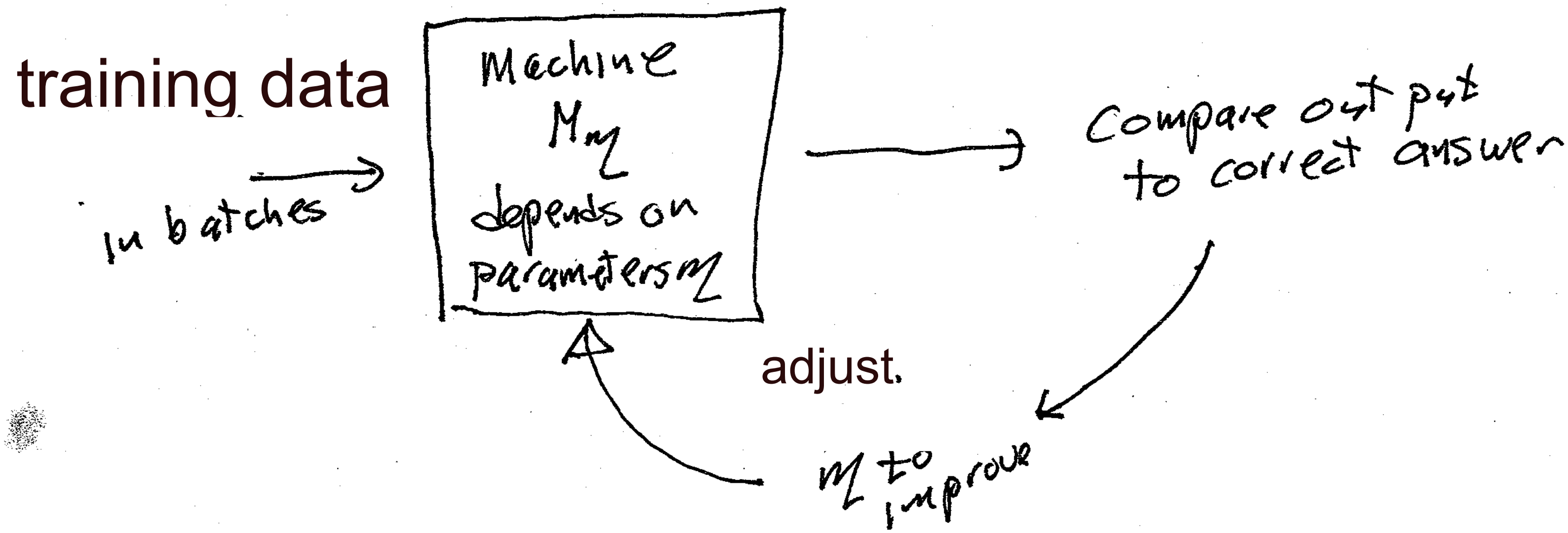
These Lectures are based on two excellent, free on-line books (links on syllabus page)

- Neural Networks and Deep Learning by Michael Nielsen
- Deep Learning by Goodfellow, Bengio and Courville

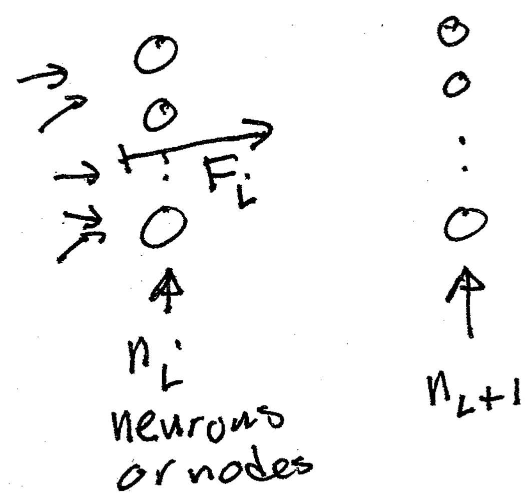
MACHINE LEARNING BIG PICTURE

Lecture ML9



DATA is sent in as mini-batches, after each mini batch $\underset{\sim}{m}$ is adjusted. After all the mini batches have been "learned" the final machine $M_{\underset{\sim}{m}_{final}}$ is run on test data to see how often it yields correct answers.

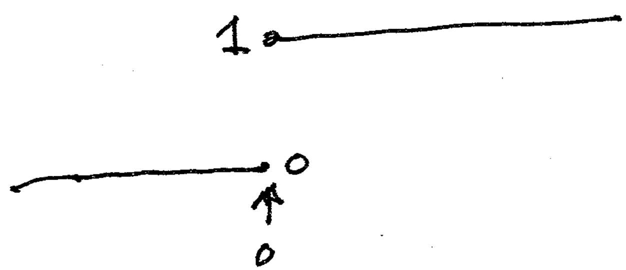
- Many possible structures for M_y, etc.
- We first cover the big math picture for a push forward, deep, fully connected net
- Each layer is represented by a function F_i. it inputs $\vec{x} \in \mathbb{R}^{n_i}$ and outputs $\vec{z} \in \mathbb{R}^{n_{i+1}}$



- $F_i(x, A_i, \vec{b}_i)$ depends on the parameters
  - A_i an $(n_{i+1} \times n_i)$ matrix of weights so $A: \mathbb{R}^{n_i} \to \mathbb{R}^{n_{i+1}}$
  - $\vec{b}_i$ an $(n_{i+1} \times 1)$ vector = the bias
  - an activation function σ

$$F(x, A_i, b_L) = \sigma(A_L x + \vec{b}_L)$$

The activation function has variou forms



- step function

- Sigmoid

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

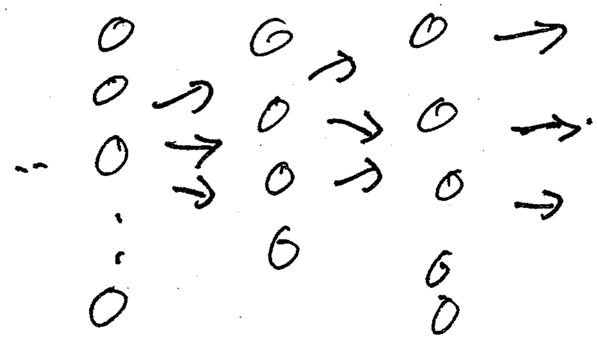
- ramp or ReLu

$$\sigma(z) = \max(z, 0)$$

- The activation function is vectorized, i.e. it acts on each componet of a vector
- so $\sigma(z_1, \ldots, z_n) = (\sigma(z_1), \ldots, \sigma(z_n))$

and $F_L : \mathbb{R}^{n_L} \to \mathbb{R}^{n_{L+1}}$

- Putting the pieces together from many layers



$$G_\eta = F_K \circ F_{K-1} \circ \cdots \circ F_1, \quad \eta = (A_1, \vec{b}_1, A_2, \vec{b}_2, \ldots, A_K, \vec{b}_K)$$

all the weights and bias together (lots of parameters!).

- Now we train the machine with training data $\vec{x}_1, \ldots, \vec{x}_n$ which are correctly characterized as $\vec{y}_1, \ldots, \vec{y}_n$.

- Now throw in all the training data and construct the cost or objective or error function

$$\Phi_{\eta}(\vec{x}_1, \ldots, \vec{x}_n) = \frac{1}{N} \sum_{i=1}^{N} \left| G_{\eta}(\vec{x}_i) - y_i \right|^2$$

(This is simplest, least squares version. More sophisticated versions later)

- We treat this as a function of η and use an optimization routine to diminish Φ_{η} to $\Phi_{\eta'}$.

- Repeat with $\eta \to \eta'$.

- In practise, a radomn subset of training data is thrown in, m is adjusted, then another minibatch, etc.
- The goal is to get a M_m given by G_m that <u>generalizes</u> ie. works well on test data that is not in the trainingset
- A big issue is how much to optimize m for just the trainging set. Don't want the machine to memorize the training data and not generalize to other test data
- This is called over-fitting.
- Now the question is why F_m takes this form?
- This is connected to why it is called a <u>neural net</u>

- It will be easier to understand the form of a neural net after we know a little about actual neurons

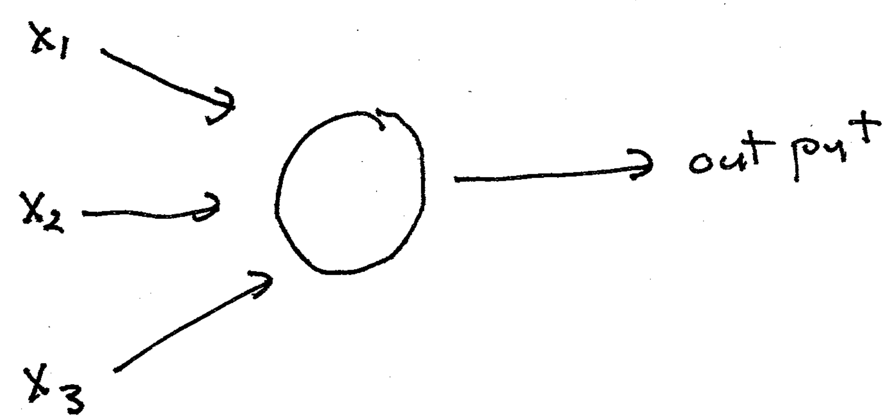
- Video on Youtube by Marc Dingman

https://www.youtube.com/watch?v=6qS83wD29PY

- $\boxed{\text{Lecture ML6}}$ →

$\boxed{\text{Artificial neurons and a single layer net}}$

- We first describe a simple artificial neuron called "the perceptron"



- The output is zero or one (fire or don't fire)
- The neuron weighs the input using weights w_1, w_2 and w_3
- A threshold $-b$ is set (minus sign explained later) also called the <u>bias</u>

- Rule! output is zero if

$$w_1 x_1 + w_2 x_2 + w_3 x_3 \leq -b$$

output is one if

$$w_1 x_1 + w_2 x_2 + w_3 x_3 > -b$$

- Example: you are trying to decide whether to do your math HW tonight
  - x_1 = how close is the due date
  - x_2 = how long is the HW
  - x_3 = what your friends are doing tonight
- You weigh up these various factors and make a decision 0 = no, 1 = yes.
- We want to express the decision process more succinctly.

- Let $\sigma(z) = 0 \quad z \leq 0$ activation function

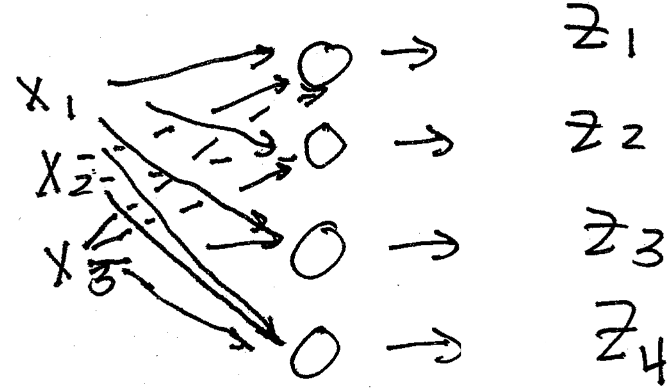
  $\qquad\qquad = 1 \quad z > 0$

- Let $F(x) = \sigma(\vec{w}^T \vec{x} + b)$ with $\vec{w}^T = [w_1\ w_2\ w_3]$, $\vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$

  then $F(x) = 0 \Rightarrow$ NO

  $\qquad\qquad = 1 \Rightarrow$ yes.

- Now we want to model a more complicated decision or classification problem using multiple perceptron



- The k'th perceptron has weights $\vec{w}_k$ and threshold or bias b_k

we get for each k

$$z_k = \sigma(\vec{w}_k^T \vec{x} + b_k) \quad k = 1, \ldots, m = \#\text{ of neurons}$$

- We want to combine all these into a matrix form
- Let $W = \begin{bmatrix} \vec{w}_1 & \cdots & \vec{w}_m \end{bmatrix}$ $\quad \vec{b} = \begin{bmatrix} b_1 \\ \vdots \\ b_m \end{bmatrix}$

then

$$W^T \vec{x} + \vec{b} = \begin{bmatrix} \vec{w}_1^T \\ \vdots \\ \vec{w}_m^T \end{bmatrix} \vec{x} + \begin{bmatrix} b_1 \\ \vdots \\ b_m \end{bmatrix}$$

$$= \begin{bmatrix} w_1^T \vec{x} + b_1 \\ \vdots \\ w_m^T \vec{x} + b_m \end{bmatrix} = \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_m \end{bmatrix}$$

- The last step is to vectorize the activation σ

$$\sigma\begin{bmatrix} z_1 \\ \vdots \\ z_m \end{bmatrix} = \begin{bmatrix} \sigma(z_1) \\ \vdots \\ \sigma(z_m) \end{bmatrix}$$

- So letting $A = W^T$, the weight matrix our one layer machine is described by

$$F(\vec{x}) = \sigma(A\vec{x} + \vec{b})$$

- We now change our point of view on the one layer of perceptions and treat it as a learning machine.

Lecture MLC

Learning and Mutiple Layes

- So we return to the characterization problem

  so $\vec{x}_1, \ldots, \vec{x}_N$ are inputs with correct

  outputs $\vec{y}_1, \vec{y}_2, \ldots, \vec{y}_N$

- We fix a value of the parameters (the weights and bias') A, $\vec{b}$

  and for each $\vec{x}_i$ input the machine outputs

$$F(\vec{x}_i) = \sigma(A\vec{x}_i + \vec{b})$$

- For each i, this has least squares error

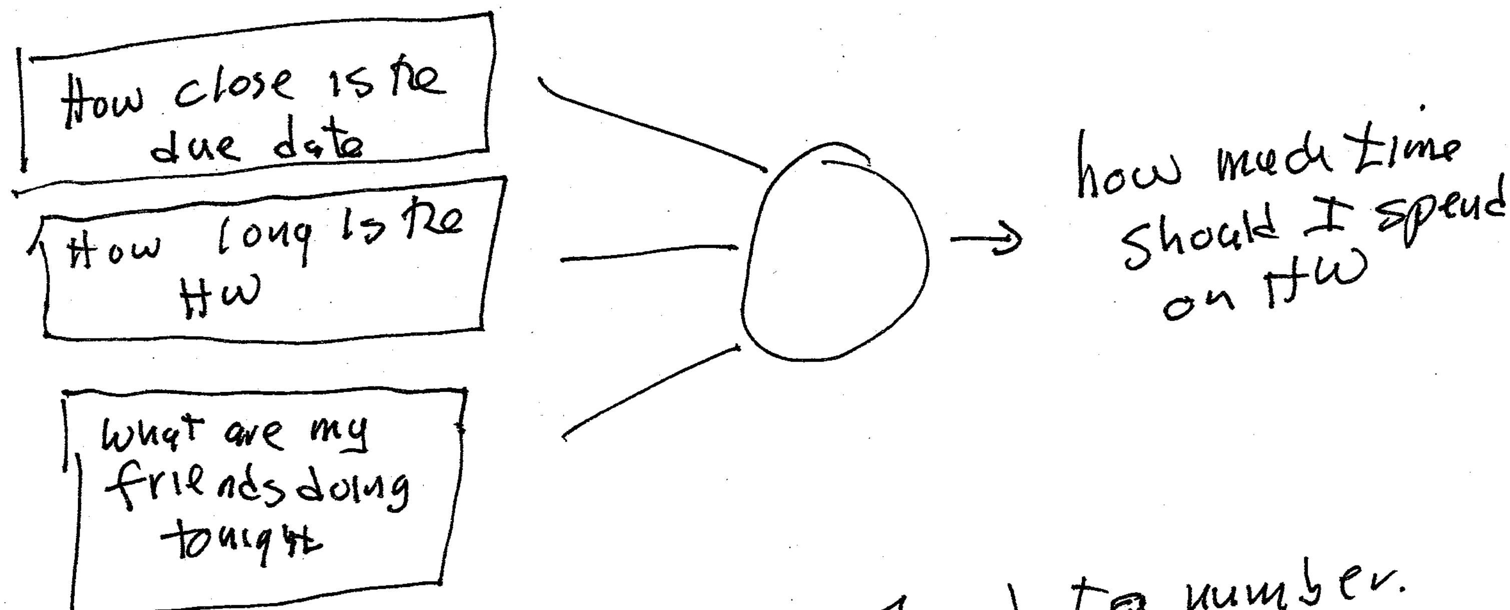
$$E_i = \|(A\vec{x}_i + \vec{b}) - \vec{y}_i\|^2$$

- For the total error over all inputs we average these.

$$\Phi(A, \vec{b}) = \frac{1}{N}\sum_{i=1}^{N} E_i = \frac{1}{N}\sum_{i=1}^{N} \|(Ax_i + \vec{b}) - \vec{y}_i\|^2$$

- Now we optimize, i.e. find A_f and $\vec{b}_f$ which minimize $\Phi(A, \vec{b})$
- We then declare our final machine to be $\sigma(A_f \vec{x} + \vec{b}_f)$
- Note the similarity to least squares and polynomial fitting
- Looking more closely, how do we optimize?
- Usual thing is to differentiate Φ with respect to A and $\vec{b}$, etc.
- But σ is not differentiable.

- Another issue with T is that it is restrictive, binary output
- BACK TO the HW example

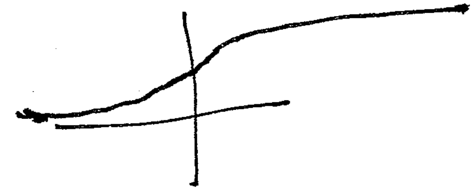


The output is not just 0 or 1, but a number.

So we probably want σ at least continuous or maybe differentiable. We want small changes in the parameter to yield small changes in the output.

- The sigmoid $\sigma(z) = \dfrac{1}{1+e^{-z}}$

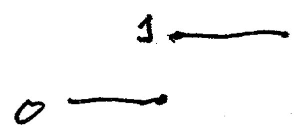
note! $\sigma(z) \to 1$ as $z \to \infty$
$\sigma(z) \to 0$ as $z \to -\infty$



is nice and differentiable but computationally expensive

- The ramp $\sigma(z) = \max(z, 0)$

is continuous, it's "derivative" is



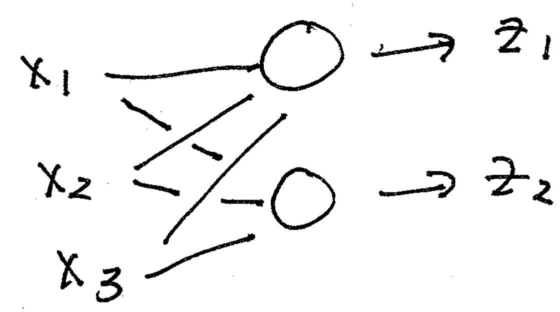
which is not so bad and is computationally tame.

For now, let σ be the sigmoid for theoretical ease

Let's study optimization or learning for one level

$$F(x, A, b) = \sigma(A x + b)$$

Let's have three inputs and 2 neurons



So $z_1 = \sigma(w_{11} x_1 + w_{12} x_2 + w_{13} x_3 + b_1)$

$z_2 = \sigma(w_{21} x_1 + w_{22} x_2 + w_{23} x_3 + b_2)$

and if the true values are y_1 and y_2 we have

$$\Phi = \Big[\big(\sigma(w_{11} x_1 + w_{12} x_2 + w_{13} x_3 + b_1) - y_1\big)^2 + \big(\sigma(w_{21} x_1 + w_{22} x_2 + w_{23} x_3 + b_2) - y_2\big)^2 \Big] \Big/ 2$$

17

- We want to minimize the error Φ as a function of the parameters the w's and b's
- So we treat Φ as a function of these and compute $\nabla\Phi$ and find critical points and see if they are loc max, min or saddles
- For example, $\nabla\Phi = \left[\frac{\partial\Phi}{\partial w_{11}}, \ldots, \frac{\partial\Phi}{\partial w_{23}}, \frac{\partial\Phi}{\partial b_1}, \frac{\partial\Phi}{\partial b_2}\right]$

(σ (same argument) - y_1)

with $$\frac{\partial\Phi}{\partial w_{11}} = \sigma'(\text{same argument}) \cdot x_1$$

$$\frac{\partial\Phi}{\partial b_1} = \sigma'(\text{same argument}) \cdot 1$$

by the chain Rule.

- This is complicated for just this simple one layer but we need many layers with many neurons and maybe thousands of parameters.

- So we need new idea

  (1) A better optimization scheme namely **Gradient Descent**

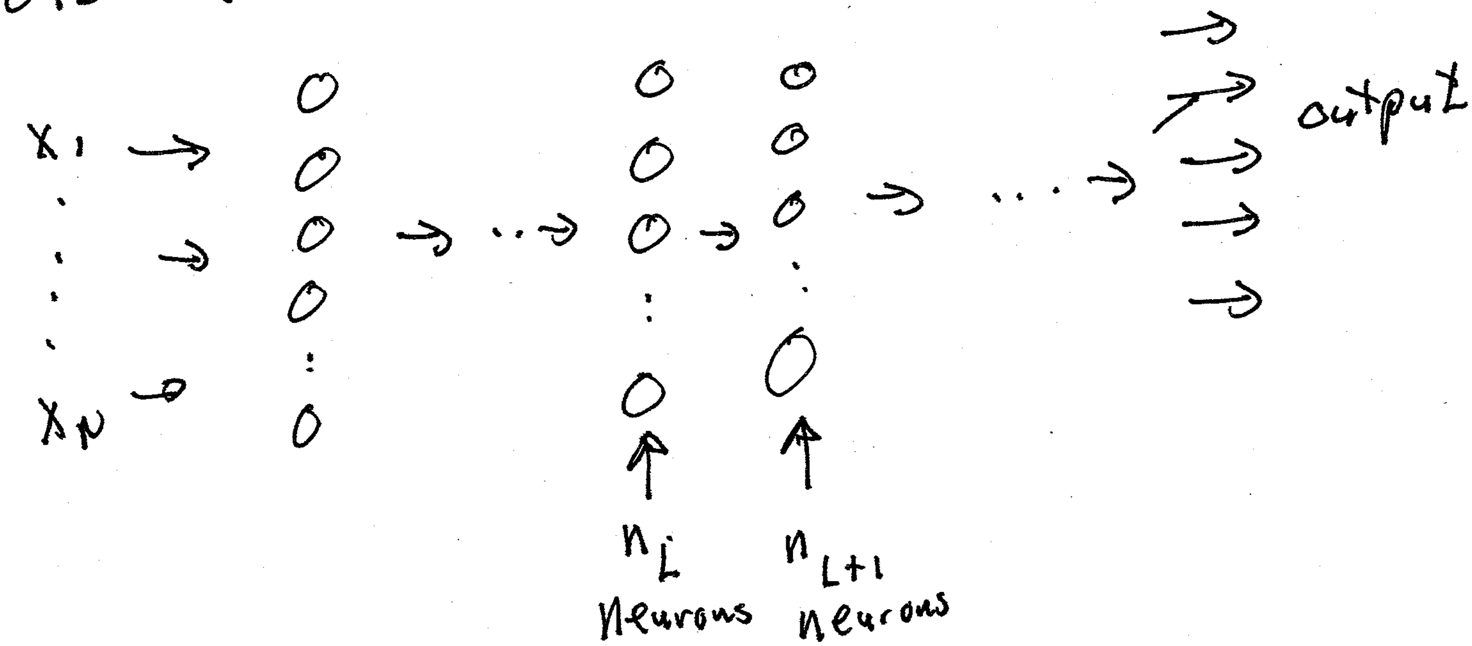
  (2) A clever way of computing $\nabla \Phi$ when there are many layers

- We will cover each of these in more detail in later lectures

- Now to finish the introduction we describe multiple layers — this is the "Deep" in *deep* learning

- One way to think of this is decision making in stages

- For example, first you decide how much time to allot to your math HW tonight, then you decide what order to fit it in with your other HW.



- each layer is given by a function

$$F_i(\vec{x}, A_i, \vec{b}_i) = \sigma(A_i x + \vec{b}_i)$$

$\boxed{\text{PIX}}$

- The layers act sequentially

$$F_1 \text{ then } F_2 \text{ then } F_3 \cdots F_L$$

- Mathematically, this is a composition (recall it is written in the reverse order)

$$F = F_L \circ F_{L-1} \circ \cdots \circ F_2 \circ F_1$$

- Then the least squares error is

$$\Phi = \sum_{i=1}^{N} \| F(x_i) - y_i \|^2$$

and it depends on all the A_i and b_i so $\nabla \Phi$ is a chore to compute

- This net is called "Feed forward" since information just flows in one direction

input $\rightarrow$ output