Explanatory Notes for 6.390

Shaunticlair Ruiz (Current TA)

Spring 2023

Contents

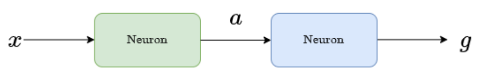
CHAPTER 9

Convolutional Neural Networks

9.0.1 Fully Connected Networks

Up to this point, we've focus on "**fully connected**" neural networks.

- "Connected" refers to the "connection" between neurons in **adjacent** layers: one neuron provides the input for another.



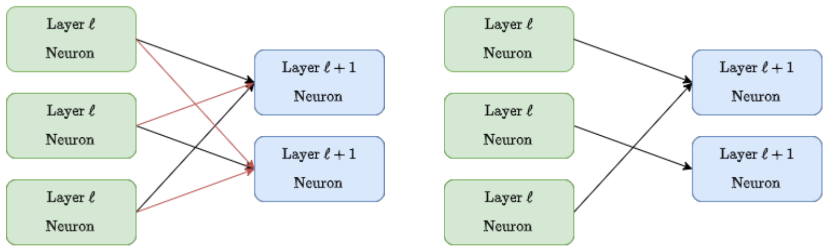
These two neurons are connected.

Thus, "fully connected" means that every possible connection between pairs of neurons **exists**.

Definition 1
A **fully connected** (FC) layer is one where every **input** neuron is connected to every **output** neuron.

The network layer only needs to be missing **one** connection between neurons to not be fully connected.

Example: We compare two networks:



The left network is fully connected: the right is not, having removed the red arrows.

9.0.2 The drawbacks of fully-connected networks

The "fully connected" approach includes a **weight** $w_{a,b}$ for every pair of input neuron a, and output neuron b.

- Each of these weights determines the **relationship** between our two neurons.
- In an FC settings, we're allowing for every possible pattern between pairs.

This is a very, very flexible model: any combination of patterns is possible.

Concept 2

Fully-connected networks are very useful when we **know very little** about how to **predict** our result.

By including so many possible connections and patterns, we're open to lots of **different models** we could try.

- This is especially helpful if we expect these relationship to be complex.

With a non-FC model, on the other hand, some connections have been severed. With this model, we're creating making some **assumptions** about which patterns **don't** exist.

- **Example:** If you think fact A is irrelevant for computing fact B, you wouldn't include it in the equation.
- This is similar to how you want to exclude inputs that won't help you predict your output.

Concept 3

Removing a connection in a neural network is equivalent to saying, "I don't think this variable a **should** affect this other variable b".

This highlights a major **drawback** of fully connected networks: sometimes, it's inappropriate to allow for every possible connection.

Having connections we don't need can cause plenty of problems:

Concept 4
Fully-connected networks come with some problems:

- Having many parameters can risk **overfitting**,
- Our model takes **more time** to converge
  - Both because it has to train **more weights**,
  - And because our model can get "distracted" by dead-end possibilities, that a simpler model wouldn't consider.
- It's often difficult to interpret **how** our neural network comes to the conclusions it does.

In this chapter, we'll introduce some more specific problems, and one model type that allows us to overcome these problems: **Convolutional Neural Networks**.

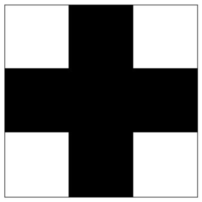
9.0.3 Intro to Image Processing

An excellent example for how FC neural networks can fail, is **image processing**.

Example: Facial recognition, self-driving vehicles, classifying the object in a picture

Let's give it a try: suppose we have an image. For simplicity, it's black and white. We'll need to represent the **brightness** of each pixel with a number.

- We'll use the most common range of values: the integers $[0, 255]$.



$$\Longrightarrow \begin{bmatrix} 255 & 0 & 255 \\ 0 & 0 & 0 \\ 255 & 0 & 255 \end{bmatrix}$$

Our machine stores the "picture" on the right.

Our neural network takes a single $d \times 1$ vector as it's input. But right now, we have an $(r \times k)$ matrix. How do we solve that? With **flattening**.

Definition 5

Flattening is the process of taking a **matrix** of inputs, and transforming it into a single **vector**.

We usually do this by **concatenating** (combining consecutively) each row/column, in order.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \longrightarrow \begin{bmatrix} a \\ c \\ b \\ d \end{bmatrix}$$

If the input is an $(r \times k)$ matrix, the result is an $(rk \times 1)$ vector.

Example: We can apply this to our data above.

We'll represent it with the transpose to save space.

$$\begin{bmatrix} 255 & 0 & 255 \\ 0 & 0 & 0 \\ 255 & 0 & 255 \end{bmatrix} \longrightarrow \begin{bmatrix} 255 & 0 & 255 & 0 & 0 & 0 & 255 & 0 & 255 \end{bmatrix}^T \tag{9.1}$$

And so transform from (3×3) to (9×1).

Looking at the image, or even the **matrix**, it's relatively easy to see the "**cross**" pattern.

But, when we **flatten** it, those patterns immediately stop being obvious.

- We've lost information above which pixels are "**beside**" one another, for example.

Remember that we only took the row vector for visualization: they're stacked vertically based on column!

Based on this, we'll find **two** main problems, that our CNNs will hopefully solve:

9.0.4 Spatial Locality

First: as we just mentioned, we need an idea of which pixels are close **horizontally**.

- In fact, our network doesn't even care which pixels are **above** each other **vertically**:

Concept 6

Review from the Feature Representation chapter

The **order** we choose for elements in a vector **doesn't** affect the behavior of our model, so long as we **consistently** use that order.

This is because a linear model is a **sum**:

$$w^T a = \sum_i w_i a_i$$

And sums are the same, regardless of **order**.

$$a + b = b + a \qquad \Longrightarrow \qquad w_1 a_1 + w_2 a_2 = w_2 a_2 + w_1 a_1$$

- We emphasize that the order does need to be **consistent** between data points.

Example: Suppose x_1 represents height and x_2 represents weight.

- We could do either [weight, height] or [height, weight]: it doesn't matter.

BUT, we have to be **consistent** with which order we pick: otherwise, someone is measured as 180 feet tall, instead of 180 pounds.

In other words, we could **shuffle** the order of our pixels, and as long as we shuffled it the **same way** for all of our training data, it wouldn't matter to our **model**.

- This was fine for the above example, but it doesn't make sense for **image processing**, where each dimension is just a **pixel**:

Human vision works differently: we look for shapes, which are often made up of pixels **near** each other: edges, points, corners, curves.

- In other words, we want to encode **local** information, across the physical **space** of our image. Thus, we call it **spatial locality**.

When we use "space" here, it's **different** from "latent space", or "input space".

Instead, we're talking about physical space: how physically **close** real things are, measured in **meters**, or in this case, **pixels**.

Definition 7

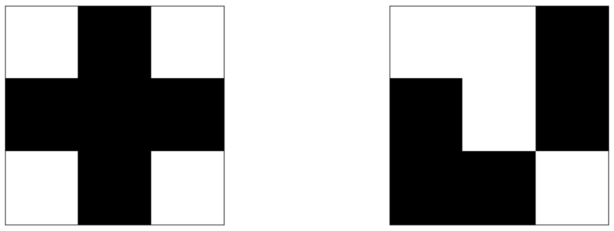
Spatial locality is the knowledge of which objects are **close** in **space** to each other.

In an **image**, we might think of which pixels are "close" in that image.

- Which pixels are next to, or on top of each other? How far apart? How are they "arranged"?

- This is a property we want to build into the structure of our CNNs.

Example: If told that the white was blank space, and the black represented "objects", a human would have a concrete understanding of how these two images might be different:



The left image contains **one** object, while the right image contains **two** objects.

We figure this out based on which pixels are **touching** or not: a spatial property.

- The neural network would struggle to encode anything like that.

9.0.5 Translation Invariance

A second problem is that, if the same **pattern** occupies different pixels, then it's completely new to the model.

- **Example:** Suppose you have a cat on the left side of an image. You **move** it to the right side of the image.
- A person would consider that image "**almost the same**".
- But our FC NN does not: the cat is occupying a completely **different set of pixels**, which have a completely separate set of weights attached.

So, our NN can't find structures that are **similar** across different parts of the input.

Instead, we want a different behavior: we want our model to treat our input as the **same** (invariant), even if we move, or **translate** it.

Not language translation: "translation" as in "moving around in space".

- Thus, we're looking for **translation invariance**.

Definition 8

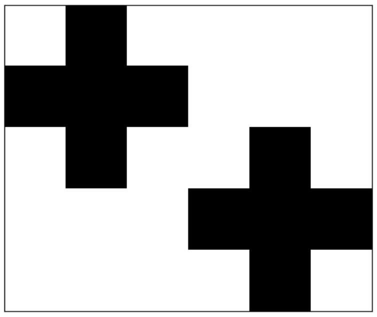
Translation invariance is the property of treating patterns as the **same** even if we **translate** them in space.

In an **image**, we might want to recognize the same **pattern** in two different **positions** on the image.

- In other words, the pattern has "translated" from one of those positions, to the other.

- This is a property we want to build into our CNN.

Example: In the following image, you would probably recognize "two crosses".



We have two of the **same** object: just **translated** over.

But, because the top left pixels have separate weights from the bottom right pixels, the NN will react differently to each.

Now that we've defined our problem, we can come up with a solution: **filters**.

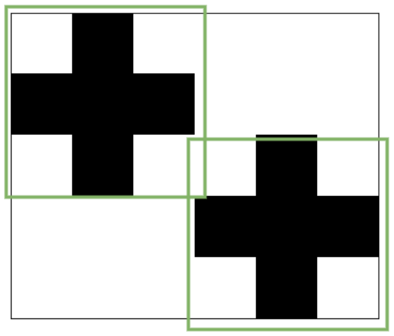
9.1 Filters

9.1.1 Motivating the Filter

So, we want a technique that handles both of these problems.

First, **translation invariance**: we want a calculation that can find the same pattern, in multiple locations.

- So, we'll apply the same calculation repeatedly, in **multiple positions** on our image.
- We'll **move** across our image, shifting to a new position each time we **scan** for that pattern.



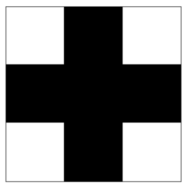
As we "scan over" our image, we'll hopefully find both of our crosses separately.

Next, **spatial locality**: we want this calculation to encode **spatial** information.

- As we "scan" across our image, each computation will look for a particular "shape", or "**pattern**" for our pixels.
- This pattern will be based on the **relative location** of each pixel.

So, we're looking for a tool that repeatedly shifts (or **translates**) across our image, and looks for a spatial **pattern** in the image.

- **Example:** Above, we would be looking for the "3x3 cross" pattern, and shift across rows/columns.



This is the shape we are looking for at each position.

The tool in question is "looking" for a pattern. Another way to see it, is that it's **filtering** out everything that doesn't match that pattern.

- Thus, we call it a **filter**.

Concept 9

Filters handle both the problems of **spatial locality** and **translation invariance** at the same time.

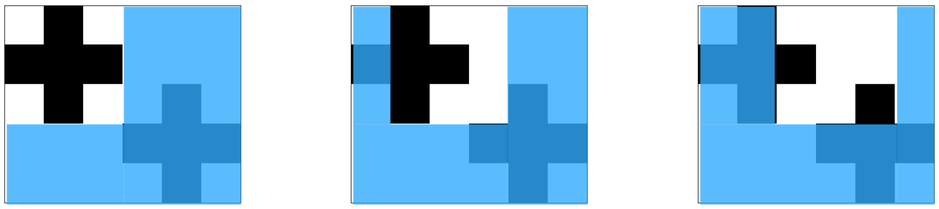
Notably, all of this works better if we keep our data in the **matrix** format, not the **flattened** one.

9.1.2 Windowing

We still need to figure out **how** we're going to find these patterns.

We've already established that our algorithm will look at a **local** region of the image, and search for the pattern.

To make life easier, we'll cut out a **piece** of the image, and only compare that to the pattern.



If we're viewing the top-left corner, we're ignoring everything else (blue-shaded). Then, we'll check the next position.

This region is the only part we "see", so we call it a **window**.

Definition 10

The **window** v is the region of our image that we are, at a given moment, looking for our **pattern** in.

This is the region we are applying our **filter** to.

The window has the **same dimensions** as our filter, so we can compare them directly.

- As we continue filtering, we'll repeatedly move our filter, shifting it to every valid position on the image.

9.1.3 1-D case

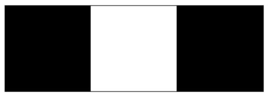
To get going, we'll start with a 1D example.

This isn't just a simplification: when processing sound data, it'll be in a 1D form.

- To make the math easier, we'll replace 0 and 255 with $+1$ and -1.

We've decided to make dark pixels +1, and bright pixels -1. Which convention we choose isn't important: it's just more easily visible.

Suppose we're looking for "bright spots": pixels that are much brighter than their surroundings.



$$\Longrightarrow \qquad \begin{bmatrix} +1 & -1 & +1 \end{bmatrix}$$

So, we're looking for something like this.

How do we find "bright spots", like this? Well, we want to find regions which are **similar** to our pattern.

- Our sequence is a vector, so we want to **get the similarity between two vectors**.
- We have a tool for this! The **dot product** $a \cdot b$.

Concept 11

Review from the Classification chapter

You can use the **dot product** between non-unit vectors to measure their "similarity" **scaled by their magnitude**.

If two vectors are more **similar**, they have a **larger** dot product.

- If angle$< 90°$ they are "similar": $\vec{a} \cdot \vec{b} > 0$
- If angle$> 90°$ they are "different": $\vec{a} \cdot \vec{b} > 0$
- If they are **perpendicular** (angle=$90°$) to each other, $\vec{a} \cdot \vec{b} = 0$

So: as an approximation, the higher the dot product, the more similar they are!

This works extra well for something like an image, where the pixels have a restricted "range" of values: it's not as easy to get an extra-high dot product just because the magnitude are too large.

Now, we know what to do: we'll get the **dot product** between our window, and the **filter**, to see how similar they are.

- If they're similar enough, then we found the pattern!

Concept 12

To determine whether the window contains our pattern, we take the **dot product** between our **window** v and our **filter** f.

$$v \cdot f$$

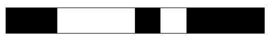
The **higher** the dot product, the **more likely** that we have our pattern.

- There's no exactly dot product value to be "sure" you've found your pattern: you have to choose your threshold based on context.

We'll show our example below.

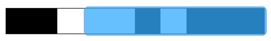
9.1.4 1D Example

So, suppose we have our input image:



$$\Longrightarrow \begin{bmatrix} +1 & +1 & -1 & -1 & -1 & +1 & -1 & +1 & +1 & +1 \end{bmatrix}$$

Our filter is size 3 (3 elements), so we'll grab a window of 3 elements.



$$\Longrightarrow \overbrace{\begin{bmatrix} +1 & +1 & -1 \end{bmatrix}}^{\begin{bmatrix} +1 & -1 & +1 \end{bmatrix}} \quad -1 \quad -1 \quad +1 \quad -1 \quad +1 \quad +1 \quad +1$$

We ignore everything after the first three elements.

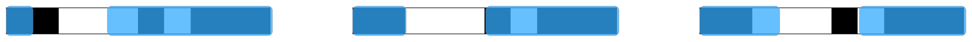
We then compute the result:

$$\overbrace{\begin{bmatrix} +1 & +1 & -1 \end{bmatrix}}^{v} \quad \Longrightarrow \quad \overbrace{\begin{bmatrix} +1 \\ -1 \\ +1 \end{bmatrix}}^{f} \cdot \overbrace{\begin{bmatrix} +1 \\ +1 \\ -1 \end{bmatrix}}^{v} \quad = \quad +1 - 1 - 1 = -1 \tag{9.2}$$

This is our first filtering: we get -1. This is the first element of our output:

$$y = x * f = \begin{bmatrix} -1 & ? & ? & ? & ? & ? & ? & ? \end{bmatrix} \tag{9.3}$$

We'll repeat for the rest of our 1d signal.



$$\begin{bmatrix} -1 & \mathbf{+1} & \cdots & ? \end{bmatrix} \quad \Longrightarrow \quad \begin{bmatrix} -1 & +1 & \mathbf{-1} & \cdots & ? \end{bmatrix} \quad \Longrightarrow \quad \begin{bmatrix} -1 & +1 & -1 & \mathbf{+1} & \cdots & ? \end{bmatrix}$$

This is **convolution**.

9.1.5 Convolution

Convolution simply applies our filter, at each position:

Concept 13

When filtering (doing **convolution**), the **output** at the i^{th} index is given by having **shifted** your window over from 0, $(i-1)$ times.

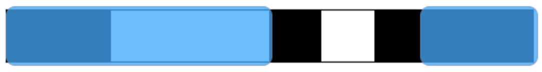
- The **indices** for our output usually start from index 1.

Example: The last example above, ending at index 3, outputs +1 after shifting right 3 times.

The result is a new vector:

$$y = x * f = \begin{bmatrix} -1 & +1 & -1 & +1 & -3 & +3 & -1 & +1 \end{bmatrix} \tag{9.4}$$

The pixel we have labelled in orange corresponds to the "bright spot" in our sequence:



As we hoped, the "matching" pattern is the highest positive magnitude!

With this, we've fully demonstrated 1-d **convolution**.

Definition 14

Convolution $x * f$ is the process of searching through a **signal** x for a particular **pattern**, using a **filter** f.

- The filter **matches** the pattern we're looking for.

The convolution process follows the following steps:

- Taking a **window** v in the same shape as the filter, **isolating** a section of your signal
- Applying a **dot product**-like operation between your filter f and your window v.
- **Sliding** your window, and **repeating**, until every output is computed.

- The $(i+1)^{th}$ index is given for the dot product computed by shifting over i times from index 1.

We call this a "**dot product-like operation**" to prepare us for **higher-dimensional** equivalents.

We can even write this in formula terms. To represent **windowing**, we'll use python slices, with the same conventions.

Key Equation 15

If we have **signal** x, **filter** f of **size** k, we can create a **window** v_i by...

Starting at the **leftmost** pixel, and shifting right by i units:

$$v_{i+1} = x\Big[i : i+k\Big] = \begin{bmatrix} x_{i+1} \\ x_{i+2} \\ \vdots \\ x_{i+k} \end{bmatrix}$$

- Note the subscript v_{i+1}: we start from $i = 0$, and thus v_1.

This is used to create our **convolution** $y = x * f$:

$$y_i = f \cdot v_i$$

The fact that x is 1-indexed, but python slicing is 0-indexed, is the reason why $x[i : i+k]$ starts at x_{i+1}.

You might see a different version of indexing in some situations:

Like in the official notes!

Clarification 16

Above, we used i to give us the leftmost slice of our input.

We did this because we assumed the **leftmost** pixel would be assigned $i = 0$.

- However, in some cases, the **middle** pixel is assigned $i = 0$: the pixels indices go equally positive or negative.

In which case, we would need to **replace** our slicing procedure above:

$$x\Big[i : i+k\Big] \quad \longrightarrow \quad x\Big[(i - \lfloor k/2 \rfloor) : (i + \lfloor k/2 \rfloor)\Big]$$

- We use the floor operator $\lfloor x \rfloor$ so that we index correctly, by integers.

One more thing: we need to be careful when we say we're doing "Convolution".

Clarification 17

In other fields, convolution requires **reversing the order** of your filter, before you apply it to your input.

However, this is typically **not** the case in machine learning.

9.1.6 Convolution Output Size

Something you might notice is that our output is **smaller** than our input was.

How much shorter? 2 elements: in general, the output of a convolution is $k-1$ elements **shorter** than the input.

Where k is, again, the size of our filter.

Why is this? We can see why, by focusing on the **leftmost** element of our filter: we can only shift it until our vector ends.

$$+1 \quad +1 \quad -1 \quad -1 \quad -1 \quad +1 \quad -1 \quad \overbrace{\begin{bmatrix}+1 & +1 & +1\end{bmatrix}}^{\begin{bmatrix}+1 & -1 & +1\end{bmatrix}} \tag{9.5}$$

But, our leftmost element hasn't reached the end of the vector: if it did, then the rest of the vector would be **sticking out**, with nothing to multiply with:

$$+1 \quad +1 \quad -1 \quad -1 \quad -1 \quad +1 \quad -1 \quad +1 \quad +1 \quad \overbrace{\begin{bmatrix}+1 & ? & ?\end{bmatrix}}^{\begin{bmatrix}+1 & -1 & +1\end{bmatrix}} \tag{9.6}$$

When our leftmost position is as far right as we can go, there are $k-1$ positions remaining: the rest of the filter is "in the way".

Concept 18

For a length-n input and a length-k filter, **1d convolution** creates an output of size:

$$n-(k-1)$$

9.1.7 Padding

We don't necessarily want to be shrinking the size of our output. How do we solve this?

Well, our equation above gives us two options: increase the input size, or decrease the filter size.

- Decreasing filter size is **restrictive**: the smaller the filter, the smaller the pattern we can search for.
- So, we'll just increase the size of our input.

We'll increase input size with **padding**: adding extra elements to the ends of our vector.

- Typically, we pad with 0's, to have the most neutral effect possible on our output.

Definition 19

Padding is a technique for increasing the size of the output of convolution.

To pad an input, you add filler values (usually 0's) to the **edges** of the input vector.

This allows the filter shift further in both directions.

A padding of p adds p values to **both sides** of our input vector, transforming our n-sized input into a $n + 2p$ sized input. Thus, our output size is:

$$(n + 2p) - (k - 1)$$

Where k is our filter size.

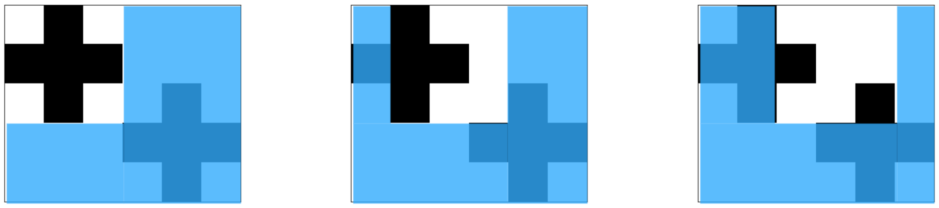
Example: Here's an example of **zero-padding** with $p = 2$:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix} \Longrightarrow \begin{bmatrix} 0 & 0 & 1 & 2 & 3 & 4 & 0 & 0 \end{bmatrix} \tag{9.7}$$

Often, we select our padding size so the output size is the same as the input size: $2p = k-1$.

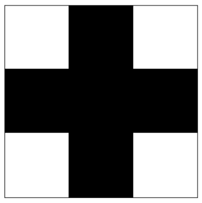
9.1.8 2D Filter

Now, we want to extrapolate this idea to higher dimensions: in particular, 2D, but this approach will work for any dimension.



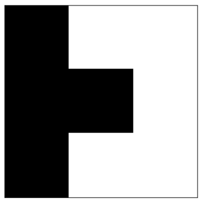
We're back to this example.

Let's review what we can already guess: first, we now have a 2-D pattern we're looking for, and a 2-D window cut out of our image.



$$\Longrightarrow \quad \begin{bmatrix} -1 & +1 & -1 \\ +1 & +1 & +1 \\ -1 & +1 & -1 \end{bmatrix}$$

This is our filter.



$$\Longrightarrow \quad \begin{bmatrix} +1 & -1 & -1 \\ +1 & +1 & -1 \\ +1 & -1 & -1 \end{bmatrix}$$

This is our window $w_{0,1}$: we've shifted over by one column.

How do we measure their similarity?

$$\begin{bmatrix} -1 & +1 & -1 \\ +1 & +1 & +1 \\ -1 & +1 & -1 \end{bmatrix} \text{ vs } \begin{bmatrix} +1 & -1 & -1 \\ +1 & +1 & -1 \\ +1 & -1 & -1 \end{bmatrix} \tag{9.8}$$

We measured similarity between vectors using the **dot product**.

We can break our matrices up into vectors, by treating them as vectors of vectors.

$$\begin{bmatrix}\begin{bmatrix}-1\\+1\\-1\end{bmatrix} & \begin{bmatrix}+1\\+1\\+1\end{bmatrix} & \begin{bmatrix}-1\\+1\\-1\end{bmatrix}\end{bmatrix} \quad \text{vs} \quad \begin{bmatrix}\begin{bmatrix}+1\\+1\\+1\end{bmatrix} & \begin{bmatrix}-1\\+1\\-1\end{bmatrix} & \begin{bmatrix}-1\\-1\\-1\end{bmatrix}\end{bmatrix} \tag{9.9}$$

So, we can compare the similarity between the first vector in our **filter**, and the first vector in the **window** using the **dot product**.

Concept 20

If the j^{th} vector that makes up **matrix** A is similar to the j^{th} vector that makes up **matrix** B, then A and B are **similar**.

$$\begin{matrix}\vec{a} \approx \vec{d} \\ \vec{b} \approx \vec{e} \\ \vec{c} \approx \vec{f}\end{matrix} \implies \begin{bmatrix}\vec{a} & \vec{b} & \vec{c}\end{bmatrix} \approx \begin{bmatrix}\vec{d} & \vec{e} & \vec{f}\end{bmatrix}$$

- We'll repeat this process for each column.

$$\overbrace{\begin{bmatrix}-1\\+1\\-1\end{bmatrix} \cdot \begin{bmatrix}+1\\+1\\+1\end{bmatrix}}^{\text{Col 1}} + \overbrace{\begin{bmatrix}+1\\+1\\+1\end{bmatrix} \cdot \begin{bmatrix}-1\\+1\\-1\end{bmatrix}}^{\text{Col 2}} + \overbrace{\begin{bmatrix}-1\\+1\\-1\end{bmatrix} \cdot \begin{bmatrix}-1\\-1\\-1\end{bmatrix}}^{\text{Col 3}} \tag{9.10}$$

So, we've matched the j^{th} column of our window with the j^{th} column of our filter.

- And a dot product matches the i^{th} row of that window vector, with the i^{th} row of the filter vector.

That means, we're multiplying **element-wise** across our matrix: the (i, j) element of f is multiplied by the (i, j) element of w.

Definition 21

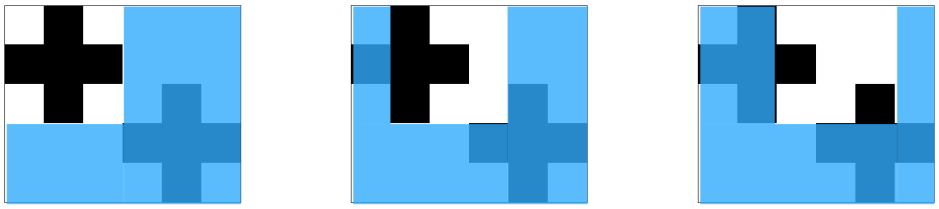
We introduce a **dot product generalization**, for a **matrix** (2-tensor):

- We compute it by multiplying **element-wise**, then **summing** all the elements.

$$A \cdot B = \sum_i \sum_j A_{ij} B_{ij}$$

This operation measures **similarity** between our two vectors.

This allows us to do higher-dimensional convolution.



We're back to this example.

9.1.9 2-D convolution

We'll need to shift around the image, in the same way that we did for the 1-d case.

- Before, we shifted over our **window** (size s) across our **input** (size k) by every possible position: this created $n-(k-1)$ outputs.

The process is the same here: we just need to shift along two axes.

- We'll need to consider every combination of shifting i rows down, and j rows right.
- In python, this is equivalent to a double for-loop: "for i in m, for j in n".

Concept 22
For **2-D convolution**, we need to **shift** our window along two axes.

- So, we have one window for each **combination** of shifting i rows down, j columns right.

If we have an **input** with an axis of length n, and a **filter** of size k, that output axis has **length**

$$(n+2p)-(k-1)$$

- k is typically the same on both of the 2d axes: it's usually **square**.

Remark (Optional)
Convolution was originally designed based on the way human eyes work: we use it to look for edges, and other distinct features in our vision.

9.1.10 Dot Product Generalization

Later, we'll need to generalize this to higher dimensions: we'll review the higher-dimensional version of a matrix, the **tensor**:

Definition 23
Review from the Matrix Derivatives Chapter:

An **array** of objects is an **ordered sequence** of them, stored together.

- The most typical example is a **vector**: an ordered sequence of **scalars**.
- A **matrix** can be thought of as a **vector** of **vectors**. For example: it could be a row vector, where every column is a column vector.

Thus, a vector is a 1-d array, and a matrix is a 2-d array.

We can extend this to any number of dimensions. We call this kind of generalization a **tensor**.

Definition 24
Review from the Matrix Derivatives Chapter:

In machine learning, we think of a **tensor** as a "**multidimensional array**" of numbers.

- Each "dimension" is what we have been calling an "**axis**".
- A tensor with c axes is called a **c-Tensor**.

Example: If we stacked a bunch of matrices in a box in 3-d, that would be a 3-tensor.

To get element-wise multiplication, we'll need a way to index into tensors: we'll use numpy notation.

Notation 25
We want to **index** into a tensor T, with n axes ("dimensions")

We'll use indices $i_1, i_2, i_3, \cdots i_n$ to get an element:

$$T\left[i_1, i_2, i_3, \cdots, i_n\right]$$

Finally, we can show the dot-product generalization for tensors:

This is where python slicing really shines: it makes it easier to talk about grabbing an element from an unknown tensor.

Definition 26

The **dot product generalization** for an arbitrary **n-Tensor**

- We compute it by multiplying **element-wise**, then **summing** all the elements.

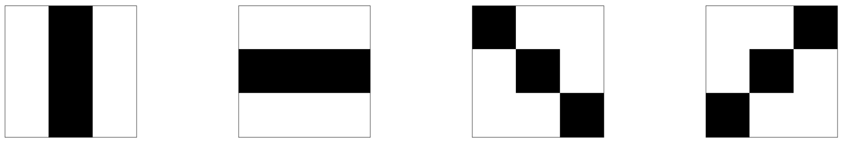
$$A \cdot B = \sum_{i_1, i_2, i_3, \cdots, i_n} A\Big[i_1, i_2, i_3, \cdots i_n\Big] \cdot B\Big[i_1, i_2, i_3, \cdots, i_n\Big]$$

9.1.11 Filter Banks

As we've shown, one filter produces one 2-d output: telling us where it "finds" or does not find the given pattern.

But, typically, when doing complex image analysis, we don't just want **one** filter. There are lots of different patterns we might be looking for.

- **Example:** Rather than programming every larger shape **directly**, it might be easier to look for smaller edges.
- You'll need a **different** filter for a vertical edge, or a horizontal edge, or a diagonal edge.



All four of these might be useful for the same image.

In practice, you almost always want to look for more than one pattern at the same time, in an image.

We'll store all of these filters together. Suppose we have m of these filters: each filter has size k.

- Each is a 2d matrix, so we'll **stack** them in the third dimension.
- This creating a **tensor** in the shape $(k \times k \times m)$.

This collection is called a **filter bank**.

Definition 27

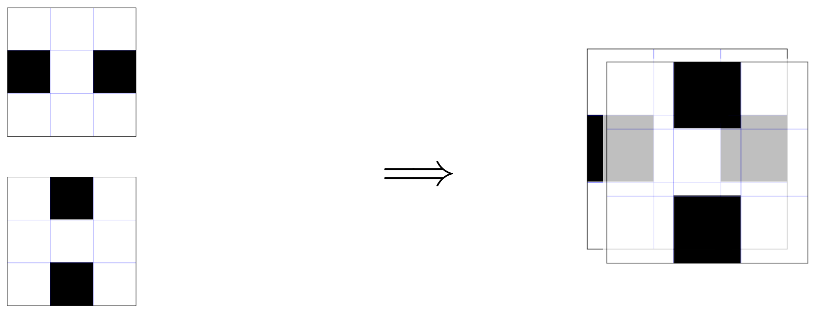
A **filter bank** is a collection of all of our $(k \times k)$ filters stacked into a **3-tensor**.

- Thus, if we have m of these filters, the **shape** is $(k \times k \times m)$.

These filters are all applied to our image in **parallel**: meaning, each is applied to the original image, and each creates a separate output.

Example: We might, for example, combine the two following filters:

It's difficult to visualize a 3d thing like this, so if this looks strange, don't worry.



Now, we have a very simple filter bank.

Clarification 28

This $(k \times k \times m)$ object could be a **filter bank**, but it could **also** be a single **3-tensor filter**, for a 3-tensor input.

Why would our **input** be a 3-tensor? We'll see why in a bit.

So, we'll use each of these filters, and **convolve** them with the input. Each creates a separate output stored in a separate **channel**.

Concept 29

A **channel** is the output of convolving **one filter** with our **image**.

In a 2d image problem, one channel is a **matrix**.

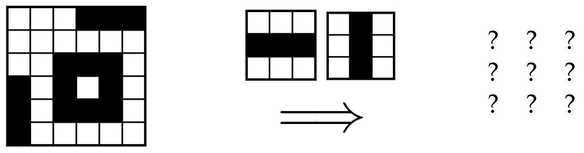
Each filter creates one channel. So, in order to depict all of our channels of output, we'll need another 3-tensor.

Concept 30

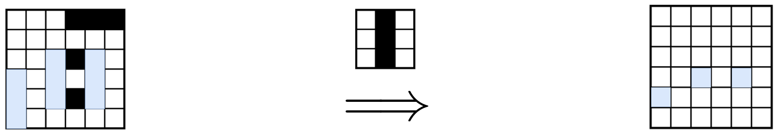
Suppose we have our 2d input.

If we have m filters in our **filter bank**, we end up with m **channels** in our output.

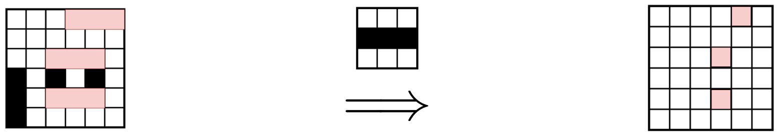
- **Example:** Here, we'll apply two filters: one detecting vertical lines, one detecting horizontal lines. It'll create two channels of output.



We're applying a simple filter bank to the image on the left.

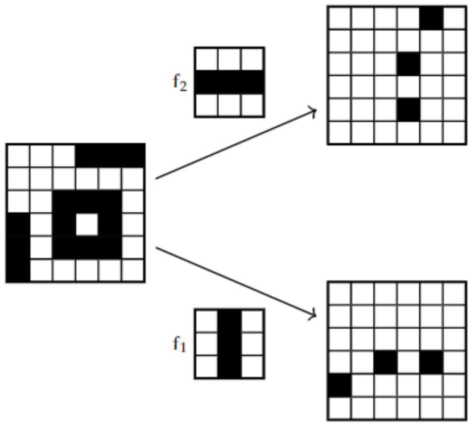


Our vertical detection.



Our horizontal detection.

Together, these create two channels:



9.1.12 Tensor Filters

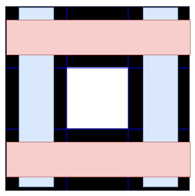
Now, we have two different channels in our output. What do we do with this result?

Each of our filters was designed to find a particular **pattern**: you could say it represents one "**perspective**" on the data.

- Our two filters above think about the data in terms of vertical lines, and horizontal lines.

We want to **combine** those perspectives to get useful information.

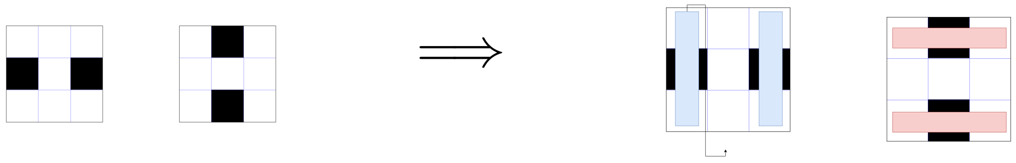
Example: A "square" is made out of two vertical lines, and two horizontal lines.



That means we want to find two **vertical** lines, and two **horizontal** lines: each on the opposite side of our center pixel.

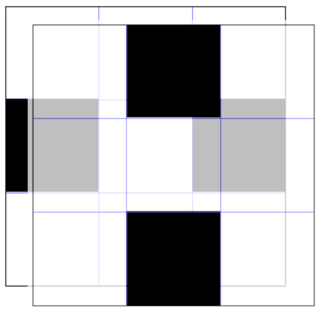
- This is a kind of **pattern** we could search for, but it's a pattern across **two channels**.
- That means that we need a filter occupying **multiple channels**.

Let's see the pattern we want to see on each channel:



The right side shows what each pixel on the filter "represents".

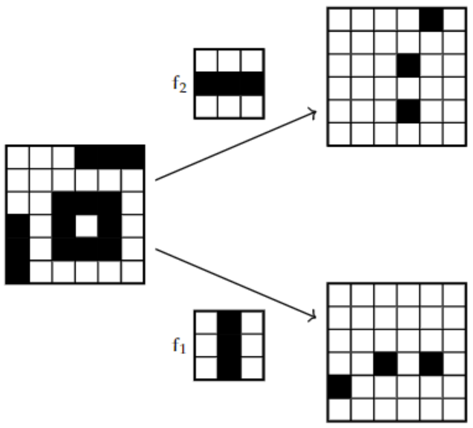
We want both at the same time, so we create a **3d filter**:



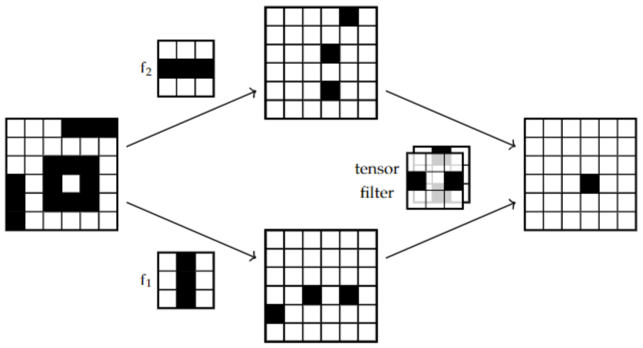
This looks like our 3d filter bank, of **2 filters**. But in this case, it's a **single** 3d filter.

Let's apply this trick to the two channels we designed earlier: we're going to find "donut" patterns in the original image.

Couldn't we have just used a donut-shaped filter in the first place, and then only need one filter?
Yes, but this is useful for demonstrative purposes.



Here's our previous work, finding vertical and horizontal lines.



And now, we'll combine those lines to create a square.

Look at that: our result is, we **only** get an output where there's a hollow square in the **original** input!

We've found a more interesting pattern, using a **second layer** of convolution.

9.1.13 Tensor Filters: All channels

When we introduce a **3-tensor filter**, we could imagine not only moving across the rows and columns of the input, as we convolve, but also the **channels**.

- Thus, we would need to shift our filter along 3 axes.

However, in practice, we frequently **avoid** this: instead, our tensor filter tends to have the **same number of channels** as the input tensor.

- If they have the same number of channels, then there's no space to "shift" along the third axis.

- That means that the output of this filtering is a single matrix again!

Technically, it's a tensor with the shape $(m \times n \times 1)$. The last dimension being 1 is why it's effectively a "matrix".

Concept 31

Typically, our **3-tensor filters** occupy **all channels of the input**.

- So, if the input is shape $(a \times b \times c)$, the filter has the shape $(k \times k \times c)$.

That means that our 3-tensor filter creates a **matrix** as its output, when we do convolution.

This allows us to add more tensor filters: our filter bank can contain multiple 3-tensors, and each one creates one channel of the output.

This means that our filter bank is a 4-tensor......... don't think too hard about it.

Concept 32

Often, we use several **3-tensor** filters: each one occupies **all of the input channels** at the same time.

- And each one outputs a single **matrix**.

That means that we can stack these 3-tensors into a **filter bank**: this object is now a **4-tensor**.

- When we apply this **4-tensor filter bank** to our **3-tensor input**, we get a **3-tensor output**.

9.1.14 Convolution is Linear

You might have noticed that, above, we could have **replaced** all of our filters with a single, donut-shaped filter.

- We didn't do this, so we could **demonstrate** how convolution works conceptually.

This is possible because convolution, being entirely made out of **multiplication** and **addition**, is a **linear** operation.

So, two consecutive convolutions are "**compressible**" to one, just like linear layers.

Concept 33

Machine learning "**convolution**", or "cross-correlation", is a **linear** operation.

- Here, we'll summarize linearity as "**multiplying** our variables x by scalars (in this case, weights from f), and **adding** the result together.

$$v \cdot f = \overbrace{\sum_i}^{\substack{\text{Summing}\\\text{Variables}}} \overbrace{v_i f_i}^{\substack{\text{Scalar}\\\text{Product}}}$$

In fact, as we'll see later when doing backprop, **convolution** between x and f can be represented using a particular **matrix multiplication**.

This isn't a problem in practice because we'll add ReLU and max-pool layers in between: both are **nonlinear**.

We haven't discussed max-pool yet.

That said, a "convolutional layer" is not a "linear layer":

Clarification 34

While convolution is **linear**, **a convolutional layer is different from a linear layer**.

Why is that?

Because convolution is a very **restricted** kind of linear:

- In **convolution**, we use the **same filter** for every dot product – every operation uses the same weights in f.
- In a **fully connected**, "**linear layer**", every input-output pair has a **separate, independent** weight, that can be tuned freely.

You could think that a FC/"linear" layer refers to the broadest, **least-restricted** kind of linear transformation:

- Every possible linear relationship is allowed.

You could also think of it as the "simplest" linear layer: it makes the least assumptions.

9.1.15 RGB colors (Optional)

One quick concern, that's more pragmatic: all of our images, so far, have been in black-and-white.

How do real pictures create color? Using the **RGB** system: each pixel has a certain brightness of red, green, and blue.

- So, to represent the pixel output, you need **three** numbers: a brightness in the $[0, 255]$ range for each color.

That means that our input isn't a 2d image: it's actually 3d.

Concept 35

If we're using **RGB** color instead of black-and-white (BW), each pixel requires **3 values** to represent the **brightness** of each color.

Thus, if our BW image had the shape $(m \times n)$, our RGB image has the shape $(m \times n \times 3)$: we use a **3-tensor** to store the extra information about color.

- Thus, our filters have to be 3d filters, as well.

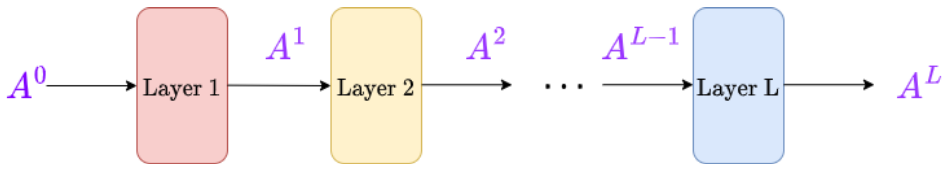
9.1.16 Adding Convolution to our Neural Networks

So, we've developed a complete system for **convolution**, using **filter banks**. How do we apply this to **machine learning**?

Well, thankfully, our neural networks are very **modular**:

- Each FC layer is **self-contained**, and **abstracted**: when we depict it this way, we only care about the input and output dimensions.

By "self-contained" and "abstracted", we mean that we can hide the contents of the layer, while still having a useful representation.



We've broken our model into "modules" that we could swap out: this representation doesn't acknowledge the weights, or the structure.

So, it's not too different if, instead of a **fully-connected** layer, we were to have a **convolutional** layer.

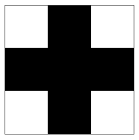
Concept 36
A **convolutional layer** can be inserted into a neural network by placing it **between** two other layers.

- You just need to make sure the input/output have the right **dimensions**.

Let's figure out how to **implement** that, while thinking in familiar NN terminology.

Convolution is based on your window and filter.

- Your **window** is simply given by your input tensor, and how far you've **shifted**.
- Your **filter** is what really defines your convolution: it chooses the **pattern** you're looking for.



$$\Longrightarrow \begin{bmatrix} -1 & +1 & -1 \\ +1 & +1 & +1 \\ -1 & +1 & -1 \end{bmatrix}$$

So, we'll focus on the filter: this is how we determine the behavior of our convolutional layer.

- This is similar to how our **weight** matrix W is used to configure a layer of our FC NN.

- So, we say that our convolutional layer is defined by the **weights** in our filters.

Note that we say **filters**: we already established that we can use multiple filters. If we do, our output will have multiple channels.

Definition 37

Our **convolutional layer** is entirely determined by the **weights** we choose for our **filter bank**.

- For **each filter** in our filter bank, we'll also include a single **offset**, or bias term.

Suppose that, in the 1d case, we have a window of size k, for our input x. Our window shifted by i, is labelled v_{i+1}.

$$v_{i+1} = x\Big[i : i+k\Big] = \begin{bmatrix} x_{i+1} \\ x_{i+2} \\ \vdots \\ x_{i+k} \end{bmatrix}$$

We've chosen weights for our filter f, with a bias term f_0. The i^{th} element of our **output** for that filter is:

$$y_i = v_i \cdot f + f_0$$

Example: If we have a 2d filter of length k, then we need k^2 weights, and 1 bias. We have $k^2 + 1$ parameters.

$$f = \begin{bmatrix} W_{11} & W_{12} & \cdots & W_{1n} \\ W_{21} & W_{22} & \cdots & W_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ W_{m1} & W_{m2} & \cdots & W_{mn} \end{bmatrix} \qquad f_0 = W_0 \tag{9.11}$$

Each convolutional layer has one filter bank, typically.

We casually tossed in a **bias** term, similar to our neural network structure.

- But we should ask, what effect does that bias term have?

Concept 38

A **higher** filter output suggests a **higher chance** that our desired **pattern** is found at that position.

Thus, our **bias/offset** term can increase or decrease how "**sensitive**" we are to inputs similar to our pattern.

- If we increase the offset, we get a higher filter output: more inputs will appear as **positive**: possibly matching our pattern.

9.1.17 Training our Convolutional Layer

In the past, experts would **hand-craft** their filters, manually experimenting with them.

However, we've defined our convolutional layer in terms of **trainable** weights and biases.

- So, as long as we can take the **derivative**, we can use **gradient descent** to find filters which are more suitable for the task.

Concept 39

We can **train** the weights and biases used in our filter bank.

This requires doing **backpropagation** to find the gradient, but that's possible so long as we have well-defined **derivatives**.

We'll come back to how to compute these derivatives in the last section of this chapter.

Sometimes, these filters teach us interesting things about the **structure** of the data, based on which ones ended up being **useful**!

- They've even been found to sometimes recreate the types of successful designs made by humans.

9.1.18 Benefits of Convolution

We've already discussed some of the benefits of convolution:

Concept 40

Convolution provides **spatial locality** and **translation invariance**.

- **Spatial Locality**: our "filter" focuses on a **local** region of the image, and looks for a specific, **spatial** arrangement of pixels.
- **Translation Invariance**: we repeatedly apply the **same** filter as we **move** across the image. So, it will find our pattern and recognize it the same, no matter the position.

Of course, there's some caveats:

Clarification 41

Convolution doesn't perfectly provide translation invariance.

This is because of the **edges** of our image.

- If we don't use zero-padding, then information close to the edge of the image is scanned over **fewer** times.
- If we do use zero-padding, then the information close to the edge is **distorted** by the zeroes.

But there's one more surprising benefit.

The same filter is used, over and over again, as we move over the image.

- That means we repeatedly re-use the **same weights** for multiple different calculations.

This can be a bit confusing: the same weights will appear in different calculations, and thus different derivatives.

Definition 42

Weight sharing is a useful property of convolution, where the **same weights** are re-used for **multiple calculations**.

- In particular, the weights in a filter are used for many **dot products**, in the same convolution.

Having fewer weights allows our model to **train faster**, and possibly **overfit** less: it's a form of **regularization**.

Example: Let's compare two situations: in both, we have a (5×5) image, and we want a (5×5) output.

- FC Layer: We flatten our input and output to (25×1).
  - To get every combination of input and output, we need $25 * 25$ weights.
  - 1 bias for each output: $25 * 1$ biases.
  - **Total**: 650 parameters.
- Conv. Layer: We keep our current shape and use a single filter.
  - We use a (3×3) filter, with one unit of padding $(p = 1)$. That means 9 weights.
  - We have one bias: 1 bias term.
  - **Total**: 10 parameters.

In a way, weight-sharing makes our model more **efficient**. In exchange, it's less **flexible**: it makes some assumptions about how our data is structured.

9.1.19 Our NN dimensions

So, we are considering introducing a convolutional layer with layer ℓ.

We should be careful of how to notate our dimensions:

Notation 43

For a convolutional layer on layer ℓ:

- **Input length**: $n^{\ell-1}$
- **Input channls**: $m^{\ell-1}$
- **Filter size**: k^{ℓ}
- **Number of filters**: m^{ℓ}
- **Padding length**: p^{ℓ}

A few notes:

- The input parameters are $\ell-1$, because they're the **output** of the previous layer $\ell-1$.
- Notice that m is used for the filter count, and the channels of the input.
  - The input is a previous output, and the **output** has the same number of **channels** as the **filter bank**.

Now, we can use these to get the shapes of some objects:

Definition 44

For a convolutional layer on layer ℓ:

- **Input tensor shape**: $(n^{\ell-1} \times n^{\ell-1} \times m^{\ell-1})$
- **Filter shape**: $(k^{\ell} \times k^{\ell})$
- **Filter bank shape**: $(k^{\ell} \times k^{\ell} \times m^{\ell})$

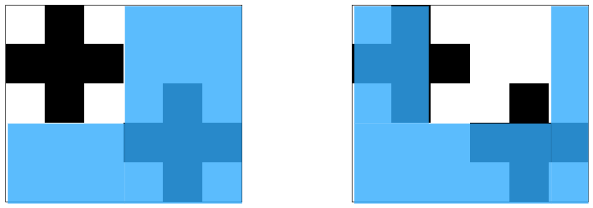
Again, we see that our input can be a **3-tensor**, if it has multiple channels.

This can either be due to filter banks, or the structure of the data (like in the RGB section).

9.1.20 Stride

There's one more thing we've skipped over: until now, we've assumed that, as we take a convolution, we move over, **one index** at a time.

But, this isn't required: we could move by **multiple** units.



This is the same as what we did before, except now, we've "skipped" one of our intermediate steps.

There are multiple reasons to do this, one of which involving "max pooling", which we discuss in the next section.

Definition 45

Stride is the distance you travel each time you **move indices** to take another **window** from your input.

Example: The stride we were using until now is 1. The stride we used in the above diagram is 2.

Naturally, this will shrink the size of your output:

Concept 46

Increasing **stride** s decreases the **size** of your output.

$$\text{New Size} = \left\lceil \frac{\text{Old Size}}{s} \right\rceil$$

We divide by s, because we're "skipping over" some windows: we are only taking a "fraction" of them.

Note the symbols on the side:

Notation 47

$\lceil x \rceil$ takes the "**ceiling**" of x: if x is between two integers, we round up.

We need this because the size of our output matrix is an **integer**.

- If we have a length-5 output with stride 1, but we take stride 2 instead, without rounding, we end up with size 2.5.
- So, instead we round.

9.1.21 Output shape

Now, we have all the tools we need, to compute the output shape, based on the input shape.

Three things can affect our input shape:

- Filter size: $n - (k-1)$
- Padding: $n + 2p$
- Stride: $\lceil n/s \rceil$

Taking all of these variables together, we get this result (which is important, and worth saving!):

Key Equation 48

Suppose we apply **convolution** to a matrix, with

- **Input size** $n^{\ell-1}$
- **Filter size** k
- **Padding** p
- **Stride** s

The **output size** will be

$$n^{\ell} = \left\lceil \frac{n^{\ell-1} - (k^{\ell} - 1) + 2p^{\ell}}{s^{\ell}} \right\rceil$$

More commonly, you will see a (surprisingly) equivalent expression:

$$n^{\ell} = \left\lfloor \frac{n^{\ell-1} - k^{\ell} + 2p^{\ell}}{s^{\ell}} + 1 \right\rfloor$$

- Instead of the ceiling function $\lceil x \rceil$, which rounds up, we have the floor function $\lfloor x \rfloor$, which rounds down.

Example: Let's take an input tensor of shape $(64 \times 64 \times 3)$.

Our filter is size 2 $(k = 2)$, with stride 2 $(s = 2)$.

- It needs to have 3 channels, to match the input. Thus, $(2 \times 2 \times 3)$.

Using our equation for the size of our output, we get

$$\lceil (64 - (2+1) + 2 \cdot 0)/2 \rceil = \lceil (63)/2 \rceil = 32$$

So, our output dimensions are $(32 \times 32 \times 1)$.

This example is slightly different from the official notes: there, we were doing max-pool, so we keep all 3 channels.

9.2 Max-pooling

So, we've used our filters to find **basic** patterns, roughly matching the filter.

- But earlier, we showed an example where we used **two layers** of convolution, to create a more complex pattern from a simpler one.

Of course, in that case, the two layers were reducible to a **single** layer, because convolution is a **linear** operation.

But we could get a more "true" version of this idea, by introducing a new function: the **max-pool** operation.

9.2.1 Aggregating information

Let's clearly state our goal:

Concept 49

One goal of **multi-layer convolution** is to

- Find **local**, smaller patterns
- Combine them to create **bigger**, more complex patterns

With each layer, we find broader and broader patterns.

However, we need a way to truly "aggregate" those patterns together.

This is the goal of our **max-pool** function.

- **Example:** We combine simple edges, into larger, **longer** edges, then into shapes like **squares**.
- Then, those combine into **windows** and doors and roofs, and finally, if they're arranged correctly, we use them to draw a "**house**".

We need a function that allows us to "**aggregate**" data this way.

Here's one idea: what happens as we move to higher size scales, building up a more **complex** object?

- We tend to care **less** about the smaller, individual details.

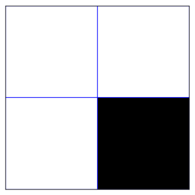
Our house picture doesn't stop being a house, if we slightly corrupt some of the edges, for example.

Rather than knowing **exactly** where a pattern is, we might simplify that to knowing **approximately** where that pattern is.

That way, we can gather information: "in this general **section** of the image, I found the pattern we're looking for!"

- How do we implement this?

It sounds like we want to replace "here's the exact pixel we found the pattern in", with "we found the pattern in this **general area**".



If black pixel indicates a "success" for finding the pattern, then this (2×2) grid does contain our pattern!

Definition 50

The **region** we're applying our **maxpool** operation to is called our **receptive field**.

- This is similar to the **window** used by filters.

Typically, it's square grid: $(k \times k)$.

9.2.2 Deriving max-pool

In this image, how did we know that the pattern was here? We noticed, "**at least** one pixel in this grid detected the pattern".

- In other words, we don't care about pixels that **don't** detect the pattern.
- And we don't care if there are **multiple**.

We can get our desired behavior by taking the **max** of these pixels: the pixel with the greatest output, is the **most similar** to our pattern.

- So, if the "most similar" pixel doesn't match our pattern, then none of the others will, either.
- Naturally, this also ignores multiple instances of our pattern.

This process, of **pooling** together all the pixels in our receptive field, and taking their **maximum**, is called the **max-pool** operation.

We'll repeatedly apply this process, all over our image: that way, we can aggregate over each region.

Definition 51

The **max-pool** operation, applied to a **receptive field**, takes the **max** of all values over that region.

Similar to convolution, we repeatedly **shift** our receptive field, and compute the max again.

- Also similar to convolution: the **number of times** we've **shifted** over, is the **index** of the output.

Example: Here's a 2x2 max-pool operation, with a **stride** of 2:

$$\begin{bmatrix} 1 & 3 & -9 & -22 \\ 44 & -10 & -11 & -1 \\ 0 & 10 & 4 & -3 \\ 11 & 9 & 321 & 99 \end{bmatrix} \xrightarrow{\text{max-pool}} \begin{bmatrix} 44 & -1 \\ 11 & 321 \end{bmatrix} \tag{9.12}$$

Concept 52

Max-pool typically only uses a **2d matrix** for its **receptive field**: we apply the max-pool operation separately, for each channel of our input.

So, the number of channels is the same, before and after our input.

9.2.3 Max-pool stride

Notice that, in our example above, we've **shrunk** the image, while preserving some general data.

Max-pooling is typically designed to "**gather**" data across our image:

- If we apply it after a **convolutional layer**, it can help us figure out if the receptive field contains a **pattern**.

In other words, we're **searching** for our pattern over a **larger** region.

So, it might be natural to take **bigger steps** in between each max-pool, since each max-pool condenses a whole receptive field of information.

Key Equation 53

In order to get a simplified, "broader" view of the data, our **max-pool** often uses a larger **stride** s:

$$s > 1$$

This means we move our receptive field by a larger amount, between max-pools. This **shrinks** our output.

- If we apply this for multiple consecutive layers, we get a "**pyramid**" shape, where our output gradually shrinks in size.

With this approach, we can store the information we care about ("pattern found roughly here/not here"), without focusing on the exact, **pixel-perfect** detail.

- It's also useful for building larger objects: if two patterns (from two **filter channels**) are **roughly** nearby, it'll be easier to recognize a **larger shape**.

Because often, different instances of a shape won't be exactly the same: this makes it easier to recognize them anyway.

If we don't want to shrink our image as much, we can use **zero-padding**, usually on our convolution step.

That said, while we want a stride that's bigger than 1, we don't want it to be so big that we **skip** some of the input.

Key Equation 54

In order to avoid **skipping** portions of the input, we don't want our **stride** s to be larger than our filter of **size** k.

$$k \geqslant s$$

9.2.4 Clarifications on max-pooling

Our max-pool essentially chooses the "**most likely** to match" output, after the filtering. Based on that result, we can guess whether this region contains our pattern.

Clarification 55

Neither **convolution** nor **max-pooling** exactly tell us if we found a pattern **match** at a particular **index**.

Instead, they give us a **number**, based on a (generalized) **dot product**, that can be **interpreted** to check for a pattern match.

- Whether that number confirms our pattern, depends on how **high** it is.

Our **offset** can help with this problem: it helps us set a "**threshold**":

$$v_i \cdot f > -b \qquad \Longrightarrow \qquad v \cdot f + b > 0$$

If we set b correctly, we could simply say, "the pattern appears if $v \cdot f + b > 0$".

- This threshold, like our other parameters, will be **learned** by the neural network.

Example: Consider the following example:

$$\begin{bmatrix} +1 \\ +1 \\ +1 \end{bmatrix} \cdot \begin{bmatrix} +1 \\ +1 \\ -1 \end{bmatrix} = 1 + 1 - 1 = +1 \tag{9.13}$$

This output is **positive**, but the two patterns aren't visually the same. Whether they're **similar** enough depends on the context.

- If they're not similar enough to justify a positive output, we could use a **negative offset** to make our filter less sensitive.

A related comment: some pattern matches might be ambiguous.

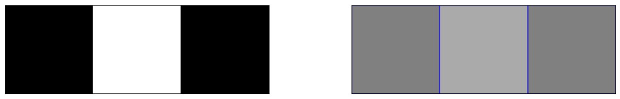
Clarification 56

In this chapter, for our images, we've exclusively used simplified images, with only **extreme** brightnesses (-1 or +1).

However, in most real images, there's a **spectrum** of brightnesses.

This can make it **harder** to figure out whether you really find a pattern, in a particular place.

Example: Whether the following image contains our pattern is unclear. The left is our filter, the right is our window.



The pixel in the center of the window is a bit brighter than the surroundings, but... not by very much.

This is another place where our bias can be useful to set a threshold.

9.2.5 Max-pool: A functional layer

Max-pool, as a specific variant of the max function, has **no parameters**: "compute the maximum input value" isn't a function that requires **adjustment**.

Concept 57

max-pool has no **parameters**: it always behaves the same way.

This also means that it doesn't need to be **trained**.

Because it has no weights, and behaves like a **function**, we can think of this a **pure functional layer**.

- Activation functions for linear layers, like **ReLU**, behave the same way: they require **no parameters**.

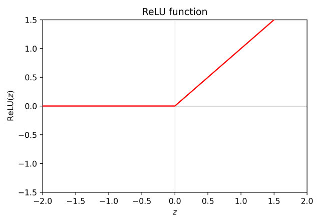
On that same note: while max-pool is *technically* nonlinear, we usually don't use it to provide nonlinearity to our model.

Instead, we accomplish this by applying ReLU **after** our convolution.

Concept 58

After convolution, we often apply a **ReLU function** to provide **nonlinearity** to our model.

Typically, we follow a sequence of **convolution**, **relu**, and then **max-pool**, before repeating.



A reminder of how the relu function appears.

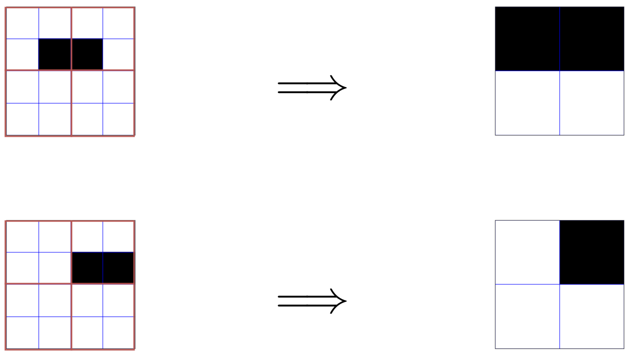
9.2.6 Max-pool: Some problems with "translation invariance".

There's one problem with having a larger stride:

- We skip some of the possible receptive fields.

This means that the same pattern could look different, based on where it's placed.

Example: We can get markedly different results of our max-pooling, just by shifting the input:



We shifted over the input, and lost one of our two black pixels: that doesn't seem very translation-invariant.

Concept 59

Using a stride s **greater than one** creates an output that isn't **translation-invariant**:

- if you **shift** part of the input slightly, it can alter the pattern recognition of the **output**.

This is notable for **max-pool**, which almost always uses $s > 1$.

Here, we provide a paper that provides a potential solution. https://arxiv.org/pdf/1904.1148

- In short, the idea is to max-pool with stride 1, and then **scale down** our output by averaging over the results.

9.3 Typical architecture

Now that we've built all the pieces of our new neural net, we can get the general flow of a neural network that implements convolution: a **Convolutional Neural Network** (CNN).

First, let's consider what we need for each layer of convolution:

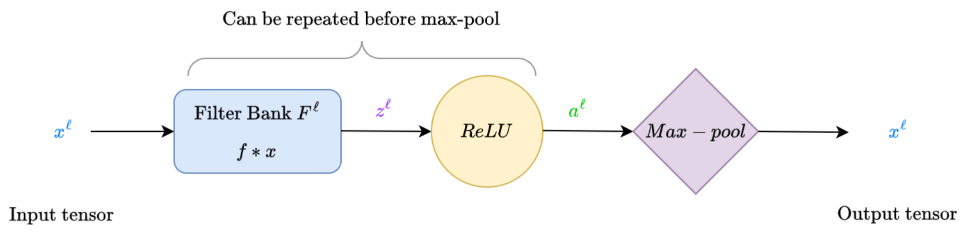
Concept copied from above, for convenience.

Concept 60

After convolution, we often apply a **ReLU function** to provide **nonlinearity** to our model.

We typically use layers of **convolution**, **relu**, and then **max-pool**, then repeat.

- Notably, we may do conv+relu multiple times before one max-pool.



This is our basic structure: often, we repeat this multiple times.

Once we reduced our input to a reasonably small size, we finish by using a **fully connected layer**.

The convolutional layers can be see as "preparing" the data, so that our fully-connected network can more easily find thep patterns it needs.

We could even model it as a very complex feature transformation!

Definition 61

A **Convolutional Neural Network** (CNN) is a neural network which uses **convolution** to transform data.

Most typically, we take the following structure:

- Several layers of **conv-relu-maxpool**, gradually shrinking the **output** size
- A **fully-connected** network, typically intended for classification or regression.

The model is **evaluated** based on the performance on the chosen classification/regression task.

- This can be viewed as several layers of convolution, applied before a regular neural network.

We can refer to our earlier comments for some benefits of CNNs:

Concept 62

Convolution provides benefits through **spatial locality**, **translation invariance**, and **weight sharing**.

- **Spatial Locality**: our "filter" focuses on a **local** region of the image, and looks for a specific, **spatial** arrangement of pixels.
- **Translation Invariance**: we repeatedly apply the **same** filter as we **move** across the image. So, it will find our pattern and recognize it the same, no matter the position.
- **Weight sharing**: the same filter weights are **re-used** for many calculations. This can speed up training, and reduce overfitting.

As a result, CNNs tend to perform well for image-based problems.

One might ask: how many **layers** of convolution? How many **filters** per layer? What **size** should these filters have?

- These questions are good ones, but they're very **difficult** to answer: very few hard rules exist for how to design this kind of network.
- Often, designs are based on what has worked in the **past**, or some **intuition** about the data.

Once we've designed our network, we can begin training.

Concept 63

CNNs can be trained just like normal neural networks: we train both the **fully connected** network, and the **filter weights** throughout the convolutional layers.

To do gradient descent, we measure the **performance** of our CNN on the classification/regression task in question.

We close out this section with an example of a "typical" CNN:

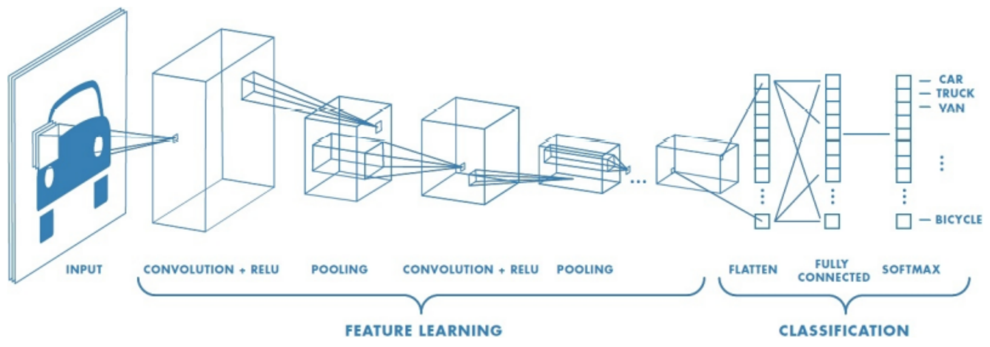


Figure source: https://www.mathworks.com/solutions/deep-learning/convolutional-neural-network.html

A few comments:

- Our input has **three** layers, to show the **RGB** color channels.
- In our convolution and pooling sections, we have **boxes**, not flat matrices: those represent **3-tensors**.
- Our last layer is **softmax**: our typical output for multi-class classification.
- We do end up **flattening** after our convolution is finished: that's necessary for feeding our FC network.

9.4 Backpropagation in a simple CNN

Now that we have a new type of neural network, it's only appropriate that we learn how to **train** it!

- Our filtering, ReLu, and maxpool functions are (mostly) **continuously differentiable**, so we can get useful derivative for **gradient descent**.

9.4.1 Our Simplest Example

We'll consider the simplest possible example, with a **1d** input.

Reminder that we're using A^ℓ notation to indicate the ℓ^{th} layer of our network.

- An n-length, one-channel, 1d input x: shape $(n \times 1 \times 1)$.
  - We'll use zero-padding of length p, to create our input $X = A^0$.

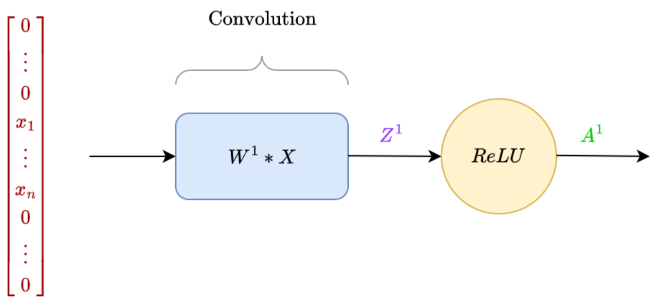
With padding, our shape is $((n+2p) \times 1 \times 1)$.

$$A^0 = X = \begin{bmatrix} 0 & \cdots & 0 & \overbrace{x_1 \quad \cdots \quad x_n}^{x} & 0 & \cdots & 0 \end{bmatrix}^T \tag{9.14}$$

- One layer of **conv-relu**
  - Our convolution has one size-k filter, with weights W^1: shape $(k \times 1 \times 1)$.
  - Stride $s = 1$.

$$Z^1 = \underbrace{W^1 * A^0}_{\text{Convolution}} \qquad \longrightarrow \qquad A^1 = \text{ReLU}(Z^1) \tag{9.15}$$

We can visualize our results so far:

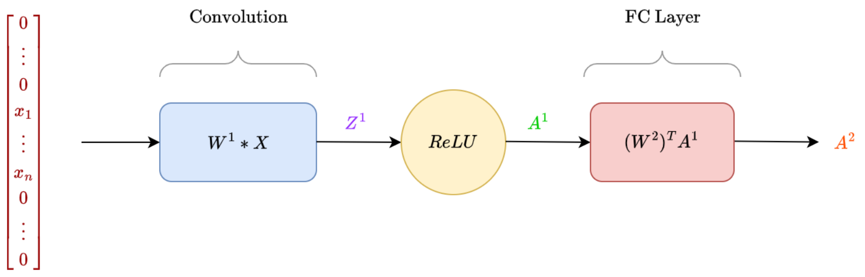


- A single FC layer for **regression**
  - Using weights W^2, with no bias.

$$A^2 = (W^2)^T A^1 \tag{9.16}$$

- A loss function using **squared difference**.

$$\mathcal{L}(A^2, y) = (A^2 - y)^2 \tag{9.17}$$



Notation 64

Reminder that, in this chapter, $a * b$ refers to the machine learning convolution/**cross-correlation** between a and b.

- In other fields, $a * b$ refers to the "true" convolution, where we flip the order of either a or b before using the same operation.

9.4.2 Chain rule to get full derivative

We know how to do gradient-descent on our **fully connected** layer already, and our **ReLU** layer is purely functional: no trainable weights.

So, all that's left is our **filter** derivative: our filter is parametrized by W^1.

$$\frac{\partial \mathcal{L}}{\partial W^1} \tag{9.18}$$

Looking at our above diagram, we can "move backwards" in steps, building up a **chain rule**, until we reach W^1.

$$\frac{\partial \mathcal{L}}{\partial A^2} \longrightarrow \frac{\partial \mathcal{L}}{\partial A^2} \cdot \frac{\partial A^2}{\partial A^1} \longrightarrow \frac{\partial \mathcal{L}}{\partial A^2} \cdot \frac{\partial A^2}{\partial A^1} \cdot \frac{\partial A^1}{\partial Z^1} \tag{9.19}$$

Finally, we get:

$$\frac{\partial \mathcal{L}}{\partial W^1} = \frac{\partial \mathcal{L}}{\partial A^2} \cdot \frac{\partial A^2}{\partial A^1} \cdot \frac{\partial A^1}{\partial Z^1} \cdot \frac{\partial Z^1}{\partial W^1} \tag{9.20}$$

9.4.3 Easy, Familiar Derivatives

We already know several of these terms:

Remember that A^2 is not "A squared": it's A for layer 2.

$$\mathcal{L}(A^2, y) = (A^2 - y)^2 \implies \frac{\partial \mathcal{L}}{\partial A^2} = 2(A^2 - y) \tag{9.21}$$

$$A^2 = (W^2)^T A^1 \implies \frac{\partial A^2}{\partial A^1} = W^2 \tag{9.22}$$

9.4.4 ReLU Derivative

Our next derivative is ReLU, one of the tricky **functional layers**.

$$A^1 = \text{ReLU}(Z^1) \tag{9.23}$$

For a full dive explanation of this derivative, go to **Explanatory Notes – Matrix Derivatives, A.9.4.**

For now, we'll take the result for granted.

Concept 65

Review from Matrix Derivative Chapter:

Each **activation** is only affected by the **pre-activation** in the **same neuron**.

So, if the **neurons** don't match, then our derivative is zero:

- i is the neuron for pre-activation z_i
- j is the neuron for activation a_j

$$\frac{\partial a_j}{\partial z_i} = 0 \qquad \text{if } i \neq j$$

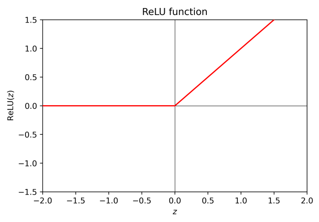
So, our only nonzero derivatives are

$$\frac{\partial a_i}{\partial z_i}$$

So, our result is a **diagonal** matrix: the off-diagonal elements are all zero.

$$\frac{\partial A_i}{\partial Z_j} = \begin{cases} f'(Z_i) & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \tag{9.24}$$

What about the **diagonal** elements? Well, that's based on the **ReLU** function.



Another picture of our ReLU function.

$$f(Z_i) = \begin{cases} Z_i & \text{if } Z_i > 0 \\ 0 & \text{otherwise} \end{cases} \qquad \Longrightarrow \qquad f'(Z_i) = \begin{cases} 1 & \text{if } Z_i > 0 \\ 0 & \text{otherwise} \end{cases} \tag{9.25}$$

We get our final result by combining these two facts: the diagonal structure, with the ReLU derivative.

Key Equation 66

The **derivative** between the length-m input Z and output A of the **ReLU** function is an $(m \times m)$ **diagonal matrix**, whose diagonals are

$$\frac{\partial A_i}{\partial Z_i} = \begin{cases} 1 & \text{if } Z_i > 0 \\ 0 & \text{otherwise} \end{cases}$$

Being a diagonal matrix, the off-diagonal elements are all zero.

$$\frac{\partial A_i}{\partial Z_j} = \begin{cases} 1 & \text{if } Z_i > 0 \text{ and } i = j \\ 0 & \text{otherwise} \end{cases}$$

Example: One possible matrix might look like

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{9.26}$$

Where Z_3 was negative, and thus it was in the 0, flat region of ReLU.

9.4.5 Filter Derivative

Lastly, we want to compute a **derivative**, based on the output of our **filter**.

$$Z^1 = W^1 * A^0 \qquad \Longrightarrow \qquad \frac{\partial Z^1}{\partial W^1} = ??? \tag{9.27}$$

The problem is: we don't **have** a derivative for our **convolution**. Let's find an **expression** to get the derivative from.

The easiest way to do that is to look at our **equations** for convolution:

Key Equation 67

Review from above, section 9.1.5

If we have **signal** x, **filter** f of **size** k, we can create a **window** v_i by...

Starting at the **leftmost** pixel, and shifting right by i units:

$$v_{i+1} = x\Big[i : i+k\Big] = \begin{bmatrix} x_{i+1} \\ x_{i+2} \\ \vdots \\ x_{i+k} \end{bmatrix}$$

- Note the subscript v_{i+1}: we start from $i = 0$, and thus v_1.

This is used to create our **convolution** $y = x * f$:

$$y_i = f \cdot v_i$$

Let's try to create something differentiable out of our equation for elements of Z:

We'll swap out v_{i+1} for v_i. This makes our slicing a little ugly, but we'll just omit that.

$$Z_i = W \cdot v_i = \begin{bmatrix} W_1 \\ W_2 \\ \vdots \\ W_k \end{bmatrix} \cdot \begin{bmatrix} X_i \\ X_{i+1} \\ \vdots \\ X_{i+k-1} \end{bmatrix} = \sum_{j=1}^{k} W_j X_{i+j-1} \tag{9.28}$$

Now we have something differentiable: a sum of products! Let's find $\partial Z_i^1 / \partial W_j^1$.

- If we're differentiating with W_j^1, we can ignore every term of the sum that doesn't include it.
- All that remains is the one term containing W_j.

$$W_j^1 X_{i+j-1} \tag{9.29}$$

Thus, we find:

$$Z_i^1 = \sum_{j=1}^{k} W_j^1 X_{i+j-1} \qquad \Longrightarrow \qquad \frac{\partial Z_i^1}{\partial W_j^1} = X_{i+j-1} \tag{9.30}$$

Key Equation 68

The **derivative** between the output of the **convolution** and its **weights** are given by a matrix, containing elements from the **input**:

$$\frac{\partial Z_i}{\partial W_j} = X_{i+j-1}$$

The matrix for $\frac{\partial Z^1}{\partial W^1}$ has the shape $(k \times n)$.

Example: Suppose we had a simple example: 4 weights in W, 6 variables in X. With stride 1, that gives 3 outputs in Z.

$$\frac{\partial Z^1}{\partial W^1} = \begin{bmatrix} X_1 & X_2 & X_3 \\ X_2 & X_3 & X_4 \\ X_3 & X_4 & X_5 \\ X_4 & X_5 & X_6 \end{bmatrix} \tag{9.31}$$

With this last derivative, we can assemble our chain rule, and compute the gradient for $\partial\mathcal{L}/\partial W^1$.

We can confirm this with some shapes:

- Size of X is m: $(m \times 1)$.
- Size of Z^1 and A^1 is n: $(n \times 1)$.
- Size of A^2 is 1: it's a scalar.
- Size of filter W^1 is k: $(k \times 1)$.

Using our knowledge from the matrix derivatives chapter, we can confirm our shapes:

$$\overbrace{\frac{\partial \mathcal{L}}{\partial W^1}}^{(k\times 1)} = \overbrace{\frac{\partial Z^1}{\partial W^1}}^{(k\times n)} \cdot \overbrace{\frac{\partial A^1}{\partial Z^1}}^{(n\times n)} \cdot \overbrace{\frac{\partial A^2}{\partial A^1}}^{(n\times 1)} \cdot \overbrace{\frac{\partial \mathcal{L}}{\partial A^2}}^{(1\times 1)} \tag{9.32}$$

9.4.6 Maxpool derivative

We didn't include a **maxpool** unit in our CNN. How do we compute the **derivative** of that?

Well, let's consider a simplified case: a 1d window of 2 elements: a_1 and a_2.

$$\text{maxpool}(A) = \max\left(\begin{bmatrix} a_1 & a_2 \end{bmatrix}\right) = \begin{cases} a_1 & \text{if } a_1 \geqslant a_2 \\ a_2 & \text{if } a_1 < a_2 \end{cases} \tag{9.33}$$

- Notice that if $a_1 = a_2$, it **doesn't matter** which of the two you select.

We can just take the derivative from one of these inputs, let's say a_1.

$$\text{maxpool}(A) = \begin{cases} a_1 & \text{if } a_1 \geqslant a_2 \\ a_2 & \text{if } a_1 < a_2 \end{cases} \qquad \Longrightarrow \qquad \frac{\partial\,\text{maxpool}(A)}{\partial a_1} = \begin{cases} 1 & \text{if } a_1 \geqslant a_2 \\ 0 & \text{if } a_1 < a_2 \end{cases} \tag{9.34}$$

This gives us something we can **generalize** to more a_i terms:

- If a_i is **biggest**, then it'll be the output of maxpool, and it'll have an effect – a **nonzero** derivative.
- If a_i is **not biggest**, then it's not included in the maxpool output, and it has **no effect** – a zero derivative.

Example: If you're taking the maximum, and the largest number is 1000, it doesn't matter if the second largest number is 999 or 2.

Key Equation 69

The **maxpool derivative** is only nonzero for its **maximum** value.

$$\frac{\partial\,\text{maxpool}(A)}{\partial a_i} = \begin{cases} 1 & \text{if } a_i = \text{maxpool}(A) \\ 0 & \text{otherwise} \end{cases}$$

When we take all of the a_i derivatives and combine them into a **vector**, we realize that we have a **one-hot vector**, telling us which output was the **maximum**.

Example: Suppose a_4 was the largest out of 6 inputs in a column vector a.

$$\frac{\partial\,\text{maxpool}(A)}{\partial a} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \tag{9.35}$$

9.4.7 Maxpool derivative: somewhat similar to sign function (Optional)

Interestingly, when two values a_i and a_j are both max, **maxpool** behaves like **ReLU**.

If $a_1 = a_2$, and both are max, we have two cases:

- If a_1 decreases, it has no effect on the output: **derivative 0**.
- If a_1 increases, it increases the maxpool output directly: **derivative 1**.

So, the derivative is different moving left or right: it's **undefined**.

We'll ignore this edge case, just like we usually do for **ReLU**.

But what if we're not at the edge case, and a_i is our max value?

- Well, **decreasing** or **increasing** a_i will have a 1-1, linear effect, until we reach the **second-largest** term, a_j.
- Then once we're below a_j, a_i it has no effect.

We still see that, for any one particular a_i term, maxpool behaves like a shifted ReLU, and its derivative like the step function.

Key Equation 70

The derivative of maxpool for a particular a_i term behaves like the **step function**.

- The transition from 0 to 1 occurs at the **highest a_j term, excluding a_i**.

This is true regardless of whether a_i is currently the max.

By integrating, we see that **maxpool**, then, behaves like a **ReLU function**, shifted on the input/output dimensions.

9.5 Terms

- Connected
- Fully Connected
- Flattening
- Spatial Locality
- Translation Invariance
- Window
- Dot Product (Review)
- Filter
- Convolution
- Cross-Correlation
- Padding
- Dot Product Generalization
- Filtering
- Tensor (Review)
- Filter bank
- Channel
- 3-tensor Filter
- Linear Layer (Review)
- Convolutional Layer
- Weight sharing
- Stride
- Max-pooling
- Receptive Field
- Functional Layer
- Convolutional Neural Network