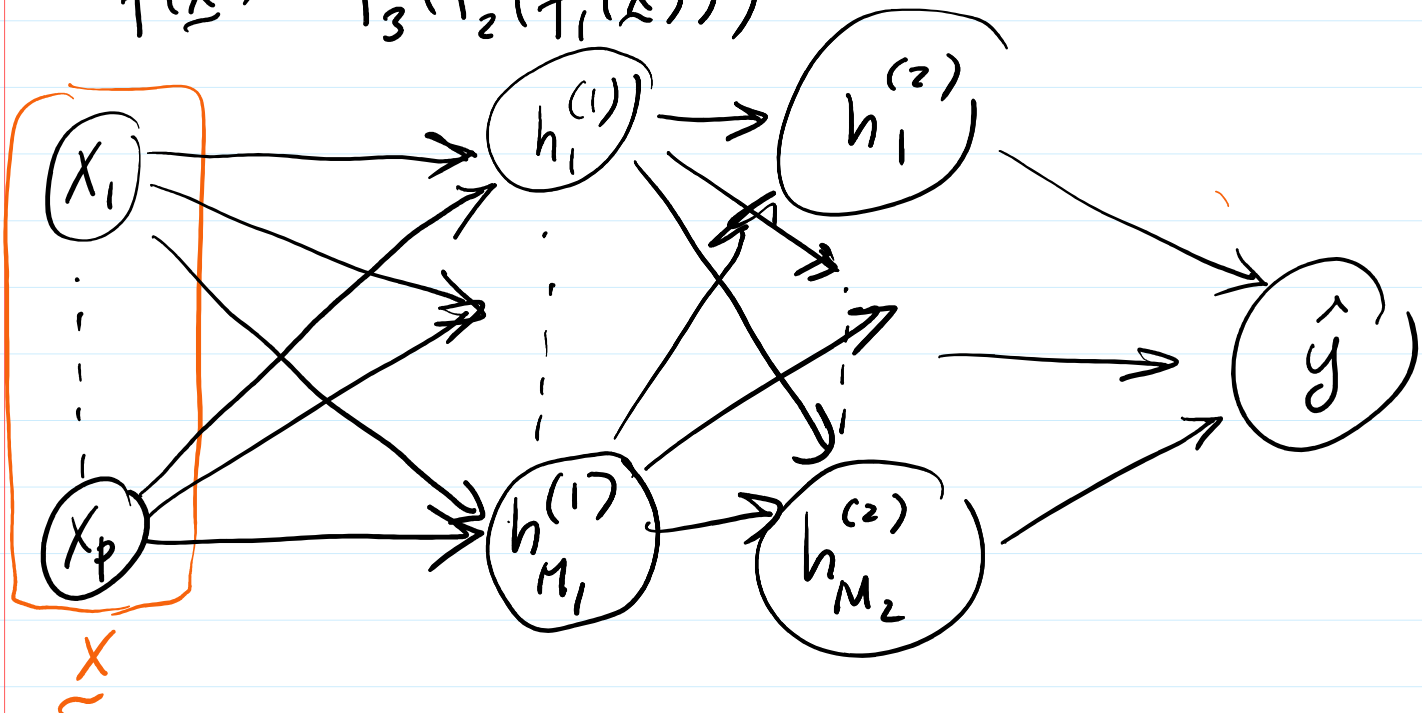
$L = 3$

$$\hat{f}(\underset{\sim}{x}) = \hat{f}_3(\hat{f}_2(\hat{f}_1(\underset{\sim}{x})))$$



$$h^{(1)} = g^{(1)}(\underbrace{W^{(1)}}_{M_1 \times P}\underbrace{\underset{\sim}{x}}_{\mathbb{R}^P} + \underbrace{b^{(1)}}_{M_1})$$

$$h^{(2)} = g^{(2)}(\underbrace{W^{(2)}}_{M_2 \times M_1}\underbrace{h^{(1)}}_{M_1} + \underbrace{b^{(2)}}_{M_2})$$

$$\hat{y} = g^{(3)}(\underbrace{W^{(3)}}_{1 \times M_2}\underbrace{h^{(2)}}_{M_2} + \underbrace{b^{(3)}}_{1}) \longleftarrow \mathbb{R}$$

Why are NNs interesting?

Claim: NNs are universal approximators.

True, but also true of many other methods.

e.g. poly regression, KNN

No, mention of convergence rate.

NNs are automatic feature engineering machines.

One way to create complex methods is via feature engineering.

Start w/ $\underset{\sim}{x} \in \mathbb{R}^p$

Could fit

$$\hat{f}(\underset{\sim}{x}) = \underset{\sim}{x}^T \beta + \beta_0$$ [linear model]

but I could also create a feature map

$$\phi : \mathbb{R}^p \to \mathbb{R}^M \quad (\text{typ. } M > P)$$

$$\phi : \mathbb{R} \to \mathbb{R}^{M} \quad (\text{typ. } M > P)$$

and then fit a linear model to this:

$$\hat{f}(\underset{\sim}{x}) = \phi(\underset{\sim}{x})^T \beta + \beta_0 .$$

<u>Problem</u>: need to specify ϕ manually

<u>Instead</u>: work w/ parameterized collection of feat maps ϕ_α

and fit

$$\hat{f}(\underset{\sim}{x}) = \phi_\alpha(\underset{\sim}{x})^T \beta + \beta_0$$

Then I need to learn parameters

$$(\beta, \beta_0, \alpha)$$

<u>Called</u>: feature learning.

non-lin.

<u>e.g.</u> $\phi_\alpha(\underset{\sim}{x}) = g\left(W\underset{\sim}{x} + b\right)$

where $\alpha = (W, b)$

hidden

This is equiv. to a single hidden layer NN:

$$\hat{f}(\underset{\sim}{x}) = \beta_0 + \beta^T g(W\underset{\sim}{x} + b).$$

Deep NNs w/ lots of layers are just learning really complicated feat maps ϕ_α

$$\phi_\alpha(\underset{\sim}{x}) = \hat{f}_{L-1}(\hat{f}_{L-2}(\cdots \hat{f}_2(\hat{f}_1(\underset{\sim}{x}))))$$

$$\alpha = (W^{(L-1)}, b^{(L-1)}, W^{(L-2)}, b^{(L-2)}, \ldots)$$

<u>For regression</u>: predicting $y \in \mathbb{R}$

last layer typ uses $g^{(L)}(x) = x$

so that

$$\hat{f}(\underset{\sim}{x}) = \underbrace{W^{(L)}}_{1 \times M_{L-1}} \underbrace{h^{(L-1)}}_{M_{L-1}} + \underbrace{b^{(L)}}_{1}$$

$(x)M_{L-1} \quad M_{L-1} \quad 1$

For classification: predict $y \in \{1, \dots, K\}$

Typically use $g^{(L)}(h) = \text{softmax}(h)$ ← K-vector

$$\text{Softmax}(h)_i = \frac{e^{h_i}}{\sum_{j=1}^{K} e^{h_j}}$$

h: K-vector

$z = \text{Softmax}(h)$

then ① $z_i \geqslant 0$

② $\sum_{i=1}^{K} z_i = 1$.

$$O(\underset{\sim}{x}) = \text{Softmax}(W^{(L)} h^{(L-1)} + b^{(L)})$$

then

$$\hat{f}(\underset{\sim}{x}) = \underset{k}{\arg\max}\, O(\underset{\sim}{x})_k$$

NNs generalize many methods:

NNs generalize many methods:

E.g. $L=1$, $g^{(1)}(x)=x$ then

$$\hat{f}(\underset{\sim}{x}) = W^{(1)}\underset{\sim}{x} + b^{(1)}$$

this is just regression.

E.g. $L=1$ and

$$O(\underset{\sim}{x}) = \text{Softmax}(W^{(1)}\underset{\sim}{x} + b^{(1)})$$

$$\hat{f}(\underset{\sim}{x}) = \arg\max_{k} O(\underset{\sim}{x})_p$$

then this is just logistic regression.

For FNNs there are lots of choices in architecture to be made

① how many layers (depth)

② how many hidden units at each

(2) how many hidden units at each layer (width)

(3) which activation functions

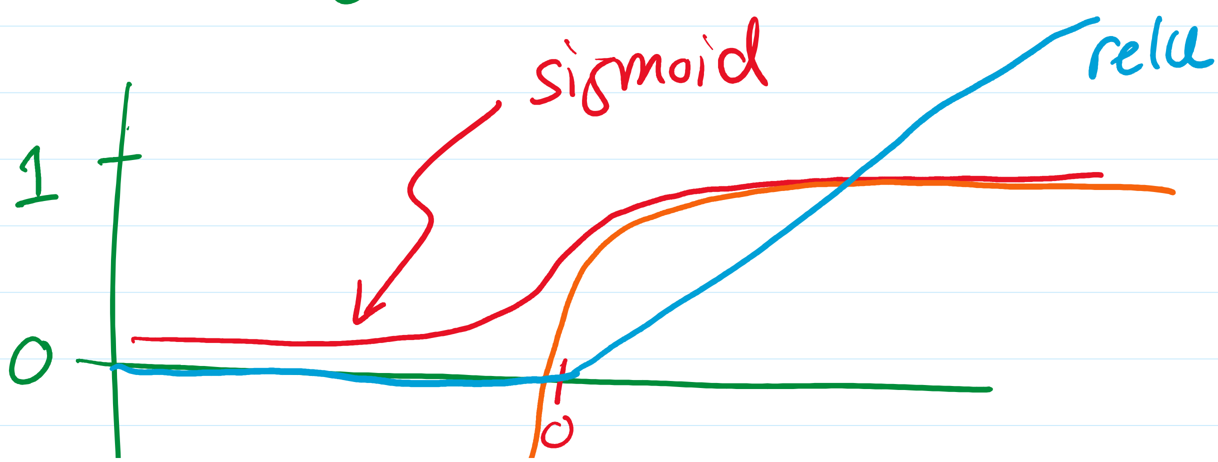
Modern wisdom: deeper networks are better than wider.

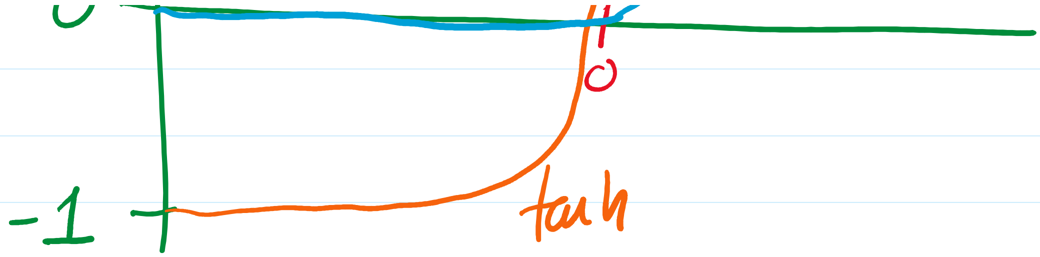
Activation Functions:

Sigmoid: $g(h) = \dfrac{1}{1+e^{-h}}$

tanh: $g(h) = \tanh(h)$

ReLU: $g(h) = \max(0, h)$





How do we learn the params?

Our params:

$$\theta = (W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)}, \dots, W^{(L)}, b^{(L)})$$

this is super high dim'l.

To learn need to specify some loss

$$L(y, \hat{f}_\theta(\underset{\sim}{x}))$$

and then find the value of θ that minimizes the empirical loss

$$\boxed{\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \frac{1}{N} \sum_{n=1}^{N} L(y_n, \hat{f}_\theta(\underset{\sim}{x}_n))}$$

traing data of

traing data of size N

For regression: $L(y,\hat{y}) = (y-\hat{y})^2$

Squared error

K-class classification:

$$O(\underline{x}) = \text{Softmax}(W^{(L)} h^{(L-1)} + b^{(L)})$$

K probs that sum to 1 (for each class)

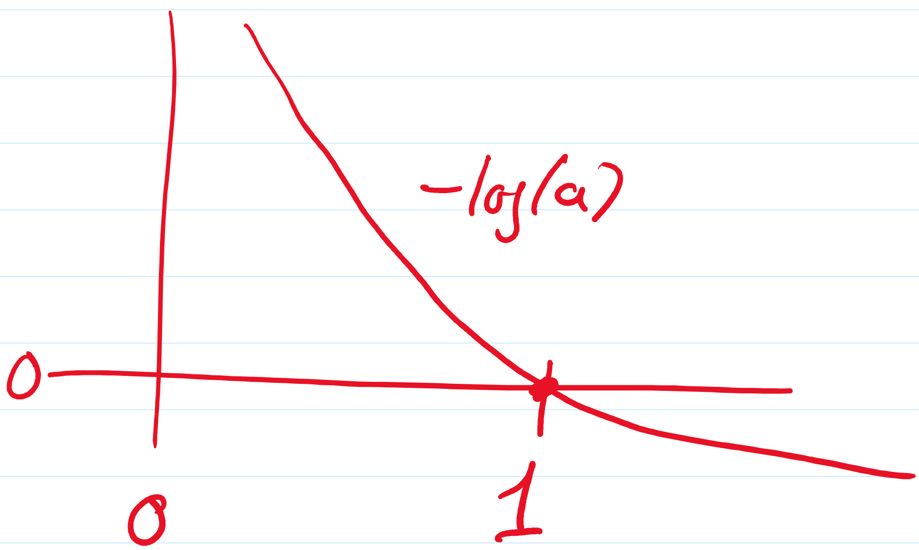
$$\tilde{y}_k = \mathbb{1}(y=k)$$

$$\left[y=2 \quad ; \quad \tilde{y} = (0,1,0,0\ldots) \right]$$

Use Cross-entropy loss:

$$L(y, O(\underline{x})) = -\sum_{k=1}^{K} \tilde{y}_k \log(O(\underline{x})_k)$$

$$= -\log(O(\underline{x})_y)$$



This is difficult b/c

1. θ is super high dim'l
2. may have lots of data
3. $\hat{f}$ is very complex.

<u>To optimize</u> use gradient descent

$$\hat{\ } \quad \min \sum^{N} L(y, \hat{f}(x_i))$$

$$\hat{\theta} = \arg\min_{\theta} \underbrace{\sum_{n=1}^{N} L(y_n, \hat{f}_\theta(x_n))}_{\mathcal{L}(\theta)}$$

<u>Idea:</u> follow the slope of $\mathcal{L}$:

- gradient $\left.\frac{\partial \mathcal{L}}{\partial \theta}\right|_{\theta_0}$ points in dir of fastest ascent at θ_0

- $-\left.\frac{\partial \mathcal{L}}{\partial \theta}\right|_{\theta_0}$ points (locally) in dir to move to decrease $\mathcal{L}$ fastest

<u>Grad Desc:</u>

Initialize $\theta^{(0)}$

For $t = 1, 2, 3, \ldots$

$$\theta^{(t)} = \theta^{(t-1)} - \alpha \left.\nabla_\theta \mathcal{L}\right|_{\theta^{(t-1)}}$$

$$\theta^{(t)} = \theta^{(t-1)} - \underset{\substack{\text{step size} \\ \text{(learning rate)}}}{\alpha} \underbrace{\nabla_\theta \mathcal{L}|_{\theta^{(t-1)}}}_{\text{grad.}}$$

<u>Problem:</u> N may be really large and so Calc. grad may be slow

$$\mathcal{L}(\theta) = \sum_n L(y_n, \hat{f}_\theta(x_n))$$

$$\frac{\partial \mathcal{L}}{\partial \theta} = \sum_n \frac{\partial L_n}{\partial \theta}$$

<u>Soln:</u> <u>Stochastic Grad Descent</u> (SGD)

Idea: instead of calc grad over all trains, we just use a (random) subset.

subset $\rightarrow$ mini-batch