Topic 6: Convolutional neural networks

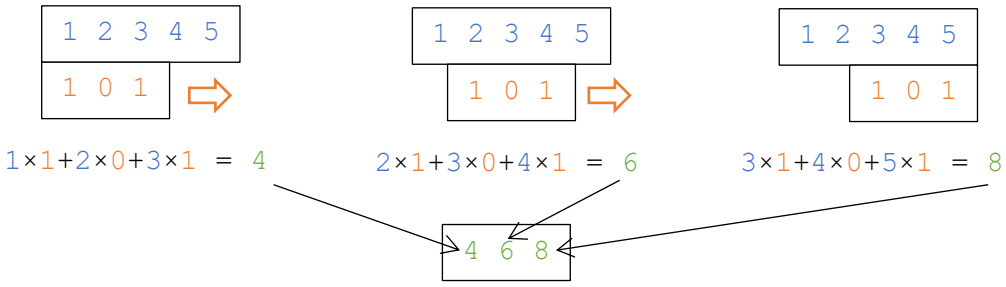
A fully connected neural network (all the neural networks seen till now) can learn to do anything but also has many parameters. More parameters requires more memory, takes longer to train, and requires more training data to learn a generalised function (since more parameters makes it easier to overfit). A fully connected neural network is also the most general way to process data, which means that you're not helping the neural network by limiting it to only do what is likely to work.

One way to limit how a neural network can process input structures is to use convolutional layers, which are layers that are inspired by the way the eye works. Rather than process the whole input at once, you make the neural network take in chunks of the input and process them separately, then combine the results afterwards. This is similar to how you read the text bit by bit rather than all at once.

1. Convolutional layers 1D

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/06_-_Convolutional_neural_networks/01_-_Convolutional_layers_1D.py

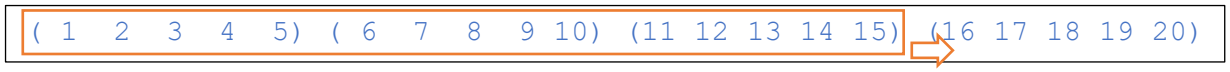
A convolution is a mathematical operation that involves sliding a tensor over a larger tensor of the same dimensionality. The smaller tensor is called a kernel or filter. At every position that the kernel is slid to, you multiply the overlapping elements of the two tensors and add the products. The result is placed in the corresponding position in a resultant tensor. Here is an example using vectors, also known as a 1D convolution, with the kernel in orange, the tensor being convolved in blue, and the resultant tensor in green:

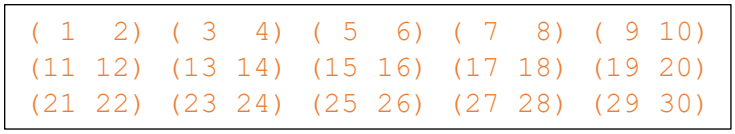


You can also control the stride of the convolution, that is, the amount by which the kernel moves after each step.

In neural networks we assume that a sequence consists of vectors rather than scalars, which means that a 1D convolution is actually applied to a matrix rather than a vector. It's actually applied to a batch of sequences so a 1D convolution is actually applied to a 3D tensor. The resultant tensor would also be a 3D tensor. In practice, the sequence could be a sentence with each vector in the sequence being a word vector.

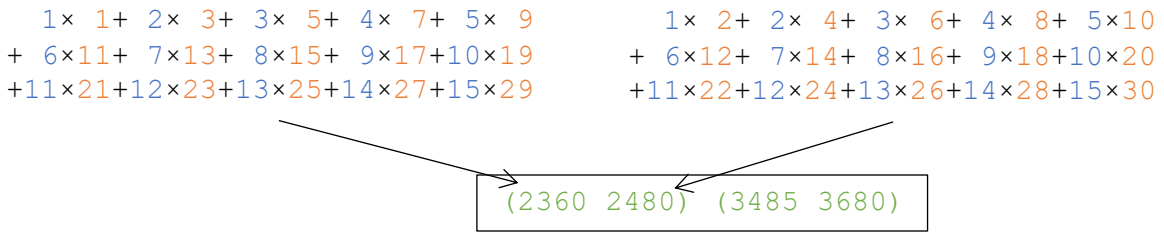
Convolutions become more complex when we consider that the size of the vectors can be changed after a convolution. In fact, a kernel should have three dimensions: the first dimension is for the kernel width (in the above diagram it was three), the second dimension is for the vector size in the original tensor, and the third dimension is for the vector size in the resultant tensor. Below is an illustration:





In the kernel:

- Each row handles a different part of the window (window size is 3)
- Each column of vectors handles a different dimension in the original tensor's vectors (original tensor has 5 element per vector)
- Each dimension in the kernel vectors handles a different dimension in the resultant tensor (resultant vector will have 2 elements per vector)



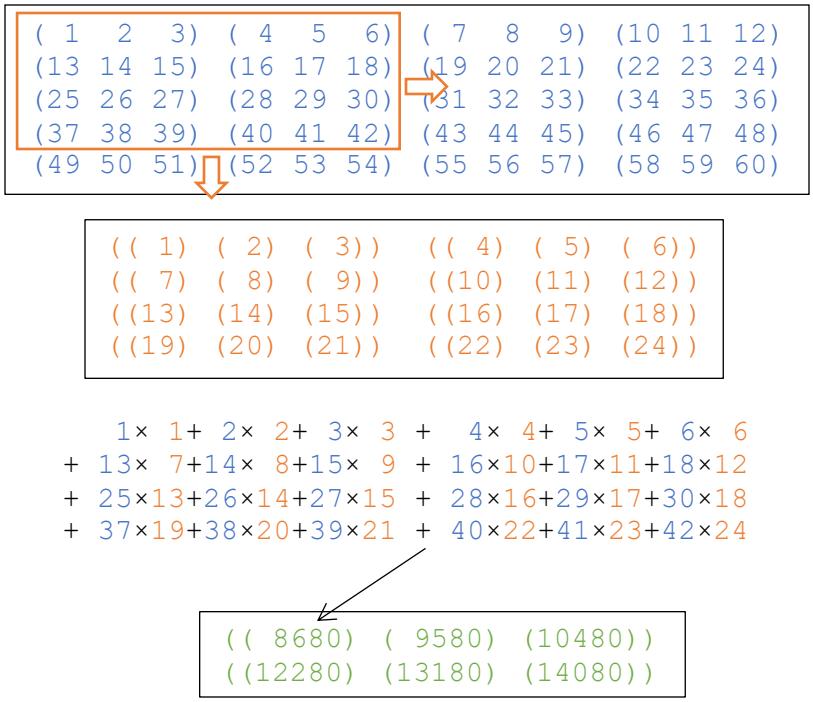
2. Convolutional layers 2D

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/06_-_Convolutional_neural_networks/02_-_Convolutional_layers_2D.py

We can also apply convolutions on grids rather than just linear sequences. This would be called a 2D convolution. In 2D convolutions the kernel would slide along the two dimensions rather than slide along one dimension as was the case above. In practice, the grid could be an image.

Again, in neural networks our grids consist of vectors and are in batches so we're really applying the convolution on a 4D tensor rather than a matrix. The resultant tensor would also be a 4D tensor.

Note that the kernel will now also have four dimensions instead of three as there is now a width and a height instead of just a width. The first dimension is for the window height, the second dimension is for the window width, the third dimension is for the vector size in the original tensor, and the fourth dimension is for the vector size in the resultant tensor. Below is an illustration:



In the kernel:

- Each triple handles a different part of the window (window size is 4x2)
- Within each triple, each column of vectors handles a different dimension in the original tensor's vectors (original tensor has 3 element per vector)
- Each dimension in the vectors handles a different dimension in the resultant tensor's vectors (resultant vector will have 1 element per vector)

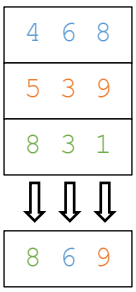
3. CNN 1D pooled

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/06_-_Convolutional_neural_networks/03_-_CNN_1D_pooled.py

One way to use a convolutional operation in a neural network is to treat the sliding kernel's numbers as weights that need to be learned. The input is first convolved, then a bias is added to each vector element in the whole resultant tensor, and finally each element in the resultant tensor is passed through an activation function.

$$y = f(\text{conv}(x, W) + b)$$

When a neural network makes use of a convolutional layer, the neural network is called a convolutional neural network (CNN). Following the convolutional layer, the next step in a CNN is to turn the convolutional layer's result (activations) into an output of the neural network. The layer's result must be converted into a single vector in order to then pass it to a sigmoid or softmax layer. In other words, we need to squash all the individual vectors in the resultant sequence into a single vector. The most common way to do this is by max pooling, meaning that you take the maximum value at each corresponding position in all the vectors like this:



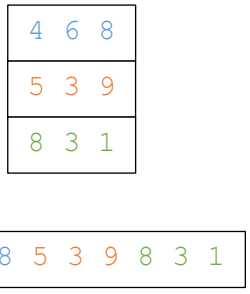
Max pooling has an interesting interpretation. Let's assume that we're using a convolutional layer on a sequence of word vectors to determine if the sequence is a positive or negative sentence. If one of the elements in the word vectors is taken to be a flag which says whether the word is typically used in positive sentences or not, then taking the maximum of all the corresponding flags in the words is equivalent to checking if at least one of the words has the flag. In other words, it performs an ANY operation on a sequence of true/false values. If minimum pooling is used instead then we can perform an ALL operation instead, but this is not commonly used.

In the provided script, there is a 1D CNN performing sentiment analysis. It will decide this using a kernel of length two words.

4. CNN 1D fully connected

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/06_-_Convolutional_neural_networks/04_-_CNN_1D_fully_connected.py

Another way to convert the convolution layer's result into a single vector is to just concatenate all the vectors into a single giant vector, a process called flattening, like this:



The disadvantage of this method is that it forces the sentences to always have the same number of words. Given that the output layer will be a fully connected layer which multiplies the flat vector by a weight matrix, the flat vector must always have the same size or it will not be possible to multiply it by the weight matrix. This means that the number of words must always be the same. On the other hand, it does away with the concept of flags that indicate whether there is some important feature or not in the sequence of vectors and instead lets the output layer make decisions based on each separate vector in the sequence which means that the position of a word matters now.

5. CNN 2D pooled

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/06_-_Convolutional_neural_networks/05_-_CNN_2D_pooled.py

Although convolutions can be used on sentences, their typical use is on images. Here we process 4D tensors (batches of images with vector pixels) using max pooling again.

In the given script, we will be classifying 3 by 3 pixel images of binary pixels. Each image contains shapes and the task is to classify whether the shape is a single line. We shall use a 2 by 2 kernel to do this. After running the script, notice how the neural network cannot learn that the empty image has no lines because it can only use independent 2 by 2 square windows to determine whether there is a line or not and just because there is nothing in one square does not mean that there is nothing in the whole image.

6. CNN 2D fully connected

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/06_-_Convolutional_neural_networks/05_-_CNN_2D_fully_connected.py

Max pooling is not typically used to turn image convolutions into fixed vectors. Instead, they are usually flattened. This is less of a problem for images than sentences because, unlike sentences, images can be resized to always have the same width and height.

7. CNN 2D fully connected 2 layer

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/06_-_Convolutional_neural_networks/07_-_CNN_2D_fully_connected_2_layer.py

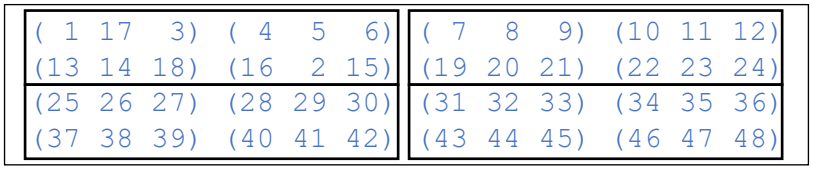
The result of passing an image through a convolutional layer is another image, sort of. The result can be treated as an image because it has the same number of dimensions. This means that a convolutional layer can process the result of another convolutional layer. This gives rise to multiple layers of convolutions, just like we could have multiple layers that are fully connected. Of course the same can be applied to 1D convolutional layers.

8. CNN 2D down sampled

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/06_-_Convolutional_neural_networks/08_-_CNN_2D_down_sampled.py

Since images typically have a lot pixels, the neural network will tend be big and slow. To reduce this problem, we can resize the images to be smaller, but this would mean less information to work on. Instead, we can let the neural network do the resizing by only allowing some of the pixels to pass through to the next layer. This is known as down sampling. This process is similar to that of max pooling which we saw earlier but instead of being applied to the whole grid of vectors (or sequence of vectors in 1D convolutions), it is applied to groups of vectors.

A neural network can down sample the result of a convolutional layer by grouping the vectors into windows (with no overlap) and then transforming each group into a single vector. Below is an illustration of an image being grouped in 2 by 2 groups:



By replacing each group with a single vector, the image becomes a quarter of its original size. The vector that is produced from each group is the maximum value of each corresponding vector position. For example, in the top-left group of the above image, the vector replacing the group would be `(16 17 18)`.

Note that this is not applied to the input image but to the result of a convolutional layer and can be applied after each convolutional layer. Generally, practical neural networks would consist of three stages per layer: a convolution, a ReLU, and a down sample.