

Machine Learning Techniques:

* Supervised Learning

Every Input has a corresponding correct output.

Includes: Classification & Regression

* Unsupervised Learning

Learning from unlabeled data to discover patterns

Includes: Clustering & Dimensionality Reduction & Association rule learning

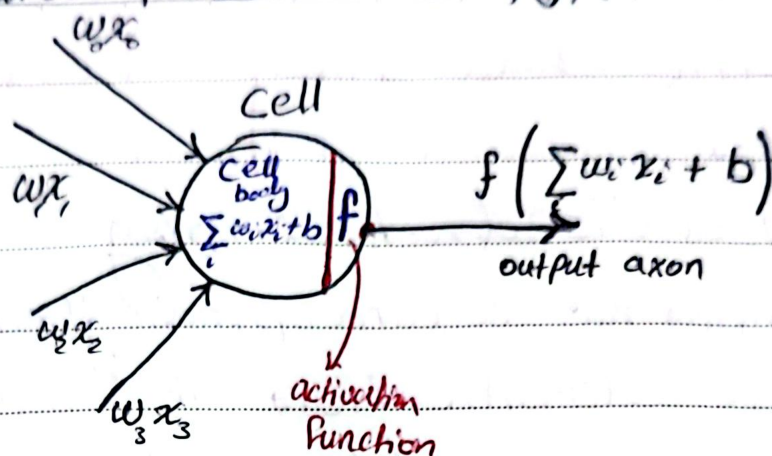
* Reinforcement Learning

An Agent learns through trial & Error

to maximize rewards by observing the environment, taking actions, receiving feedback

Learning through Neural Networks can either be supervised or unsupervised depending on how the data is provided and the task. They learn labeled data for tasks like classification, or with unlabeled data to find patterns, grouping or reduce complexity using clustering or autoencoders.

The forward pass of a NN



What is a loss function?

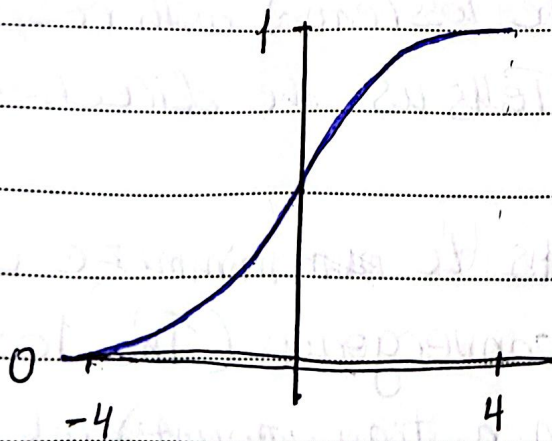
Measures the performance of the model by comparing predicted and actual values. Used to guide the training model. The goal is to minimize the loss (error) of the entire data set.

How do we choose the right loss function?

- Based on task (classification? Regression?)
- Based on type of output (categorical? continuous?)

What is Softmax?

- * For multiclass classification problems, the model outputs a value for each class. Softmax squashes these values into a probability distribution that represents the probability of each class.
- * The outputs of softmax are between 0 and 1 and sum up to 1.
- * The exponential makes every value positive and amplifies the difference.



Logits Scores	softmax	Probabilities
$y = \begin{bmatrix} 2.0 \\ 1.0 \\ 0.1 \end{bmatrix}$	$s(y_i) = \frac{e^{y_i}}{\sum e^{y_j}}$	$p = \begin{bmatrix} 0.7 \\ 0.2 \\ 0.1 \end{bmatrix}$

How Do we learn from loss?

* We want to find the best weights and Parameters that achieve the best lowest loss possible across the dataset.

* This is the role of optimizers

* Gradient Descent is the Fundamental Optimization algorithm.

* How does it work?

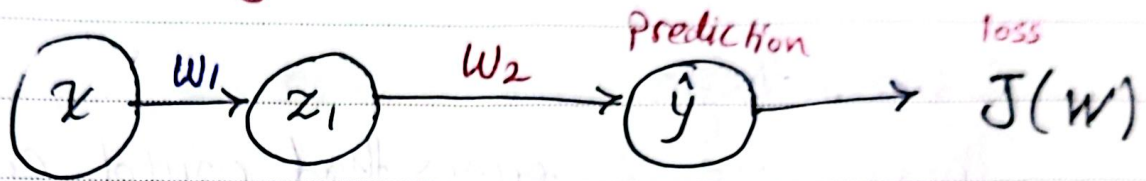
① Initialize weights randomly

② Compute the loss gradient (which is the derivative of $J_{\text{total}} = \text{loss}(\text{error})$ with respect to each weight): Tells us the direction of the slope.

③ Update weights to ~~min~~ minimize errors.

④ Repeat until convergence (The loss (error) changes in a tiny amount ~~3~~ which is not significant).

Computing Gradients: Backpropagation



* Backpropagation is the algorithm to calculate the gradients

* How much does each weight affect the final loss?

also: $\frac{\partial J(w_1 + w_2 + w_3 + \dots + w_n)}{\partial w_k}$ ← $\frac{\partial J(W)}{\partial w_2}$ ← what is the loss gradients with respect to w_2

J : The loss function

W : Just means we are going to calculate the loss in terms of weight $\{w_1, w_2, w_3, w_4, \dots\}$

Diagnosing the Training data

* How do you find out how well your model is learning?

* You split your data into training, validation and test set:

- ① Training set: to update weights of the model
- ② Validation set: used by the developer to tune hyperparameters and monitor performance
- ③ test set: used after the model is trained to test real-world accuracy.

Learning issues:

~~Learning issues:~~

There are 2 errors that could cause underfitting or overfitting:

① Variance (The sensitive error):
caused by a model that is too sensitive to the noise in the training data.

② Bias (The simple error): Caused by a model that is too simple to capture the patterns of the data.

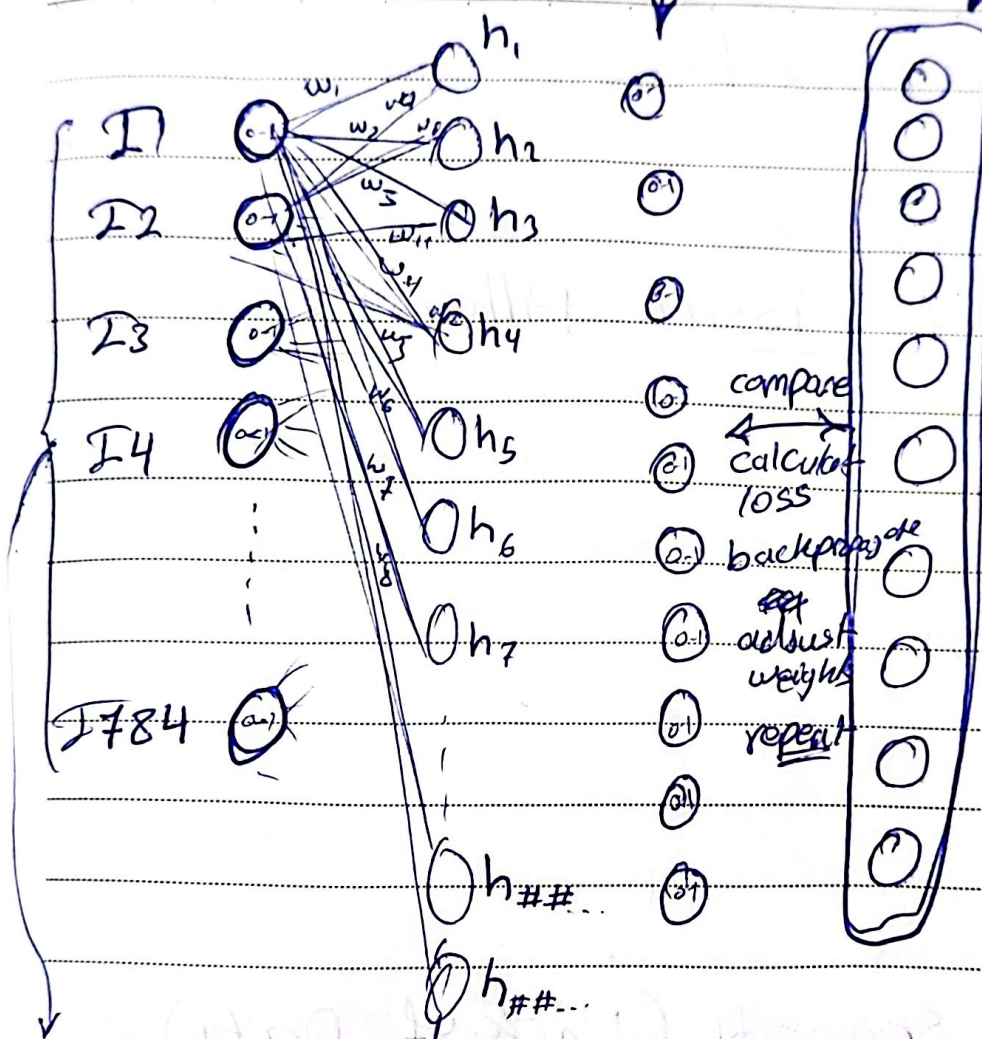
→ High Bias: The model oversimplifies ~~the~~ and misses the patterns and underfits the data.

→ Low Bias: The model captures the patterns too well, and is close to the true values, might become too close.

Final Prediction

RELIEVE THE TOUCH OF PEN HOLDING

Correct Prediction



inputs are

for one flattened
image 28x28
pixels

$$h_1 = I_1 \cdot w_1 + I_2 \cdot w_2 + I_3 \cdot w_3 + \dots + I_{784} \cdot w_{784} + b_1$$

$$h_2 = I_1 \cdot w_9 + I_2 \cdot w_{10} + I_3 \cdot w_{11} + I_4 \cdot w_{12} + \dots + I_{784} \cdot w_{79} + b_1$$

$$h_3 = \dots$$

What are expecting from neurons in the hidden layers?

We are expecting from these neurons (in hidden layer), to pick up certain patterns, where each neuron is responsible for activating or deactivating depending on the state and outputs of the previous layer. That way, neurons become "specialized", and you can detect certain shapes & edges or patterns.

Common Issues with ML Datasets

Three main issue pillars:

- ① Quality
- ② Quantity
- ③ Distribution

Issues

① Data scarcity (Lack of Data)

Def: Insufficient data points to accurately represent the underlying data distribution.

Impact: overfitting or underfitting

Solutions:

- Create new samples
- leverage pre-trained models
- Synthetic data generation
- Active classes, Few shot learning

② Data Impalalance

For example there is 95% percent data about class A and 5% about class B.

Solutions?

→ Oversampling or undersampling

→

③ Data Drift & concept dripte

A change in data distribution over time that invalidates the assumptions the model was trained on.

Solutions:

online / Reinforcement Learning

④ Missing Values

Data points are absent from features

Solutions:

- Delete rows/columns
- Fill missing data using mean, median or mode
- Advanced imputation: KNN imputation

⑤ Noisy Data and Outliers

Def: • Random error or Variance

- Data points significantly distant from other observations

Solutions:

- Data preprocessing
- Trimming. Reduces the effect of outliers
- Robust models: using models that are less sensitive to outliers

⑥ Data Leakage

When you for example leak test data to the training data, that way final results of accuracy on test data is completely wrong.

Golden Rule

Any step that learns from the data must be performed only on the training data. The resulting parameters should then be applied on test sets.

Diamond Rule

The test set must remain completely untouched and unseen until the model is fully trained. It is used exactly once for the final, unbiased performance measurement.

Lecture - Deep computer Vision

Dr. Tamer khattab

It is the ability for computers to process an image and classify and identify the objects in the image.

~~How it works (learn)?~~

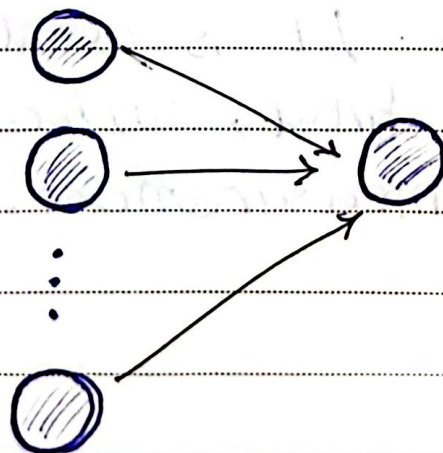
How can we learn a hierarchy of features directly from data instead of manually engineering it?

Maybe we can use Normal Dense Neural networks which are fully connected.
But would it work?

The Dense Neural Network:

Input:

- 2D image
- Vector of Pixel values
- Pixels are flattened



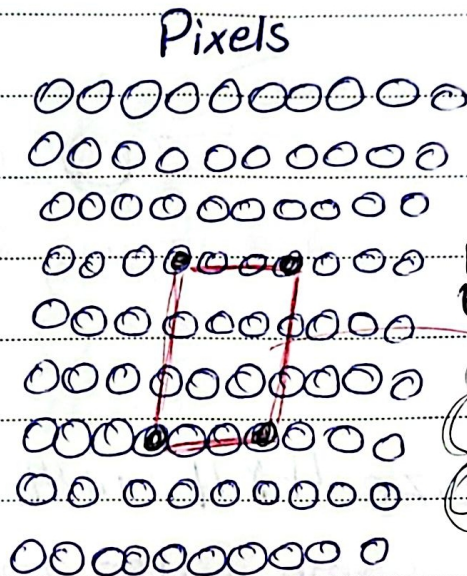
Fully connected:

- Connect neuron in hidden layer to all neurons in the input layer
- No spatial information!
- Many, many parameters

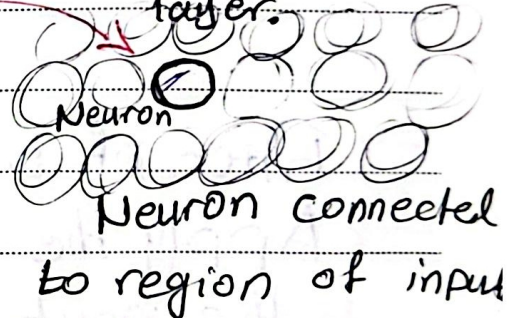
There is another idea, which is using Spatial Structure

Input:

- 2D image
- Array of pixel values



Idea: connect patches of input to neurons in hidden layer.



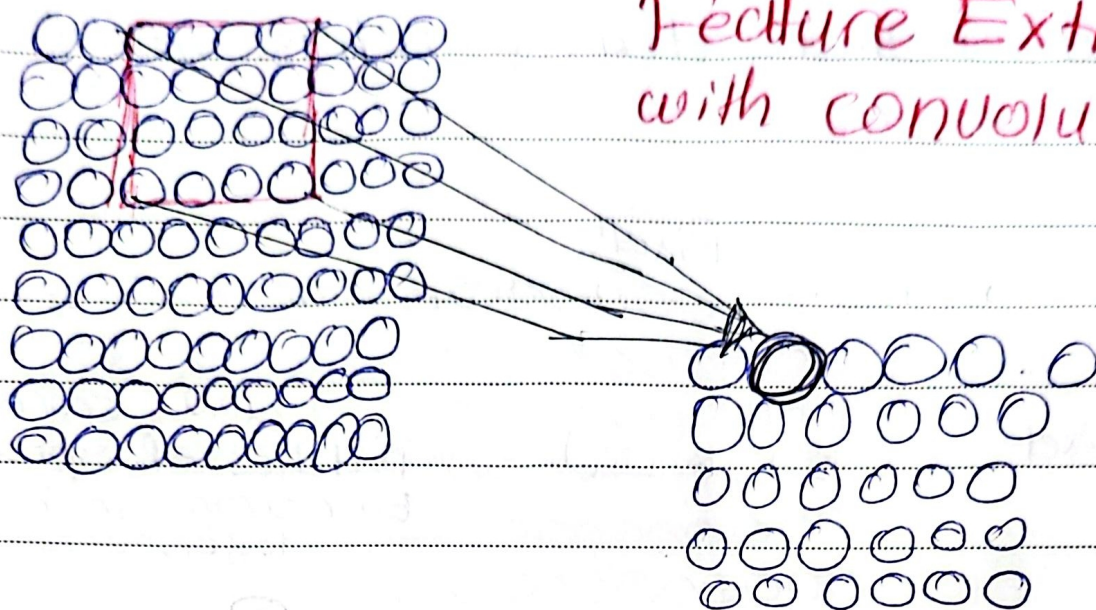
Only "sees" these values

Patching Filters: Neurons map specific patches of the picture. Neuron location within the 2D Dense network will represent the location of the patch within the image.

Sliding Patching Filters:

- Connect patch of input layer to a single neuron in subsequent layer.
- Use sliding window to define connections.
- Weight the patch to detect particular features.

Feature Extraction with convolution



- Filter of size 4×4 with 16 different weights
 - Apply the same filter to 4×4 patches in the input.
 - Shift by 2 pixels for next patch
- This "patchy" operation is simply a **convolution**

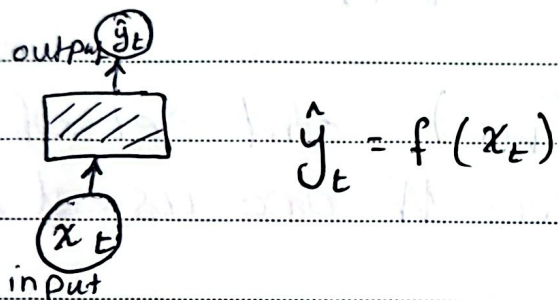
Feature Extraction

- Apply a set of weights (a filter) to extract local features
- Use multiple filters to extract different features
- Spatially share parameters of each filter.
- Once you end sliding the filter across all patches, you end up with a feature map
- Feature maps amplify certain characteristics or features, like edges or shadows.

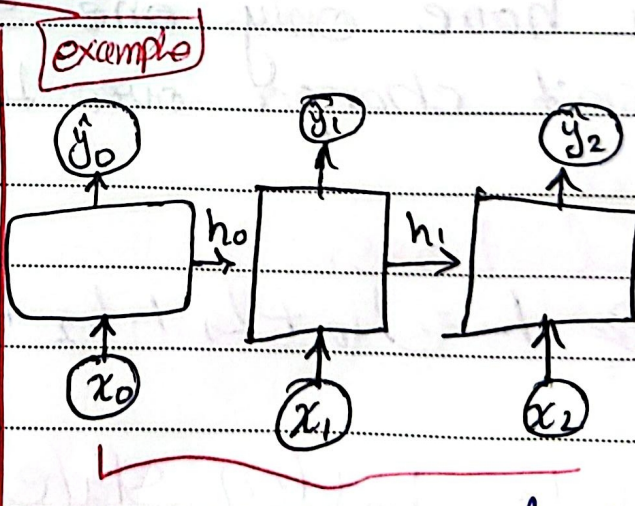
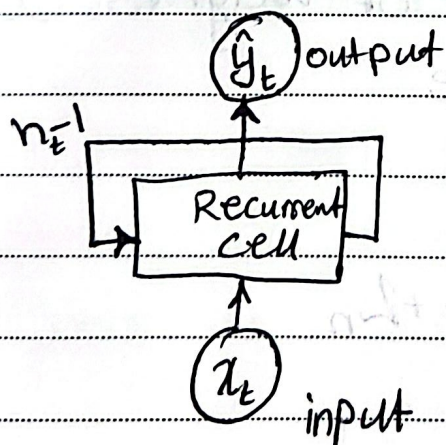
Deep Sequence Modeling

Dr. Tamer khattab

Adding memory to neural networks
Individual Time steps (snapshots
of the neural network over time)



The recurrent cell



$$\hat{y}_t = f(x_t, h_{t-1})$$

Same neural network
at different time stamps

Recurrent Neural Networks (RNNs)

Apply a recurrence relation at every time step to process a sequence

$$h_t = f_W(x_t, h_{t-1})$$

- The same function $f_W()$ and set of weights and parameters W are used at every time step
- RNNs have a state, h_t , which is updated at each time step as a sequence is processed
- You have only one set of weights. It doesn't change over time

Total Loss:

$$L = L_0 + L_1 + L_2 + \dots + L_n$$

Update hidden state

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

- RNNs are going to be important for recognizing text, that is why we need memory or states to be able to process sentences.

Words cannot be represented like that. We should convert them to numbers

[a, same, you, ...]

↳ index to get word

same = [0, 1, 0, 0, ..., 0]

Natural Language processing

Word Embeddings

A Deep Dive (once again) on NNs.

So we said previously that each neuron just represents a number which is the result of activating the sum of the matrix multiplication of previous neurons and their biases and weights. Activating the function (for sigmoid and tanh) just make the value of a neuron between 0 and 1.



$$a_z^{(n+1)} = \sigma(a_0^{(n)} w_0 + a_1^{(n)} w_1 + a_2^{(n)} w_2 + \dots + a_k^{(n)} w_k)$$

~~neuron~~ neuron value Sigmoid

n : the ^{index} ~~number~~ of the layer

z : the index of the neuron

m : the index of the weight

σ : the sigmoid activation function (could use tanh also)

For now: we are just hoping that each neuron is activated or deactivated once it picks up a pattern from previous ~~layers~~ layers. That way, we have knobs and switches that ~~make~~ make many combinations possible, so we have a way to identify many certain patterns. By the way, this is exactly how the human brain works. So the idea isn't new, it was just duplicated.

But now, how will the network actually "learn"? How are we going to adjust the many ~~weights~~ weights and biases so that neurons become "specialized" and detect various patterns? How are we supposed to make the perfect ~~combination~~ combination of weights and biases (parameters)?

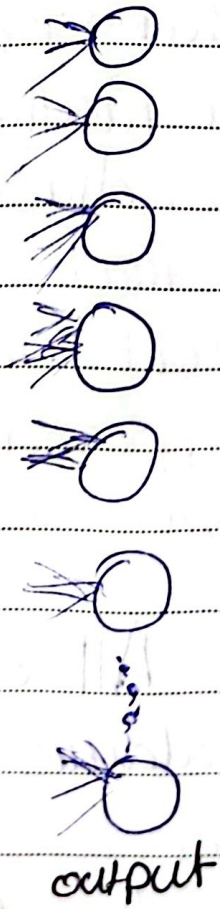
All of that opens the door for Calculus to answer the question. (A small part is for calculus). We aren't really "learning" or teaching the machine. Instead, we are literally trying to find the perfect combination

of parameters. So, we can use derivatives to know for each feedforward, how ~~steep~~ "steep" we are towards the ~~minimal error~~ minimal error, so that hopefully we come up with the correct answer with the right combination of the parameters.

Let's try to understand this better.

Let's go to the output layer:

Since our parameters started random, the results are going to be HORRIBLE. How can we know how "horrible" our answer was? That way we can know each time how far we are from the correct answer. Well, how can we know how horrible our answer was? We are going to use something called a "loss function", which is just a fancy way of saying error measurement function.



There are many loss functions, however for now we will stick with the Mean Squared Error (MSE).

$$MSE = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

$\hat{y}_i$: predicted output.

y_i : correct output.

The higher the number of MSE, the "worse" we are performing.

Example of bad output

~~0.2~~

0

0.45

0

$$MSE = \frac{1}{7} ((0 - 0.2)^2 +$$

0.36

0

$$(0 - 0.45)^2 + (0 - 0.36)^2 +$$

0.65

0

$$(0 - 0.65)^2 + (0.9 - 1)^2 + (0 - 0.65)^2 + (0 - 1)^2)$$

0.9

1

$$MSE = 0.261$$

0.65

0

↑ the number may seem low, but

0.9

0

relatively it is very bad.

predicted

correct

HO RIBBLE

Just knowing how bad we are doing isn't really useful though. How can we fix the weights and biases so that we have a better result (less loss)?

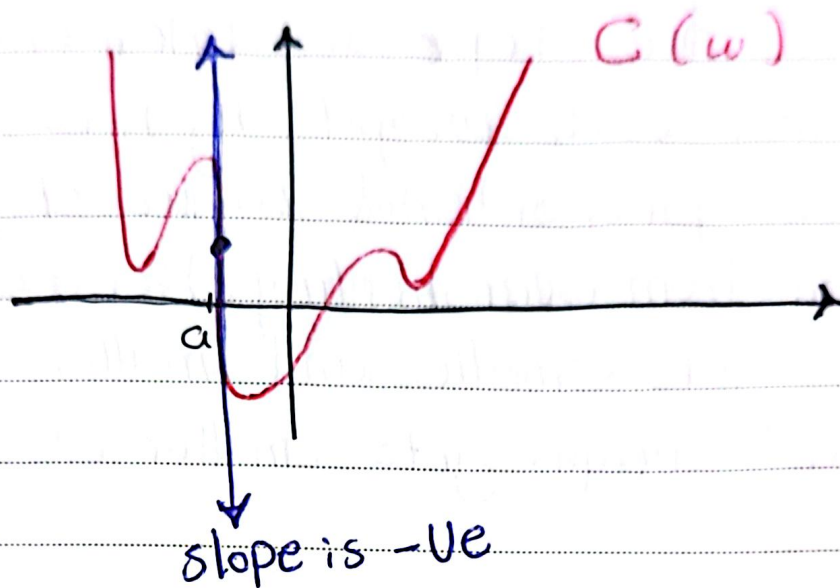
~~XXXXXXXXXX~~

~~WE~~

Since the cost function had all weights as input and produced a single output of how bad the weights combination was, we can know which input 'causes' the increase in error, and hence we can tweak it to get a better result.

$$C(w_0, w_1, w_2, \dots, w_n)$$

Let's give an example visually ~~for~~
~~only one~~ when considering only one weight for simplicity:



when $w = a$, the slope at a is -ve, which means we can move a bit ~~to~~ to the right to settle at the lowest point for $C(w)$

notice: for actual Cost functions with many parameters, we don't have an ~~explicit~~ easy form like x^2 , but instead there are many variables involved ($w_0, w_1, w_2, \dots, w_n$). So calculation the global minimum, is going to be crazy hard.

For each weight.

but of local minimum it is doable. And another note is, finding the global minimum is a ~~lot~~ lot harder since our choice for each weight is completely random.

local minimum = Doable
Global minimum = crazy hard

Another note: The steps you take (how much you change each weight by), ~~if~~ if you make them proportional to the slope, this helps you from overshooting. Because as the slope gets smaller and smaller, your steps for each weight gets smaller also.

~~Each weight~~ ~~of the many weights~~

Each weight of the many weights will be edited. And each weight gives us a "direction" to where to edit the weight (Increase it or decrease it). This "direction" is called the Gradient.

And the process of going with the Gradient to decrease the total error is called Gradient descent.

Alright! ~~Now~~ Now that we have wrapped up most of the process of the NN, I want to confess.

we previously said that we want the network to "detect" certain patterns and ~~images~~ edges in the inputs. For example, if we pass an image of a 9, then it should detect the straight line and the loop. But does the network actually do this? ~~NO~~ **NOT AT ALL!**

~~Our~~ Our poor NN instead just deals with the numbers we passed to it strictly and doesn't "detect" something. For example, if you gave it a random image with noise, it will confidently spit out a specific answer. That is why NNs are not the best. In addition, if we told it to draw some image of a specific number, the NN wouldn't be able to (by knowing it did very bad in the noise example). This is a starting point to CNNs and LSTMs, which are far better than normal NNs. Actually, NNs are very old and considered old technology, But ^{if it's} ~~they~~ the ~~found~~ foundation to understand CNNs and LSTMs.

✖ Our Network just found itself a nice local minimum that got it working well. But it doesn't detect the edges that we were hoping for.

Backpropagation Calculus

$$a^{(L)} = \sigma(w^{(L)} a^{(L-1)} + b^{(L)})$$

not an exponent
just indexing

↑

When we want to calculate the activation of a certain neuron when there is only 4 total neurons

$$\text{Cost} \rightarrow C_0(\dots) = (a^{(L)} - y)^2$$

the cost of the first training example

predicted correct

input $a^{(L-2)}$ $a^{(L-1)}$ output $a^{(L)}$

When we want to see how much the cost of the first output changes with respect to some weight:

$$\frac{\partial C_0}{\partial w^{(L)}}$$

← this is what we want to calculate. However there is not direct relation between both.

$$\frac{\partial C_0}{\partial w^{(L)}} = \frac{\frac{\partial (w^{(L)} a^{(L-1)} + b^{(L)})}{\partial w^{(L)}} \cdot \frac{\partial a^{(L)}}{\partial (w^{(L)} a^{(L-1)} + b^{(L)})}}{\frac{\partial C_0}{\partial a^{(L)}}}$$

chain rule

$$\frac{\partial C_0}{\partial a^{(L)}} = \frac{(a^{(L)} - y)^2}{\partial a^{(L)}} = 2(a^{(L)} - y)'(1)$$

$$= \boxed{2(a^{(L)} - y)}$$

Notice: $\frac{\partial C_0}{\partial a^{(L)}}$ is proportional ~~to~~ to $a^{(L)}$ (the output we got) and y (the wanted correct output). Or more precisely, $\frac{\partial C_0}{\partial a^{(L)}}$ is proportional to the difference between $a^{(L)}$ and y .

~~$$\frac{\partial C_0}{\partial a^{(L)}} = \frac{\partial C_0}{\partial (w^{(L)} a^{(L-1)} + b^{(L)})} \cdot \frac{\partial (w^{(L)} a^{(L-1)} + b^{(L)})}{\partial a^{(L-1)}} = \frac{\partial [\sigma(w^{(L)} a^{(L-1)} + b^{(L)})]}{\partial (w^{(L)} a^{(L-1)} + b^{(L)})} \cdot \frac{\partial (w^{(L)} a^{(L-1)} + b^{(L)})}{\partial a^{(L-1)}}$$~~

kind of big $z^{(L)} = w^{(L)} a^{(L-1)} + b^{(L)}$

$$\frac{\partial a^{(L)}}{\partial z^{(L)}} \cdot \frac{\partial [\sigma(z^{(L)})]}{\partial z^{(L)}} = \boxed{\sigma'(z^{(L)})}$$

$$\frac{\partial z^{(L)}}{\partial w^{(L)}} = \boxed{a^{(L-1)}}$$

every one change in $w_k^{(L)}$ results in this amount of change for C_0 .
That way we know that w_k affects the overall cost C .

$$\frac{\partial C_0}{\partial w^{(L)}} = \frac{\partial z^{(L)}}{\partial w^{(L)}} \cdot \frac{\partial a^{(L)}}{\partial z^{(L)}} \cdot \frac{\partial C_0}{\partial a^{(L)}}$$

$$= a^{(L-1)} \cdot \sigma'(z^{(L)}) \cdot 2(a^{(L)} - y)$$

$$\frac{\partial C_0}{\partial w^{(L)}} = \boxed{2 a^{(L-1)} \sigma'(z^{(L)}) (a^{(L)} - y)}$$

$$\frac{\partial C}{\partial w^{(L)}} = \frac{1}{n} \sum_{k=0}^{n-1} \frac{\partial C_k}{\partial w^{(L)}}$$

cost of
 C_k : one training
 example

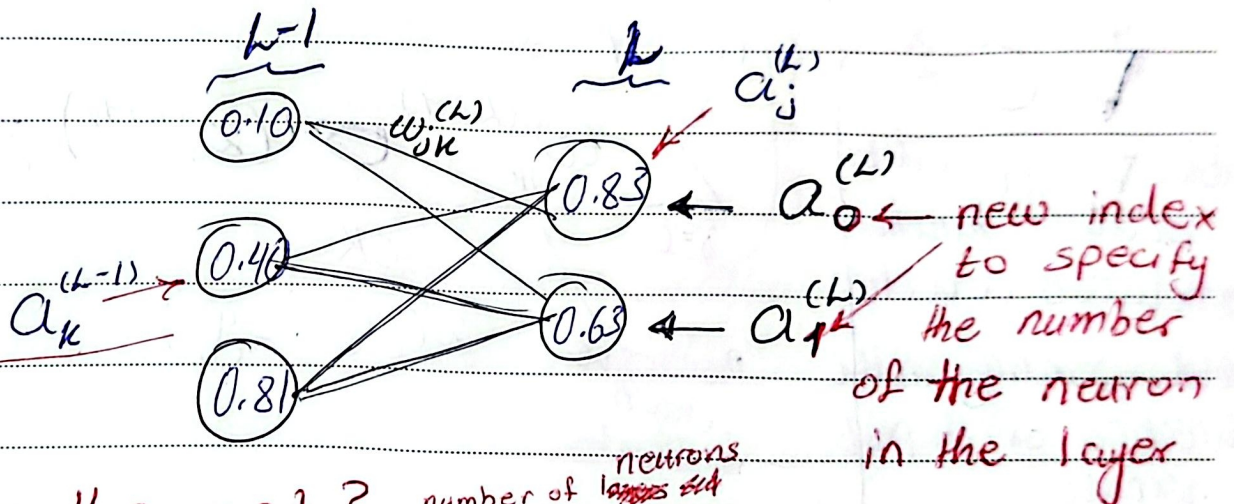
Average of all
 training examples

$$\nabla C = \begin{bmatrix} \frac{\partial C}{\partial w^{(L)}} \\ \frac{\partial C}{\partial b^{(L)}} \\ \vdots \\ \frac{\partial C}{\partial w^{(1)}} \\ \frac{\partial C}{\partial b^{(1)}} \end{bmatrix}$$

Same process applies for the bias.
 Just change $\frac{\partial z^{(L)}}{\partial w^{(L)}}$ to $\frac{\partial z^{(L)}}{\partial b^{(L)}}$

What about networks with layers containing additional neurons?

It doesn't really get that bad. Not much actually change. Instead you just have to add a couple of more indexes.



For the cost?
 number of ~~layers~~ ^{neurons} in the output layer - 1

$$C_0 = \sum_{j=0}^{n_L-1} (a_j^{(L)} - y_j)^2$$

index of the layer neuron $j=0$

the activation output of the j -th neuron in the Layer 2

$$z_j^{(L)} = w_{j0}^{(L)} a_0^{(L-1)} + w_{j1}^{(L)} a_1^{(L-1)} + w_{j2}^{(L)} a_2^{(L-1)} + b_j^{(L)}$$

$$\frac{\partial C_0}{\partial w_{jk}^{(L)}} = \frac{\partial z_j^{(L)}}{\partial w_{jk}^{(L)}} \cdot \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \cdot \frac{\partial C_0}{\partial a_j^{(L)}}$$

$$\frac{\partial C_0}{\partial a_k^{(L-1)}} = \sum_{j=0}^{n_L-1} \frac{\partial z_j^{(L)}}{\partial a_k^{(L-1)}} \cdot \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \cdot \frac{\partial C_0}{\partial a_j^{(L)}}$$

there is a sum here because that activation (neuron) influences the loss through multiple paths (weights)

∇C ← nabla
 nabla ∇ applied to a scalar function produces a vector field pointing in the direction of steepest increase.

$$\frac{\partial C}{\partial w_{jk}^{(L)}} = a_k^{(L-1)} \sigma'(z_j^{(L)}) \cdot \frac{\partial C}{\partial a_j^{(L)}}$$

$$\sum_{i=0}^{n_{(L+1)}-1} w_{jk}^{(L+1)} \sigma'(z_i^{(L+1)}) \cdot \frac{\partial C}{\partial a_j^{(L+1)}}$$

→ the number of neurons in layer $L+1$
 $n_{(L+1)} - 1$
 $i=0$
 Recursive chain
 OR
 $2(a_j^{(L)} - y_j)$

∇C : is a vector that contains the partial derivatives of the cost function C with respect to all the network's parameters (weights and bias). The components of that vector tells you how much the cost function would change if that particular weight or bias was adjusted.

$$W_{\text{new}} = W_{\text{old}} - \underset{\substack{\uparrow \\ \text{learning} \\ \text{rate}}}{\eta} \nabla C$$

By repeatedly moving in the direction of $-\nabla C$ we are minimizing the loss function.

Computer Vision & CNNs

An image is just a matrix of numbers ranging $[0, 255]$. e.g. for an RGB image with Resolution 1080×1080 , the image will be a matrix of size $1080 \times 1080 \times 3$

Computer vision tasks:

Regression: output is continuous. (eg. image enhancement)

Classification: output is either a label or probability of belonging to class

Before: Scientists and researchers had to manually program specific feature extraction

But that didn't work due to the image have the chance of being of different size, viewpoint, illumination, background feature, and Occlusion

How can we learn features?

Can we learn a hierarchy of features directly from data instead of manually engineering it?

How about Dense regular NNs?

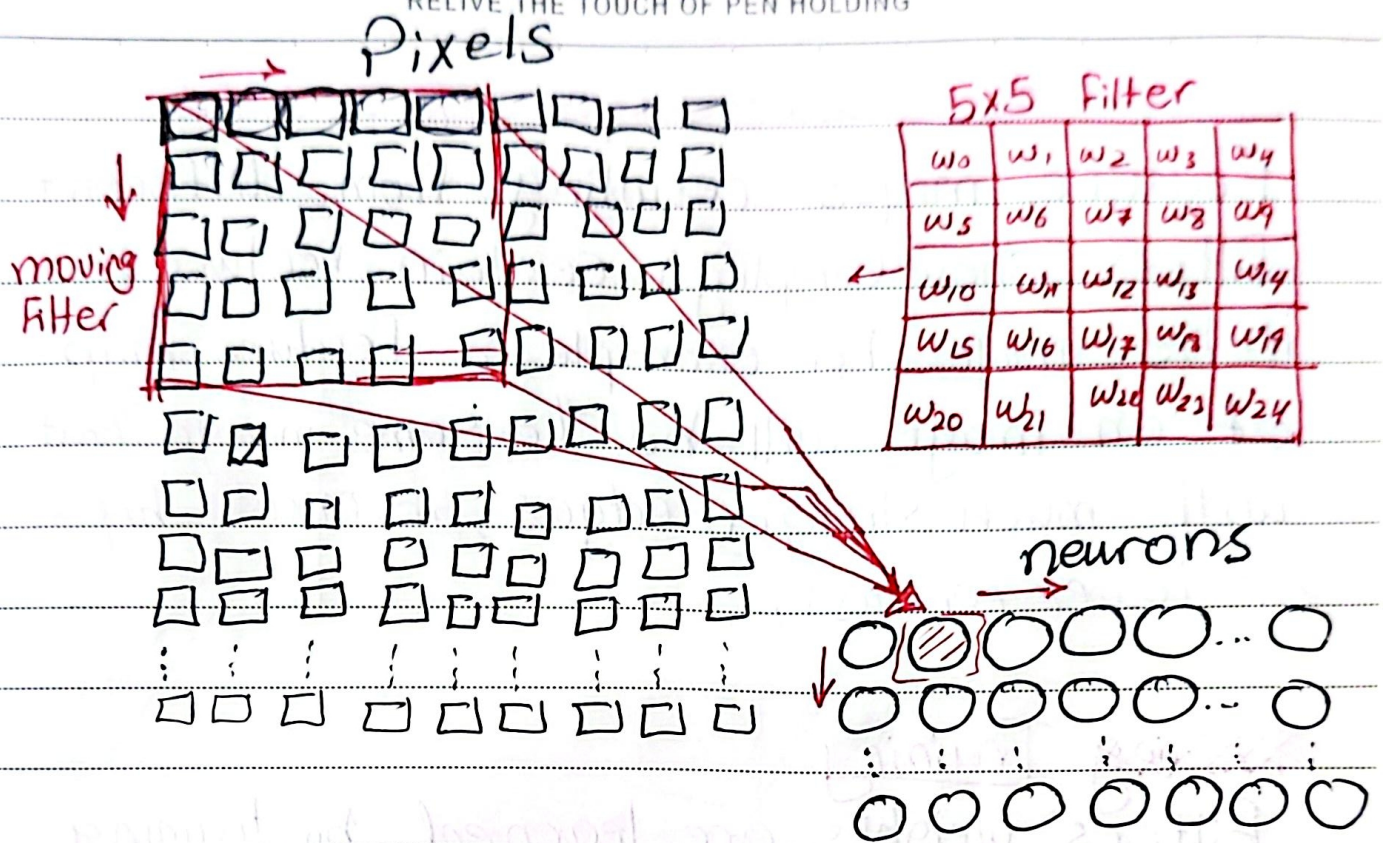
Can we use regular NNs to extract features? NO, Because images in NNs are split into single pixels and distributed across the entire network, destroying the spatial structure and memory of where of the image is where.

We need to preserve 2D spatial structure
IDEA: connect patches of input (pixels) to neurons in hidden layer. Neuron connected to region of input only sees these values (pixels)

Neurons map specific patches of the picture. Neuron location within the 2D dense network will represent the location of the patch within the image.

Sliding patching Filters

- * Use sliding window to define connections
- * Weight the patch to detect particular features



- * The same 5x5 filter goes through all the pixels in the image.
- * The location of the filter in the image at a given moment reflects the location of the corresponding neuron related to it (holding its information (numbers)).
- * Weights in a filter stay constant.
- * There are multiple filters, each responsible for extracting different features.
- * Weights in filter are updated with backpropagation.
- * how much pixels the filter is shifted by after each ~~iteration~~ iteration could be specified.
- * Element wise multiplication is done with each filter convolution ~~iteration~~ iteration, then the results are added and stored in the neuron.
- * After all convolutions, you will end up with a Feature map.

What do feature maps do?

Feature maps resulting from different filters show (amplify) certain features in the image. For example, a feature map of an image will be the same image but with much stronger edges, or much higher or lower contrast.

~~Training~~ Training

Filters' weights are learned by training with data

Convolution

Apply filter to generate feature map

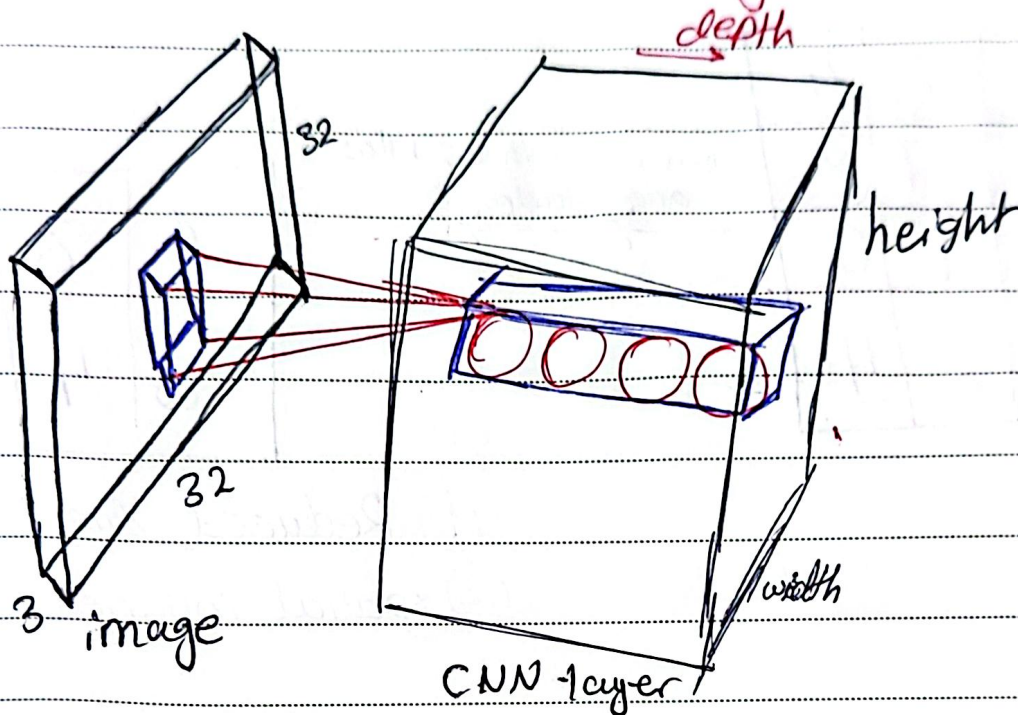
Non-linearity

Typically ReLU

Pooling

Downsample each feature map

CNN spatial arrangement of output



Layer Dimensions: $h \times w \times d$

* h and w are spatial dimensions

* d is depth = number of filters

Stride

Filter step size (by how many pixels to walk after each ^{full} element wise multiplication of a filter with a patch)

Receptive Field

Locations in input image that a node is path connected to.

The Pooling

1	1	2	4
5	6	7	8
3	2	1	0
1	2	3	4

max pool with 2×2 filters
and stride = 2

6	8
3	4

- ① Reduced dimensionality
- ② spatial invariance

With CNN Learning

- * Low level Features (like edges, dark spots) are extracted in the first conv layer.
- * Mid level Features (like Ears, Eyes, nose) are extracted in the second conv layer.
- * Higher level (Facial structure) is extracted in the 3rd conv layer.

CNN for classification: Feature learning

- ① Learn Features in input image through convolution
- ② Introduce non-linearity through activation function (as real world data is not linear)
- ③ Reduce dimensionality and preserve spatial invariance with pooling

CNN for classification: class probabilities

- Conv and Pool layers output high level features as input to Dense layers
- Fully connected layers uses these features to classify
- Express output as probability of image belonging to a particular class.

$$\text{Softmax}(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$$

Training with back propagation

Learn weights for convolutional filters and fully connected layers.

Backpropagation: Cross-Entropy loss

$$J(\theta) = \sum_i y^{(i)} \log(\hat{y}^{(i)})$$

FCNs

- Fully convolutional networks are ~~nn~~ networks with only convolutional layers, with downsampling and upsampling operations.

R-CNNs: Find Regions that we think have objects. Use CNN to classify.