**Convolutional Neural Networks**

Convolutional Neural Networks are mainly used for image detection/identifiation problems. First I’ll review convolutions.

**Convolutions**

Convolutions are kind of like wavelet transforms. It’s one way to reduce the dimensionality of a data set, but also extract the import features of the (image) data set.

So say we wanted to classify an image as being either an X or an O.

A picture containing square, rectangle, pattern, tile

Description automatically generated A black and white checkered pattern

Description automatically generated with low confidence

Since the image is 6×6 = 36 pixels, we could construct and train a classification neural network with 36 input nodes. Each node would take on the value black or white. In a real gray scale image, each node would have a pixel value between 0 and 255. We’d scale them so that the min/max values are -1/1 though. So for instance, these two pictures would be:

A black and white crossword puzzle

Description automatically generated A black and white crossword puzzle

Description automatically generated

And we could connect the 36 input nodes to some variety of internal hidden layers, and an output node(s) to figure out the identity of a given image. But for larger images, with thousands/millions of pixels, this brute force method would be too unwieldy. But to distinguish an O from an X, we don’t need all that information anyway. This is kind of like how we can use thermodynamics to effectively describe a system with a reduced set of variables (S, E, V, N), rather than recording the positions and momenta of all N ~ 1023 particles. Seems to me that we could describe these things with a reduced set of variables comprising, say, various moments of the pixel density distribution function. But instead, what we do is this:

**1**. **Convolution w/ Filter**

First we take the convolution of the image with a given filter/kernel, like this one for example.

A number in a grid

Description automatically generated with medium confidence

We start by overlaying the filter, on top of the image, I, in the upper left corner. [Parenthetical note: there are variations on this. It is common to first **pad/frame** the image with 0 pixel layers. And we can make the padding any number of layers thick. It obviously wouldn’t make sense to make it thicker than the filter itself though. Padding is useful because that way we get more information about the edge of the image.] And we calculate a number to determine how well the filter matches the part of the image which it overlays. We can could add up all the times the filter and image had the same pixel values. But instead we just take a straight ‘dot’ product between the filter values and pixel values. I think this is called the ‘feature’. FWIW, it looks like we also throw in a bias, b. And if we slide the filter along the image, then we will get a feature map, whose values are given by:



(implicitly assumed that stride = (1,1) here) If we have (r,g,b) values for each pixel, instead of just a single grayscale number, then I’m guessing we’d scale r, g, b each to the range (-1,1), and then take the dot product between the Image pixel vector and Filter pixel vector, and throw in a bias.



Typically the filters for the different color channels are different. Apparently these weights (and bias) are learned as part of the process of minimizing the loss. Could write this using vector notation for short:



Moving on…And we see that F­mn will be maximized when the feature is present in the filter. So for instance, the first four overlaps, below,

A picture containing square, rectangle, tile, mosaic

Description automatically generated A picture containing square, rectangle, tile

Description automatically generated A picture containing square, rectangle, tile, mosaic

Description automatically generated A black and white crossword puzzle

Description automatically generated with low confidence

would have the value (shortcut: white = dark blue = 1, light blue = grey = -1): 1, -3, 1, 9, respectively. Note I have implicitly set Stride = (1,1) here. Stride is the number of spaces you slide the filter over each time. The next row would look like,

A picture containing square, rectangle, colorfulness, tile

Description automatically generated A picture containing square, rectangle, colorfulness, tile

Description automatically generated A picture containing square, rectangle, pattern, line

Description automatically generated A picture containing square, rectangle, tile, mosaic

Description automatically generated

and have values -3, -3, 3, 1, respectively. And the next row would be:

A picture containing square, rectangle, pattern, line

Description automatically generated A picture containing square, rectangle, pattern, line

Description automatically generated A picture containing square, rectangle, pattern, line

Description automatically generated A picture containing square, rectangle, pattern, tile

Description automatically generated

with values, 1, 5, -3, -3. And the last row would be:

A picture containing square, rectangle, line, pattern

Description automatically generated A picture containing square, rectangle, pattern, line

Description automatically generated A picture containing square, rectangle, pattern, line

Description automatically generated A picture containing square, rectangle, tile

Description automatically generated

with values 9, 1, -3, 1. We can put all our values in a so-called *Feature Map*, Fmn (setting the bias term to zero for simplicity)

A grid of numbers with black text

Description automatically generated

But we’d want to normalize it by dividing everything by are max value, 9. So

A grid of numbers and symbols

Description automatically generated

So the closer it is to 1, the more the filter overlaps with the image in that region. The closer it is to -1, the less the filter overlaps with the image in that region. Can see that it overlaps quite strongly at the bottom left and top right corners. This makes sense. And in general it overlaps pretty well along that bottom left to top right diagonal. This is indeed appropriate for the character X. Finally, one will typically apply many filters, not just one. You could have a diagonal filter going the other way. And you could have an O filter. Or a vertical line filter, or horizontal line filter. Each of filters would ‘detect’ different features of the image. So at the end, we’d have a bunch of feature maps, Fij(α).



where α = 1, 2, 3, …, A, say. Let’s do some counting. So if our image has c color channels, and we apply A filters, each of which have dimensions dim(Filter)x × dim(Filter)y, then this will introduce the following number of weights and biases,



What is the number of neurons that the filters will reduce our image to? In general, if we start with a c × dim(I)x × dim(I)y pixel image and apply a dim(Filter)x × dim(Filter)y filter with padding p, and stride dim(s)x × dim(s)y, we will generate a feature map, Fmn. And the map will have the following dimensions:



where [ ] is the floor function. The size of the feature map would consequently be:



Every cell in this feature map would correspond to a neuron in the network. And if we have A of these filters, and so A of these feature maps, then the total number of neurons would be:



Moving on,

**2. Pool the Feature Map Values**

Finally, we don’t want to deal with even this many inputs. So we use pooling to abridge the feature map, in some sense,



We do this by breaking our feature map down into sections [aside: again, there are variations on this. Analogous to the previous step, we can first **pad/frame** the feature map with some number of 0 pixel layers, and then apply Pooling] If we use 2×2 Pool matrices, and Stride = (2,2), then we get this:

A grid of numbers and symbols

Description automatically generated

(Note if we used a 2×2 Pool matrix, but with Stride = 1, then we’d get a 3×3 pooled feature map matrix) And then we either replace the sections with their max value (Max Pooling) or mean value (Mean Pooling). For instance, Max Pooling would give us the following Pij:

A white rectangular box with black numbers

Description automatically generated

and Mean Pooling would be (if I’m mathing right) this Pij:

A white rectangular box with black numbers

Description automatically generated

And so then we’d use this 2×2 pooled feature map matrix as input instead of the original 6×6 Image matrix. And if we have multiple feature maps, then we’d have multiple pooled feature maps now,



It seems that we use the same pooling technique for all of them by default. Now for some counting. If we then apply pooling with a dim(Pool)x × dim(Pool)y pool matrix via stride dim(s)x × dim(s)y, and add padding p, then the pooled feature map will have these dimensions:



So size(P) = number of neurons we’re at now.



and if we have A pooled feature maps, we now have A times this number of neurons,



No new weights or biases are introduced here.

**3. Repeat for more Filters**

Often we will now repeat the process and apply another convolution layer to our feature maps. It would work like this. First, we combine the pooled feature maps Pmn(α) into a single ‘image vector’, which α = 1, 2, …, A channels, just like the *actual* image vector in the first step had three color channels.



Then when we apply a filter to this, it works like it did before. The filter has different arbitray weights for each channel. So we have for the resultant feature map:



And if we have multiple filters, then we get multiple feature maps, like before:



where β runs between 1, 2, 3, …, B = however many filters we want to employ. The number of *additional* weights and biases we will have introduced is:



And the number of neurons we’ll be at now is:



And as before, we will employ pooling on the feature maps, F, with some padding p, and some pooling matrix Pool, with some stride s, to reduce these to some set of more concise pooled feature maps.



This will reduce the number of neurons to:



We can repeat this any number of times I guess. We’ll note that the number of neurons in the network only depends on the N from the last pooling layer, so we don’t have to go through the entire layer to work it out.

This should contain a lot of relevant information about the image, while not being as verbose as the image itself. One of the nice things about the convolutional approach is that if you shift the image up/down or right/left, that will basically shift the feature map values in the same direction. And will by and large leave the reduced Max/Mean Pooled feature map the same as before. And so our learning algorithm will be insensitive to simple displacements of the image, as it should be. I wonder if there’s something you can do that would make it invariant to rotations too.

**4. Flatten and connect to a normal neural network**

The last step is to take our Pij(β) (or γ, δ, whatever) and flatten it all out into a single string of neurons, and then connect this to a normal neural network with whatever number of hidden layers, and then output to a softmax function appropriate to the requisite number of classes.

**CNN Architecture**

Now how are CNN’s structured? I think it’s something like this….

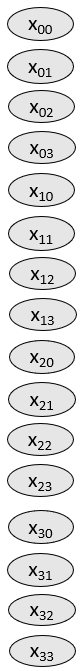
**Input**

So say we have a 4×4 image,

A grid of black squares with white text

Description automatically generated

We can flatten this array into a list of 16 inputs,



**Two Filters**

Then say we’re trying to identify whether the image, X, is a letter X (original!) or not. We could create two 2×2 filters: a backwards slash, and a forward slash. The weights in the filters are given as shown, but typically, one would not specify these weights. One would merely say they want two 2×2 filters, and let Keras/PyTorch determine the weights by minimizing the loss.

A black and white text

Description automatically generated with medium confidence

If we pass these filters over our X image, with Stride = (1,1), then each will generate a 3×3 feature map array, which I’ll call F and B respectively:

A black text on a white background

Description automatically generated

This would look like below.

A diagram of a network of data

Description automatically generated with medium confidence

For instance, observe how x00, x01, x10, and x11 go into B00. If the weights in the FilterBS are as shown, then we must have, including the bias: B00 = -x00 + x01 – x10 + x11 + b(B), and so the weights connecting x00, x01, x10, x11 to B00 would be -1, 1, -1, 1 respectively. Similarly consider F22. According to convolution stuff, F22 should involve elements x22, x23, x32, x33. And if the weights in the FilterFS are as shown, then we must have, including the bias: F22 = x22 – x23 – x32 + x33 + b(F). And so the weights connecting x22, x23, x32, x33 to F22 would be 1, -1, -1, 1 respectively. So if we completely specify our filters, then we will have completely specified the weights along all of those lines. But again, one doesn’t specify the weights in the filters, but lets them be chosen so as to minimize the loss. And so, like with all of our previous neural networks, we would have unknown weights associated with each of the lines (not *all* different weights).

*Counting*

So how many unknown weights and biases do we have? This is:



That is, we have one set of 4 unknown weights and 1 bias coming into F and a separate set coming into B. Next let’s get the dimension of the Feature Map,



This agrees. And now number of neurons is:



which also agrees.

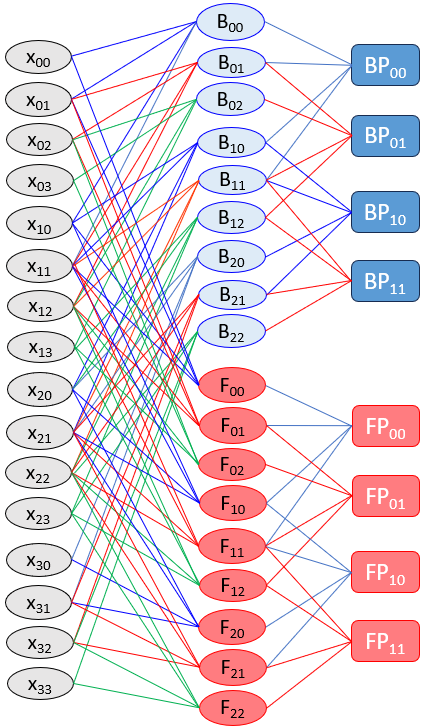
**Pooling**

Then we’d take these filtered matrices and pool them. If we use a 2×2 matrix with Stride = (1,1), then this will generate a 2×2 pooled matrix for each of our filtered matrices.

A black and white image of a square

Description automatically generated with medium confidence

If we use max pooling, then, for instance, FP00 will be max(F00, F01, F10, F11), and BP10 will be max(B10, B11, B20, B21). We can represent this below. The BP and FP boxes would be designed to output the maximum value of their inputs if using max pooling, and the average value of their inputs if using mean pooling. No arbitrary weights or variables are introduced in this step. So still, in this picture, 10 parameters total.



*Counting*

So the number of weights/biases is still the same. But pooling reduces the number of neurons. If we then apply pooling with Pool dim(Pool)x × dim(Pool)y, padding p, and stride dim(s)x × dim(s)y, then the pooled feature map will have these dimensions:



which matches. And the number of neurons we have now is:



**Two Filters**

Next we can apply to more filters, say. We’ll call them: Left and Right. I’m not sure what these would actually *mean* per se´, as far as what features they correspond to in the image. But whatever.

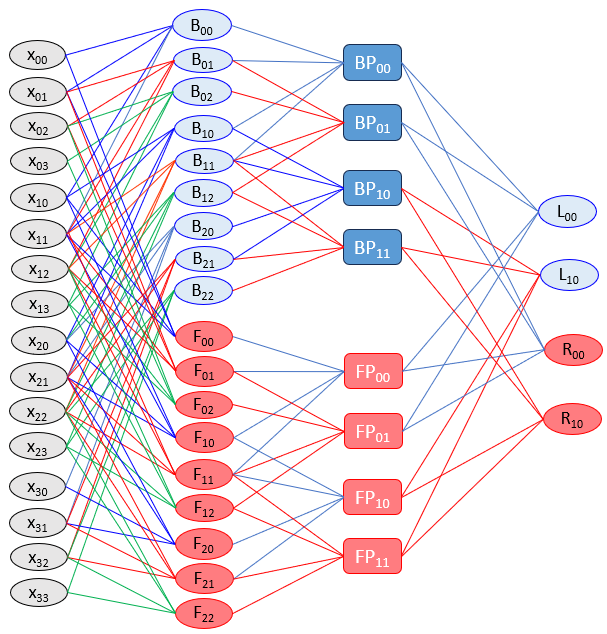


Passing these filters over our P matrix (Stride = (0,1), the only one possible) would result in two filtered matrices,

A black and white text

Description automatically generated with medium confidence

If the weights in our filters are as specified, then we’d have, e.g., L00 = 1∙BP00 + 0∙BP01 + 1∙FP00 + 0∙FP01 + b and L10 = 1∙BP10 + 0∙BP11 + 1∙FP10 + 0∙FP11 + b. But again, I think the weights in the filters would not be prespecified, but would rather be determined via minimization. Also, as discussed above, the weights would even be different for each pooled feature map the filter is applied to. So it’d look something like: L00 = w00∙BP00 + w01∙BP01 + w´00∙FP00 + w´01∙FP01 + b and L10 = w00∙BP10 + w01∙BP11 + w´00∙FP10 + w’01∙FP11 + b. In any event, now we have this below. And the activation function for the L and R filter nodes would be Alinear perhaps, though Arelu might be more frequently used):



*Counting*

So the number of additional weights and biases we have is:



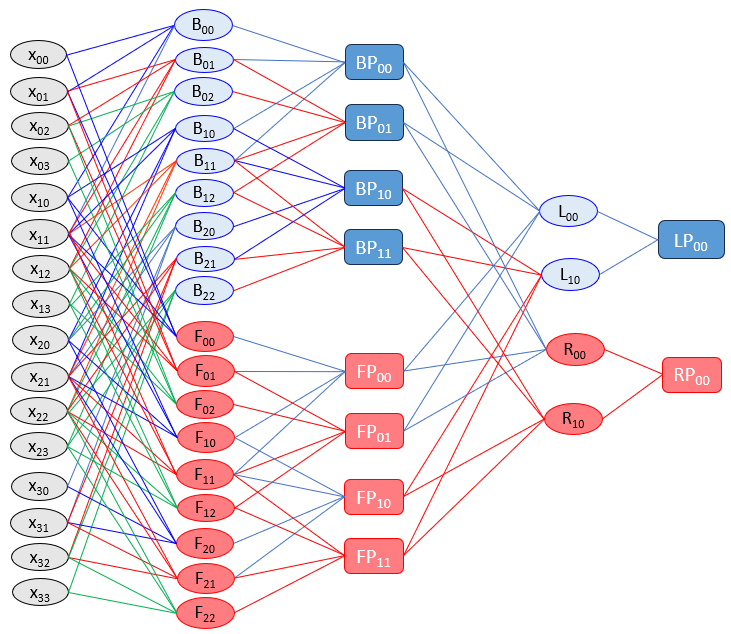
This is, four unknown weights and a bias for each filter. And the number of neurons we’ll be at now is:



which agrees with our figure.

**Pooling**

Next we come to the pooling layer again. Maybe we wouldn’t bother to put one in this time. But I guess I will still. There are only two values to pool. If we do max pooling then we’d have one node outputing LP00 = max(L00, L10) and another node outputing RP00 = max(R00, R10). This would look like,



*Counting*

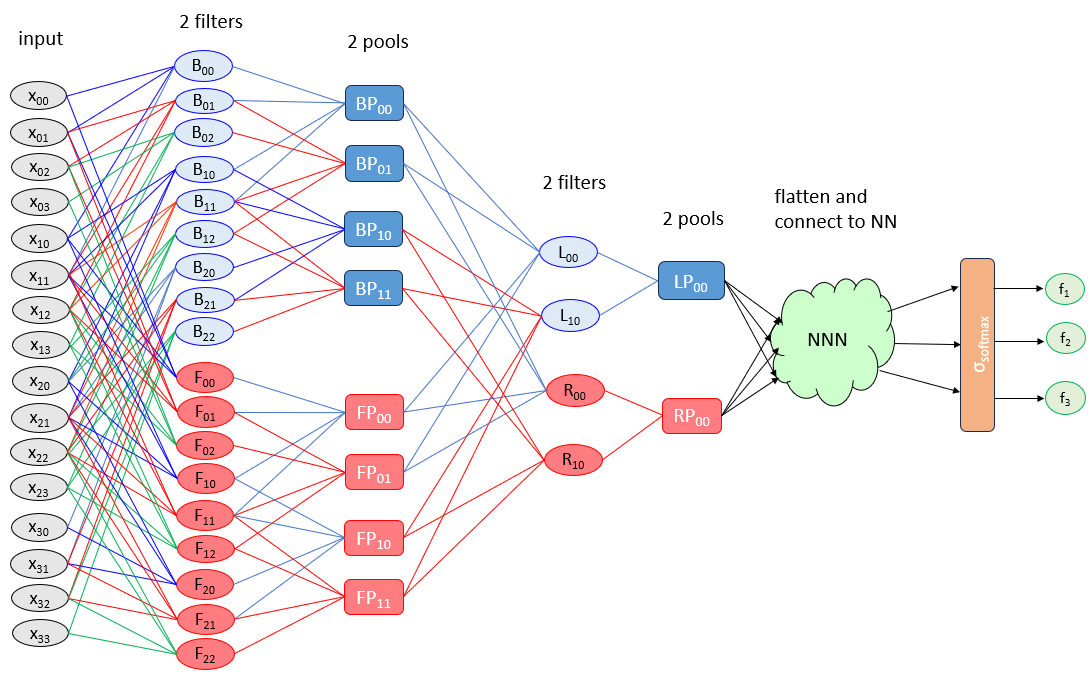
So no new weights and biases will be introduced. We’ll just be reducing the number of neurons.



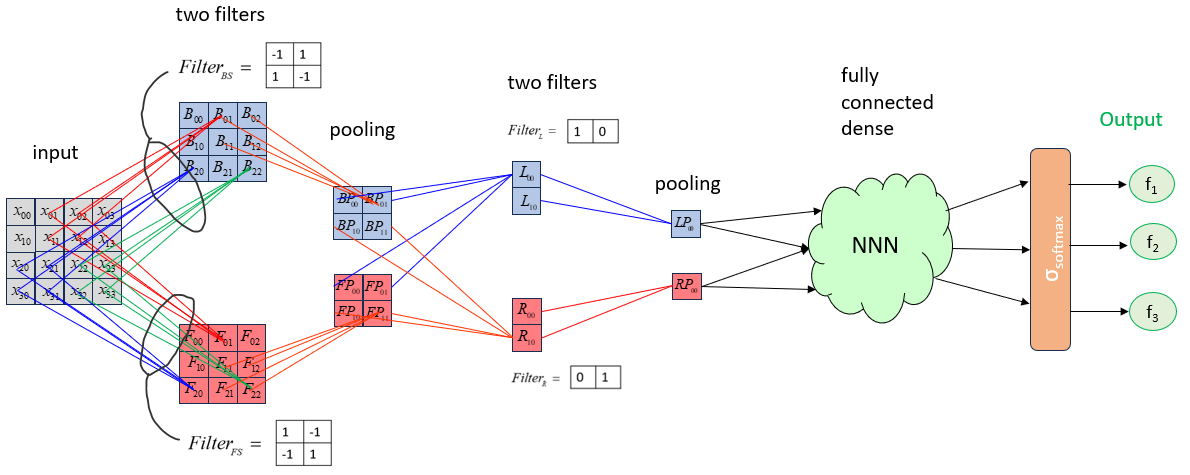
as fits the figure.

**Fully Dense Connected Layer**

And now the last part is to simply connect the remaining node(s) to a regular dense neural network layer with softmax output. So can say, presuming we have three possible output classes (and we’d probably use waaaay more than just two nodes in that fully connected layer.



Here’s another picture, below, of the same thing, though most connections in the picture have been left out for clarity. And the filters are shown too. So for instance, FilterBS’s values are the weights along each group of four lines leading into a B matrix element. And FilterFS’s values are the weights along each group of four lines leading into an F matrix element. Not shown is the fact that each group of four lines leading into a B matrix element gets a bias (same for every B element). And each group of four lines leading into an F matrix element also gets a bias (the same for every F element, though not the same as B’s bias, presumably). Also, the weights shown in the filters are just one possibility. The optimum weights are ascertained by minimization. Same goes for the two biases. And same goes for the other filter. So all total, our neural network would have: 2(4+1) (from first set of filters) + 2(4+1) (from second set of filters) = 20 free parameters, in addition to however many are used in the cloud NN.



Should also mention that if we have 3d input matrices (because of RBG values), everything works similarly. According to TensorFlow/Keras, our filters would still have to be 2d, but then all the matrices in the diagram will be 3 layers deep into the page (for RBG values). The same filters would be applied for each layer into the page. So we’d basically have three independent neural networks in parallel. And then after the final aggregating step, we’d flatten the array, and then connect to the fully dense neural network. One more topic here,

**Computer Vision**

There are three main areas of interest in computer vision:

A dog and cat in grass

Description automatically generated

Let’s discuss the YOLO algorithm, and its application to image classification and detection.

Yolo breaks an image up into an S×S grid of cells. Then it scans the image through many convolutional layers, resulting in a convolved image of the same size? At this point, each of the convolved image’s S2 cell’s receptive field, thanks to the convolution, encompasses much of the entire image. Yolo uses each convolved cell’s pixel values to detect the presence of an object, and to regress its bounding box, as well as the probability that there is an object in said box at all. This is called the confidence score. The cell might actually predict more than one bounding box, and confidence score, if it thinks it detects multiple objects? In fact, it seems each cell *has* to predict a certain number, say B, of bounding boxes. So we’ll say each grid cell predicts B bounding boxes, along with their confidence scores. So we have a bunch of boxes.

Then it looks for the center of each of these boxes. Say a box’s center is encompassed by some cell (could be the same cell that drew it, and could be a different cell). Then that cell is responsible for making the prediction of the identity of the object within the box. The cell does this not by looking inside the box per se´, but rather by looking at its own receptive field, which the box sits in the middle of. And it runs an image classification scheme on its receptive field. Since its receptive field is centered, more or less, about the alleged object in the box, it should have the best chance to identify the object. The algorithm doesn’t just output an estimate for the object’s identity, but a vector of probabilities for each of the identities. Obviously, one will be the greatest. So now each box will have a classification score and a class probability vector. What if two bounding box centers are inside a given grid cell? Well that grid cell will do the same thing regardless. It will run a classification algorithm on its receptive field and associate the probability vector output with both boxes. Hopefully the aforementioned two bounding boxes are for the same object. If they are of different objects then too bad so sad, because the algorithm cannot distinguish them. It can only give a single classification vector for its receptive field. So now we have boxes, and a confidence score and class probability vector for each box (but remember that boxes whose centers are in the same grid cell will have identical object identity predictions, and also that the grid cell that drew the box is not necessarily/likely the one that predicts the identity of the object allegedly within it). All total, our data is a S×S×B(5+C) dimensional matrix. 5 is for height, width, x, y, and confidence score. And C is for the probabilities of each class.

The following is somewhat conjectural – see <https://medium.com/analytics-vidhya/yolo-explained-5b6f4564f31>. So to train the algorithm, we compare all the boxes, with their confidence scores, and class probability prediction vectors, to the original image, with its known objects, and their bounding boxes, and their identities. We will single out as definitive predictions of the model the boxes that have the highest overlap with the actual object bounding boxes from the image. Then we compute the following loss function:



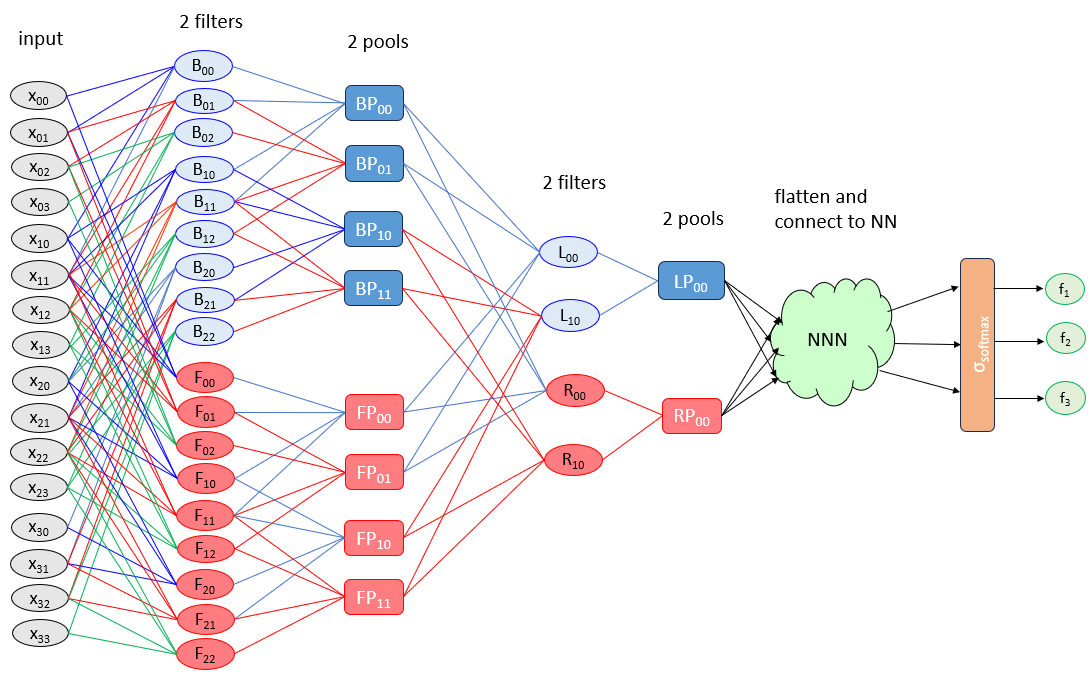
So the sum goes over all S2 cells of the grid, and over all B bounding boxes predicted by that cell. 1ijpred is like a Kronecker delta which gives 1 if box ij is used for prediction and 0 otherwise. And again, a box is used for prediction if it has the highest overlap (IOU technically) with an actual object box from the image. (xi,yi) and (I, i) are actual and predicted coordinates of the center of the box. (wi,hi) and (I, i) are actual and predicted width and height of the box. Ci and i are the actual and predicted confidence scores (wouldn’t Ci be 1 at all times?). Iijnopred is the opposite of Iijpred; it’s 1 if the box wasn’t used for prediction, and 0 if it was. And pi(c), i(c) are the actual and predicted class probabilities (wouldn’t pi(c) just be all zeros and a one, like (0,0,1,0)?). Can see that we’re basically using an SSE (sum of squared errors) approach. It’s interesting that we just use SSE for the probability vector, rather than a log-loss of something. We use √w and √h, rather than w and h so that errors with larger widths and heights are weighted less than errors with smaller widths and heights. For instance, the error involved for an actual, predicted width of 8, 10 would be 0.1, while the error involved for an actual, predicted width of 98, 100 would be 0.01. λcoord is there to give a possibly different weight to bounding box location errors vs. bounding box confidence errors (i.e., boundary boxes that don’t encompass anything). And λnopred is there to weight confidence scores for boxes that weren’t ultimately used for prediction less heavily than confidence scores for boxes that were.

Finally, once the model is trained, and we run it on an image, and we have all of these boxes, with their confidence scores, and class predictions, how do we choose the correct boxes? So we start with highest confidence score box, and eliminate boxes with which it has high IOU (regardless of confidence score), down to some IOUmin I guess. This step eliminates superfluous bounding boxes that are likely for the same object. But we don’t want to eliminate boxes down to too low an IOU, because then we are probably eliminating boxes that belong to a different object. I guess if we have higher IOUmin then this might improve ability to distinguish nearby, but distinct, objects. But then at the same time, we might instead classify a single object as two different objects. Anyway, now we have the most probable bounding box for the hypothetical object, along with its associated probability class vector. Then we move on to the remaining boxes and locate the one with the next highest confidence score. Presumably this box will be well separated from our just-located object. And like before, we will cull the surrounding highly overlapping boxes, down to some minimum IOUmin. Then we have our most likely box, along with its probability class vector, for our second object. And we keep doing this until all boxes are eliminated. Presumably, at some point in this process, we might be left only with boxes with confidence scores less than some Cmin. Maybe we just discard these?

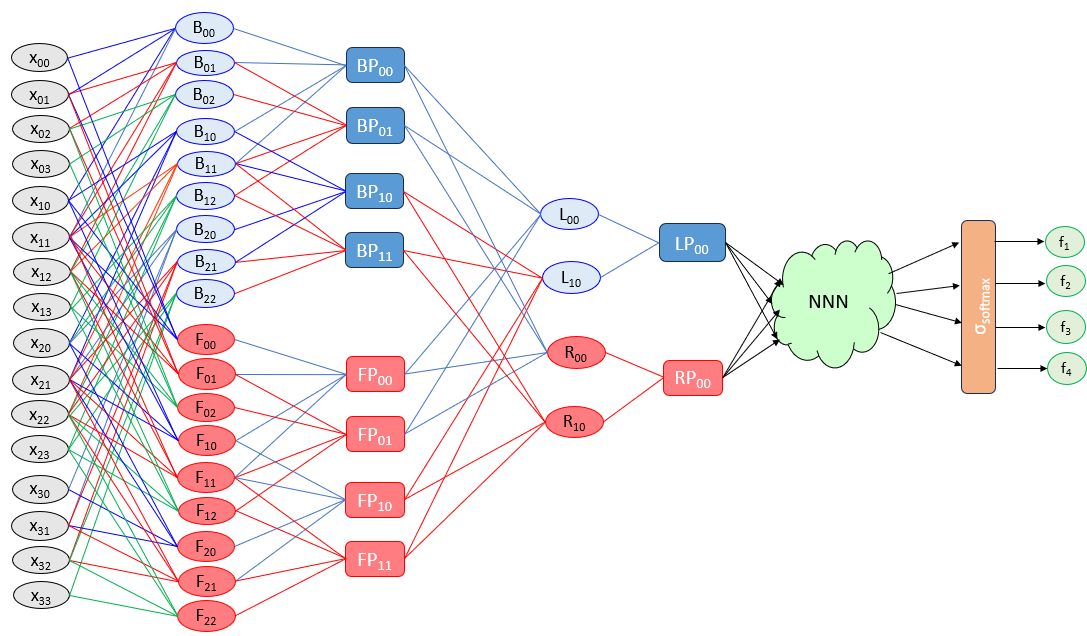
One more comment. To address the problem of distinguishing two objects within the same cell, we can introduce the concept of anchor boxes. The only thing this changes in our algorithm is the part where we are classifying the objects in the boxes the algorithm predicts. So now, instead of a cell outputing just one object prediction (or class probability vector) for all the boxes within its receptive field, it will predict 2.

**Transfer Learning**

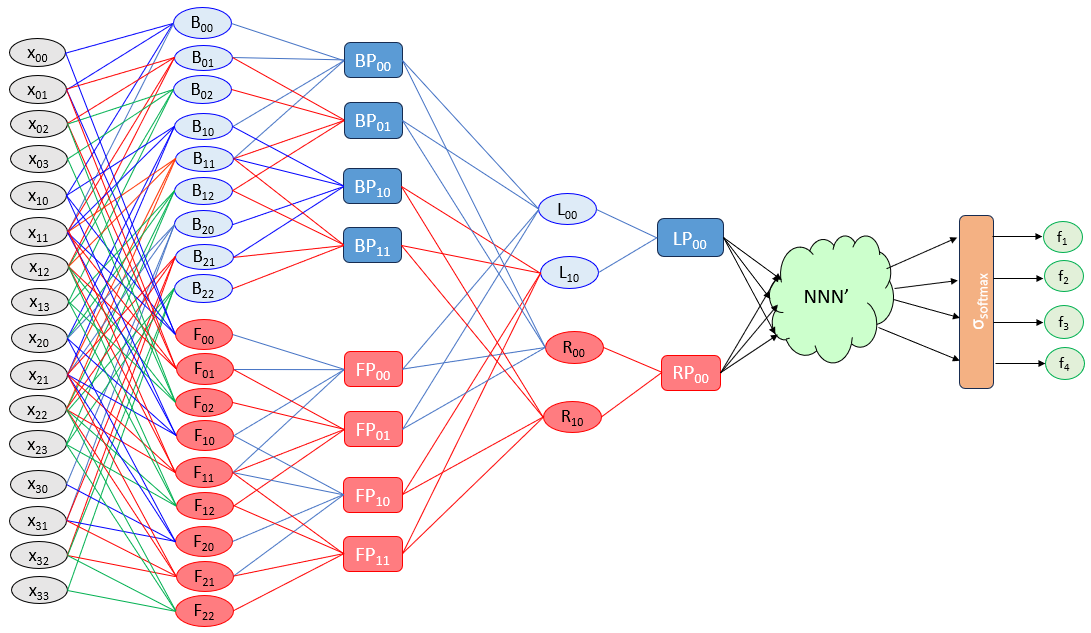
Often times, instead of developing our own algorithm, we’ll just borrow someone else’s. For instance, consider the algorithm above,



It might be trained to identify various animals. But we want to identify various cars. Well, maybe we can use the same model, same weights/biases, etc., but just chop off the soft max layer and add our own,



And then we could just train the weight leading to the softmax function. This would be a quick and dirty approach that often works well. Or if necessary, we could also tinker with NNN --> NNN´.



Another option is to retrain *all* the weights and biases of the entire model, but this might take days.