CIS 522 - University Of Pennsylvania, Spring 2020

Lecture 6: Introduction to CNNs

4 February 2020

Lecturer: Konrad Kording *Scribe: Saket, Ben*

1 Visual processing in the brain

What we see in front of us informs our behavior – to reach for objects in front of us we must know their location, and to speak about objects in front of us we must recognize them. Our visual system extracts this information from visual inputs. Significant progress was made in understanding the mammalian visual system in 1959 by Hubel and Wiesel. While recording the activity of neurons in a region of cat visual cortex known as 'V1' (primary visual cortex), they presented the animal with moving bars of light, at different angles. By repeating such an experiment many times with different angles and neurons, they identified two types of neurons. They termed them simple cells and complex cells. Simple cells responded strongly to stimuli at a particular orientation, and in a specific spatial location in their visual field. Complex cells on the other hand responded strongly to stimuli moving in a particular direction, but had some degree of spatial invariance – it didn't matter as much where the oriented bar was located to elicit the same response.

Since then many other regions of the cortex have been recorded from while being presented with different visual stimuli of varying complexity. What has emerged is a picture of hierarchical visual processing. Visual inputs enter the retina and excite neurons through a hierarchy of brain regions. Neurons early in this pathway respond to low-level features like oriented bars in specific locations (i.e. simple cells in V1 cortex). Complex cells obtain a degree of spatial invariance by receiving inputs from a number of simple cells. Deeper into this hierarchy neurons respond preferentially to increasingly abstract and complicated inputs. At the end of this hierarchy, in a region of cortex named IT (inferior temporal cortex), neurons appear to respond to parts of behaviorally relevant objects, e.g., features with primitive face-like appearance. Along this pathway, the receptive fields of neurons (the regions in space in which they are sensitive to input from) grows in size; that is, there is increasing spatial invariance. This makes sense for the visual pathway involved in object recognition – to identify an object we do not need to know its exact location.

Many of the above features are in fact mirrored in convolutional neural networks. Indeed, convolutional neural networks were originally inspired by the early findings of Hubel and Wiesel. In 1980 Kunihiko Fukushima developed a model known as the neocognitron. This model contained alternating layers of S and C units, S for simple and C for complex. This model could successfully be trained to recognize, for instance, handwritten digits. Yann Lecun, in his work on convolutional neural networks has stated he was inspired by the neocognitron. The success of CNNs in computer vision and other areas has prompted interest in using CNNs as a model of the visual system. This remains a very active area of research.

2 What is a Convolutional Neural Network?

A Convolutional Neural Network (CNN) is very similar to a regular neural network covered in previous lectures. It is also made up of neurons, has learnable weights and biases. Input to each neuron is multiplied by the corresponding weights followed by a non-linearity to produce a hidden representation. Each layer is still followed by few more such layers which eventually expresses a differentiable function. The only difference is that the CNN is a special kind of neural network made for image inputs (or in general 2D or 3D input).

Why not use a fully connected neural network for image data?

Regular neural networks don't scale well to image data. A fully connected network for an image input of size $28 \times 28 \times 3$ (MNIST) will require 28 * 28 * 3 = 2352 neurons in the first layer. This is still acceptable. However, consider a dataset with images of greater size. For example, $300 \times 300 \times 3$ which is a reasonable assumption will now require 270,000 neurons. This drastically scales up the number of learnable parameters and thus makes the problem extremely hard for the network to learn. Moreover, images usually carry some spatial information per pixel. Concretely, any pixel cares about the pixels surrounding itself more than pixels which are on the other corner of the image. Clearly, having a fully connected network is wasteful.

A CNN thus works with a 3 dimensional arrangement of neurons. Thus, it takes a 3D input and produces a 3D output. Each neuron is connected only to a small number of neighbouring neurons. Figure 13 shows how a typical CNN looks like.

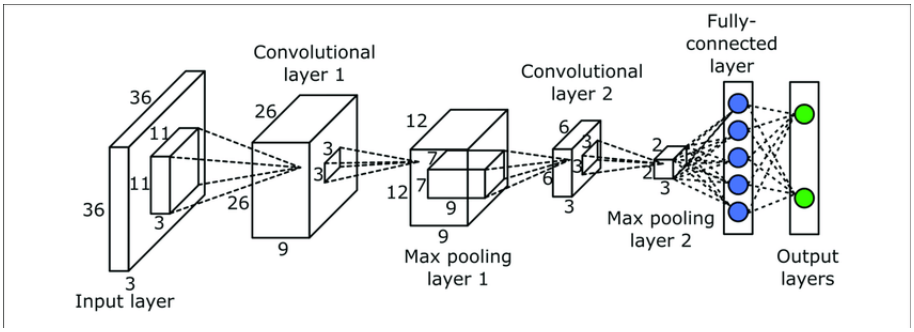

Figure 1: Structure of a typical CNN

There are three main types of layers in a CNN.

- **Convolution** - Computes the output of neurons that are connected to local regions in the input. More on this in section 3.
- **Pooling** - This layer performs a downsampling along the spatial dimensions. More on this is section 3.
- **Fully Connected Layer** - This is same as a layer in a regular neural network. Each neuron is connected to every other neuron in the following layer. Usually used as the trailing layers of a CNN as a classifier or regressor depending on the task.

3 Convolution

3.1 The mathematical definition

In mathematics, a convolution is an operation on two functions f and g defined as an integral of the product of the two functions after one of them is reversed and shifted. The convolution operation expresses how the shape of one function is modified by the other function. More formally, the convolution operation can be expressed as,

$$(f * g)(t) \triangleq \int_{-\infty}^{\infty} f(\tau)g(t-\tau)\,d\tau.$$

However, this is **not** the same convolution operation in deep learning. A related concept called cross-correlation is what deep learning literature refers as convolution. Cross-correlation is similar to convolution (from math) where the only difference is that one of the functions is reflected along the y-axis. Concretely, the cross correlation of $f(x)$ and $g(x)$ is same as the convolution of $f(-x)$ and $g(x)$ or $f(x)$ and $g(-x)$. Formally, cross-correlation is defined as,

$$(f \star g)(\tau) \triangleq \int_{-\infty}^{\infty} \overline{f(t)}g(t+\tau)\,dt$$

Figure 2 summarizes the difference between convolution and cross-correlation.

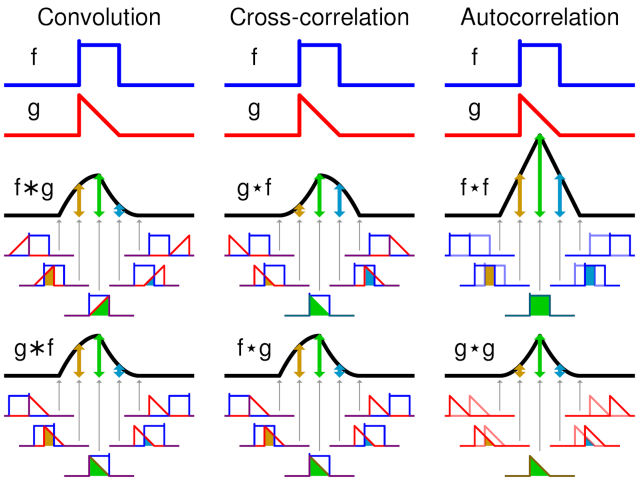

Figure 2: Comparison between convolution, cross-correlation and autocorrelation on a 1D input

3.2 The Convolution Layer

In deep learning, the convolution operation is actually the cross-correlation operation as explained above. However, we need to define it in the discrete domain since images are discrete in nature. Concretely, a conv layer consists of a set of learnable kernels (also called filters). The size of the kernel defines the local region of the input every output neuron will see in the convolution operation. Usually this kernel is small along the spatial dimensions (width and height) but covers the entire depth of the input. For example if the input size is $28 \times 28 \times 3$, then the kernel can be of size $5 \times 5 \times 3$. In the forward pass, the kernel slides over the input computing the output by

taking a dot product between the input and weights corresponding to the pixel on which the kernel is centered. Since the depth of the kernel is usually equal to the depth of the input, this operation produces a 2D output for every filter on 3D inputs. We use multiple such kernels in each layer which is stacked along the depth axis to get a 3D output. For example if each kernel produces an output of 20×20 for the $28 \times 28 \times 3$ input, with 12 such kernels the output of this layer will be $20 \times 20 \times 12$.

The size of the kernel is one of the hyperparameters of this layer which defines the receptive field of each output neuron. Note that typically we use square kernels where the width and height of the kernel is equal. However, some architectures might use kernels with unequal width and height. There are three other hyperparameters that define the size of the output - number of kernels, stride and padding.

- **Number of Kernels** - This is simply the number of kernels used in this layer which defines the depth of the output of this layer. It represents the number of features used to represent each pixel (spatial location) in the output.
- **Stride** - This defines the number of pixels in the input we skip when sliding the kernel over the input. For example, when the stride is 1, we calculate the output with the kernel centered at every input pixel. For stride of 2, we would skip 1 pixel when sliding over the input.
- **Padding** - When sliding over the input, it is not possible to center the kernel over the pixels along the edges. So, in order to consider these pixels as well we add add zeros along the border of the input (along all spatial dimensions). This also allows us to control the size of the output.

We can compute the size of the output for a convolution layer with a square kernel of size **F**, a stride **S**, padding **P** along each of the spatial dimensions and input of size $\mathbf{I_1} \times \mathbf{I_2}$ as,

$$O_1 = \frac{I_1 - F + 2P}{S} + 1$$

$$O_2 = \frac{I_2 - F + 2P}{S} + 1$$

For example, for an input of size $28 \times 28 \times 3$, kernel size 5, stride 2 and padding 2, we get an output of size ((28 - 5 + 2 * 2) / 2) + 1 = 14. Suppose we use 20 such filters, we will get an output of size $14 \times 14 \times 20$. Note that, we assume here that the kernel is of size $5 \times 5 \times 3$ such that the depth matches that of the input.

As we can see here, the learnable parameters in a convolution layer are independent of the input. They are only present as part of the kernel which slides over the input regardless of the size of the input. So for example, in a layer if we have 4 kernels in a conv layer each of size 3×3, the total number of parameters will be 3 * 3 * 4 = 36.

3.3 2D vs 3D Convolutions

The convolutions discussed so far refer to 2D convolution as each kernel produces a 2D output which is then stacked up from all the kernels along the depth axis to get a 3D output. We get a

2D output because we use a kernel whose depth dimension matches that of the input. The kernel slides over the input in only two dimensions. Figure 3 makes it easy to visualize the computation.

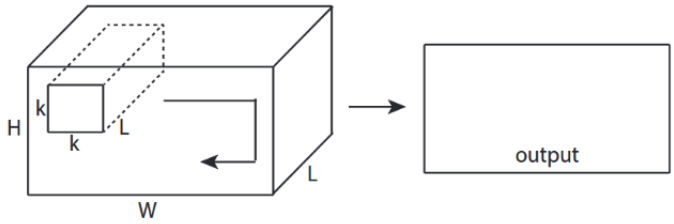

Figure 3: 2D Convolution

On the other hand, in case of a 3D convolution the output of a single kernel is 3D. This can be achieved by using a kernel whose depth dimension is strictly less than that of the input. The kernel will then slide over the input in all three directions. Figure 4 visualizes how a 3D convolution would look like.

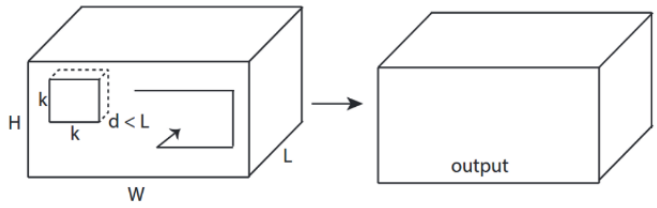

Figure 4: 3D Convolution

We rarely use 3D convolutions in practice since we usually want the features (depth axis) to be fully connected to the next layer.

3.4 Back Propagation through Convolution layer

Consider the forward propagation on the following input matrix and kernel.

$$\begin{bmatrix} X_{11} & X_{12} & X_{13} \\ X_{21} & X_{22} & X_{23} \\ X_{31} & X_{32} & X_{33} \end{bmatrix} \quad * \quad \begin{bmatrix} W_{11} & W_{12} \\ W_{21} & W_{22} \end{bmatrix} \quad = \quad \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix}$$

The forward propagation can be expressed with the following equations.

$$\begin{aligned} h_{11} &= W_{11}X_{11} + W_{12}X_{12} + W_{21}X_{21} + W_{22}X_{22} \\ h_{12} &= W_{11}X_{12} + W_{12}X_{13} + W_{21}X_{22} + W_{22}X_{23} \\ h_{21} &= W_{11}X_{21} + W_{12}X_{22} + W_{21}X_{31} + W_{22}X_{32} \\ h_{22} &= W_{11}X_{22} + W_{12}X_{23} + W_{21}X_{32} + W_{22}X_{33} \end{aligned}$$

The chain rule and the concept of computation graphs explained in previous lectures makes defining backpropagation for the convolution layer much easier. Since we get the gradients from

the next layers in the form of $\partial h = \partial L/\partial h$, our goal is to define only the local gradients for this layer i.e. $\partial h/\partial W$ and $\partial h/\partial X$.

By chain rule we can calculate $\partial L/\partial W$ and $\partial L/\partial X$ as,

$$\partial W = \frac{\partial L}{\partial W} = \frac{\partial L}{\partial h} \times \frac{\partial h}{\partial W}$$
$$\partial X = \frac{\partial L}{\partial X} = \frac{\partial L}{\partial h} \times \frac{\partial h}{\partial X}$$

Note that since W and X are matrices, we will have to calculate the derivative with respect to each element. The equations can be easily differentiated to obtain,

Gradients with respect to weights (to update the weights),

$$\delta W_{11} = X_{11}\delta h_{11} + X_{12}\delta h_{12} + X_{21}\delta h_{21} + X_{22}\delta h_{22}$$
$$\delta W_{12} = X_{12}\delta h_{11} + X_{13}\delta h_{12} + X_{22}\delta h_{21} + X_{23}\delta h_{22}$$
$$\delta W_{21} = X_{21}\delta h_{11} + X_{21}\delta h_{12} + X_{31}\delta h_{21} + X_{32}\delta h_{22}$$
$$\delta W_{22} = X_{22}\delta h_{11} + X_{23}\delta h_{12} + X_{32}\delta h_{21} + X_{33}\delta h_{22}$$

Gradients with respect to the input (to pass to the previous layer),

$$\delta X_{11} = W_{11}\delta h_{11}$$
$$\delta X_{12} = W_{12}\delta h_{11} + W_{11}\delta h_{12}$$
$$\delta X_{13} = W_{12}\delta h_{12}$$
$$\delta X_{21} = W_{21}\delta h_{11} + W_{11}\delta h_{21}$$
$$\delta X_{22} = W_{22}\delta h_{11} + W_{21}\delta h_{12} + W_{12}\delta h_{21} + W_{11}\delta h_{22}$$
$$\delta X_{23} = W_{22}\delta h_{12} + W_{12}\delta h_{22}$$
$$\delta X_{31} = W_{21}\delta h_{21}$$
$$\delta X_{32} = W_{22}\delta h_{21} + W_{21}\delta h_{22}$$
$$\delta X_{33} = W_{22}\delta h_{22}$$

3.5 Expressing convolution as matrix multiplication

The convolution operation is simply the dot product of the kernel with local regions of the input. We can take advantage of this and express the convolution operation as a simple multiplication of two matrices. This makes the computation of the convolution much more efficient. This is because there has been substantial amount of work done to speed up matrix multiplications on a GPU. Thus, expressing convolutions as matrix multiplications helps us get an exponential speed up.

Lets understand the process using an example.

Suppose we have an input of shape 4×4. Consider the following 3×3 kernel. With a stride of 1 and no padding, we will get an output of size 2×2.

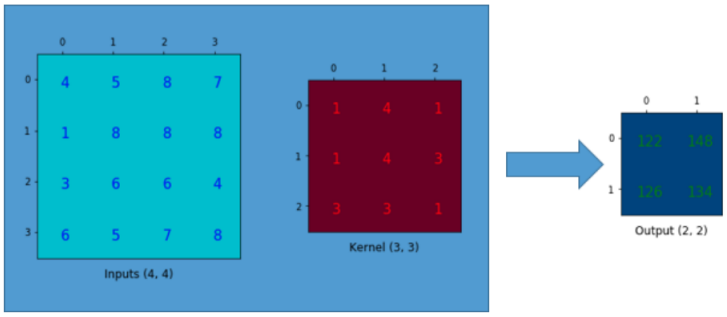

Figure 5: Sample input, kernel and convolution output

We will now rearrange the rows of this kernel into a 4×16 matrix with zeros padded in between as follows. Why do we rearrange it as a 4×16 matrix? Because we have an input with 4 * 4 = 16 elements and the output will have 2 * 2 = 4 elements.

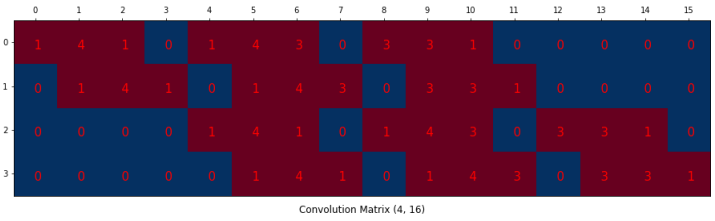

Figure 6: Kernel matrix rearranged as a 4 x 16 matrix

Each row in the above matrix represents one convolution operation. It is just the kernel matrix rearranged with zero padding at different places depending on the stride and padding.

Then, we will flatten the 4×4 input matrix into a 16×1 column matrix.

A simple matrix multiplication between the rearranged kernel matrix and the flattened input in that order will give us a flattened form of the output. This output can then be reshaped to get the output of the convolution.

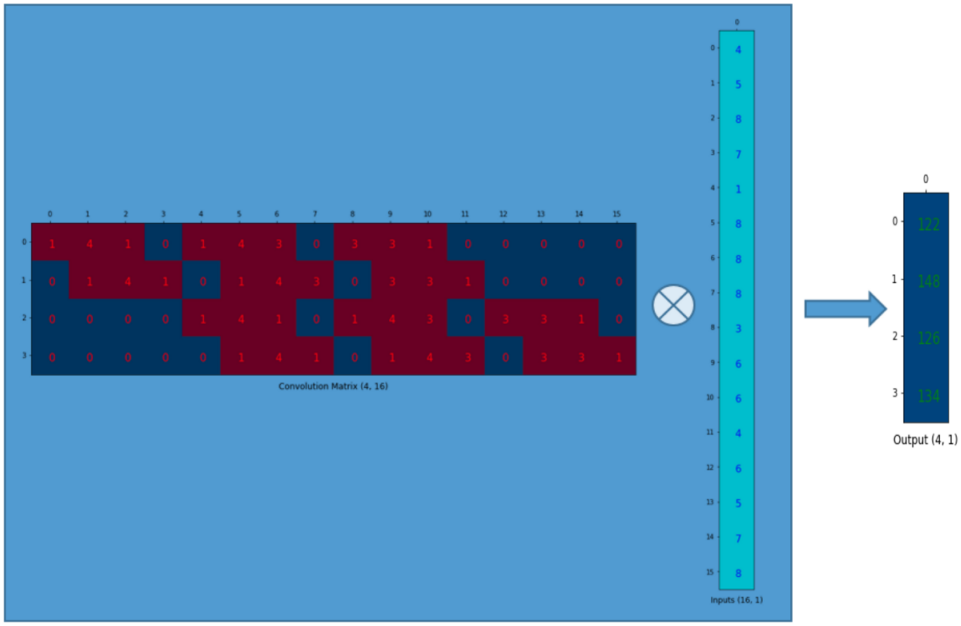

Figure 7: Multiply the rearranged kernel with the flattened input to compute the convolution

The trade off is we might need more space to store the huge matrices. However, as we can see these matrices are mostly sparse and hence should not cause any significant memory issues.

We can see from the equations below how the kernel matrix is rearranged for different stride and padding.

3.6 De-convolution

As we can see, the convolution operation generally reduces the spatial dimensions of the input or down-samples the input. For certain use cases like for Generative Adverserial Networks (GANs) or convolutional autoencoders (more on this later) we need to up-sample input during the forward pass. For example, a 2×2 input needs to be converted to 4×4. We call this operation as de-convolution (aka Transposed Convolution).

This operation needs to maintain the relationship between the input elements and the respective output elements just like the regular convolution operation, just reversed. To do this, we need to define the kernel matrix such that the relationship between respective elements is maintained.

The simplest way to think about it is the consider the rearranged kernel matrix as described in the previous section. We got a 4×16 convolution matrix for converting a 4×4 input to a 2×2 output. Simply taking the transpose of this convolution matrix gives us a 16×4 transposed convolution matrix as seen in figure 8.

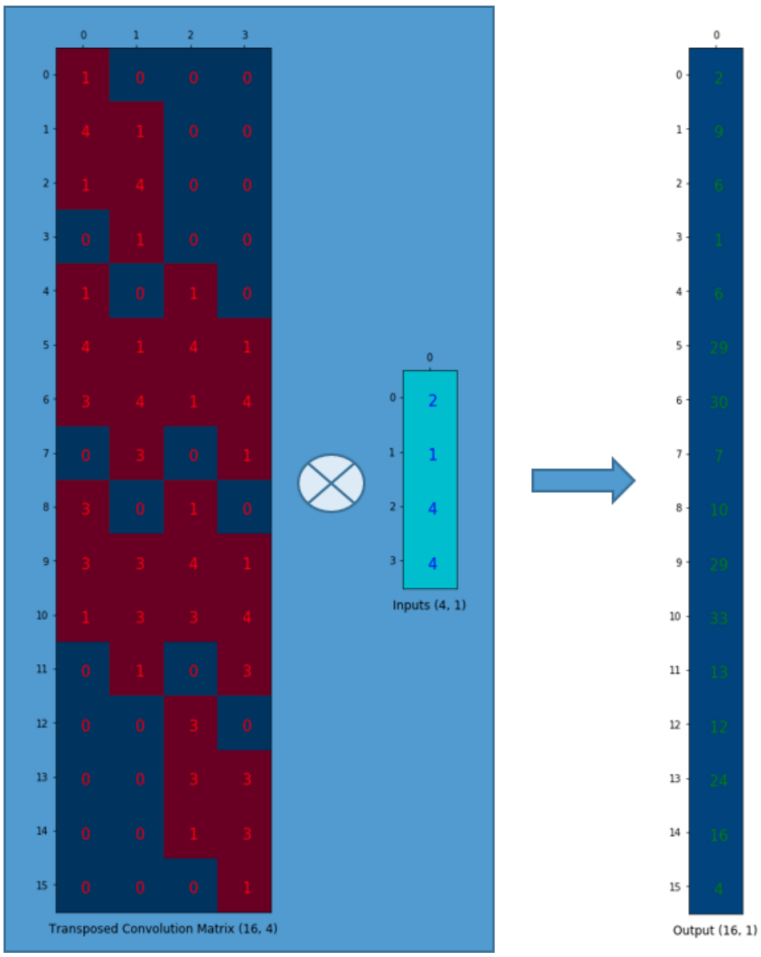

Figure 8: Multiply the transpose of the rearranged kernel with the flattened input to compute the deconvolution

Multiplying this transposed convolution matrix to the flattened 2×2 input, we get a 16×1 output. This output can be reshaped as a 4×4 matrix to get the output of the deconvolution.

4 Pooling

Pooling layers are commonly used between convolutional layers in CNNs. By pooling (summarizing, reducing) multiple inputs, they reduce the layer size. Pooling layers introduce no new parameters, thus they reduce the number of parameters and computation required to run and train the network. The pooling function operates independently for each channel, thus the operation does not reduce the depth of the output layer.

Pooling regions are generally small, either pooling neurons in a 2x2 patch, or 3x3 patch. Typically the max operation (max-pooling) is applied to each region. Like convolutional layers, pooling layers also use padding and stride. It is common to apply a stride of 2 to either the 2x2 or 3x3 pool operations. Zero-padding is often applied. The insensitivity to the specific neuron that is the maximum introduces a form of spatial invariance.

Other pooling operations can be used, for instance average pooling of taking the L2-norm of the patch. Max-pooling is generally found to be better performing, however.

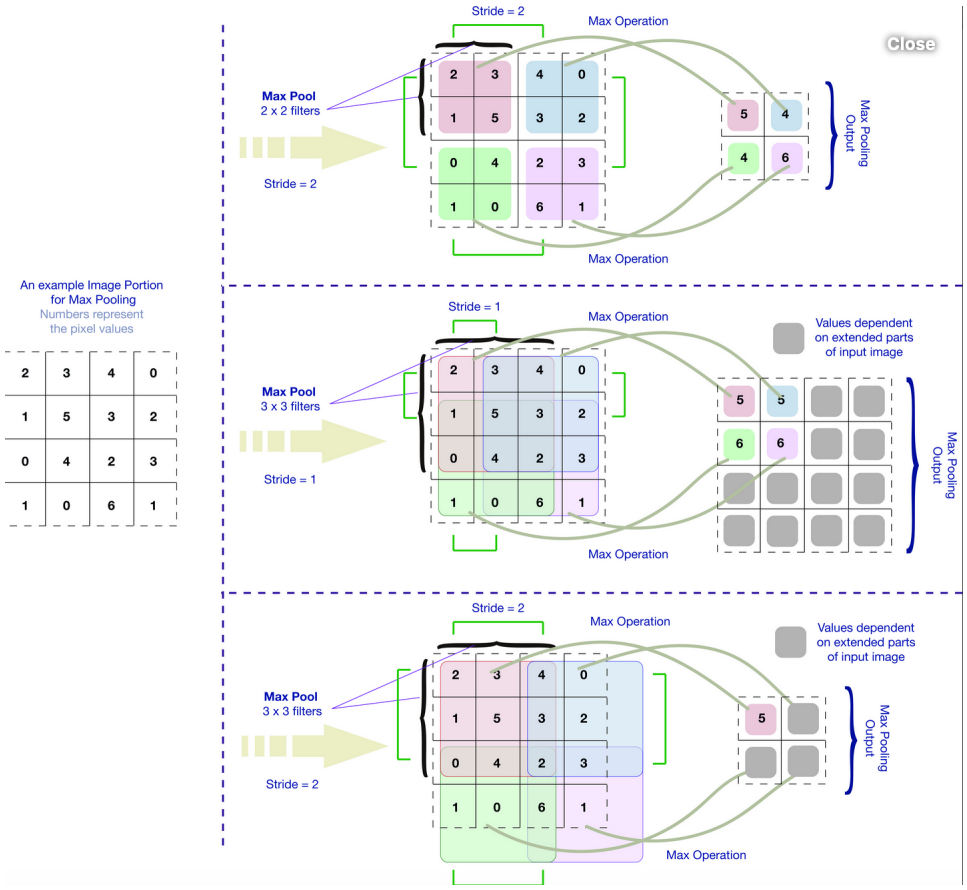

Figure 9: Max pooling for different sizes, strides and padding. Sukrit Shankar 2017.

Backpropagation through a pooling layer is straightforward. For max-pooling, we can think of the max operation as being linear in the element which is the maximum, and 0 for all others. Thus the gradient only flows through this maximum element, with all other elements being zero. That is, if

$$y = \max(x_1, \ldots, x_n)$$

then

$$\frac{\partial y}{\partial x_i} = \begin{cases} 1, & x_i = \max(\boldsymbol{x}) \\ 0, & \text{otherwise} \end{cases}$$

In practice this means in the forward pass, it is useful to keep track of *which* element was the maximum element, so this can be used in the backpropagation step to efficiently compute the backwards pass.

Some network architectures involve both successive reductions of layer size, and increases – downsampling and upsampling. These could be used, for instance, in networks used for image segmentation. The idea is that through convolutional and pooling layers, a low-dimensional representation of the image is produced that segments an image into different categories. This is then upsampled to a representation of the same dimensions of the original image which is segmented.

Unpooling can be used for the opposite purpose as a pooling layer – increase the dimensions of the image without the introduction of additional parameters. There are different choices in how unpooling may be performed. This includes the bed of nails unpooling, nearest-neighbor unpooling, and max-unpooling. Like with backpropagaiton through a max-pooling layer, to implement max-unpooling the index corresponding to the maximum value must be tracked for this to be applied.

Figure 10: Unpool upsamples images. Max unpool tracks which unit was the max in the pooling stage, and propagates the input into this upsampled unit.

5 Inductive Biases: Why use a CNN?

Inductive biases can be seen as characteristics that an algorithm produces in its solutions to a problem. These can be anything – linear regression produces linear functions, L1 regularization produces sparse weights, the support vector machine produces decision boundaries with maximum margin. They can be seen as assumptions underlying or desiderata one specifies when using a particular algorithm. When encountering new data not in the training set, it is the inductive biases that help determine a prediction that particular algorithm will make. In a sense they can also be seen as Bayesian priors, but ones that may be hard-coded into an algorithm.

By sharing weights in a CNN, for instance, we produce models with translation invariance, and by using small kernels we can only detect local features. If these features are present in the task we're solving, then we get sample efficient methods. These inductive biases can help CNNs generalize to new data.

6 Visualizing a CNN

Neural networks are famously 'black-box' models, known for being difficult to interpret. This was for some time a reason against their widespread adoption. Once we train a model, we want insight into why it is making the decisions it is, which features it is using to make categorizations, etc. Neural networks, including CNNs, remain somewhat opaque, relative to, say, linear models. But there are ways of gaining insight into what a CNN is doing and why.

First and most straightforward is to simply examine the filters the model estimates. This is most interpretable in the first layer, where filters extract features direct from the images (pixel-space). This can give an idea how the network is building up representations of textures relevant

for the task. For example, the filters may represent wavelets that extract frequency content from the image. They may act as edge detectors. They may extract low or high frequency content.

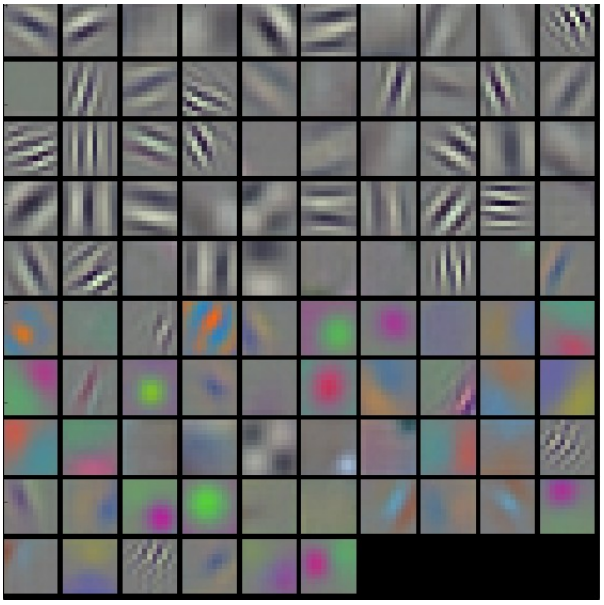

Figure 11: Examining the first layers filters can give an idea of the sort of low-level image processing the network is doing. It is common when using natural image datasets to observe filters that represent wavelets. This allows them to extract spatial frequency and orientation from a region of the image. (http://cs231n.github.io/understanding-cnn/)

A clever trick is to think of trying to identify what neurons, layers, or channels are most responsive to through optimization. That is, say we're interested in which inputs most excite a particular neuron. This would tell us something about its role in the network. We can try to identify this by setting up an optimization to find the inputs that most activate the neuron of interest. Conceptually this is straightforward, though there are some subtleties in getting this to work. You can read about the idea and these issues in this article (https://distill.pub/2017/feature-visualization/). The approach can be used to probe layers, channels, or even to find inputs that generate most confidence in a given image label.

This approach can be a better way of dissecting the predictions of a network than looking at example images that the network labels as a particular class. For instance, say an image category is dog. Looking at a set of images that the network labels dogs it is not clear exactly which features the network used to make this determination. It could have been dog-like features we might expect, and indeed desire from the model, like snout, eyes, ears, etc. However, from these images alone we can't eliminate the possibility that it is other features the network is using to label them dogs. For instance, perhaps the dog-labeled images are mostly in a green field, with a frisbee, or with a leash, and it is these features the network pays attention to. These features are correlated with the dogs in the images, but the network may not be capturing the relationship we want it to. And this could affect generalization to datasets with images of dogs, but no frisbees.

The optimization method is one way of getting around this issue. This idea to is simply optimize the inputs to maximize the probability that the input is labeled a dog. The aim to is see which features it uses to do this. If the resulting input looks like fragments of dogs – snout, ears, etc –

this is a good thing. If it looks like fences, dog houses and dog bowls, the network may struggle to generalize to other dog-related problems.

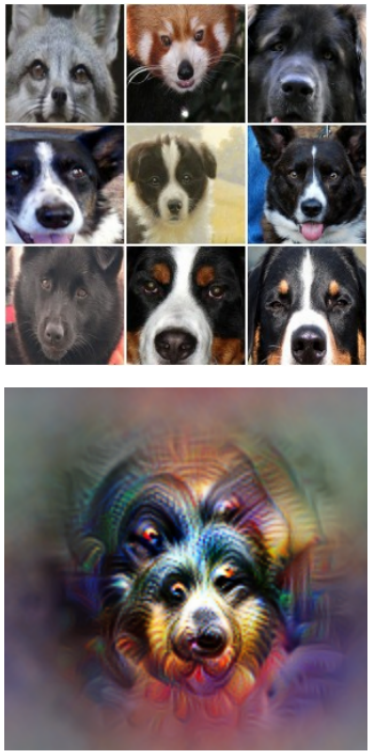

Figure 12: Optimizing input image to generates a high confidence animal face image. (https://distill.pub/2017/feature-visualization/)

Another way to get at this is to try to gain insight into which parts of an image the network is most using to make its determination. A way of doing this is to occlude the image in different regions. When the important regions are occluded, this should have a large effect on the classification label. This can be overlaid onto the original image, as a heatmap, which shows which areas the network used to predict. (See figure below).

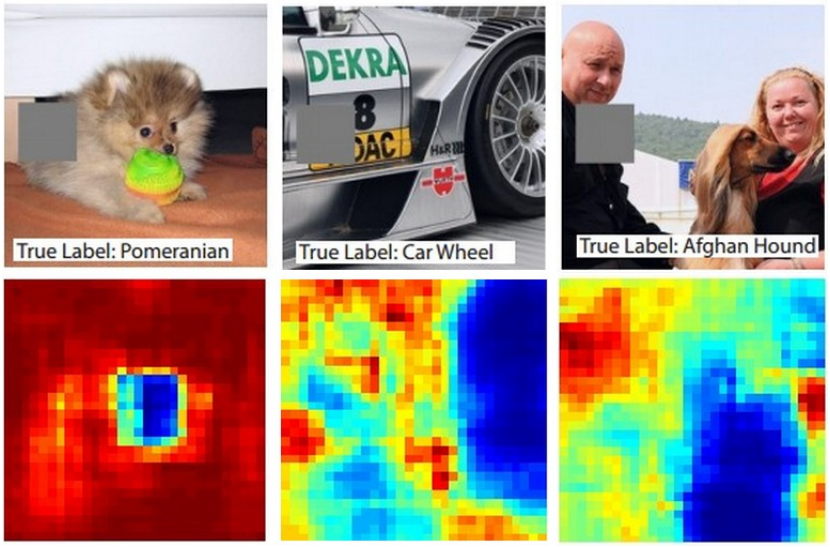

Figure 13: A probability map of an image being labeled when part of the image is occluded. Here the occlusion is the gray square. When the relevant parts of the image are blocked, this should have a large effect on the classification. Hence this serves as a sort of attention map. (http://cs231n.github.io/understanding-cnn/)

7 References

- http://cs231n.github.io/convolutional-networks/
- https://becominghuman.ai/back-propagation-in-convolutional-neural-networks-intuition-and-code-714ef1c38199
- https://medium.com/activating-robotic-minds/up-sampling-with-transposed-convolution-9ae4f2df52d0
- https://www.researchgate.net/figure/Structure-of-the-convolutional-neural-network_fig3_323227084