Vision Models and CNNs

Shaun Ho

February 2024

1 Shift-invariant Pattern Classification

1.1 Scanning MLPs

Where the expected patterns are expected to be composed of similar structures, differing only by their precise location in the input space, multi-layer perceptrons can classify them with a scanning algorithm.

This is done by segmenting each input into windows (which may overlap) and feeding all of the windows into an MLP. The resulting outputs, produced in sequence of the windows as they are passed into the MLP, can then be fed into a softmax layer.

This scanning of each input window can be done by using a giant, shared-parameter network with common subnets, with one subnetwork assigned to each window.

1.2 Scanning MLPs vs Regular MLPs

Scanning MLPs differ from regular MLPs because each neuron in a scanning MLP is connected only to a subset of the previous layer's neurons, unlike a fully-connected MLP. Also, due to the use of shared parameters across the various subnets, the weights matrix is sparse and not full like it would be in a conventional MLP, and it is structured with non-unique blocks.

1.3 Distributing Scanning Behavior Between Layers for Hierarchical Feature Learning

It must first be noted that instead of having a large, shared-parameter network of many subnets, it is equivalently possible for each window of the input to be passed sequentially into a single network if all of the inputs to layer L are processed before layer $L+1$ begins its operations. This reordering of the operations results in the final output sequence being the same. The key point is that even when processing all inputs to a layer sequentially before proceeding to the next, the network can still maintain the input's spatial structure.

Given this reordering of the operations (where processing is done by each layer and not by each input), the full sequence of the previous layer's outputs is fully computed before being passed through to the the subsequent layer. When we do this, the window-scanning behavior that we used in the first layer (for scanning the input data) can be implemented for subsequent layers, which thus would scan windows of the outputs of the previous layers. This "distributes" the scanning behavior across more layers throughout the network (instead of just confining it to the first layer). The scanning of output maps using fixed window sizes is called a **convolution**.

Doing this, we obtain the following benefits:

1.3.1 Better performance by learning a hierarchical build-up of features over the scanning window

The ability of MLPs to universally approximate functions is traced to the behavior of individual neurons, each representing a decision boundary in the feature space, aggregating to build up an arbitrarily complex pattern. Each neuron in a higher layer of an MLP, being a function aggregator learns **combinations** of patterns that are detected by lower layers and represented in the outputs of those lower layers. We also know that deepening an MLP (as opposed to widening it) is more data and parameter efficient.

Convolutions are valuable because the use of scanning across various layers forces the network to adopt this desirable hierarchical aggregation of representations using the localized patterns in lower layers. This makes the network more generalizable.

1.3.2 Fewer parameters

The use of scanning convolutions means that higher layers are aggregating the outputs of lower layers. As such, to cover the same receptive field or input space, neurons in lower layers can concentrate on smaller, more manageable windows.

When outputs of lower layers are aggregated before being processed by the higher layers, this expands the effective receptive field processed by that higher layer (since the total input area being processed is being summed up for each output that is aggregated). As a result, the total breadth of input that neurons in the lower layers are required to process to capture the same size of receptive field at the higher layer becomes smaller (than if the higher layer was not aggregating their outputs). As a result, there are far fewer connections in the lower layer since it does not have to connect to every single input in the span of the receptive field, only a subset of it.

As an illustration, consider a three-layer network with N_1 neurons in the first layer, N_2 neurons in the second layer, and N_3 neurons in the third one. If the second layer convolves 4 outputs of the first layer, then if the first layer scans an input size of 2, the receptive field of the second layer with respect to the input is effectively 8. For a similar three-layer network to achieve an effective receptive field of 8 in the second layer without convolving the outputs of the first layer, each neuron in the first layer will be expected to cover the entire receptive field width of 8 (as opposed to 2), which increases connections between the input and the first layer by a factor of 4. Moreover, in the distributed case, N_1 features are generated from 2 inputs which requires that $4N_1$ features be generated from $(4 \times 2 = 8)$ inputs in the non-distributed case, implying a requirement of 4 times more neurons in the first layer in addition to each of those neurons having to be fully connected to 8 inputs as opposed to 2.

In sum, a distributed scanning architecture requires fewer parameters to achieve the same level of representation as a non-distributed architecture.

1.3.3 Computational efficiencies

When windows overlap, then at each position of the window, higher level neurons are capturing within their effective receptive fields some outputs from the previous layers that were also present in the adjacent window positions previously scanned. As such, they are reusing the computations performed at the previous steps.

1.4 Pooling

The work of Hubel and Wiesel (1959, 1962) indicates that the mammalian model of vision relies on the activation of striate cortex neurons, the most effective stimuli for which are oriented slits of light. Within the striate cortex, two levels of processing were identified: First, simple S-cells, which fire in response to stimuli but are susceptible to noise and deformations, and complex C-cells which aggregate and "clean up" the outputs of a bank of S-cells to achieve an oriented response that is robust to distortion. Fukushima (1980) made the model robust to variations in the position of the stimulus in the input space by modeling S-cells and C-cells as modules, each containing planes of either S or C-cells with each adjacent cell in these planes taking a correspondingly shifted region of inputs from the previous plane. The C-cells in each plane had an identical fixed response and each S-plane was followed by a C-plane.

This model arguably forms the basis for the CNN architecture, which relies on distributed scanning of mapped areas of the previous layer's outputs followed by pooling layers that aggregate windows of the activation channels.

1.5 Transform Invariance

The model can be modified to include invariance to rotations in the pattern by producing an enumerated and finite set of transformed maps for each input map, each reflecting a discrete set of rotations, scalings, reflections, and other transformations, although this is computationally expensive and thus generally addressed by way of data augmentation. These are some possible augmentations:

1. Rotation $\in [0°, +360°]$
2. Translation by $[-10, +10]$ pixels
3. Random rescaling with log-uniform scale factor of $[\frac{1}{1.6}, 1.6]$
4. Flipping $[1, 0]$
5. Shearing $\in [-20°, +20°]$
6. Random stretching with log-uniform stretch factor of $[\frac{1}{1.3}, 1.3]$

1.6 Variations of Model Outputs and Multi-Task Learning

The general conception of the convolutional model feeds the final, flattened output of the convolutional layer into a single output which predicts the class of the output.

This can easily be adapted to feed the flattened output into a second layer that adds another task such as identifying the bounding box of the pattern within the original input (subject to the availability of such labels in the training data).

1.7 Depthwise Convolutions

Depthwise convolutions serve as a parameter-light alternative to pointwise convolutions. They differ from the pointwise case in that the convolution step is performed only once, as opposed to N times for each of the N channels in the output.

In a pointwise convolution, each filter is three-dimensional with M layers of size $(K \times K)$, and there are N of them - one for each output channel.

In a depthwise convolution, each filter is assigned to one input channel, thus resulting in M two-dimensional $(K \times K)$ filters, which are then used to produce an intermediate output which has the same number of channels as the input. Then, to match the number of output channels, N one-dimensional filters are applied to the M channels to produce the output channels.

The result of the depthwise convolution is that each of the output channels is created by applying a weighted sum across the intermediate convolved output instead of being created by its own unique set of convolutions.

1.8 Receptive Fields

The receptive field of a neuron refers to the specific pattern it has learned to search for (in the input image). The shape of the receptive field is easy to compute in the first layer by observing its arrangement of weights but due to the hierarchical consolidation and combination of features in higher layers, calculating the receptive fields of neurons in those higher layers is not immediately obvious and must be calculated by setting the output of the neuron to 1 and learning the input by backpropagation.

Empirically, the shapes of the receptive fields learned by the neurons in the first layer are usually remarkably similar in appearance to those oriented slits of light that Hubel and Wiesel identified as being active in activating S-cells in the mammalian cortex.

2 CNN Operations

2.1 Convolution

Each channel in layer L has an associated filter (which is effectively a perceptron with weights and a bias) which is used to compute its affine map. At each location, the filter and underlying map values are multiplied componentwise and a bias is broadcasted. In each filter is one set of weights for every input channel, with each pixel in the output map being computed from the aggregated application of that set of weights to each input channel.

2.1.1 Output Sizes

The size of the convolution depends on the size of the input and the size of the filter. For an input map of size $N \times N$ and a filter size of $M \times M$ the output size will be $(N - M) + 1$ on each side. This results in a reduction in the output size compared to that of the input, which is sometimes considered unacceptable.

2.1.2 Padding

Convolving a map might result in information loss if the windows are confined within the boundaries of the input dimensions. For filters of with L:

1. Odd L: Pad with $\frac{L-1}{2}$ on all sides.
2. Even L: Pad with $\frac{L}{2}$ on one column/row and with $\frac{L}{2} - 1$ on the other.

With the dimensionality of the resulting image being $(N + L - 1)$ regardless of whether L is odd or even, the convolved output would thus be $(N \times N)$.

Padding is avoided in the case of pooling.

2.2 Pooling

As described above, this is done to make the model robust to jitter and deformations in the input.

The pooling operations typically come in the form of max or mean-pooling but many alternatives are available, including learned filters such as MLPs.

2.3 Striding

Defining striding in terms of downsampling and upsampling with respect to its preceding and following layers reduces complexity when computing gradients during backpropagation.

2.3.1 Downsampling

A **convolution** with stride S can be expressed as a full convolution followed by downsampling.

Pooling with stride S can be expressed as a full pooling followed by downsampling.

A stride of S drops $S - 1$ of every S rows and columns from the map, effectively downsampling by a factor of S in every direction.

This sort of striding is usually done to save computation, but this might result in information loss because striding shrinks along each axis by a factor equal to the stride size.

Convolving an input of size $(N \times N)$ with filters of size $(M \times M)$ and a stride of S, the size of each output map shrinks to $\lfloor \frac{N-M}{S} \rfloor + 1$ on each side.

Pooling an input of $(N \times N)$ with a pooling filter of size $(M \times M)$ and stride S, the size of each output map is $\lceil \frac{N-M}{S} \rceil + 1$.

To prevent information loss over this stride of S, the number of output maps must be S^2 times the number of input maps.

2.3.2 Upsampling

Convolving with a fractional stride of $\frac{1}{S}$ is equivalent to upsampling by a factor of S before convolving.

This introduces $(S - 1)$ rows and columns of zeros for every map in the layer.

Upsampling is typically not done before pooling or before the final convolution layer.

3 Learning a CNN

The parameters of the network can be learned through backpropagation. In the general case, conventional backpropagation can be done through the final flat layers which are organized as a regular MLP. Then, the first flat layer is folded back to each channel of the output of the final CNN layer $Y(l)$

The components we need to compute for **convolutional layers** are:

1. The derivative with respect to the affine map $Z(l)$ from the activation map $Y(l)$
2. The derivative with respect to the previous layer $Y(l-1)$ from $Z(l)$
3. The derivative with respect to the weights $w(l)$ from $Z(l)$

And for **pooling layers**:

1. The derivative with respect to the activation map of the previous layer $Y(l-1)$ from the pooling layer $Y(l)$.

3.1 A note on shapes

The derivative with respect to any variable must always be of the same shape as the variable itself.

3.2 Convolutional layers

3.2.1 The affine map

Since the activation map is obtained by a simple pointwise application of a known activation function $f(z)$ to the affine maps, then we have:

$$\frac{dDiv}{dZ(l,m,x,y)} = \frac{dDiv}{dY(l,m,x,y)} f'(z(l,m,x,y)$$

Which computes the derivative for each pixel (x,y) of the m-th affine map of the l-th layer with respect to the loss.

3.2.2 The previous layer's outputs

We wish to compute the derivatives of each individual pixel in each output channel of the previous layer, which is the derivative with respect to $Y(l-1,m,x,y)$.

We know that the m-th channel of the previous layer influences all of the channels in the current layer's affine map through the m-th channel of each of the current layer's n filters.

Looking in the opposite direction, each n-th channel of the current layer's affine map $Z(l, n)$ is influenced by all of the channels m in the previous layer through every plane m in the n-th filter.

We look at the (x, y)-th pixel from the m-th channel of the previous layer $(l-1)$. To compute the influence this pixel holds over the values in all channels of $Z(l)$, we must then **sum** its influence over every n-th channel in $Z(l)$.

As such:

$$\nabla_{Y(l-1,m)} Div = \sum_n \nabla_{Z(l,n)} Div \nabla_{Y(l-1,m)} Z(l, n) \tag{1}$$

Which simply translates to: The derivative with respect to each channel in the previous layer is computed by summing that channel's influence over all of the n affine maps in the current layer.

Now, how do we compute $\nabla_{Y(l-1,m)} Z(l, n)$?

We recall that each input pixel (x, y) is, depending on the parameters of the convolution, used to produce one or more corresponding pixels (x', y') in the affine map by applying an inner product with the relevant filter.

As such, to compute the derivative with respect to each each input pixel (x, y), we have to sum its influence over all pixels (x', y') in all of the n affine maps. This gives us the following:

$$\frac{d\text{Div}}{dY(l-1, m, x, y)} = \sum_n \sum_{x',y'} \frac{d\text{Div}}{dZ(l, n, x', y')} \frac{dZ(l, n, x', y')}{dY(l-1, m, x, y)} \tag{2}$$

How do we determine how much a single (x, y) pixel in $Y(l-1, m)$ affects any given (x', y') in $Z(l, n)$?

We know that the m-th channel of $Y(l-1)$ "connects" to the n-th affine map in $Z(l)$ through the m-th channel of the n-th filter. We denote this filter with the notation $w_l(m, n)$.

We also know that the filter $w_l(m, n)$ is itself a grid comprising its own individual pixels. So as we slide the filter over the plane of m, each input pixel (x, y) on that plane is being repeatedly applied to a different, adjacent pixel on the filter grid.

(Looking from the other direction, each pixel (x', y') in the affine map is influenced by the sum of all (x, y) pixels in the input map that are covered by the filter grid at that position, specifically it is a sum of the pointwise operations of the grid $w_l(m, n)$ on plane m.)

Returning to each individual (x, y), in the context of convolution it can be shown that because of the way we are sliding the filter over the m-th plane from left to right and top to bottom, then after sliding each pixel of the filter over (x, y), the map of pixel (x, y)'s contribution to the affine map is simply a transpose and flip of the filter $w_l(m, n)$.

Expanding on this, since we also know that the m-th channel in $Y(l-1)$ convolves the m-th plane of each filter, then the summing of each (x, y)'s contribution to $Z(l)$ can be expressed as a convolution of $\frac{\partial Div}{\partial Z(l,n,x,y)}$ maps by each corresponding transposed and flipped filter (with the additional caveat that the derivative maps of Z are zero-padded with $K-1$ zeros on each side for a kernel size K).

Expressing this in notation:

$$Z(l, n, x', y') \pm = Y(l-1, m, x, y) w_l(m, n, x-x', y-y') \tag{3}$$

$$\frac{dZ(l, n, x', y')}{dY(l-1, m, x, y)} = w_l(m, n, x-x', y-y')$$

Plugging the result back into equation (2):

$$\frac{d\text{Div}}{dY(l-1, m, x, y)} = \sum_n \sum_{x', y'} \frac{d\text{Div}}{dZ(l, n, x', y')} w_l(m, n, x-x', y-y')$$

Which ultimately expresses the following propositions:

1. The derivative of the divergence with respect to pixel (x, y) in the m-th channel of the previous layer, is its contribution to every pixel (x', y') in every affine map, which can be measured in relation to the transpose of each corresponding filter map.
2. Computing the derivative for $Y(l-1, m)$ is a convolution of the zero-padded derivative map with respect to $Z(l, n)$ by the set of corresponding transposed and flipped filters (after zero-padding to control for the size of the filter).

Thus, the algorithm for computing the derivative for $Y(l-1)$ is as follows:

1. Zero-pad the derivative map of Z
2. For each channel-filter pair between $Y(l-1)$ and $Z(l)$, transpose and flip each filter
3. For each possible pixel in each channel of $Y(l-1)$, identify the relevant segment within the (padded) derivative map of Z, and apply a tensor inner product of the flipped filter to that segment to obtain the value of the derivative.

3.2.3 The weights

We aim to compute the total contribution of a single pixel (i, j) in a single weight, $w_l(m, n)$ to the entire affine layer $Z(l)$. We know that the influence of $w_l(m, n)$ is limited to the n-th affine map $Z(l, n, x', y')$ since each filter corresponds to exactly one affine map.

We also know that the derivative of the weights is computed using the values of the inputs $Y(l-1)$. Applying the logic of convolution, the total contribution of any (i, j) pixel of the weight is the sum of its operations as the entire weight matrix slides over the input map $Y(l-1, m)$. As such, we obtain:

$$\frac{d\text{Div}}{dw_l(m, n, i, j)} = \sum_{x,y} \frac{d\text{Div}}{dz(l, n, x', y')} Y(l-1, m, x'+i, y'+j)$$

Which stipulates that the derivative of the n-th affine map convolves with each channel m of the output to obtain the derivative of the associated weight. Specifically, $w_l(m, n, i, j)$ - the derivative of the (i, j) pixel of weight (m, n) - is computed by "placing" the derivative map of channel n on the m-th map of the previous layer at position (i, j) and computing the inner product.

The algorithm for computing the weights is as follows:

1. For each weight, then for each (i, j) pixel in the weight, we take the dot product of the derivative map of channel n in $Z(l)$ with the map of the channel m in $Y(l-1)$ bounded by kernel size K and starting coordinate (i, j).

3.3 Pooling layers

3.3.1 Max pooling

Only the maximum element in any given input window is relevant when computing the derivative of the max pooling layer. This is because the operation passes only that maximum element to the next layer, with the implication that the derivatives of all other elements in the window with respect to the final loss becomes zero.

As such, the objective is to keep track of the index at which the maximum value in the window is located. In the forward pass:

1. For each channel in the max pooling layer,
2. Save the index of the largest element in any given window (x, y)
3. The output of the max pooling layer at (x, y) is that single element within that window.

In the backward pass:

1. Initialize the gradient of the pooling layer with zeros with each channel having the same dimensions as those of the input channels.

   For each channel in the input and each of its corresponding windows as indexed by coordinate (x, y):

2. Take the derivative of the pixel (x, y) in the output channel of the max pooling layer. This would have been computed with respect to the maximum element in the corresponding window during the forward pass.

3. Retrieve the index position of that element as it appeared in that window during the forward pass.

4. Using the position of the window within the input channel and the position of the index of that largest element within that window, map the derivative value to the index position of that element within the entire input channel.

3.3.2 Mean pooling

Working forward, we see that each pixel in the pooled output is the sum of the contributions of all elements in the corresponding window used to compute that output pixel, with the caveat that each contribution is scaled down by a factor of K^2 (the number of elements in the window).

Thus, working backward, where we have a given derivative for an output pixel, we distribute this derivative equally to each element in the corresponding window used to compute that output pixel.

Unlike the case in max pooling there is no need to save specific index positions during mean pooling since every pixel of the input is being used.

In the forward pass:

1. Over each input channel and each window of kernel size K,

2. Take the mean of all elements within the kernel and assign the value to the (x, y)-th pixel of the output.

In the backward pass:

1. Initialize the gradient of the pooling layer with zeroes for each channel, corresponding to the dimensionality of the input channels.

   For each channel in the input and each of its corresponding windows as indexed by coordinate (x, y):,

2. Take the derivative of the output pixel that was computed from all elements in that window and divide it by the number of elements in the window.

3. Assign that distributed derivative to each element in that specific window of the input.

3.4 Resampling and Striding

3.4.1 Downsampling

Downsampling by a factor of S involves the computation of a full map followed by the dropping of $S-1$ of every S rows and columns in the map.

When backpropagating convolution or pooling with a positive stride, it is useful to view it as a full convolution/pooling, followed by downsampling.

This implies that the only elements in the map that ultimately contributed to the loss were those that were not dropped during downsampling.

The backward pass aims to distribute the gradients to the positions of the elements which were not dropped from the original map.

1. Initialize a map of the size of the input that was downsampled.
2. Assign a value of 0 to all elements in the map that belonged to columns and rows that were dropped during downsampling.
3. Distribute the derivatives of each element in the downsampled map to the corresponding positions in the input map by copying them over to the appropriate locations. This is done by using list comprehension in a way that adds a column/row of zeros for every $S-1$ rows and columns copied.

3.4.2 Upsampling

Upsampling by a factor of S introduces $S-1$ rows and columns into the map. There exist within the upsampled map rows and columns that were added during upsampling, which as such are not considered functions of the input.

When backpropagating convolution with a fractional stride, it is useful to view it as upsampling followed by a full convolution.

During the backward pass, we copy over to the input map the derivatives of elements which do not exist within the upsampled rows and columns. We do this by using list comprehension in a way that skips every $S-1$th row and column in the copying operation.

4 References

Hubel, D. H., & Wiesel, T. N. (1959). Receptive fields of single neurones in the cat's striate cortex. The Journal of Physiology, 148(3), 574–591.

Hubel, D. H., & Wiesel, T. N. (1962). Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. The Journal of Physiology, 160(1), 106–154.

Raj, B. & Singh, R. (2024). Intro to Deep Learning (Lecture Notes).