What do you observe? Re-use the code from the previous lab.

Step 4: Run the model with different optimizers Plot the validation accuracy using different optimizers (Adam, SGD, Adagrad, Adadelta, RMSprop) by keeping the rest of the hyperparameters fixed. What do you observe? Re-use the code from the previous lab.

3 Convolutional Neural Network with Pytorch using textual data

For this exercise, you should have the gensim library installed (version 3.6.0 would work) and your code will run faster on a GPU. If you use colab, gensim is already installed for you and in the case that you want to use a GPU, go to the Runtime menu $\rightarrow$ Change runtime type $\rightarrow$ Hardware accelerator $\rightarrow$ GPU.

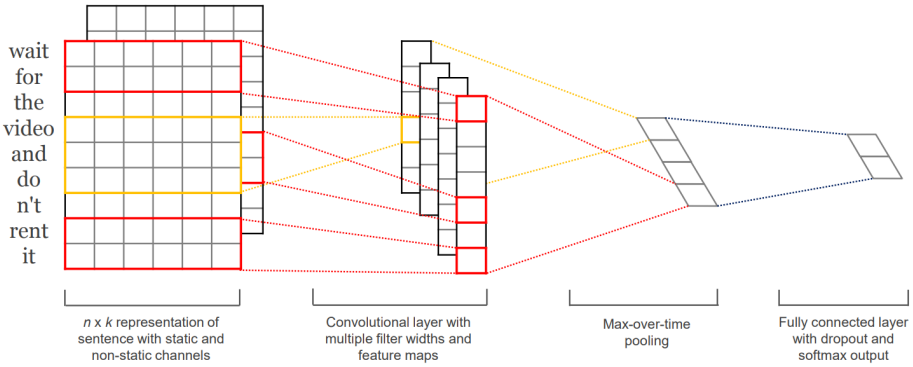

Figure 2: CNN model for text classification [5].

In this exercise, we are going to work on classifying text using CNNs. In particular, we are going to classify reviews as either positive or negative. The code for this exercise is in the file *text CNN.ipynb*. Note that we have added TODO comments for all the lines that input is required from your side.

So why use CNNs on text? In the same way that a 3x3 filter can look over a patch of an image, a 1x2 filter can look over a 2 sequential words in a piece of text, i.e., a bi-gram. In this exercise, in the CNN model we will instead use multiple filters of different sizes which will look at the bi-grams (a 1x2 filter), tri-grams (a 1x3 filter) and/or n-grams (a 1xn filter) within the text. The intuition here is that the appearance of certain bi-grams, tri-grams and n-grams within the review will be a good indication of the final sentiment [3]. The architecture that we will follow for classifying the reviews is depicted in Fig. 2. More details you can find in the work of [5]. In the following subsections, you can see the main building blocks for text classification using CNNs. The main difference with the previous exercises, where we focused on image classification, is that we should preprocess the input text and finally transform the input words (i.e., tokens) to vectors in order to be able to proceed further with numerical computations (e.g., multiplications).

3.1 Word embeddings

The first layer of such a neural network is the word embeddings layer. Given a sentence $w = w_1, ..., w_k$ as a sequence of tokens, the word embedding layer is responsible to map each token to a word vector. We use pre-trained word embeddings here. You can think of an embedding layer as a lookup table, where the rows are indexed by a word token and the columns hold the embedding values.

3.2 CNNs

We can then use a filter that is $[n \times emb\_dim]$. This will cover n sequential words entirely, as their width will be emb_dim dimensions. The kernel sizes would be (2, 50), (3, 50), and (4, 50); to look at 3-, 4-, and 5- sequences of word embeddings at a time (50 is the size of the word embeddings). The kernels only move in one dimension: down to a sequence of word embeddings. In other words, these kernels move along a sequence of words!

We implement the convolutional layers with nn.Conv2d. The in_channels argument is the number of "channels" in your image going into the convolutional layer. In actual images this is usually 3 (one channel for each of the red, blue and green channels), however when using text we only have a single channel, the text itself. The out_channels is the number of filters and the kernel_size is the size of the filters. Again each of our kernel_sizes is going to be [n x emb_dim] where n is the size of the n-grams.

3.3 Classification

For the classification layer, we will use a linear layer to project to the number of classes and sigmoid that turns scores into probabilities. Here we will use the binary cross-entropy loss since we have a binary text classification problem.
Note that most of the functions are already implemented for you.

Step 1: Download and load the dataset Run the code to load the dataset.

Step 2: Print first characters of the dataset Fill in the relevant code.

Step 3: Run the data processing steps In this step you will pre-process the data and split them in order to create a list of reviews and labels as indicated at the .ipynb file.

Step 4: Load the word embeddings Load the word embeddings and follow the instructions at the notebook in order to be familiar with the relevant methods from the gensim library .

Step 5: Tokenize the reviews and create sequences of the same length Fill in the relevant part of the notebook.

Step 6: Build CNNs Build CNNs that are able to encode 3-grams, 4-grams, 5-grams. Instructions and details are available at the notebook.

Step 7: Train and evaluate your model The code for that is already available at the notebook except for the fact that for evaluating your model you should get the predicted outputs from the model and transform the probabilities to classes.

References

[1] `https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html`

[2] `https://pytorch.org/docs/stable/generated/torch.nn.MaxPool2d.html`

[3] `https://github.com/bentrevett/pytorch-sentiment-analysis`

[4] `https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks`

[5] Kim, Yoon (2014) *Convolutional Neural Networks for Sentence Classification*, Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), `https://aclanthology.org/D14-1181`