**ECE/CS 559: Neural Networks** **Fall 2024**

# Mock Final

**Wednesday** December 11, 2024 — There are **7 questions**; the actual exam will have 5.

**Note**: ***Please write your name on every page!*** *You are expected to work on your own on this exam. This exam is* ***closed notes*** *and no calculators are allowed. Other electronic devices or communication with others are also* ***not*** *allowed. Include your reasoning, not just the final answer. Be clear and concise. Watch your time. If stuck, move on then come back later.* ***Good luck!***

**NAME:** Solutions

— You may use this page and backs of pages for scratch work. —

Lecture 17

HW 7

1. Consider the following data points in $\mathbb{R}^2$: $\begin{bmatrix}-0.01\\1\end{bmatrix}, \begin{bmatrix}1\\1\end{bmatrix}, \begin{bmatrix}0.01\\-1\end{bmatrix}, \begin{bmatrix}-1\\-1\end{bmatrix}$. ($x_1$ $x_2$ $x_3$ $x_4$)

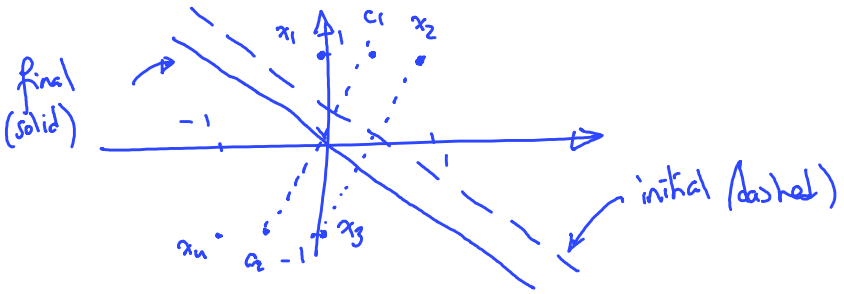
(a) (3 pts) Perform, step by step, Lloyd's algorithm for $k$-means with $k = 2$ until convergence, starting with the second and third points as initial centers. Express the final centers clearly. Draw the initial and final Voronoi cells (along with the data points) on the same plot.

Initialization: $c_1 = \begin{bmatrix}1\\1\end{bmatrix}$ $c_2 = \begin{bmatrix}0.01\\-1\end{bmatrix}$

Step 1: $V_1 = \left\{\begin{bmatrix}-0.01\\1\end{bmatrix}, \begin{bmatrix}1\\1\end{bmatrix}\right\}$ (closer to $c_1$), $V_2 = \left\{\begin{bmatrix}0.01\\-1\end{bmatrix}, \begin{bmatrix}-1\\-1\end{bmatrix}\right\}$ (closer to $c_2$)

updated centers: $c_1 = \begin{bmatrix}0.99/2\\1\end{bmatrix}$ $c_2 = \begin{bmatrix}-0.99/2\\-1\end{bmatrix}$

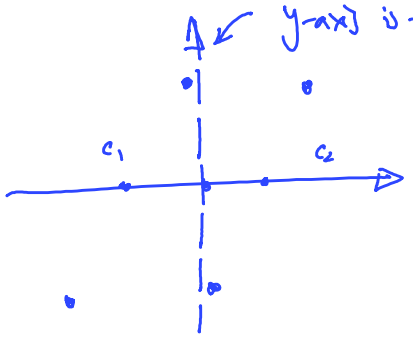
Step 2: $V_1$ the same $V_2$ the same → algorithm stops




(b) (2 pts) Suggest a different initialization that will not converge to the same final centers.

To not converge the same way we can use try and group the points differently in a way that stays consistent with proximity. In particular, group ($x_1$ & $x_4$) and ($x_2$ and $x_3$). Their centers are then $c_1 = \begin{bmatrix}-1.01/2\\0\end{bmatrix}$ and $c_2 = \begin{bmatrix}1.01/2\\0\end{bmatrix}$

If we initialize these, $V_1 = \{x_1, x_4\}$ and $V_2 = \{x_2, x_3\}$ thus the algorithm will not update and stop!



Lecture 18

2. Consider the following data points in $\mathbb{R}^2$: $\begin{bmatrix}0\\1\end{bmatrix}, \begin{bmatrix}1\\1\end{bmatrix}, \begin{bmatrix}0\\-1\end{bmatrix}, \begin{bmatrix}-1\\-1\end{bmatrix}$.

(a) (1 pt) State the competitive (winner-take-all) learning rule in clear mathematical notation. What "kind" of rule is this?

It is a Hebbian rule.

$i = \max_{i'(\text{neurons})} w_i^T x$

$z = w_i^T x$ (output of winner)

update: (only winner)

$w_i \leftarrow w_i + \eta z_i (x - w_i)$

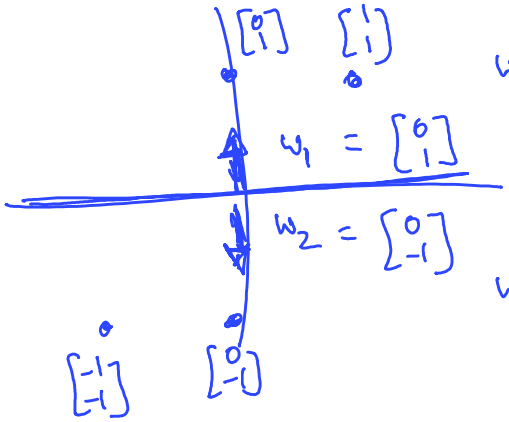
$= (1-\eta z_i) w_i + \eta z_i x$

(b) (4 pts) Perform this learning rule, with one difference: *after every update, round weights to the nearest integer*. Process the data in the given order, for 2 epochs. Use $\eta = \frac{1}{2}$ and initialize with weights $\begin{bmatrix}-1\\2\end{bmatrix}$ and $\begin{bmatrix}2\\-1\end{bmatrix}$. Give the clustering interpretation for the final weights.

(this means we have 2 neurons. will make this explicit)

| $w_1$ | $w_2$ before iteration | Epoch | Iteration | Data Point | winner info | update |
|---|---|---|---|---|---|---|
| $\begin{bmatrix}-1\\2\end{bmatrix}$ | $\begin{bmatrix}2\\-1\end{bmatrix}$ | Ep 1 | It 1 | $x=\begin{bmatrix}0\\1\end{bmatrix}$ | winner $i=1$, $z=2$, | $w_1 = (1-\frac{1}{2}2)w_1 + \frac{1}{2}2\begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}0\\1\end{bmatrix}$ |
| $\begin{bmatrix}0\\1\end{bmatrix}$ | $\begin{bmatrix}2\\-1\end{bmatrix}$ | Ep 1 | It 2 | $x=\begin{bmatrix}1\\1\end{bmatrix}$ | winner $i=1$, $z=1$, (tie, break either way) | $w_1 = (1-\frac{1}{2}1)w_1 + \frac{1}{2}\cdot 1\begin{bmatrix}1\\1\end{bmatrix} = \begin{bmatrix}\frac{1}{2}\\1\end{bmatrix} \xrightarrow{\text{round}} \begin{bmatrix}0\\1\end{bmatrix}$ |
| $\begin{bmatrix}0\\1\end{bmatrix}$ | $\begin{bmatrix}2\\-1\end{bmatrix}$ | Ep 1 | It 3 | $x=\begin{bmatrix}0\\-1\end{bmatrix}$ | winner $i=2$, $z=1$, | $w_2 = \frac{1}{2}w_2 + \frac{1}{2}\begin{bmatrix}0\\-1\end{bmatrix} = \frac{1}{2}\begin{bmatrix}2\\-2\end{bmatrix} = \begin{bmatrix}1\\-1\end{bmatrix}$ |
| $\begin{bmatrix}0\\1\end{bmatrix}$ | $\begin{bmatrix}1\\-1\end{bmatrix}$ | Ep 1 | It 4 | $x=\begin{bmatrix}-1\\-1\end{bmatrix}$ | winner $i=2$, $z=0$, | $w_2 = 1\,w_2 + 0\begin{bmatrix}-1\\-1\end{bmatrix} = \begin{bmatrix}1\\-1\end{bmatrix}$ |
| $\begin{bmatrix}0\\1\end{bmatrix}$ | $\begin{bmatrix}1\\-1\end{bmatrix}$ | Ep 2 | It 5 | $x=\begin{bmatrix}0\\1\end{bmatrix}$ | winner $i=1$ | (no change, since $w_1 = x_1$) |
| $\begin{bmatrix}0\\1\end{bmatrix}$ | $\begin{bmatrix}1\\-1\end{bmatrix}$ | Ep 2 | It 6 | $x=\begin{bmatrix}1\\1\end{bmatrix}$ | winner $i=1$, $z=1$, | $w_1 = \frac{1}{2}w_1 + \frac{1}{2}\begin{bmatrix}1\\1\end{bmatrix} = \begin{bmatrix}\frac{1}{2}\\1\end{bmatrix} \xrightarrow{\text{round}} \begin{bmatrix}0\\1\end{bmatrix}$ |
| $\begin{bmatrix}0\\1\end{bmatrix}$ | $\begin{bmatrix}1\\-1\end{bmatrix}$ | Ep 2 | It 7 | $x=\begin{bmatrix}0\\-1\end{bmatrix}$ | winner $i=2$, $z=1$, | $w_2 = \frac{1}{2}w_2 + \frac{1}{2}\begin{bmatrix}0\\-1\end{bmatrix} = \begin{bmatrix}\frac{1}{2}\\-1\end{bmatrix} \xrightarrow{\text{round}} \begin{bmatrix}0\\-1\end{bmatrix}$ |
| $\begin{bmatrix}0\\1\end{bmatrix}$ | $\begin{bmatrix}0\\-1\end{bmatrix}$ | Ep 2 | It 8 | $x=\begin{bmatrix}-1\\-1\end{bmatrix}$ | winner $i=2$, $z=1$, | $w_2 = \frac{1}{2}w_2 + \frac{1}{2}\begin{bmatrix}-1\\-1\end{bmatrix} = \begin{bmatrix}\frac{1}{2}\\-1\end{bmatrix} \xrightarrow{\text{round}} \begin{bmatrix}0\\-1\end{bmatrix}$ |

Clustering:




winners with $w_1$

winner with $w_2$

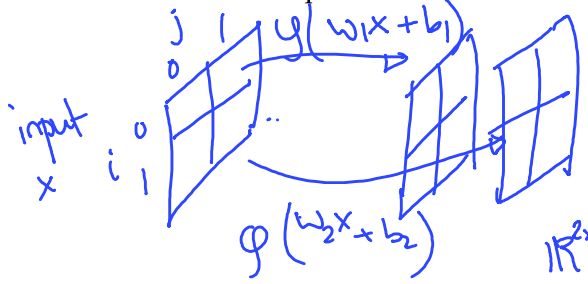
} 2 clusters

3. Consider a simple single-layer CNN, which takes as input a $2 \times 2$ image and produces two $2 \times 2$ channels, by applying kernels of width 1 with bias.

Lecture 16

HW 6

(a) (2 pts) Describe the parameters of this CNN and write the forward equations from the input to the two channels.




channel 1: $w_1, b_1$ (both $\mathbb{R}^1$)

channel 2: $w_2, b_2$ (both $\mathbb{R}^1$)

Forward equations:

$$f^1 = \varphi(w_1 x + b_1) \qquad \mathbb{R}^{2\times 2} \to \mathbb{R}^{2\times 2}, \ \mathbb{R}^1 \to \mathbb{R}^1$$

$$f^2 = \varphi(w_2 x + b_2)$$

(per coordinate)

(b) (3 pts) Write the backpropagation equations for this CNN layer, starting with the gradients of the loss at the output. (You don't need to know the loss.)

We assume we have $\frac{\partial \ell}{\partial f^1_{ij}}$ and $\frac{\partial \ell}{\partial f^2_{ij}}$

at each coordinate of $f^1, f^2$ $\quad i \in \{0,1\}, \ j \in \{0,1\}$

$$\frac{\partial \ell}{\partial w_1} = \frac{\partial}{\partial_1} \ell(f^1_{00}, f^1_{01}, f^1_{10}, f^1_{11})$$

$f^1_{00} \to w_1 x_{00} + b_1$, $f^1_{01} \to w_1 x_{01} + b_1$, $f^1_{10} \to w_1 x_{10} + b_1$, $f^1_{11} \to w_1 x_{11} + b_1$

$$= \sum_{ij} \frac{\partial \ell}{\partial f^1_{ij}} \cdot \frac{\partial f^1_{ij}}{\partial w_1} = \sum_{ij} \frac{\partial \ell}{\partial f^1_{ij}} \cdot x_{ij} \cdot \varphi'(w_1 x_{ij} + b_1)$$

Similarly:

$$\frac{\partial \ell}{\partial w_2} = \sum_{ij} \frac{\partial \ell}{\partial f^2_{ij}} \cdot x_{ij} \cdot \varphi'(w_2 x_{ij} + b_2)$$

$$\frac{\partial \ell}{\partial b_1} = \sum_{ij} \frac{\partial \ell}{\partial f^1_{ij}} \cdot 1 \cdot \varphi'(w_1 x_{ij} + b_1) \qquad \frac{\partial \ell}{\partial b_2} = \sum_{ij} \frac{\partial \ell}{\partial f^2_{ij}} \cdot 1 \cdot \varphi(w_2 x_{ij} + b_2)$$

Gaussian Autoencoder Model

Gaussian/Normal → mean, covariance

$z = \mathcal{N}(0, I)$ (prior)

$z|x = g(x) + \mathcal{N}(0, I) = \mathcal{N}(g(x), I)$ (encoder)

$\hat{x}|z = f(z) + \mathcal{N}(0, \Sigma) = \mathcal{N}(f(z), \Sigma)$ (decoder)

NAME: $P_{prior}$ (forced, not optimized) $P_{enc}$ (optimized via $g$) $P_{dec}$ (optimized via $f$ and $\Sigma$) QUESTION 4

Lecture 20, 21 HW7

4. Consider an autoencoder, with input $x$, embedding $z$, and output $\hat{x}$. Model the prior of the embedding to be a standard Gaussian vector. Let the encoder be a neural network parametrized by $W$ producing an output $g(x; W)$ plus a standard Gaussian noise. Let the decoder be a neural network parametrized by $W$ producing an output $f(z; W)$ plus a Gaussian with a *learnable covariance matrix* $\Sigma$. ($W$ has both sets of parameters)

(a) (2.5 pts) Write the general form of the ELBO loss and specialize it to this autoencoder, simplifying it in the same way as the Gaussian autoencoder in class.

green = same as lecture

The only thing that changes is the decoder's distribution. It becomes:

$$P_{dec}(x|z) \propto \frac{1}{(\det\Sigma)^{1/2}} \exp\left[-\frac{1}{2}(x - f(z;w))^T \Sigma^{-1} (x - f(z;w))\right]$$

The general ELBO loss is $-\mathbb{E}\left[\log \frac{P_{prior}(z)\, P_{dec}(x|z)}{P_{enc}(z|x)}\right]$. Here, it becomes:

$$\mathbb{E}[-\log P_{prior}(z) + \log P_{enc}(z|x)] = \mathbb{E}\left[\text{constant} + \frac{1}{2}\|z\|^2 - \frac{1}{2}\|z - g(x)\|^2\right]$$

$$= \mathbb{E}\left[\text{constant} + \frac{1}{2}\|z\|^2 - \frac{1}{2}\|z\|^2 + z^T g(x) - \frac{1}{2}\|g(x)\|^2\right]$$

drop constants as they don't affect optimization

$$= \mathbb{E}\left[\mathbb{E}\left[z^T g(x) - \frac{1}{2}\|g(x)\|^2 \,|\, X\right]\right] = \mathbb{E}\left[g(x)^T g(x) - \frac{1}{2}\|g(x)\|^2\right] = \frac{1}{2}\mathbb{E}\left[g(x)^2\right] \quad (1)$$

since $\mathbb{E}[z|x] = g(x)$

ignore constants (don't affect optimization)

- $\mathbb{E}[-\log P_{dec}(x|z)] = \text{constant} + \frac{1}{2}\mathbb{E}\left[(x - f(z))^T \Sigma^{-1} (x - f(z))\right]$

$$= \frac{1}{2}\mathbb{E}\left[(x - f(g(x)+N))^T \Sigma^{-1} (x - f(g(x)+N))\right] \quad (2)$$

ELBO = (1) + (2)

In our code, $z$ referred to $g(x)$.

(b) (2.5 pts) In your homework, you concatenated $z$ and the reconstruction error $x - \hat{x}$. Explain why this won't work here. Then, use the fact that you can write $\Sigma^{-1} = M^T M$ to suggest a simple modification that would work.

- In class, we had ELBO = (ignoring constants and constant factors) $\mathbb{E}[\|g(x)\|^2]$ + $\mathbb{E}[\|x - \underbrace{f(g(x)+N)}_{\hat{x}}\|^2]$
  - $\|g(x)\|^2$: $z$ in the code
- The first term is still the same, but the second is changed to:

$$\mathbb{E}\left[(x - \hat{x})^T \Sigma^{-1} (x - \hat{x})\right]$$

- By using the hint, we can reparametrize $\Sigma^{-1} = M^T M$. By substituting:

$$\mathbb{E}\left[(x - \hat{x})^T M^T M (x - \hat{x})\right]$$

- We recognize this as the square norm of $M(x - \hat{x})$.

$$\mathbb{E}\left[\|M(x - \hat{x})\|^2\right]$$

- So, if instead of giving $x - \hat{x}$ to the MSE Loss, we give $M(x - \hat{x})$, concatenated with $z$ (in this case $g(x)$), then we get ELBO.
- If $M$ is learnable ("requires_gradients" in Pytorch), part of the model, then $M$ will be optimized along the way, as desired, via backpropagation.

In office hours there was a question of where exactly we use the idealized GAN loss.
(a) In Lecture 23 ("analyzing GANs"), we considered this same idealized GAN loss (with expectation instead of sum). We said that, because the discriminator is optimizing the cross-entropy between P(S|X) and P_disc(S|X), the best thing to do is to set D(x) = P_disc(S=1|X=x) = P(S=1|X=x). So, finding the optimal D is equivalent to find this posterior distribution.
(b) We also use the idealized loss by using the fact that with the optimal discriminator above, the generator is optimizing the Jensen-Shannon divergence between P(X) (the true/population) and P(G(Z)). This is almost a "distance" and it is smallest when zero, which is achieved by setting them to be equal. In other words, it is achieved by making G(Z) have the same distribution as X, as the question states.



Lectures 22, 23

5. Consider a GAN to generate a real-valued random variable $X$. Say the population of $X$ is distributed in $[0,1]$ with density $f_X(x) = 2 - 2x$. Use a noise $Z$ uniformly distributed in $[0,1]$. Use the idealized GAN loss we considered in class:

$$\underset{X,Z}{\mathbf{E}}\left[-\log D(X) - \log(1 - D(G(Z)))\right]$$

(a) (2.5 pts) Say the initial generator is just the identity function $G(z) = z$. What is the optimal discriminator in this case?

We saw in class that the optimal discriminator is:

$$D(x) = \mathbb{P}(S=1 \mid X=x) = \frac{\mathbb{P}(S=1)\cdot \mathbb{P}(X=x \mid S=1)}{\mathbb{P}(S=0)\,\mathbb{P}(X=x \mid S=0) + \mathbb{P}(S=1)\,\mathbb{P}(X=x \mid S=1)}$$

$\mathbb{P}(X=x \mid S=1) = f_X(x) = 2-2x$

$\mathbb{P}(X=x \mid S=0) = f_Z(x) = 1$ (since $G$ is passing $Z$).

$\mathbb{P}(S=0) = \mathbb{P}(S=1) = \frac{1}{2}$

$$\Rightarrow D(x) = \frac{2-2x}{2-2x+1} = \frac{2-2x}{3-2x}$$

(b) (2.5 pts) What is the optimal generator $G$? That is, find $G$ such that $G(Z)$ has the same distribution as $X$.

Lecture 19 ("derived distributions")

$G$ is not necessarily unique, but let's choose it monotonically increasing. This, in particular, means that it's invertible: $x = G(z)$, $z = G^{-1}(x)$.

$$G(Z) \sim X \Rightarrow \mathbb{P}(G(Z) \le x) = \mathbb{P}(X \le x) = F_X(x) = \int_0^x f_X(t)\,dt$$

$$= \int_0^x 2-2t\,dt = 2t - t^2 \Big]_0^x = 2x - x^2$$

But also $\mathbb{P}(G(Z) \le x) = \mathbb{P}(Z \le \underbrace{G^{-1}(x)}_{z}) = F_Z(z) = z$ (uniform)

$$z = G^{-1}(x) = 2x - x^2 \iff z = 2G(z) - G(z)^2 \Rightarrow 1 - z = [1 - G(z)]^2$$

$$\Rightarrow G(z) = 1 - \sqrt{1-z}$$

(the other solution produces $G(z) \notin (0,1)$, which isn't acceptable!)

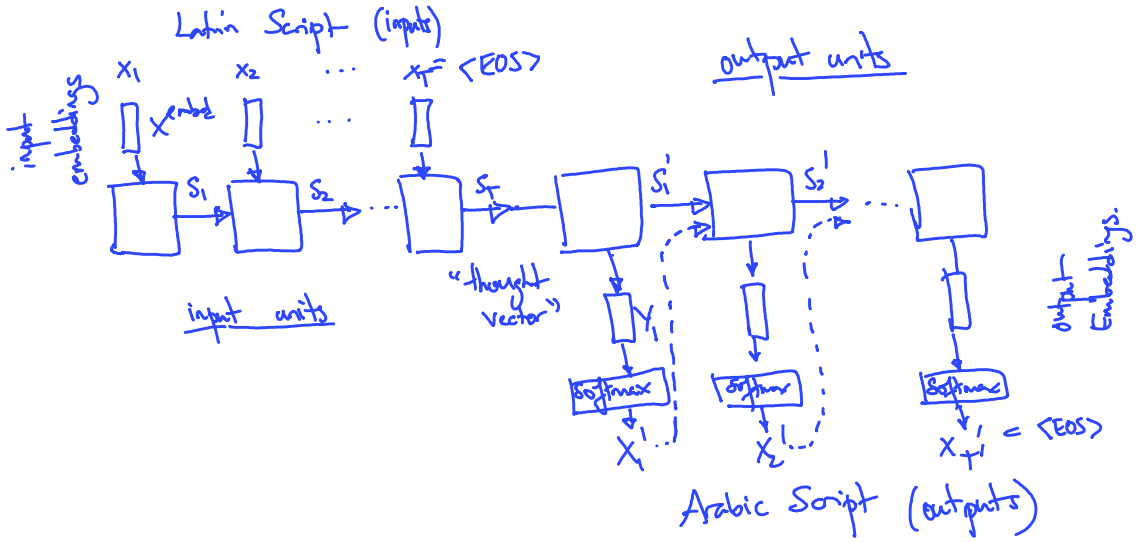
Lecture 24

6. Say we would like to build an ML model that takes as input Arabic return in Latin script, and outputs the same text in Arabic script. When written in Latin script, many Arabic letters require multiple characters to represent,

   (a) (1 pt) What kind of sequence model is most appropriate for this problem? Explain why.

   This is an asynchronous sequence to sequence model, because the output is a sequence and the input and output don't line up exactly.

   (b) (4 pts) Sketch an RNN architecture that you could use for this model. Draw a diagram with embeddings, units, inputs, and outputs clearly marked. Write a simple input-state-output equation for each unit.




- Input embedding $x_t^{embed} = Ax_t$, where $x_t$ = one-hot
- Input units: $s_t = \varphi(W^x x_t + W^s s_{t-1} + b)$
- Output units $s'_t = \varphi(V^s s_{t-1} + V^x x'_{t-1} + c)$

  optional but will make the RNN aware of what's generated
- Output embedding $y'_t = \varphi(U s'_t + d)$
- Outputs: $x'_t \sim$ Sample from $\text{Softmax}(y_t)$.

$$= \frac{e^{y_t[j]}}{\sum_{j'} e^{y_t[j']}}$$

I should have named this differently to not confuse with values.

Lectures 25, 26

7. Say you have a graph $\mathcal{G} = (V, E)$ consisting of vertices $v \in V$ and edges $E \subset V^2$, the subset of pairs of vertices that are connected. You want to make a binary classification (e.g., is there a disturbance or not) that depends on the inputs $x_v$, sitting at each node.

   (a) (2 pts) You decide to design a *graph* self-attention, by remaining consistent with the graph at any given layer, i.e., uses only inputs connected to each vertex. Write the self-attention equations at a node $v$ in clear mathematical notation.

The only thing we need to change is to make the attention over neighbors, i.e. $v'$ s.t. $(v,v') \in E$

$Q_v = W^Q x_v$, $K_{v'} = W^K x_{v'}$, attention at $v$:

$$\alpha_{v,v'} = \underset{v':(v,v')\in E}{\text{Softmax}}\left(\frac{1}{\sqrt{k}} Q_v^T K_{v'}\right) = \begin{cases} 0 & ; \text{if } (v,v') \notin E \\ \frac{e^{\frac{1}{\sqrt{k}} Q_v^T k_{v'}}}{\sum_{v':(v,v')\in E} e^{\frac{1}{\sqrt{k}} Q_v^T K_{v'}}} & ; \text{if } (v,v') \in E \end{cases}$$

   (b) (3 pts) Explain (1) how you would build transformer layers using this modified self-attention, (2) whether or not the outputs of the last layer will only depend on neighboring inputs in the graph, and (3) how you would use the last layer outputs to make the binary classification with a clear description of the architecture and loss function.

(1) Calculate value at each node $V_{v'} = W^V x_{v'}$.
Combine according to attention: $S_v = \sum_{v':(v,v')\in E} \alpha_{v,v'} V_{v'}$
Possibly create multiple heads + concatenate
Pass through add+norm, FF, add+norm, get $x_v^{new}$.
Repeat, to obtain multiple layers.

(2) The output at higher layers can depend on nodes further than the immediate neighbors, e.g. on the second layer, they will depend on neighbors of neighbors. (information propagates)

(3) Say $x_v^{last}$ are the outputs of the last layer.
One approach is to add a linear layer + sigmoid

- $f = \sigma\left(\sum_v W_v x_v^{out} + b\right) \in (0,1)$

We can then use binary cross entropy, if data is $(x,y)$ binary

- $\ell(f,y) = y \log\frac{1}{f} + (1-y)\log\frac{1}{1-f}$.

**ECE/CS 559: Neural Networks** **Fall 2024**

# Homework 7

Due: **Tuesday** November 26, 2024 (by 9pm, on Gradescope)

**Note**: *You may discuss these problems in groups. However, you must write up your own solutions and mention the names of the people in your group. Also, please do mention any solution manuals, online material, or other sources you used in a major way.*

Submit your code as a **two** `.py` **files**, one for each question, in Gradescope. Make sure that it **runs properly** and generates all the plots that you report on in the main submission.

1. **[K-Means using a Neural Network]**

   Load the MNIST dataset in PyTorch. Your goal is to cluster the images into $k = 10$ clusters, without relying on the label. Reuse your prior code, especially the loading, training, and testing aspects based on the PyTorch tutorial. Make the following significant changes.

   - Build a neural network that defines three elements in its constructor: a parameter `self.centers` that is a $10 \times 784$ tensor defined using `nn.Parameter` and referring to the centers matrix $C$, a flatten layer, and a softmax layer.
   - In the forward pass, do this:
     - Use the flatten layer on the input `x` and call the resulting variable `x0`, to keep access to it, and keep using `x` for subsequent stages.
     - Use the following single tensor equation to manually implement the 10 neurons of the first layer that we saw in class, for $i = 0, 1, \cdots, 9$, $xC_i^\intercal - \frac{1}{2}\|C_i\|^2$:
       ```
       x = torch.matmul(x0, self.centers.t())
             -0.5*torch.sum(self.centers**2,1).flatten()
       ```
     - Multiply the result by 20 then pass it through a softmax layer.
     - Reconstruct the input using a center, by multiplying the softmax output with the tensor $C$, from the right. [1-D tensors in PyTorch are row vectors.]
     - Return the error `x-x0`, where `x` is the reconstruction/center.
   - Use the MSE loss. To do this, when you call the loss, use one argument as the model's prediction (which is simply the reconstruction error) and let the other argument be a 0 tensor, the same size as the prediction batch (so that it computes compute $\|(x - x_0) - 0\|^2 = \|x - x_0\|^2$.)

     [The reason we're doing this is in order not to bother with manipulating batches ourselves. When you want the center, you can still get it, by adding the image to the reconstruction error: $(x - x_0) + x_0 = x$.]

- Modify the test loop to only evaluate and display loss, and not accuracy.

(a) Say we're using a batch size of $b = 64$. Recall that PyTorch by default *averages* the losses over each batch. Assume that, in each batch, roughly $\frac{1}{10}^{\text{th}}$ of the images are in each cluster, and assume that the softmax behaves likes an argmax. Explain why the update at iteration $t$ looks like:

$$C_i(t) = \left(1 - \frac{\eta}{5}\right) C_i(t-1) + \frac{\eta}{5} \underbrace{\frac{1}{|\mathcal{V}_i(t)|} \sum_{x \in \mathcal{V}_i(t)} x}_{m_i(t)}$$

where $\mathcal{V}_i(t)$ are all images in the Voronoi cell of $C_i$, in this batch. $m_i(t)$ are the average centers in each cell *only in this batch*. Use $\eta = 4.5$. See the last page of the homework for an explanation of why this is a good idea for approximating Lloyd's algorithm, with the batch size we chose.

(b) Cheat, by initializing the cluster centers (`model.centers`) to individual images from the corresponding digit, i.e., set $C_i$ to an image of digit $i$ by saying:

`model.centers[i,:] =` a flattened version of the image.

Then, train your $k$-means neural network for 10 epochs and produce two outputs (pasted in your report):

i. All the cluster centers at the end of the training.

ii. A table of size $10 \times 10$ that contains for each row $i$ and column $j$, the number of training images of digit $i$ that were clustered in center $j$. (The sum of all entries should be 60,000.)

Discuss these results by commenting on what places $k$-means clustering is having trouble and what you think the reasons are behind it. [Hint: Use a data loader with batch size 1 to iterate over the data points individually.]

(c) Try to initialize the centers differently (e.g., sampling each coordinate uniformly randomly from $[0, 1]$), and try training the network again. Report on your choice of initialization and what kind of results you obtain (e.g., do you see something that you could reasonably call "mode collapse"?)

2. **[Autoencoder]**

Continue to use the same base code. Replace your neural network with an autoencoder. Let your encoder be, in one sequence, the same as the CNN from HW6 Q3 ending in a 10-dimensional representation `z`. For the decoder, use the following (roughly inverted) architecture in one sequence:

- A fully-connected stage with ReLU activations, going from 10 to 360, then from 360 to 720.

- Unflatten dimension 1 (we don't touch dimension 0, as it is the batch dimension) to `torch.size([20,6,6])`. This gives 20 channels of 6×6 images.
- Perform bicubic upsampling by a factor of 2.
- Use a (transpose) convolutional layer with 20 output channels, 4×4 filters, stride 2, and 1 output padding, followed by ReLU.
- Use a final (transpose) convolutional layer with 1 output channel, 4×4 filters, **stride 1**, **input padding 1**, and no output padding, followed by sigmoid.

Add batch normalization and dropout (with probability 0.1) after every ReLU activation, in both the encoder and decoder. Your final output should be 28×28.

In the forward pass, do this:

- Get `z` by passing the input image `x` through the encoder.
- Add standard Gaussian noise to `z` to make `z2`. To make generation easy, give the forward function an additional binary argument called `enc_mode`, set by default to 1, and use:

  ```
  z2 = enc_mode*z + (2-enc_mode)*torch.randn(z.shape)
  ```

  If we're in encoding mode (during training), this will act as additive noise. During generation mode (`enc_mode=0`), this will replace `z` itself with Gaussian noise. (Slightly different than in class: we're adding all noise before the decoder.)
- Get the output image `f` by passing `z2` through the encoder.
- Flatten `f` and return the concatenation of `z` (the embedding) and `f-x` (the reconstruction error), along dimension 1 (we don't touch dimension 0, as it is the batch dimension).

The reason we do this is because we're going to simulate the ELBO loss, and we saw that for Gaussian autoencoders it is simply the mean square of the embedding + reconstruction error. Therefore, you can simply use `MSELoss`. Set the batch size to 64 and the learning rate to 0.1.

(a) Draw a rough tensor block sketch of the encoder architecture and, to its right, the decoder architecture. Highlight which stage in the first is "inverted" by which stage in the second, by connecting them with brackets underneath.

(b) Train your autoencoder for 10 epochs. Visualize its performance by generating 20 random digits, by calling the model with the additional argument `enc_mode=0`. Note that you should eliminate the first 10 entries, which correspond to the embedding, by using a slicing, e.g., `[0,10:]`.

(c) Suggest modifications to the architecture, regularization, or training to obtain better results. Report your changes and visualize the results in the same way.

In Q1, using the minibatch approach has some advantages, including faster processing and avoiding local minima. The minibatch updates average centers *only within the minibatch*,

$$C_i(t) = \left(1 - \frac{\eta}{5}\right) C_i(t-1) + \frac{\eta}{5} \underbrace{\frac{1}{|\mathcal{V}_i(t)|} \sum_{x \in \mathcal{V}_i(t)} x}_{m_i(t)}$$

Lloyd's algorithm, in contrast, averages centers *across the whole data set*. This means we have some bias (offset) and variance (noisiness) due to using minibatches. For a fixed batch size, the choice of $\eta$ creates a tradeoff between bias and variance.

To understand this, let's focus on when $\eta \in [0, 5]$ (bigger $\eta$ will make learning unstable by moving *away* from prior centers.) We first explain the intuition of why bias and variance cannot be improved simultaneously, i.e., there's a tradeoff between them. Let's first think about the extremes. If $\eta = 0$, then we do not make updates. The variance is thus 0, but the bias is very large: we're very far from what we should be doing, which is averaging centers. If $\eta = 5$, we cancel everything from before and only use the last minibatch. This is the least amount of data we could use, and thus the variance is large, however $m$ is doing what we should be doing, it is an unbiased estimate of what Lloyd's would have calculated.

How can we estimate bias? Because successive updates shift the centers, they cause $m$ to drift away from Lloyd's average. By how much? The average pixel value is about $a = 0.23$. We'll assume that the latest average is the "freshest" and that the further back we look in batches, each coordinate of $m$ could drift by as much as $a$ times how far back we look, in average. Regarding noisiness, we can assume the noise level of each data point is constant. Recall that $b$ is the minibatch size. The minibatch averaging reduces the variance by $\frac{1}{b}$, then averaging across minibatches will reduce the variance based on how the averaging is being done.

To simplify the notation, let's consider a single coordinate. Also, let $\alpha = \eta/5$. Thus, we have that $\alpha \in [0, 1]$, and . Let $N$ be the training size and let $T$ be the number of minibatches, so $T = N/b$.We have

$$\begin{aligned}
C(T) &= (1-\alpha)C(T-1) + \alpha m(T) \\
&= (1-\alpha)^2 C(T-2) + (1-\alpha)\alpha m(T-1) + \alpha m(T) \\
&= \ldots \\
&= (1-\alpha)^T C_0 + \sum_{t=1}^{T} \alpha (1-\alpha)^{T-t} m(t) \\
&\approx \sum_{t=1}^{T} \alpha (1-\alpha)^{T-t} m(t)
\end{aligned}$$

where the last approximation is true if $\alpha < 1$ and $T$ is large. In our case with $b = 64$ and $N = 60,000$, $T$ is indeed quite large.

Let $M$ be the ideal average of all centers. We can model our assumptions of bias and variance roughly as:

$$m_t = M + a(T-t) + aZ_t/\sqrt{b}, \qquad Z_t \sim \mathcal{N}(0,1) \text{ (i.i.d.)}$$

We can now calculate our bias and variance:

$$\begin{aligned} \text{Bias} &= M - \mathbf{E}[C(T)] = M - \sum_{t=1}^{T} \alpha(1-\alpha)^{T-t}[M + a(T-t)] \\ &= M - \sum_{s=0}^{T-1} \alpha(1-\alpha)^{s}[M + as] \\ &\approx \frac{1-\alpha}{\alpha} a \end{aligned}$$

where the last expression takes $T \to \infty$, uses the fact that $\alpha(1-\alpha)^s$ is a the distribution of a geometric random variable with parameter $\alpha$ starting at 0. We use both the fact that this distribution sums to 1 (to cancel the $M$'s) and that formula for its expected value is $(1-\alpha)/\alpha$.

$$\begin{aligned} \text{Variance} &= \sum_{t=1}^{T} \left[\alpha(1-\alpha)^{T-t}\right]^2 \frac{a^2}{b} \\ &= \frac{1}{b} \frac{\alpha^2}{1-(1-\alpha)^2} \sum_{s=0}^{T-1} (1-(1-\alpha)^2)((1-\alpha)^2)^s \\ &\approx \frac{a^2}{b} \frac{\alpha^2}{1-(1-\alpha)^2} \end{aligned}$$

where the last expression takes $T \to \infty$ and uses the fact that a geometric distribution sums to 1.

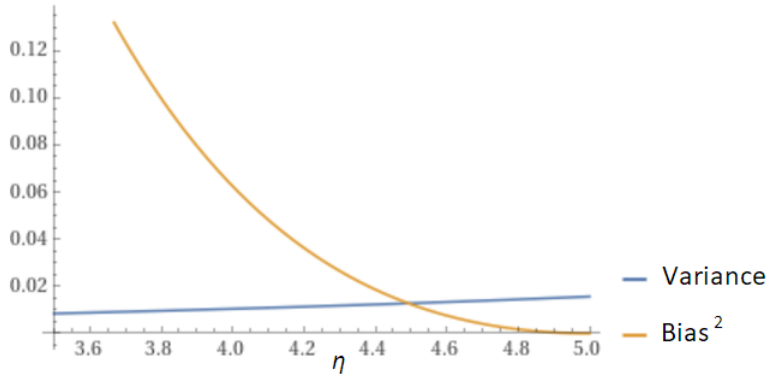
We now have estimates of what the repercussion is on bias and variance for a specific choice of $\eta$ (or $\alpha$), for a given batch size $b$ and assuming average pixel value of $a$. As we observed with the extreme cases, we can see that with increasing $\alpha$ (or $\eta$), the bias goes down, but the variance goes up.

Bias and Variance combine as $\text{Bias}^2 + \text{Variance}$ to give the overall error, so to balance them out, we need to find a point where:

$$\left(\frac{1-\alpha}{\alpha} a\right)^2 \approx \frac{a^2}{b} \frac{\alpha^2}{1-(1-\alpha)^2}$$

(We could also try to minimize the sum, because we don't quite know the leading coefficients of each in this back-of-the-envelope calculation, those are both good heuristics.) Notice that because $a$ affects both sides equally, it comes out in the wash. In terms of $\eta$, we need to find where the two curves touch, which we plot below:

$$\left(\frac{1-\frac{\eta}{5}}{\frac{\eta}{5}}\right)^2 \quad \text{and} \quad \frac{1}{64}\frac{(\frac{\eta}{5})^2}{1-(1-\frac{\eta}{5})^2}$$




From this, we see that the balance is achieved around $\eta = 4.5$, which is the choice suggested by the question. This also gives us an idea of how we should change $\eta$ as we change $b$. For example, when you set $b = N = 60,000$, we should choose $\eta \approx 5$, reverting back to Lloyd's algorithm exactly. And if we set $b$ smaller, we should make $\eta$ smaller.

**ECE/CS 559: Neural Networks** **Fall 2024**

# Homework 5

Due: **Tuesday** October 1, 2024 (by 9pm, on Gradescope)

**Note**: *You may discuss these problems in groups. However, you must write up your own solutions and mention the names of the people in your group. Also, please do mention any solution manuals, online material, or other sources you used in a major way.*

Submit your code as a **two** `.py` **files**, one for each question, in Gradescope. Make sure that it **runs properly** and generates all the plots that you report on in the main submission.

1. **[Gradient Descent]**

   Let $\mathrm{w} \in \mathbb{R}^2$. Consider the following function:

   $$R(\mathrm{w}) = 13w_1^2 - 10w_1w_2 + 4w_1 + 2w_2^2 - 2w_2 + 1$$

   (a) Find the gradient of $R$ and solve for the optimality condition exactly.

   (b) Implement gradient descent. At each iteration, calculate and save the distance between the iterate and the optimal solution. Run this with $\eta = 0.02, 0.05$, and 0.1 and 500 iterations each. Plot distance vs. iteration in each case. Report it.

   (c) Is even larger $\eta$ beneficial? Why or why not?

2. **[Backpropagation]**

   Revisit the dataset you generated in HW2 Q2. Imagine that a friend gives you the $1,000$ points generated with that neural network, tells you what the architecture looks like (but not the weights), then asks you to train a neural network that imitates theirs.

   (a) You want to use gradient descent, but know that you can't use a step function to do the training. Choose instead to use a sigmoid function $\varphi$ with parameter $a = 5$. Write down $\varphi$ and $\varphi'$ explicitly and implement Python functions for each, that operate on `numpy` arrays.

   (b) Call the first layer weights $W$ and its biases $b$, the second layer weights $U$ and its bias $c$, and the output $f$. Let the hidden layer output be $z$. Write down the dimensions of all weights, biases, and variables, then write down the forward equations from $x$ to $v_z$, from $v_z$ to $z$, from $z$ to $v_f$, and from $v_f$ to $f$. Translate these into Python code.

(c) Use the squared loss $\ell(y, f) = (f - y)^2$. Since we are targeting binary outputs and $\varphi$ limits values to $[0, 1]$, this is not that different from 0-1 loss. Start writing down the backward equations for a single data point, by first writing the expression for $\nabla_f \ell$ followed by the expression for $\delta_f$. Then, write the backward equation from $\delta_f$ to $\delta_z$, then from $\delta_z$ to $\delta_x$. Use these along with $x$ and $z$ to compute the gradients $\nabla_W \ell$, $\nabla_b \ell$, $\nabla_U \ell$, and $\nabla_c \ell$. (Remember that these gradients should have the same dimensions as the respective weights and biases.) Translate these into Python code.

(d) In Python, initialize all your weights and biases with i.i.d. Gaussian$(\mu = 0, \sigma = 0.1)$ samples. Then, create an outer `epoch` loop and inner `i` loop ranging over the $1,000$ data points. For each data point $(x, y)$, run the forward pass code from (b), then the backward pass code from (c). Then, perform gradient descent with that single point (this is a form of *stochastic gradient descent, SGD*): $W \leftarrow W - \eta \nabla_W \ell$, etc. Use $\eta = 0.01$ and perform 100 epochs. At the end of each epoch, calculate and save the MSE. At the end, plot and report the MSE vs. epoch curve.

(e) In HW2 Q2, you visualized the decision boundary of your friend's neural network. Visualize the decision boundary of yours, by plotting a 3D scatter plot of $(x_1, x_2, f(x_1, x_2))$ using the weights at the end of your training. Report the plot. (You can also visualize this during the training, to see how the boundary evolves.) You may find the following snippet useful.

```
fig = plt.figure(figsize = (7, 10))
ax = plt.axes(projection ="3d")
ax.scatter3D(x1, x2, y_predicted, color = "green")
ax.set_xlabel('x1')
ax.set_ylabel('x2')
ax.set_zlabel('y')
ax.zaxis._axinfo['juggled'] = (2, 2, 1)
plt.show()
```

(f) Try at least three of the following hacks individually or simultaneously. Say what you tried, then report on the MSE vs. epoch curve, the decision boundary, a description of what changed, and your hypothesis as to why.

- Change $\eta$.
- Start $\eta$ high but multiply by 0.9 if the MSE increases (or if it increases for $T$ epochs, where you choose $T$).
- Choose a different $a$ in $\varphi$.
- Change the number of epochs that you perform.
- Do a proper gradient descent (accumulate gradients, update only in the end of an epoch, zero-out gradients, and move to next epoch).
- Use minibatches (accumulate $B$ gradients, update, zero-out, and continue).
- Change the architecture by including more or fewer hidden neurons.
- Change how you initialize the weights and biases, e.g. by setting them to 0 or choosing $\sigma$ very small or large.

# Question 1

(a) $R(w) = 13w_1^2 - 10w_1w_2 + 4w_1 + 2w_2^2 - 2w_2 + 1$

$$\frac{\partial R}{\partial w_1} = 26w_1 - 10w_2 + 4 = 0 \quad (1)$$

$$\frac{\partial R}{\partial w_2} = -10w_1 + 4w_2 - 2 = 0 \quad (2)$$

(optimality)

$$(2) \Rightarrow -25w_1 + 10w_2 - 5 = 0 \quad (3)$$

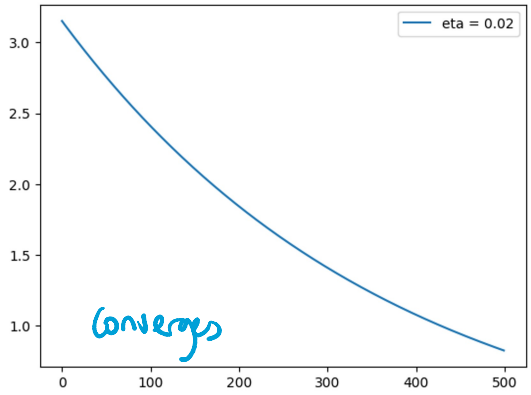
$$(1) + (3) \Rightarrow w_1 - 1 = 0 \Rightarrow w_1 = 1 \quad (4)$$

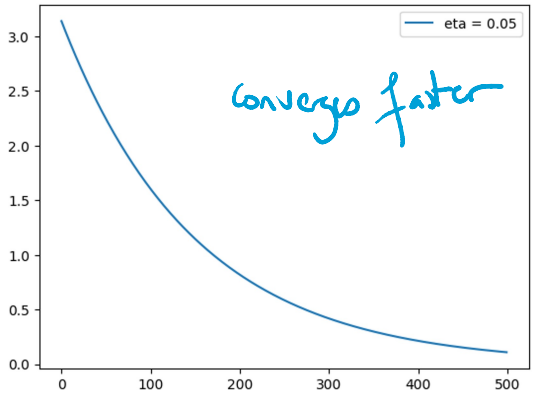
$$(2) + (4) \Rightarrow -10 + 4w_2 - 2 = 0 \Rightarrow w_2 = 3$$

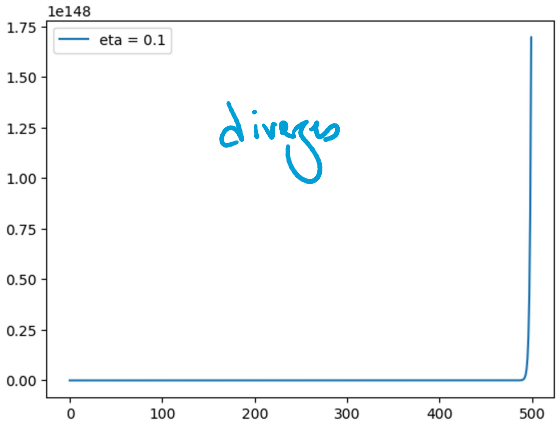
(b)

```
import numpy as np
import matplotlib.pyplot as plt
w_opt=[1,3]
number_of_epochs = 500
for eta in [0.02, 0.05, 0.1]:
  dist=np.zeros(number_of_epochs)
  w=np.zeros((2,))
  for i in range(number_of_epochs):
    grad = np.array([4+26*w[0]-10*w[1],-2-10*w[0]+4*w[1]])
    w = w - eta*grad
    dist[i]=np.linalg.norm(w-w_opt)
```

```
plt.plot(dist,label="eta = "+ str(eta))
plt.legend()
plt.show()
```




(c) Larger $\eta$ is not always beneficial, because it would lead to instability (even if the gradients are exact, like here). This is because the 1st-order Taylor approximation is no longer very accurate.

## Question 2

(a) $\varphi(v) = \frac{1}{1+e^{-5v}}$

$\varphi'(v) = 5\varphi(v)(1-\varphi(v))$

(b)

| | | | |
|---|---|---|---|
| $W$ | is | $3 \times 2$ | $(\dim(z) \times \dim(x))$ |
| $b$ | is | $3$ | $(\dim(z))$ |
| $U$ | is | $1 \times 3$ | $(\dim(f) \times \dim(z))$ |
| $c$ | is | $1$ | $(\dim(f))$ |

Forward equations!

$v_z = Wx + b \qquad z = \varphi(v_z)$

$v_f = Uz + c \qquad f = \varphi(v_f)$

```
v_z = np.matmul(W,x)+b
z = phi(v_z)
v_f = np.matmul(U,z)+c
f = phi(v_f)
```

(c) Backward equations for $\ell(y,f) = (y-f)^2$

$\nabla_f \ell = \frac{\partial \ell}{\partial f} = -2(y-f) \qquad \delta_f = \frac{\partial f}{\partial \ell} \cdot \varphi'(v_f)$

$$\nabla_f \ell = \frac{\partial \ell}{\partial f} = -2(y-f) \qquad \delta_f = \frac{\partial f}{\partial e} \cdot \varphi'(v_f)$$

$$\nabla_z \ell = U^T \delta_f \qquad \delta_z = \nabla_z \ell \cdot \varphi'(v_z)$$

$$\nabla_W \ell = \delta_z x^T \qquad \nabla_b = \delta_z$$

$$\nabla_U \ell = \delta_f z^T \qquad \nabla_c = \delta_f$$

```
dloss_df = -2*(y-f)
delta_f = dloss_df * phi_prime(v_f)
dloss_dz = np.matmul(np.transpose(U),delta_f)
delta_z = dloss_dz * phi_prime(v_z)
grad_W = np.matmul(delta_z, np.transpose(x))
grad_b = delta_z * 1
grad_U = np.matmul(delta_f, np.transpose(z))
grad_c = delta_f * 1
```

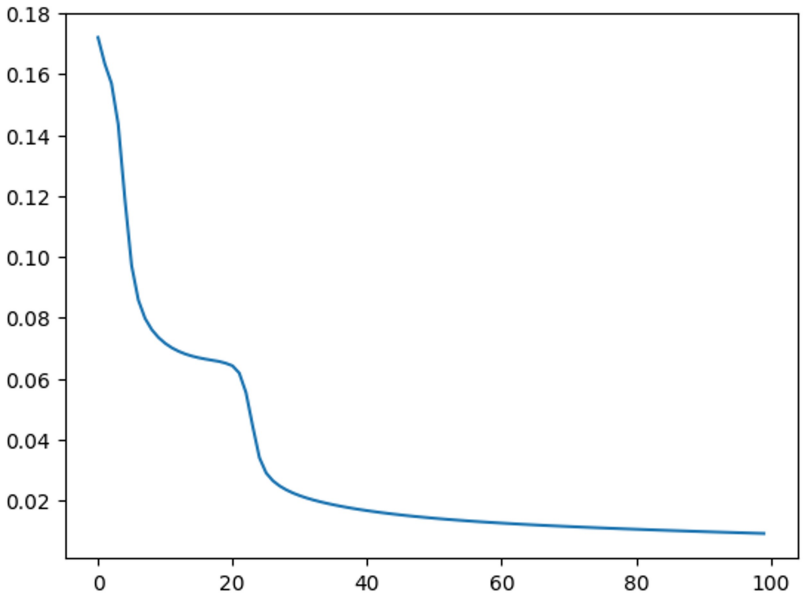
(d) Initialization:

```
sigma=0.1
W = np.random.randn(k,X.shape[0])*sigma
b = np.random.randn(k,1)*sigma
U = np.random.randn(1,k)*sigma
c = np.random.randn(1,1)*sigma
```

SGD:

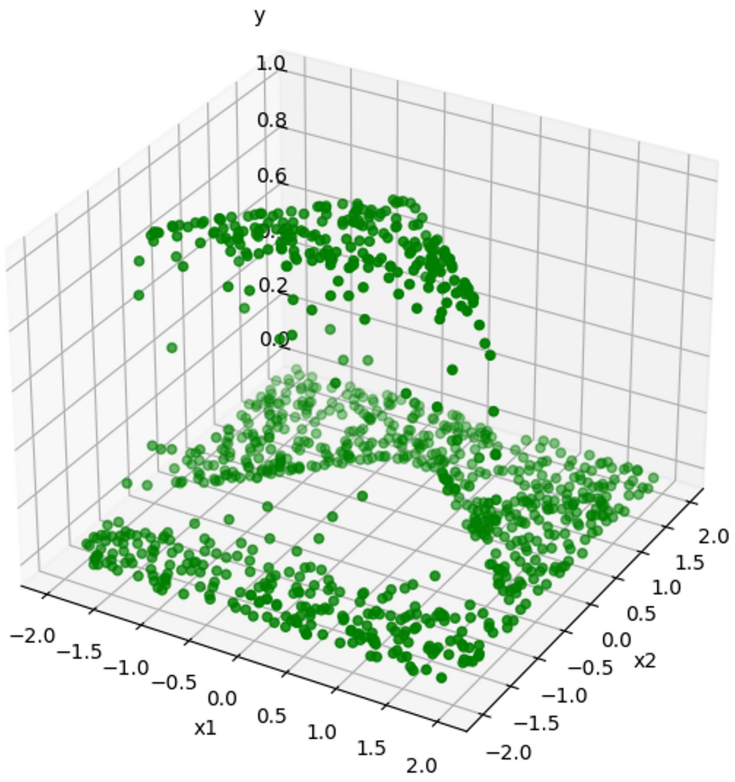
```
eta = 0.01
epochs = 100
risk = np.zeros(epochs)
```

```
for epoch in range(epochs):
  for i in range(N):
    x = X[:,i].reshape(x.shape)
    y = Y[0,i].reshape(f.shape)
    # Forward - Question 2(b)
    v_z = np.matmul(W,x)+b
    z = phi(v_z)
    v_f = np.matmul(U,z)+c
    f = phi(v_f)
    # Backward - Question 2(c)
    dloss_df = -2*(y-f)
    delta_f = dloss_df * phi_prime(v_f)
    dloss_dz = np.matmul(np.transpose(U),delta_f)
    delta_z = dloss_dz * phi_prime(v_z)
    grad_W = np.matmul(delta_z, np.transpose(x))
    grad_b = delta_z * 1
    grad_U = np.matmul(delta_f, np.transpose(z))
    grad_c = delta_f * 1
    # SGD
    W = W - eta * grad_W
    b = b - eta * grad_b
    U = U - eta * grad_U
    c = c - eta * grad_c
  Y_predicted = phi(np.matmul(U,phi(np.matmul(W,X)+b))+c)
  risk[epoch] = np.sum((Y-Y_predicted)**2)/N
plt.plot(risk)
plt.show()
```



(e) Visualization

```
fig = plt.figure(figsize = (7, 10))
ax = plt.axes(projection ="3d")
ax.scatter3D(x1, x2, Y_predicted, color = "green")
ax.set_xlabel('x1'); ax.set_ylabel('x2'); ax.set_zlabel('y')
ax.zaxis._axinfo['juggled'] = (2, 2, 1)
plt.show()
```




(e) This question is a bit open-ended.

- Changing $\eta$ has its usual pitfalls. An adaptive $\eta$ works remarkably well (many methods like Adam and Adagrad do this by monitoring the

Adam and Adagrad do this by monitoring the gradients themselves.)

- Minibatch is helpful to smooth out the gradients of SGD. However, we shouldn't simply accumulate the gradients, since then ther variance increases. Instead, we need to average across the minibatch.

- Initializing weights at 0 slows the convergence considerably, even halting it (by getting stuck in local minima.) Local minima are also an issue with very large $\sigma$ for the random initialization.

- Making a big in $\varphi$ makes it "less differentiable" do more unstable, because of both large (at the transition) and very small (at saturation) gradients. However,

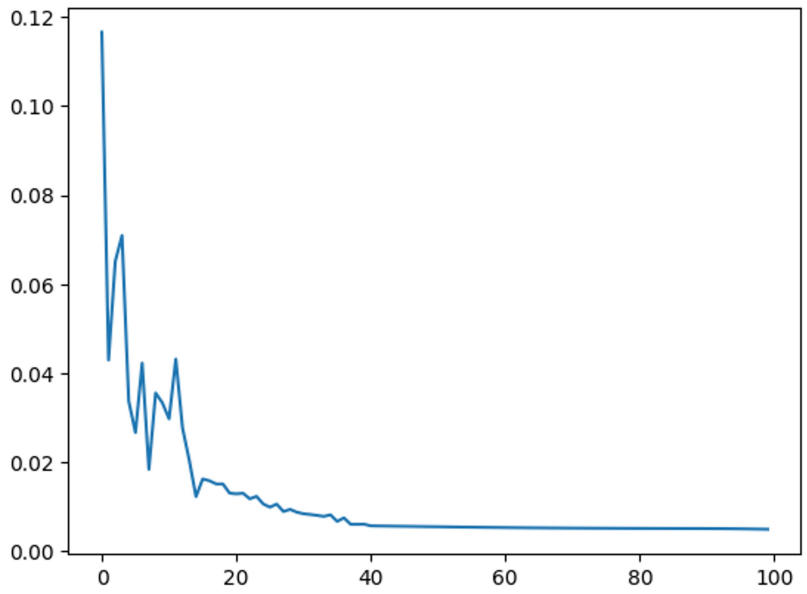
small (at saturation) gradients. However, the end results are crisper, as long as we can stabilize (adaptive $\eta$ + minibatch help.)

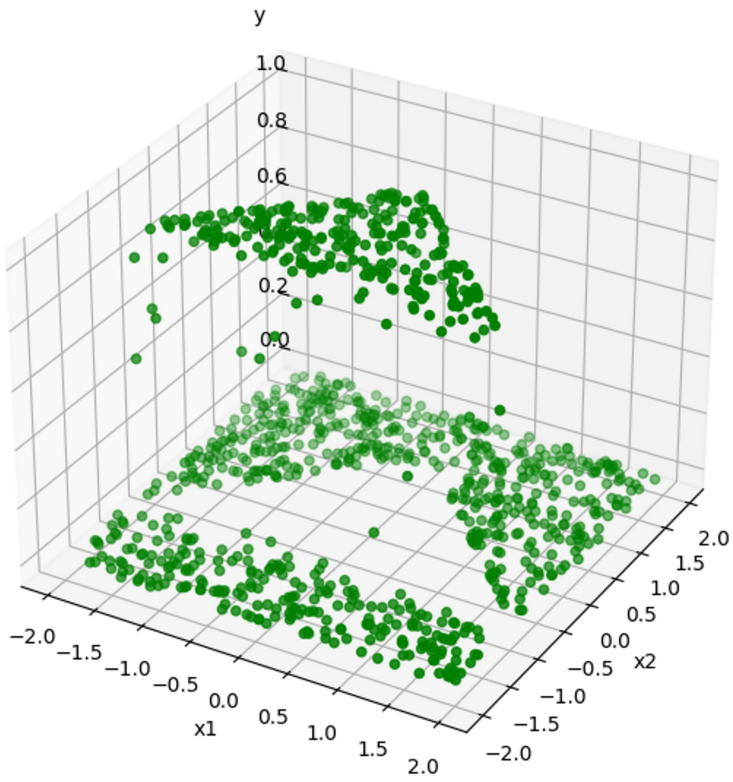
The following code does:

- Sets $\alpha = 10$ in $\varphi$.
- Adaptive $\eta$, starting at $\eta = 0.1$
- Minibatch with batch size 5, and with averaged gradients.

```
def phi(v):
  return 1./(1.+np.exp(-10*v))
def phi_prime(v):
  return 10*phi(v)*(1-phi(v))
sigma=0.1
# Initialization
W = np.random.randn(k,X.shape[0])*sigma
b = np.random.randn(k,1)*sigma
U = np.random.randn(1,k)*sigma
c = np.random.randn(1,1)*sigma
# W=W*0 # Zero initialization
# b=b*0
# U=U*0
# c=c*0
eta = 0.1
epochs = 100
batch_size = 5
risk = np.zeros(epochs)
grad_W = 0*W
grad_b = 0*b
grad_U = 0*U
grad_c = 0*c
for epoch in range(epochs):
  for i in range(N):
    x = X[:,i].reshape(x.shape)
    y = Y[0,i].reshape(f.shape)
    # Forward
    v_z = np.matmul(W,x)+b
    z = phi(v_z)
```

```
    v_f = np.matmul(U,z)+c
    f = phi(v_f)
    # Backward
    dloss_df = -2*(y-f)
    delta_f = dloss_df * phi_prime(v_f)
    dloss_dz = np.matmul(np.transpose(U),delta_f)
    delta_z = dloss_dz * phi_prime(v_z)
    grad_W += np.matmul(delta_z, np.transpose(x))
    grad_b += delta_z * 1
    grad_U += np.matmul(delta_f, np.transpose(z))
    grad_c += delta_f * 1
    # Minibatch SGD
    if (i+1)%batch_size==0:
      W = W - eta * grad_W/batch_size # Averaged
      b = b - eta * grad_b/batch_size # gradients
      U = U - eta * grad_U/batch_size
      c = c - eta * grad_c/batch_size
      grad_W = 0*W # Zeroing out the gradients
      grad_b = 0*b
      grad_U = 0*U
      grad_c = 0*c
  Y_predicted = phi(np.matmul(U,phi(np.matmul(W,X)+b))+c)
  risk[epoch] = np.sum((Y-Y_predicted)**2)/N
  if epoch > 1:
    if risk[epoch]>risk[epoch-1]:
      eta = eta*0.9
plt.plot(risk)
plt.show()
# Visualization
fig = plt.figure(figsize = (7, 10))
ax = plt.axes(projection ="3d")
ax.scatter3D(x1, x2, Y_predicted, color = "green")
ax.set_xlabel('x1'); ax.set_ylabel('x2'); ax.set_zlabel('y')
ax.zaxis._axinfo['juggled'] = (2, 2, 1)
plt.show()
```






Note the faster convergence and the crisper output. The convergence is noisier due to the large step sizes, but it's controlled thanks to the adaptation of $\eta$ and the averaging inside the mini-batch.

(set batch_size=20 to see it get smoother but a bit slower.)

# L25

ECE/CS 559 Lecture 25 11/26/2024

Last time: RNNs

① **Language modeling**

Language as a sequence of symbols (tokens) $x_1, x_2, \dots, x_t, \dots$

Objective: Maximize $P(x_1, x_2, \dots, x_t, \dots)$ of data

However, just like images, this is a high-dimensional space. Any specific sequence has exponentially small probability!

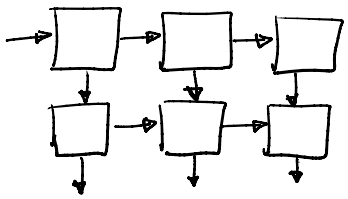
Instead: Write $P(x_1, x_2, \dots, x_t, \dots)$ as $P(x_1) \cdot P(x_2|x_1) \cdots P(x_t|x_{t-1}, \dots) \cdots$

Maximize $\sum_t \log P(x_{t+1} | x_t, x_{t-1}, \dots)$ ← this probability is not as small!

- RNNs can be used to represent these prediction probabilities. How? Often, we let $y_t$ (the output) be logits. We go from logits to probabilities via a softmax:

$$P(x_{t+1} = i \mid x_t, x_{t-1}, \dots) = \frac{\exp(y_t[i])}{\sum_{i'} \exp(y_t[i'])}$$

- Issue with RNN: they're best with multiple layers



← This layer has to finish before we can do backpropagation on this layer

Not Parallelizable

② **(Self) Attention**

- Instead of recurrence, why not directly assess what past information is relevant? (like in databases)
  - Query: What the current symbol represents
  - Key: The relevance of other sequence symbols to a query.
  - Value: What each sequence query contributes

weights $R^{k \times d}$ embedding $\in \mathbb{R}^d$ attention weights

$$Q_{t,h} = W_h^Q x_t$$
$$K_{t',h} = W_h^K x_{t'}$$ ← other

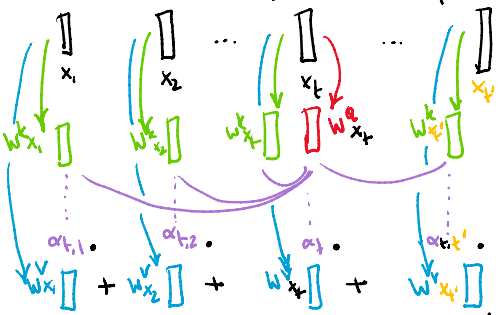
$$\alpha_{t,t',h} = \text{softmax}\left(\frac{1}{\sqrt{k}} Q_{t,h}^T K_{t',h}\right)$$

scaled, inner product

$$V_{t',h} = W_h^V x_{t'}$$

(ignore all h at first)

- Then, to generate the new state variable, we combine values based on attention $S_t = W^O \underset{h}{\text{concat}} \sum_{t'} \alpha_{t,t',h} V_{t',h}$
- We could also have multiple heads, copies w/ different weights




(issue: $T^2$ complexity)

Notes:
- Each head is given by the triple $(W^Q, W^K, W^V)$
- The weights don't change by position.
- Position info can be placed inside $x_t$ (positional embedding)
- Usually dimensions are maintained: #heads × d = k, $W^O$ square.

③ **Transformers**

- Attention only tells us where the relevant information is
- We still need to act on it. We do it using:

(a) Add and normalize (residual connection)

$$S' = \text{LayerNorm}(S + X)$$

$$\mu_t = \frac{1}{T}\sum_t S_{t,i} + X_{t,i}$$
$$\sigma_t^2 = \frac{1}{T}\sum_t (S_{t,i} + X_{t,i} - \mu_t)^2$$
$$S'_{t,i} = \gamma \frac{S_{t,i} + X_{t,i} - \mu_t}{\sqrt{\sigma_t^2 + \varepsilon}} + \beta$$

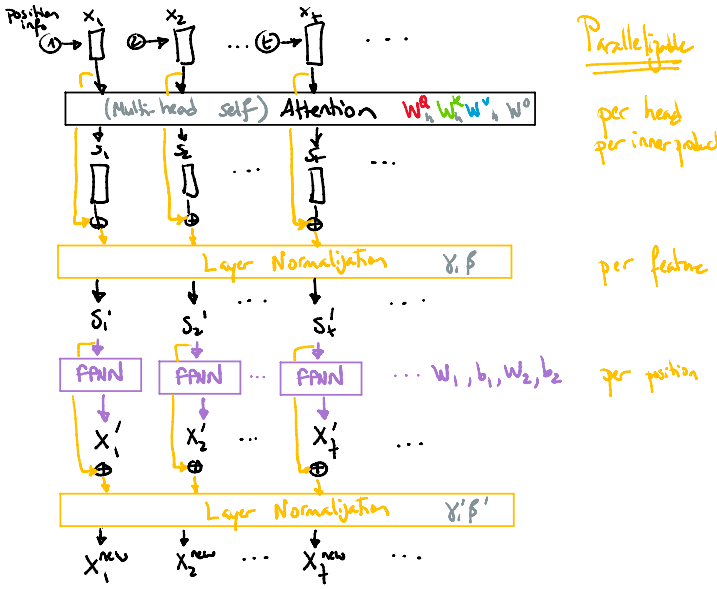
learnable, learnable

(b) 2-Layer Feedforward Neural Network

$$X'_t = W^2 \text{Relu}(W^1 S'_t + b^1) + b^2$$

(c) Add and normalize (again)

$$X^{new} = \text{LayerNorm}(S' + X')$$




Parallelizable

per head per inner product

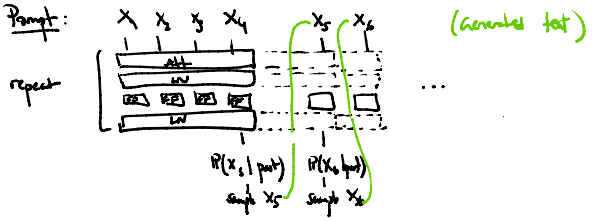
per feature

per position

## (4) Using a transformer:

### (a) Decoder-only architecture (ChatGPT)

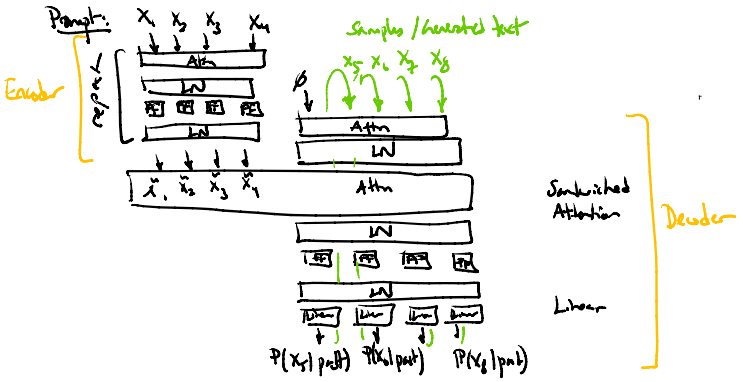
- After the prompt, we past outputs as inputs.
- Attention restricted to past outputs (since we generate forward)




- Variations: Beam Search of most likely generation:
  - Maintain k sequences, sample m (expand to km), prune back to top k

### (b) Encoder-Decoder Architectures

- The outputs of an encoder transformer become the inputs of a decoder transformer, usually in sandwiched layer:



# L24

# Recurrent Neural Networks (1)

Erdem Koyuncu

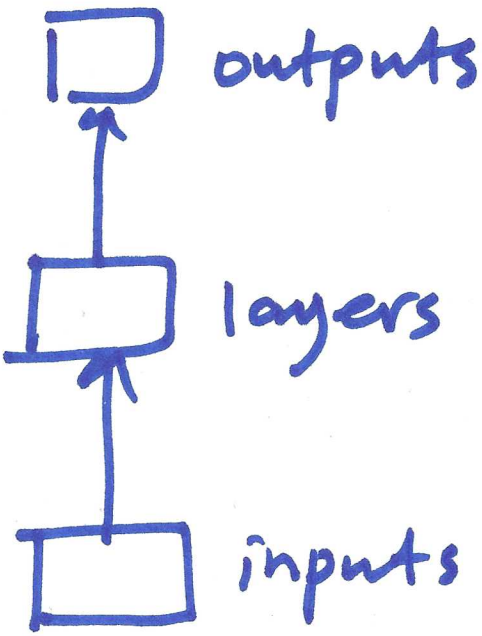
RNNs are used to process sequences (a sequence of images, words, letters, etc) in a way that takes into account the dependencies between the individual elements (or, the memory) of the sequence.

In regular (feedforward) neural networks, we provide an input to the network and observe the output after possibly many hidden layers. There is no feedback of the outputs back to the input.

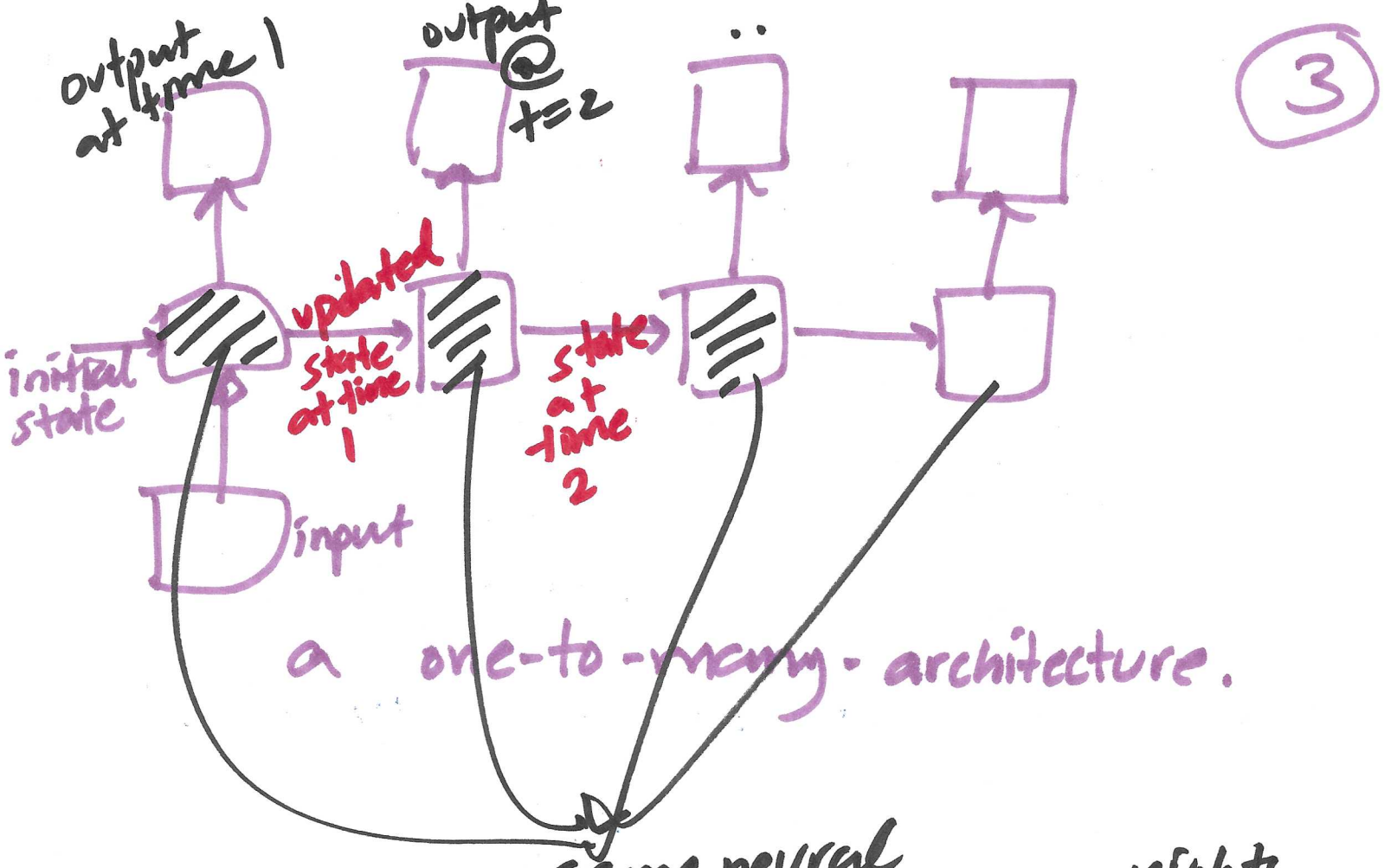
In RNNs, in general, we feed the input sequence one element at a time. Each input updates an internal state of the neural network,

which also serves as the RNN's memory. The output at any given time depends on the input at that time as well as the memory. One can imagine that the network feeds the internal states back to the network. (2)

Ordinary (feedforward) NN:




a one-to-one architecture.




3

Ex: image description/captioning

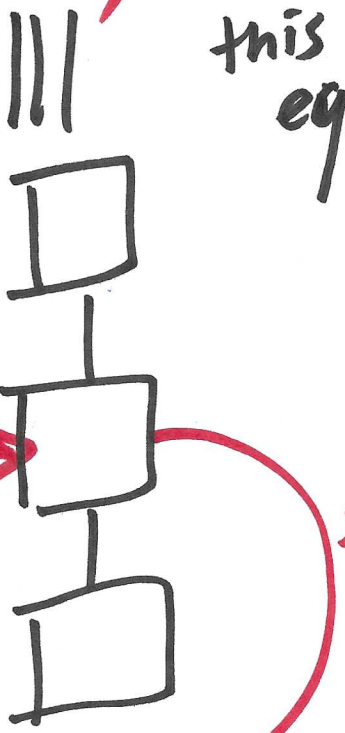
Input can be an image:

Output can be a sequence of words describing the image

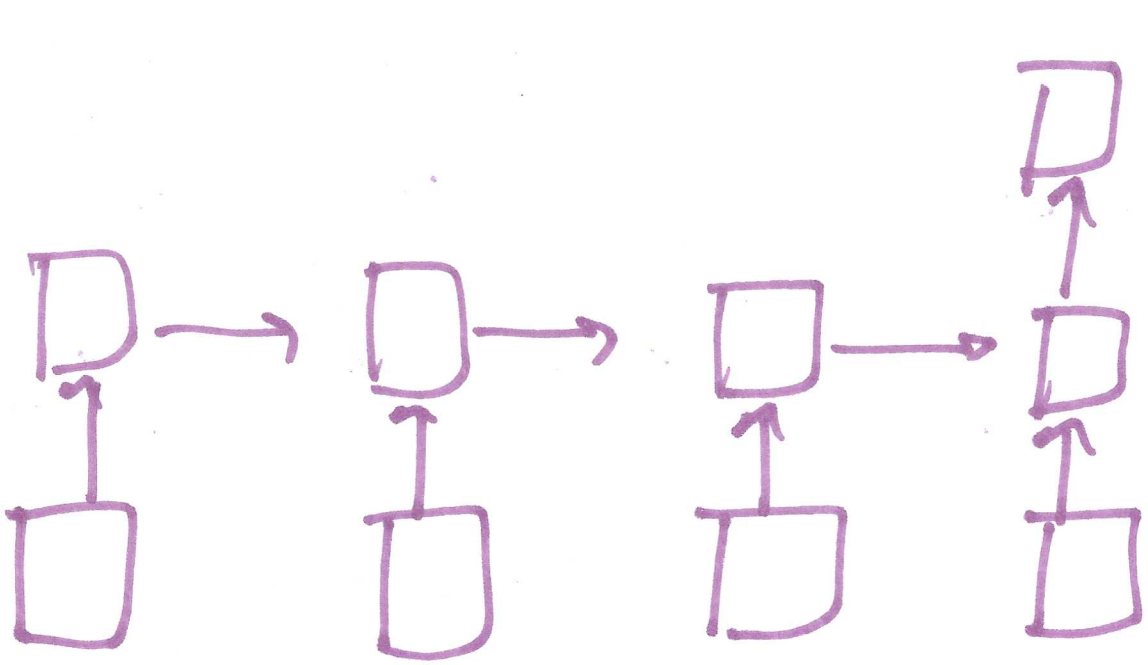
"person holding an apple"

outputs at different times will be different words.

this is equivalent to




unrolled 3 times.

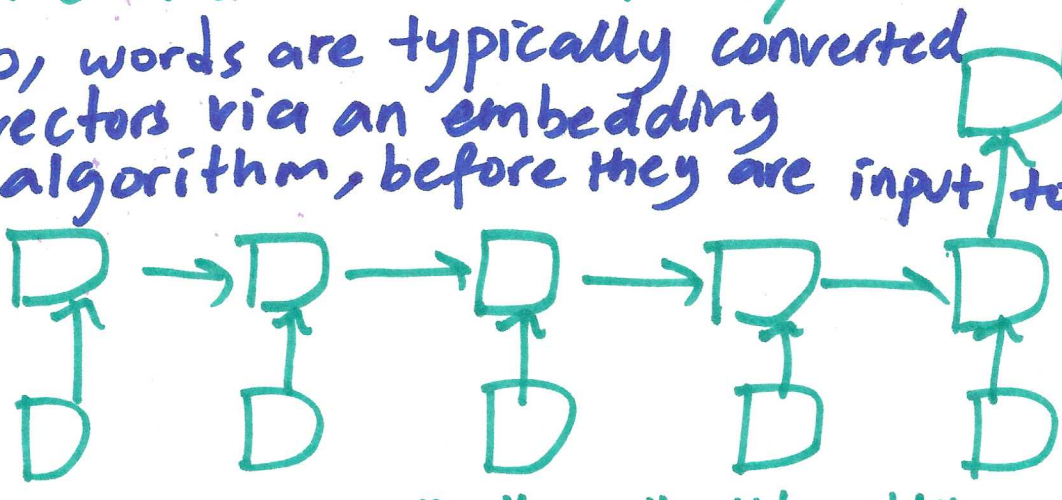


many-to-one architecture.

Ex: sentiment analysis
Input is a sequence of words
Output is whether that describes a good or bad sentiment (or any other class that we are interested in)

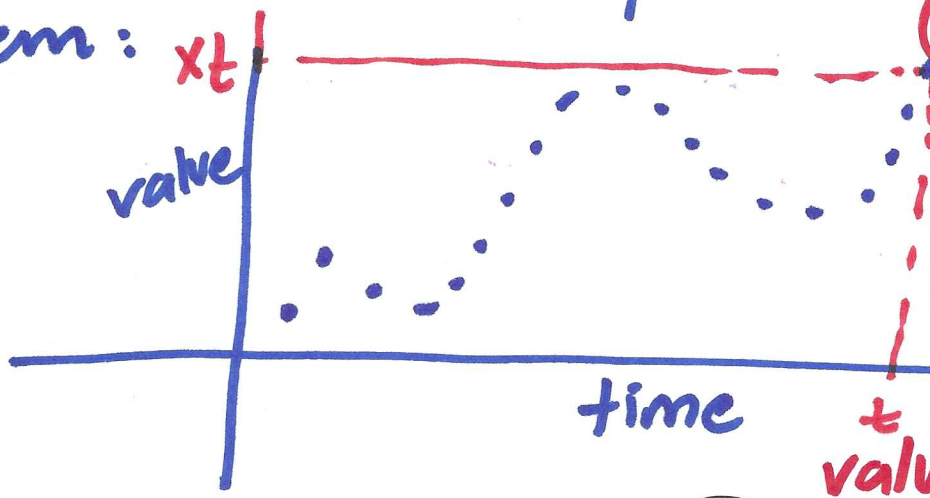
In most of these problems we will need an "END of sequence" word EOS.

Also, words are typically converted to vectors via an embedding algorithm, before they are input to the RNN.

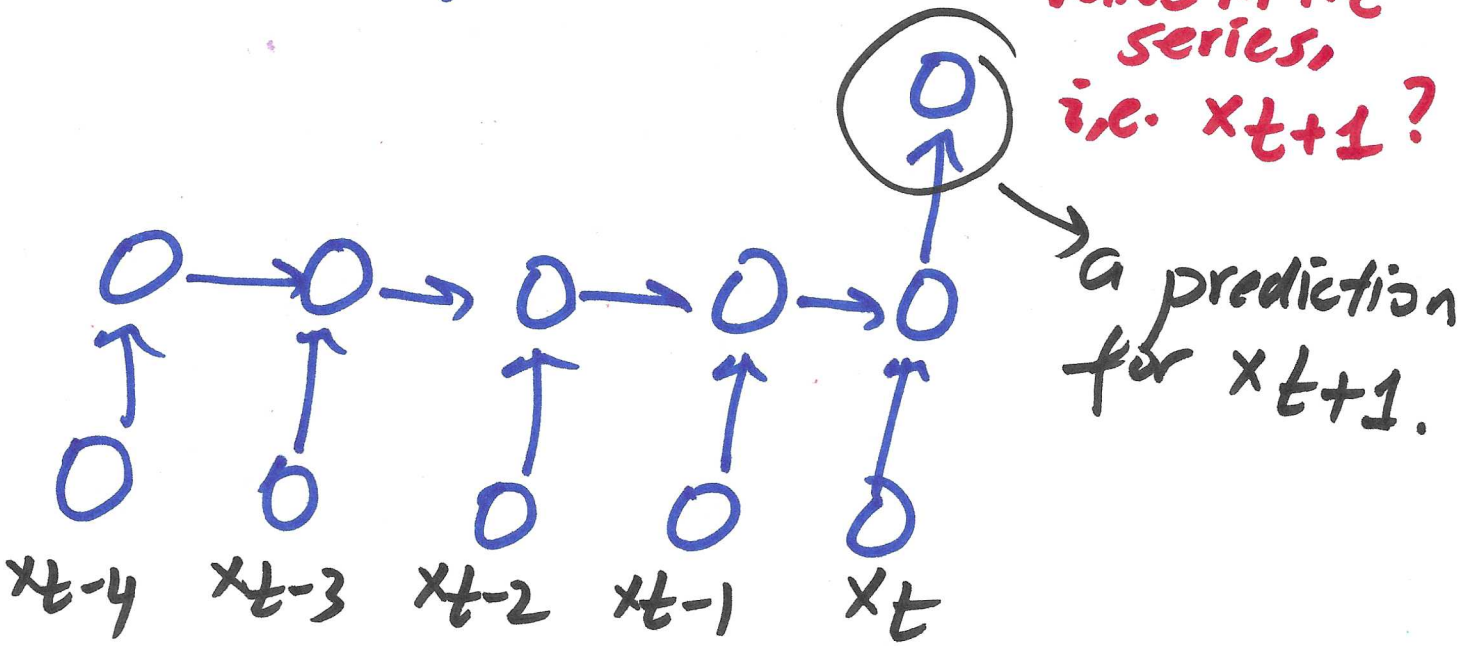



So, we are not inputting "words" of course but their embedded versions.

Another application of the many to one architecture could be a general time-series prediction problem:



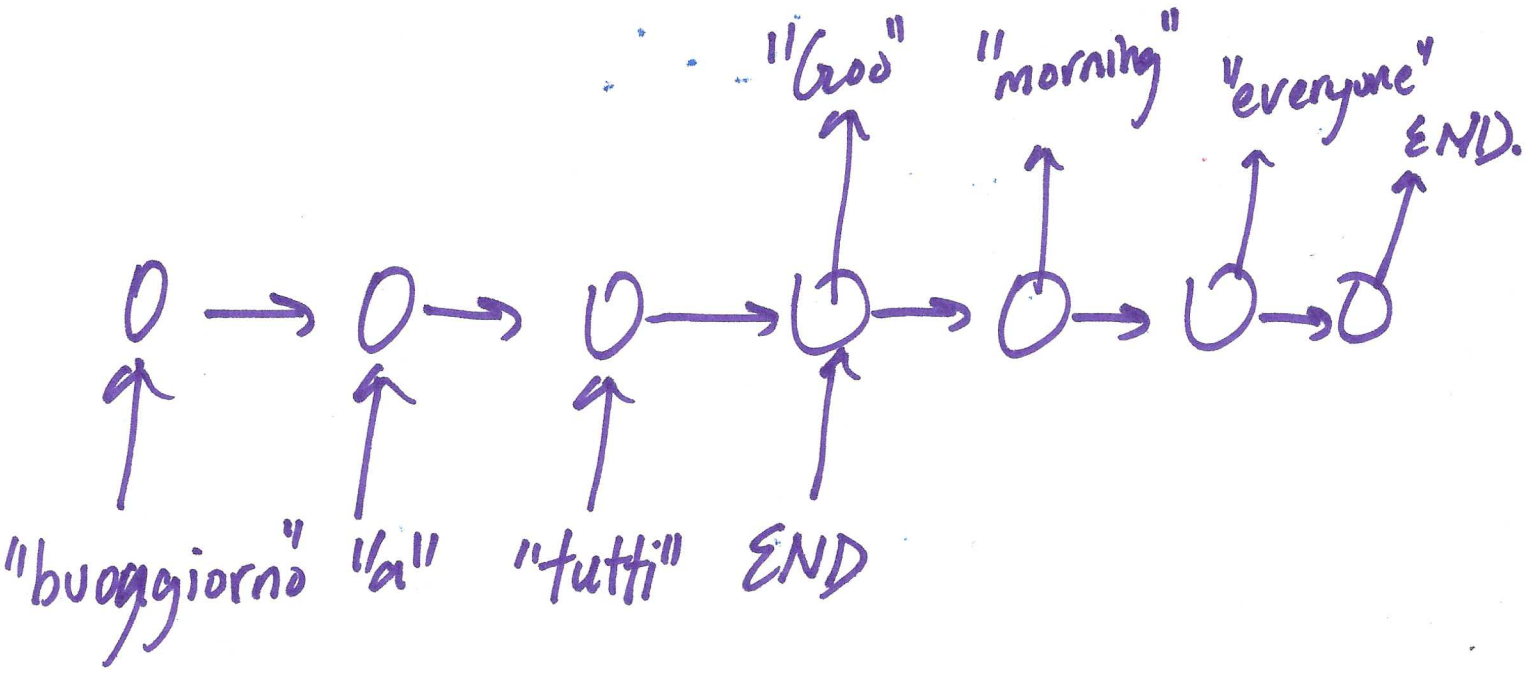

what is the next value in the series, i.e. $x_{t+1}$?



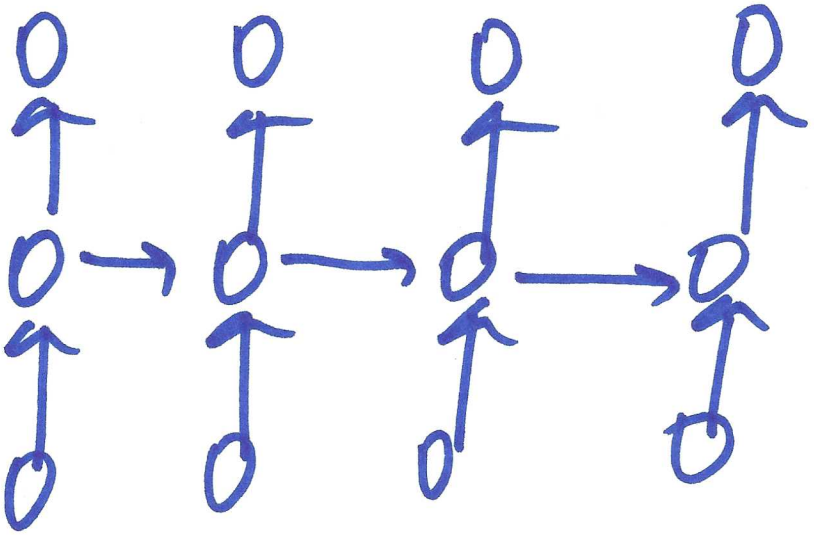

a prediction for $x_{t+1}$.

# Many-to-many architecture. (6)

## Example: Machine translation




## Synch. many-to-many architecture.



e.g. frame-by-frame video classification. etc.

many other variants...

# Simple RNN architecture (7)

$x_t$: Input at time $t \in \mathbb{R}^{d \times 1}$

$s_t$: State at time $t \in \mathbb{R}^{N \times 1}$

$y_t$: Output at time $t \in \mathbb{R}^{d \times 1}$.

$$s_t = \tanh(Ux_t + Ws_{t-1})$$

$s_{t-1}$: recurrence

$W, U \in \mathbb{R}^{N \times d}$ are trainable weights.

$$y_t = Vs_t$$

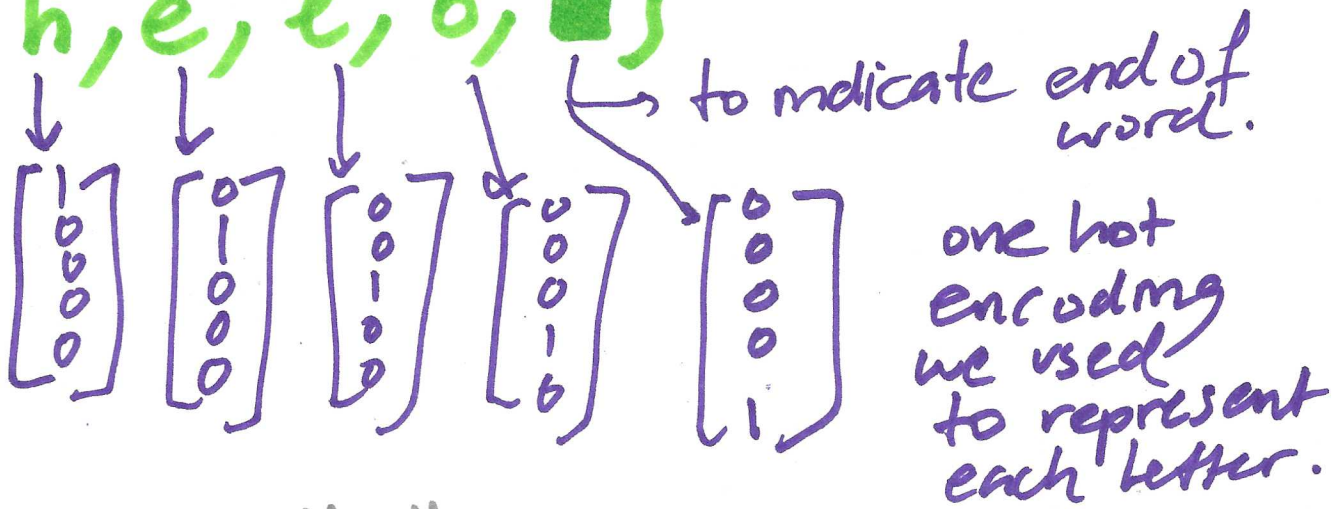
(one can also do sigmoid, hard decisions (step func.) etc.

e.g. $y_t = \sigma(Vs_t)$ where $\sigma$ is the sigmoid act. function.

# Ex: An RNN trained to generate words over the alphabet



$\{h, e, l, o, \blacksquare\}$




E.g: $x_1 =$ "e", $x_2 =$ "e"

(First two letters of the word are "ee" what will come after is likely "l", indicating eel (a certain kind of fish))

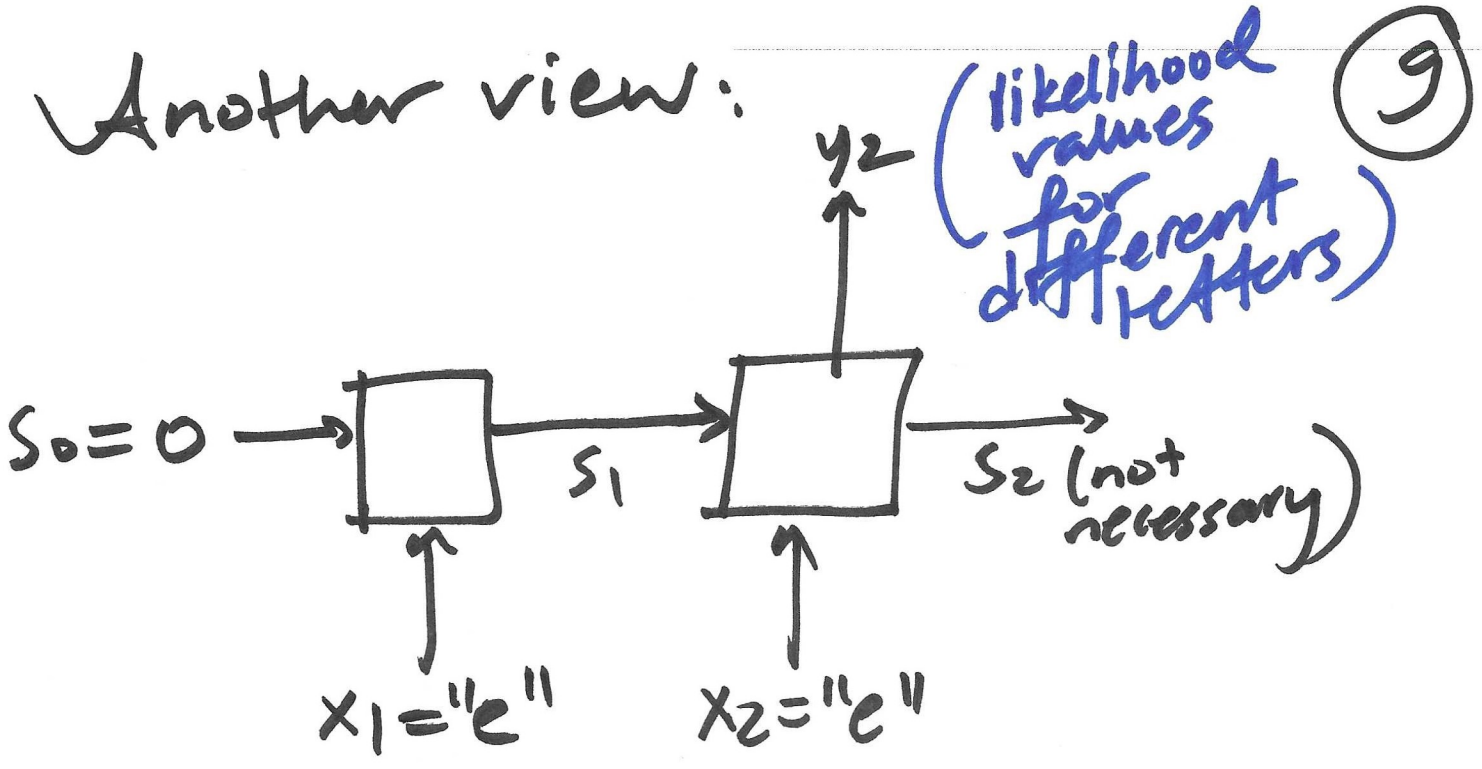
We are interested in predicting what comes after "ee", which we determine through $y_2$.

$$y_2 = \sigma(Vs_2) = \sigma(V \tanh(Ux_2 + Ws_1))$$

$$s_1 = \tanh(Ux_1 + Ws_0)$$

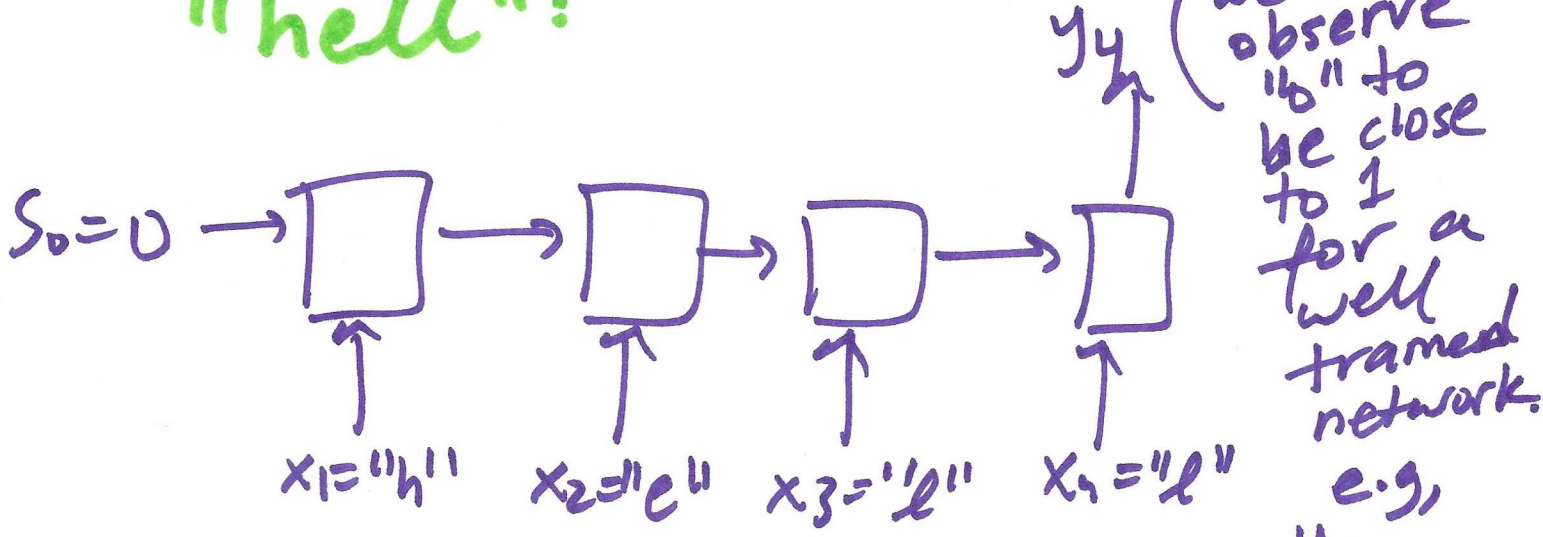
$s_0 =$ some fixed 5-dim vector (e.g. all zero vector).

Another view:




9

Same network for initial sequence "hell":

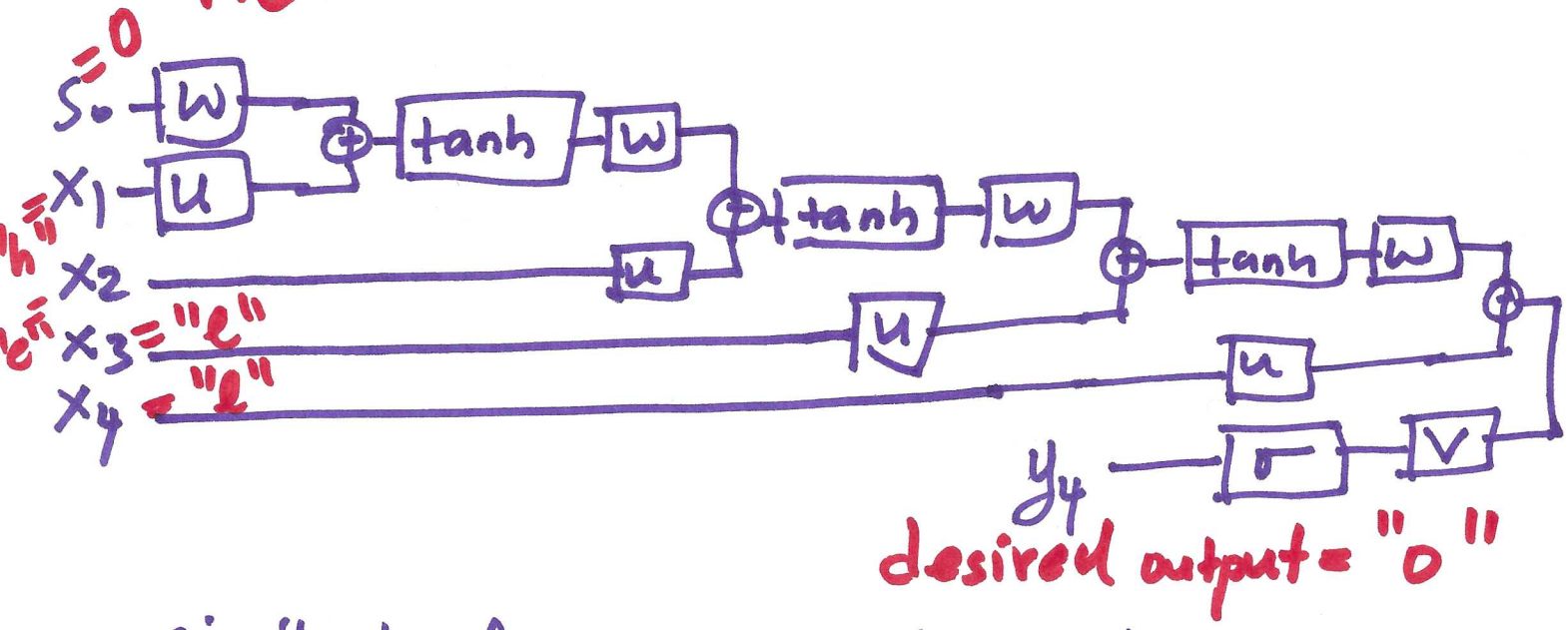



Note that "o" = $[0, 0, 0, 1, 0]$

# Training RNNs



View each example above as one instance of an ordinary feedforward NN. For example, for the "hello" prediction, the network is:




Similarly for any such ~~desired~~ inputs - desired output pairs the RNN can be unrolled any number of times and trained using standard backpropagation.

The resulting algorithm is referred to as backpropagation through time (BPTT).

# Long-Short Term Memory

(11)

As we consider longer sequences, we face the vanishing & exploding gradient problems in the standard RNN structure.

LSTMs alleviate (though not entirely solve) the vanishing & exploding $\nabla$ problems of RNNs.

We simply replace the basic recurrence block with another block. Define

$x_t$: Input at time $t$

$y_t$: Output at time $t$

$s_t$: State at time $t$

As usual. LSTMs define several new "gates":

"Forget" Gate: (how much of the previous state to forget)

$$f_t = \sigma(U^f x_t + W^f y_{t-1} + b^f)$$

Input Gate: (how much of the new candidate state to keep)

$$i_t = \sigma(U^i x_t + W^i y_{t-1} + b^i)$$

Output Gate: (how much of the calculated output to expose)

$$o_t = \sigma(U^o x_t + W^o y_{t-1} + b^o)$$

New candidate state:

$$g = \tanh(U^g x_t + W^g y_{t-1} + b^g)$$

New state: → Hadamard (element-wise) product

$$s_t = s_{t-1} \odot f_t + g \odot i_t$$

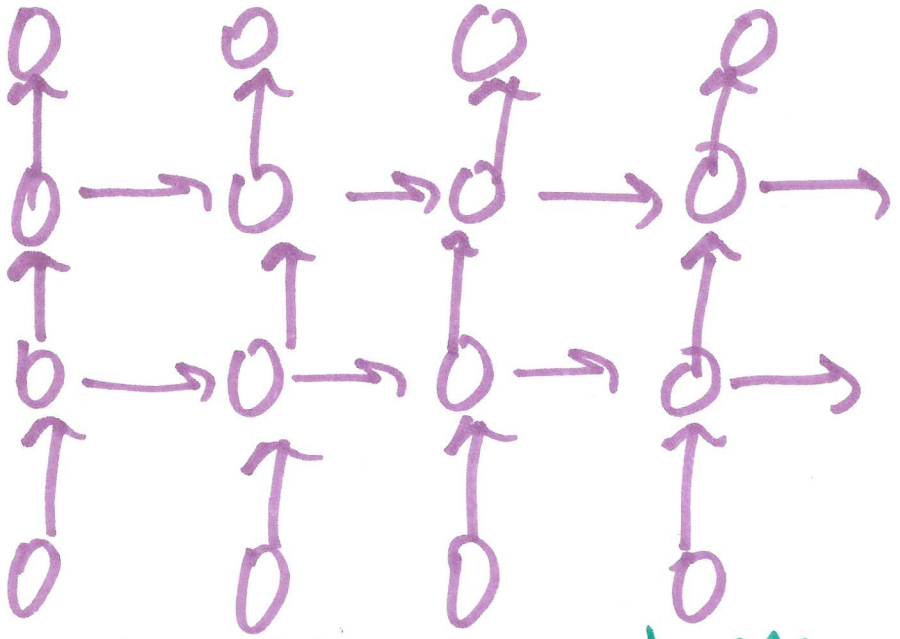
output:

$$y_t = \tanh(s_t) \odot o_t$$

- All weights, biases are trainable.
- Peephole LSTMs include a trainable weight × previous state to the induced local fields.

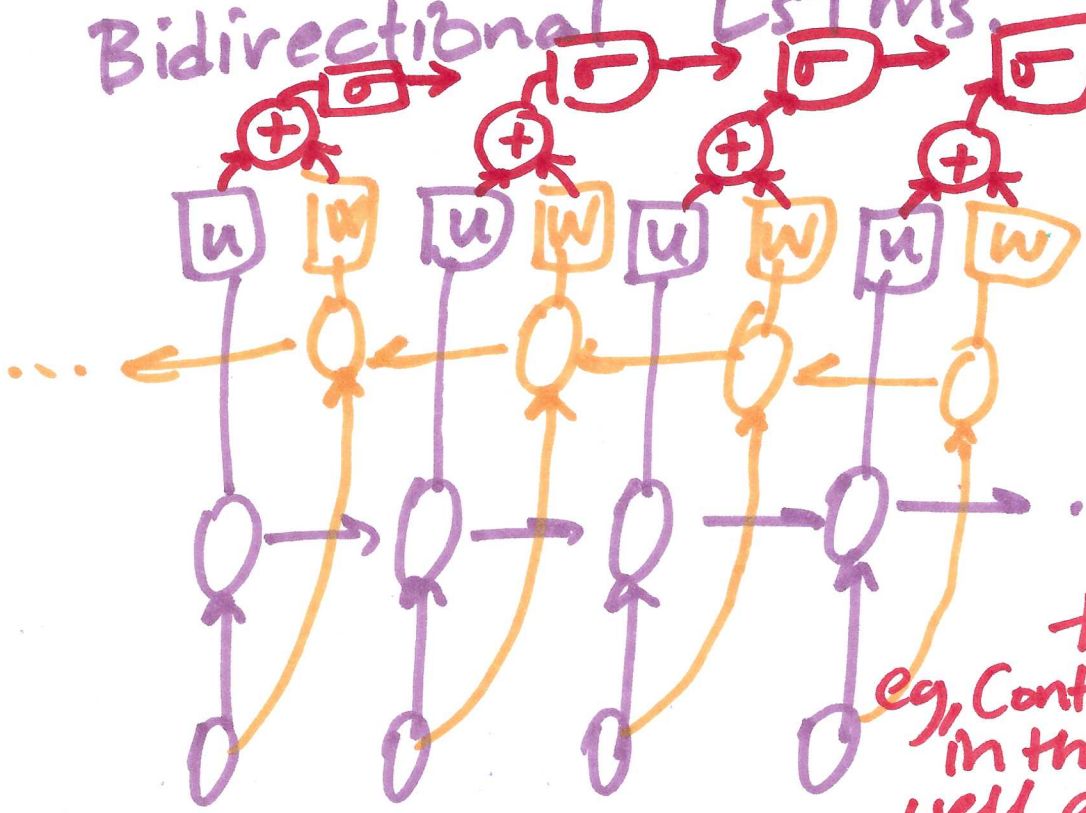
# Stacked LSTMs






* motivation: capturing more complex features (as in deep vs shallow NNs).
* Not much gains to have >3 stacks.

# Bidirectional LSTMs




- Information flows in both negative & positive time directions.
- Useful in machine translation.

eg, Context can be hidden in the beginning as well as at the end of a sentence.

# L22&23

ECE/CS 559 Lecture 23 11/19/2024

Last time: Hierarchical Autoencoders

## ① Generative-Adversarial Networks (GANs)

Motivation: In both density estimation and auto-encoders, we use likelihoods / ELBO losses:
- These are problematic in high dimensions, where the sparsity of the data makes it hard to be accurate.
- They evaluate each data point individually, instead of seeing whether the generation overall is good.


- Intuition: If a very good discriminator cannot distinguish actual images from those made by a generator, then the generator is quite good. Note:
  - No $P_X$, instead the discriminator judges images (perceptual)
  - The discriminator judges the overall (not individual) performance

Model
- Let $S$ be a coin toss: $P_S(s) = \begin{cases} \frac{1}{2} & s=H \text{ (actual)} \\ \frac{1}{2} & s=-1 \text{ (fake)} \end{cases}$
- The discriminator produces $P_{disc}(s \mid x)$, judging probability of actual/fake
- The generator samples $x$ from $P_{gen}(x)$, by sampling $z$ then $x=G(z)$
- The discriminator sees $x$ from data, $P_{pop}$, if $s=+1$, or generator if $s=-1$

$$P_X(x) = P_S(+1)\cdot P_{pop}(x) + P_S(-1)\cdot P_{gen}(x)$$
(seen by discriminator) $\quad P_{X|S}(x|+1) \quad P_{X|S}(x|-1)$

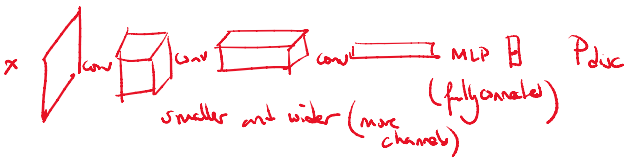
- We measure the performance of the discriminator via binary cross-entropy: $\mathbb{E}_S\left[\mathbb{E}_{x|S}\left[-\log P_{disc}(S|x)\right]\right]$
- Empirically, generate as many $z$'s as $x$'s in Data, match each $x$ to a $z$, then calculate:

$$R(W) = \frac{1}{|Data|}\sum_{x\in Data} -\frac{1}{2}\log \underbrace{P_{disc}(+1|x)}_{D(x)} - \frac{1}{2}\log \underbrace{P_{disc}(-1 \mid G(z))}_{1-D(G(z))}$$

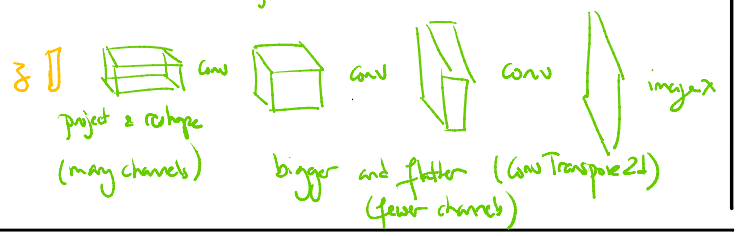
- Goal: Discriminator tries to ↓ cross-entropy and generator tries to ↑. (get better at telling apart) (get better at fooling)

$$\arg\max_{P_{gen}} \min_{P_{disc}} R(W)$$
all weights (disc & gen)

- What do the discriminator and generator look like.
- Discriminator: CNNs, standard




- Generator: CNNs, in reverse




## ② Analyzing GANs

Why do we think this would work?
- Let's assume infinite data + universality (can approximate all)

$$\arg\max_{P_{gen}} \min_{P_{disc}} \mathbb{E}_S\left[\mathbb{E}_{x|S}\left[-\log P_{disc}(S|x)\right]\right]$$

actual $P(S|x) = \frac{P(S)\cdot P(x|S)}{P(x) = \frac{1}{2}P_{pop} + \frac{1}{2}P_{gen}}$

$$\arg\max_{P_{gen}} \frac{1}{2}\mathbb{E}_{x\sim P_{pop}}\left[-\log P(+1|x)\right] + \frac{1}{2}\mathbb{E}_{x\sim P_{gen}}\left[-\log P(-1|x)\right]$$
$P(+1|x) = \frac{\frac{1}{2}\cdot P_{pop}}{\frac{1}{2}P_{pop}+\frac{1}{2}P_{gen}}$ $\quad P(-1|x) = \frac{P_{gen}}{\frac{1}{2}P_{pop}+\frac{1}{2}P_{gen}}$

KL Divergence

$$\arg\max_{P_{gen}} KL\left(P_{pop} \,\|\, \tfrac{1}{2}(P_{pop}+P_{gen})\right) + KL\left(P_{gen} \,\|\, \tfrac{1}{2}(P_{pop}+P_{gen})\right)$$

$$\arg\max_{P_{gen}} JSD(P_{pop}, P_{gen}) \Rightarrow \boxed{P_{gen}^* = P_{pop}}$$

Jensen-Shannon Divergence

## ③ Conditional Generation

- All a form of translation, e.g., image-to-image

style change $x'$ → $x$

(a) With paired images $(x', x)$ given $x'$ generate $x$

$$\underset{gen}{\operatorname{argmin}} \mathbb{E}\left[\|x - G(z,x')\|_1\right] + \underbrace{\lambda_{max}}_{disc}\underbrace{\mathbb{E}\left[-\log P_{disc}(S|x)\right]}_{R_{disc}}$$

(b) with unpaired images $\{x'\}$ $\{x\}$

Cycle GAN: 2 generators & 2 discriminators (for $x$ & $x'$)

$$\mathbb{E}\left[\|G_2(G_1(z_1,x')) - x'\|_1\right] + \mathbb{E}\left[\|G_1(G_2(z_2,x)) - x\|_1\right] \text{ (cycle loss)} + \lambda R_{disc'} + \lambda R_{disc}$$

# L19&20

ECE/CS 559 Lecture 19/20 Th 10/31; Th 11/7

Last time: Hebbian updates

## ① Generative Models

- All unsupervised learning is about understanding $P(x)$
- Generative models attempt to create more $x$:
  - They can do this by modeling $P(x)$ & sampling from it.
  - Or, they can directly model the sampling mechanism
    e.g., sample $z$ (some standard distribution), then let $x = f(z) + \text{noise}$
- Conditional generative models extend to generation via prompts: $P(x|y)$

Examples: $y$ = caption, $x$ = image (diffusion models)
$y$ = question, $x$ = answer (transformers)

## ② Density Estimation

Given Data $x$, assume i.i.d. from $P$.

Parametric methods: Let $P(x) = f(x; w)$

Maximize likelihood: $\arg\max_w P(\text{Data}) = \prod_{x \in \text{Data}} P(x)$

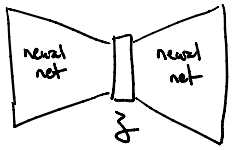
$\Leftrightarrow \arg\min_w \sum_{x \in \text{Data}} -\log P(x)$ ← log loss

Non-parametric methods: Allow # of parameters (size of $w$) to grow with data size. Example, kernel density estimation


Works well when $x$ discrete or small dimensional.
Otherwise, requires regularization, e.g. smooth densities.

## ③ Autoencoders (Variational; VAEs)

- They try to capture $P(x)$ by reducing the "essence" of $x$ to a simple representation $z$ (encoding)
  then generating $x$ from a sample of $z$ (decoding)




$f(z)$, approximates $x$ (+ noise)

Generally: $P_{enc}(z|x)$ $P_{prior}(z)$ $P_{dec}(x|z)$

- Why sampling $z \sim P_{prior}(z)$ then letting $x \sim f(z)$ work?
  How do we choose $P_{enc}$, $P_{dec}$?

## ④ Derived distributions:

- Say $x \in \mathbb{R}^1$. Let $z \sim P_z(z)$ and let $x = f(z)$.
- What is $P_x(x)$? Let's assume $P$'s are densities, $f$ monotonic)
- $F_x(x) = P(X \le x) = P(f(z) \le x) = P(z \le f^{-1}(x))$ decreasing

thus $P_x(x) = \frac{dF}{dx} = \frac{d}{dx} F_z(f^{-1}(x)) = \frac{d}{dz} F_z \cdot \frac{dz}{dx} = P_z(f^{-1}(x)) \cdot \left|\frac{df^{-1}}{dx}\right|$

Example: $P_z \sim \text{Uniform}[0,1]$ $x = -\log z$ $f^{-1}(x) = e^{-x}$

$\Rightarrow P_x(x) = \begin{cases} -1 \cdot (-e^{-x}) & e^{-x} \in (0,1) \\ 0 & \text{otherwise} \end{cases} = \begin{cases} e^{-x} & x \ge 0 \\ 0 & x < 0 \end{cases}$

- Can create any new random variable from standard random variables (e.g. uniform, Gaussian, etc.)

## ⑤ From maximum likelihood to ELBO

How do we train an autoencoder?

Idea 1: Fix $P_{prior}(z)$ and only train the decoder $P_{dec}(x|z)$

by minimizing $\sum_{x \in \text{Data}, z \sim P_{prior}} -\log P_{dec}(x|z)$

Issue: Creates disconnect between $x$ & $z$ (close $x$'s $\not\Rightarrow$ close $z$'s)

Idea 2: Fix $P_{enc}(z|x)$ and train $P_{prior}(z)$ & $P_{dec}(x|z)$

by minimizing $\sum_{x \in \text{Data}, z \sim P_{enc}(\cdot|x)} -\log P_{prior}(z) \cdot P_{dec}(x|z)$

approximates $\mathbb{E}_x \mathbb{E}_{z \sim P_{enc}|x} -\log P_{prior}(z) P_{dec}(x|z)$ $\Leftrightarrow$

$\mathbb{E}_x[\log \frac{1}{P_{gen}(x)}] + \mathbb{E}_x[\mathbb{E}_{z \sim P_{enc}|x}[\log \frac{1}{P_{gen}(z|x)}]] - \mathbb{E}_x[\mathbb{E}_{z \sim P_{enc}|x}[\log \frac{1}{P_{enc}(z|x)}]]$

implicit $P_{gen}(x)$ $P_{gen}(z|x)$

min at $P_{gen}(x) = P_x(x)$ min at $P_{gen}(z|x) = P_{enc}(z|x)$ $-$ Entropy (Penc)

Issue: If the neural net of the decoder cannot easily "invert" the choice of $P_{enc}$, it won't work well.

Idea 3 Train $P_{enc}$ and $P_{dec}$ jointly.

Issue: $P_{enc}$ in the loss is just a sampling distribution.

Hard to optimize: bad local minima / $P_{enc}$ can degenerate by colluding w/ $P_{dec}$

Idea 4: Include $P_{enc}$ in the loss: $\mathbb{E}\left[-\log \frac{P_{prior}(z) P_{dec}(x|z)}{P_{enc}(z|x)}\right]$

"max-entropy" effect

Evidence Lower Bound (ELBO) Loss $< \log P(x)$

(c) Example: Gaussian Variational Auto encoder

$p_{prior}(z) \propto \exp\left(-\frac{\|z\|^2}{2}\right)$ $\quad \mathcal{N}(0, I)$

$p_{encoder}(z|x) \propto \exp\left(-\frac{\|z - g(x;w)\|^2}{2}\right)$ $\quad \mathcal{N}(g(x;w), I)$

$p_{decoder}(x|z) \propto \exp\left(-\frac{\|x - f(z;w)\|^2}{2}\right)$ $\quad \mathcal{N}(f(z;w), I)$

ELBO has explicit form: $\mathbb{E}\left[\frac{\|z\|^2}{2} - \frac{\|z - g(x;w)\|^2}{2}\right] = \frac{1}{2}\mathbb{E}\left[\|g(x;w)\|^2\right]$

(minimize representation norm)

$+ \frac{1}{2}\mathbb{E}\left[\|x - f(\underbrace{g(x;w) + N}_{\text{encode}}; w)\|^2\right]$ $\quad N \sim \mathcal{N}(0, I)$

(minimize reconstruction error) — decode

This looks almost like kmeans and it's not a coincidence! The similarity is closer if we quantize $z$.

(4) Hierarchical VAEs

- Instead of one representation stage, have multiple:

$x \to z_1 \to z_2 \to z_3 \to z_2 \to z_1 \to x$

encode | decode

- Why? Allows for gradual tweaks, making it easier to train.

Example: Diffusion models use encoders that just add noise.

ELBO: $\log \frac{p_{prior}(z_3)\, p_{dec}(z_2|z_3)\, p_{dec}(z_1|z_2)\, p_{dec}(x|z_1)}{p_{enc}(z_3|z_2)\, p_{enc}(z_2|z_1)\, p_{enc}(z_1|x)}$

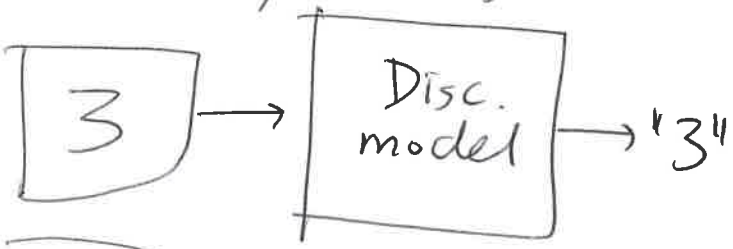
# ECE/CS 559 - Neural Networks.

## Generative Models.

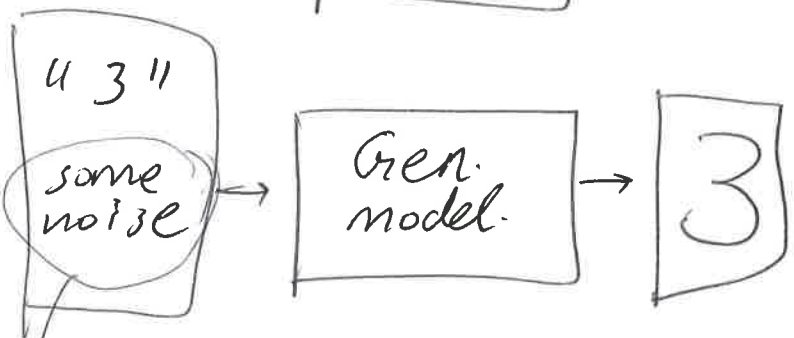
(1)

* Two main approaches in machine learning: generative vs. discriminative approach.
* Observation: $X$, Target: $Y$ (e.g. an input image) (e.g. a class label).
* A discriminative model can provide $P(Y|X=x)$ (e.g. given the input, what are the likelihoods of different class labels).
* Whereas in a generative model, we are interested in finding $P(X|Y=y)$ Given label, generate some samples belonging to that label (kind of an inverse problem):

Discriminative model:



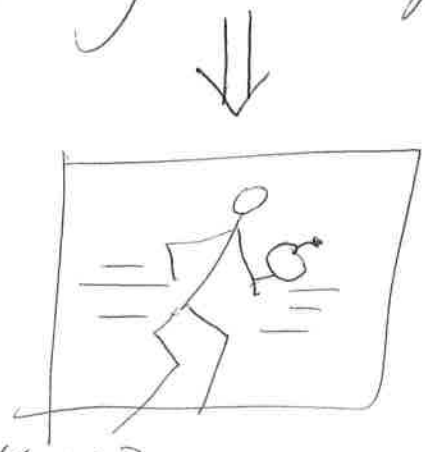

Generative model




* Noise could be considered as a "seed". Different noises would generate different images of 3's.

# Applications of generative models (2)

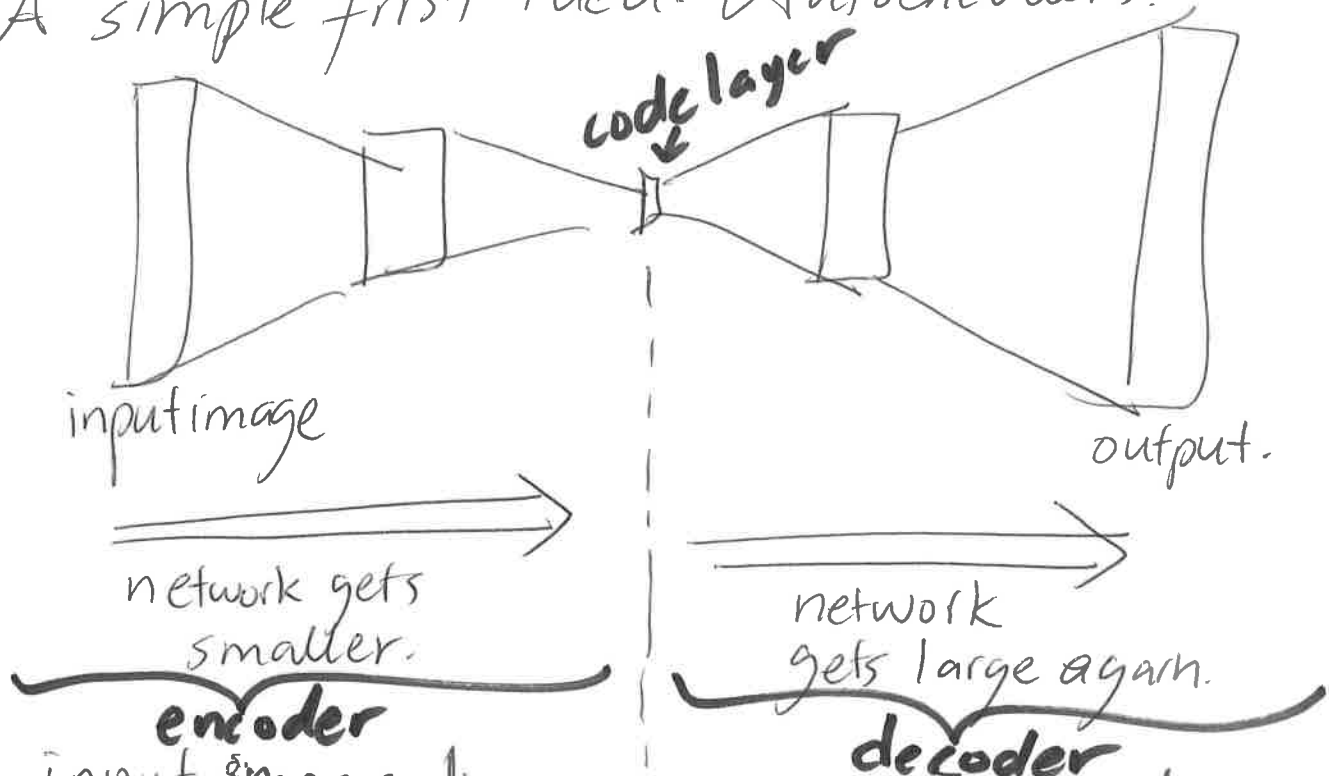
- Low to high resolution image synthesis.
- Text to image translation
  "A guy running with an apple on his hand".



- Speech synthesis.
- Error correction

:

# A simple first idea: Autoencoders.




input image dims = output image dims.

IDEA: Desired output for a given input X

=

Input X.

The network learns to compress/encode X to a very-low-dimensional representation (through the encoder layers) and then reconsruct the original X through the decoder layers.

After training, one can use the decoder part of the network as a "generator".

# Generative Adversarial Networks. (4)

Introduced by Goodfellow et al 2014.

* **Two main ideas:**

A discriminator network D.

A generator network G.

Ideally $D(x) = \begin{cases} 1, & \text{if } x \text{ is a real image} \\ 0, & \text{if } x \text{ is a fake image provided by e.g. the generator network.} \end{cases}$

In general, D(x) is the likelihood of an image being real.

G takes noise z as an input and generates G(z), a generated image.

* Define the objective

$$\min_G \max_D \mathbb{E}\left[\log D(x) + \log(1 - D(G(z)))\right]$$

* Theorem: The solution satisfies $f_{G(z)}(x) = f_X(x)$.

($f_X$: probability density)

Hence, the output of the generator matches the input data distribution.

→ Proof (Sketch): ① For a fixed G, find the optimal discriminator.

② Optimize over G.

# How to train GANs?

5.

1. Generate data samples $x_1, \dots, x_m$
2. Generate noise samples $z_1, \dots, z_m$.
3. Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left( \log D(x_i) + \log(1 - D(G(z_i))) \right)$$

$\theta_d$ → discriminator parameters.

4. Generate new noise samples $z_1, \dots, z_m$
5. Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log(1 - D(G(z_i))).$$

$\theta_g$ → generator parameters

6. Goto (1) until convergence.

At convergence, one arrives at a solution where $p_{G(Z)}(x) = p_X(x)$ and $D(x) = \frac{1}{2}$ $\forall x$. (all images are equally likely to be real or fake as the generator is perfect)

(6) Taken from Goodfellow et al.'s paper.

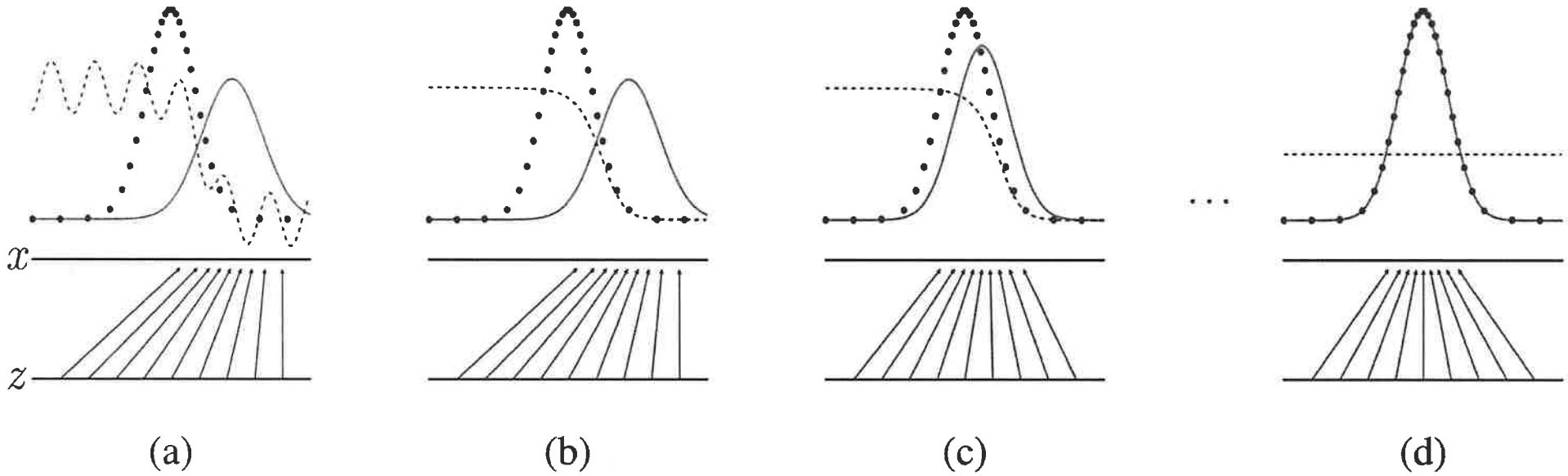



Figure 1: Generative adversarial nets are trained by simultaneously updating the discriminative distribution ($D$, blue, dashed line) so that it discriminates between samples from the data generating distribution (black, dotted line) $p_{\boldsymbol{x}}$ from those of the generative distribution $p_g$ (G) (green, solid line). The lower horizontal line is the domain from which $\boldsymbol{z}$ is sampled, in this case uniformly. The horizontal line above is part of the domain of $\boldsymbol{x}$. The upward arrows show how the mapping $\boldsymbol{x} = G(\boldsymbol{z})$ imposes the non-uniform distribution $p_g$ on transformed samples. $G$ contracts in regions of high density and expands in regions of low density of $p_g$. (a) Consider an adversarial pair near convergence: $p_g$ is similar to $p_{\text{data}}$ and $D$ is a partially accurate classifier. (b) In the inner loop of the algorithm $D$ is trained to discriminate samples from data, converging to $D^*(\boldsymbol{x}) = \frac{p_{\text{data}}(\boldsymbol{x})}{p_{\text{data}}(\boldsymbol{x}) + p_g(\boldsymbol{x})}$. (c) After an update to $G$, gradient of $D$ has guided $G(\boldsymbol{z})$ to flow to regions that are more likely to be classified as data. (d) After several steps of training, if $G$ and $D$ have enough capacity, they will reach a point at which both cannot improve because $p_g = p_{\text{data}}$. The discriminator is unable to differentiate between the two distributions, i.e. $D(\boldsymbol{x}) = \frac{1}{2}$.

# Conditional GANs.

(7)

GANs cannot natively generate images for a given label. E.g. "Generate an image of a 3". For this purpose, we can use a conditional GAN. All that has to be done is to feed the class label (e.g. as a one-hot-encoded vector) to the generator as well as the discriminator during training (and, of course, during generation). The class label thus becomes an extra input to both D and G.

# L17&18

# ECE/CS 559 - Neural Networks Lecture Notes: Unsupervised Learning

Erdem Koyuncu

This is also referred to as self-organization. We similarly have a training set. But, for each member of the training set, we do not have a correct output information. The NN is required to organize itself, by possibly detecting the similarities between the patterns (statistical properties) and arranges its outputs accordingly. A typical case is clustering, which may be considered as a form of classification.

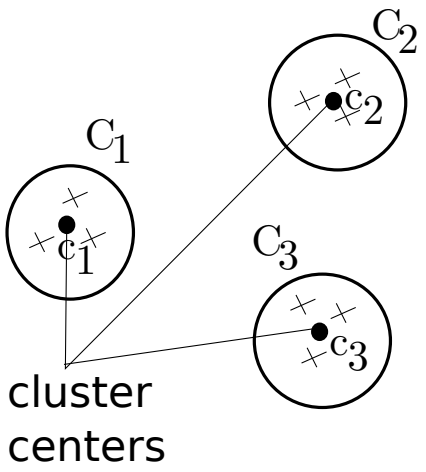



Figure 1: Clustering

- Suppose that the inputs will form clusters $\mathcal{C}_i$ with cluster centers $\mathbf{c}_i$.
- If we know the cluster centers, we can then distinguish the clusters by distance: Given $\mathbf{x} \in \mathcal{S}$, we have $\|\mathbf{x} - \mathbf{c}_i\| < \|\mathbf{x} - \mathbf{c}_j\|, \forall j \neq i \implies \mathbf{x} \in \mathcal{C}_i$.
- Hence one way of looking at unsupervised learning is to find appropriate cluster centers for each clusters, hence detect the clusters in the given training set.
- Here, the cluster centers $\mathbf{c}_i$ are also known as the quantization vectors.
- Unsupervised learning tries to minimize quantization error by finding appropriate cluster centers:
- If we have $m$ clusters, the quantization error given $\mathbf{c}_i$, $i = 1, \ldots, m$ is given by

$$E = \sum_{\mathbf{x} \in \mathcal{S}} \min_i \|\mathbf{x} - \mathbf{c}_i\|^2$$

- The goal is then to minimize $E$, i.e. finding the appropriate cluster centers (quantization points) $\mathbf{c}_i$ such that $E$ is minimized.

# 1 $k$-means algorithm

- In general, the problem of minimizing $E$ can be solved iteratively. Let $\mathcal{V}_i = \{\mathbf{x} : \|\mathbf{x} - \mathbf{c}_i\| \leq \|\mathbf{x} - \mathbf{c}_j\|\}$ denote the Voronoi cell for cluster center $i$, where ties are broken arbitrarily. Then, we have

$$E = \sum_{i=1}^{m} \sum_{\mathbf{x} \in \mathcal{V}_i} \|\mathbf{x} - \mathbf{c}_i\|^2 \tag{1}$$

  This decomposition suggests the following algorithm:

  1) Initialize cluster centers $\mathbf{c}_1, \ldots, \mathbf{c}_m$ (e.g., arbitrarily).

  2) Until convergence (of the energy $E$ or cluster centers)

  2.1) Calculate Voronoi cells $\mathcal{V}_1, \ldots, \mathcal{V}_m$ given cluster centers $\mathbf{c}_1, \ldots, \mathbf{c}_m$.

  2.2) Set $\mathbf{c}_i \leftarrow \min_{\mathbf{c}} \sum_{\mathbf{x} \in \mathcal{V}_i} \|\mathbf{x} - \mathbf{c}\|^2$, $i = 1, \ldots, m$. In other words, we update each cluster center while considering the corresponding Voronoi cells to be fixed.

  Note that $\arg\min_{\mathbf{c}} \sum_{\mathbf{x} \in \mathcal{V}_i} \|\mathbf{x} - \mathbf{c}\|^2 = \frac{1}{|\mathcal{V}_i|} \sum_{\mathbf{x} \in \mathcal{V}_i} \mathbf{x}$, i.e. the solution of the optimization problem is the geometric centroid of the voronoi region $\mathcal{V}_i$. Thus, instead of 2.2 above, I can equivalently write:

  2.2) Set $\mathbf{c}_i \leftarrow \frac{1}{|\mathcal{V}_i|} \sum_{\mathbf{x} \in \mathcal{V}_i} \mathbf{x}$, $i = 1, \ldots, m$.

  This is known as the $k$-means algorithm or the Lloyd algorithm. Each step of the algorithm provides a lower energy $E$, and thus the algorithm is guaranteed to converge in terms of the energy-function sense. The convergence of the cluster centers is another matter that we will not go into, but typically you may assume that you also have the convergence of the cluster centers.

  It is possible to reinterpret the algorithm using neural networks, as we will show soon.

# 2 Hebbian Learning

Another important case we will look at is Hebbian learning.

- This learning rule was first proposed by D. Webb in 1940's. Hebb's idea was (1949):
- When an axon of a cell A is near enough to excite a cell B and repeatedly or persistently fires it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased.
- Nodes that tend to be either both positive or both negative at the same time have strong positive weights, while those that tend to be opposite have strong negative weights.
- Mathematically, we have $\mathbf{w}_i(n+1) = \mathbf{w}_i(n) + \eta \mathbf{x} y_i(n)$, where $y_i(n)$ is the output of neuron $i$ for training sample $n$. This is also called the activity product rule.
- This type of update leads to saturation problems. For example, suppose $y_i(n) = \text{sgn}(\mathbf{x}^T \mathbf{w}_i(n))$, with the initial condition $\mathbf{w}_i(n) = \mathbf{1}$, and consider a fixed input $\mathbf{x} = 1$. We obtain the sequence $\mathbf{1}, \mathbf{2}, \mathbf{4}, \mathbf{8}$, with exponential divergence to infinity.
- So, sometimes, the update $\mathbf{w}_i(n+1) = \alpha \mathbf{w}_i(n) + \eta \mathbf{x} y_i(n)$ is used, where $\alpha < 1$. For the previous example, we obtain $1, 1+\alpha, 1+\alpha(1+\alpha), 1+\alpha(1+\alpha(1+\alpha)), \ldots$. Solution $1 + \alpha x = x$, $x = \frac{1}{1-\alpha}$ (finite).
- Sometimes, this update is also used $\mathbf{w}_i(n+1) = \mathbf{w}_i(n) + \eta(\mathbf{x} - \overline{\mathbf{x}})(y_i(n) - \overline{y_i})$. Here, $\overline{\mathbf{x}}$ and $\overline{y_i}$ are time averages of their respective quantities.
- Example: Suppose $\mathcal{C}_1 = \left\{ \mathbf{x}_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \mathbf{x}_2 = \begin{bmatrix} 0.9 \\ 1 \end{bmatrix}, \mathbf{x}_3 = \begin{bmatrix} 1 \\ 1.1 \end{bmatrix} \right\}$, and $\mathcal{C}_2 = \left\{ \mathbf{x}_4 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \mathbf{x}_5 = \begin{bmatrix} 1.1 \\ -1 \end{bmatrix}, \mathbf{x}_6 = \begin{bmatrix} 1 \\ -1.1 \end{bmatrix} \right\}$. The problem is to find a single neuron that can distinguish between the two classes. Although the example introduces the two classes separately, we assume we do not know the desired output for any given pattern. Obviously, if the desired output were given for all patterns, we could have used the perceptron training algorithm to solve the classification task.

- Here, we consider weights without any thresholding. Suppose the initial weights are $\mathbf{w}(1) = \begin{bmatrix}0.1\\0.1\end{bmatrix}$. Note that given input $\mathbf{x}_1$, the output is $\text{sgn}(\mathbf{x}_1^T\mathbf{w}(1)) = 1$. Given input $\mathbf{x}_5$ on the other hand, the output is $\text{sgn}(\mathbf{x}_5^T\mathbf{w}(1)) = 1$. However $\mathbf{x}_1$ and $\mathbf{x}_5$ were supposed to be on different classes. Hence, the initial weights cannot distinguish the two classes.
- Begin Epoch: 1. Suppose $\eta = 1$.
  - Feed $\mathbf{x}_1$, the output is $\text{sgn}(\mathbf{x}_1^T\mathbf{w}(1)) = 1$. Hence, we set $\mathbf{w}(2) = \mathbf{w}(1) + 1 \times 1 \times \mathbf{x}_1 = \begin{bmatrix}1.1\\1.1\end{bmatrix}$.
  - Feed $\mathbf{x}_2$, the output is $\text{sgn}(\mathbf{x}_2^T\mathbf{w}(2)) = 1$. Hence, we set $\mathbf{w}(3) = \mathbf{w}(2) + 1 \times 1 \times \mathbf{x}_2 = \begin{bmatrix}2\\2.1\end{bmatrix}$.
  - Feed $\mathbf{x}_3$, the output is $\text{sgn}(\mathbf{x}_3^T\mathbf{w}(3)) = 1$. Hence, we set $\mathbf{w}(4) = \mathbf{w}(3) + 1 \times 1 \times \mathbf{x}_3 = \begin{bmatrix}3\\3.2\end{bmatrix}$.
  - Feed $\mathbf{x}_4$, the output is $\text{sgn}(\mathbf{x}_4^T\mathbf{w}(4)) = -1$. Hence, we set $\mathbf{w}(5) = \mathbf{w}(4) - 1 \times 1 \times \mathbf{x}_4 = \begin{bmatrix}2\\4.2\end{bmatrix}$.
  - Feed $\mathbf{x}_5$, the output is $\text{sgn}(\mathbf{x}_5^T\mathbf{w}(4)) = -1$. Hence, we set $\mathbf{w}(6) = \mathbf{w}(5) - 1 \times 1 \times \mathbf{x}_5 = \begin{bmatrix}0.9\\5.2\end{bmatrix}$.
  - Feed $\mathbf{x}_6$, the output is $\text{sgn}(\mathbf{x}_6^T\mathbf{w}(4)) = -1$. Hence, we set $\mathbf{w}(7) = \mathbf{w}(6) - 1 \times 1 \times \mathbf{x}_6 = \begin{bmatrix}-0.1\\6.3\end{bmatrix}$.
- We can proceed with more epochs like this, but let us stop at the end of Epoch 1. It can be verified that the final weight vector $\mathbf{w}(7)$ provides an output of 1 for input patterns $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3$, and it provides an output of $-1$ for input patterns $\mathbf{x}_4, \mathbf{x}_5, \mathbf{x}_6$.

# 3 Competitive learning

- Also called the winner takes all learning rule.
- Output neurons compete around themselves to be active.
- Consider a one-layer network. Suppose we have $m$ input neurons and $n$ output neurons without any bias, and the activator function is identity. Let $\mathbf{W} \in \mathbb{R}^{n\times m}$ denote the weight matrix with the weight going to output neuron $i$ from neuron $j$ denoted by $w_{ij}$. Then, given input $\mathbf{x} \in \mathbb{R}^{m\times 1}$, the induced local fields are $\mathbf{v} = \mathbf{W}\mathbf{x} \in \mathbb{R}^{n\times 1}$ and the outputs are $\mathbf{y} = \phi(\mathbf{v}) \in \mathbb{R}^{n\times 1}$. The weights for the $i$th output neuron is then $i$th row $\mathbf{w}_i$ of $\mathbf{W}$. We may also use the extended notation.
- After this, we look at the output neuron with the largest output. Due to the monotonicity of the activation function, this neuron is the one with the largest induced local field.
- Suppose that the $i$th neuron has the largest induced local field, i.e. $v_i = \max_{j\in\{1,\dots,n\}} v_j$ according to some tie-breaking rule. Then, the $i$th neuron is declared the winning neuron, and its outputs are updated as $\mathbf{w}_i(n+1) = \mathbf{w}_i(n)+\eta(\mathbf{x}-\mathbf{w}_i(n))$. The others remain the same, i.e. $\mathbf{w}_j(n+1) = \mathbf{w}_j(n)$, $j \neq i$. Here, $\eta$ is the learning constant.
- If $\|\mathbf{w}_i\| = \|\mathbf{w}_j\|$ for every $i \neq j$, then the winning neuron coincides with the neuron whose weight vector is closest to the input $\mathbf{x}$ in terms of the Euclidean distance - prove this. So, roughly speaking: The learning rule has the effect of moving the input $\mathbf{x}$ towards the direction of the winning neuron. The winner is more likely to win next time as well.
- Usually, for the success of the algorithm, most of the time input vectors and weights are normalized.

- Suppose that some input pattern $\mathbf{x}$ is repeatedly applied, and at each time, neuron $i$ wins. We have

$$\begin{aligned}
\mathbf{w}_i(2) &= \mathbf{w}_i(1) + \eta(\mathbf{x} - \mathbf{w}_i(1)) = (1-\eta)\mathbf{w}_i(1) + \eta\mathbf{x} \\
\mathbf{w}_i(3) &= \mathbf{w}_i(2) + \eta(\mathbf{x} - \mathbf{w}_i(2)) = (1-\eta)^2\mathbf{w}_i(1) + \eta(1 + (1-\eta))\mathbf{x} \\
&\vdots \\
\mathbf{w}_i(n+1) &= (1-\eta)^n\mathbf{w}_i(1) + \eta(1 + (1-\eta) + \cdots + (1-\eta)^{n-1})\mathbf{x} \\
&= (1-\eta)^n\mathbf{w}_i(1) + (1 - (1-\eta)^n)\mathbf{x}
\end{aligned}$$

  If $|\eta| < 1$, then we have $\mathbf{w}_i(n+1) \to \mathbf{x}$ as $n \to \infty$. Hence, if we apply patterns from some cluster $\mathcal{C}$, then (eventually) we expect the same neuron to win the competition, and that the weight of the winner neuron converges to the cluster center of $\mathcal{C}$ (Typically, for this to precisely happen, the learning parameter should also go to zero as the number of epochs go to infinity. Otherwise, the winning neuron will oscillate around the cluster center of $\mathcal{C}$).

Example: Say the learning parameter is 0.5 with $\mathbf{x}_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$, $\mathbf{x}_2 = \begin{bmatrix} 0.9 \\ 1 \end{bmatrix}$, $\mathbf{x}_3 = \begin{bmatrix} 1 \\ 1.1 \end{bmatrix}$, $\mathbf{x}_4 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$, $\mathbf{x}_5 = \begin{bmatrix} 1.1 \\ -1 \end{bmatrix}$, $\mathbf{x}_6 = \begin{bmatrix} 1 \\ -1.1 \end{bmatrix}$.

Suppose $\mathbf{w}_1(1) = \begin{bmatrix} 0.8 \\ 0 \end{bmatrix}$, $\mathbf{w}_2(1) = \begin{bmatrix} 0.9 \\ 0.1 \end{bmatrix}$

$[\mathbf{w}_1(1)]^T\mathbf{x}_1 = 0.80$, $[\mathbf{w}_2(1)]^T\mathbf{x}_1 = 1.00 \implies$ 2 wins.
$[\mathbf{w}_1(1)]^T\mathbf{x}_2 = 0.72$, $[\mathbf{w}_2(1)]^T\mathbf{x}_2 = 0.91 \implies$ 2 wins.
$[\mathbf{w}_1(1)]^T\mathbf{x}_3 = 0.80$, $[\mathbf{w}_2(1)]^T\mathbf{x}_3 = 1.01 \implies$ 2 wins.
$[\mathbf{w}_1(1)]^T\mathbf{x}_4 = 0.80$, $[\mathbf{w}_2(1)]^T\mathbf{x}_4 = 0.80 \implies$ Draw.
$[\mathbf{w}_1(1)]^T\mathbf{x}_5 = 0.88$, $[\mathbf{w}_2(1)]^T\mathbf{x}_5 = 0.89 \implies$ 2 wins.
$[\mathbf{w}_1(1)]^T\mathbf{x}_6 = 0.80$, $[\mathbf{w}_2(1)]^T\mathbf{x}_6 = 0.79 \implies$ 1 wins.

Epoch - 1

- $\mathbf{x}_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$, : $[\mathbf{w}_1(1)]^T\mathbf{x}_1 = 0.80$, $[\mathbf{w}_2(1)]^T\mathbf{x}_1 = 1.00 \implies$ 2 wins. Thus, we update

$$\begin{aligned}
\mathbf{w}_1(2) &= \mathbf{w}_1(1) = \begin{bmatrix} 0.8 \\ 0 \end{bmatrix}, \\
\mathbf{w}_2(2) &= \mathbf{w}_2(1) + \alpha(\mathbf{x}_1 - \mathbf{w}_2(1)) \\
&= \begin{bmatrix} 0.9 \\ 0.1 \end{bmatrix} + 0.5\left(\begin{bmatrix} 1 \\ 1 \end{bmatrix} - \begin{bmatrix} 0.9 \\ 0.1 \end{bmatrix}\right) = \begin{bmatrix} 0.95 \\ 0.55 \end{bmatrix}.
\end{aligned}$$

- $\mathbf{x}_2 = \begin{bmatrix} 0.9 \\ 1 \end{bmatrix}$, : $[\mathbf{w}_1(2)]^T\mathbf{x}_2 = 0.72$, $[\mathbf{w}_2(2)]^T\mathbf{x}_2 = 1.405 \implies$ 2 wins. Thus, we have

$$\begin{aligned}
\mathbf{w}_1(3) &= \begin{bmatrix} 0.8 \\ 0 \end{bmatrix}, \\
\mathbf{w}_2(3) &= \begin{bmatrix} 0.925 \\ 0.775 \end{bmatrix}
\end{aligned}$$

- $\mathbf{x}_3 = \begin{bmatrix} 1 \\ 1.1 \end{bmatrix}$, : $(0.8, 1.7775)$ 2 wins. Thus, we have

$$\begin{aligned}
\mathbf{w}_1(4) &= \begin{bmatrix} 0.8 \\ 0 \end{bmatrix}, \\
\mathbf{w}_2(4) &= \begin{bmatrix} 0.9625 \\ 0.9375 \end{bmatrix}
\end{aligned}$$

- $\mathbf{x}_4 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$, : $(0.8, 0.025)$ 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 $i$th 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 $i$th 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 $R(C)$?

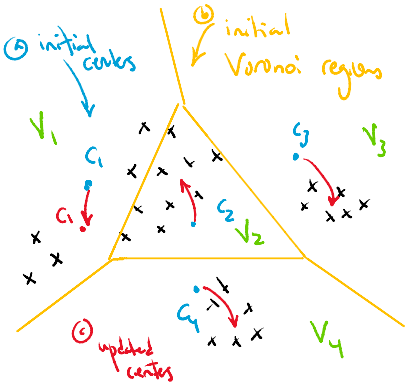
- It's possible to do gradient descent. It's non-convex!
- Or... use Lloyd's algorithm aka "the" k-means algorithm.

### 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 $C$

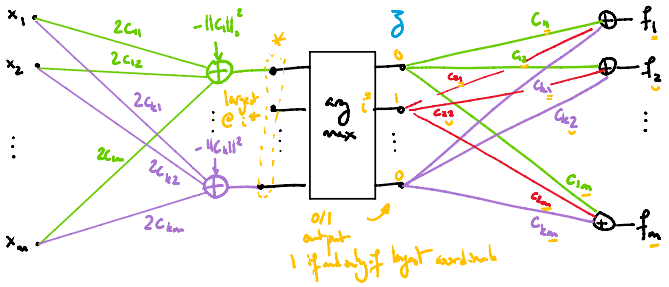



Voronoi regions partition the space according to proximity to specific centers

## 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 $f_j = c_{ij}$

$\delta_i$ ↑ 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

- $\delta$ 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:**

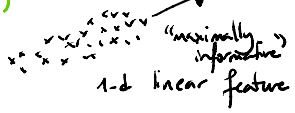
**Definition:** $w_i \leftarrow w_i + \eta g_i (x - w_i g_i)$ — difference; doesn't affect it if 0/1

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

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

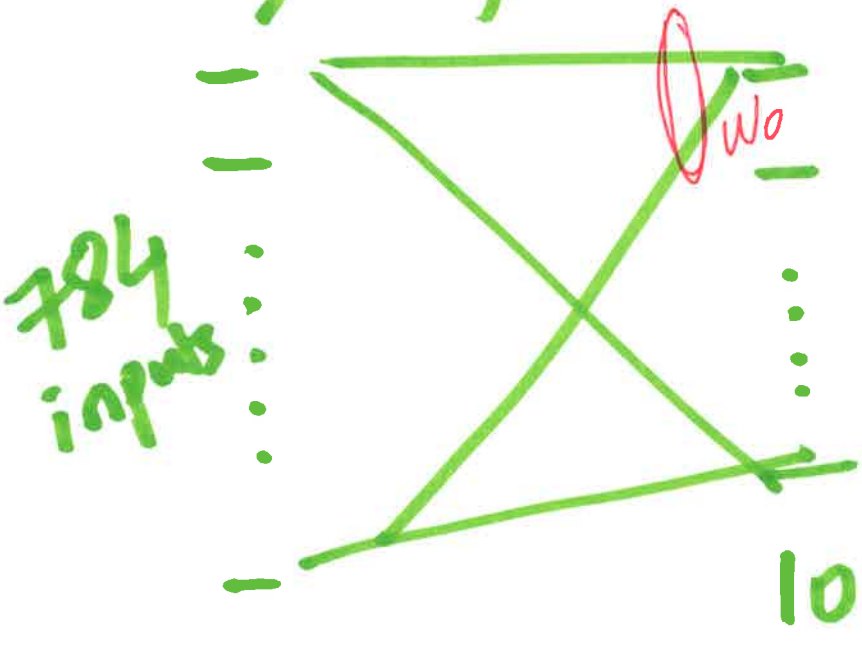
- Say $x = 1$ vector, fixed $w_i(0) = 1$ vector also initially. $\eta = 1$.
  $w_i(1) = 1 + 1 = 2$ $w_i(2) = 2 + 2 = 4$ $w_i(3) = 8 \ldots$
- If we stabilize with $\alpha < 1$:
  $w_i(1) = \alpha + 1$ $w_i(2) = \alpha(\alpha+1) + 1 = \alpha^2 + \alpha + 1$
  $w_i(3) = \alpha(\alpha^2 + \alpha + 1) +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

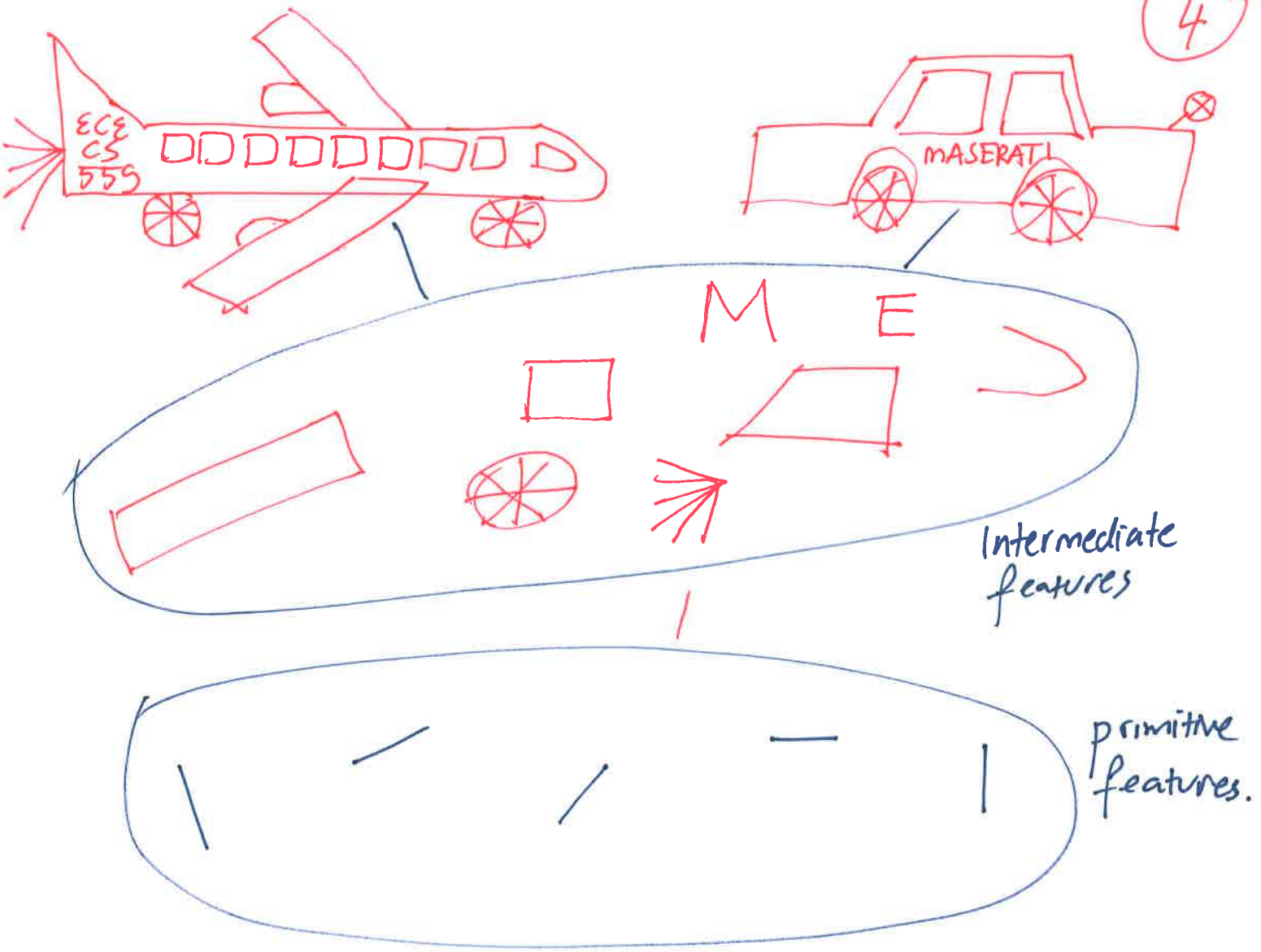
Problems: Too many parameters may result in overfitting, performance degradation, increased implementation complexity.

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 |

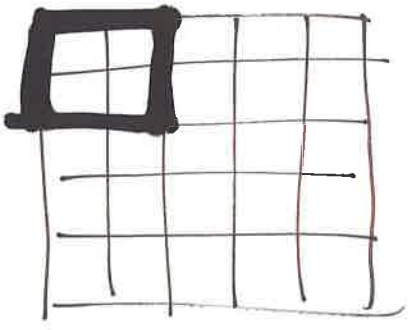
This is the input

| | |
|---|---|
| 1 | -1 |
| 0 | 2 |

This is the filter.

[redacted]

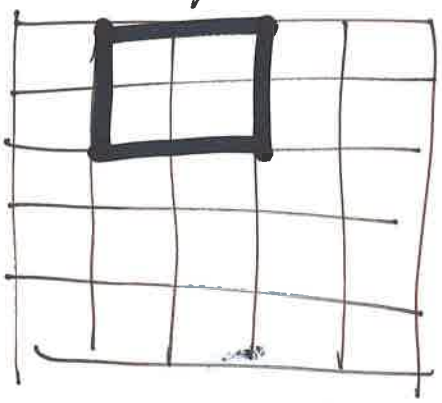
**Typically**, we would put the filter on the top left of the input:



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

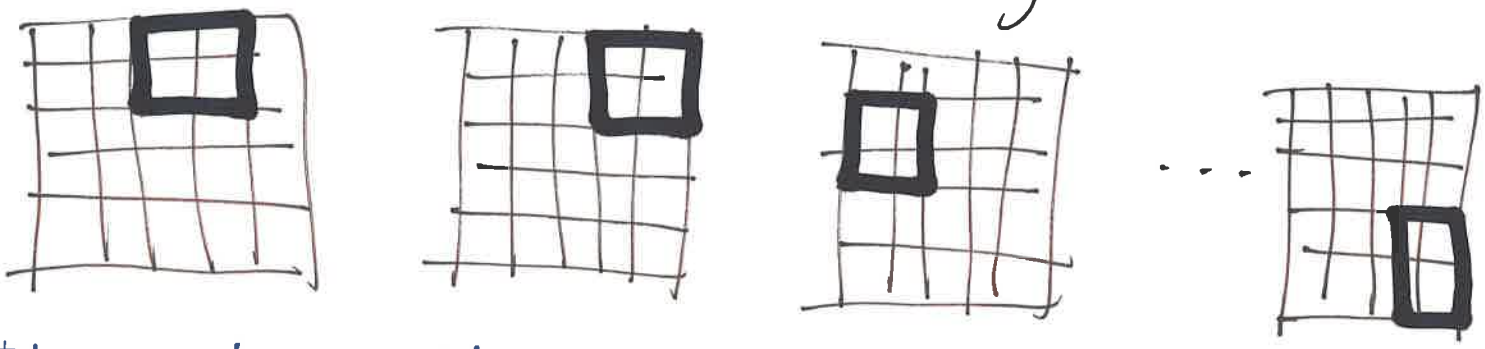
$$1 \cdot 1 + (2)(-1) + (-1) \cdot 0 + (-1)(2) = -3$$

Then, we slide the filter one unit to the right and get the second output.



$$2 \cdot 1 + 0 \cdot (-1) + (-1) \cdot 0 + 2 \cdot 2 = 4.$$

The sequence of calculations can go like this.



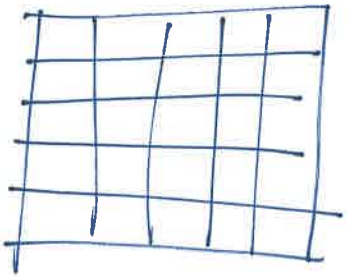
The end result is a 4 x 4 feature map consisting of all the 16 numbers we have calculated by sliding the filter:

| | | | |
|---|---|---|---|
| -3 | 4 | ... | ... |
| ... | .. | | |
| | | | |
| | | | |

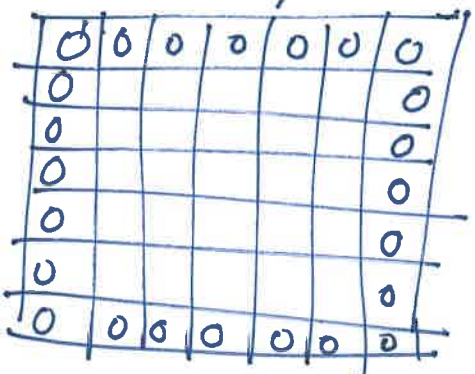
You can also imagine that we have 16 neurons that share the weights [1 −1 0 2]

Each neuron is only connected to a subset of the inputs called its local receptive field.

* We almost always have more than one filter to extract multiple features. In this case, one creates multiple feature maps in a convolutional layer. Each filter is allowed to have different weights.
* Several variations of the filtering idea exists, such as zero padding. Instead of



we can apply convolutions to the zero padded version.

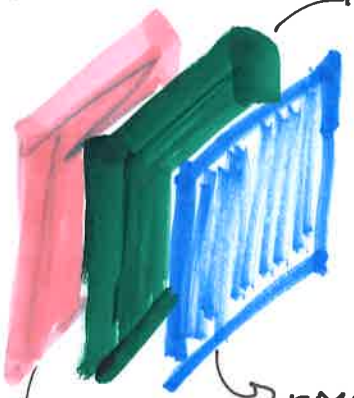


* One can also define a "stride", i.e how many squares to "jump" when sliding the filter.
* The sliding manner in which local fields of neurons are calculated is analogous to signal convolution. This is also why the networks are called convolutional networks.
* Convolutions may be followed by any activation function like Relu, tanh, etc.

# Handling colored inputs.



In this case, the image is decomposed to R, G, and B channels. Each channel is a separate image. We can stack the channels on top of each other to get a multichannel image that looks like as follows:




A filter to be applied to this "3D" input should now also be 3D. For example, if the image is 4×4, and the filter is 2×2; the first neuron local field would be:

$1 \cdot 1 + 1 \cdot -1 + 2 \cdot 1 + 0 \cdot 2$
$+ 1 \cdot 1 + (-1) \cdot 0 + 0 \cdot 0 + 1 \cdot 6$
$+ 4 \cdot 3 + 1 \cdot 2 + 2 \cdot 1 + 3 \cdot 0$
$=$ whatever.

3-channel input

| | | | |
|---|---|---|---|
| 1 | 1 | 2 | 1 |
| 2 | 0 | -1 | 1 |
| -1 | 1 | 0 | -1 |
| 0 | 1 | 2 | 3 |

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

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

| | |
|---|---|
| 1 | -1 |
| 1 | 2 |

| | |
|---|---|
| 1 | 0 |
| 0 | 6 |

| | |
|---|---|
| 3 | 2 |
| 1 | 0 |

3-channel filter.

# Handling multi-channel inputs



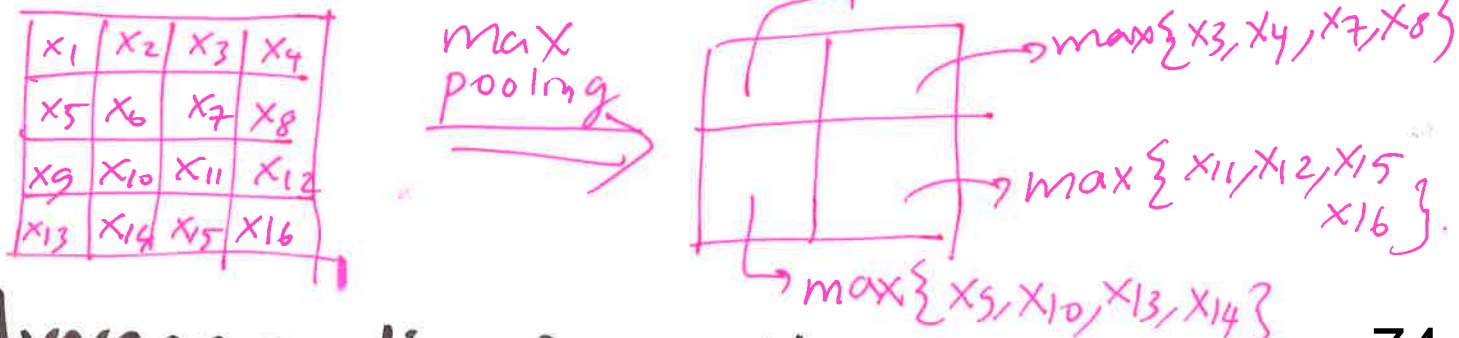
Are done in the same manner.

Suppose you applied a convolutional layer and got 10 feature maps. You can follow this by another convolutional layer. The filters of this new layer may act on all 10 channels of the previous layer (or a subset of them).

## Pooling layers.

Convolutional layers are typically followed by pooling layers to reduce the dimensions of the feature maps. A commonly used method is max-pooling.

Consider, eg, a stride of 2:




Average pooling is another possibility.

# A typical CNN architecture
(From LeCun et.al. "Gradient-Based Learning Applied to Document Recognition").

11

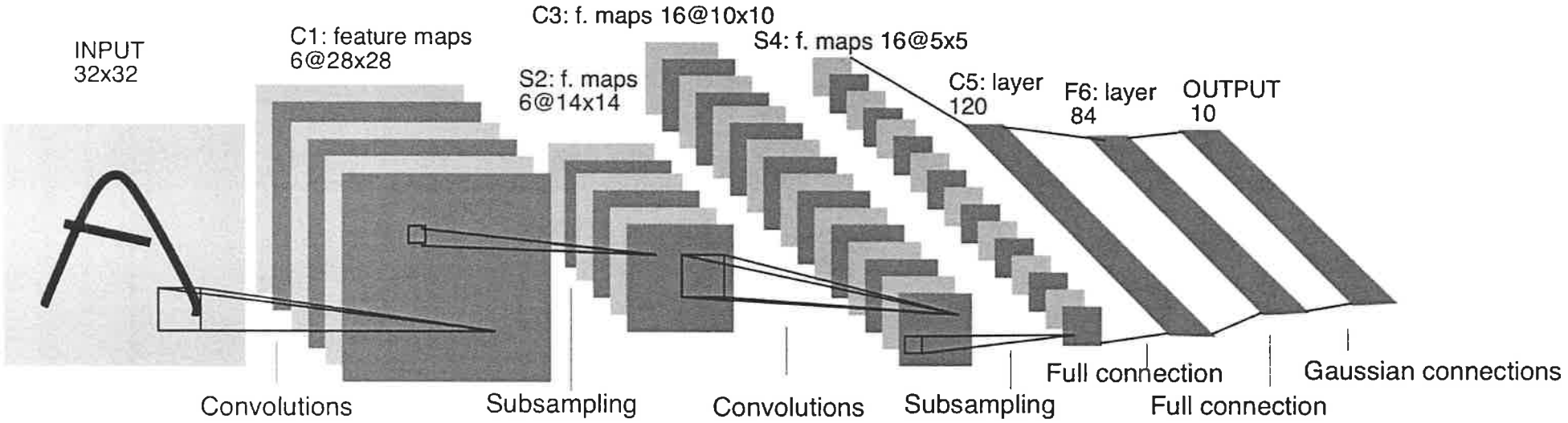



Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | X | | | | X | X | X | | | X | X | X | X | | X | X |
| 1 | X | X | | | | X | X | X | | | X | X | X | X | | X |
| 2 | X | X | X | | | | X | X | X | | | X | | X | X | X |
| 3 | | X | X | X | | | X | X | X | X | | | X | | X | X |
| 4 | | | X | X | X | | | X | X | X | X | | X | X | | X |
| 5 | | | | X | X | X | | | X | X | X | X | | X | X | X |

TABLE I

EACH COLUMN INDICATES WHICH FEATURE MAP IN S2 ARE COMBINED BY THE UNITS IN A PARTICULAR FEATURE MAP OF C3.

13

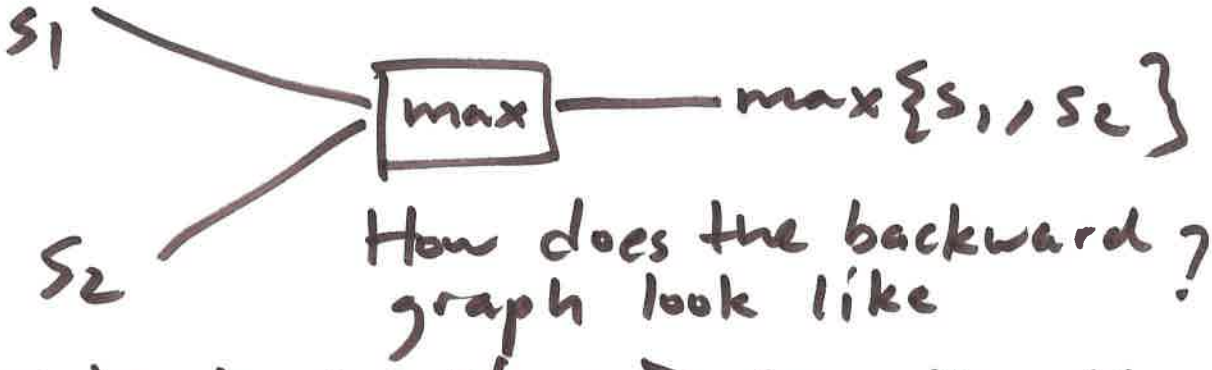
Fig. 3. Initial parameters of the output RBFs for recognizing the full ASCII set.

# Training a CNN



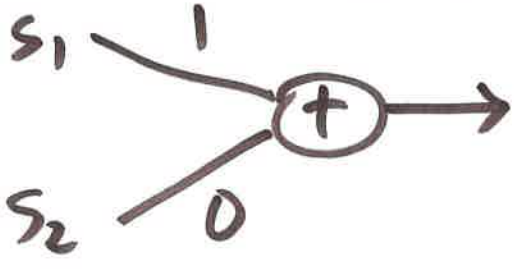
Is accomplished through the Backpropagation algorithm (similar to any multilayer NN).

- Should take care of issues like:
  - * Weight sharing
    (Not a big problem; just accumulate gradients / add them up).
  - * max. pooling:

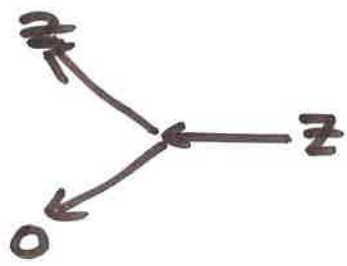



How does the backward graph look like?

Note that, if $s_1 \geq s_2$, then the above graph is equivalent to:




, so that the backward graph should be:




Similarly the case $s_1 \leq s_2$ is handled.

# L13

ECE/CS 559 Lecture 13 T 10/8

Last time: Regularization

- Idea: Avoid overexplaining by limiting to reasonable explanations: prior knowledge / constraints on weights / biases
- $\min_w R(w)$ for $\|w\|_2 \le C$ $\iff$ $\min_w R(w) + \lambda \|w\|_2^2$ ($L_2$)
- During gradient descent simply add: $\nabla_w(R(w) + \lambda\|w\|_2^2) = \nabla_w R + 2\lambda w$ ($L_1$: $\|w\|_1 = |w_1| + \dots + |w_n|$, other choice; $\sim \text{sign}(w)$)
  - regularized / penalized risk
- Dropout: Randomly prune weights during training (with prob. $p$). Include all weights (scaled by $p$) at test time.

## ① Optimization

### ⓐ Stochastic Gradient Descent

Empirical risk $R(w) = \frac{1}{n}\sum_{(x,y)\in Data} \ell(y, f(x;w))$

"True" risk $R'(w) = \mathbb{E}[\ell(Y, f(X;w))]$ — what we really want to minimize, behavior on future points.

Note: $R(w)$ is an unbiased estimate of $R'(w)$: $\mathbb{E}[R(w)] = R'(w)$

This means the gradients are unbiased: $\mathbb{E}[\nabla_w R(w)] = \nabla_w R'(w)$

random/stochastic gradients

- But... this remains true for even a single point $\mathbb{E}[\nabla_w \ell(Y, f(x;w))] = \nabla_w R(w)$ or a mini batch of points $\mathbb{E}\left[\nabla_w \sum_{(x,y)\in B} \ell(y, f(x;w))\right] = \nabla_w R(w)$
- What changes? The variance. # points ↓ variance ↑ noisiness ↑

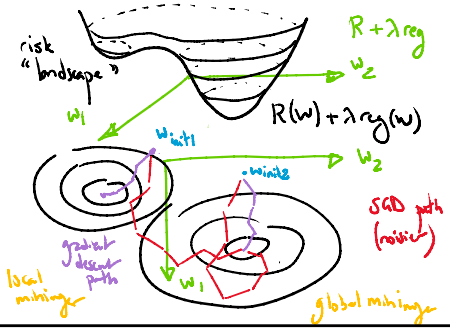
Then why don't we always use all the points?

- Reasons for minibatch SGD:
  - Computation: It's faster to get an update with small batches
  - Statistics: Homogeneous data ⇒ small batches give a glimpse of the whole
  - Optimization: The variance can be helpful to get out of local minima.

(choose batch size to trade off this part with convergence speed.)

- General form of SGD:

```
for epoch in #of epochs:
    t=0 ; Σ∇ = 0
    For (x,y) in Data:
        Forward(x); Backward(y) → gives ∇R + λ∇reg ; t += 1
        Σ∇ += ∇R + λ∇reg
        if (t+1) % B == 0:
            W += −η Σ∇
            Σ∇ = 0
```




### ⓑ Choice of $\eta$, the "learning rate"

- Because SGD introduces noise, $\eta$ should be adjusted
  - Otherwise, performance can worsen/improve/worsen (oscillate)
- Typically: Reduce $\eta$ by a factor (e.g. 90%) if sustained worsening.
- Advanced: Adapt to the landscape (e.g. flatness, steepness)
- Vary $\eta$ by layer: Later layers $\nabla_w$ ↑, make $\eta$ ↓
  by neuron: # Inputs ↑, make $\eta$ ↓ $\propto 1/\sqrt{\#\text{ inputs}}$ (heuristic)

### ⓒ Controlling instability:

- Gradients are repeatedly multiplied during back propagation.
  - As a result, they could vanish ($\times 0.9 \times 0.9 \times \dots$) or explode ($\times 1.1 \times 1.1 \times \dots$)
  - Solution: normalize gradients (across a batch or a layer), clip gradients

## ② Heuristics

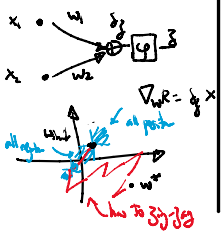
The "craft" of training neural networks.

### ⓐ Choosing the activation function:

- For the longest time: Sigmoid $\sigma$, hyperbolic tangent tanh
  - Issue: $\varphi(v)$ saturates at +1 / −1 — saturation slows training
  - Instead: ReLU $\varphi(v)$ — no saturation there; hack + fix; ... but still an issue too

### ⓑ Preprocessing inputs:

- Ideally, input coordinates are uncorrelated/centered
  - Otherwise, may slow down learning
  - Example: if $x_i$'s are always + or − all together
- Preprocess to make them have 0 mean, unit variance, 0 covariance: "whitened"




- How? $\mu = \frac{1}{|Data|}\sum_{x\in Data} x$ (mean (vector)) $\quad C = \frac{1}{|Data|}\sum_{x\in Data}(x-\mu)(x-\mu)^T$ (covariance (matrix))

$C$ is positive semidefinite: can be diagonalized $C = UDU^T$, $UU^T = U^TU = I$

Let $x_{new} = D^{-\frac{1}{2}}U^T(x-\mu)$ ← has 0 mean, covariance = I

### ⓒ Initializing the weights:

- Intuition: maintain similar statistics from layer to layer.
- Ignoring activation, assuming whitened inputs, covariance I, next layer's inputs is:

$$C_i = \frac{1}{|Data|}\sum_{x\in Data}(Wx)(Wx)^T = WW^T = n\left[\ \ \right]n\left[\ \ \right]$$

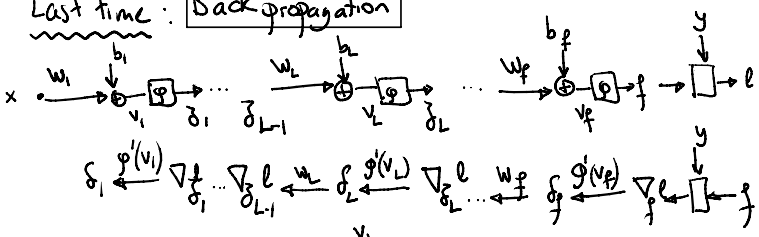
Ideally, we want to make $\mathbb{E}[WW^T] = I$

- If $w$ have 0 mean + uncorrelated, then off-diagonals are 0 ✓
- On the diagonal, we'd be adding $n$, $\mathbb{E}[w^2]$, so we get $n\,\text{var}(w)$.
- To make it identity, we just have to choose $\text{var}(w) = 1$
- E.g., choose $w$ to be i.i.d. $\mathcal{N}\left(0, \frac{1}{\sqrt{\#\text{ inputs}}}\right)$

# L11

ECE/CS 559 Lecture 11 T 10/1

Last time: **Backpropagation**



Forward: $\ell(z_L) = \ell\big(\varphi(\underbrace{W_L z_{L-1} + b_L}_{v_L})\big)$

Backward: $\nabla_{z_{L-1}} \ell = W_L^T \big(\varphi'(v_L) \circ \nabla_{z_L} \ell\big) = W_L^T \delta_L$ (element wise)

Gradients: $\nabla_{W_L} \ell = \big(\varphi'(v_L) \circ \nabla_{z_L} \ell\big) z^T = \delta_L z^T$, $\nabla_{b_L} \ell = \delta_L$

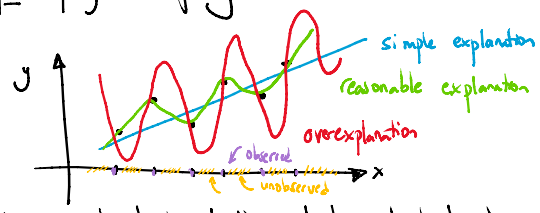
- When learning neural networks, issue is that we train on the observed contexts $x$. However, we mostly care about performance on new unobserved contexts!
- Performing well in unobserved contexts = **generalization**
  Overfitting prevents generalization (unless it's mild)
- How do we quantify this?

  Training Data $= \{(x,y),\dots\}$ ↳ use this to build NN

  Testing Data $= \{(x',y'),\dots\}$ ↳ use this to assess generalization
- We must be careful!
  We can't keep on using testing data to refine the NN, otherwise we may overfit it too and not generalize to newer data.

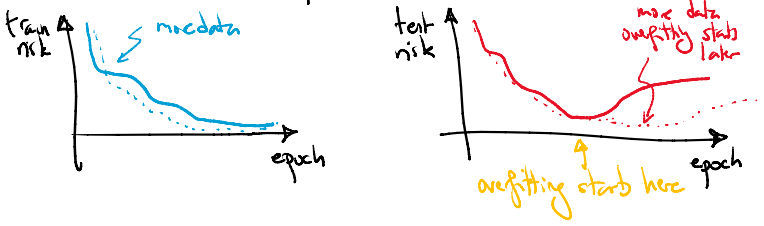
## ① Overfitting

- Tendency to overexplain the training data.
- Overexplain = when many explanations describe experience equally well, choose one that adds many complicated details and twists (typically what conspiracy theories do)
- Example: polynomial fitting




- Typical property leading to this: high sensitivity to changes in data

- Typical behavior of train vs. test risk:



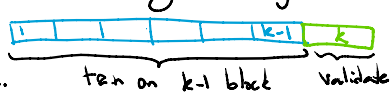

- Observations:
  - Stopping early could help alleviate overfitting
    Why? The NN will not fit the noise.
    How? By staying closer to its initialization (staying "simpler")
  - More training data reduces overfitting, delays its onset.

## ② Cross-Validation

- Since training risk is not a good enough proxy to testing risk, we could hold out some of the training data for validation:

Training Data $= \{(x,y),\dots\}$ (train on this often) Validation Data $= \{(x'',y''),\dots\}$ (check if okay only sometimes) Testing Data $= \{(x',y'),\dots\}$ (test on this rarely)

- What is validation good for?
  - Choose what epoch to stop at.
  - Choose between possible architectures, learning rates, etc. ("hyper parameters")
  - These choices are "coarser" than choosing the weights & biases
- k-Fold Cross-validation:




Change validation block, repeat. Average.

## ③ Regularization

- Cross-validation helps see if we overfit. Regularization helps prevent it.
- The word comes from inverse problems in physics:
  - To decide between many possible explanations, we need to favor some.
  - We tend to favor "simple" ones (Occam's Razor) (e.g., low energy)
- E.g., for polynomials and NNs, favor small weights:

  $\min_w R(w)$ for $\|w\|^2 \le C$ $\Leftrightarrow$ $\min_w R(w) + \lambda \|w\|_2^2$ (regularized/penalized risk)
- What does it mean? Prefer smoother functions:
- Why does it work? Can't overfit noise
  Regularized risk is a better proxy for test-risk

  $\|w\|_1 = |w_1| + \dots + |w_n|$ (other choice)
- How does it affect backpropagation? Simple!

  $\nabla_w \big(R(w) + \lambda \|w\|_2^2\big) = \nabla_w R + 2\lambda w$


Note: Any change that helps prefer some $w$ over others is regularization. This introduces what we call inductive bias, which (if right) improves generalization.

# L9&10

ECE/CS 559 Lecture 9 & 10 T,Th 9/24, 9/26

Last time: Gradient Descent; Widrow-Hoff LMS Algorithm

$$\min_w R(w) = \sum_{(x,y)\in Data} \ell(y, f(x;w))$$

$$w' \leftarrow w - \eta \nabla R(w) \qquad \nabla_w R = \left[\frac{\partial R}{\partial w_1}, \dots, \frac{\partial R}{\partial w_k}\right]$$

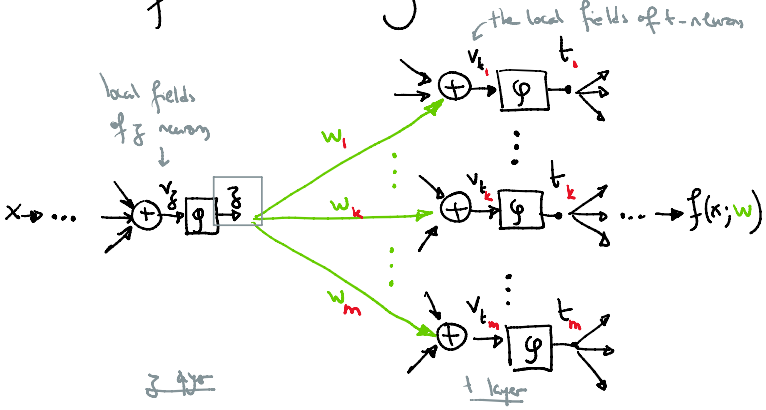
- $\ell(y, f(x;w)) = \| y - f(x;w) \|^2$

$f(x;w) = wx$

$$w' \leftarrow w - \eta \nabla R(w) = w + 2\eta \sum_{(x,y)} (y - wx) x^T \quad \text{(absorb)}$$

## Back propagation

- So far we've been calculating gradients with 1 neuron.
- What if we connect many neurons?




- We will apply the chain rule multiple times:

$$\frac{\partial R}{\partial w_k} = \sum_{(x,y)} \left[\frac{\partial \ell}{\partial w_k}\right] \quad (?)$$

need [this]

(?) $$\frac{\partial \ell}{\partial w_k} = \frac{\partial}{\partial w_k} \ell(t_k, \dots)\Big|_{t_k = \varphi(w_k z + \dots)} \quad (v_{t_k} = w_k z + \dots)$$

have this from forward pass

(two chain rules) $$= \left[\frac{\partial \ell}{\partial t_k}\Big|_{t_k=\varphi(v_{t_k})}\right] \cdot \varphi'(v_{t_k}) \cdot z \quad (*)$$

need [this] at every neuron output

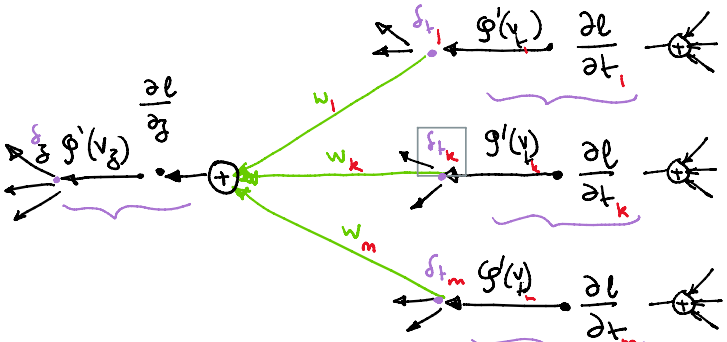
$$\delta_{t_k}$$

(*) $$\frac{\partial \ell}{\partial z}\Big|_{z=\varphi(u)} = \frac{\partial}{\partial z} \ell(t_1, \dots, t_k, \dots, t_m)\Big|_{t_k=\varphi(w_k z + \dots)} \quad (v_{t_k} = w_k z + \dots)$$

(two chain rules) $$= \sum_{k=1}^{m} \frac{\partial \ell}{\partial t_k}\Big|_{t_k=\varphi(v_{t_k})} \cdot \varphi'(v_{t_k}) \cdot w_k$$

$$\delta_{t_k}$$

- To calculate (?) and (*), we perform a forward pass to get all the outputs $(u, z, v_t, t, \dots)$, then we perform a backward pass:




Gradient of $w$ = forward signal $z$ × backward signal $\delta_t$

- Of course, the layer of $z$ has other (say $n$) neurons. Let's index the equations for these (blue $j$)

$$\frac{\partial \ell}{\partial w_{kj}} = \delta_{t_k} \cdot z_j \qquad \frac{\partial \ell}{\partial z_j} = \sum_{k=1}^{m} \delta_{t_k} w_{kj} \qquad j = 1, \dots, n$$

- If we think of $\delta_t$ as a $m$-dim vector
  of $z$ as an $n$-dim vector
  of $W$ as a $m \times n$-dim matrix
- We get the linear algebraic backward equation:

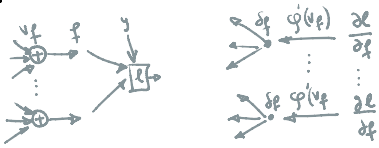
$$\nabla_W \ell = \delta_t z^T \qquad \nabla_z \ell = W^T \delta_t \qquad \delta_z = \nabla_z \ell \bullet \varphi'(v_z)$$

outer product; matrix multiplication; elementwise product

- To complete this, we need 2 things: how to start, how to handle bias
- Start at the last layer $\nabla_f \ell$ depends on the choice of $\ell$

e.g. if $\ell = \|f - y\|^2$ then

$\nabla_f \ell = 2(f - y)$, $\delta_f = \nabla_f \ell \bullet \varphi'(v_f)$

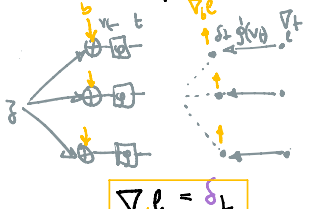



- Biases are like dead-ends on the backward path.

$$\frac{\partial \ell}{\partial b} = \frac{\partial}{\partial b} \ell(t_k, \dots)\Big|_{t_k=\varphi(\dots + b)} \quad (v_{t_k} = \dots + b)$$

$$= \frac{\partial \ell}{\partial t}\Big|_{t_k=\varphi(v_t)} \cdot \varphi'(v_{t_k}) \cdot 1$$

$$\delta_{t_k}$$




$$\nabla_b \ell = \delta_t$$

Note: Can we forward eqs to get these: $\ell(t) = \ell(\underbrace{\varphi(w_j z + b)}_{t})$ $\Rightarrow$ $\nabla_w \ell = [\underbrace{\nabla_t \ell \bullet \varphi'(v_t)}_{\delta_t}] z^T$, $\nabla_b \ell = [\underbrace{\nabla_t \ell \bullet \varphi'(v_t)}_{\delta_t}] \cdot 1$, $\nabla_z \ell = W^T [\underbrace{\nabla_t \ell \bullet \varphi(v_t)}_{\delta_t}]$

# L8

ECE/CS 559 Lecture 8 Th 9/19

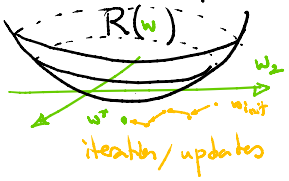
Last time: Linear Regression with Squared Loss

- Data: $\{(x,y),\dots\}$ $x \in \mathbb{R}^n$ $y \in \mathbb{R}^m$
- Predictor: $f(x;W) = Wx$ $W \in \mathbb{R}^{m\times n}$
- Goal: minimize $_W R(W) = \sum_{(x,y)\in Data} \underbrace{\| y - Wx \|^2}_{\ell(y, Wx) \equiv \ell((x,y),W)}$

$m\left[\hat{W}\right] \cdot \left[\overset{Data}{\overset{\downarrow}{\;\;X\;\;}}\right] = m\left[\overset{Data}{\overset{\downarrow}{\;\;Y\;\;}}\right]$

$W = Y\,[X^T(XX^T)^{-1}]$ Pseudo inverse

it's a quadratic equation




## ① Gradient Descent

Goal: minimize $_W R(W) = \sum_{(x,y)\in Data} \underbrace{\| y - f(x;W) \|^2}_{\ell(y, f(x;W))}$

Idea: Set the derivative to zero! → reduce step by step.

Gradient: $\nabla_W R = \left[\frac{\partial R}{\partial w_1}, \dots, \frac{\partial R}{\partial w_k}\right]$

Each step: Change $W$ by $\Delta W$ : $W' \leftarrow W + \Delta W$.

$$R(W') = R(W+\Delta W) = R(W) + \nabla R(W)^T \Delta W + O(\|\Delta W\|^2)$$

Taylor expansion; ignore

How should we choose $\Delta W$ to get the best reduction?

- To minimize $_{\Delta W} \nabla R^T \Delta W$, let $\Delta W \propto -\nabla R$ (proportional): $W' \leftarrow W - \eta \nabla R(W)$

1-D Example: $R(w) = w^4$ $\frac{dR}{dw} = 4w^3$ $\eta = \frac{1}{8}$

$w(0)=1$

$w(1) \leftarrow w(0) - \nabla R(w(0)) \cdot \eta$

$1 - 4\cdot 1^3 \cdot \frac{1}{8} = \frac{1}{2}$

$w(2) \leftarrow w(1) - \nabla R(w(1)) \cdot \eta$

$\frac{1}{2} - 4\left(\frac{1}{2}\right)^3 \cdot \frac{1}{8} = \frac{7}{16}$

- What's happening?

$w(t) \leftarrow w(t-1) - 4\,w(t-1)^3\,\eta = [1 - 4\eta\, w(t-1)^2]\, w(t-1)$

- If $\eta$ is small enough, always diminishes. ($<1$) Since bounded from below (by 0) will converge.
- But will it converge to 0? Yes, think about why.
- What if $\eta$ is too big? Try $\eta = \frac{1}{2}$ with $w(0) = 1$.

## ② Widrow-Hoff LMS Algorithm

- Let's apply gradient descent to linear regression. Recall:
- $\nabla R(W) = -2 \sum_{(x,y)\in Data} (y - Wx)x^T$ same shape as $W$

$$W \leftarrow W - \eta \nabla R(W) = W + 2\eta \sum_{(x,y)} (y - Wx)x^T$$

absorb

- When $y \in \mathbb{R}^1$ then $W = w^T$ and $y - w^T x \in \mathbb{R}$

The updates become: $w \leftarrow w + \eta \sum_{(x,y)} (y - w^T x)\, x$

- We could do this 1 data point at a time:

For each $(x,y) \in$ Data: $w \leftarrow w + \eta (y - wx)\, x$

Similar to the perceptron learning algorithm!

## ③ Delta Rule

- Squared loss for single neuron with differentiable $\varphi(\cdot)$:

$$R(w) = \sum_{(x,y)} (y - \varphi(w^T x))^2$$

$$\nabla R(w) = -2 \sum_{(x,y)} (y - \varphi(w^T x))\, \varphi'(w^T x)\, x^T$$ chain rule

$$w \leftarrow w + 2\eta \sum_{(x,y)} (y - \varphi(w^T x))\, \varphi'(w^T x)\, x$$

- 1 data point at a time:

$$w \leftarrow w + \eta (y - \varphi(w^T x))\, \varphi'(w^T x)\, x$$

Example $\varphi(\cdot)$:

- Sigmoid: $\varphi(v) = \frac{1}{1+e^{-av}}$

$$\varphi'(v) = \frac{0 - (-ae^{-av})}{(1+e^{-av})^2} = \frac{ae^{-av}}{(1+e^{-av})^2} = a\varphi(v)(1-\varphi(v))$$

Since $1 - \varphi(v) = \frac{e^{-av}}{1+e^{-av}}$

- ReLU: $\varphi(v) = \begin{cases} v & v \geq 0 \\ 0 & \text{otherwise} \end{cases}$

$$\varphi'(v) = \begin{cases} 1 & v \geq 0 \\ 0 & \text{otherwise} \end{cases} = \text{step}(v)$$

# L7

ECE/CS 559 Lecture 7 9/17

Last time: Perceptron Learning Example — Supervised Learning

- Data: $\{(x,y),\dots\}$ $x \in \mathcal{X}$ (feature space), $y \in \mathcal{Y}$ (label space)
- Task: Predict/imitate $y$ using only $x$.
- How: Use a predictor — neural network $y_w(x) \equiv f(x;w)$
- Loss: how far $y$ and $f(x;w)$ are: $\ell(y,f) = \mathbb{1}\{y \neq f\}$ or $\|y-f\|^2$
- Risk: (empirical) $R(w) = \frac{1}{|\text{Data}|} \sum_{(x,y)\in\text{Data}} \ell(y, f(x,w))$
- Goal: Minimize (empirical) risk, make few mistakes/errors / stay close to $y$.

- Types of algorithms:
  - Online: - See $(x,y)$, update $w$ - See $(x',y')$, update $w$ - See $(x'',y'')$, update $w$ ⋮
  - Batch: - Get Data $= \{(x,y),\dots\}$ • Epoch 1: use Data, update $w$ • Epoch 2: use Data, update $w$ ⋮

① Linear Regression with Squared Loss

- Data: $\{(x,y),\dots\}$ $x \in \mathbb{R}^n$ $y \in \mathbb{R}^m$
- Predictor: $f(x;w) = Wx$ $W \in \mathbb{R}^{m\times n}$ (this is a neuron with $g(v) = v$, identity activation)
- Goal: $\min_W R(W) = \sum_{(x,y)\in\text{Data}} \underbrace{\|y - Wx\|^2}_{\ell(y,Wx)}$

Analytic Solution: Optimality conditions, unconstrained $\Rightarrow \nabla_w R = 0$ (gradient)

- Single data point $\ell = \sum_{k=1}^{m} \left(y_k - \sum_{j=1}^{n} w_{kj} x_j\right)^2$

$$\frac{d\ell}{dw_{kj}} = -2x_j\left(y_k - \sum_{j'=1}^{n} w_{kj'}x_{j'}\right) = -2y_k x_j + \sum_{j'=1}^{n} w_{kj'}(x_j x_{j'})$$


$$\Rightarrow \nabla_W \ell = -2yx^T + Wxx^T$$

- All data points: $R(w) = \sum_{(x,y)\in\text{Data}} \ell(y,Wx) \Rightarrow \nabla_w R = \sum_{(x,y)} \nabla_w \ell$

$$\nabla_W R = \sum_{(x,y)\in\text{Data}} -2yx^T + 2W \sum_{x\in\text{Data}} xx^T$$

$$= -2\underbrace{\begin{bmatrix} | & | & & | \\ y & y & \cdots & y \\ | & | & & | \end{bmatrix}}_{m\times|\text{Data}|} \underbrace{\begin{bmatrix} - & x & - \\ - & x & - \\ & \vdots & \\ - & x & - \end{bmatrix}}_{|\text{Data}|\times n} + 2W \underbrace{\begin{bmatrix} | & | & & | \\ x & x & \cdots & x \\ | & | & & | \end{bmatrix}}_{n\times|\text{Data}|} \begin{bmatrix} - & x & - \\ - & x & - \\ & \vdots & \\ - & x & - \end{bmatrix}$$

$$= -2YX^T + 2WXX^T$$

- $\nabla_W R = 0 \Rightarrow -2YX^T + 2WXX^T = 0$

$$\Rightarrow W = Y\boxed{X^T(XX^T)^{-1}}$$ Moore-Penrose Pseudo inverse

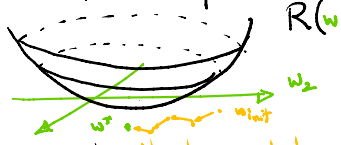
Compare to: $Wx \approx Y$ not invertible if $|\text{Data}| > n$

$$m\begin{bmatrix} & W & \end{bmatrix} n\begin{bmatrix} | & & | \\ x & \cdots & x \\ | & & | \end{bmatrix} \approx m\begin{bmatrix} | & & | \\ y & \cdots & y \\ | & & | \end{bmatrix}$$ $|\text{Data}|$

but it's "like" we inverted it above.

- Where are the updates? One single batch update!

Intuition: it's a quadratic equation




- Other shapes → solve by iterative updates

② Gradient Descent

Goal: $\min_W R(W) = \sum_{(x,y)\in\text{Data}} \underbrace{\|y - f(x;W)\|^2}_{\ell(y,f(x;W))}$

Idea: Set the derivative to zero! reduce step-by-step.

Gradient: $\nabla_w R = \left[\frac{\partial R}{\partial w_1}, \dots, \frac{\partial R}{\partial w_k}\right]$

Each step: Change $w$ by $\Delta w$: $w' \leftarrow w + \Delta w$.

$$R(w') = R(w+\Delta w) = R(w) + \nabla_w R(w)^T \Delta w + O(\|\Delta w\|^2)$$

Taylor expansion; ignore

How should we choose $\Delta w$ to get the best reduction?

- To minimize $\nabla_w R^T \Delta w$, let $\Delta w \propto -\nabla_w R$: $w' \leftarrow w - \eta \nabla R(w)$

proportional

1-D Example: $R(w) = w^4$ $\frac{dR}{dw} = 4w^3$ $\eta = \frac{1}{8}$

$w(0) = 1$

$w(1) \leftarrow w(0) - \nabla R(w(0))\cdot\eta = 1 - 4\cdot 1^3 \cdot \frac{1}{8} = \frac{1}{2}$

$w(2) \leftarrow w(0) - \nabla R(w(0))\cdot\eta = \frac{1}{2} - 4\left(\frac{1}{2}\right)^3 \cdot \frac{1}{8} = \frac{3}{16}$

- What's happening?

$$w(t) \leftarrow w(t-1) - 4w(t-1)^3\eta = \underbrace{[1 - 4\eta w(t-1)^2]}_{<1} w(t-1)$$

- If $\eta$ is small enough, always diminishes. ($<1$) Since bounded from below (by 0) will converge.
- But will it converge to 0? Yes, think about why.
- What if $\eta$ is too big? Try $\eta = \frac{1}{2}$ with $w(0) = 1$.