DEEP LEARNING FOR VISION SYSTEMS.

ACTIVATOR FUNCTIONS:

1. HEAVISIDE STEP FUNCTION: BINARY OUTPUT 0 or 1.
2. SIGMOID / LOGISTIC FUNCTION: THE SIGMOID SQUISHES ALL OF THE VALUES TO A PROBABILITY BETWEEN 0 & 1.
3. SOFTMAX FUNCTION: FORCES THE OUTPUTS TO SUM TO 1. (i.e) SO IT TRANSFORMS THE INPUT VALUES TO PROBABILITIES BETWEEN 0 & 1.

$$\begin{bmatrix} 1.2 \\ .9 \\ .4 \end{bmatrix} \xrightarrow{\text{SOFTMAX}} \begin{bmatrix} .46 \\ .34 \\ .20 \end{bmatrix}$$

4. HYPERBOLIC TANGENT FUNCTION (TANH)
   TANH SQUISHES ALL VALUES TO BETWEEN -1 & 1
   IT HAS THE EFFECT OF CENTERING THE DATA SO THAT THE MEAN OF THE DATA IS CLOSE TO 0 WHICH IS DIFFERENT THAN SIGMOID WHICH TENDS TO PUT THE MEAN AROUND .5
5. RECTIFIED LINEAR UNIT: ACTIVATES A NODE ONLY IF THE INPUT IS ABOVE 0. IF THE INPUT IS BELOW 0 THE OUTPUT WILL BE 0. $f(x) = \max(0, x)$
   ReLU (RECTIFIED LINEAR UNIT) TENDS TO TRAIN BETTER THAN SIGMOID & TANH IN HIDDEN LAYERS.
6. Leaky ReLU: LeakyReLU introduces a small negative slope (~.01) for x values less than 0. (ie) $f(x) = \max(.01x, x)$

```
def leaky_relu(x):
    if x<0:
        return x*.01
    else:
        return x
```

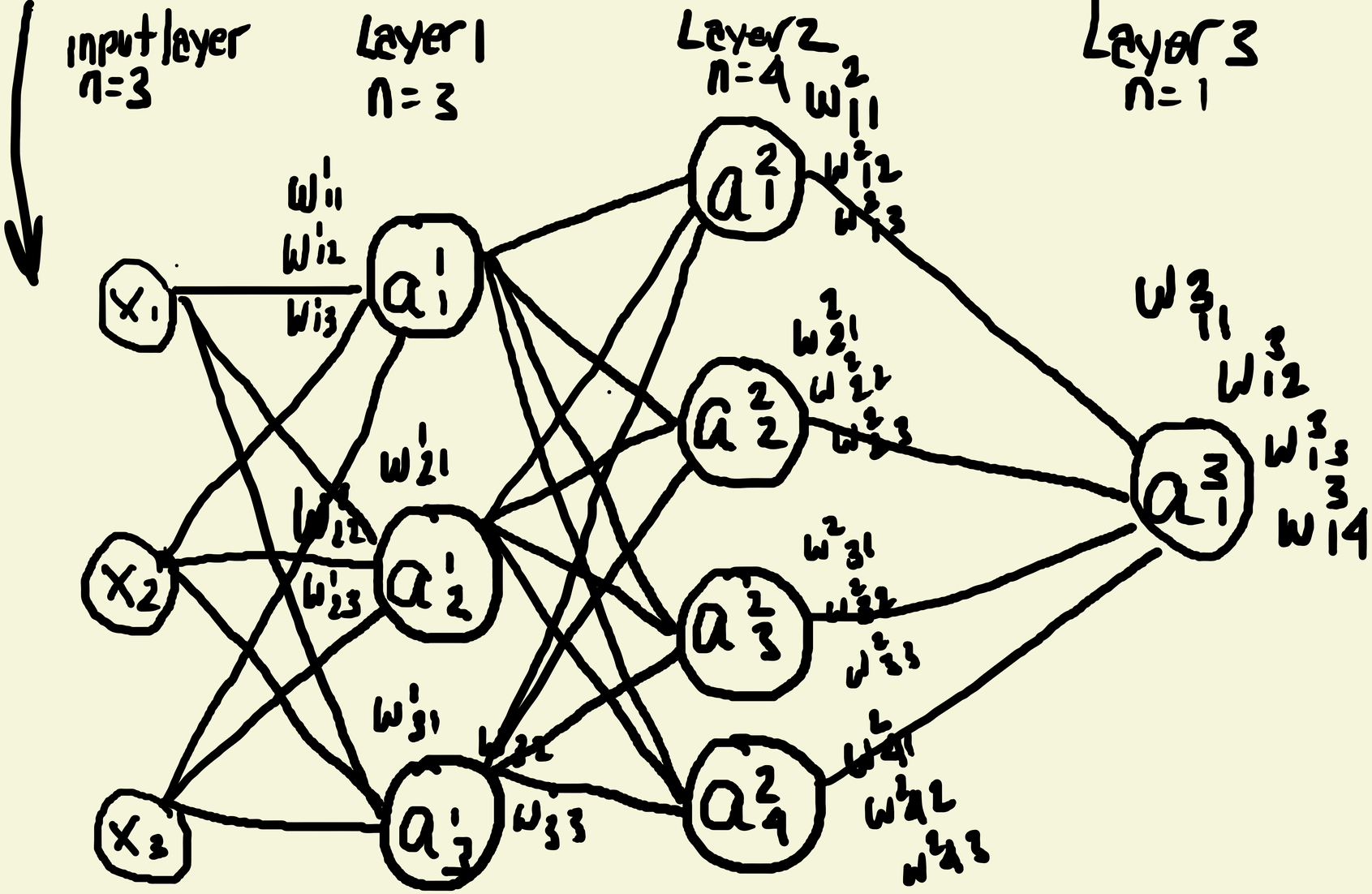
ACTIVATION FUNCTION SELECTION

FOR HIDDEN LAYERS : MOSTLY USE ReLU
FOR OUTPUT LAYERS : SOFTMAX WHEN CLASSES ARE MUTUALLY EXCLUSIVE.
SIGMOID FOR BINARY CLASSIFICATIONS.
FOR REGRESSION PROBLEMS DON'T USE AN ACTIVATION AT ALL SINCE THE WEIGHTED SUM NODE PRODUCES THE NECESSARY OUTPUT. (IE) PREDICT HOUSING PRICE BASED ON COMPS.

FEED FORWARD PROCESS

THE PROCESS OF COMPUTING THE LINEAR COMBINATION & APPLYING THE ACTIVATION FUNCTION IS CALLED FEED FORWARD.
(IE) THE FORWARD DIRECTION IN WHICH THE INFORMATION FLOWS FROM THE INPUT LAYER THROUGH THE HIDDEN LAYERS, ALL THE WAY TO THE OUTPUT LAYER. THIS PROCESS HAPPENS THROUGH THE IMPLEMENTATION OF TWO CONSECUTIVE FUNCTIONS: THE WEIGHTED SUM & THE ACTIVATION FUNCTION. THE FORWARD PASS IS THE CALCULATIONS THROUGH THE LAYERS TO MAKE A PREDICTION.

[2-20]



$W^{(n)}_{ab}$ (n): Layer #

(ab): weighted edge connecting the a^{th} neuron in layer (n) to the b^{th} neuron in the previous layer (n-1).

weights for biases are denoted as (w)

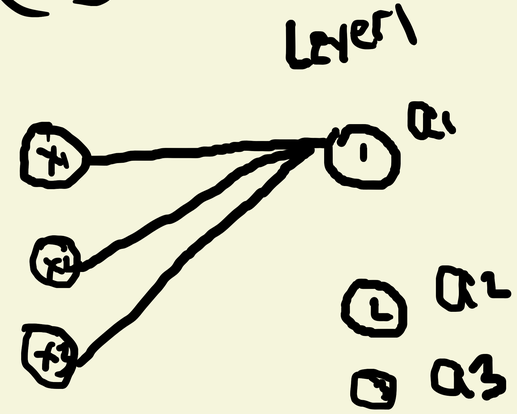
Activation functions are denoted as $\sigma(x)$.
Node values are denoted as (a)

* we calculate the weighted sum, apply the activation function and assign this value to the node a_{mn}, where n = layer
m = node index
(ie) a_{23} means node 2 layer 3

WEIGHTS + BIASES: (w, b): THE EDGES BETWEEN NODES ARE ASSIGNED RANDOM WEIGHTS DENOTED AS $W_{ab}^{(n)}$, WHERE (n) = Layer
(ab) = WEIGHTED EDGE CONNECTING THE ath NEURON IN LAYER (n) TO THE bth neuron in THE PREVIOUS LAYER

* IN DL LITERATURE (w) refers TO THE WEIGHTS + BIASES.
THIS IS BECAUSE WEIGHTS + BIASES ARE TREATED SIMILARLY IN THE SENSE THAT THEY ARE RANDOMLY INITIALIZED.

$\sigma(x)$: ACTIVATION FUNCTION.
(a): NODE VALUE.



$$a_1^{(1)} = \sigma(w_{11}^{1} x_1 + w_{12}^{1} x_2 + w_{13}^{1} x_3)$$
$$a_2^{(1)} = \sigma(w_{21}^{1} x_1 + w_{22}^{1} x_2 + w_{23}^{1} x_3)$$
$$a_3^{(1)} = \sigma(w_{31}^{1} x_1 + w_{32}^{1} x_2 + w_{33}^{1} x_3)$$

I think this is correct

FEED FORWARDS CALCULATIONS

W^{L}_{ab} ← L: Layer; b: b^{th} neuron in Layer $L-1$; a: a^{th} neuron in Layer L

$$a^{(1)}_1 = \sigma(W^{(1)}_{11} x_1 + W^{(1)}_{12} x_2 + W^{(1)}_{13} x_3)$$

$$a^{(1)}_2 = \sigma(W^{(1)}_{12} x_1 + W^{(1)}_{22} x_2 + W^{(1)}_{32} x_3)$$

$$a^{(1)}_3 = \sigma(W^{(1)}_{13} x_1 + W^{(1)}_{23} x_2 + W^{(1)}_{33} x_3)$$

THEN DO SAME FOR LAYER 2

$(a^{(1)}_1, a^{(2)}_2, a^{(2)}_3,$ and $a^{(2)}_4$ ALL THE WAY TO OUTPUT PREDICTION LAYER 3

$$\hat{y} = a^{(3)}_1 = \sigma(W^{(3)}_{11} a^{(2)}_1 + W^{(3)}_{12} a^{(2)}_2 + W^{(3)}_{13} a^{(2)}_3 + W^{(3)}_{14} a^{(2)}_4)$$

SO, BASICALLY THERE ARE A LOT OF CALCULATIONS HERE SO MATRICES CAN SPEED THIS UP. NUMPY CAN BE USED TO DO THIS.

- STACK ALL INPUTS TOGETHER IN ONE VECTOR (ROWS, COLUMN). IN THIS CASE (3, 1)
- MULTIPLY THE INPUT VECTOR BY THE WEIGHTS MATRIX FROM LAYER 1 ($W^{(1)}$) & THEN APPLY THE SIGMOID FUNCTION. MULTIPLY RESULT FOR LAYER 2 $\Rightarrow \sigma \cdot W^{(2)}$ & Layer 3 $\Rightarrow \sigma \cdot W^{(3)}$

IF YOU HAVE A FOURTH LAYER THEN MULTIPLY THE RESULT FROM STEP 3 BY $\sigma \cdot W^{(4)}$ AND SO ON UNTIL WE GET TO THE FINAL PREDICTION OUTPUT $\hat{y}$.

$$\hat{y} = \sigma \cdot W^{(3)} \cdot \sigma \cdot W^{(2)} \cdot \sigma \cdot W^{(1)} \cdot (x)$$

READ THIS FROM RIGHT TO LEFT

$W^{(3)}$

$$\hat{y} = \sigma\left[W^3_{11} \cdot W^3_{12} \cdot W^3_{13} \cdot W^3_{14}\right] \cdot \sigma \begin{bmatrix} W^2_{11} & W^2_{12} & W^2_{13} \\ W^2_{21} & W^2_{22} & W^2_{23} \\ W^2_{31} & W^2_{32} & W^2_{33} \\ W^2_{41} & W^2_{42} & W^2_{43} \end{bmatrix} \cdot \sigma \begin{bmatrix} W^1_{11} & W^1_{12} & W^1_{13} \\ W^1_{21} & W^1_{22} & W^1_{23} \\ W^1_{31} & W^1_{32} & W^1_{33} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

Layer 3 | Layer 2 | Layer 1 | Input vector

FEATURE LEARNING

FROM [2-20] WE HAVE THREE FEATURE INPUTS...

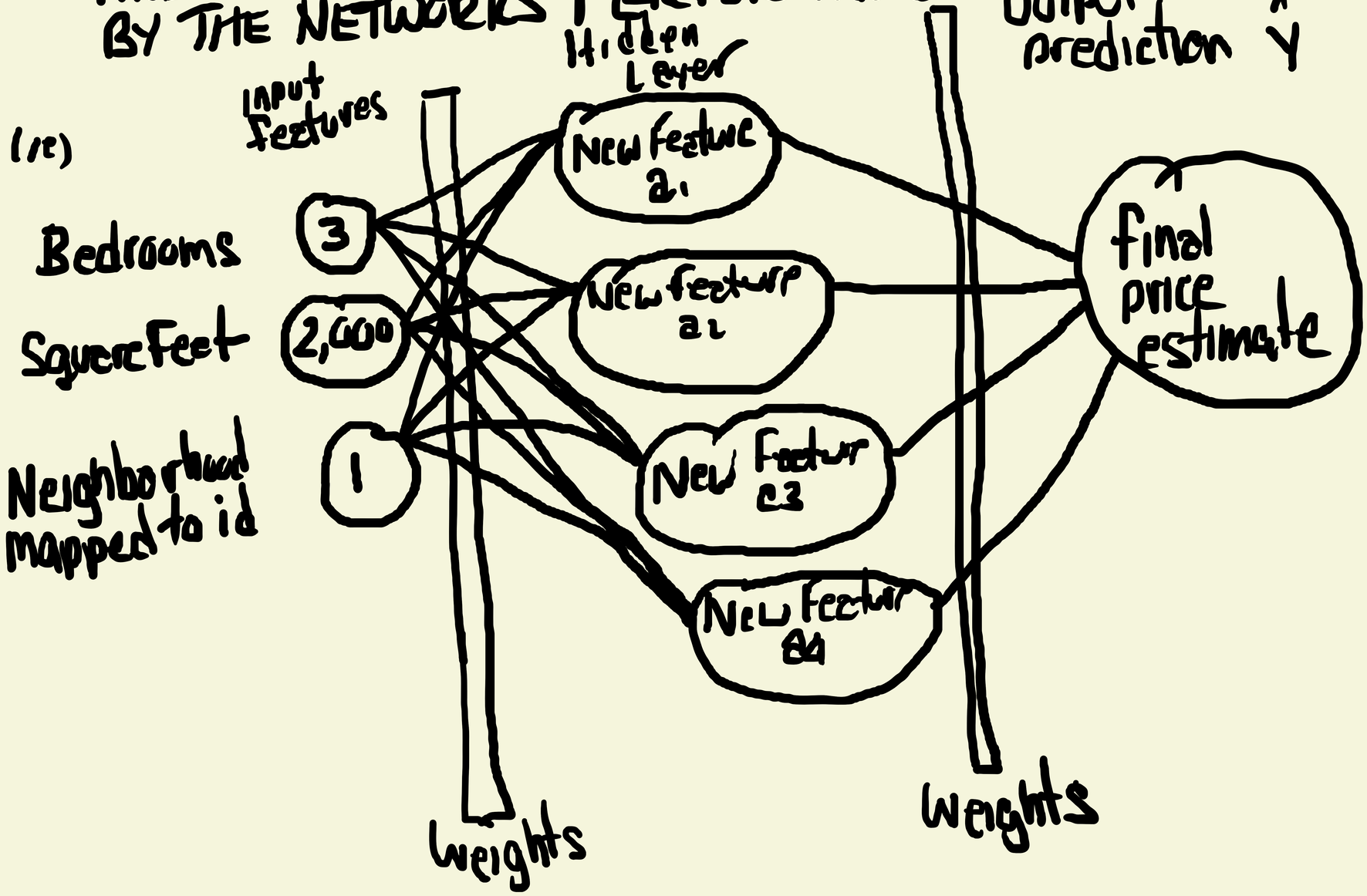
AFTER COMPUTING THE FORWARD PASS IN THE FIRST LAYER THE NETWORK LEARNS PATTERNS AND THE FEATURES ARE TRANSFORMED TO NEW FEATURES WITH DIFFERENT VALUES. THEN IN THE NEXT LAYER THE NETWORK LEARNS PATTERNS WITHIN PATTERNS.

(X 3)

AND PRODUCES NEW FEATURES AND SO FORTH.

* THE PRODUCED FEATURES AFTER EACH LAYER ARE NOT COMPLETELY UNDERSTOOD. WE DON'T SEE THEM NOR DO WE HAVE MUCH CONTROL OVER THEM. WHAT WE DO IS TO LOOK AT THE FINAL OUTPUT PREDICTION AND KEEP TUNING SOME PARAMETERS UNTIL WE ARE SATISFIED BY THE NETWORKS PERFORMANCE

(1c)



THE FIRST LAYER LEARNS BASIC FEATURES. THE SECOND LAYER BEGINS TO LEARN MORE COMPLEX FEATURES. THE PROCESS CONTINUES UNTIL THE LAST LAYERS OF THE NETWORK LEARN EVEN MORE COMPLEX FEATURES THAT FIT THE DATASET.

VECTORS & MATRICES REFRESHER

A SCALAR IS A SINGLE NUMBER.
A VECTOR IS AN ARRAY OF NUMBERS.
A MATRIX IS A 2D ARRAY.
A TENSOR IS AN N-DIMENSIONAL ARRAY WITH N>2.

SCALAR VECTOR MATRIX TENSOR

$$1 \qquad \begin{bmatrix} 1 \\ 2 \end{bmatrix} \qquad \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \qquad \begin{bmatrix} [1\,2] & [3\,2] \\ [1\,7] & [5,4] \end{bmatrix}$$

SCALAR: n
Vector: x
Matrix: X

MATRIX DIMENSIONS ARE WRITTEN AS (ROW×COLUMN)

MULTIPLICATION

SCALAR MULTIPLICATION: MULTIPLY THE SCALAR BY ALL THE NUMBERS IN THE MATRIX

$$2 \cdot \begin{bmatrix} 10 & 6 \\ 4 & 3 \end{bmatrix} = \begin{bmatrix} 2 \cdot 10 & 2 \cdot 6 \\ 2 \cdot 4 & 2 \cdot 3 \end{bmatrix} = \begin{bmatrix} 20 & 12 \\ 8 & 6 \end{bmatrix}$$

MATRIX MULTIPLICATION:

(1c)

$$[3\ 4\ 2] \cdot \begin{bmatrix} 13 & 9 & 7 \\ 8 & 7 & 4 \\ 6 & 4 & 0 \end{bmatrix} = [x\ y\ z]$$

1×3 $\quad$ 3×3

$x = (3\cdot13)+(4\cdot8)+(2\cdot6) =$

$y = (3\cdot9)+(4\cdot7)+(2\cdot4)$

$z = (3\cdot7)+(4\cdot4)+(2\cdot0)$

MATRIX TRANSPOSITION

symbol for: (A^T) invert the shape $(m\times n) \rightarrow (n\times m)$

(1c) Converting a row vector to a column vecter.

Hint: Visuallydo this by shifting the object to the left.

(1e) $A = \begin{bmatrix} 2 \\ 8 \end{bmatrix}$ $A^T = [2\ 8]$

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad A^T = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

$$A = \begin{bmatrix} 0 & 1 \\ 2 & 4 \\ 1 & -1 \end{bmatrix} \quad A^T = \begin{bmatrix} 0 & 2 & 1 \\ 1 & 4 & -1 \end{bmatrix}$$

ERROR FUNCTION

$\hat{y}$ IS MEASURED AGAINST THE ERROR FUNCTION SO WE CAN EVALUATE THE RESULT.

↑ Prediction

ERROR FUNCTION = COST FUNCTION = LOSS FUNCTION.

THE ERROR FUNCTION IS A MEASURE OF HOW WRONG THE PREDICTION IS WITH RESPECT TO THE EXPECTED OUTPUT (LABEL).

CALCULATING ERROR IS AN OPTIMIZATION PROBLEM.
OUR GOAL IS TO FIND THE OPTIMAL WEIGHTS TO MINIMIZE THE ERROR FUNCTION.
THE ERROR FUNCTION IS USED DURING PROGRESSIVE ITERATIONS.

Loss functions

2 MOST COMMON: MEAN SQUARED ERROR: FOR REGRESSION PROBLEMS
CROSS-ENTROPY: CLASSIFICATION PROBLEMS.

MEAN SQUARED ERROR (MSE): USED IN REGRESSION PROBLEMS THAT REQUIRE THE OUTPUT TO BE A REAL VALUE. (IE) HOUSING PRICE.

INSTEAD OF COMPARING THE PREDICTED OUTPUT WITH THE LABEL $(\hat{y} - y_i)$
THE ERROR IS SQUARED & AVERAGED OVER THE NUMBER OF DATA POINTS.

(1e) $$E(W,b) = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}-y_i)^2$$

$E(W,b)$ = Loss function
W = weights matrix
b = biases vector
N = number of training examples
$\hat{y}_i$ = prediction output
y_i = the correct output (label)
$(\hat{y}_i - y_i)$ = the residual.

CROSS-ENTROPY: COMMONLY USED IN CLASSIFICATION PROBLEMS BECAUSE IT QUANTIFIES THE DIFFERENCE BETWEEN TWO PROBABILITY DISTRIBUTIONS.

(1e) TRUE DISTRIBUTION.
P(CAT) 0.0
P(DOG) 1.0
P(FISH) 0.0

LEARNING PREDICTION
P(CAT) .2
P(DOG) .3
P(FISH) .5

$$E(W,b) = -\sum_{j=1}^{m} y_i \log(p^i)$$

where y: target probability (true)
p: predicted (learning)
m: number of classes.

So, $E = 0.0 \cdot \log(.2) + 1 \cdot \log(.5) + 0 \cdot \log(.3) = 1.2$

1.2 is how "wrong" or "far away" our prediction is from the true distribution.

Example Learning #2

P(Cat)	.3
P(Dog)	.5
P(Fish)	.2

$E = 0 \cdot \log(.3) + 1 \cdot \log(.5) + 0 \cdot \log(.2) = .69$ = Better prediction because .69 is less far away.

Here is generalized formula (adding summation of all training examples)

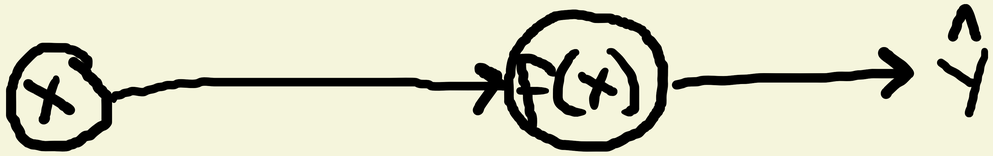
$$E(W,b) = -\sum_{i=1}^{n}\sum_{i=1}^{m} y_{ij} \log(p_{ij})$$

The libraries will do calcs for us. Tensorflow, PyTorch, and Keras.

WE WANT THE ERROR FUNCTION TO BE AS CLOSE TO ZERO AS POSSIBLE

HOW TO MINIMIZE THE ERROR.

RELATIONSHIP BETWEEN WEIGHT & ERROR.



$x = .3$
$y = .8$
then $\hat{y} = w \cdot x = w \cdot .3$

the error in its simplest form is simply comparing $\hat{y}$ + label y
error: $|\hat{y} - y| = |(w \cdot x) - .8| = |w \cdot 3 - .8|$
since the input x and the goal y are fixed then only two values can change: error + weight so if we want to get the minimum error then the variable that we need to adjust is weight. The weight is like a knob  that the network adjusts in order to minimize the error.

The process of finding the goal weights happens by adjusting the weight values in an interative process using an optimization algorithm.

Optimization Algorithm

TRAINING A NEURAL NETWORK INVOLVES SHOWING THE NETWORK MANY EXAMPLES. (A TRAINING DATASET).

MOST POPULAR OPTIMIZATION ALGORITHM

GRADIENT DESCENT

BGD: Batch Gradient Descent

SGD: Stochastic Gradient Descent

Mini-batch GD

Batch Gradient Descent

THE DEFINITION OF A GRADIENT (aka Derivative) IS THAT IT IS THE FUNCTION THAT TELLS YOU THE SLOPE OR RATE OF CHANGE OF THE LINE THAT IS TANGENT TO THE CURVE AT ANY GIVEN POINT.

Gradient descent means to update the weights iteratively to descend the slope of the error curve until we get to the point with the minimum error.

SO... AT THE INITIAL WEIGHT POINT WE CALCULATE THE DERIVATIVE OF THE ERROR FUNCTION TO GET THE SLOPE (DIRECTION) OF THE NEXT STEP. WE KEEP REPEATING THIS PROCESS TO TAKE STEPS DOWN THE CURVE

UNTIL WE REACH THE MINIMUM ERROR.

NOW, IN ORDER TO descend the curve we need to do two things for each step.

① step direction (gradient)
② step size (learning rate)

THE DIRECTION (GRADIENT)

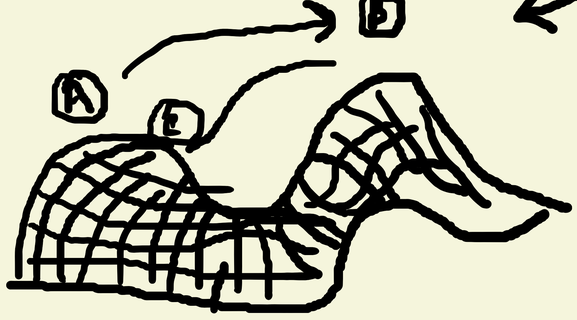
AT EACH STEP WE PICK THE DESCENT BASED UPON THE STEEPEST SLOPE AT THAT STEP. BY TAKING THE DERIVATIVE OF THE ERROR WITH RESPECT TO WEIGHT dE/dW, WE GET THE DIRECTION WE SHOULD TAKE.

THE STEP SIZE (LEARNING RATE)

LEARNING RATE IS DENOTED BY α (alpha). IT IS ONE OF THE MOST IMPORTANT HYPER PARAMETERS TO TUNE.

A LARGE LEARNING RATE WILL DESCEND MORE QUICKLY BECAUSE IT TAKES LARGER STEPS. THUS, THE LEARNING WILL TAKE LESS TIME.

HOWEVER, IF LARGE STEPS ARE TAKEN YOU COULD WIND UP AT A PEAK ... OSCILLATING AND NOT MAKING DOWNWARD PROGRESS.



← SETTING A LARGE LEARNING-RATE CAUSES ERROR TO OSCILLATE AND NEVER DESCEND.

BY MULTIPLYING THE DIRECTION (DERIVATIVE) by the step size (learning rate) we get the change of the rate for each step.

$\Delta w_i = -\alpha \frac{dE}{dw_i}$: we add minus sign because derivative always calculates slope in upward direction.

$W_{next\text{-}step} = W_{current} \cdot \Delta W$

CALCULATING PARTIAL DERIVATIVE

$E(x)$: ERROR FUNCTION

$dE(x)/dw$: Derivative of error function with respect to weight. this will show how much the total error will change when we change the weight.

DERIVATIVE RULES

Constant Rule: $d/dx(c) = 0$

Constant Multiple Rule: $d/dx[cf(x)] = cf'(x)$

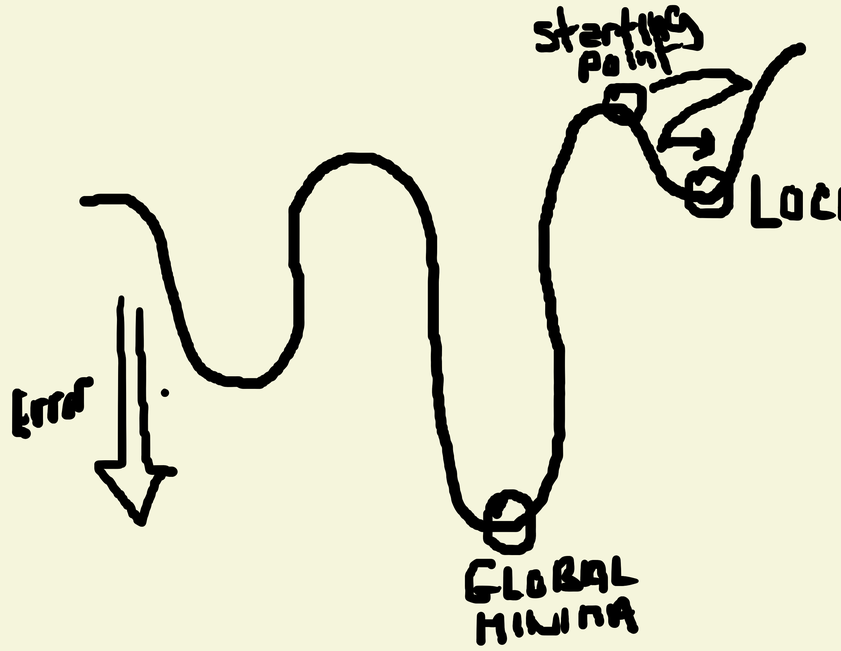
Power Rule: $d/dx(x^n) = nx^{n-1}$

Sum Rule: $d/dx[f(x) - g(x)] = f'(x) - g'(x)$

Quotient rule: $\frac{d}{dx}\left[\frac{f(x)}{g(x)}\right] \quad \frac{g(x)f'(x) - f(x)g'(x)}{[g(x)]^2}$

(ie) $f(x) = 10x^5 + 4x^7 + 12x$

$50x^4 + 28x^6 + 12$



Pitfalls Of Gradient Descent

(1) Since the starting point is randomly selected it is possible to start out in such a fashion as to descend into a local minima.

(2) Batch gradient descent uses the entire training set to compute the gradients at every step it can be extremely computationally expensive.

Stochastic Gradient Descent

(Most Popular in Deep Learning)

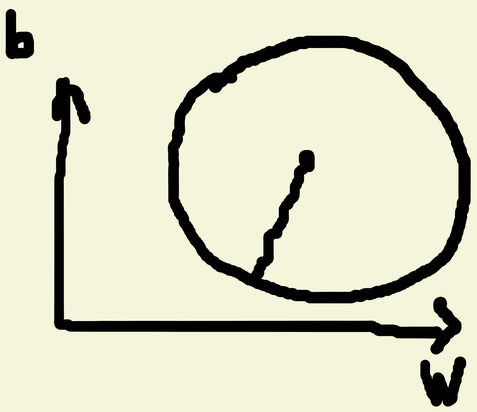
Stochastic is just a fancy word for random.

Stochastic gradient descent randomly selects data points + goes through the gradient descent one datapoint at a time. SGD descends all paths to calculate their minima. Then the minimum value of all the local minimas is chosen to be the global minima.

SGD randomly picks one instance in the training set for each one step + calculates the gradient based only on that single instance.

Gradient Descent

1. TAKES ALL DATA
2. COMPUTE GRADIENT
3. UPDATE WEIGHTS + TAKE STEP DOWN.
4. REPEAT FOR n NUMBER OF EPOCHS.



Stochastic Gradient Descent

1. RANDOMLY SHUFFLE SAMPLES IN TRAINING SET
2. PICK ONE DATA INSTANCE
3. COMPUTE Gradient
4. update the weights + take step down
5. PICK ANOTHER ONE DATA INSTANCE
6. REPEAT FOR n NUMBER OF EPOCHS (TRAINING ITERATIONS).

Mini-Batch Gradient Descent

INSTEAD OF COMPUTING GRADIENT FROM ONE SAMPLE (SGD) OR ALL SAMPLES (BGD), WE DIVIDE THE TRAINING SAMPLE DATA INTO MINI BATCHES.

MB-GD CONVERGES IN FEWER ITERATIONS THAN BGD

MB-GD HAS COMPUTATIONAL PERFORMANCE OVER SGD DUE TO IT BEING ABLE TO USE VECTORIZED OPERATIONS.

TIPS

ADJUST THE LEARNING RATE BY TRIAL + ERROR. START AT 0.01

AND THEN GO DOWN TO .001, .0001, .00001 (WATCH FOR INFINITY WONT DESCEND.)

batch_size is another hyperparameter that we tune.
32, 64, 128, 256

SUMMARY OF NETWORK TRAINING

① $\hat{y} = \sigma \cdot W^{(3)} \cdot \sigma W^{(2)} \cdot \sigma W^{(1)} \cdot (X)$ Get Linear Combination (weighted sum), and apply the activation function to get output prediction $\hat{y}$

② $L(W,b) = \frac{1}{N} \sum_{i=1}^{N} (\hat{y}_i - y_i)^2$ COMPARE THE PREDICTION WITH THE LABEL TO CALCULATE THE ERROR OR LOSS FUNCTION.

③ $\Delta W_i = -\alpha \frac{dE}{dW_i}$ USE A GRADIENT DESCENT OPTIMIZATION ALGORITHM TO COMPUTE THE ΔW THAT OPTIMIZES THE ERROR FUNCTION.

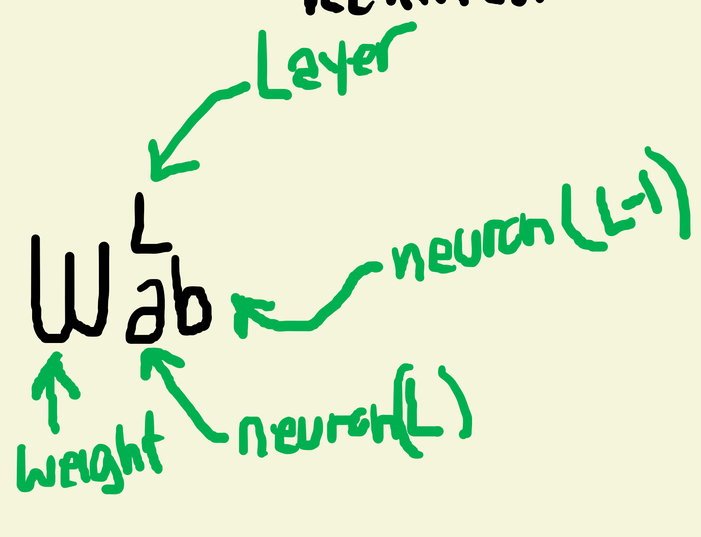
THEN...

④ BACKPROPAGATE THE ΔW THROUGH THE NETWORK TO UPDATE THE WEIGHTS.

$$W_{new} = W_{old} - \left(\frac{dError}{dW_x}\right)$$

- W_{new} ← new weight
- W_{old} ← old weight
- $\left(\frac{dError}{dW_x}\right)$ ← derivative of error with respect to weight.
- Learning rate.

Reminder



BACKPROPAGATION

PROPAGATING DERIVATIVES OF THE ERROR WITH RESPECT TO EACH SPECIFIC WEIGHT dE/dw_i FROM THE LAST LAYER (output) BACK TO the first layer (inputs) to adjust weights.

QUESTION: HOW DO WE COMPUTE THE CHANGE OF THE TOTAL ERROR WITH RESPECT TO w_{13} $\left(\frac{dE}{dw_{13}}\right)$? RECALL, $\frac{dE}{dw_{13}}$ BASICALLY SAYS "HOW MUCH WILL THE TOTAL ERROR CHANGE WHEN WE CHANGE THE PARAMETER w_{13}.

WE NEED TO APPLY CHAIN RULE IN ORDER TO COMPUTE THE DERIVATIVES OF THE TOTAL ERROR WITH RESPECT TO THE WEIGHTS ALL THE WAY BACK TO THE INPUTS.

CALCULUS REFRESHER: CHAIN RULE IN DERIVATIVES.

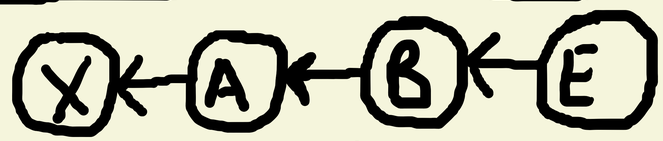
Chain Rule: $\frac{d}{dx} f(g(x)) = f'(g(x)) g'(x)$. Remember $F'(x) = \frac{df}{dx}$

$$\frac{d}{dx} f(g(x)) = \frac{d}{dx} \text{outside function} \cdot \frac{d}{dx} \text{inside function}$$

$$= \frac{d}{dx} f(g(x)) \cdot \frac{d}{dx} g(x)$$

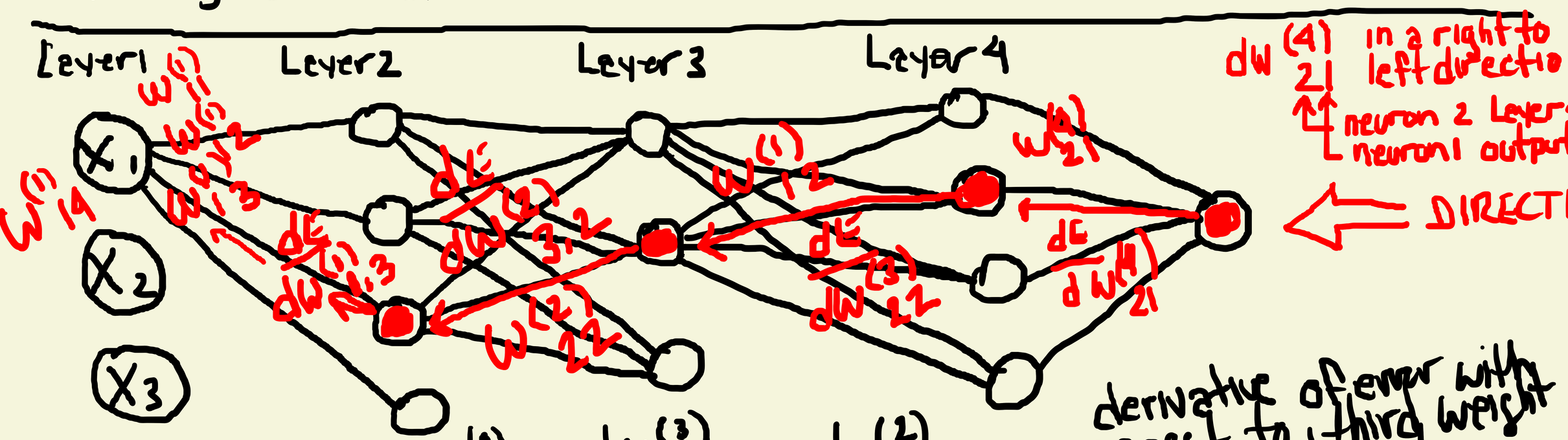
* feedforward is function composition while backpropagating is taking

derivatives of functions...



$$\frac{dE}{dx} = \frac{dE}{dB} \cdot \frac{dB}{dA} \cdot \frac{dA}{dx}$$

So, all this does is to multiply the upstream gradient by the local gradient all the way until we get to the target value.



$$\frac{dE}{dw^{(1)}_{1,3}} = \frac{dE}{dw^{(4)}_{21}} \cdot \frac{dw^{(4)}_{21}}{dw^{(3)}_{22}} \cdot \frac{dw^{(3)}_{22}}{dw^{(2)}_{32}} \cdot \frac{dw^{(2)}_{32}}{dw^{(1)}_{13}}$$

derivative of error with respect to third weight on the first input $W^{(1)}_{1,3}$

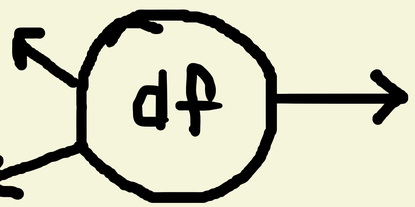
So.. FORWARD PASS.

calculate output prediction

BACKWARD PASS

$$\frac{dL}{dx} = \frac{dL}{dz} \cdot \frac{dz}{dx}$$

$$\frac{dL}{dy} = \frac{dL}{dz} \cdot \frac{dz}{dy}$$



pass derivative of error backward to update the weights.

We want to minimize the cost or loss function by choosing the best set of weight parameters.

Hyperparameters (example): number of layers, activation functions, loss functions, optimizers, early stopping, learning rate. We tune these before training the model.

Convolutional Neural Networks

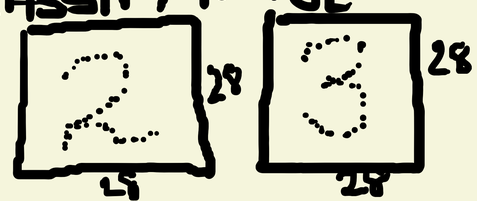
- classifying images using MLP (MULTI LAYER PERCEPTRON)
- Working with the CNN Architecture to classify images.
- understanding convolution on colored images.

PREVIOUSLY DESCRIBED WERE ARTIFICIAL NEURAL NETWORKS (ANN's)
CONVOLUTIONAL NEURAL NETWORKS (CNN's) ARE AN EVOLUTION OF ANN's
THAT PERFORM BETER ON IMAGES.

MAIN COMPONENTS OF CNN's.
a) CONVOLUTIONAL LAYER
b) POOLING LAYER
c) FULLY CONNECTED LAYER.

IMAGE CLASSIFICATION PROBLEM

CLASSIFY IMAGES OF DIGITS FROM 0 TO 9 (10 CLASSES).



INPUT LAYER

NEED TO CLASSIFY IMAGES INTO SOMETHING THE NETWORK CAN UNDERSTAND.
IMAGES AS 28x28 matrix of pixels @ pixel values ranging from 0 → 255
(0 = black, 255 = white, gray scale in between.

NOW, INPUT FOR MLP'S ARE 1D VECTORS. (1x) (1,p) WE NEED TO TRANSFORM THE IMAGE from (x,y) to (1,p) with all of the pixel data going into the 1D vector. Vector will be (1,784)
So the input layer in this example will have 784 nodes.
*1 SEE C:\GIT\Test\Test.py for an example of Flattening in Keras.

HIDDEN LAYERS

FOR EXAMPLE 2 HIDDEN LAYERS WITH 512 NEURONS EACH. DON'T FORGET TO ADD ReLU ACTIVATION FUNCTION FOR EACH HIDDEN LAYER.

CHOOSE ACTIVATION FUNCTION.

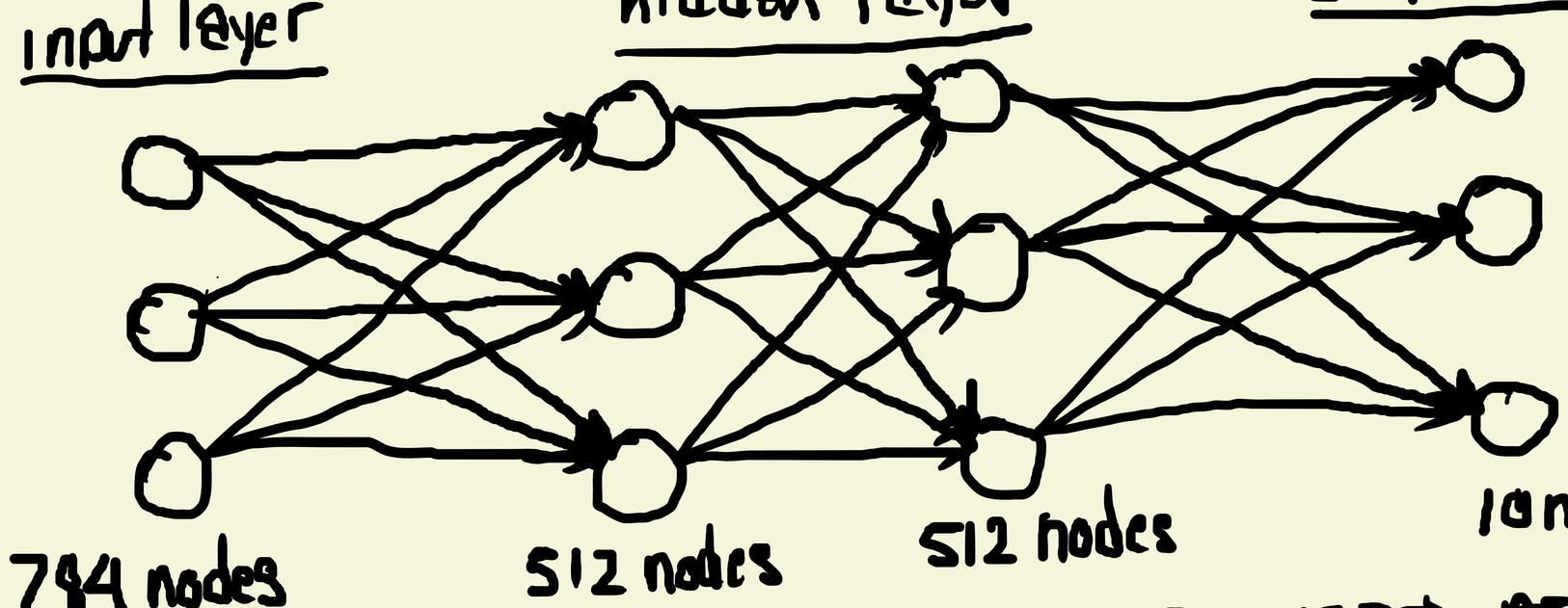
IN GENERAL WE WILL USE ReLU IN THE HIDDEN LAYERS AND SOFTMAX IN THE OUTPUT. SOFTMAX WILL GIVE PROBABILITY THAT THE INPUT IMAGE DEPICTS ONE OF THE CLASSES. (*1)

OUTPUT LAYER

THE NUMBER OF OUTPUT LAYERS SHOULD BE EQUAL TO THE NUMBER OF CLASSES WE ARE TRYING TO DETECT. SO FOR THE DIGITS EXAMPLE ABOVE WE WOULD HAVE 10 OUTPUT LAYERS. (*1)

PUTTING IT ALL TOGETHER

THE ABOVE WILL GIVE US A NEURAL NETWORK LIKE THIS....



THIS NETWORK CLASSIFIES THE IMAGE REPRESENTED AS (1, 784) VECTOR, EACH PIXEL APPEARING AS AN INPUT NODE, INTO THE 10 OUTPUTS.

NUMBER OF PARAMS (WEIGHTS) PRODUCED AT EACH LAYER IS 669706

Breakdown.

Params after Layer 1. (784 nodes in input) · (512 nodes hidden) + (512 biases)
(784 · 512) + 512 = 401920

Params after Layer 2. (512 nodes from layer 1) * (512 nodes from layer 2) + (512 biases)
(512 · 512) + 512 = 262,656

Params after Layer 3 (512 nodes layer 2) * (10 output layer) + 10 (biases)
(512 · 10) + 10 = 5130

Total params in network $401920 + 262656 + 6130 = 669,706$

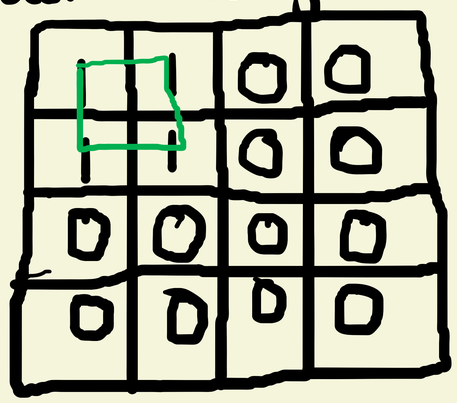
MAJOR DRAWBACK OF MLP (Multilayer Perceptron) is NETWORK SIZE.

MLP's VS. CNN's

DRAWBACKS OF MLP's FOR PROCESSING IMAGES.

1) 1D VECTOR INPUT REQUIREMENT THROWS AWAY SPATIAL INFORMATION. IN A CNN NETWORK WE CAN FEED IN THE RAW IMAGE MATRIX. THE CNN WILL UNDERSTAND THAT PIXELS THAT ARE CLOSE TOGETHER ARE MORE HEAVILY RELATED THAN THOSE THAT ARE FAR APART.

1 = white pixel, 0 = black pixel. MATRIX IMAGE with white square on black background.



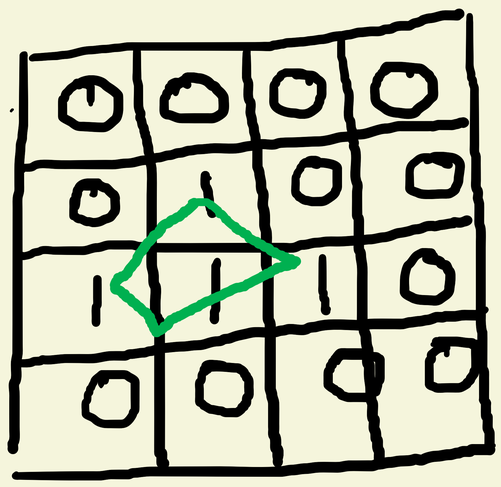
Now, since MLP's flatten this to 1D vector, we wind up with....

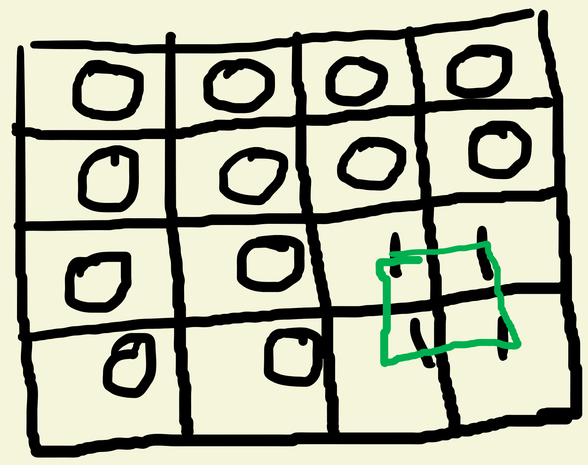
[1,1,0,0,1,1,0,0,0,0,0,0,0,0,0,0]

These are the inputs $x_1, \ldots, x_{16}$.

WHEN THE TRAINING IS COMPLET THE NETWORK WILL ONLY IDENTIFY THE WHITE SQUARES WHEN x_1, x_2, x_5, x_6 ARE FIRED.

MLP WILL FAIL HERE AND HERE

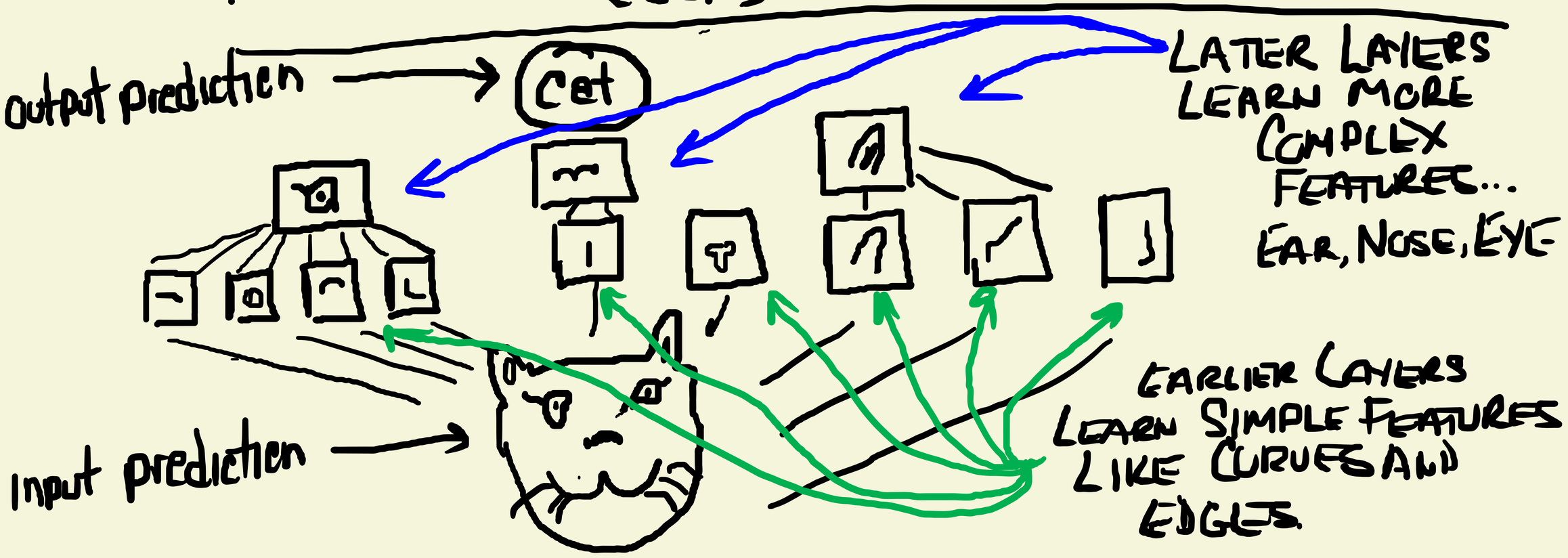




THE MLP WILL HAVE NO IDEA THAT THESE ARE THE SHAPES OF SQUARES BECAUSE THE NETWORK DIDN'T LEARN THE SQUARE SHAPE AS A FEATURE. INSTEAD, IT LEARNED THE INPUT NODES THAT, WHEN FIRED, MIGHT LEAD TO A SQUARE SHAPE. IF WE NEED OUR NETWORK TO LEARN SQUARES THEN WE WOULD NEED A LOT OF SQUARE SHAPES LOCATED EVERYWHERE IN THE IMAGE. * PROBLEM DOESN'T SCALE WELL**.

WE NEED THE NETWORK TO LEARN THE SHAPES OF FEATURES WHEREVER THEY MAY OCCUR. THIS CAN ONLY HAPPEN WITH THE NETWORK LOOKS AT THE IMAGE AS A SET OF PIXELS WHICH, THE PROXIMITY OF NEARBY PIXELS, IS RELATED.

HOW A CONVOLUTIONAL NETWORK (CNN) LEARNS FEATURES.

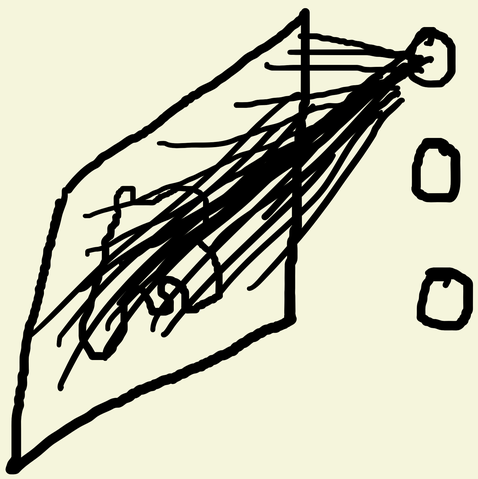


Fully Connected (Dense) Layers.

MLP's are composed of dense layers where every node in one layer is connected to every node in the previous layer. For a 1,000 x 1,000 image we will have billions of parameters. No Good.

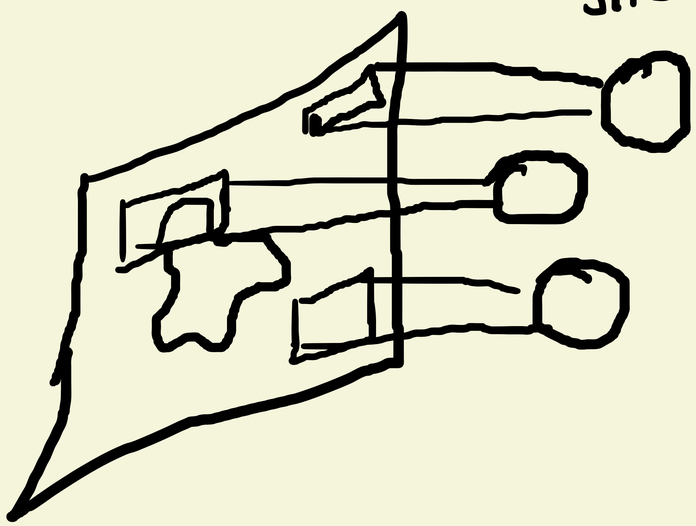
CNN's are locally connected layers. Nodes are connected to a small subset of a previous layers nodes. Locally connected layers use far fewer parameters than densely connected layers.

Fully Connected



Locally Connected

sliding windows



CNN Architecture

The main difference between CNN & MLP is that CNN layers learn successively more pronounced features. Basic features in early layers and more complex features in later layers

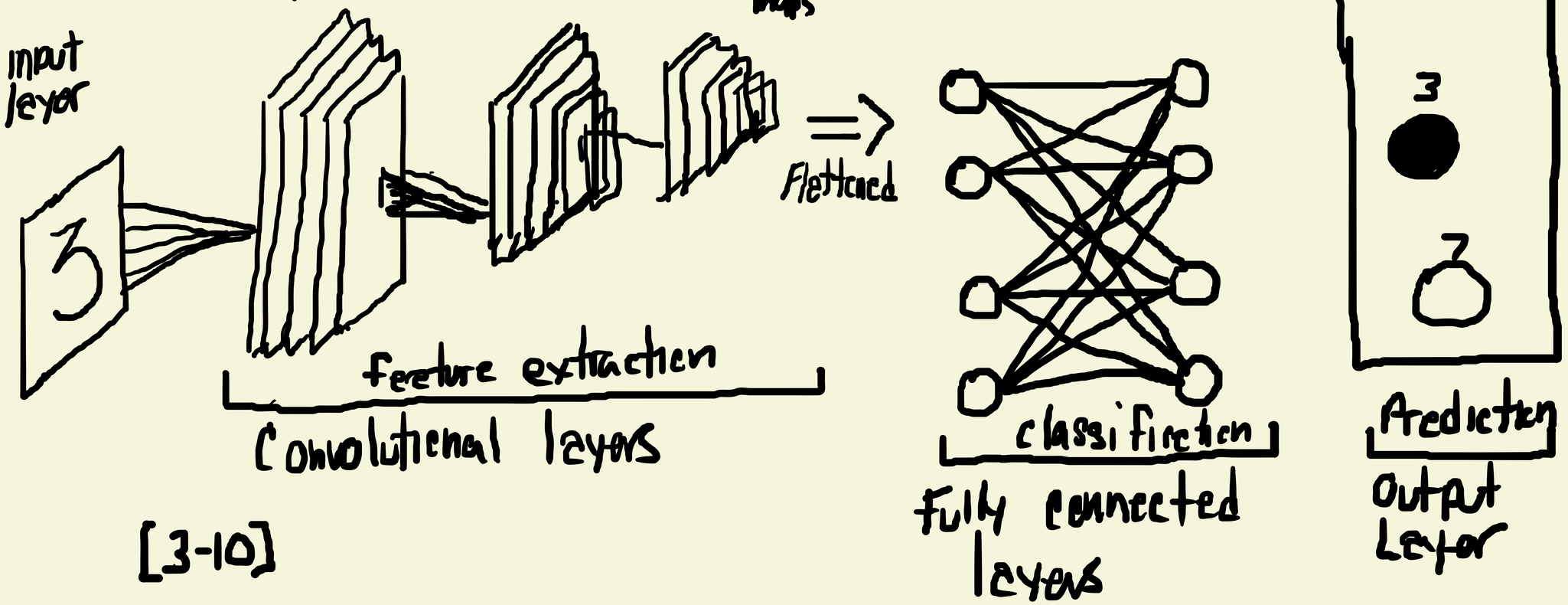
- images
- videos image frames

getting data ready
standardize images
color transformation

finding distinguishing information about image

learn from extracted features to predict + classify objects

WE WILL USE CNN FOR STEP #3 (FINDING FEATURES)
WE WILL USE FULLY CONNECTED LAYERS FOR STEP #4 (CLASSIFICATION)



[3-10]

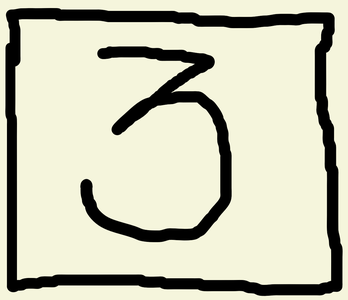
Summary:
① feed raw image to convolutional layers.
② The image passes through the conv layers to detect patterns and extract features called feature maps. The output of this step is flattened to a vector of the learned features of the image.

* THE IMAGE DIMENSIONS SHRINK AFTER EACH LAYER AND THE NUMBER OF FEATUREMAPS (LAYER DEPTH) INCREASES UNTIL WE HAVE A LONG ARRAY OF SMALL FEATURES IN THE LAST LAYER OF THE FEATURE-EXTRACTION PART.

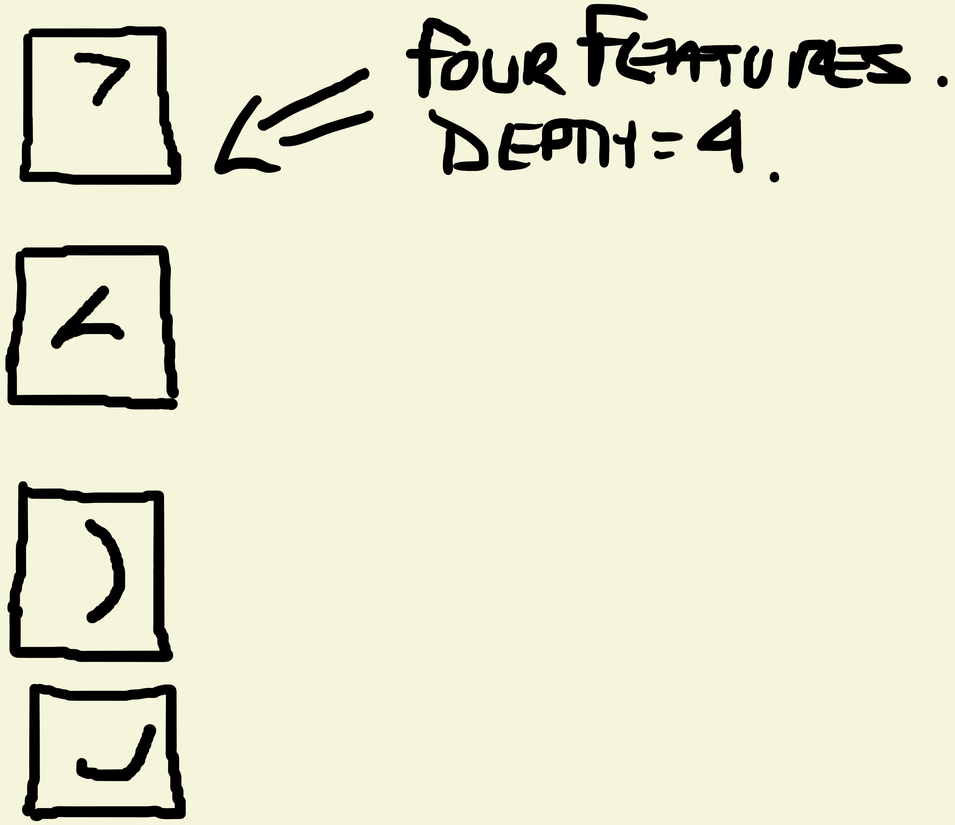
③ THE FLATTENED FEATURE VECTOR IS FED TO THE FULLY CONNECTED LAYER TO CLASSIFY THE EXTRACTED FEATURES OF THE IMAGE.

④ THE NEURAL NETWORK FIRES THE NODE THAT REPRESENTS THE CORRECT PREDICTION OF THE IMAGE.

* THE FEATURE MAP IS A MAPPING OF WHERE A CERTAIN KIND OF FEATURE IS FOUND IN THE IMAGE. CNN'S LOOK FOR STRAIGHT LINES, EDGES, OR EVEN OBJECTS. WHENEVER THEY FIND THEM THEY REPORT THEM TO THE FEATURE MAP. EACH FEATURE MAP IS LOOKING FOR SOMETHING SPECIFIC; ONE COULD BE LOOKING FOR STRAIGHT LINES AND ANOTHER FOR CURVES.



FEATURE EXTRACTION



FOUR FEATURES.
DEPTH = 4.

CNN'S EXTRACT MEANINGFUL FEATURES THAT SEPARATE THIS OBJECT FROM OTHER IMAGES IN THE TRAINING SET AND STACK THEM IN AN ARRAY OF FEATURES.

CLASSIFICATION

Layer 1 : Learns patterns
Layer 2 : Learns patterns within patterns.
Layer 3 : Learns patterns within patterns within patterns.
... and so forth.

Basic Components of a CNN

3 main types of layers.

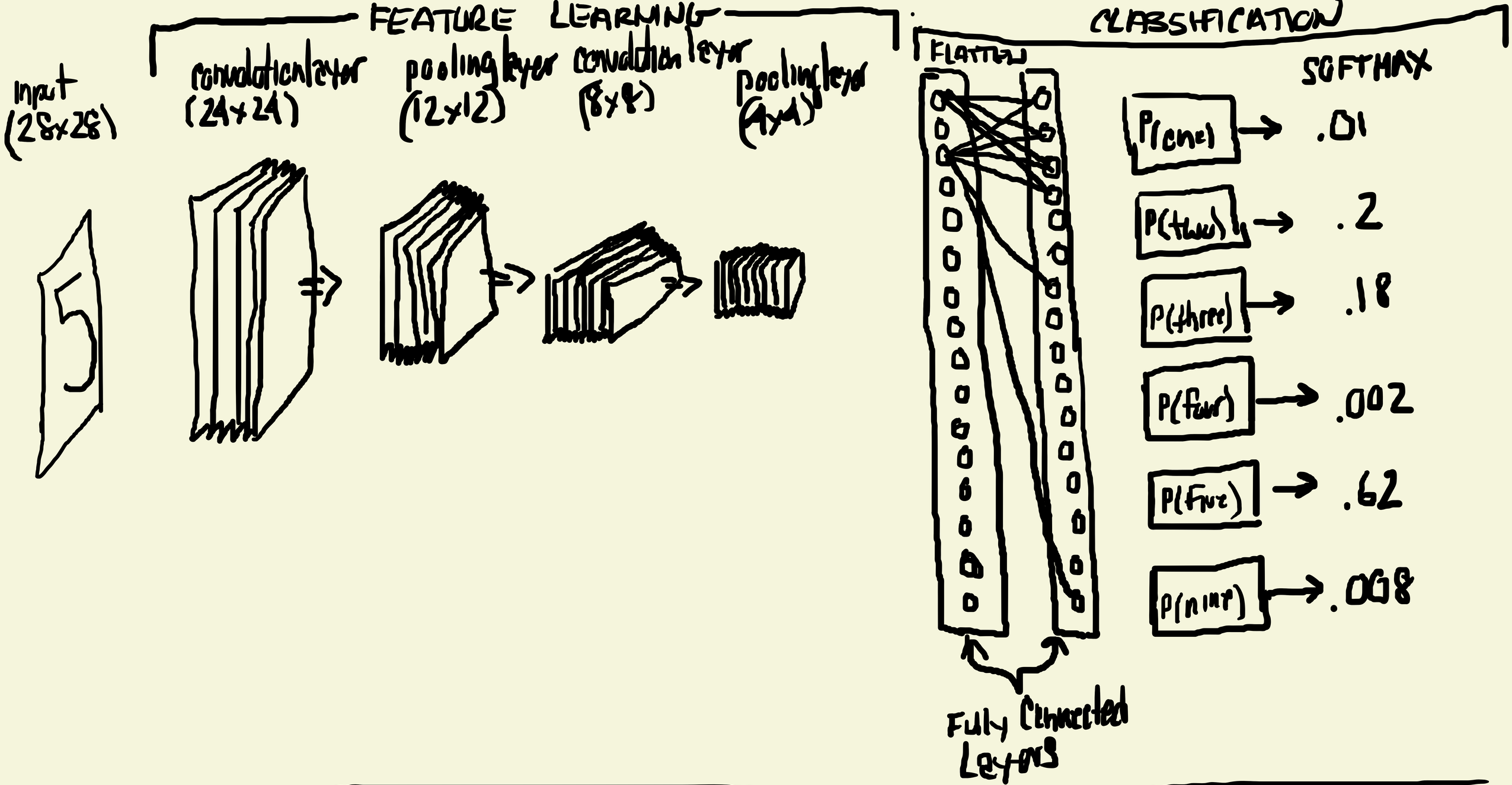
1. Convolutional layer (CONV)
2. Pooling Layer (POOL)
3. Fully Connected Layer (FC)

CNN Text REPRESENTATION

CNN ARCHITECTURE: INPUT -> CONV -> RELU -> POOL -> CONV -> RELU -> POOL -> FC -> SOFTMAX

INPUT: input layer
CONV -> RELU -> POOL: convolutional layer
CONV -> RELU -> POOL: convolutional layer
FC -> SOFTMAX: fully connected layer

The convolutional layers are for feature learning or extraction and the fully connected layer is for classification.



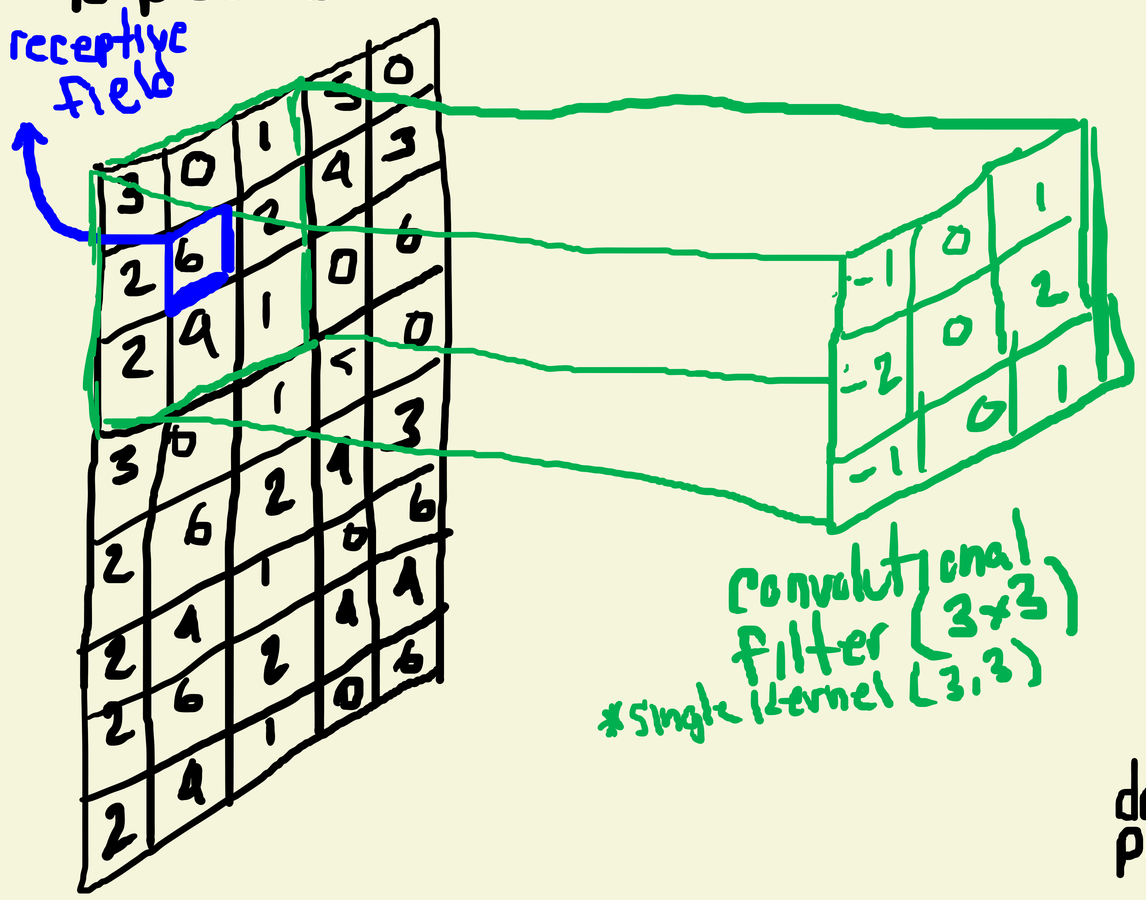
CONVOLUTIONAL LAYERS

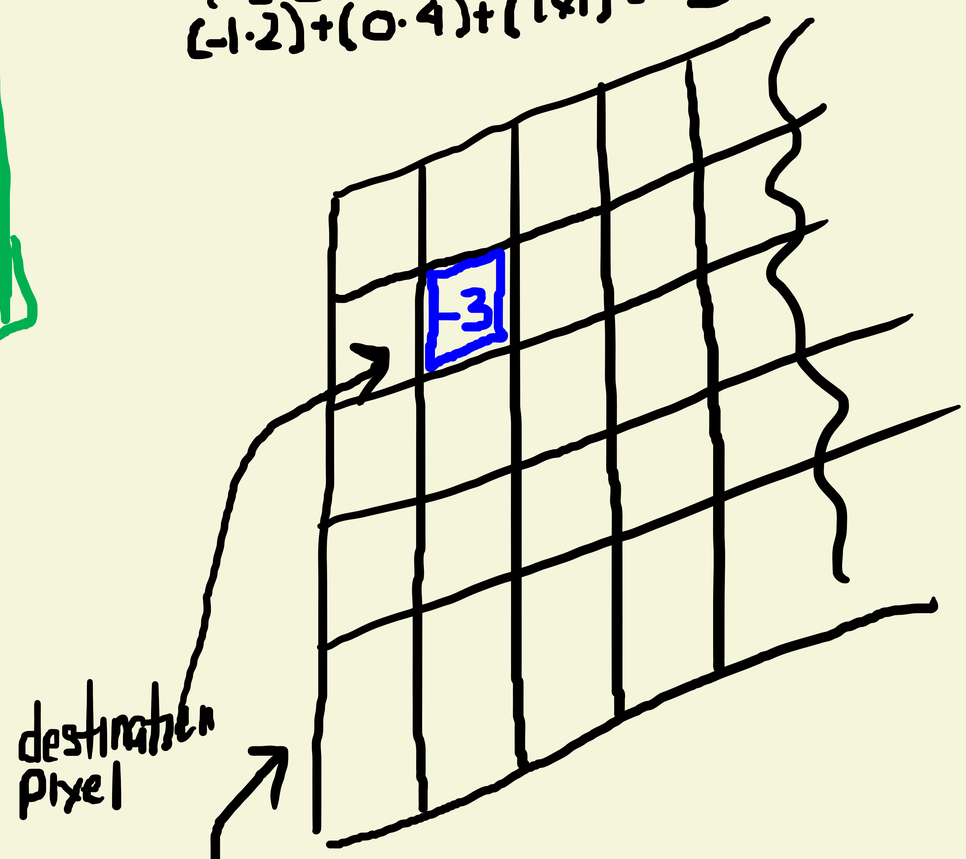
CONVOLUTIONAL LAYERS ACT LIKE A FEATURE FINDER WINDOW THAT SLIDES OVER THE IMAGE PIXEL BY PIXEL TO EXTRACT MEANINGFUL FEATURES THAT IDENTIFY THE OBJECTS IN THE IMAGE.

WHAT IS CONVOLUTION?

IN MATHEMATICS CONVOLUTION IS THE OPERATION OF TWO FUNCTIONS TO PRODUCE A THIRD MODIFIED FUNCTION.
IN THE CONTEXT OF CNN THE FIRST FUNCTION IS THE INPUT IMAGE and the second function is the convolutional filter. We perform some mathematical operations to produce a modified image with new pixel values.

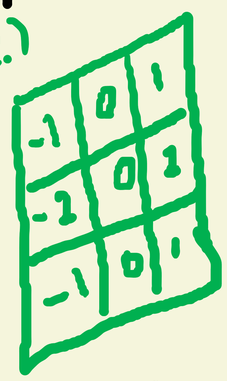
$(-1\cdot3)+(0\cdot0)+(1\cdot1)+$
$(-2\cdot2)+(0\cdot6)+(2\cdot2)+$
$(-1\cdot2)+(0\cdot4)+(1\times1) = -3$





BY SLIDING THE CONVOLUTIONAL FILTER OVER THE INPUT IMAGE, THE NETWORK BREAKS THE IMAGE INTO LITTLE CHUNKS AND PROCESSES THOSE CHUNKS INDIVIDUALLY TO ASSEMBLE THE MODIFIED IMAGE, A FEATURE MAP OR ACTIVATION MAP

- THE SMALL 3x3 MATRIX ABOVE IS THE CONVOLUTIONAL FILTER ALSO CALLED A KERNEL.
- THE KERNEL SLIDES OVER THE ORIGINAL IMAGE PIXEL BY PIXEL AND DOES SOME MATH CALCULATIONS TO GET THE VALUES OF THE NEW CONVOLVED IMAGE.
- IN CNN'S THE CONVOLUTION MATRIX IS THE WEIGHTS. THIS MEANS THEY ARE RANDOMLY INITIALIZED AND THEIR VALUES ARE LEARNED BY THE NETWORK. (12.)



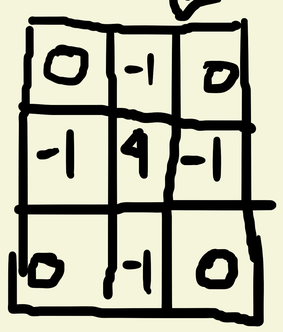
CONVOLUTIONAL FILTER (3x3) KERNEL.

Weighted sum: $(x_1 \cdot w_1) + (x_2 \cdot w_2) + (x_3 \cdot w_3) + \dots + (x_n \cdot w_n)$

HOW DOES APPLYING FILTERS LEAD TO FEATURE EXTRACTION?

IN IMAGE PROCESSING FILTERS ARE USED TO FILTER OUT UNWANTED INFORMATION OR AMPLIFY FEATURES IN AN IMAGE.

FOR EXAMPLE..



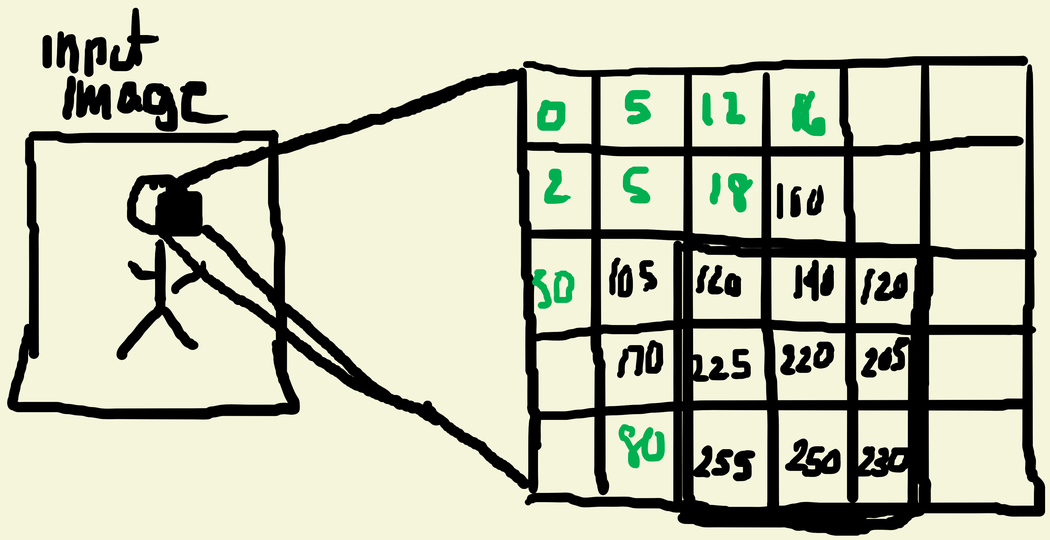
← Edge detection filter

image input

convolved image

← Supposed to be image after edge detection applied. Try this in paint.net with an image.

HERE IS HOW EDGE KERNEL WORKS ON IMAGE DATA.



Edge Detection Kernel

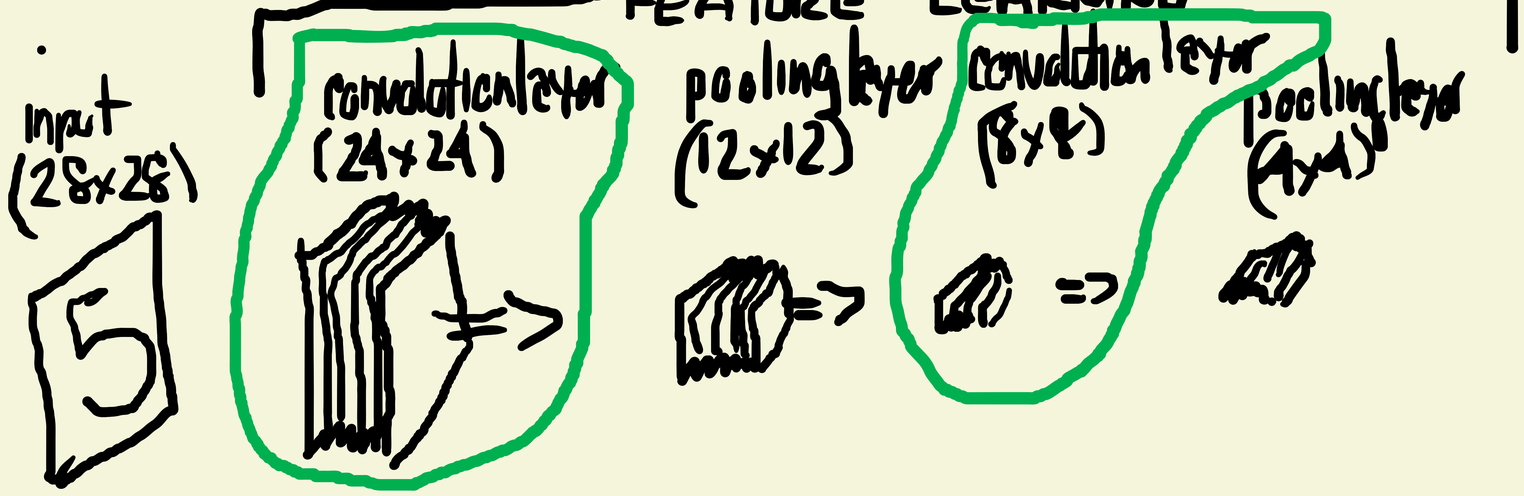
0	-1	0
-1	4	-1
0	-1	0

$(0 \cdot 120)+(-1 \cdot 140)+(120 \cdot 0)+$
$(-1 \cdot 225)+(4 \cdot 220)+(-1 \cdot 205)+$
$(0 \cdot 225)+(-1 \cdot 250)+(0 \cdot 230) = 60$

THE NEW VALUE OF THE MIDDLE PIXEL IN THE CONVOLVED IMAGE IS 60. THE PIXEL VALUE IS >0 WHICH MEANS THAT A SMALL EDGE HAS BEEN DETECTED.

OTHER FILTERS CAN BE USED TO DETECT DIFFERENT TYPES OF FEATURES... (IE) HORIZONTAL EDGES, VERTICAL EDGES, CORNERS, ETC.

EACH CONVOLUTIONAL LAYER CONTAINS ONE OR MORE CONVOLUTIONAL FILTERS.



THE NUMBER OF FILTERS (KERNELS/CONVOLUTIONAL FILTERS) DETERMINES THE DEPTH OF THE NEXT LAYER, BECAUSE EACH KERNEL PRODUCES ITS OWN FEATURE MAP (CONVOLVED IMAGE).

* SEE convtest.py

model.add(Conv2D(filters=16, kernel_size=2, strides=(1,1), padding='same', activation='relu'))

Filters: The number of convolutional filtes in each layer. This represents the depth of the output.

Kernel Size: The size of the convolutional matrix. (ie) 2x2, 3x3, 5x5.

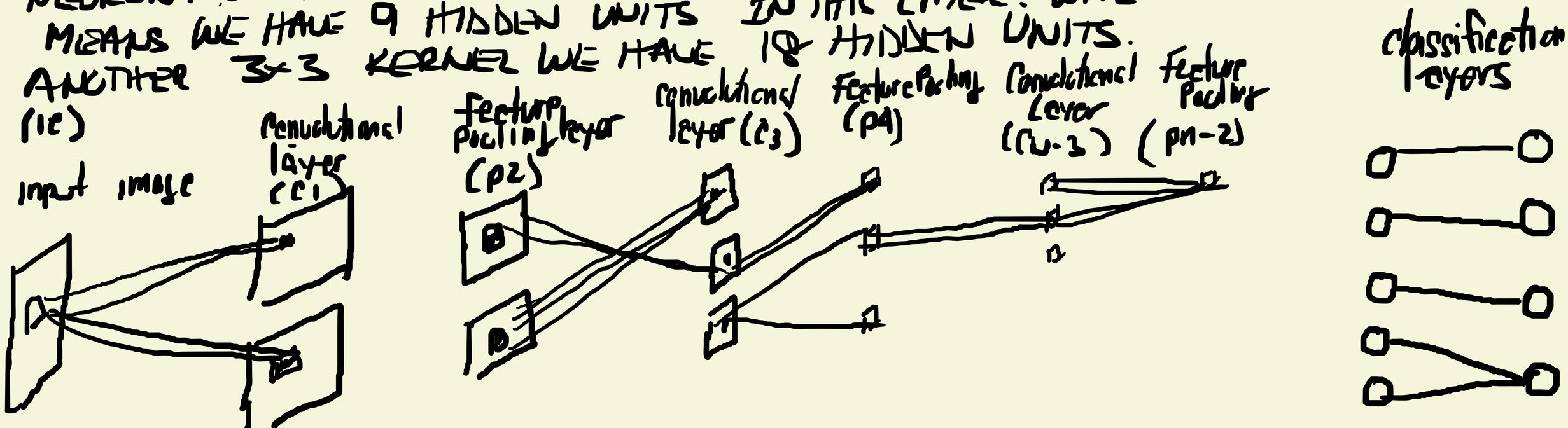
stride: TBD

Padding: TBD

Number of Filters in Convolutional Layer

WITH CNN'S THE CONVOLUTIONAL LAYERS ARE THE HIDDEN LAYERS.
SO, LIKE MLP'S WHERE WE INCREASE THE NUMBER OF NEURONS IN A HIDDEN LAYER, WITH CNN'S WE INCREASE THE NUMBER OF KERNELS IN A CONVOLUTIONAL LAYER. EACH KERNEL UNIT IS CONSIDERED A NEURON. (IE) IF WE HAVE A 3x3 KERNEL IN A CONVOLUTIONAL LAYER THIS MEANS WE HAVE 9 HIDDEN UNITS IN THIS LAYER. WHEN WE ADD ANOTHER 3x3 KERNEL WE HAVE 18 HIDDEN UNITS.

(IE)



CONVOLUTIONAL FILTER = KERNEL. IT IS A MATRIX OF WEIGHTS THAT SLIDES OVER THE IMAGE TO EXTRACT FEATURES.

THE KERNEL SIZE REFERS TO THE DIMENSIONS OF THE CONVOLUTION FILTER (width times height). (ie) 3x3

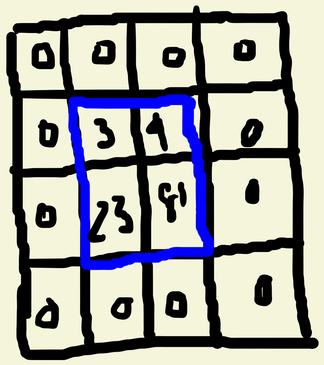
* smaller filters will capture very fine detail & bigger filters will miss minute details in the image.

Since filters contain weights that will be learned by the network, so the bigger the kernel_size, the deeper the network which means the better it learns. They range from 2x2 to 5x5

strides & padding

strides: The amount by which the filter slides over the image. 1 pixel at a time, the strides value is 1 etc. strides of 1 will make the output image roughly the same size as the image. A stride of 2 will make the output roughly half.

Padding: often called zero padding because zero's are added around the border.



This would be used in the case that input & output volume needs to be the same. Idea is to use this in convolutional layers for building deeper networks without shrinkage.

Pooling Layers (Subsampling)

Subsampling/pooling helps reduce the size of the network by reducing the number of parameters that are passed to subsequent layers.

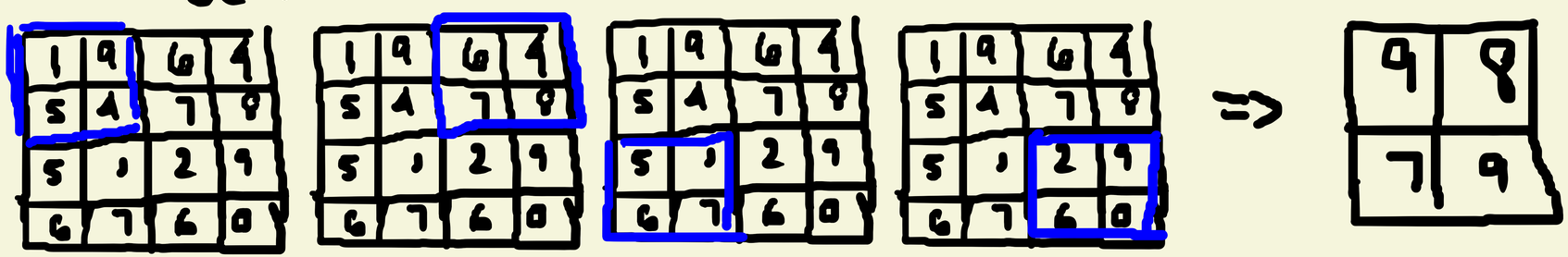
- The pooling operation resizes its input by applying a summary statistical function (ie) maximum or average. In order to reduce the number of parameters passed to the next layer.
- subsampling down-samples the feature maps produced by the convolutional layer into a smaller number of parameters thus reducing computational complexity.
- It is common practice to add pooling layers after every one or two convolutional layers.

Max Pooling Vs Average Pooling.

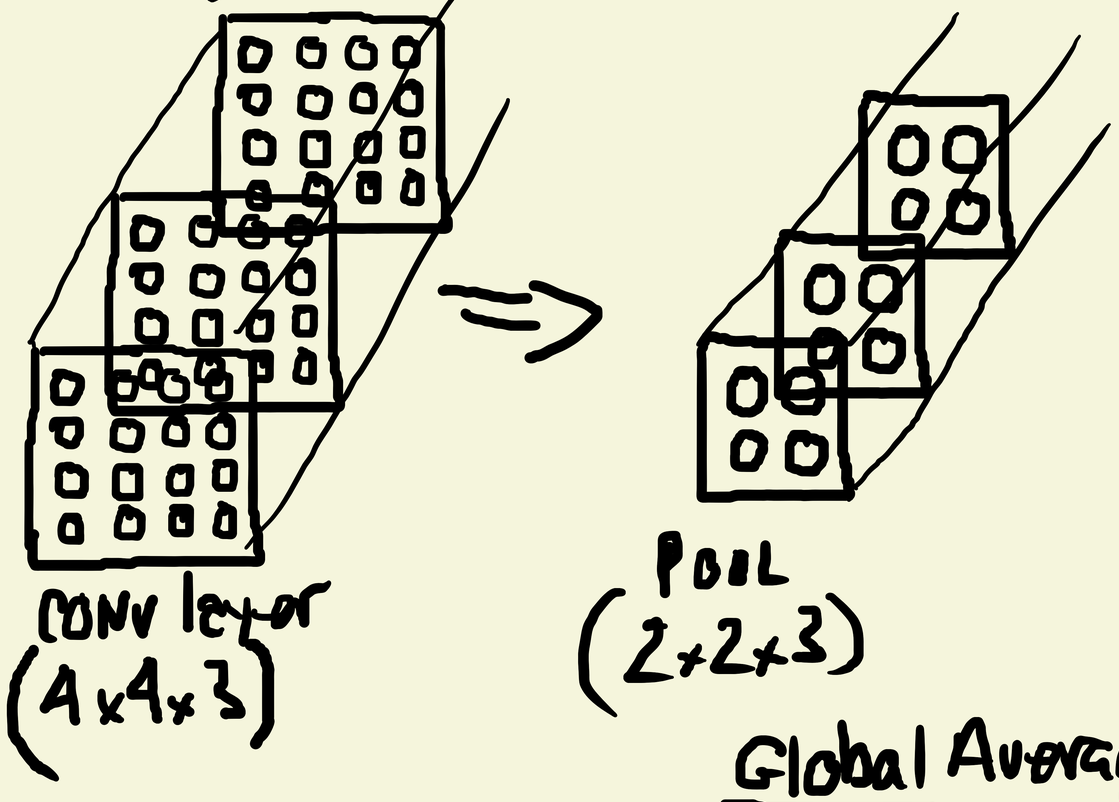
2 main types of pooling layers. max pooling & average pooling.

MaxPooling: Max pooling uses kernels similar to convolutional kernels. They don't have weights or values they just slide over the feature map created by the previous convolutional layer and Select the max pixel value to pass along to the next layer. A pooling filter with size 2x2 and strides of two will reduce a feature map of 4x4 down to 2x2.

(ie)

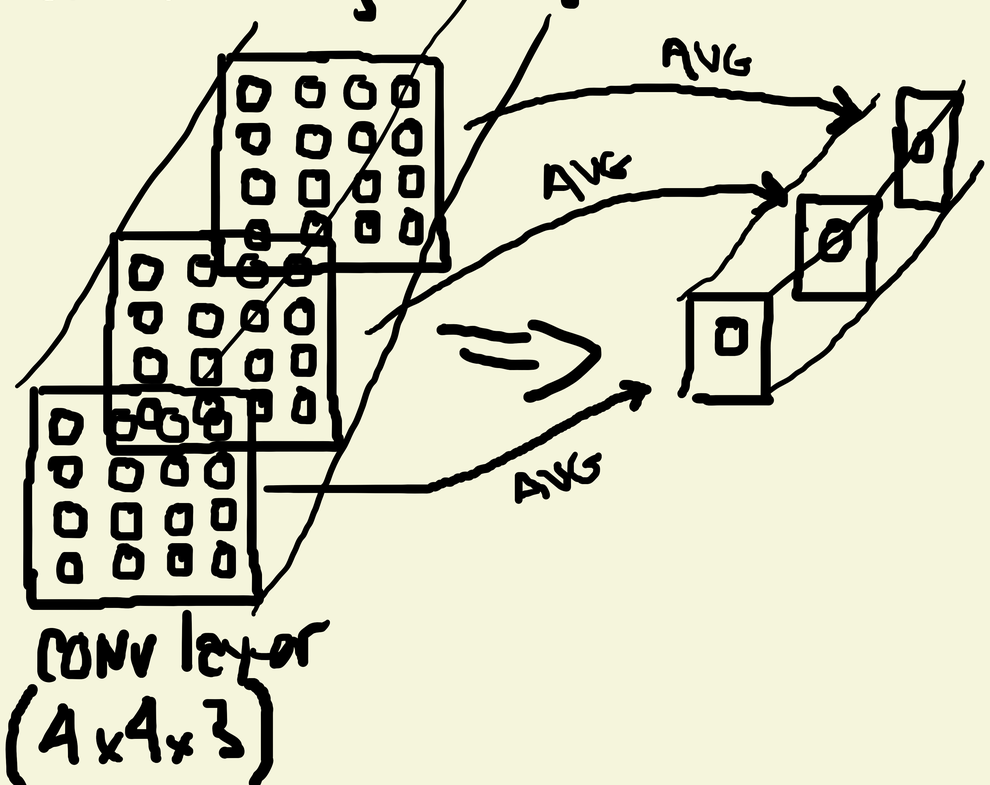


Pooling does not change the number of feature maps... just their size.



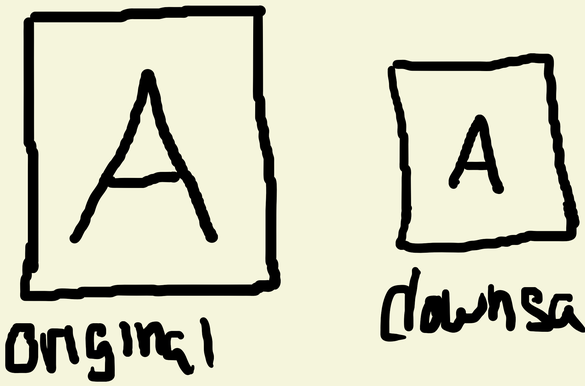
Global Average Pooling

Global Average Pooling takes an average of all pixels in the feature map. So it takes a 3D array and turns it into a vector.



Why Use Pooling Layers?

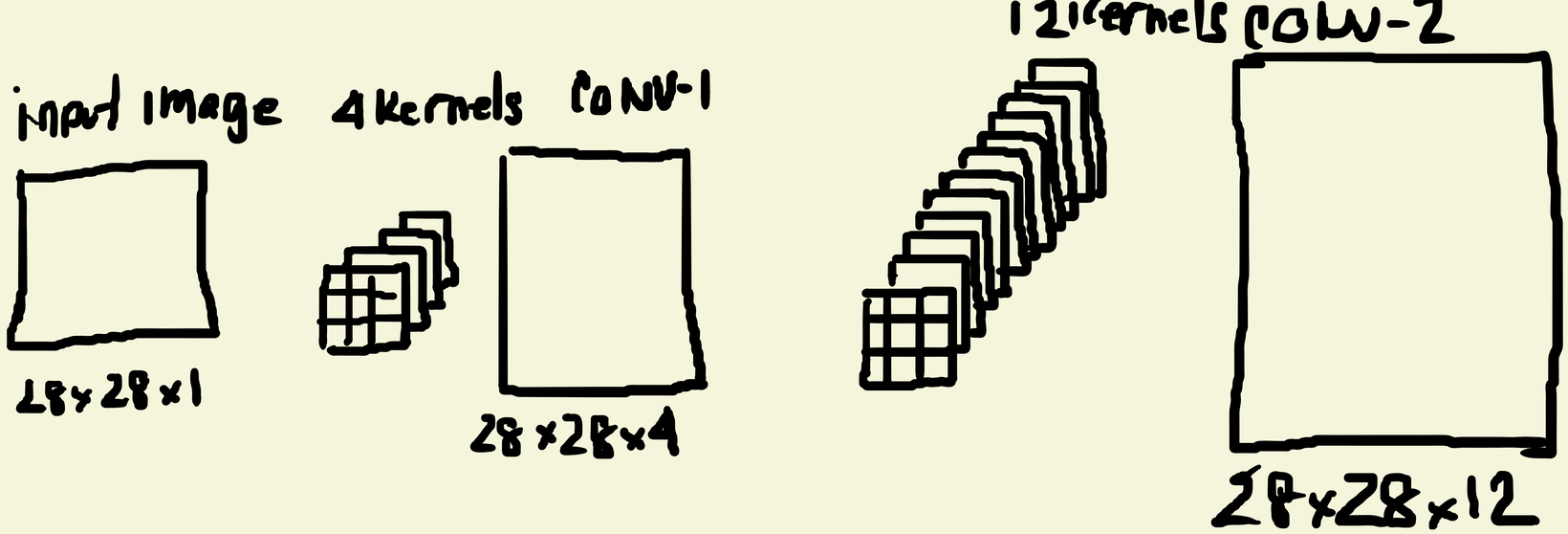
- Pooling reduces dimensionality
- Reduces complexity.. pooling is like image-compression.



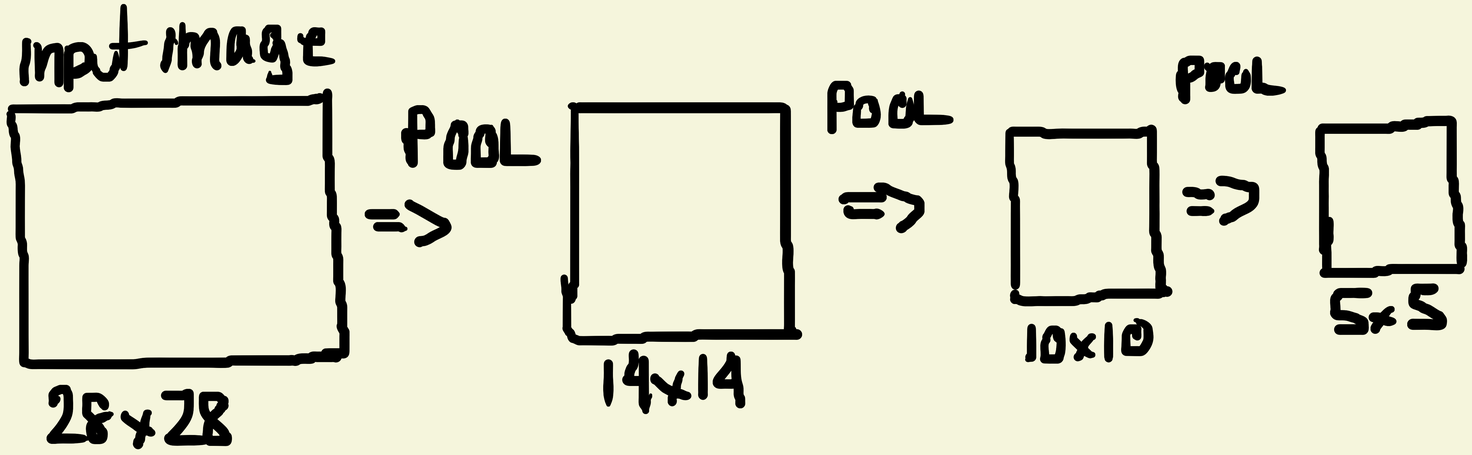
original

downsampled.

After the convolutional layers the image keeps its dimensions but gets deeper a deeper after each layer.

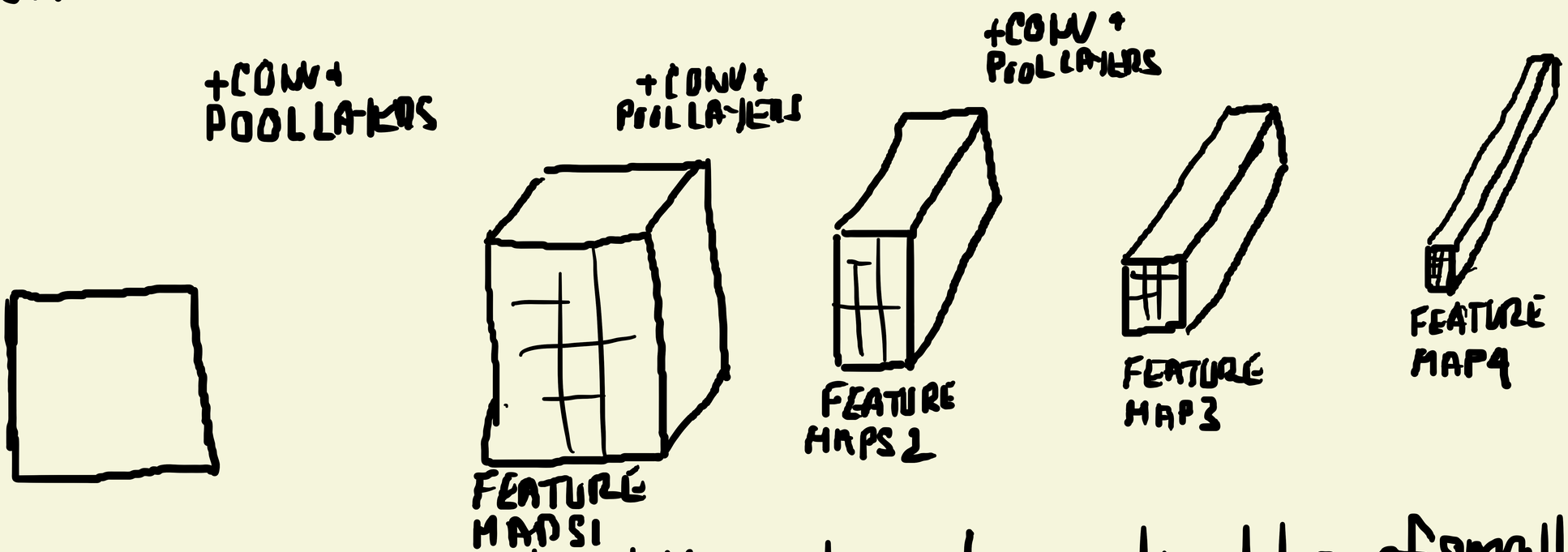


After pooling



AFTER THE POOLING LAYERS THE IMAGE KEEPS ITS DEPTH BUT SHRINKS IN WIDTH & HEIGHT

PUTTING CONVOLUTIONAL & POOLING Together.

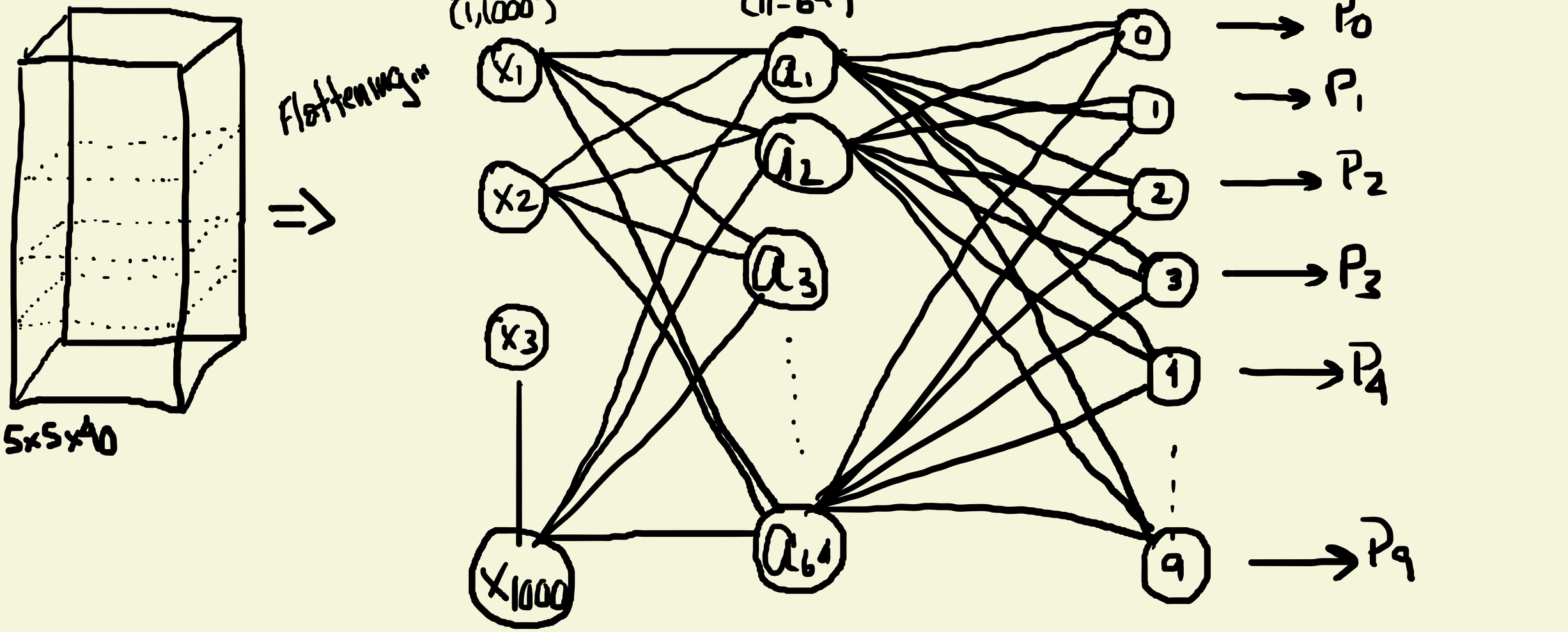


This keeps happening until at the end, we have a long tube of small shaped images that contain all the features in the original image.

The flattened layer will have the dimensions of (1, M) where M = 5x5x40
= 1,000 neurons.

So now once we have this tube of extracted features we then use it to classify images using the regular neural network architecture (MLP)

classification



* MLP's ARE CALLED FULLY CONNECTED LAYERS. ALSO CALLED DENSE LAYERS. (MLP, FULLY CONNECTED, DENSE, FEED FORWARD).

3.4! Building The Model Architecture.

See: c:\git\Deeplearning\Model.py

How are parameters calculated?

number of params = filters × Kernel size × depth of previous layer + number of filters (for biases)

Dropout Layers To Avoid Overfitting

underfitting: model too simple to fit training data.
overfitting: fitting the data too much. (ie) memorizing the training data.

Dropout layers prevent overfitting by turning off a percentage of neurons (nodes) that make up a layer of the network. It is a hyper parameter. This way it can help eliminate either dominant nodes (that is nodes that have come to dominate the network or nodes that make large mistakes. It helps create a resilient network by ensuring that no node becomes too weak or too strong.
helps reduce inter dependent learning among neurons & promoting ensemble learning.

* we add dropout layers between the fully connected layers.

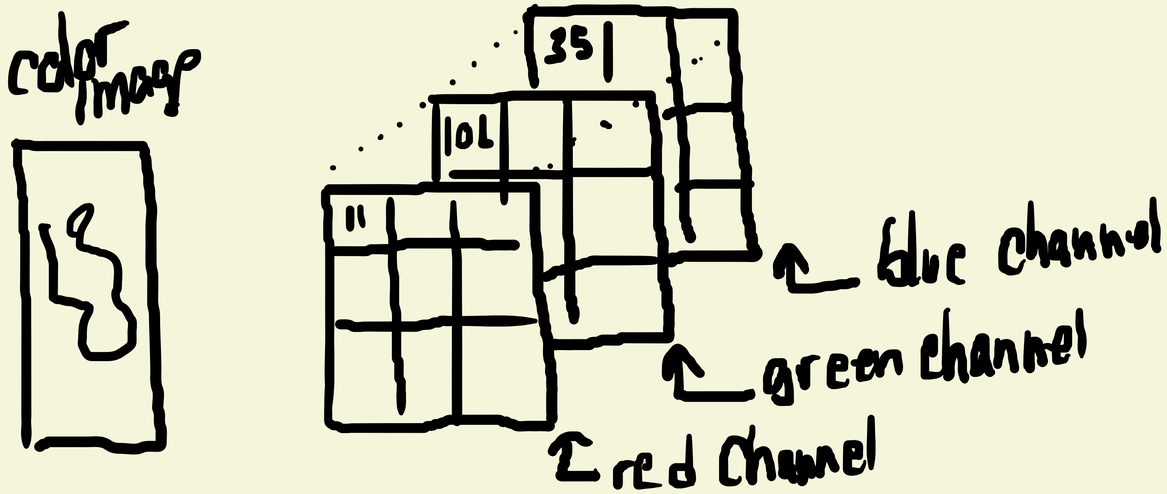
CONV => POOL => Flatten => DO => FC => DO => FC

DO: Drop out

FC: Fully Connected.

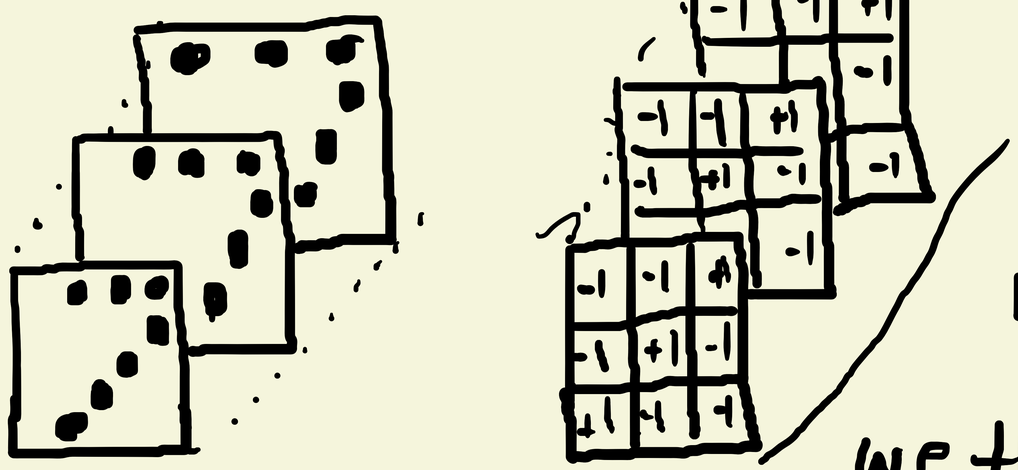
Setting dropout rate to .3 = 30%.

Convolution Over Colored Images (3D Images)



Images represented as 3D Array
height * width * depth
for grayscale depth = 1
for color depth = 3

* we use a 3 dimensional kernel to slide over the image.



* Each color channel has it's own corresponding filter.
* Each filter will slide over its image, multiply every corresponding pixel element wise & then add them all together to produce the convolved image.

We then add the three values to get the value of a single node in the convolved image or

feature map. Don't forget to add bias of 1

7×7

red

0	0	0	0
0	0	1	1
0	2	1	2
0	1	1	2
1	2	0	1

green

0	0	0	0
0	0	1	1
0	0	0	2
2	1	0	0
1	2	0	1

blue

0	0	0	2
0	0	2	1
0	1	2	0
2	0	2	0
1	0	1	1

filter (3×3×2)

0	0	0
1	0	0
1	0	0

$W_0[:,:,1]$

$0\cdot0+0\cdot0+0\cdot0+$
$0\cdot1+0\cdot0+1\cdot0+$
$0\cdot1+2\cdot0+1\cdot0=0$

1	0	-1
-1	1	1
-1	0	1

$W_0[:,:,1]$

$0\cdot1+0\cdot0+0\cdot-1+$
$0\cdot-1+0\cdot1+1\cdot1+$
$0\cdot-1+0\cdot0+0\cdot1=1$

$W_0[:,:,2]$

1	-1	0
1	1	0
0	1	-1

$0\cdot1+0\cdot-1+0\cdot0+$
$0\cdot1+0\cdot1+2\cdot0+$
$0\cdot0+1\cdot1+2\cdot-1=-1$

+1 for bias

1	7	5
4	4	9
5	5	2

← These are actually 7×7 I only drew a few rows/cols.

Color is much denser than grayscale a more computationally demanding.
Certain applications can use grayscale. others. like cancer detection, require color.

The next section is a classification project.
C:\Git\DeepLearning\model_cf.py (will classify object from CIFAR dataset)

*when using gradient descent make sure that all features have a similer scale.

Preparing Labels

Let's say we have labels for images ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck'].

we need to convert text labels into a form that can be processed digitally.

one-hot-encoding →

Image	Label
image_1	dog
image_2	automobile
image_3	airplane
image_4	truck
image_5	bird

	plane	auto	bird	cat	deer	dog	frog	horse	ship	truck
image_1	0	0	0	0	0	1	0	0	0	0
image_2	0	0	0	0	0	0	0	0	0	0
image_3	0	0	0	0	0	0	0	0	0	0
image_4	0	0	0	0	0	0	0	0	0	1
image_5	0	0	1	0	0	0	0	0	0	0

Keras has utilities that do this for us

Training dataset: The sample of data used to train the model
Validation dataset: The sample of data used to provide an unbiased evaluation of model fit on the training dataset while tuning hyperparameters.
Test dataset: The sample of data used to provide an unbiased evaluation of final model fit on the training set.
Refer to C:\GIT\Deep Learning\model_cf.py

Model Architecture

* The more layers the better. Costs: computation complexity
  memory
  overfitting.

* as the image goes through the network its depth increases while its dimensions shrink layer by layer.

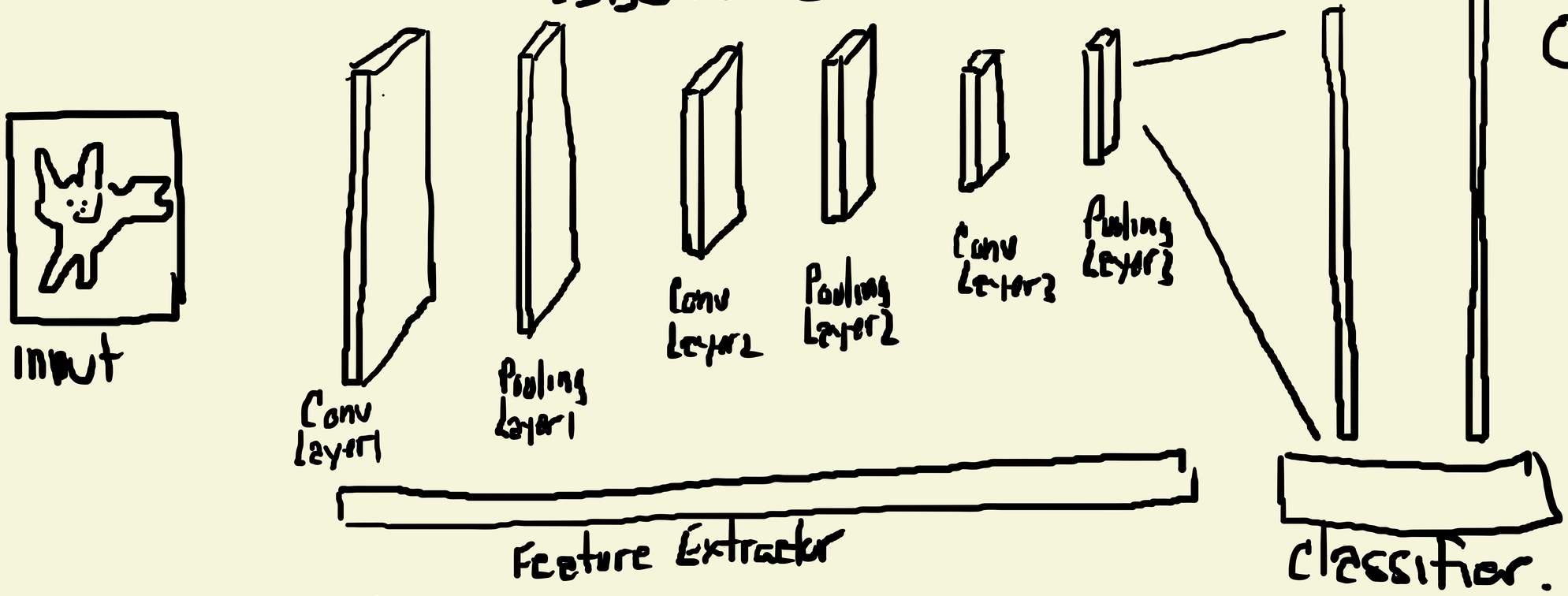
* In general two or three layers of 3×3 convolutional layers followed by a 2×2 pooling can be a good start for smaller data sets. Add more convolutional layers and pooling layers until your image is a reasonable size (4×4) or (5×5) & then add a couple of fully connected layers for classification.

* Need to set up hyperparameters: filters, kernel_size, padding

AlexNet CNN Architecture: 5 convolutional & pooling layers
3 fully connected layers.

The book project will incorporate a smaller version of AlexNet
3 convolutional & 2 pooling.

CNN: INPUT => CONV_1 => POOL_1 => CONV_2 => POOL_2 => CONV_3 => POOL_3
=> DO => FC => DO => FC (softmax).



* use ReLU in activation functions.
* use softmax in final layer with 10 nodes to get probability for 10 image classes.

---

3 hyperparameters
1) Loss function: Performance measurement.
2) Optimizer: (ie) one of the variants of Gradient Descent.
3) Metrics to monitor during training & testing (ie) use metrics=['accuracy']

model training

In Keras we call "model.fit()" (ie) fit model to training data.

When the model is run.. output will show "loss" and "acc" which are the error & accuracy. val_loss & val_acc are the error & accuracy for the validation data.

* we want val_loss to decrease over the epochs & val_acc to increase

* if val_loss oscillates then the learning rate may be too high.

* if val_loss is not improving then this may be an indication of underfitting (ie) model too simple .. add more hidden layers.

* loss is decreasing & val_loss stopped improving may be sign of overfitting the training data which failed to decrease the error for the validation data. Consider using dropout layers to help prevent overfitting

---

a test accuracy of .67 is 67%

The test project is complete... located in C:\GIT\DeepLearning\model_cv.py

Structuring DL projects & hyperparameter tuning

Performance Metrics

model accuracy: accuracy = $\frac{\text{correct predictions}}{\text{total number of examples}}$

Is accuracy a good measure? example is disease detection where disease occurs 1 out of every million people. A system could hard code 'false' to the disease test & still boast 99.99% accuracy. BUT it would not be able to detect the disease.

We want to measure "goodness" of the model.

Confusion Matrix

table that describes the performance of a classification model.

(ie) 1,000 patients... predict sick or healthy.

	Predicted sick (Positive)	Predicted sick (negative)
Sick patients (positive)	100 True Positives (TP)	30 False Negative (FN)
Healthy patients (negative)	70 False Positives (FP)	900 True Negatives (TN)

True Positive: The model correctly predicted Yes (patient has disease)
True Negative: The model correctly predicted No (Patient does not have disease)
False Positive: The model falsely predicted yes (patient does not have disease)
False Negative: The model falsely predicted no (Patient has the disease).

In this case we care more about the number of false negatives because we want to capture all cases where a person has the disease.
This metric is called recall.

$$\text{Precision} \propto \text{Recall}$$

Recall is also know as Sensitivity. Recall: $\frac{\text{true positive}}{\text{true positive} + \text{false negative}}$.

Precision (a.k.a Specificity) is how many well patients system described as sick.

$$\text{Precision}: \frac{\text{true positive}}{\text{true positive} + \text{false positive}}$$

Using Recall or Precision depends on use case.
if FP is more consequential then precision is what we use.
if FN " then recall is what we use.

In some cases you may care about both.. this is an F-Score.

F-Score

Convert precision (p) and recall(r) into a single metric called F-Score.

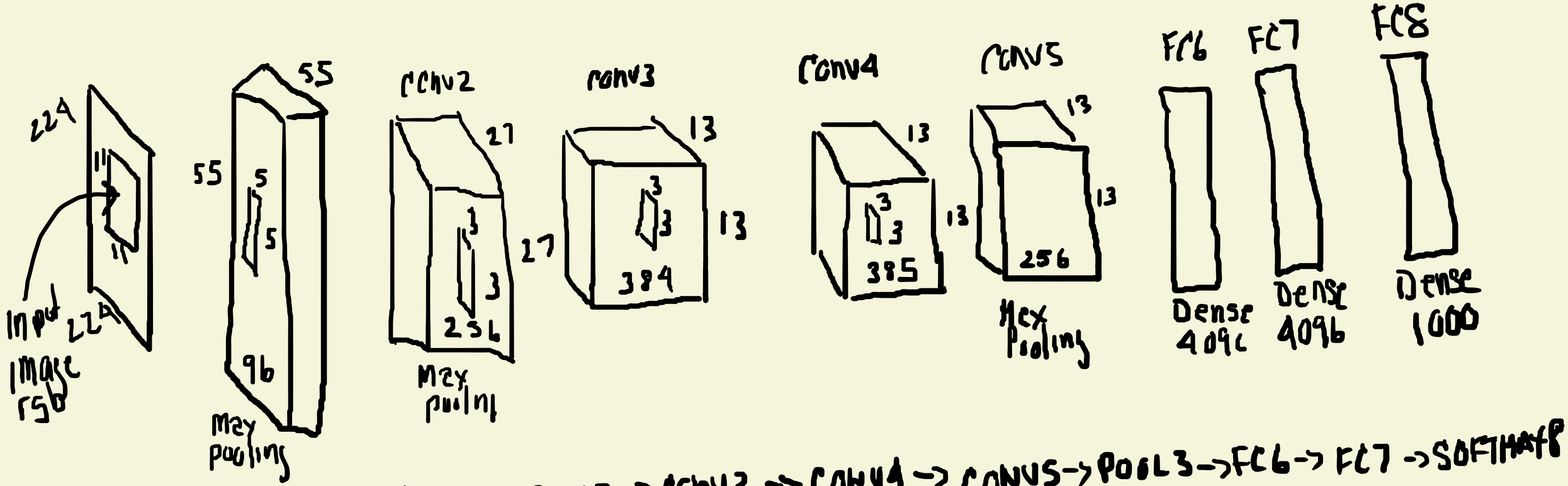
This is called harmonic mean of p & r.

$$\text{F-Score} = \frac{2pr}{p+r}$$

	Precision	Recall	F-Score
Classifier A	95%	90%	92.4%
Classifier B	98%	85%	91%

Designing A Baseline Model

(1) AlexNet
5 convolutional, 3 fully connected



Input -> conv1 -> pool1 -> conv2 -> pool2 -> conv3 -> conv4 -> conv5 -> pool3 -> FC6 -> FC7 -> SOFTMAX8

Layer Depths: CONV1 = 96
CONV2 = 256
CONV3 = 384
CONV4 = 385
CONV5 = 256

FILTER SIZES: 11x11, 5x5, 3x3, 3x3, 3x3
ReLU as activation in hidden layers CONV1->FC7
MaxPooling Layers after CONV1, CONV2, CONV5
FC6 & FC7 with 4096 neurons each.
FC8 1000 neurons, softmax activation.

Data Preparation

When training ML split data into train & test datasets.
training: training/weight adjust.
test: evaluate final performance.
* Never use the test data for training.
validation: we use the validation data after each epoch.

once the model has completed training we test its final performance over the test dataset. 60/20/20 or 70/15/15 split train, test, validate.

IMAGE RESIZING

All images should have same dimensions & color depth.
we feed dimensions into first convolutional layer.

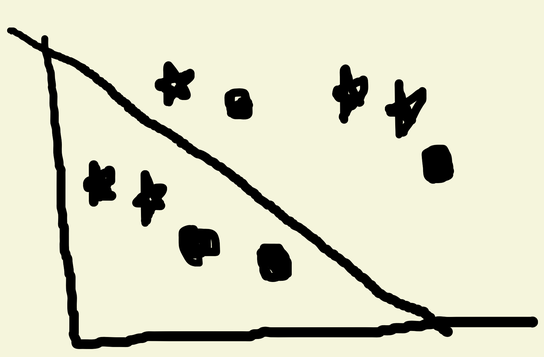
```
model.add(Conv2D(filters=16, kernel_size=2, padding='same', activation='relu',
                 input_shape=(32,32,3))
                                              ↑ color channels.
```

Normalize features: similar data distribution.
- should have small values, typically in range [0,1].
- homogenous: all pixels should have values in the same range.

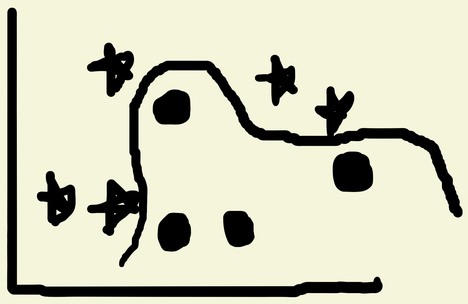
*data normalization is done by subtracting the mean from each pixel and dividing by the standard deviation.

- Neural Networks are "black boxes" when they work well it is very difficult to understand why they are working well.

OVERfitting & Underfitting Revisited



UNDERFITTING
← model too simple, tries to fit a straight line through the data.



OVERFITTING occurs when the model is too complex for the problem. Instead of learning features the model memorizes the training data. fails to generalize when presented with new data.

To diagnose underfitting & overfitting we focus on training error & validation error.

① if train_error = 1% } model does well on training data,
val_error = 10% } but is failing to generalize. So it is overfitting.

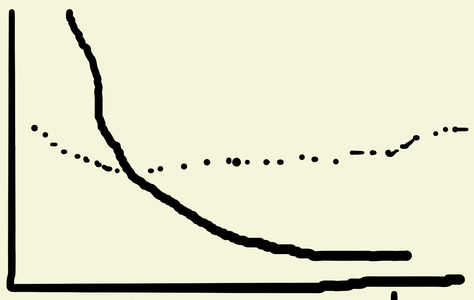
② model does poorly on training sets then it is underfitting.

Bayes Error Rate

since 0% error is unrealistic we need to define a Bayes error rate.
Basically, by comparing to human error rate.

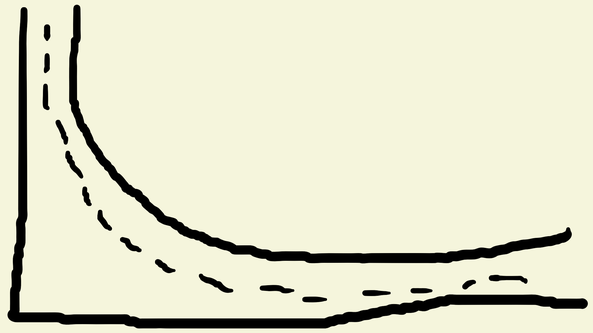
— test
---- validation

Plot the test & validation results



good test results
bad validation results
model has learned test data & failed to generalize.
overfitting

bad test results
bad validation results
model is underfitting (too simple?)



good test results
good validation results.

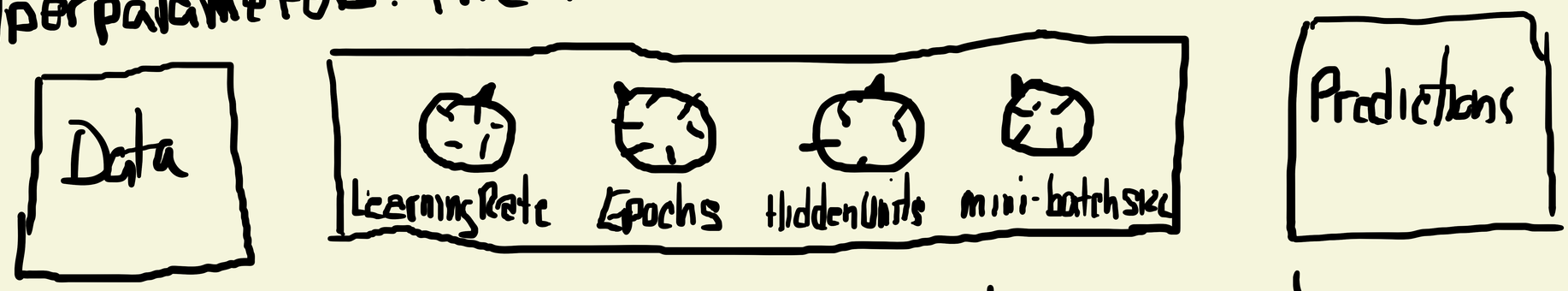
4.4.3 Exercise in building, training, and evaluating a network

This is a coding project. C:\Git\Deeplearning\model_training.ipynb

4.5.1 Collecting more data vs tuning hyperparameters

Look at train_acc & val_acc [training accuracy < validation accuracy]
if poor performance on training then underfitting so try tuning hyperparameters.
if performance on training is ok but worse on test dataset the network is overfitting training data & failing to generalize to the validation dataset.
in this case collecting more data may be effective.

Parameters: the weights & biases that the network adjusts.
hyperparameters: the variables that the ML engineer adjusts.



3 main categories of hyperparameters

Network architecture
- Number of hidden layers (depth)
- Number of neurons in each layer
- Activation Type

Learning + Optimization

- Learning rate + decay schedule
- mini-batch size
- optimization algorithms.
- Number of epochs + early stopping criteria

Regularization Techniques to avoid over fitting

- L2 Regularization
- Dropout Layers
- Data Augmentation

Network Architecture

Hyperparameters inclued : number of hidden layers (network depth).
Number of neurons in each layer. (network width)
Activation functions.

Depth/width of the Neural Network

set it to be large enough to learn the features.
Smaller network may underfit. Larger network may overfit.
* Generally, it is a good idea to add hidden neurons until the validation no longer improves. Having more depth is ok as long as we add dropouts.

Activation Type

ReLU / Softmax

Learning + Optimization

Learning Rate + Decay Schedule

The learning rate is the most important hyperparameter to tune.

Too high vs too low learning rate

Too low requires more epochs to converse, often too many.
Too high + we could wind up diverging from the optimal value.

lr: learning rate.
When plotting loss value against number of epochs... time to converge
- much smaller lr: Loss decreases but needs more time to converge
- larger lr: loss is better but still not optimal
- much larger lr could produce divergence (getting further + further away from optimal value).

Good lr: The loss decreases consistently until it reaches the minimum possible value.

Systematic Approach to Finding Optimal Learning Rate

Start @ defaults for whatever platform you are using..
then try lr: .1 lr: .01 lr: .001 .... .0001, :00001, .000001
The way to debug improvements here is to look at val_loss.

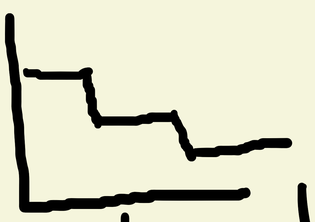
we want val_loss to keep decreasing. so keep training until it stops decreasing.

- if training is complete and val_loss is still decreasing then likely it has not converged yet. 1) Can increase number of epochs 2) Can increase lr.
- if val_loss starts to increase or oscillate the lr is too high so try turning down.

Learning Rate Decay & Adaptive Learning

This is where lr can be adjusted DURING training. often performs better than just statically setting and often reduce amount of time to get optimal results.

We can use a linear decay to reduce lr by some fixed amount every n number of epochs (ie)



We can also use an exponential decay to reduce lr. (ie) multiply by .01 for every 8 epochs.

We can also use adaptive learning which will adjust lr based upon when training steps. This means that in addition to decreasing lr it can also increase lr for when improvements are too slow.

Mini-Batch Size

"batch_size".

This has to do with gradient descent (GD). Recall.. batch, stochastic, mini-batch.

BGD: feed entire dataset to network. This can take very long to process entire dataset at each step. Requires a lot of memory cycles

SGD: feed network a single instance of dataset at a time. SGD can be noisy but often produce better results than BGD. converge but can adjust lr to compensate.

mini-batch is a compromise between BGD & SGD. we divide training samples into mini-batches from which we compute the gradient from. Best of both worlds. Faster, better than BGD & SGD

Mini-batch size

for small datasets (< 2,000) use BGD instead of mini-batch.

more on mini-batch size: start @ size 64 or 128. full scale here.. 32, 64, 128, 256, 512, 1024. The higher the mini-batch size, the faster the training.

Optimization Algorithms

Gradient Descent Optimizers Adam & Momentum

Gradient Descent with Momentum

progress toward minimum in horizontal direction. ⟺

Gradient Descent with Momentum makes learning slower along the vertical-direction oscillations and faster along the horizontals. It does this by incorporating a velocity term when updating the weights.

$W_{new} = W_{old} - \alpha \, dE/dW_i$ (original update rule)

$W_{new} = W_{old}$ - learning rate * gradient + velocity term (new rule)

* The velocity term is the weighted average of the past gradients.

4.7.2 ADAM

Adaptive Moment Estimation.
Adam keeps an exponentially decaying average of past gradients.
(similer to momentum).
Momentum can be seen as a ball rolling down a slope
ADAM can be seen as a heavy ball with friction to slow down the momentum.

* ADAM outperforms other optimizers.

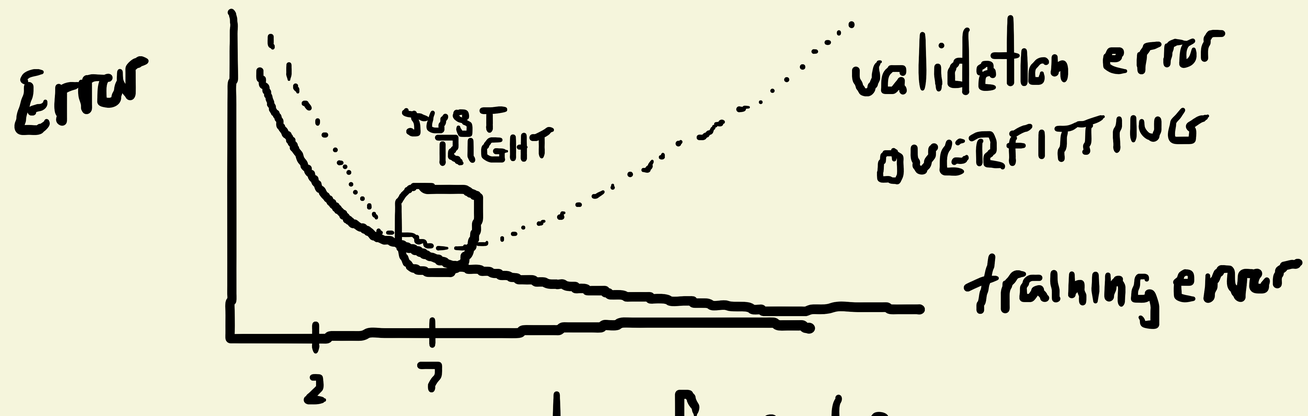
(ie) Keras.optimizers.Adam(lr=.001, beta_1=.9, beta_2=.999, epsilon=None, decay=0.0)

- The learning rate needs to be tuned.
  $\beta_1 = .9$, $\beta_2 = .999$, $\varepsilon = 10^{-8}$

4.7.3

number of Epochs & early stopping criteria

* Keep your eye on the number of training & validation errors.
  Intuitively, we want to keep training as long as the error value is decreasing.
an improving train_err along with not improving val_err means the network is starting to overfit and failing to generalize to the validation data.



We need an early way to stop the training just before it starts to overfit.
hence early stopping.

EarlyStopping(monitor='val_loss', min_delta=0, patience=20)

monitor: metric to monitor.
min_delta: the minimum change that qualifies as an improvement.

patience : number of epochs to wait before stopping training.
Tip: We can set a high number of epochs and let the stopping algorithm terminate the run.

4.8 Regularization Techniques to avoid Overfitting

We use this if we detect that the network is overfitting.
Three common regularization techniques.
L2, Dropout, DataAugmentation.

L2: Penalizes the overfn by adding a regularization term to it. which reduces the weight values to terms close to zero which simplifies the model.

Back propagation function that updates weights.

$$W_{new} = W_{old} - \alpha\left(\frac{\partial Error}{\partial W_{old}}\right)$$

W_{new} ↑ new weight
W_{old} ↑ old weight
α ↑ Learning Rate
$\frac{\partial Error}{\partial W_{old}}$ ↑ derivative of error with respect to weight

* since we add regularization term to the error function, the new error becomes larger than the old error. Derivative is also larger, leading to smaller W_{new}.
This forces the weights to decay toward zero.

→ Reducing weights leads to a simpler neural network.
It by adding the regularization term + multiplying by learning rate would make the

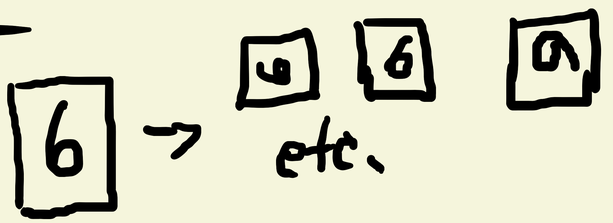
term equal to W_{old}, then W_{new} would become zero and nullify the effect of the neuron. In practice, the W_{new} does not become zero.. just close to it. The point is that L2 regularization works by reducing the effect of the neuron.

4.8.2 Dropout layers

At every training session every neuron has a probability p of being temporarily dropped out during this training iteration. This technique has a good effect. start with a value of .3

4.8.3 Data Augmentation

modify the input image by rotation, scaling etc.



In my case I might try reproducing some images with various moving averages applied to the price data, 5, 10, 20, 50, 100, 252 etc.

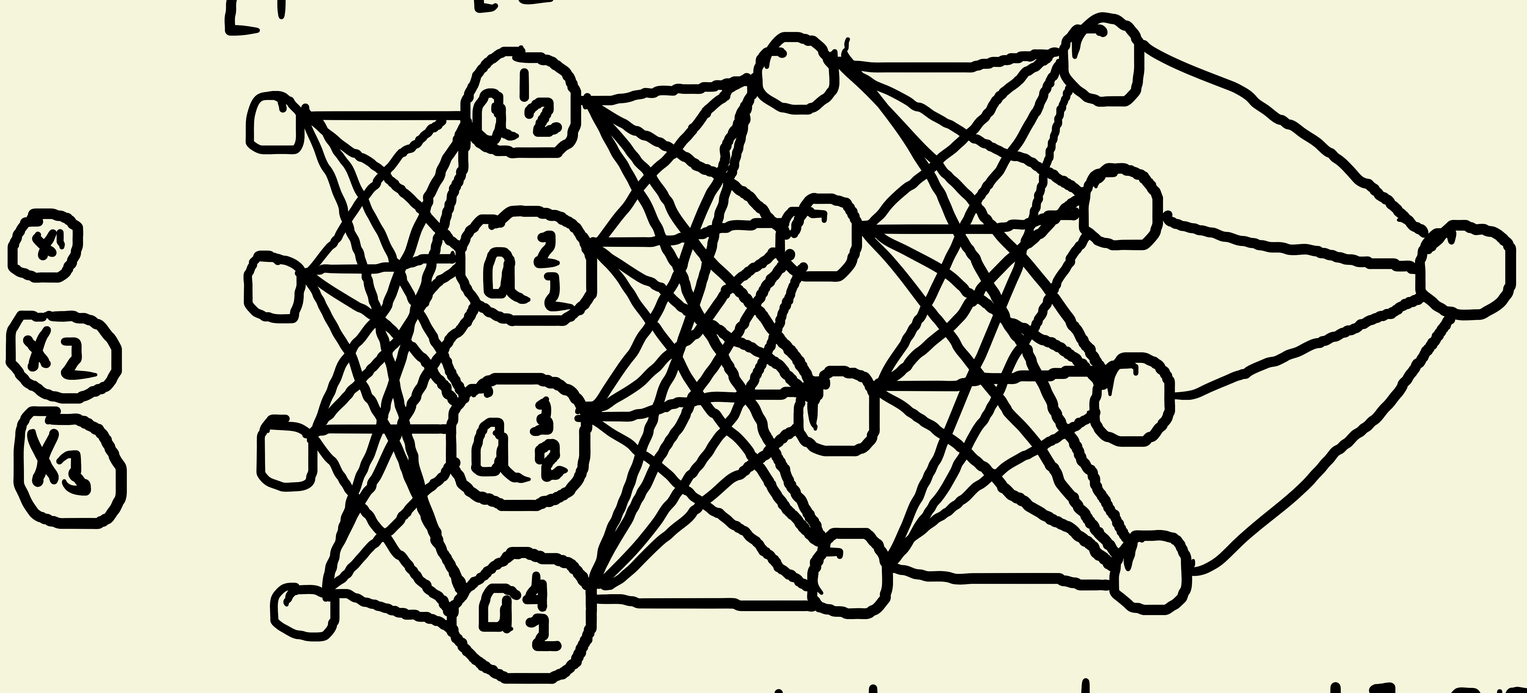
```
from Keras.preprocessing.image import ImageDataGenerator
dataGen = ImageDataGenerator(horizontal_flip=True, vertical_flip=True)
dataGen.fit(training_set)
```

4.9 Batch Normalization

Normalizing the extracted features in hidden units.

4.9.1 The Covariate Shift Problem

Definition: If a model is learning to map dataset x to label y, then if the distribution of x changes it is known as covariate Shift. If that happens you might need to retrain the learning algorithm.



↑ The activation values in L2 are inputs to L3.

from the perspective of L3: L3 is trying to map the inputs $a_{12}, a_{22}, a_{32}, a_{42}$ to $\hat{y}$ to get as close to y as possible. While L3 is doing this the rest of the network is adapting the values of parameters from previous layers. As the parameters (W, b) are changing in L1, the activation values in L2 are changing as well. So from L3's viewpoint the values from L2 are changing all the time. The MLP is suffering from Covariate Shift. Batch Normalization reduces the degree of change in the distribution of the hidden unit values, causing these values to become more stable so that the later layers in the neural network have a firmer foundation.

Batch normalization adds an operation in the neural network just before the activation function.

1) Zero-center inputs
2) Normalize the zero-centered inputs.
3) Scale & shift the results.

This lets the model learn the optimal scale and mean of the inputs for each layer.

The math

input mean

standard deviation.

$$\mu_B \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \sigma_B^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$$

① Zero-center inputs. calculate input mean & standard deviation.

m = number of instances in the mini batch

μ_B = mean

σ_B = standard deviation over the current mini batch

② Normalize Input:

$$\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$$

$\hat{x}$ = zero-centered and normalized input

ε = tiny number added to avoid dividing by zero.

③ Scale & Shift:
multiply normalized output by γ to scale it & Add β to shift it.

$$y_i = \gamma \hat{x}_i + \beta$$

y_i = output of BN operation, scaled & shifted.

BN introduces 2 new learnable parameters: γ & β
so now the optimization will update γ & β just like it does for weights & biases.

4.9.4 BatchNormalization

Batch normalization can be handled in code usually in a single line.
see example: c:\git\DeepLearning\model_bn.py for batch normalization

Summary

Batch normalization makes training easier by ensuring that the hidden units have a standardized distribution (mean & variance) controlled by two explicit parameters, γ & β, which the learning algorithm sets during training.

4.10 Project

See: c:\git\DeepLearning model_ic.py

Part 2
Image Classification Detection

Patterns

Pattern1

Convolutional Nets always have feature extraction followed by classification.

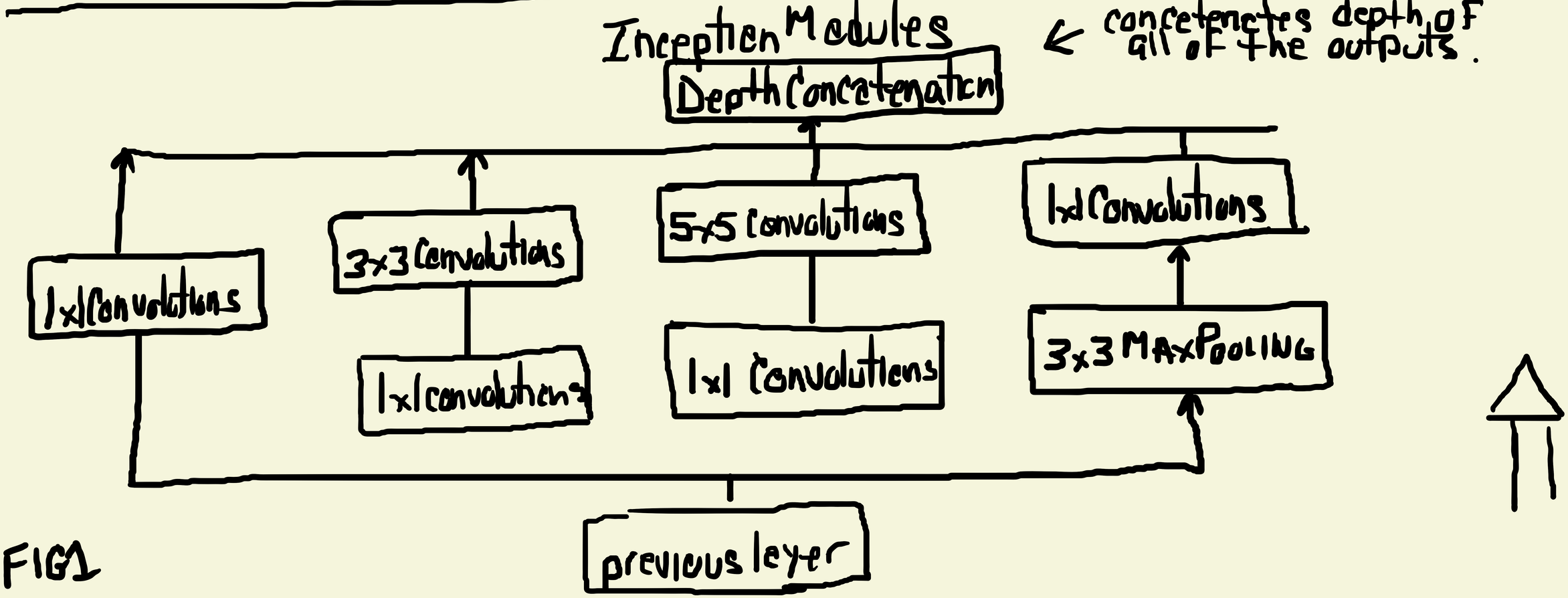
FEATURE EXTRACTION CLASSIFICATION

Pattern 2 Image depth increases & dimensions decrease.
Pattern 3 Fully connected layers. The number of fully connected layers either remains constant at each layer or decreases.

LeNet-5

INPUT IMAGE -> C1 -> TANH -> S2 -> C3 -> TANH -> S4 -> C5 -> TANH -> FC6 -> SOFTMAX7

C = CONVOLUTIONAL LAYER.
S = SUBSAMPLING or POOLING.
FC = FULLY CONNECTED.



← concatenates depth of all of the outputs.

FIG1

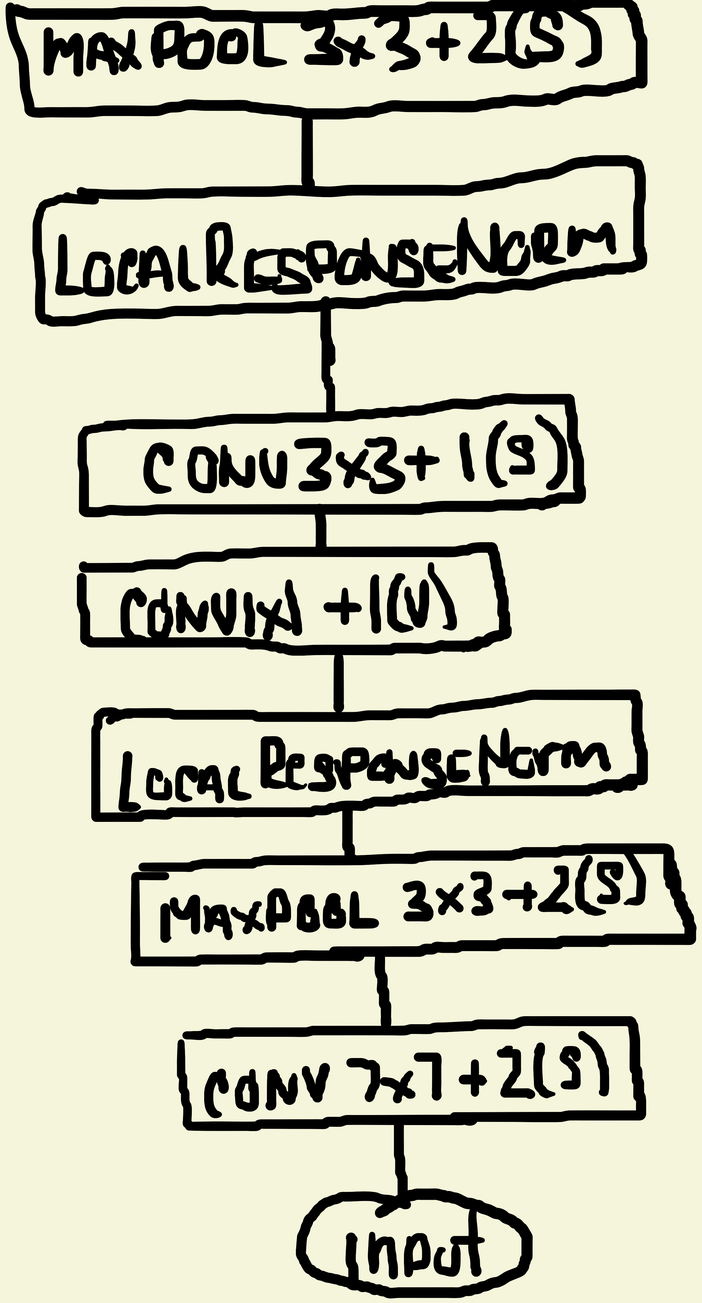
INCEPTION MODULES REDUCE DIMENSIONALITY.
BUILD DEEPER NETWORKS WHILE IMPROVING UTILIZATION OF
COMPUTING RESOURCES.

WE CAN STACK AS MANY INCEPTION MODULES AS WE WANT & BUILD
A VERY DEEP CONVOLUTIONAL NETWORK.

In Python we build the inception module as a function.

See: C:\Git\Deeplearning\Models\inception_module.py.
This is the stacked depth reduction module depicted in figure 1 above.

Here are the other parts of that model (inception)



ResNet

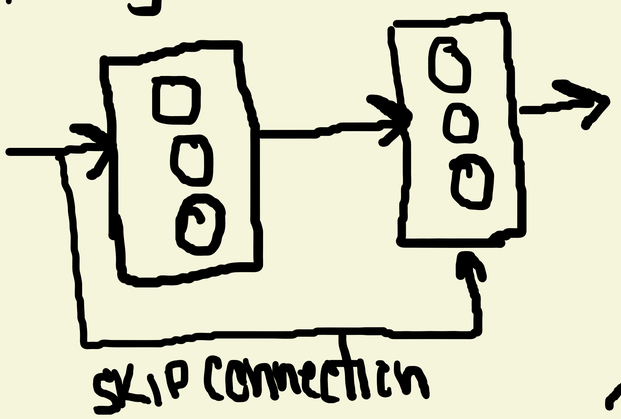
Residual Neural Network
developed by microsoft research team.

* can build very deep network with reduced complexity.
* very deep networks can extract complex functions which allow the network to learn features at many different levels of abstraction.

Need to solve for vanishing gradients

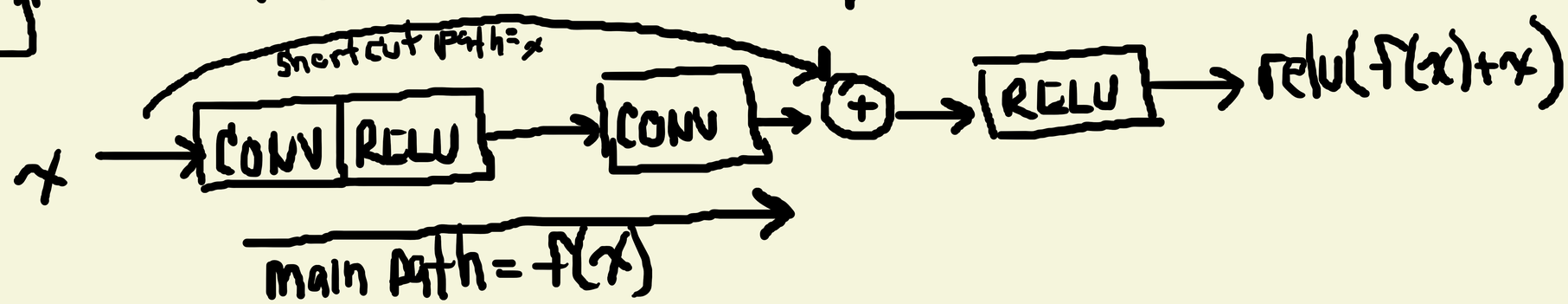
vanishing gradient can occur during backpropagation of the weights through to the earlier layers of the network. The gradient (derivative) can become very small and prevent those earlier layers from learning.

skip connectors are used which allows the gradient to be directly backpropagated through the network.



*the skip connection is added to the main path.
Note: the addition occurs prior to application of the activation function.

(ie) relu(f(x)+x)



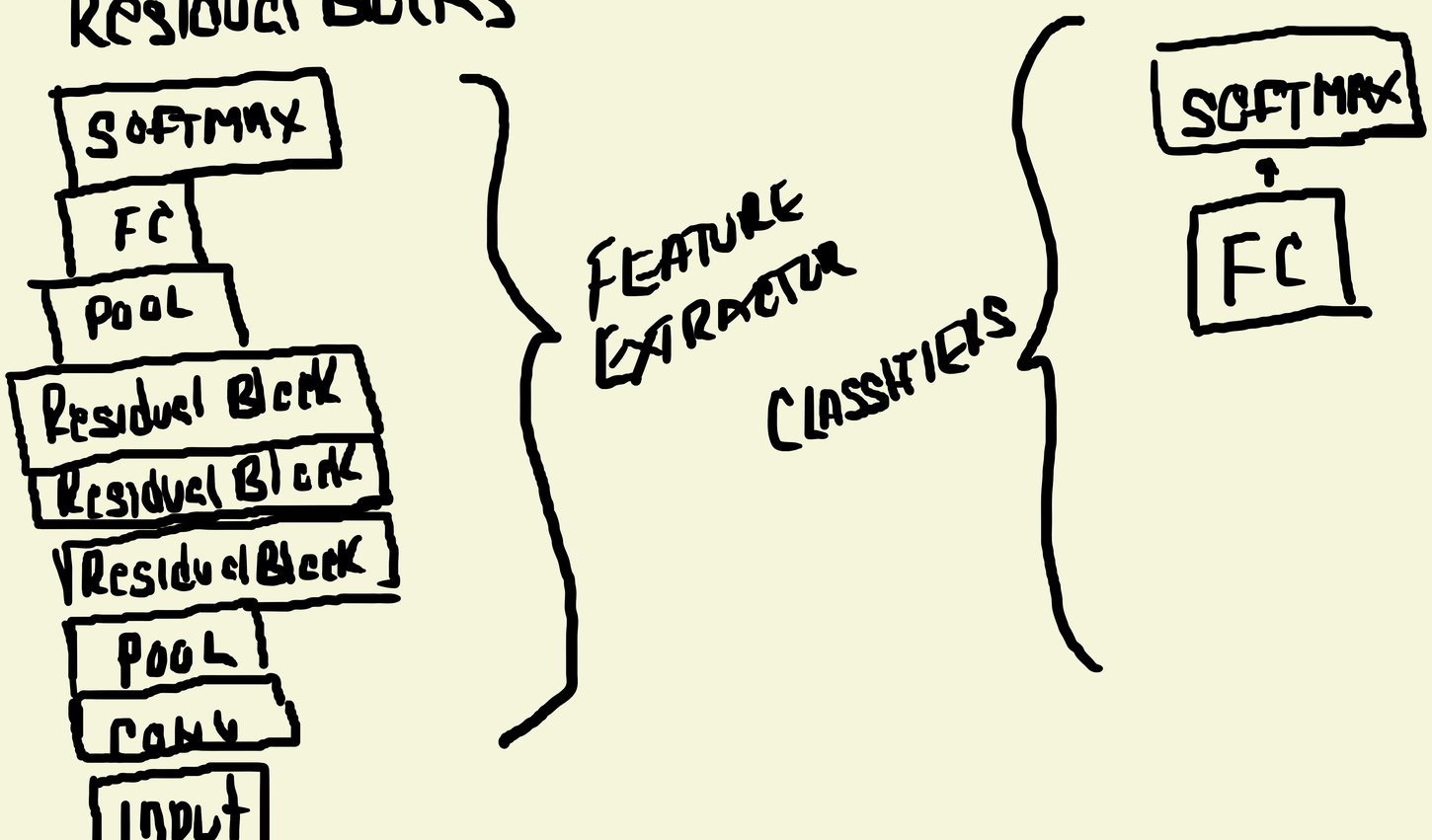
(ie)

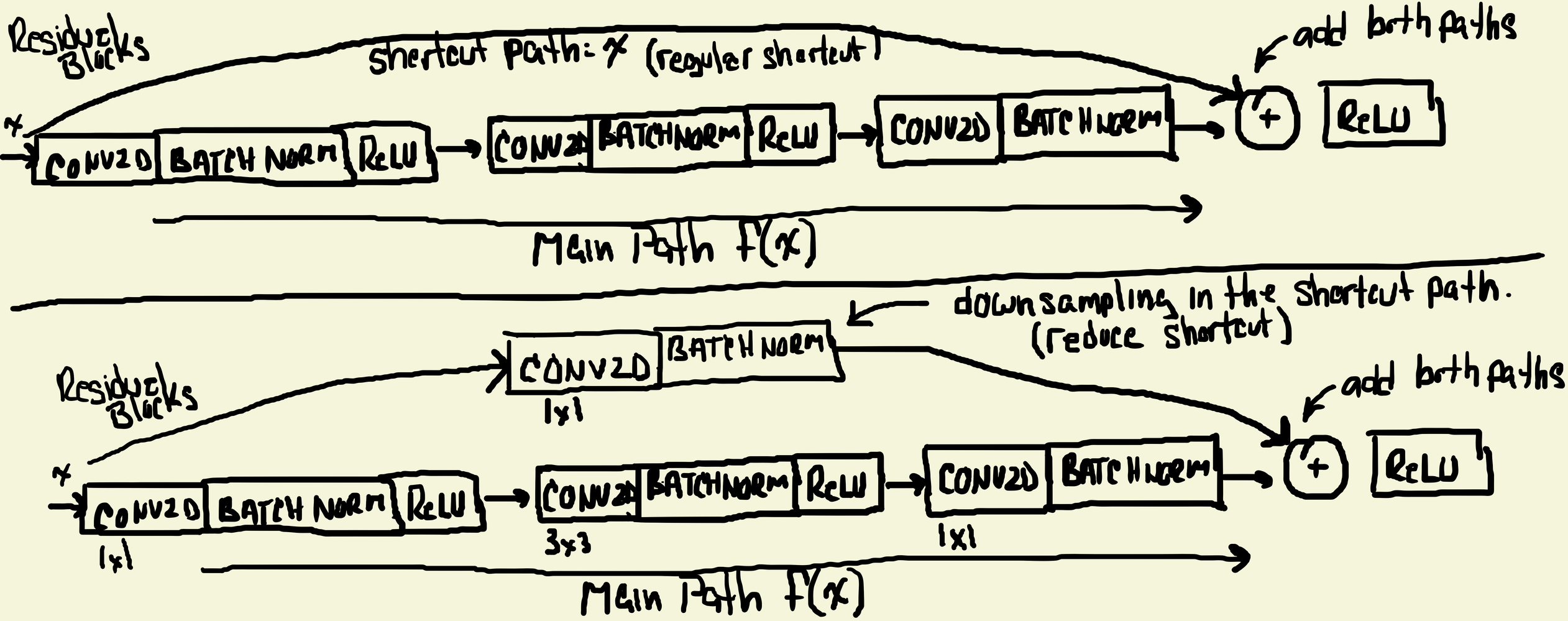
```
X_shortcut = X
X = Conv2D(filters=F1, kernel_size=(3,3), strides=(1,1))(X)
X = Activation('relu')(X)
X = Conv2D(filters=F1, kernel_size=(3,3), strides=(1,1))(X)
X = Add()([X, X_shortcut])
X = Activation('relu')(X)
```

} skip connector combination called Residual Block

RESNET is composed of a series of Residual Block building blocks that are stacked on top of each other.

Residual Blocks





THESE CNN'S CAN BE VERY TRICKY TO PROGRAM & TRAIN. FIND ON GITHUB AN IMPLEMENTATION & START FROM THERE

GOING TO COPY & TRY TO RUN RESNET50 FROM THE BOOK
THIS INCLUDES bottleneck_residual_block & ResNet50 function.