CHAPTER 7

Convolutional Neural Networks - Supplementary Notes

Unofficial notes by Shaunticlair Ruiz

7.1 Introduction

7.1.1 Fully Connected Networks

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

They're called "fully connected" because between two layer, every single unit in one layer can affect every single unit in the next layer, via some weight.

Definition 1

A **fully connected** layer is ones where every number in the **input** is connected (by a weight) to every element in the **output**.

For example, unit a in the first layer is multiplied by $w_{a,b}$ to affect unit b in the second layer, for every single a^{th} input and b^{th} output.

In this case, we have as many connections as possible, so we use this when we don't know enough about our problem: we let the machine decide how to handle it.

But, this isn't always the most effective way to structure our algorithm. To show why this is, we'll introduce a particular type of problem: image processing.

7.1.2 A different type of problem

For example, we might think of face recognition, or the vision used by self-driving cars.

We'll simplify for now by assuming we're working with black and white images. Let's start by thinking about how we'd put our image into our fully connected network.

Our image has the dimensions $(r \times k)$: r pixels wide, k pixels tall.

So, we list our pixels out in a single line of rk inputs: this process of turning a 2d input into a 1d input is called "flattening".

Definition 2
Flattening: Taking an entire **matrix** of inputs, and turning them into a single **vector**, by listing all the numbers in a single line.

7.1.3 What's wrong?

Here, we present two issues:

- First: our computer has no concept of pixels being near or far from each other, except in the same row.

  If two pixels were on top of each other, the computer loses that information when you line up the pixels. Are they diagonal? How close are they?

  Arguably, the computer could figure out the shape eventually. But, that's time wasted, and could create a less robust solution.

  We want our computer to know how the 2d **space** in our image works, so it's "spatial".

  Specifically, we want to know what pixels are nearby- which ones are **local**? We're talking about "locality".

  The combination of these two is

  Definition 3
  Spatial locality asks: Which pixels are **near** each other in **space**, and how?

  How big a block of pixels do we need to know in order to find, say, a cat?

  These kinds of questions fall under "spatial locality".

- Second: imagine if your computer is looking for something: maybe a cat, for example. If you move the cat within the image, it should still recognize it, right?

  The problem is, in a fully connected network, it doesn't!

  If it finds the cat in the top-left of one image, that includes different inputs than if it finds the cat in the bottom right of the image.

We want our computer to know that these two are the same: meaning nothing *changes*. Something that doesn't change is "invariant".

The shift we just described (sliding the cat around the image) is called a "translation.

So, the combination here is

Definition 4

Translation Invariance: The pattern you're looking for is the **same**, no matter **where** in the image you look for it.

We'll come up with a strategy to handle both of these problems.

7.2 Filters - Introduced

7.2.1 The pieces of our solution

Let's figure out how to solve those two problems at once - we want to do a calculation that doesn't depend on location, for example.

So, we'll make a smaller calculation at multiple different positions.

And we want this calculation that gives an idea of "locality": which pixels are near each other.

The solution is to do a calculation over a small region of an image (local) and repeat it all over!

That might not be entirely clear right now, but we'll get into the details.

7.2.2 Looking for patterns

We call this calculation a filter. Here's how it works:

Let's say we're looking for some pattern; we could say we're "filtering for" this pattern.

So, we'll take each section of the image, and look for this matching patterns.

We need an idea of how similar these patterns are. We're not sure how to do that... let's try to do a 1-D example, for inspiration.

This isn't an entirely useless example, either - when you want to do sound processing, you'll have 1D data, just like this!

7.2.3 The 1D case

We have some string of numbers that's length r, and a pattern we're looking for, length n. So, we'll look at every length-n segment of our input.

Which means, we'll be comparing two length-n lists of numbers, i.e. vectors.

This might be registering something familiar, or maybe it isn't - but, it turns out, you can measure how similar two vectors are using the dot product!

The dot-product, as we've seen in cases like linear classification ($\theta^T x$, for example), has two interesting ideas we care about.

7.2.4 Dot Products

To make explaining easier, let's assume i represents the dimension we're focused on, and v and w are our two vectors.

- The dot product can be calculated by multiplying together the each dimension of each vector ($v_i w_i$, $v_j w_j$, so on) and adding all of those together

- This operation can sort of measure how "similar" these vectors are

The "similarity" idea makes the most sense if you imagine we have a binary input with 1's and -1's:

- if you multiply 1 by 1, you get 1, and the two vectors are the same in that dimensions.
- if you multiply -1 by 1, you get -1: the two vectors are opposite (most dissimilar!) in that dimension.
- if you multiply -1 by -1, you get 1, the two dimensions are more similar again.

So, if our dot product is more positive, then our vectors are more similar. The same general logic works (sort of) for non-binary outputs.

7.2.5 Finally, 1D filtering

Back to the example we're working on: one of these two vectors is the pattern we're looking for.

So, this dot product is a good measure of how similar this piece is to that pattern! That's exactly what we're looking for - a pattern detector.

So, for every substring inside this input, we'll calculate this measure of similarity - the resulting new vector is our output.

If all of the number are too positive or negative, maybe we'll add a bias, where we either subtract from or add to our dot product.

This procedure is called **convolution**.

Definition 5

Convolution: the process of looking for a **pattern** in an image (or any signal) by **sliding** it over the input, and repeatedly doing a **dot product**-like calculation.

7.2.6 A comment about our output

Notice that, in this case, our output smaller than our input: because our filter can only go as far right as our vector allows, our length has shrunk by $n - 1$.

Why that number? Well, if we have a length-1 filter, we're all good. If we have a length-2 filter, our filter can start on every input, except for the very last one, so it has shrunk by 1.

So on and so forth: every element after the first of our filter "blocks" the start of our filter from using that element.

How do we deal with that? Well...

7.2.7 Padding

One very common solution is to do something called "padding".

The idea is: our output shrank because our filter is limited by how long it is - it can't shift past the matrix; it has nothing to multiply with.

So, what if we... let it stick out the side?

How do we do this? Well, the reason we can't shift the filter outside the matrix is because we are missing some values.

So... we'll just add some extra values to the edge of the vector.

We can pad with whatever values we want, but most often, you pad with 0's. You can also adjust how many 0's you pad with, depending on how big you want your output to be.

Definition 6

Padding is when you add filler values (usually 0's)to the **edge** of your input so your filter can scan over the whole input, without being stopped by the edge.

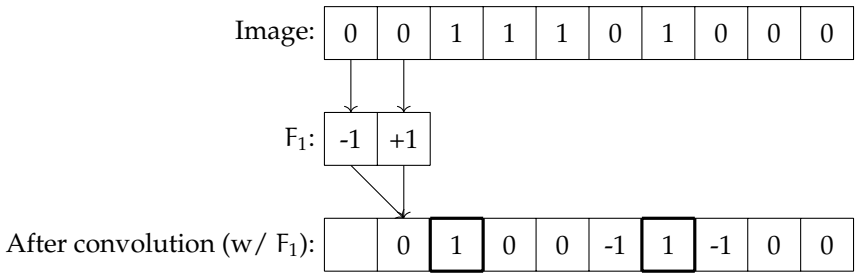
7.2.8 A specific example - a length-2 filter

Here, we introduce a concrete example. We have a filter of size two, and take a 1-dimensional, binary "image" to start off with.

First, our first filter is $F_1 = (-1, +1)$.

This filter looks for a very simple pattern: a smaller element on the left, and a larger element on the right. You could think of this as the numbers "increasing" from left to right.

Then given the first image below, we can "convolve" it with filter F_1, like so.



As shown above, we do this operation for every pair of numbers in the image, to get the post-convolution result.

Notice that this calculation detects exactly what we're looking for - any time we have an increasing sequence of two numbers, we get a 1 in the output!

7.2.9 Another example - a length-3 filter

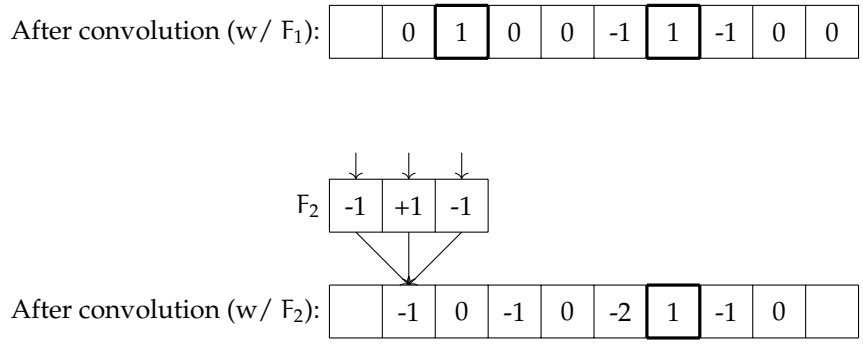
Let's try another example: to show that we can apply convolutions one-after-another, we'll use the previous output as our new input - our new image.

This filter will be length 3: we'll use $F_1 = (-1, +1, -1)$

This pattern looks like a point greater than those around it - you might think of it as a local maximum.

We'll pad one zero on the left (not shown below) to make our vector shrink by only 1, instead of 2.

So, we'll see how this filter does.



And again, it works! It only gives a positive result on the sequence $(-1, +1, -1)$, where we have a local maximum.

It also gives an especially negative result at a local minimum: $(0, -1, +1)$.

7.3 Filters - In higher dimensions

So, how do we extrapolate this to higher dimensions?

7.3.1 Multiplication in 2D

Surprisingly, the answer is pretty simple: we just do the same operation as before, but without the perspective of vectors.

Before, we did the dot product for each sub-segment. This means multiplying the components "element-wise" (multiply i^{th} dimension of each vector together), and adding together those results.

Now, we do the exact same: our filter is 2D, so we take every continuous 2D frame of our input, and multiply, element-by-element.

The same logic as before applies: if two elements are similar, we get a larger result. If they are more different, we'll get a smaller result.

7.3.2 Dot products, but... bigger?

Here's one way to think about this that might be helpful, might be confusing.

Break your matrix up into rows. For each of the rows, you're multiplying element-wise - like a dot product.

So, you're taking a dot product for each row, which gives the similarity for those rows.

Then you add up all these dot products ("similarities") to measure the total similarity between two matrices.

In a way, because we're still multiplying and adding, you might intuitively think about this as a "bigger" dot product.

7.3.3 Comments - Evolution

As a fun fact, these kinds of 2D filters appear in the visual part of mammal brains - including yours!

This seems to show that this way of thinking about visual information is similar to how evolution settled on it - and so it probably does a pretty good job.

7.4 Filter Banks Pt. 1

7.4.1 More Filters!

One filter can encode lots of information, but different filters have different data: one might look for edges, maybe another can find circles, maybe a more complicated one can find a face!

What if we want to have different kinds of information, at the same time? What if we want to be able to combine these pieces of information?

You might think "well, I could use multiple filters", but, how do we do that? If we have multiple filters one after another, the first one may have removed information the later ones need.

So, instead, what if we use multiple filters at the same time ("in parallel") - we run each filter over the original image, and save each result?

Each of these calculations creates a new matrix output, which we store separately. We end up with a bunch of matrices as our output. Each of these matrices is called a **channel**.

Definition 7

Channel: the result of using(convolving) **one filter** on an **image**. If multiple filters are used, we can end up with many channels.

7.5 How to store matrices - Tensors

7.5.1 Our goal

Now that we have all this data to store, we need to be careful how we work with it - this is a lot to keep track of, after all.

In order to do that, we'll introduce what's called a "tensor".

7.5.2 Building Up

Consider the following review of what we know so far.

- If we only have one number, we store that as a **scalar**.

  This scalar can be thought of like a single point - since there's no axis along which we can move, sometimes this is called "0-dimensional".

- If we have a sequence of numbers, we store that as a **vector**.

  You can do this by stacking scalars on top of each other in a line.

  You can also think of this as a 1D object, where the "dimension" is the index we can move along to get different scalars.

  In a way, we have a "line" of numbers, at least visually.

- If you have several sequences of numbers, you can store that as a **matrix**.

  You can do this by stacking vectors next to each other. If you break your matrix up into rows or columns, you get a bunch of vectors!

  This is a 2D object: you can move along the rows, or the columns: each of these is a dimension.

  We have a grid of numbers, or maybe you can imagine it as a square or "plane".

- What if we want to stack several matrices on each other?

  This would create a sort of 3D object, where we have rows, columns, and height.

  Visually, you might imagine the 1d case as a line of numbers, the 2d case as a square of numbers.

  Now, we have what you could think of as a "cube" of numbers.

  This cube of numbers, this 3D object, is what we call a **tensor**.

Definition 8

A **tensor** is the more "generalized" version of a matrix: it's just a **"box"** of numbers in some **dimension**. In our case, our **tensor** is **cube** (or cuboid) of numbers.

As you see in the definition above, a tensor is technically any version of this kind of object, in any dimension. A scalar, vector, and matrix are all tensors.

But, for our purposes, we only care about what is called a **3-tensor**, or a 3D box of numbers. The one that looks like a **cube**.

7.5.3 An important warning: dimensions

Before we continue, we should make a quick warning about dimensions.

You might have noticed that the way we are using the word "dimension" here is different from how we have used it in other parts of the course.

For example, if you have a vector that has 3 elements, you could call that a "3D vector", because you can imagine it as an arrow in 3D space.

But above we said vectors were "1D" objects. What gives?

This is because, unfortunately, there are different ways you can talk about "dimensions".

All have some different mathematical reason behind them, where in some way, they are related, **but these definitions are different and you should be careful.**

Let's get into that.

7.5.4 Dimensions, Clarified

We'll break this down. Dimensions are about asking the question, "how many separate ways can I change what I'm looking at?"

For example, we live in 3D space because we can vary our position in the x, y, and z directions.

A paper is 2D because there are only two separate ways to move on the paper.

Based on this, we can think of two definitions for dimensions:

- Version 1: If we drew our vector in space, how many ways can we vary the direction it points in?

  Or, in other words, how many *dimensions* of space are we thinking of when we draw our vector?

  This definition focused on vectors, and each number in that vector as a unique dimension.

  In this perspective, we imagine that we can vary each scalar in our vector to create a different vector, and we have n scalars that we can vary.

  To summarize:

  - A scalar is a 1D vector.

  - A vector with 2 numbers is 2D by this definition.
  - A (4×4) matrix (4 vectors, each in 4D), is 4D.

Definition 9
One way to define **dimensions** is to ask how many different **numbers** (scalars) we have in a **vector**.

- Version 2: How many axes of numbers do we have to work with?

  This definition is more abstract, and talks about the dimension of what we're storing our numbers in.

  In this perspective, we imagine that we can vary the index on each axis (row, column, height, etc.) to find a different scalar, and we have k axes we can vary.

  To summarize:

  - A scalar is a 0D object.
  - A vector with 2 numbers is 1D by this definition.
  - A (4×4) matrix (4 vectors, each in 4D), is 2D.

Definition 10
Another way to define **dimensions** is to ask how many **axes** our **box** of numbers, or **tensor**, has.

Now that we've clarified the difference, back to tensors.

7.5.5 Back on track: What's a tensor?

Using what we've carefully built up, we have everything we need to understand the basics of tensors:

A tensor is, more or less, a bunch of matrices of the same shape stacked on top of each other.

You can visualize it as a **cube of numbers**.

Not too bad, right?

7.5.6 Tensor Math (Optional)

We went through all this work to carefully build understanding because tensors are not easy, but they are very, very important.

Unfortunately, just like how matrix multiplication, algebra, and calculus have their own special operations and rules, so too, do tensors.

However, simply put: tensor multiplication and calculus are really hard, so we're not going to do it. We'll let the computer handle it.

The rules involved are incredibly general and very, very powerful.

Physicists, computer scientists, mathematicians, and other smart people solving hard problems have to grapple with the details for years, sometimes, to understand everything they need.

We do not have that time. So, we'll leave those details to the computer.

7.5.7 What about higher dimensions? (Optional)

One little plot twist: what if we need something even bigger than a tensor?

What if we need to stack multiple tensors, for an even more complicated problem?

Well, thankfully, tensors actually include things higher than 3 dimensions!

If you have a "4D cube of numbers", that still follows the rules of tensors.

Tensor math can be applied for any higher dimensions. So, if we were to ever need it, our computer knows how to handle it.

7.6 Filter Banks Pt. 2

7.6.1 Review

Now that we know what a tensor is, let's pick back up where we left off:

We wanted to convolve multiple different filters with our image at the same time ("in parallel"). We call this collection of filters a "filter bank".

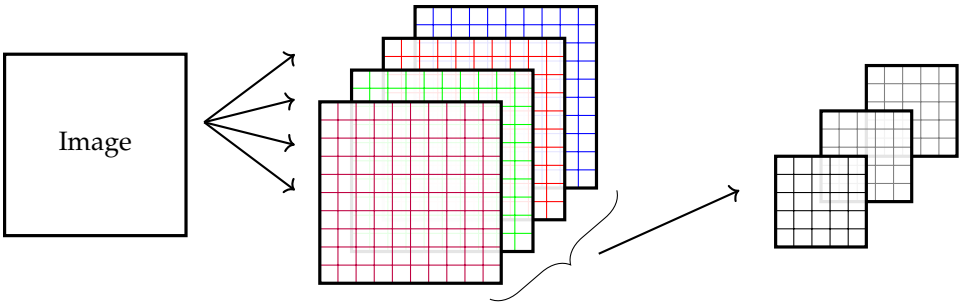
Definition 11

A **filter bank** is a collection of **filters** used on the **same image**.

(Side comment: notice you can store this filter bank as a tensor too! Since each filter is a matrix, and you can just stack them).

7.6.2 Using filter banks

The output after applying all these filters is a tensor: each filter outputs to one "channel", which is what we call the third dimension of our tensor (row, column, channel)



This diagram loosely shows the process: the first object on the left is our image.

The many arrows represents that the image is applied to each of the filters in multi-color filter bank. Finally, each of those outputs is combined into the tensor on the right.

If we apply k filters, we will end up with k channels: one for each filter.

So, if a single filter resulted in a $(r \times r)$-sized matrix, then the filter bank results in a $(r \times r \times k)$ tensor.

7.6.3 An example: Using a Filter Bank

Let's show an example of 2d filtering, with a filter bank.

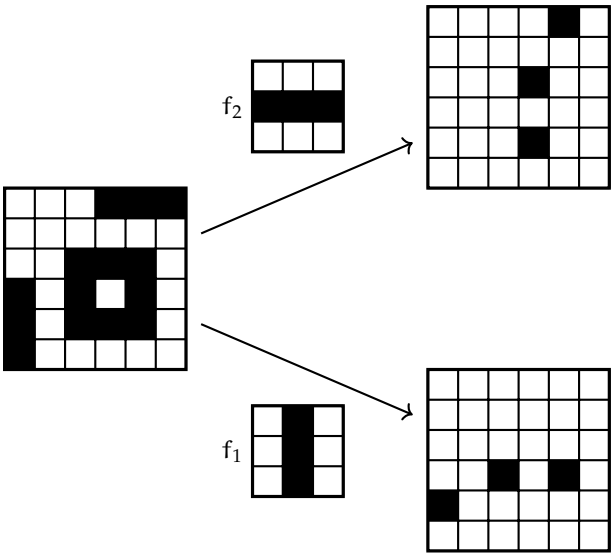
Our filter bank has two filters, f_1 and f_2. So in this case, $k = 2$.

Each filter is (3×3) in size. So we preserve the size in our output, we'll add 1 layer of 0-padding all the way around our input matrix (not shown).

As mentioned before, each filter will be convolved with the input image, "looking" for the pattern they represent.

In this case, we see that f_1 seems to be looking for a vertical line, and f_2 is looking for a horizontal line; both 3 pixels long.

Our input image is $(n \times n)$, so our output of the first step will be an $(n \times n \times 2)$ tensor.



Notice that the top output puts three dots where it finds horizontal lines(you can match them up visually!)

The bottom output puts three dots where it finds vertical lines. So, it seems to be working!

7.7 Tensor Filters

7.7.1 What now?

Now we have used multiple filters, and we got a tensor... great!

But... what on earth do we do with a tensor?

7.7.2 Motivation

So, the whole reason we used multiple filters, was so we could look for multiple patterns at the same time.

Often, the goal of your trained machine is to recognize certain shapes.

It may be easier to look for smaller, simpler shapes first, and then combine those results to look for bigger shapes; and so on.

For example: imagine you're looking in an image for a child's depiction of a house.

You might look for the signature rectangle on the bottom, and a triangle "roof" on top. These are two distinct features you can find with different filters.

So, using our previous steps, we have a filter that finds squares, and one that finds triangles.

But what about combining them?

7.7.3 Filters, again!

The solution is to use another filter!

But this time, we need to go up a dimension again.

We went from 1D filters to 2D filters because we wanted to look for 2D patterns, and we did this by multiplying element-wise as we moved across the grid.

Now, we have a 3D tensor we're analyzing.

In this case, we want to find a point on one layer (representing the roof), closer to the top of the image than a point in the next layer (representing the "body" of the house)

So, if this is the pattern we're looking for, we could replicate this pattern in a new filter! We'll create a tensor that represents this pattern, and again, slide this pattern around the image.

For simplicity's sake, we'll assume this tensor has the same number of channels as the input, so we only have to slide it around the image, and not through the different channels (doesn't that sound complicated...)

Thus, we have invented the "tensor filter". All the same math from before (multiplying element-wise, sliding around the image, maybe padding...) is the same.

7.7.4 An example

We'll take the output from our earlier example and run a tensor filter over it.

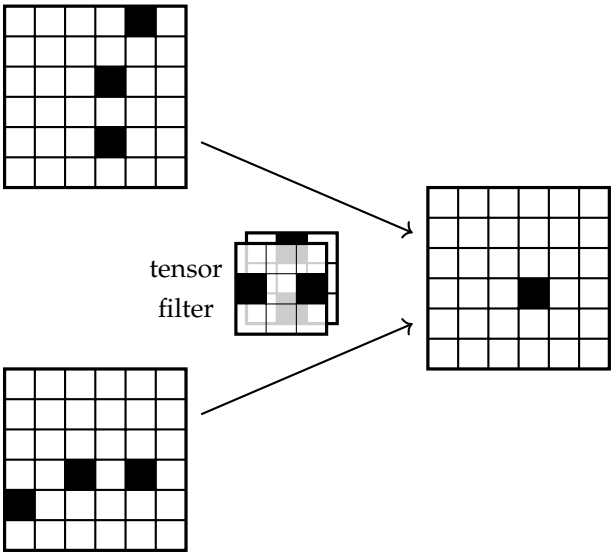
Let's say the output we're looking for a "circle", or really, a (3×3) square with a hole in the center.

This is gotten by having two vertical lines on the sides, and two horizontal lines on top and bottom of the section of the image.

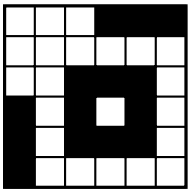
So, we'll create a filter so that's exactly what we're looking for!

One channel will look for the two horizontal lines, the other will look for the two vertical lines.

Remember that below, the top image gave us horizontal lines, and the bottom image gave us vertical lines.

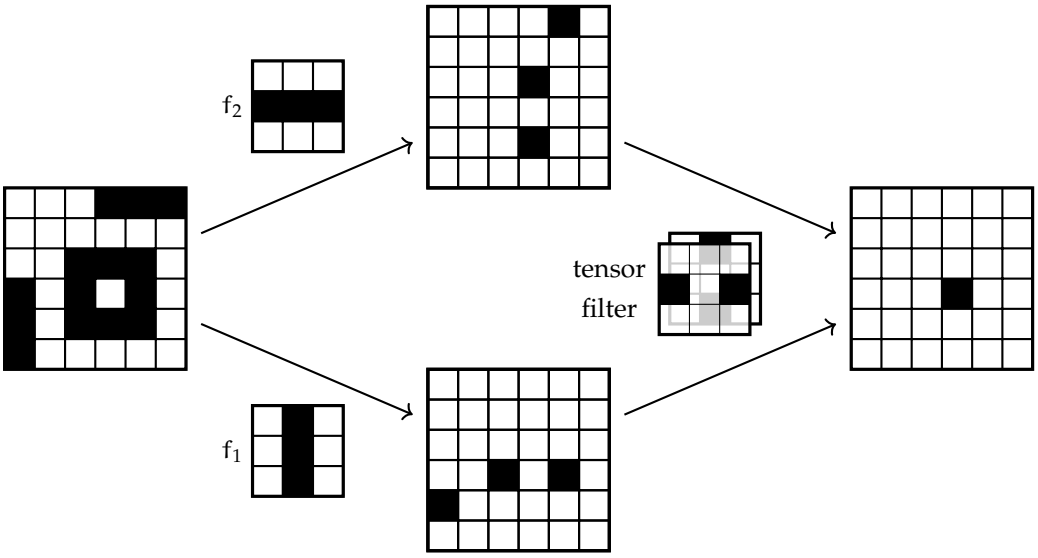


Like magic, we get exactly what we're looking for! Notice that it has a dot in exactly one place. If we go back to the original...



We have our dot right in the center of our empty square/"circle"! We can see that the process works.

Here's what the whole thing looks like when we put it together:



We can see the full process of running two separate filter banks, and getting the results we're looking for.

7.7.5 Really small side comment: RGB color (Optional)

If your image is RGB (made up of red, green, blue), you'll need three values at every pixel: this is most easily described with a tensor!

So, if you have an image that is $(n \times n)$ in size, the RGB representation will be a tensor of the shape $(n \times n \times 3)$.

7.8 Machine learning and Convolution

7.8.1 Applying to ML: Layering

Now, we have learned how to use convolution to process information.

We can see it's good for finding patterns, so it makes some sense we'd want to add that ability to our network.

So, let's try that: how do we use convolution in our neural network?

Well, because our network is modular (broken up into separate pieces, or "modules", that can be thought of separately), the process isn't too painful.

All we have to do is insert a new layer of convolution into our neural network.

It takes in inputs from whatever the last layer gave, and sends its output to the next layer after.

In fact, our example above shows two consecutive layers of convolution.

This can be used to build larger features (circles) from smaller ones (lines).

7.8.2 Thinking in ML terms

Each filter bank we use will represent a single layer.

If we want to think in the terms of ML, we can think of each number inside a filter as a "weight": a number used for calculation we can adjust.

As we mentioned early on, we might also have one bias for our filter - making the output of our element-wise multiplication more positive, more negative, etc.

7.8.3 Training Filters

In the past, machine vision experts would hand-craft their filters, and experiment with them.

However, because we can think of our filters in terms of their weights, we can use gradient descent to train them!

As long as we can take our derivatives, we can use gradient descent to find better filters. Our computers know how to do this.

Sometimes, these filters can even tell us about the structure of the data!

7.8.4 Benefits of Convolution

This convolution has a lot of benefits. Two of those were the original motivations for trying out convolution:

- **Spatial Locality**: A filter tells you about what the "local" area around a pixel looks like.

  Thus, it can give the neural network the ability to associate that information.

- **Translation Invariance**: A filter also, by virtue of being slid around, will find the same pattern no matter where in the image it appears.

  This means the neural network can now more easily tell that an object in the bottom right is often the same as an image in the top left.

But there's one more surprising benefit: efficiency.

7.8.5 Efficiency of Convolution: Fewer Weights

The interesting detail: a filter is (typically) much smaller than an image, and each pixel in the filter only has one weight.

So, rather than having many weights for every pixel in the input, we have a small number of weights for our filter.

On top of that, these weights are reused multiple times during convolution, as we slide around the image.

So, we have a much, much smaller number of weights to train.

Because we are "sharing" weights between different calculations, we call this benefit **weight sharing**.

Definition 12

Weight sharing is where the **same weights** are used for **multiple calculations**. Thus, the model trains **faster**.

For example:

if we have a (5×5) image, and want a (5×5) output, using a fully connected layer, we would need 25 inputs, 25 outputs.

This creates $25 * 25$ weights, and $25 * 1$ biases: 650 parameters.

If we have a (3×3) filter, and we use length-1 padding all the way around the image, we can get the same size output.

This has $3 * 3$ weights, and 1 bias: 10 parameters.

And this efficiency grows the larger your input and output are.

This does mean we have to be careful when training, though!

7.8.6 Variable definitions

Let's carefully define this with some variables. Let's say we place our filter bank on layer l.

Note that we'll assume the image and filter are both square (same number of rows and columns).

This isn't necessary, but makes our definition simpler.

Additionally, remember that exponents in this notation only represent which layer you're referring to.

Our variables:

- **Length** of the square **input**: n^{l-1}

  We use $l-1$ instead of l because this is technically the **output** of the previous layer.

- **Number** of **filters**: m^l

- **Length** of each square **filter**: k^l

- **Padding** around the **input**: p^l

  Remember that we usually pad with the number 0.

For each filter, we also have one additional bias term.

To use this bias, we do the element-wise multiplication, add up the terms, and *then* add the bias.

We have one bias per filter, or m^l total biases in a layer.

7.8.7 Shapes

Now, we need to find the shape of each object.

If we have only **one filter**, we just end up with some squares:

- Our filter: $(k^l \times k^l)$
- Our input: $(n^{l-1} \times n^{l-1})$

What if our previous layer had filters, though?

Well, if our previous layer had m^{l-1} filters, then it would give the output (our new input) a third dimension of m^{l-1}.

We mentioned before we only want to slide our filter across rows and columns, so the third dimension needs to match.

Thus, we need our filter to have as many channels as these previous output: our third dimension is m^{l-1}.

Thus, our shapes are

- Our filter: $(k^l \times k^l \times m^{l-1})$
- Our input: $(n^{l-1} \times n^{l-1} \times m^{l-1})$

7.9 Output dimensions

Let's round out some other important considerations related to size. Specifically, we'll look at things affecting the output dimensions.

7.9.1 Filter Size

Recall that, in the 1D case, we showed that our output (vector) dimension is decreased by $k - 1$.

This was because every element of the filter after the first prevented the filter from moving further to the right.

We can extrapolate the same concept to multidimensional inputs: each dimension is shrunk by $k-1$.

So, an $(n \times n)$ input turns into a $((n-k+1) \times (n-k+1))$ output.

For the rest of the problem, we shall refer to n- squares to mean squares of length n, rather than writing out the full dimensions.

So in this case, the length is

$$n^{l+1} = n^l - k^l + 1$$

7.9.2 Padding

We have mentioned padding, but it's important to remember how this affects the input and output.

If we add padding symmetrically all the way around the input, then we add p 0's to the left of our image, and p 0's to the right.

So, the dimension increases by $2p$.

Thus, we can say our n-square has its length changed to

$$n^{l+1} = n^l + 2p^l$$

7.9.3 Stride

Here, we introduce a new concept: stride.

When we convolve normally, we move our filter over by 1 for each step.

But, we could also move over by r for each step. Basically, we "skip over" several pixels each time.

A stride of 1 means we move our pixel over by one each time, same as normal.

A stride of 2 means we move our pixel over by 2 each time: you "skip" every even pixel.

In 1D, you would convolve with pixel 1, 3, 5...

Definition 13
Stride: How many **pixels** you **slide** over with each **o**f convolution.

As you can see, we skip one pixel each time, the size of our output image becomes half the size.

So we start by guessing we get a square of length n/s.

We have to round, though: if we have a stride of 2, and 5 pixels, that would imply a length 2.5 square.

But as we can see above, if we have 5 pixels, we have 3 pixels to convolve with.

Rounding up has its own notation: if you want to round up x, you write that as $\lceil x \rceil$

So, we get a square of length

$$n^{l+1} = \lceil n^l / s^l \rceil$$

7.9.4 The convolution output size

We combine all three of these factors to get a single equation.

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

We get this result (which is important, and worth saving!):

$$n^{l+1} = \lceil (n^l - k^l + 1 + 2p^l)/s^l \rceil$$

7.10 Max Pooling

7.10.1 Gathering information

As we've alluded to, convolution can be used to get basic features, which we combine into larger features later.

Whether lines into circles, or shapes into childlike houses, we often want to combine these patterns.

However, as we gather this information, it might make sense for us to shrink the output.

Why? Well, there are going to be many more small, simple patterns (like edges) in an image than large, complex ones (like a house).

This is natural: if it takes many edges to make one house, there will be fewer houses than edges.

So, combining our outputs into a smaller image let's us get a "bigger picture" on whatever is going on.

7.10.2 How to combine patterns

Our first thought might be to increase the filter size, or increase the stride size.

However, these both present problems: we may not want a larger filter, and using one might focus on patterns we don't care about.

Meanwhile, even though stride size makes the image smaller, you lose some information:

If the pattern is between two of your stride steps, for example, you might miss it. So, greater stride doesn't collect all the information over that range.

What we'll do instead is a new tool called **max pooling**.

7.10.3 What is max pooling?

Max pooling is a tool similar to ones we've used before, but with its own twist.

It behaves like a filter: we pick a size we're looking over, do a calcuation, and then move our "window" around all over the input.

Sometimes we call the area that we're currently applying our filter to (to get the maximum value) the "receptive field" or just a "field".

So, we get the maximum value over our "receptive field".

Definition 14
Receptive Field: the **portion** of our input we're currently **applying** our **filter** to.

In this case, however, the calculation is different: all we do is **return the largest value over a certain range**.

Definition 15
Max pooling is the process of looking over our **field** and returning the **maximum** value over that region.

For an example: consider a (3×3) max pool window.

We would take a 3-length square and slide it around our input image, and for every position, we put the maximum value in a new output matrix.

7.10.4 Considerations for Max Pooling

A few comments:

First, notice that, since we're taking the maximum, there are no weights: the layer always operates in the same way.

In fact, there is no bias, either.

You can think of this kind of like a ReLU layer, which also has no weights, because it only relies on a simple function.

Because of this, we might call it a "pure functional layer".

Second, it has filter size, so it decreases size according to its size and stride, just like a regular filter.

Finally, remember that the goal of our max pooling layer is meant to aggregate information. So, we put two restrictions:

- Our stride $s > 1$, so we end up with a smaller output.

  Hopefully, this is how we collect our information together.

- Our filter width $k \geqslant s$. This is so we don't skip over some values.

  To understand why, consider that if we had a stride of 5, and a filter width of 1, then we would never touch a large part of our input!

  We could miss crucial information that way.

Because we're only taking the maximum over a broad area, we lose some of the precise information about where a pattern was found.

This can take away some information, but it lets our machine learn to find patterns regardless of where they are.

7.10.5 An example

For example, let's take a matrix of size $(64 \times 64 \times 3)$.

We'll set our filter to size 2, with stride 2.

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

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

So, our output dimensions are $(32 \times 32 \times 3)$.

7.10.6 Concerns about Max Pooling

One problem with max pooling is that they don't perfectly handle "translation invariance", or, our algorithm not caring exactly where our pattern is.

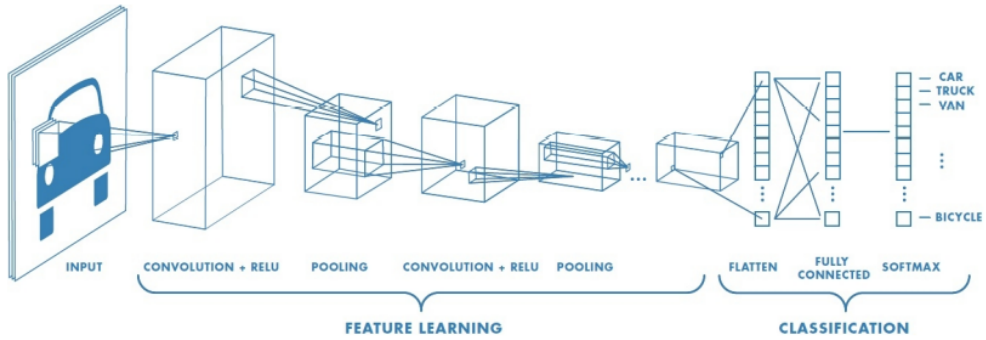
This is because, if your stride $s > 1$, then you could shift the pattern within that stride to change the max pooling output significantly.

This problem will not be elaborated further here, but here's a relevant paper (https://arxiv.org/pdf/1904.11486.pdf)

7.11 Typical architecture

7.11.1 An example of a CNN

Here, we give the form for a "typical" convolutional network:



The "depth" dimension in the layers shown as cuboids corresponds to the number of channels in the output tensor. (Figure source: https://www.mathworks.com/solutions/deep-learning/convolutional-neural-network.html)

7.11.2 Common features of CNNs

First and foremost: a CNN is just a regular neural network, with some convolutional layers, like filters and max pooling.

Here, we get into some common features:

- After each filter layer there is generally a ReLU layer
- there maybe be multiple filter/ReLU layers
- Max pooling is often used after filtering to shrink the output.
- Once the output is relatively small, we finish with a fully-connected layer
- We close out with our activation function. For example, we might use softmax in a classification problem.

There's very few hard rules for how exactly to design this kind of network.

There are no good guidelines, either in theory or in practice, to tell us how our choice affects how well the networks does.

7.12 Training our Network

7.12.1 Can we train it?

Given the subject matter of this course, "Machine Learning", your first question might be, can we train this, and if so, how?

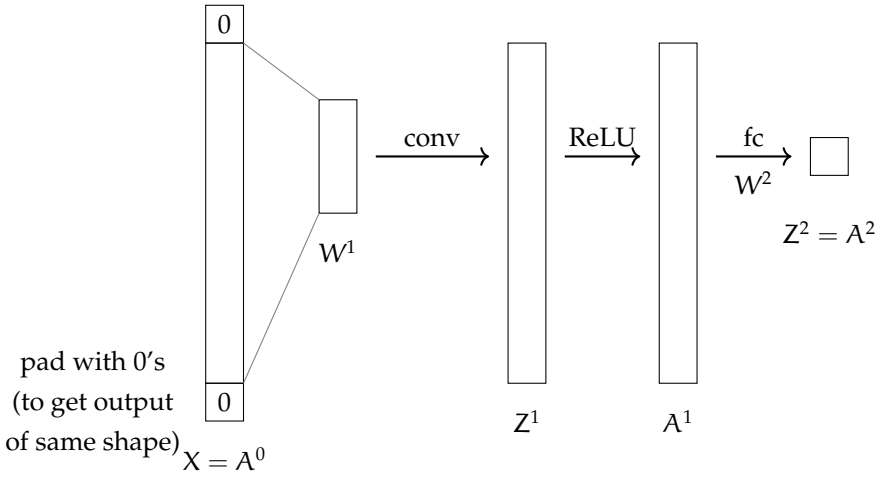
Well, thankfully, it turns out we can!

The various weights of filters and other components are (mostly) continuously differentiable, so we can return to our old friend, back-propagation.

(Technically ReLU and max pooling both don't have continuous derivatives, but we can usually get away with this)

7.12.2 An example

We'll try a simple example, just to understand how we can do back-propagation on convolutional networks.



In this example, we take our input, and run a convolution (getting Z^1), a ReLU (getting A^1), and then a fully connected network (finally, $Z^2 = A^2$).

We'll assume we have a 1D, single-channel input image, so its shape is $(n \times x1 \times 1)$.

We also have a single filter with no bias, of the shape $(k \times 1 \times 1$.

7.12.3 Forward pass - Convolution

For starters, we need to figure out the output of each sub-layer.

First, we want to apply the convolution.

The convolution, as always, involves taking some slice of the input, and multiplying it (in a dot product, in this case) with our filter.

Our filter weights can be represented with a series of numbers as W^1.

We take our input, X, or A^0, and decide to index into it.

We'll use python notation for indexing, using square brackets. We'll also assume our filter is odd, because that's more common.

We'll use i to represent us sliding our filter. If we want to index the right length, we can do it two ways:

- We index starting from the very left of our field.

  This would be $[i : i + k]$.

- We index starting in the middle of our field: we have an odd number on our filter, so we need to round down.

  Thus, we get $[i - \lfloor k/2 \rfloor : i + \lfloor k/2 \rfloor]$

Which version we use depends on how we handle indices after padding.

For simplicity, we'll use the former (note that the official notes use the latter; do with this what you will)

Thus, we're doing a dot product: we'll represent this the same way we normally do, when we're doing linear regression: $\theta^T X$.

Thus, we have

$$Z_i^l = (W^1)^T \cdot \{A^0[i : i + k]\}$$

7.12.4 Forward pass - The other steps

The rest of the steps are, thankfully, much simpler. They look more or less the same as before.

First, our ReLU step (remember that ReLU generally follows convolution):

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

Now, we do the fully connected step:

$$A^2 = Z^2 = {W^2}^T A^1$$

Finally, we get the square loss

$$L(A^2, y) = (A^2 - y)^2$$

7.12.5 Updating our filter

Now, how do we update our filter?

Well, we can use the chain rule.

First, we start at the end: how does our output, A^2, affect the loss, L?

$$\frac{\partial L}{\partial A^2}$$

We can write that with this simple derivative, and we have previously derived it.

Now, we move a step back. How does A_1 affect L?

We combine how much A_1 affects A_2, with how A_2 affects L.

After all, if we double the change in A_1, that should double the chance in A_2 (since the derivative pretends our function is linear), and thus doubles the change in L.

$$\frac{\partial L}{\partial A^1} = \frac{\partial L}{\partial A^2} \cdot \frac{\partial A^2}{\partial A^1}$$

We repeatedly move backwards, and get

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

7.12.6 Familiar Derivatives

Each of these derivatives except one is familiar.

We start with the square loss; we get it using the power rule.

$$L(A^2, y) = (A^2 - y)^2$$

$$\frac{\partial L}{\partial A^2} = 2(A^2 - y) \tag{7.1}$$

Then, we handle the fully connected layer. We just take the derivative normally, as if we were doing $\frac{d}{da}(wa) = w$.

$$A^2 = Z^2 = {W^2}^T A^1 \tag{7.2}$$

$$\frac{\partial A^2}{\partial A^1} = W^2$$

We finally handle the ReLU.

As discussed previously (check the matrix derivatives notes!), we get it as follows:

$\partial A^1 / \partial Z^1$ is the $n \times n$ diagonal matrix such that

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

7.12.7 Our New Derivative

(Note that this gets complicated.)

Finally, we have

$$Z_i^l = (W^1)^T \cdot \{A^0[i : i + k]\}$$

We take the derivative like we did for the fully connected network, but instead of with respect to a, we'll do it for w: $\frac{d}{dw}(wa) = a$.

$$\frac{\partial Z^1}{\partial W^1} = A^0[i : i + k]???$$

This is a bit confusing: what do we make of this?

Well, we'll use the rule built in the matrix derivative notes:

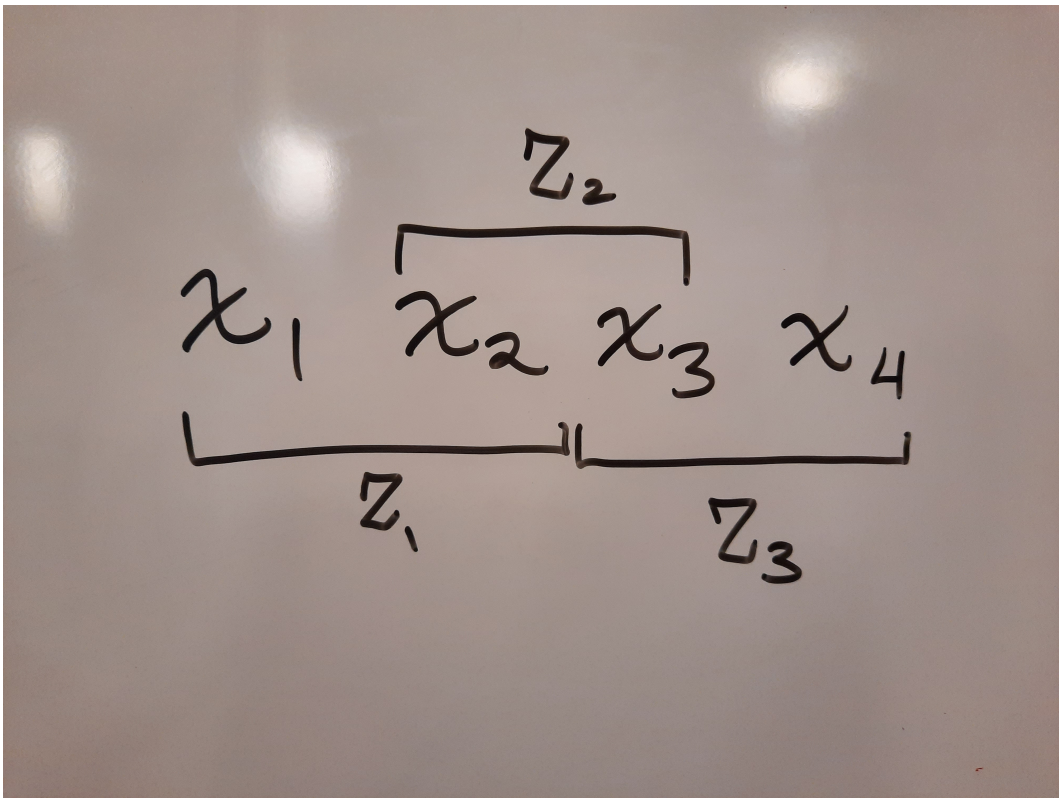
$$\left(\frac{\partial Z^1}{\partial W^1}\right)^T \Delta W^1 = \Delta Z^1$$

7.12.8 Derivatives: what's going on?

To make it clearer what's going on, we'll create a concrete example:

We'll do a 1D example. Our filter is 2 numbers long, and our input is 4 numbers long.

So, we'll get three outputs: z_1, z_2, and z_3. Like so:



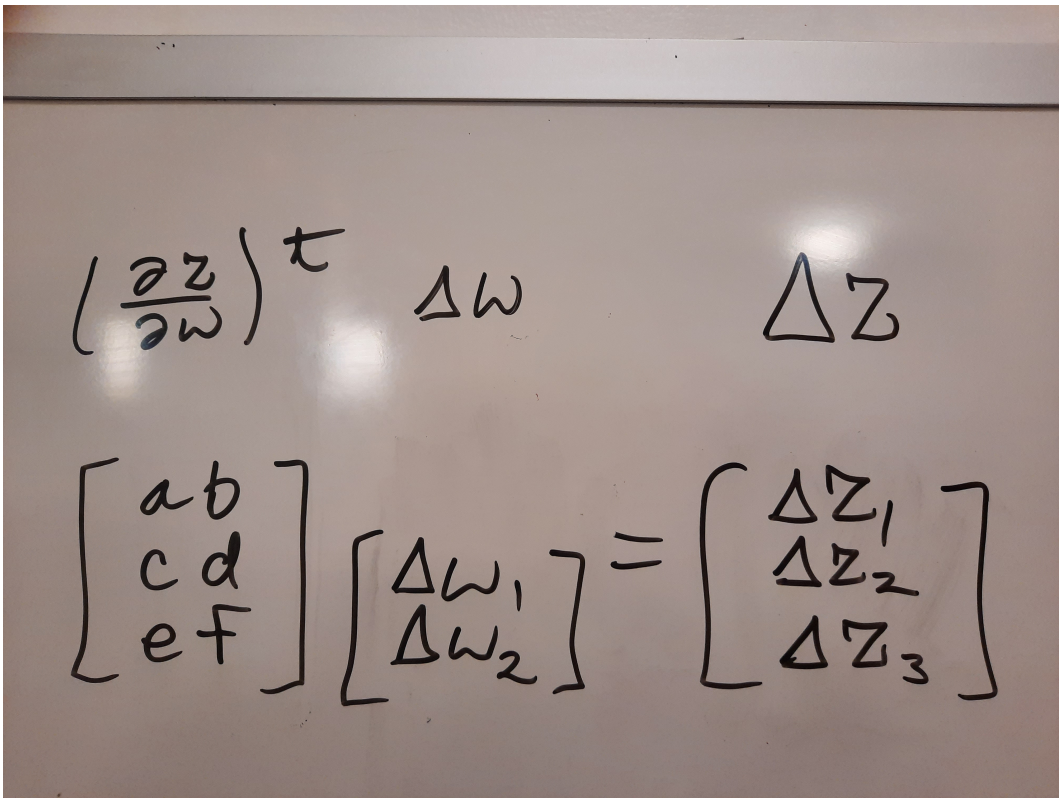
So our equations are:

$$z_1 = w_1 x_1 + w_2 x_2$$

$$z_2 = w_1 x_2 + w_2 x_3$$

$$z_3 = w_1 x_3 + w_2 x_4$$

Let's show what our matrix looks like:



We'll focus on the first row.

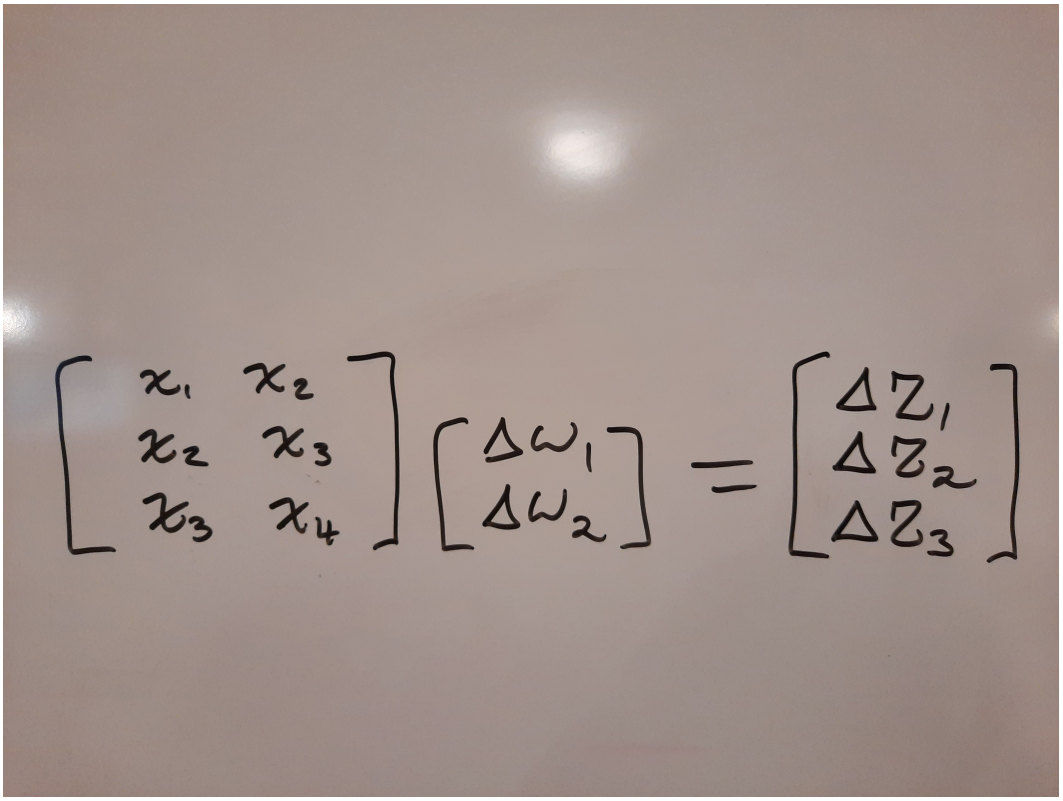
$$a\Delta w_1 + b\Delta w_2 = \Delta z_1 \tag{7.3}$$

Notice that only w_1 and w_2 affect z_1.

How much w_1 affects z_1 is proportional to x_1. Same for w_2 and z_2. So, we get

$$x_1\Delta w_1 + x_2\Delta w_2 = \Delta z_1 \tag{7.4}$$

We can use the same logic for each of these. We get



This sort of pattern can be used for all of our derivatives.

So, each row is represented by $A^0[i : i + k]$, where i is given by the row.

This explains why it appears in the derivative we calculated before!

So, now we just take the transpose, and this gives us our derivatives.

Thus, our derivative is

$$\frac{\partial Z_1}{\partial W_1} = [A^0[: k], A^0[1 : 1 + k], ...] \tag{7.5}$$

Where $A^0[i : i + k]$ is a column vector.

We have k rows, because that's the length of $A^0[i : i + k]$.

We have n^{l+1} columns - the number of outputs we expect.

So, the shape of our derivative is $(n^{l+1} \times k)$.

7.12.9 Buffer

7.13 Backpropagation in a simple CNN

7.13.1 Max Pooling

One last detail: how do we handle max-pooling?

Well, our function is

$$maxpool = \max(a_1, a_2, a_3...)$$

If a_1 is largest, then this simplifies to

$$maxpool = a_1$$

So, if we take the derivative with respect to each component:

$$\frac{d}{da_1} maxpool = 1$$

While

$$\frac{d}{da_2} maxpool = 0$$

This means that the back-propagation will affect a_1 and all of the parts of the network that fed into a_1.

Meanwhile, there will be no change to any other a_n value, or anything that feeds into it.

So, that's how we handle derivatives for maxpool.

Thankfully, most of this mess is handled by existing software.