Statistical NLP: course notes

Çağrı Çöltekin — SfS / University of Tübingen

2020-07-13

These notes are prepared for the class *Statistical Natural Language Processing* taught in Seminar für Sprachwissenschaft, University of Tübingen.

10 *Deep neural networks*

Deep learning refers to a set of machine learning methods that have recently been (re)popularized. One of the important aspects of the deep learning is the use of deeper neural networks with more than one hidden layers. They have been successfully applied to many machine learning tasks, and they are also the dominant approach used in the natural language processing. Deep ANNs are not just fully-connected feed-forward networks with multiple hidden layers as the one presented in Figure 10.1. Besides the use of multiple layers, the deep learning architectures used in practice diverge from the typical feed-forward networks by use of *sparse* connectivity and *weight sharing*.

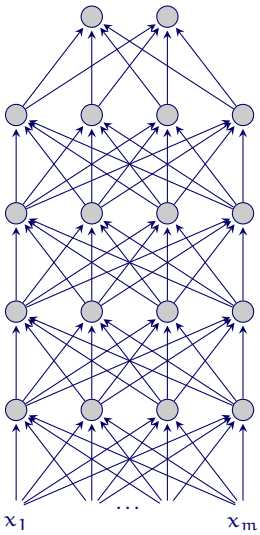

Figure 10.1: A deep feed-forward (fully-connected) network.

Earlier, we noted that an ANN with a single hidden layer is a universal function approximator. That is, it can approximate any computable function with arbitrary precision. Then, a natural question to ask is 'why should one use more than one layer?' The first reason is related to the formal proof that ANNs with a single hidden layer are universal function approximators. The proof is very general, and there is no way to tell how many units are needed in the hidden layer. The second reason has to do with the fact that certain problems seem to suit well to ANN architectures with multiple layers. These involve problems where layers, or hierarchies of features are useful. A common example is object recognition in images. Recognizing objects (such as cars, animals or faces) in images involve recognizing simple shapes that occur in these objects, which are combination of even simpler lines or curves (e.g., edges in the image) which in turn are certain combinations of pixels. These features build on each other, and consecutive layers in a network learn such hierarchy of features. However, the depth does not have to be only for a hierarchy of features. As we will see soon, representing a temporal sequence also results in effectively increasing the depth of the network.

Although there has been many interesting recent developments, many of the ideas that drive current deep learning methods were developed in 1980's and 1990's. The most important reason for the present success and the renewed interest is probably the developments in computing hardware. Particularly, availability of vector processors, such as graphical processing units (GPUs) in personal computers, that perform linear algebra operations efficiently made training large neural networks feasible. The increased availability of labeled and unlabeled data is another reason. At present, the deep

networks are the default or dominant method in many fields, including the NLP. In this lecture, we will introduce two architectures that are commonly used in NLP, namely *recurrent neural networks (RNNs)* and *convolutional neural networks* (CNNs), and discuss some of the common practices and issues that arise while training deep networks.

10.1 Recurrent neural networks

Recurrent neural networks (RNNs) are sequence learning models. Unlike feed-forward networks which have only a forward flow of information during prediction, RNNs include (time-delayed) loops. Figure 10.2 presents a typical RNN. Without the thick recurrent link presented, the RNN is simply a feed forward network. What makes RNNs special is the backwards loop over the hidden units. This makes an RNN to use the information in the previous hidden states as well as the current input. Hence, although the RNNs process a single input item (e.g., word) at a time, the output of the hidden layer is a function of the output of the hidden layer in the previous time step as well as the current input. As a result they can make use of the information from the past observations (e.g., earlier words).

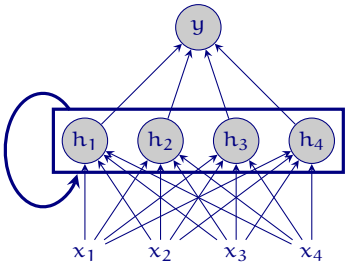

Figure 10.2: A schematic representation of a recurrent network. The thick recurrent link on the hidden layer indicates connections from each hidden unit to every hidden unit (including itself).

Another way to look at a recurrent network, often used for introducing *simple recurrent networks* (SRNs) is to assume that we have a set of 'context' units which are the copies of the hidden units from the previous time step. This is shown in Figure 10.3, where the special link labeled 'copy' does not have any learned weights, but the other links, including the one from the context unit to the hidden units, have weights that are learned. As a result, the hidden units can combine the information from the past hidden representation and the current input. This representation should make the forward operation of an RNN clearer.

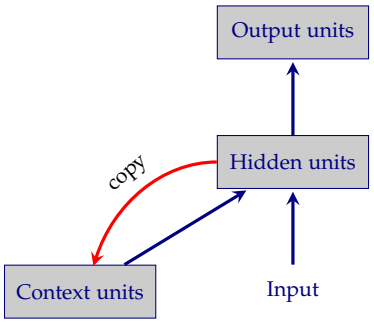

Figure 10.3: Another schematic representation of a recurrent network, which used describing simple recurrent networks (SRNs, also known as Elman networks). The link with label 'copy' does not have any associated weights.

In an SRN, like the one presented in Figure 10.3, it is also possible to apply the standard backpropagation (BP) algorithm, since the weights that are learned are feed-forward. However, applying the backpropagation in this architecture means that error is not back-propagated more than one time step.

In modern recurrent networks, a modified version of the BP algorithm, often called *backpropagation through time* (BPTT), is used. To understand the BPTT, it is useful to *unfold*, or *unroll*, the network. An unrolled recurrent network is presented in Figure 10.4.

The representation in in Figure 10.4 describes the same type of network depicted in Figure 10.2. In the unrolled network in Figure 10.4, each time step is shown separately. This representation also shows the network is effectively a deep feed-forward network. And application of the back propagation algorithm is also straightforward. The error made at any time step is reflected to the input and hidden layer weights before this time step. It is also important to realize that most parameters are shared. The weights that are shared are shown with the arrows with same color in Figure 10.4.

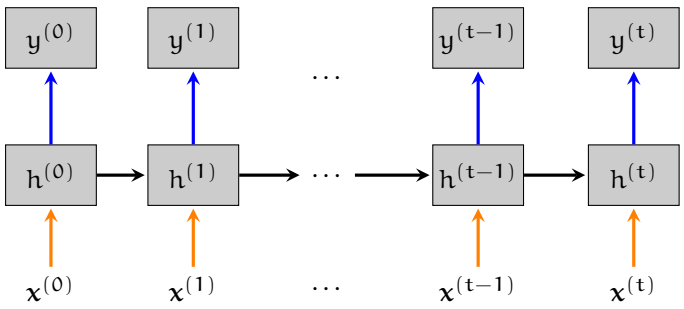

Figure 10.4: An unrolled RNN. The superscripts indicate the time steps. Note that the weights represented with the links with the same color are *shared*.

Although BPTT gives us a way to apply BP to recurrent networks, the (time) depth in RNNs leads to a problem called *unstable gradients*. To appreciate the problem, remember that updates applied to weights in each layer is calculated using the chain rule of derivatives. As a result, the update applied to the weights in earlier stages of a deep network will be composed of a long chain of (matrix) multiplications. Multiplying a series of numbers with absolute values less than 1 will result in a number close to 0, slowing down learning, maybe to the extent that nothing is learned. Similarly, multiplying a series numbers with absolute values greater than 1 will cause the error signal to be too large, causing instabilities due to large weight updates. The former case is called *vanishing gradients*, and the latter is called *exploding gradients* in the literature.

To solve the exploding gradients, often a simple technique called *gradient clipping* is used. Gradient clipping simply means truncating the gradients larger than a particular value. The solution of the vanishing gradients is more involved.

10.1.1 Gated recurrent networks

Gated RNNs are a solution to the vanishing gradient problem. The general idea with gated RNNs is to use possibly multiple gating mechanisms that determine which dimensions of the hidden representation kept for a long time or forgotten quickly. The architectures used are rather complex. We will not go into the details of the gated recurrent networks in this class. However, we briefly mention two variants that are popular in the field.

The *long-short-term memory* (LSTM) cell, which is presented in Figure 10.5, controls the information kept, added or removed in the hidden representation through a number of 'gates'. The LSTM keeps two vectors of hidden representations, the one called the 'hidden state' ('$\mathbf{h}$' in the figure) and the other one is called the 'cell state' ('$\mathbf{c}$' in the figure). Both the cell state and the hidden layer are passed to the next time step after a number of operations. The unshaded square blocks in the figure are called gates. They are simply ANN layers with sigmoid activation function. The circles (and the ellipse) in the figure represent element-wise vector operations. The forget gate (σ_f) controls what is removed (or kept) from the cell state and they control what is removed, and added to cell state based on the

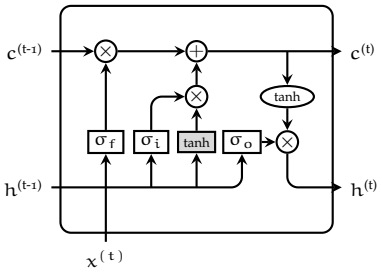

Figure 10.5: A schematic representation an LSTM cell. The drawing similar to the ones from a blog post by Chris Olah.

previous hidden state and the current input. The input gate (σ_i) controls what is added to the cell state. And the output gate (σ_o) controls the hidden unit output.

The LSTM and its variants have been used successfully in many sequence learning tasks. A somewhat simpler variant, called simply *gated recurrent unit* (GRU), has also become quite popular, and likely to be found in many standard neural network tools and libraries. The gated RNNs are complex models, the success of one variant or the other differs in different applications. However, in for most uses, gated RNNs yield better results than simple RNNs.

10.1.2 Different uses of RNNs

RNNs have been used in a number of different linguistic problems. The architecture is flexible, and can be extended in many ways. Here, we briefly go through some of the common variations.

A very common practice is to use *bidirectional* RNNs. A bidirectional RNN is composed of two RNNs, one run forward as we discussed above, and another one run backwards through the sequence. The hidden representations from both RNNs are then combined and fed to the later layers in the network architecture. Unless the application requires online sequential processing, bidirectional networks are possible, and often perform better than unidirectional variants. A bidirectional RNN is shown in Figure 10.6.

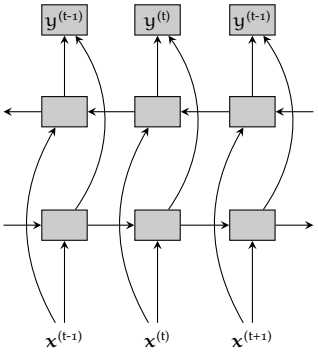

Figure 10.6: A bidirectional RNN.

RNNs can be used for a typical sequence model such as hidden Markov models, In this case, the we use output of the RNN at each time step to predict a label as shown in Figure 10.4. Such a network learns a one-to-one mapping between equal-length inputs and the outputs. The output layer is typically a classification (e.g., using softmax activation). This type of RNNs have many applications in NLP typical examples including POS tagging, and named entity recognition (NER).

Another use of RNNs is depicted in Figure 10.7. In this case the intermediate representations build by the RNN is not used for any prediction. The network builds a representation $h^{(t)}$ for the whole sequence, and this representation is used for assigning a label to the sequence. This RNN configuration is used frequently for sequence classification tasks, e.g., text classification tasks like spam detection.[1]

[1] A variation of this architecture, where all the intermediate representations are combined somehow for a single final prediction is also common.

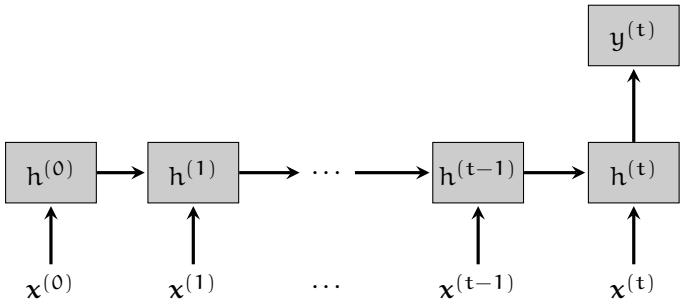

Figure 10.7: An RNN for sequence classification. Only the final representation built by the RNN is used for prediction.

One final standard variant that is interesting for NLP applications

called a *sequence-to-sequence* (or seq2seq) network. In fact, this is an encoder–decoder architecture, where both encoder and the decoder are recurrent networks. In this setup, shown in Figure 10.8, the encoder RNN builds a representation for the complete input sequence, which typically is terminated by a special end-of-sequence symbol. The decoder's hidden layer is initialized using this representation, and it is expected to produce the output sequence, followed by the end-of-sequence symbol. A very common variation in many applications is to provide the previous output as input to the encoder (shown with the gray curved arrows in the figure). Note that we can train the network using the gold-standard output sequence. However, at prediction time, the model has to rely on its own output.

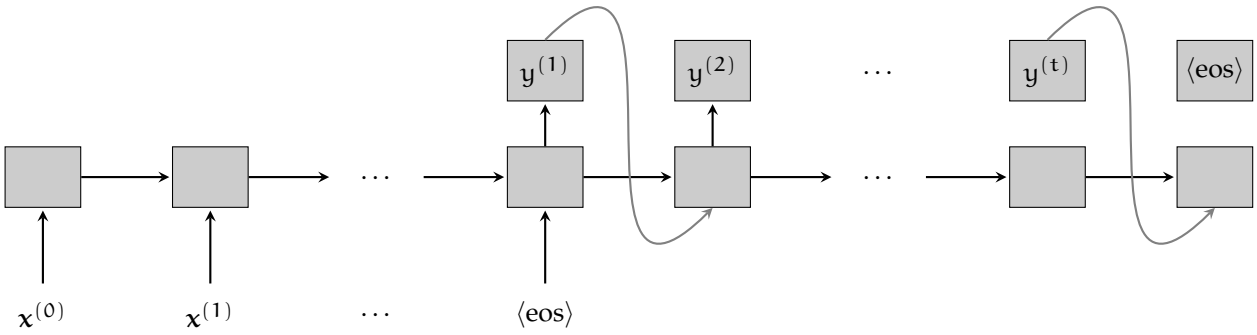

Figure 10.8: A sequence-to-sequence model.

Sequence-to-sequence networks similar to the one in Figure 10.8 are capable of transforming a sequence to another sequence with a different length. They are used in many applications, probably most popular application being machine translation. The modern seq2seq models are generally more complex than the one described above. A very popular extension to such models, called an *attention mechanism* to provide the intermediate representations built by the encoder to the decoder time steps, often passing through another network component that learns what parts of the input is more important for the present prediction task.[2]

It is also common to use deeper, stacked, RNN layers for both encoder and the decoder, and bidirectional RNNs for the encoder part of the network.

[2] The attention idea was even developed further, leading to architectures using only attention without recurrent links.

10.2 Convolutional networks

Convolutional neural networks (CNNs) are another type of popular ANN architecture. They have become particularly popular in image processing tasks, but they also made their way into language processing.

Convolution is an operation, a filter, applied to a signal, which

transforms it based on the neighboring values at any point. It has its roots in in signal processing, where it is typically applied to a continuous signal. However, for our purposes it is a filter that transforms each discrete unit based on its neighbors.

In image processing a filter is typically a square matrix, that slides over the complete image to transform every pixel, as demonstrated in Figure 10.9. Note that the convolution is not well-defined on the pixels at the edges of the image. In practice, one 'pads' the images (with values appropriate for the filter applied) to obtain a transformed image with the same size as the original image. Otherwise, the result would be a smaller image (as in Figure 10.9).

In standard image processing software many of the operations on images are done through convolution operations. Figure 10.10 shows two common filters used by image processing software. The first one, blurring, replaces a pixel with a weighted average of its neighborhood, making all pixels similar to their neighbors and removing the details from the image. The second filter shown, edge detection, is probably more useful for machine learning. The filter replaces pixels with intensities similar to its neighbors with numbers close to 0, while the pixels that are different from their neighbors are assigned to larger intensity values.

The fixed filters demonstrated above are useful (and used) in image processing software. However, in machine learning applications we want to *learn* these filters from data. In a typical CNN application, we learn many such filters, trained on the task we are interested in.[3] The hope is that each filter extracts some useful features. For example, edges with different slants from given pixels. It is also very common to stack the convolutional layers. In a nutshell, the idea with multiple layers of convolution is to learn a hierarchy of filters. Continuing with the examples with edges with different orientations, another layer build on edges may learn useful (geometric) shapes, and yet another layer may recognize object parts composed of these shapes, and so on. Figure 10.11 first two stages of this hypothetical scenario. If our aim is, for example, predicting whether an image contains people, convolutions over the shapes shown in the lower part of Figure 10.11 are likely to be useful.

We discussed convolutions in the context of 2D objects (images), since they are most commonly used in this area. However, they can easily be extended to 3D objects (e.g., for processing videos), or applicable to 1D sequences (for speech and language processing).

We now look at the convolutions more closely, but assuming that we work on a single-dimensional sequence. The units in the sequence, for our purposes, can be (representations of) words, characters, phonemes, or other linguistic objects. Figure 10.12 shows convolutions applied to such a sequence. If we were running this convolutional network on words, the convolution would learn something (useful) about word trigrams. For example, the final aim is sentiment classification, this convolution would result in higher values at the hidden representation if for trigrams that are associated with nega-

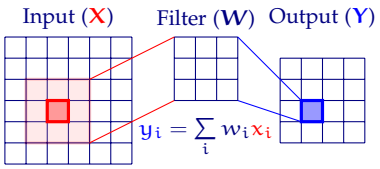

Figure 10.9: A demonstration of convolution in image processing. Every pixel in the image is passed through a filter, where the transformed value of the pixel is a weighted combination of its original value and its neighbors.

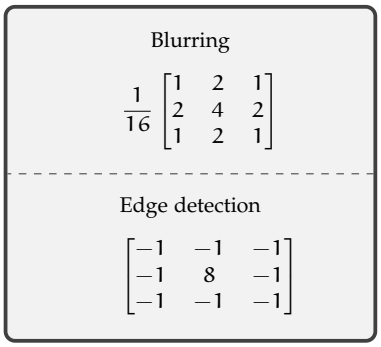

Figure 10.10: Two example filters (convolutions) used in image processing: blurring (top) and edge detection (bottom).

[3] The values in the filter matrices above are the parameters that we want to learn.

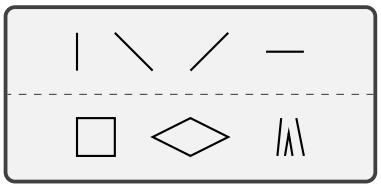

Figure 10.11: Results of possible applications of layers of convolutions. First layer of convolution may learn different filters for edges with different orientations (top), while another layer built on it may learn geometric shapes built with them (bottom). Yet another layer may be used to detect objects, like houses, windows, people, based on these shapes (not shown in the figure).

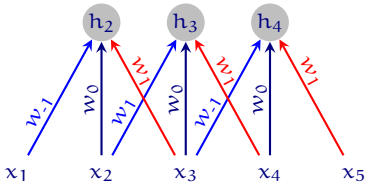

Figure 10.12: Demonstration of 1D convolution. The weights indicated with the same colors are the same regardless of their position in the sequence.

tive or positive sentiments. There are two aspects of the CNNs that set them apart from typical neural networks such as MLP. First, the weights are *shared*, the same filter is run through the entire sequence without modifying the weights during prediction. And second, the input layer and the hidden layers are not fully connected. As well as being suitable for picking certain features, these aspects reduce the number of parameters learned, and complexity of the network. In practice, many such filters (possibly with different input window size) used in combination with multiple layers of convolutions, and finally with a fully connected prediction layer, e.g., a sigmoid or softmax classifier.

Returning to the example of sentiment classification, a CNN layer like the one in Figure 10.12 will discover a trigram with high-sentiment content wherever it occurs in the sequence. However downstream classification layer need to still consider hidden layer activations as separate features. In many problems, we do not want this location sensitivity. For example, a phrase like *not worth seeing* in a movie review is an indication of a negative sentiment wherever it appears. To make the features learned by convolutions *location invariant*, a concept called *pooling* is applied. Pooling simply calculates a statistics, most commonly 'maximum' over a range of its inputs. When it is applied to convolutions as in Figure 10.13, the new features are relatively location invariant. Another aspect of the pooling to note is that, it is a fixed operation, there are no weights to learned in the pooling layer.

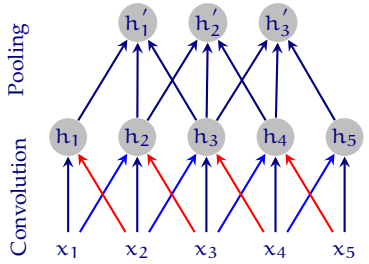

Figure 10.13: Demonstration of pooling.

If we apply the 'max pooling', the value of h'_1 in the figure is the maximum of h_1, h_2 and h_3, which means if the convolution detects any interesting trigrams from x_1 to x_4, it h'_1 will indicate it. Hence, to some extent, a classifier that uses the output of the network shown in Figure 10.13 will be insensitive to the location of the feature detected by the convolution. However, Figure 10.13 still retains some location sensitivity, which may be useful for some applications. For example, for a face recognition network, it is likely important to detect eyes above a nose. In problems where location is not useful at all (which is the case in many text classification examples) one can pool over the complete convolution output, passing a single feature to the classifier from this filter. Remember that we typically use many convolutions, hence, in this case, the classifier will be given a single feature from each convolution.

For both convolutions and pooling, the examples we looked at so far cover the whole sequence by shifting the filter one unit at a time. It is common to define a larger *stride*, that shifts the convolution or pooling over more than one unit at a time. Figure 10.14 repeats the network shown in Figure 10.13 with a stride of 3 on the pooling layer. With this configuration, each output of the pooling layer covers exactly half of the convolutions. However, note that due to hierarchical nature of the network, they are affected by larger spans of input.

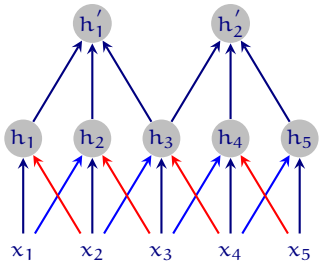

Figure 10.14: The same network presented in Figure 10.13, but the pooling layer has a stride of 3.

Most CNNs used in practice are deep, resulting in diminishing numbers of features when successive convolution or pooling layers

are stacked without *padding* as shown in Figure 10.15. This is sometimes called *valid* padding, meaning that each convolution is calculated on real data. However, it is a common practice to pad the input, typically with 0s, so that the output of the network stays stable. Figure 10.16 shows an example of padding applied to the same network. Here, each layer is padded from both sides with a single 'imaginary' input. For an image, this would mean padding the matrix from all sides. With a stride of 1, padding only one unit from all sides as in Figure 10.16 results in an output the same dimensions as the input. Hence, it is often called *same* padding. With a larger stride, however, this would result in a reduction in the output dimension more than expected from the stride. In such cases it is an option to pad as many values as necessary to make sure that the output is of expected dimension, which is sometimes called *full* padding.

In NLP, most common use of CNNs is text classification. Given an input text, we typically define multiple convolutions, or filters. Each convolution, after training, will detect an 'n-gram' feature with the width of the convolution. Although in theory a larger convolution width should learn features within its window that are based on a (discontinuous) sub-sequence, examples of convolutions with different sizes are sometime used. Figure 10.17 demonstrates a possibly way to use CNNs for text classification. The first layer in the network is typically an *embedding* layer, a dense (as opposed to sparse one-hot) representation of the words (we will cover embeddings later in this class). The example uses three convolutions, first two with width 2, and the last one with width 3. The first convolution in the example, finds something interesting in input bigram *not really*, while the second one is more sensitive to bigram *really worth*. The network does a max pooling over the whole sequence, and then uses the resulting representations as an input to a classifier with a single hidden layer. The final layer, a single (likely sigmoid) unit is used for binary classification.

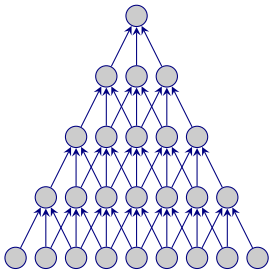

Figure 10.15: A deep CNN network without padding.

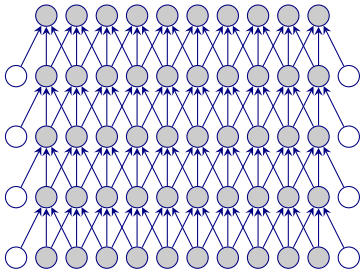

Figure 10.16: The same network in Figure 10.15 with padding.

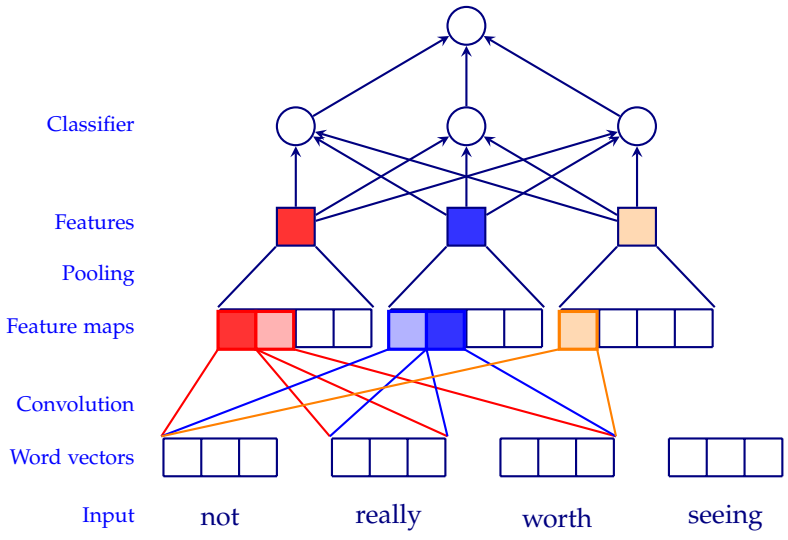

Figure 10.17: A demonstration of CNNs used for text classification.

In real-life examples, especially in image processing systems, CNNs are typically deeper and also more structured. This may make their training difficult despite the sparse connectivity and shared weights. For very large systems, it is also a common practice to pre-train components of the network piece-by-piece. In the NLP applications like the one demonstrated in Figure 10.17, the embeddings are typically trained separately, and most of the time training continues on the task as well.

Summary

In this lecture we covered some of the methods and practical issues in deep learning architectures. The two architectures we introduce, recurrent neural networks and convolutional neural networks are rather 'traditional' methods in the field. As of this writing, methods and applications of deep learning are active areas of research. And, the perceived 'best architecture' for a particular task often shifts in a short time. A notable recent development we did not cover is the so-called transformer architecture used as a non-recurrent alternative to RNNs (Vaswani et al. 2017).

In any case, the concepts introduced in this lecture is important to understand newer, more complex methods. Similar to many other subjects in machine learning, it probably better to supplement this lecture from our usual machine learning textbook references. However, due to popularity of deep learning, the online information on these methods are abundant, and it may also be a good idea to follow some of the online practical tutorials.

Bibliography

Vaswani, Ashish et al. (2017). "Attention is all you need". In: *Advances in neural information processing systems*, pp. 5998–6008.