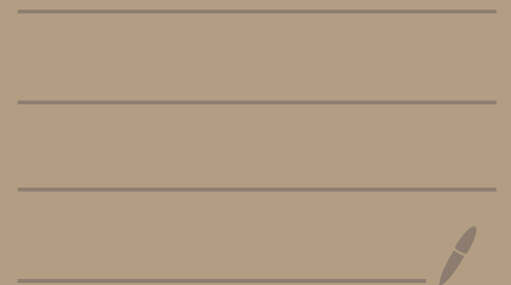


DATA DRIVEN TECHNIQUES

Lecture 7 Unknown Function Minimization



LEARNING ALGORITHMS

①

In previous lectures we considered vector function $H(x, \theta)$ and defined

$$G(\theta) = \mathbb{E}_x [H(x, \theta)] = \int H(x, \theta) f_x(x) dx.$$

where x random vector with density f_x

and proposed iterative schemes that estimate θ_* : $G(\theta_*) = 0$ root of $G(\theta) = 0$ when density $f_x(x)$ is unknown and instead we are given $\{x_1, \dots, x_n\}$ realizations of x .

We proposed two ideas:

1) $G(\theta) \approx \frac{1}{n} \sum_{i=1}^n H(x_i, \theta) = \hat{G}(\theta)$ LLN

and instead of finding the roots of $G(\theta)$ we compute roots of $\hat{G}(\theta)$. Since as $n \rightarrow \infty$ we have $\hat{G}(\theta) \rightarrow G(\theta)$ we expect $\text{roots}(\hat{G}(\theta)) \rightarrow \text{roots}(G(\theta))$.

To find roots we apply

2

$$\hat{\theta}_t = \hat{\theta}_{t-1} + \eta Q \hat{G}(\hat{\theta}_{t-1}) = \hat{\theta}_{t-1} + \eta \frac{1}{n} \sum_{i=1}^n H(x_i, \hat{\theta}_{t-1})$$

with the LLN we learn $G(\theta)$.

When iteration converges to $\hat{\theta}_\infty$ then

$$\frac{1}{n} \sum_{i=1}^n H(x_i, \hat{\theta}_\infty) = \hat{G}(\hat{\theta}_\infty) = 0$$

If $\hat{G}(\theta)$ close to $G(\theta)$ we expect $\hat{\theta}_\infty$ close to θ_* : $G(\theta_*) = 0$.

We showed that $\sqrt{n}(\hat{\theta}_\infty - \theta_*) \sim \mathcal{N}(0, \Sigma)$

The estimation error is asymptotically Gaussian zero mean with a covariance matrix that can be computed directly from $H(x, \theta)$ and the data distribution.

We have a CLT type result for the error.

2) Simplified algorithm

(3)

$$\tilde{\theta}_t = \tilde{\theta}_{t-1} + \epsilon Q H(x_t, \tilde{\theta}_{t-1})$$

This algorithm learns gradually $G(\theta)$ and solves the equation $G(\theta) = 0$

Here as well we can analyse the algorithm and show that

AT STEADY-STATE

$\tilde{\theta}_t - \theta_*$ is zero mean with a

covariance matrix which is of the form $\epsilon \cdot C + o(\epsilon)$.

C can be computed by solving a Lyapunov equation

FUNCTION MINIMIZATION

Let the scalar function $h(x, \theta)$. We define

$$g(\theta) = \mathbb{E}_x [h(x, \theta)] = \int h(x, \theta) p_x(x) dx.$$

which we like to minimize with respect to θ .

④

If density $f_X(x)$ is unknown and instead we are given realizations $\{x_1, x_2, \dots, x_n\}$ of the random vector X then:

$$g(\theta) \approx \frac{1}{n} \sum_{i=1}^n h(x_i, \theta) = \hat{g}(\theta) \quad \theta = \begin{bmatrix} \theta_1 \\ \vdots \\ \theta_L \end{bmatrix}$$

and I can define: gradient descent

$$\theta_t = \theta_{t-1} - \mu \nabla_{\theta} \hat{g}(\theta) \big|_{\theta = \theta_{t-1}} = \theta_{t-1} - \mu \frac{1}{n} \sum_{i=1}^n \nabla_{\theta} h(x_i, \theta) \big|_{\theta = \theta_{t-1}}$$

If I call $H(x, \theta) = \nabla_{\theta} h(x, \theta) = \begin{bmatrix} \frac{\partial h(x, \theta)}{\partial \theta_1} \\ \vdots \\ \frac{\partial h(x, \theta)}{\partial \theta_L} \end{bmatrix}$ then

$$\theta_t = \theta_{t-1} - \mu \frac{1}{n} \sum_{i=1}^n H(x_i, \theta_{t-1}) \quad \text{Gradient descent}$$

computes a (local) minimum of $\hat{g}(\theta) = \frac{1}{n} \sum_{i=1}^n h(x_i, \theta)$ (say $\hat{\theta}$)

But we are interested in finding the minimum of

(5)

$$g(\theta) = \int h(x, \theta) f_a(x) dx. \text{ Call it } \theta_*$$

What is $g(\hat{\theta}_\infty)$? Obviously

$$\text{minimum of } \hat{g}(\theta) \quad g(\theta_\infty) - g(\theta_*) \geq 0 \quad \text{minimum of } g(\theta)$$

We can show that

$$\mathbb{E}[g(\theta_\infty) - g(\theta_*)] = \frac{c}{n}$$

Annotations:
- θ_∞ is labeled "random" (blue arrow).
- θ_* is labeled "deterministic" (blue arrow).
- $\frac{c}{n}$ is labeled "can be computed." (red arrow).

Simplified explanation

$$\theta_t = \theta_{t-1} - \eta \nabla_\theta h(x_t, \theta) |_{\theta = \theta_{t-1}} = \theta_{t-1} - \eta H(x_t, \theta_{t-1})$$

Stochastic Gradient Descent SGD.

(6)

θ_t at steady state is $\theta_t = \theta_* + v_t$ therefore

$$g(\theta_t) - g(\theta_*) \geq 0$$

random ↑ deterministic ↑

$$\mathbb{E}[g(\theta_{\infty}) - g(\theta_*)] = \frac{C}{n}$$

Is this true when training set is finite?

ADAPTIVE ALGORITHMS

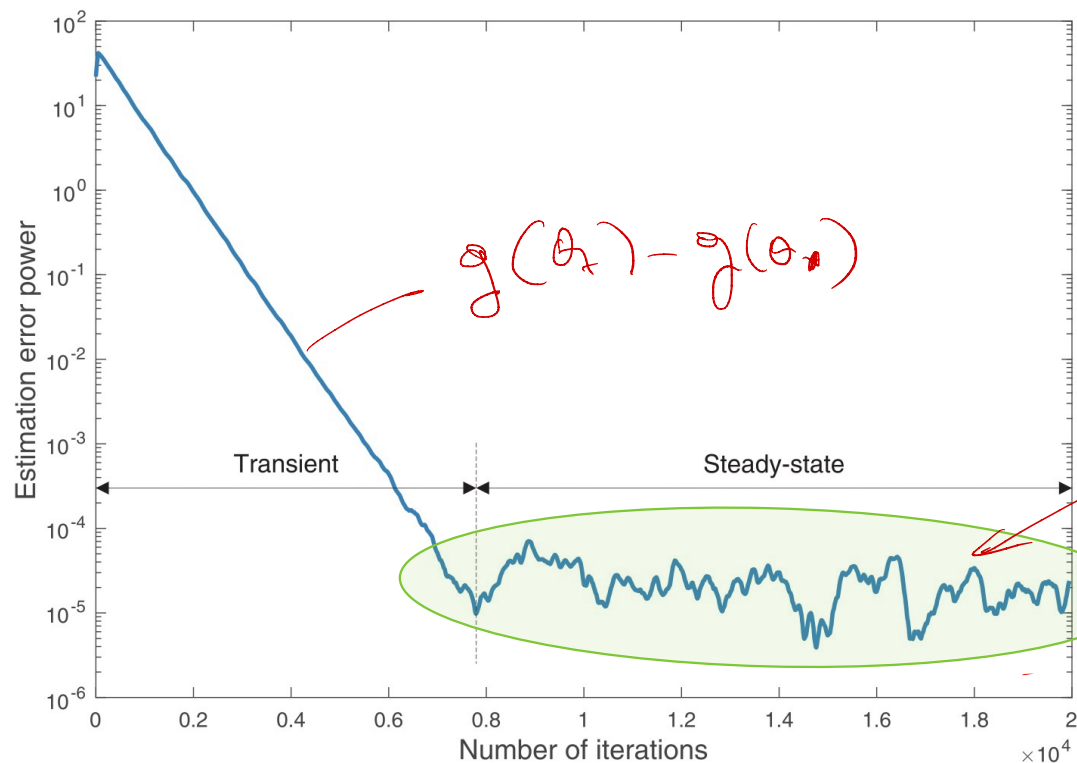
If infinite sequence $\{x_1, x_2, \dots\}$

$$\theta_t = \theta_{t-1} - \mu H(x_t, \theta_{t-1}), \quad \underline{H(x, \theta) = \nabla_{\theta} h(x, \theta)}$$

then we know that at steady-state

$$\theta_t = \theta_* + v_t \quad \text{with} \quad \mathbb{E}[\|v_t\|^2] = \mu \cdot C + o(\mu)$$

If we now follow $g(\theta_t) - g(\theta_*)$



Error I make because $g(\theta)$ is unknown and I continuously learn it.

We previously showed that

$$\lim_{t \rightarrow \infty} \mathbb{E}[\|\theta_t - \theta_*\|^2] = \mu C + \text{negligible}$$

Since (at steady state) $\theta_t = \theta_* + V_t \rightarrow g(\theta_t) = g(\theta_*) + \nabla g(\theta_*) \cdot V_t + \frac{1}{2} V_t^T \nabla^2 g(\theta_*) V_t$

$$\mathbb{E}[g(\theta_t) - g(\theta_*)] = \underbrace{\nabla g(\theta_*)}_{=0} \mathbb{E}[V_t] + \frac{1}{2} \mathbb{E}[V_t^T \underbrace{\nabla^2 g(\theta_*)}_{\text{Hessian positive definite}} V_t] =$$

$$\frac{1}{2} \text{trace}(\underbrace{\nabla^2 g(\theta_*)}_{\mu C + o(\mu)} \mathbb{E}[V_t V_t^T]) = \mu C' + \text{negligible}$$

If I have two algorithms

8

$$\theta_t = \theta_{t-1} + \eta H(x_t, \theta_{t-1})$$

$$v_t = v_{t-1} + \eta \mathcal{H}(x_t, v_{t-1})$$

that can estimate ~~to~~ the minimizer of $g(\theta)$ in order to compare them fairly I need to select η so that at steady-state we have

$$\mathbb{E}[g(\theta_t) - g(\theta^*)] = \mathbb{E}[g(v_t) - g(\theta^*)]$$

and then the faster wins. We achieve this in exactly the same way as the solution of equations

EXAMPLE

θ^* vector of length L .

Random vector process $\{z_1, z_2, \dots\}$, Generate scalar process $\{y_1, y_2, \dots\}$

$$y_t = \theta^{*T} z_t + w_t \leftarrow \text{scalar additive noise independent from } \{z_t\}.$$

At every time t we measure $X_t = \begin{Bmatrix} y_t \\ \mathbf{z}_t \end{Bmatrix}$

9

Define as $g(\theta) = \mathbb{E}[(y - \theta^T \mathbf{z})^2]$

$$y = \theta_*^T \mathbf{z} + w$$

$$g(\theta) = \mathbb{E}_{\mathbf{z}}[(y - \theta^T \mathbf{z})^2] = \mathbb{E}[(w - (\theta - \theta_*)^T \mathbf{z})^2]$$

$$= \mathbb{E}[w^2] + \mathbb{E}[(\theta - \theta_*)^T \mathbf{z} \mathbf{z}^T (\theta - \theta_*)] + 2 \mathbb{E}[(\theta - \theta_*)^T \mathbf{z} \cdot w]$$

because w_t, \mathbf{z}_t independent $\Rightarrow 0$

$$= \sigma_w^2 + (\theta - \theta_*)^T \Sigma_{\mathbf{z}} (\theta - \theta_*)$$

positive definite so
minimum for $\theta = \theta_*$

and $g(\theta_*) = \sigma_w^2$

Possible algorithms

Stochastic Gradient Descent

10

$$h(x, \theta) = (y - \theta^T z)^2 \rightarrow H(x, \theta) = \nabla_{\theta} h(x, \theta) = -2(y - \theta^T z) \cdot z$$

Therefore SGD

$$\theta_t = \theta_{t-1} + 2\eta (y_t - \theta_{t-1}^T z_t) \cdot z_t$$

2L multiplications
2L additions

LMS

EXTREMELY POPULAR IN PRACTICE !!!

Alternative Algorithm

This algorithm estimates θ^* BUT it is NOT the SGD of any optimization problem!

$$\theta_t = \theta_{t-1} + \eta \psi(y_t - \theta_{t-1}^T z_t) z_t$$

$\psi(u)$ is scalar

such that $\mathbb{E}[\psi(w_t)] = 0$.

For example if w_t symmetric a.s. then $\psi(u)$ can be odd-symmetric (like $\psi(u) = \text{sgn}(u)$)

Extension

(11)

We optimize with SGD $g(\theta) = \mathbb{E}_x[h(x, \theta)]$

Suppose I have a deterministic scalar function $R(u_1, u_2)$ and two scalar functions $h_1(x, \theta)$, $h_2(x, \theta)$. Consider

$$g(\theta) = R(\underbrace{\mathbb{E}_x[h_1(x, \theta)]}_{g_1(\theta)}, \underbrace{\mathbb{E}_x[h_2(x, \theta)]}_{g_2(\theta)}) \text{ which we like}$$

to minimize.

We compute the Gradient

$$\begin{aligned} G(\theta) &= \frac{\partial R}{\partial u_1} \nabla_{\theta} \mathbb{E}_x[h_1(x, \theta)] + \frac{\partial R}{\partial u_2} \nabla_{\theta} \mathbb{E}_x[h_2(x, \theta)] = 0 \\ &= \frac{\partial R}{\partial u_1} (\mathbb{E}_x[h_1(x, \theta)], \mathbb{E}_x[h_2(x, \theta)]) \mathbb{E}_x[h_1(x, \theta)] + \\ &\quad \frac{\partial R}{\partial u_2} (\mathbb{E}_x[h_1(x, \theta)], \mathbb{E}_x[h_2(x, \theta)]) \mathbb{E}_x[h_2(x, \theta)] \end{aligned}$$

(12)

$$\theta_t = \theta_{t-1} - \gamma \left\{ \frac{\partial R}{\partial a_t} (A_t, B_t) H_1(x_t, \theta_{t-1}) + \frac{\partial R}{\partial b_t} (A_t, B_t) H_2(x_t, \theta_{t-1}) \right\}$$

where

$A_t = \text{estimate of } \mathbb{E}_x[h_1(x, \theta_{t-1})]$

$B_t = \text{estimate of } \mathbb{E}_x[h_2(x, \theta_{t-1})]$

Possibilities:

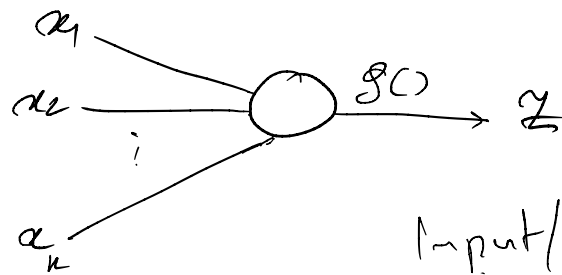
$$A_t = (1-\gamma) A_{t-1} + \gamma h_1(x_t, \theta_{t-1}) = A_{t-1} + \gamma (h_1(x_t, \theta_{t-1}) - A_{t-1})$$

$$B_t = (1-\gamma) B_{t-1} + \gamma h_2(x_t, \theta_{t-1}) = B_{t-1} + \gamma (h_2(x_t, \theta_{t-1}) - B_{t-1})$$

NEURAL NETWORKS

2

Basic element: Neuron



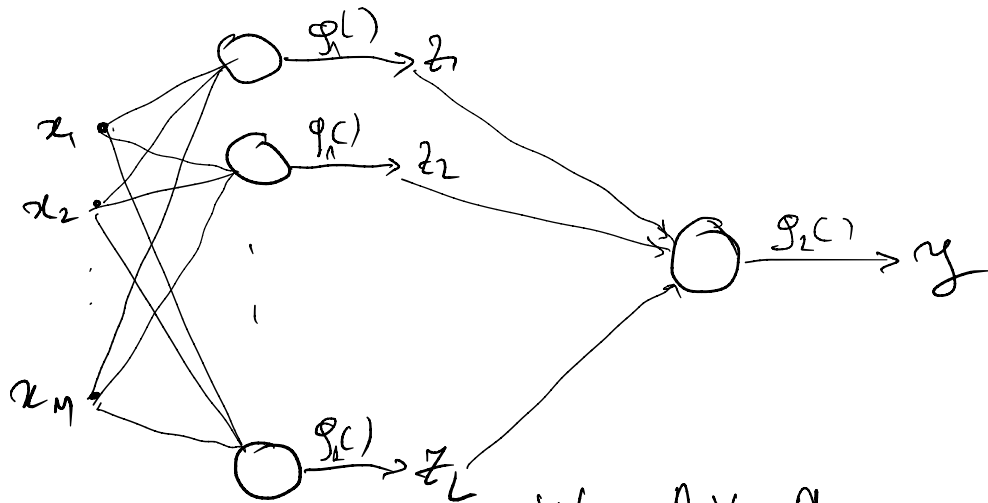
Multiple inputs single output

Input/output relationship: $z = g(\alpha_1 x_1 + \dots + \alpha_n x_n + \alpha_0)$

where $g(\cdot)$ is scalar function called activation function

$\alpha_0, \alpha_1, \dots, \alpha_n$ are the weights and x_1, \dots, x_n the inputs

We can combine multiple neurons



$$W = AX + a$$
$$z = g_1(W)$$

$$\tilde{W} = B^T z + \beta$$
$$y = g_2(\tilde{W}) = \phi(x, \theta) \quad \theta = \{A, a, B, \beta\}$$

when $g(w)$ is scalar with scalar w then if $W = \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix}$ we define $g(W) = \begin{bmatrix} g(w_1) \\ \vdots \\ g(w_n) \end{bmatrix}$

Combine linear functions with simple nonlinearities (activations) and end up with a nonlinear function

2

$\phi(x, \theta)$ collects all parameters of the network.
input

What we just designed is a neural network with a single hidden layer.

What can I represent with $\phi(x, \theta)$; **EVERYTHING !!!**

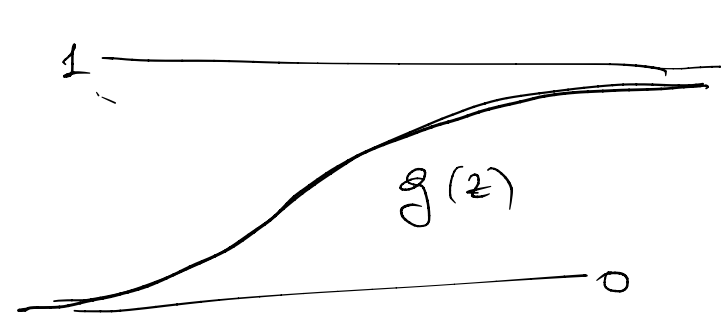
THEOREM (Cybenko 1989) UNIVERSAL APPROXIMATION THEOREM

Any scalar nonlinear function $u(x)$ under proper conditions of smoothness can be approximated **arbitrarily close** by a neural network with a large enough single hidden layer and sigmoid-like activations.

For $\epsilon > 0$ I can design $\phi(x, \theta)$ such that

$$|u(x) - \phi(x, \theta)| \leq \epsilon \text{ for } x \in D \leftarrow \text{Bounded domain (unit cube)}$$

Cybenko's result relies on activation functions of the form (3)

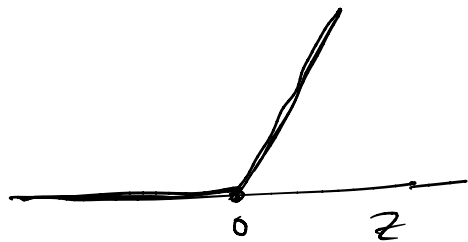


known as sigmoid.

Most popular

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

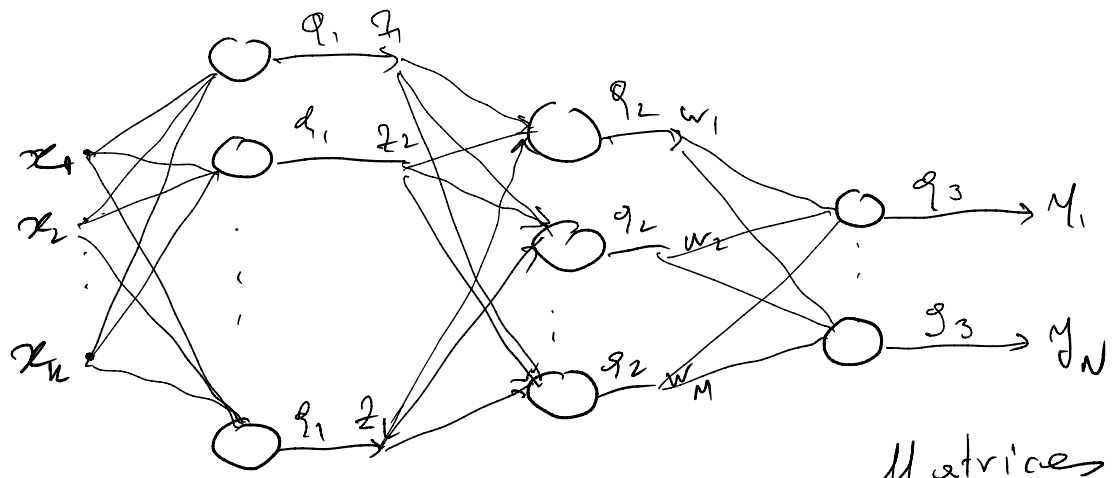
In practice people tend to use instead of the sigmoid the alternative



$$g(z) = \max\{z, 0\}$$

known as ReLU (Rectifier Linear Unit)

Cybenko's work is an "existence" result. It does not describe how to select the parameters. Furthermore instead of a single hidden layer (known as shallow networks) we can use multiple hidden layers.



$\Phi(x, \theta)$ is a neural network with two hidden layers and k inputs / N outputs. Input/output equations are:

$$z = g_1(Ax + a)$$

Matrices

$$\theta = \{A, a, B, \beta, C, \gamma\}$$

vectors

$$W = g_2(Bz + \beta)$$

$$Y = g_3(CW + \gamma)$$

This network is known as "full" because every neuron output becomes input to all neurons of the next layer.

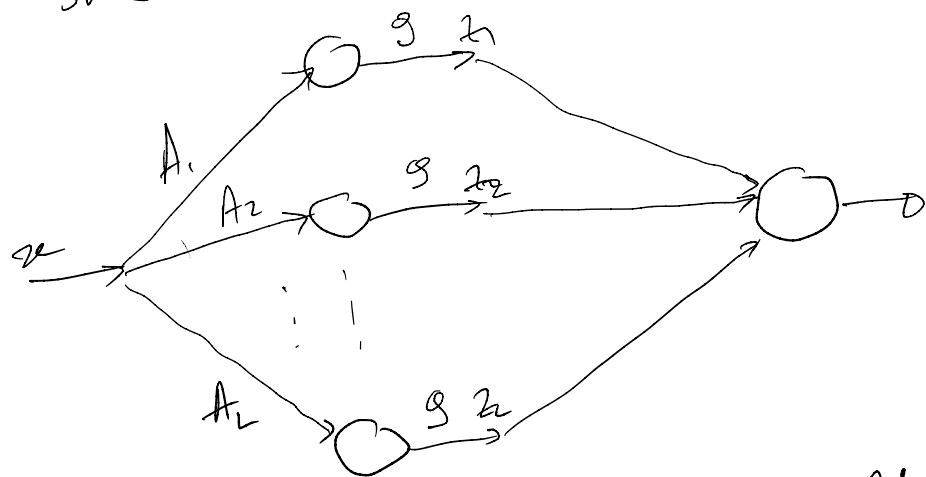
If the number of hidden layers is large the network is called "Deep".

Major Question: Are deep networks more efficient than shallow with 1 or 2 hidden layers? There are articles claiming that this statement is true, but their comparisons are dubious.

Cybenko's Result - Special case

(5)

Suppose we are interested in approximating $W(x)$ with a shallow network when x is scalar.



$$\left. \begin{aligned} z_1 &= g(A_1 x - B_1) \\ &\vdots \\ z_L &= g(A_L x - B_L) \end{aligned} \right\} Z = g(Ax - B)$$

$$y = C_1 g(A_1 x - B_1) + \dots + C_L g(A_L x - B_L) + C_0$$

$g(w)$ is sigmoid. Nice for computations \rightarrow derivatives etc.

But the ideal form of $g(w)$ is unit step



$$u(w) = \begin{cases} 1 & w > 0 \\ \frac{1}{2} & w = 0 \\ 0 & w < 0 \end{cases}$$

The "ideal" version of the output takes the form:

$$y = C_0 + C_1 \mathcal{U}(A_1 x - B_1) + \dots + C_L \mathcal{U}(A_L x - B_L)$$

Assume $A_i \neq 0$ because otherwise the term can be part of C_0 .

Then

$$\begin{aligned} y &= C_0 + C_1 \mathcal{U}\left(\text{sign}(A_1) x - \frac{B_1}{|A_1|}\right) + \dots + C_L \mathcal{U}\left(\text{sign}(A_L) x - \frac{B_L}{|A_L|}\right) \\ &= C_0 + C_1 \mathcal{U}\left(\text{sign}(A_1) \left(x - \frac{B_1}{A_1}\right)\right) + \dots + C_L \mathcal{U}\left(\text{sign}(A_L) \left(x - \frac{B_L}{A_L}\right)\right) \end{aligned}$$

Since $\mathcal{U}(-x) = 1 - \mathcal{U}(x)$ we conclude that

$$y = \hat{C}_0 + \hat{C}_1 \mathcal{U}\left(x - \frac{B_1}{A_1}\right) + \dots + \hat{C}_L \mathcal{U}\left(x - \frac{B_L}{A_L}\right)$$

where $\hat{C}_i = \text{sign}(A_i) C_i$ and $\hat{C}_0 = C_0 + \sum_{i=1}^L C_i \mathbb{1}_{\{A_i < 0\}}$

Without loss of generality assume

$$\frac{B_1}{A_1} < \frac{B_2}{A_2} < \dots < \frac{B_L}{A_L}$$

then

$$\text{for } x < \frac{B_1}{A_1} : y = C_0 = b_0$$

$$\frac{B_1}{A_1} < x < \frac{B_2}{A_2} : y = C_0 + C_1 = b_1$$

$$\frac{B_{L-1}}{A_{L-1}} < x < \frac{B_L}{A_L} : y = C_0 + C_1 + \dots + C_{L-1} = b_{L-1}$$

$$\frac{B_L}{A_L} < x : y = C_0 + C_1 + \dots + C_{L-1} + C_L = b_L$$

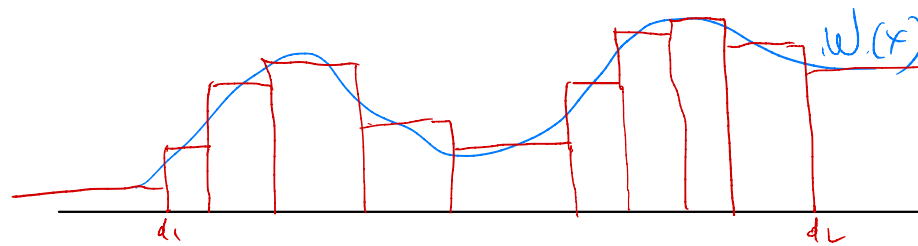
Piece-wise

constant functions

In other words network output can be equivalently written by using a different parametrization

(7)

$$y = b_0 \mathbb{1}_{\{x < d_1\}} + b_1 \mathbb{1}_{\{d_1 < x < d_2\}} + \dots + b_{L-1} \mathbb{1}_{\{d_{L-1} < x < d_L\}} + b_L \mathbb{1}_{\{d_L < x\}}$$



$$d_1 < d_2 < \dots < d_L$$

$$d_i \sim \frac{B_i}{A_i}$$

With such functions we can approximate arbitrarily close $W(x)$

For example with MSE

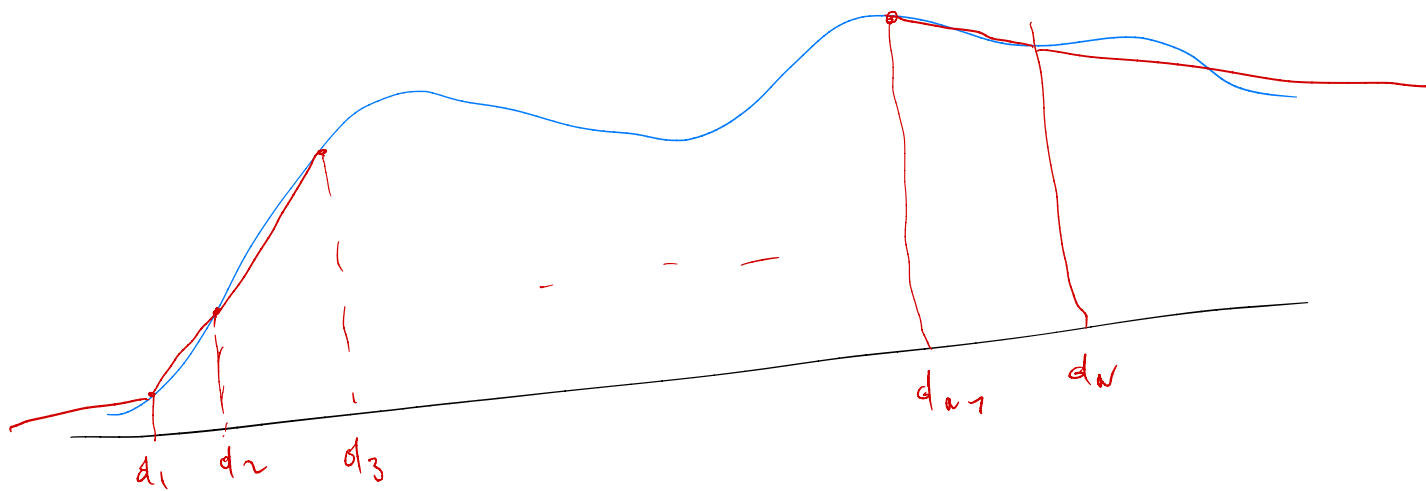
$$\min_{b_0, \dots, b_L, d_1, \dots, d_L} \int (W(x) - \sum b_i \mathbb{1}_{\{d_i < x < d_{i+1}\}})^2 f(x) dx.$$

Always possible

Shallow network is a different parametrization of piece-wise constant function

If we use a ReLU in the output of each neuron then
 shallow networks can be shown to be equivalent
 to piece-wise linear approximation.

(8)



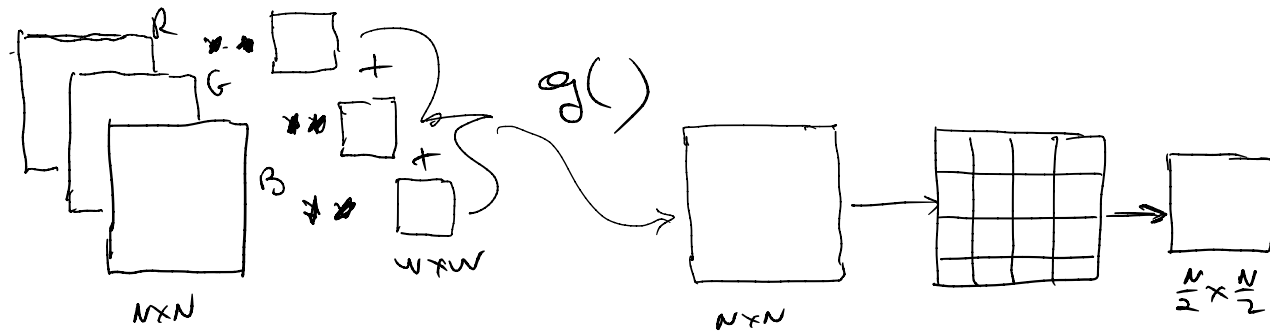
$$y(x) = \sum_{j=1}^n b_j (x - d_j)^+ \mathbb{1}_{\{d_j < x < d_{j+1}\}}$$

$$\min_{b_1, \dots, b_n, d_1, \dots, d_n} \int \left(w(x) - \sum_{j=1}^n b_j (x - d_j)^+ \mathbb{1}_{\{d_j < x < d_{j+1}\}} \right)^2 f(x) dx$$

There are other classes beside the full neural network.

Very popular Convolutional neural networks capable of identifying "features" in the data.

Mostly applied to 2D information. Assume a colored image of dimension $N \times N$ with R, G, B chromatic components



For each chromatic component I select a filter of dimensions $w \times w$ and apply 2D convolution. The result is an image of

dimension $N \times N$ for each component. We add

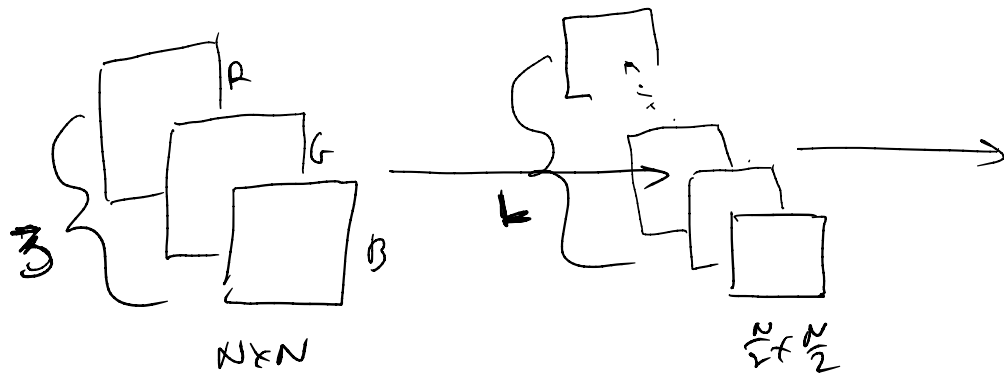
these images and generate a single $N \times N$ image. Then we apply an activation function $g()$ on each element (pixel) of the $N \times N$ image.

The three (R, G, B) images $N \times N$ after applying three filters $w \times w$ generate a single $N \times N$ image. This image is divided into 2×2 blocks and the four elements of each block are replaced by a single element. This element could be the average of the four elements or the maximal element or some other selection. This operation will result in an image of dimensions $\frac{N}{2} \times \frac{N}{2}$.

I can now replace the initial three $w \times w$ filters with another collection of three filters and generate a second image of dimensions $\frac{N}{2} \times \frac{N}{2}$.

and I can repeat this as many times as I want (say L)

10



In the next stage we will apply the same idea. Only now since we have L images $\frac{N}{2} \times \frac{N}{2}$ we will need L filters of dimension $w \times w$ to generate an image of dimensions $\frac{N}{4} \times \frac{N}{4}$ with the subsampling

method we described. For the next stage (layer) we will generate $L' > L$ images of dimensions $\frac{N}{4} \times \frac{N}{4}$.

We continue this process until we end up in the final layer with many "images" of dimensions 1×1 , namely a vector. We finally ^{linearly} combine the vector elements and apply an activation function resulting in a scalar function $\phi(R, G, B; \theta)$

where R, G, B is the "input" the original image and θ contains all the filter parameters.

We can generate different ϕ functions when we would like to have an output which is a vector....

What is a neural network for us? a function of the form $\phi(x, \theta)$ that describes whatever configuration we like. (11)

We know (at least for shallow networks) that if we select $\phi(x, \theta)$ sufficiently large then we can approximate arbitrarily close any function $u(x)$.

How do we select θ ? How do we design neural networks?

By solving optimization problem with respect to θ .

In other words we select a function $g(\theta)$ which when we minimize, we select θ optimally. In all existing applications we have a known function $h(x, z)$ and the function we like to minimize is

$$g(\theta) = \mathbb{E}_x [h(x, \phi(x, \theta))]$$

But instead of the density $f(x)$ we have training data,

x_1, x_2, \dots, x_n . This is where neural networks and previous algorithms meet !!!