- $\mathbf{x}_4 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$, : $(0.8, 0.025)$ 1 wins. Thus, we have

$$\begin{bmatrix} 0.9 \\ -0.5 \end{bmatrix}, \begin{bmatrix} 0.9625 \\ 0.9375 \end{bmatrix}$$

- $\mathbf{x}_5 = \begin{bmatrix} 1.1 \\ -1 \end{bmatrix}$, : $(1.49, 0.12125)$ 1 wins. Thus, we have

$$\begin{bmatrix} 1 \\ -0.75 \end{bmatrix}, \begin{bmatrix} 0.9625 \\ 0.9375 \end{bmatrix}$$

- $\mathbf{x}_6 = \begin{bmatrix} 1 \\ -1.1 \end{bmatrix}$. : $(1.825, -0.06875)$ 1 wins. Thus, we have

$$\begin{bmatrix} 1 \\ -0.925 \end{bmatrix}, \begin{bmatrix} 0.9625 \\ 0.9375 \end{bmatrix}$$

The final weights can do the clustering as intuitively desired.

4 Batch competitive learning interpretation of k-means

Note the decomposition $\|\mathbf{x} - \mathbf{c}_i\|^2 = \|\mathbf{x}\|^2 - 2\mathbf{x}^T\mathbf{c}_i + \|\mathbf{c}_i\|^2$. So, if we consider a network of m neurons, where the ith neuron has weights $2\mathbf{c}_i$ and bias $-\|\mathbf{c}_i\|^2$, then the winning neuron (the one with the largest induced local field) coincides with the neuron for which $\mathbf{c}_i$ is closest to $\mathbf{x}$. Consider showing all $\mathbf{x} \in \mathcal{S}$, where $\mathcal{S}$ is the training set, and recording the winning neurons for each example shown. Equivalently, we are calculating the Voronoi regions $\mathcal{V}_1, \ldots, \mathcal{V}_m$ given the cluster centers $\mathbf{c}_1, \ldots, \mathbf{c}_n$. At the end of the Epoch, we update the weights and the biases of each neuron according to the k-means update rule described previously, i.e. we set the weights of the $i$the neuron to be $2\frac{1}{|\mathcal{V}_i|}\sum_{\mathbf{x}\in\mathcal{V}_i}\mathbf{x}$, and the bias of the ith neuron to be $-\|\frac{1}{|\mathcal{V}_i|}\sum_{\mathbf{x}\in\mathcal{V}_i}\mathbf{x}\|^2$. This variant of a competitive batch learning rule coincides with the k-means algorithm.

ECE/CS 559 Lecture 17/18 Th 10/24 / T 10/29

Last time: CNNs (max pooling, resnets), PyTorch

① Unsupervised Learning

Before, data (x, y). Now, data is just x.

What is the task? Represent x "nicely".

- Clustering: Place x in one of a handful of "clusters"
- Dimensionality reduction: Represent $x \in \mathbb{R}^m$ by $x \in \mathbb{R}^k$ $k \ll m$.
- Density estimation: Find the distribution of x

Relationship to generative models & feature extraction.
(represent → create more) ("enough" to represent)

② k-means Clustering

Input $x \in \mathbb{R}^n$, parameters $\underbrace{c_1, c_2, \cdots, c_k}_{C}$, $c_i \in \mathbb{R}^n$.

Output $f(x; C) = \underset{c \in C}{\operatorname{argmin}} \|x - c\|_2^2$

Loss: $\ell(x, f) = \|x - f\|_2^2$

Risk: $R(C) = \frac{1}{|Data|} \sum_{x \in Data} \ell(x, f(x; C))$

How do we minimize $R(C)$?

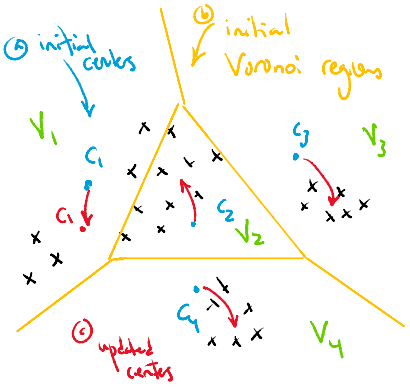
- It's possible to do gradient descent. It's non-convex!
- Or... use Lloyd's algorithm aka "the" k-means algorithm.

Lloyd's Algorithm

Input: Data, k

- Initialize $C = \{c_1, \cdots, c_k\}$ (arbitrarily / randomly) ⓐ
- Repeat:
  - Cluster according to C (Voronoi regions) ⓑ
  - $c_i \leftarrow$ Mean $(x : x \text{ closest to } c_i) = (x : x \in V_i)$ ⓒ
- Stop when convergence slows.

Output: Cluster centers C

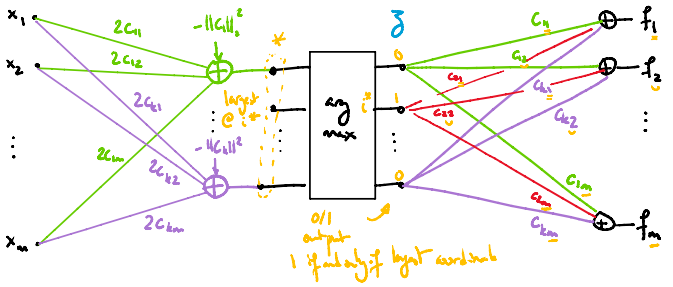


Voronoi regions partition the space according to proximity to specific centers

③ From k-Means to Hebbian updates

Since $\|x - c\|_2^2 = (x-c)^T(x-c) = \|x\|_2^2 + \|c\|_2^2 - 2x^Tc$ *

We can write the clustering as a neural network:



Note that f will be the closest center.

- Because argmax is not differentiable (constant, except at jumps), no gradient information backpropagates before it.
- We still have gradients after it, starting at: $\frac{\partial \ell}{\partial f_j} = -2(x_j - f_j)$

Then: $\frac{\partial \ell}{\partial c_{ij}} = \frac{\partial \ell}{\partial f_j} \cdot \frac{\partial f_j}{\partial c_{ij}} = -2\delta_i(x_j - f_j)$ — for those $f_j = c_{ij}$

δ_i ↑ only nonzero at $i = i^*$ (argmax index)

- Thus: $\nabla_{c_i} R = \sum_{x \in V_i} -2(x_j - f_j) = -2 \sum_{x \in V_i} x + 2|V_i| c_j$

Gradient descent becomes:

$$c_i \leftarrow c_i - \eta \nabla_{c_i} R = c_i + 2\eta \sum_{x \in V_i} x - 2\eta |V_i| c_j$$

if $\eta_i = \frac{1}{2|V_i|}$ then $= \frac{1}{|V_i|} \sum_{x \in V_i} x$ same as Lloyd's algorithm!

- Assume $\|x\| = \|w\|$ (or $\|c\|$) $= 1$ for all weights.
- Reasonable by going one dimension higher
- Distance → (minus) inner-products! cosine(angle) ↑ distance ↓
- That was full batch. With 1 point, we get instead

$\forall i$: $c_i \leftarrow c_i + \eta(x - c_i)\, \mathbb{1}\{\underbrace{\operatorname{argmax}_i (c_i^T x) = i}_{\text{self-supervision} = \delta_i}\}$ no bias needed

- δ is the neurons' output (decision). In general, it doesn't have to be restricted to 0/1.
- Definition: Competitive (winner-takes-all) learning rule:

$w_i \leftarrow w_i + \eta\, \delta_i (x - w_i)$ only for $i = \operatorname{argmax}_i \delta_i$

- If $g_i \in (0,1)$ then

$$w_i \leftarrow (1-\eta g_i) w_i + \eta g_i x$$

↳ weights that look not like x are made to look more like x (they specialize at identifying things similar to x / clusters)

④ **Oja's rule:**

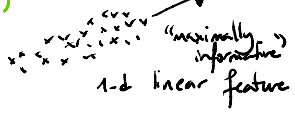
Definition: $w_i \leftarrow w_i + \eta g_i (x - w_i g_i)$ — difference; doesn't affect it if 0/1

- Claim, for small η, $w + \eta g (x - wg) \approx \frac{(w + \eta x g)}{\|w + \eta x g\|}$

vector scalar scalar vector (for each i)

- Extracts principle component:

$$\rightarrow \underset{w:\|w\|=1}{\operatorname{argmax}} \sum_{x \in \text{data}} (w^T x)^2$$



⑤ **Hebbian rules**

- Most of these rules are "Hebbian":
  "Neurons that fire together, wire together"
- If x is coming from another neuron and w is a neuron that mimics it, then it will strengthen.
- **Definition:** Original Hebbs rule: $w_i \leftarrow (w_i + \eta x g_i)$
  (compare to Perceptron with true supervision)
- It can also cluster! But it's unstable:
  Fixes: Oja's rule, $w_i \leftarrow \alpha w_i + \eta x g_i$
  ↳ dampening < 1.

Variants: $w_i \leftarrow w_i + \eta (x - \bar{x})(g_i - \bar{g}_i)$

Example: $g_i = \text{Sign}(w_i^T x)$

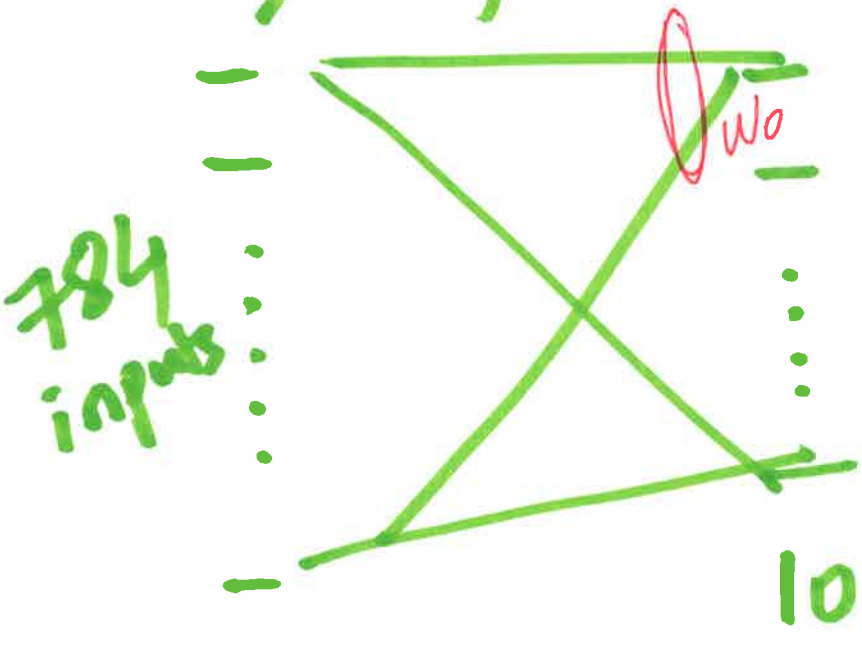
- Say $x = 1$ vector, fixed $w_i(0) = 1$ vector also initially. $\eta = 1$.
  $w_i(1) = 1 + 1 = 2$ $w_i(2) = 2 + 2 = 4$ $w_i(3) = 8 \ldots$
- If we stabilize with $\alpha < 1$:
  $w_i(1) = \alpha + 1$ $w_i(2) = \alpha(\alpha+1) + 1 = \alpha^2 + \alpha + 1$
  $w_i(3) = \alpha(\alpha^2 + \alpha + 1) + 1 \quad \cdots \rightarrow \frac{1}{1-\alpha}$
- In the notes: Examples of Hebbs rule and competitive

L14

ECE/CS 559 - Neural Networks (1)

Convolutional Neural Networks

- Remember why the single-layer FF network for classifying MNIST digits failed:



The output should be $[1, 0, \ldots, 0]$ if the input is a "0" digit, $[0\ 1\ 0 \ldots 0]$ if the input is a "1" digit, and so on.

Now imagine what should the corresponding weights / filters "look like"?

For example, let us call the weights corresponding to the first output neuron as W_0, as shown on Page 1. This neuron should output a "1" for all "0" digits and a "0" for all digits $\neq 0$. Suppose that we use a "1" for a black pixel and "-1" for a white pixel. If we want to maximize the output for an input (say)

1	1	-1	1	-1

, then the filter should exactly be matched to the input, i.e, the filter should also be:

1	1	-1	1	-1

So, if we want an output of "1" for an image that looks like a zero, we ideally need a filter that "looks like a zero". Meanwhile this filter should look unlike a "1, 2, 3, 4, 5, 6, 7, 8, 9" so that it will not be matched to these undesired images. So, try to imagine a digit that looks like a zero, but does not look like a one, two, three, four, etc. Tough, huh?

So, why does a single layer fail? We try to very quickly decide on complex features.

Solution: Use multilayer networks. (3)

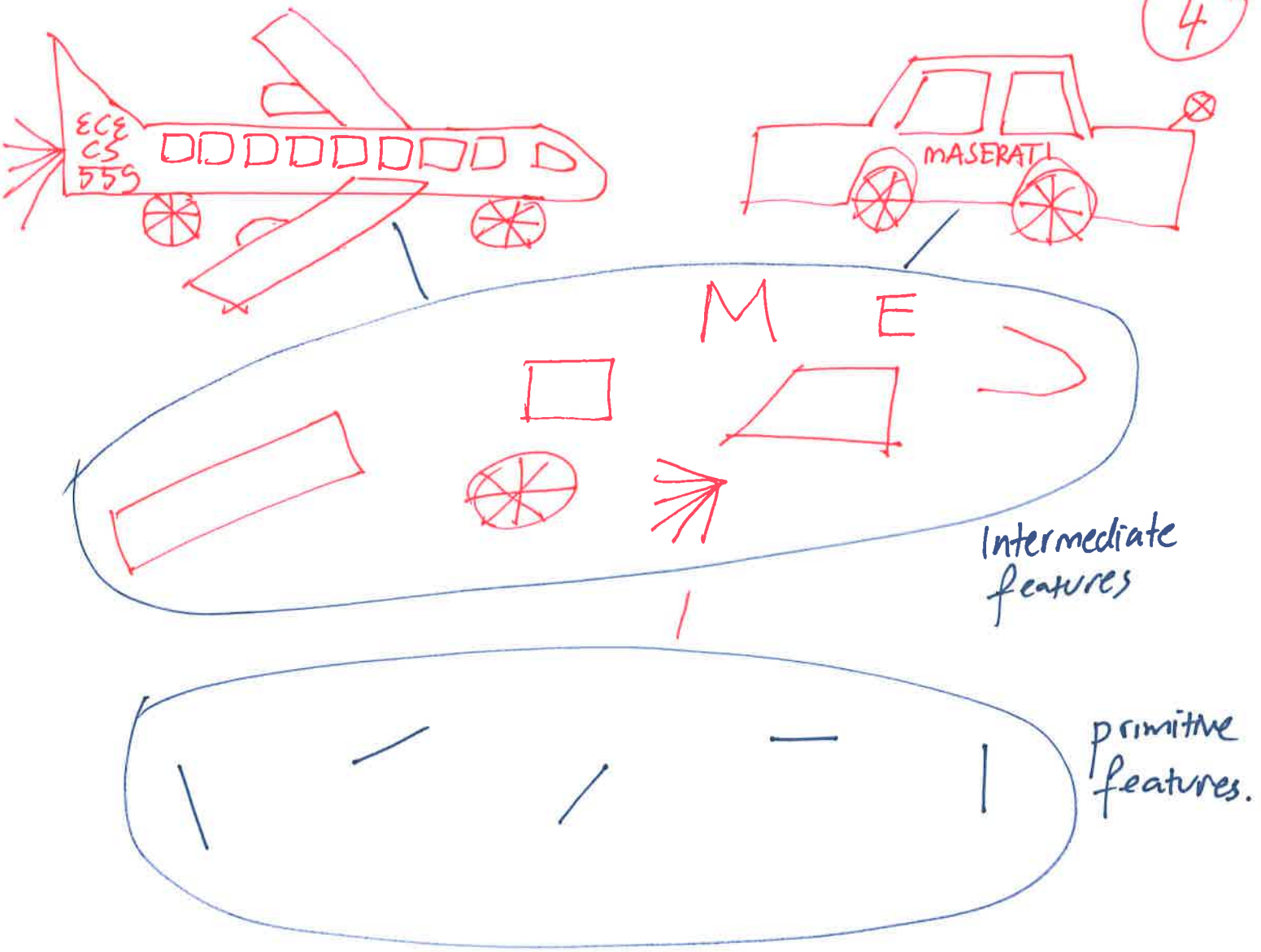
Problems: Too many parameters may result in overfitting, performance degradation, increased implementation complexity.

e.g. increased input image resolution
⇒ more free parameters..

Convolutional Neural Networks (CNNs): Became state of the art for image & video processing.

- provides invariance to translation, scaling, skewing, rotation, & other distortions relevant to image processing.
- inspired from the visual cortex of the cat and studies by Hubel & Wiesel.

- Alternatively, we first imagine extracting simple features like straight edges, corners, then move on to moderately complex features like a wheel, an arm or a leg, and then move to objects like a car, human, airplane, etc.



Primitive features form intermediate features, which in turn, form more complex features, or ultimately the objects we wish to classify.

* Extract primitive features in the first layer
Intermediate features #1 in the second layer
⋮
and so on.

Observation. **Primitive features**

1. are small in size
2. can be anywhere in the image.

Due to (1), we just need a small filter to capture one primitive feature. Due to (2), we will slide the filter all over the image to capture the primitive feature, wherever it may be! This is called a convolution operation.

How do we choose the primitive filters??

We will not!!! Backpropagation algorithm will train the filters for us. In the past, people used to work with handcrafted filters (back when computers where not as powerful and deep learning methods were not as well-developed).

CNNs introduce two new kinds of neuron layers. One is the convolutional layer as outlined in the previous page. A convolutional layer is fundamentally no different than a layer of a typical feedforward neural network, but there are constraints on the weights and the "receptive fields" of the neurons. Typically, a convolutional layer consists of multiple [redacted] filters to be slided across the input to the convolutional layer. Assume a 5x5 input image, and consider a convolutional layer with one 2x2 [redacted] filter.

1	2	0	3	1
-1	-1	2	0	1
1	-1	-1	0	1
3	1	2	-1	1
1	4	0	1	2

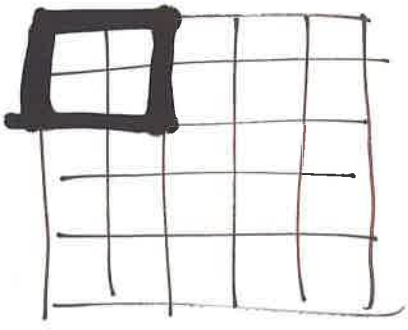
This is the input

1	-1
0	2

This is the filter.

[redacted]

Typically, we would put the filter on the top left of the input:



... and take the inner product... we would get:

$$1 \cdot 1 + (2)(-1) + (-1) \cdot 0 + (-1)(2) = -3$$