Applied Machine Learning

Multilayer Perceptron

Reihaneh Rabbany

McGill
School of Computer Science

Learning objectives

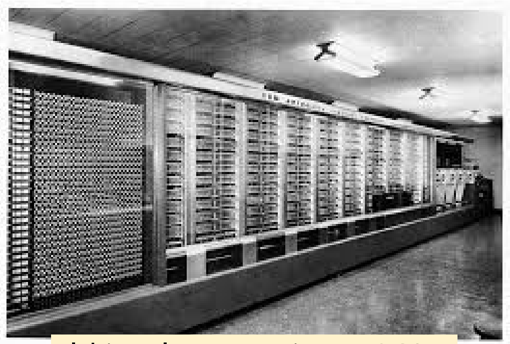
perceptron:

- model, objective, optimization

multilayer perceptron:

- model
  - different supervised learning tasks
  - activation functions
  - architecture of a neural network
- regularization techniques

Perceptron



old implementation (1960's)

historically a significant algorithm

(first neural network, or rather just a neuron)

biologically motivated model

simple learning algorithm

convergence proof

beginning of *connectionist* AI

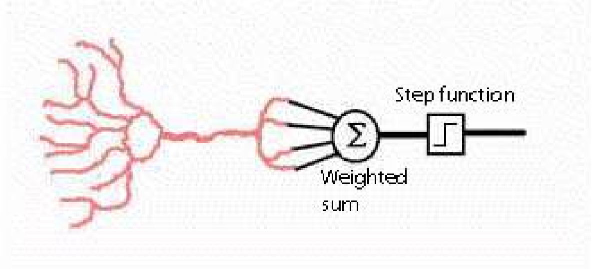
it's criticism in the book "Perceptrons" was a factor in AI winter

Model $f(x) = \mathrm{sign}(w^\top x + w_0)$

compare with models for linear and logistic regression:

$f(x) = w^\top x + w_0$

$f(x) = \sigma(w^\top x + w_0)$



note that we're using +1/-1 for labels rather than 0/1.

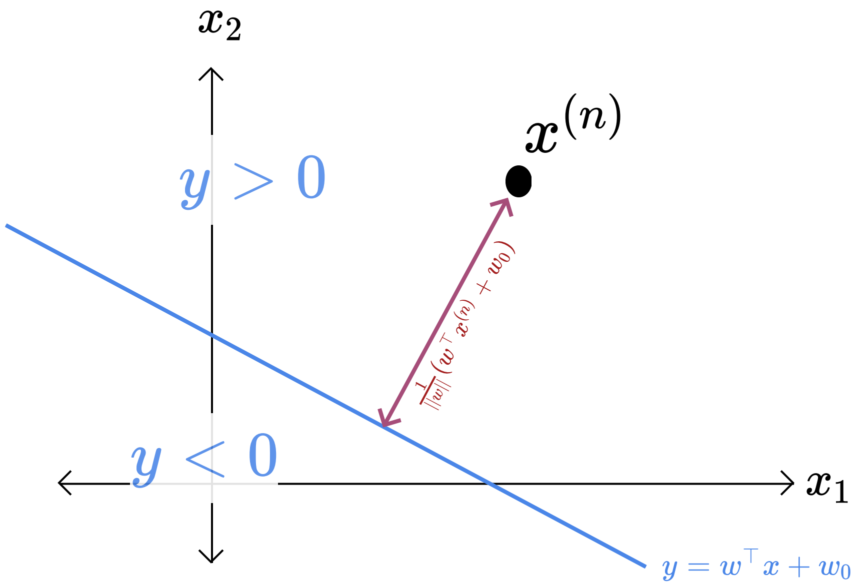
image:https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/Neuron/index.html

Perceptron: objective

$$\hat{y}^{(n)} = \text{sign}(w^\top x^{(n)} + w_0)$$

misclassified if $y^{(n)} \hat{y}^{(n)} < 0$, try to make it positive

label and prediction have different signs



$$\text{minimize } \underline{-y^{(n)} \overbrace{(w^\top x^{(n)} + w_0)}^{\longleftarrow \hat{y}^{(n)} = \text{sign}(\downarrow) \longrightarrow}}$$

this is positive for points that are on the wrong side, minimize it and push them to the right side

Perceptron: optimization

if $y^{(n)}\hat{y}^{(n)} < 0$ minimize $J_n(w) = -y^{(n)}(w^\top x^{(n)})$

now we included bias in w

otherwise, do nothing

use stochastic gradient descent $\nabla J_n(w) = -y^{(n)} x^{(n)}$

$$w^{\{t+1\}} \leftarrow w^{\{t\}} - \alpha \nabla J_n(w) = w^{\{t\}} + \alpha\, y^{(n)} x^{(n)}$$

Perceptron uses learning rate of 1

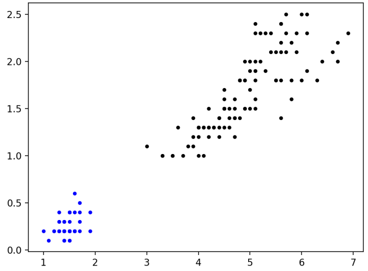
this is okay because scaling w does not affect prediction

$$\mathrm{sign}(w^\top x) = \mathrm{sign}(\alpha\, w^\top x)$$

Perceptron convergence theorem

the algorithm is guaranteed to converge in finite steps if linearly separable

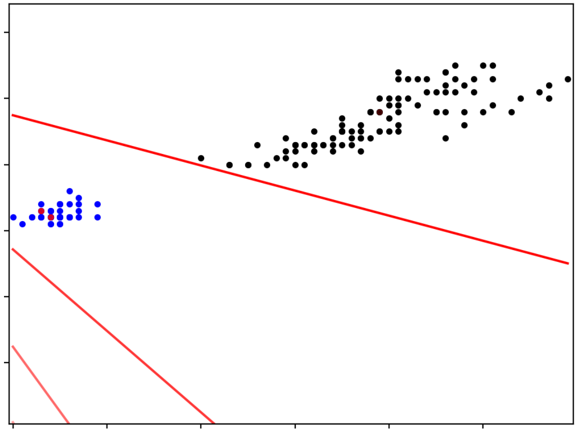
Perceptron: example



Iris dataset

(linearly separable case)

iteration 10



```
N,D = x.shape
w = np.random.rand(D)
for t in range(max_iters):
    n = np.random.randint(N)
    yh = np.sign(np.dot(x[n,:], w))
    if yh != y[n]:
            w = w + y[n]*x[n,:]
```

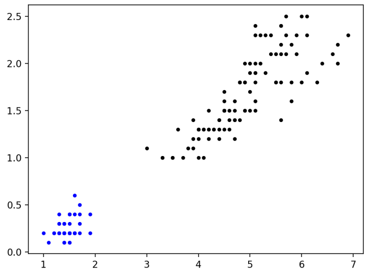
note that the code is not chacking for convergence

observations:

after finding a linear separator no further updates happen

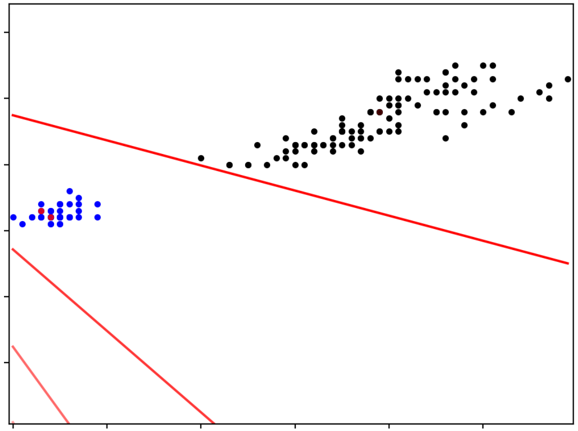
the final boundary depends on the order of instances
(different from all previous methods)

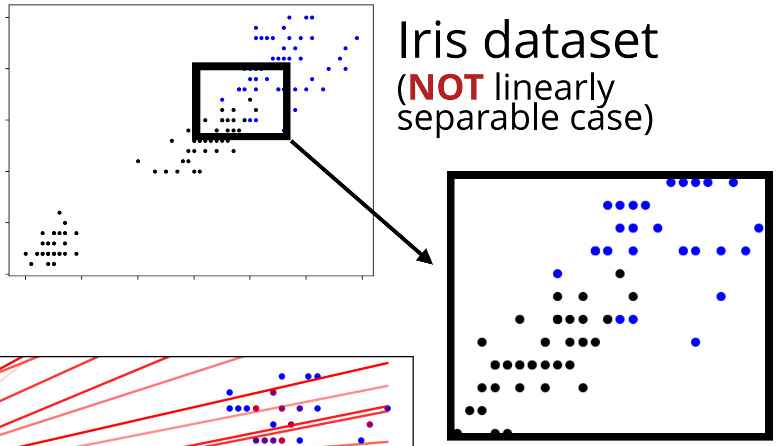
Perceptron: example



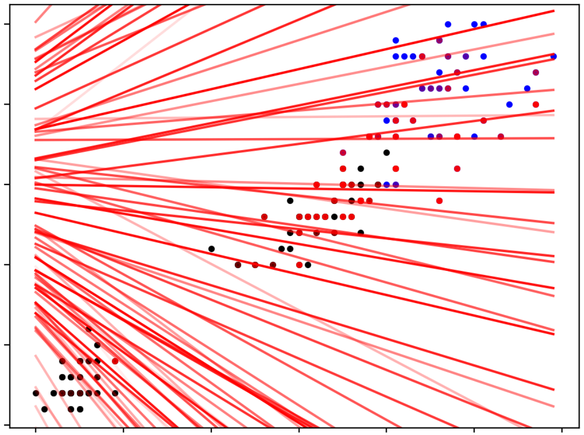
Iris dataset
(linearly separable case)

converged at iteration 10





Iris dataset
(**NOT** linearly separable case)

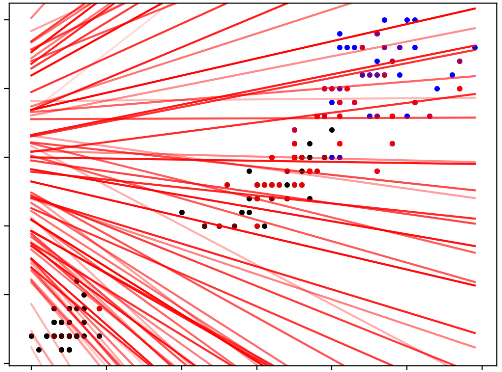


the algorithm does not converge

there is always a wrong prediction and the weights will be updated

Building more expressive model

Perceptron is not expressive enough, can not model the data that is not linearly separable (gets stuck in cyclic updates)

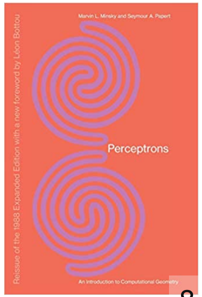


how to increase the model's expressiveness?

use **fixed** nonlinear bases: similar to what we have seen before

use **adaptive** bases: learn the parameters of the bases as well

- e.g., in regression $f(x) = \sum_m w_m \phi_m(x; v_m)$



There is an influential book on the limitations of the perceptrons, see here

Adaptive Gaussian Bases

example

input has one dimension (D=1)

non-adaptive case

model: $f(x; w) = \sum_m w_m \phi_m(x)$

cost: $J(w) = \frac{1}{2} \sum_n (f(x^{(n)}; w) - y^{(n)})^2$

the model is linear in its parameters

the cost is convex in w

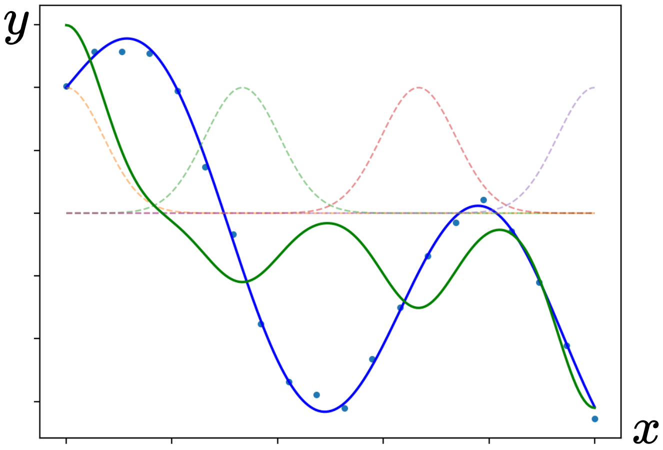
adaptive case

we can make the bases adaptive by learning the *centers*

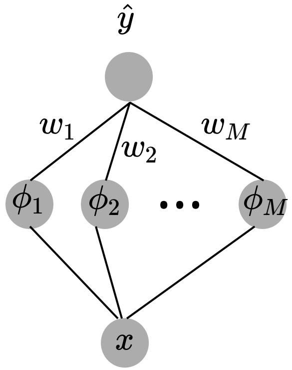
model: $f(x; w, \mu) = \sum_m w_m \phi_m(x; \mu_m)$

not convex in all model parameters

use gradient descent to find a local minimum

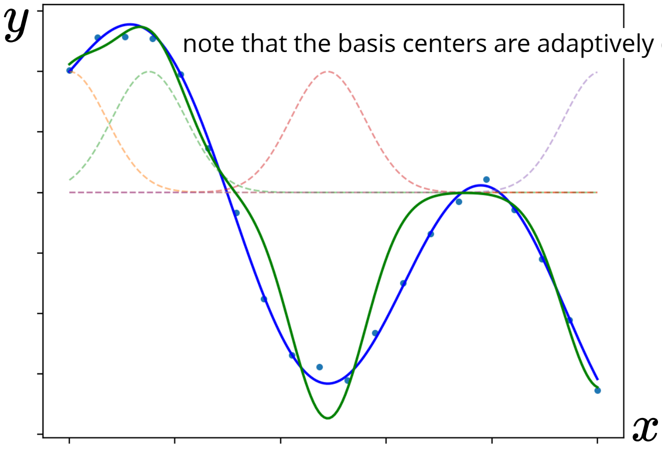


we have seen this before, centers (μ_m) are fixed



$\phi_m(x) = e^{-\frac{(x-\mu_m)^2}{s^2}}$

$s = 1$



adaptive case gives a better fit with the same number of bases (4)

Adaptive Sigmoid Bases

example input has one dimension (D=1)

non-adaptive case

model: $f(x; w) = \sum_m w_m \phi_m(x)$

cost: $J(w) = \frac{1}{2} \sum_n (f(x^{(n)}; w) - y^{(n)})^2$

the model is linear in its parameters

the cost is convex in w

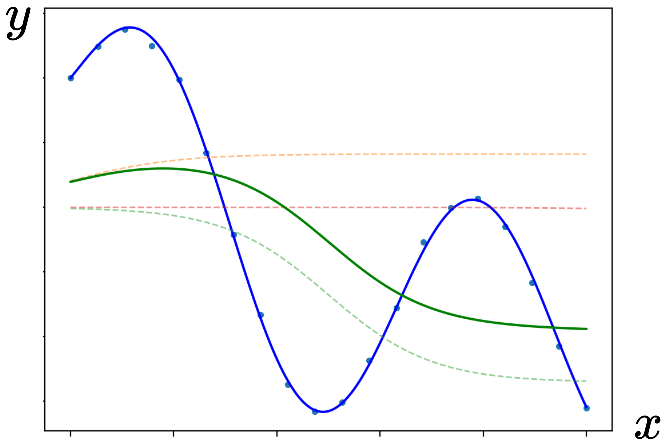
adaptive case

rewrite the sigmoid basis

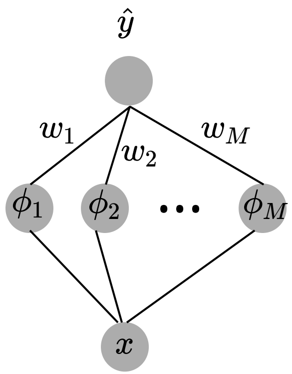
$$\phi_m(x) = \sigma(\frac{x-\mu_m}{s_m}) = \sigma(v_m x + b_m)$$

model: $f(x; w, v, b) = \sum_m w_m \sigma(v_m x + b_m)$

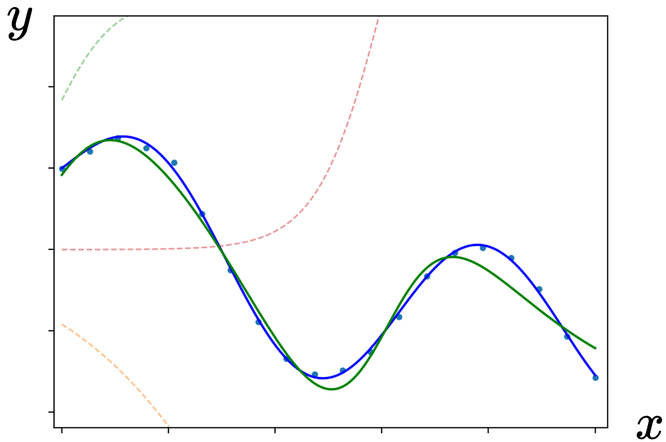
optimize using gradient descent (find a local optima)



we have seen this before, centers (μ_m) are fixed



$$\phi_m(x) = \frac{1}{1+e^{-(\frac{x-\mu_m}{s_m})}}$$



adaptive case gives a better fit with the same number of bases (3)

Adaptive Sigmoid Bases: General Case

this is a **neural network** with two layers!!

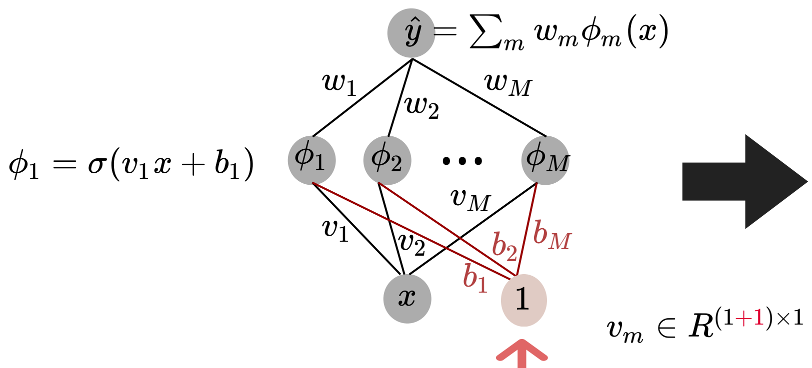
each basis is the logistic regression model

$$\phi_m(x) = \sigma(v_m^\top x + b_m) \quad \forall m$$

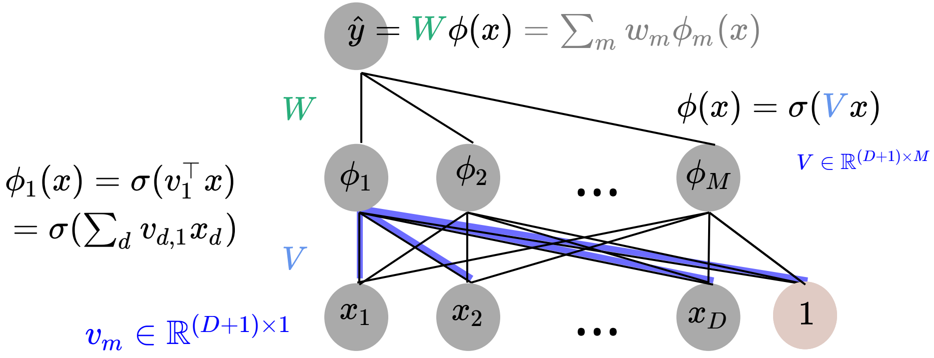
optimize V, W using gradient descent (find a local optima)

input has 1 dimension

input has D dimension



this is the same as having a bias for each nonlinear basis



Multilayer Perceptron (MLP)

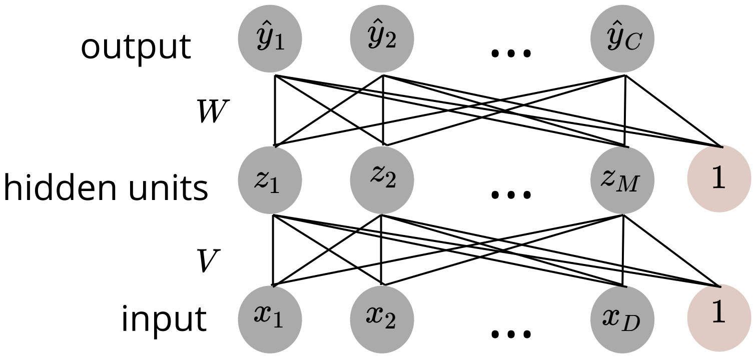
suppose we have

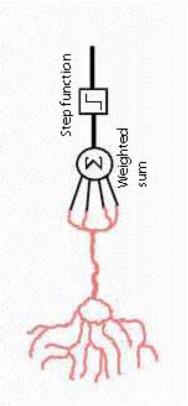
- D inputs $x_1, \ldots, x_D$
- C outputs $\hat{y}_1, \ldots, \hat{y}_C$
- M hidden *units* $z_1, \ldots, z_M$

model

$$\hat{y}_c = g\left(\sum_m W_{c,m} h\left(\sum_d V_{m,d} x_d \right) \right)$$

nonlinearity, activation function: we have different choices





for simplicity we may drop bias terms

more compressed form

$$\hat{y} = g(W\, h(V\, x))$$

non-linearities are applied elementwise

$x \in \mathbb{R}^{D \times 1}$

$V \in \mathbb{R}^{M \times D}$

$Z = h(Vx) \in \mathbb{R}^{M \times 1}$

$W \in \mathbb{R}^{C \times M}$

$y \in \mathbb{R}^{C \times 1}$

Regression using Neural Networks

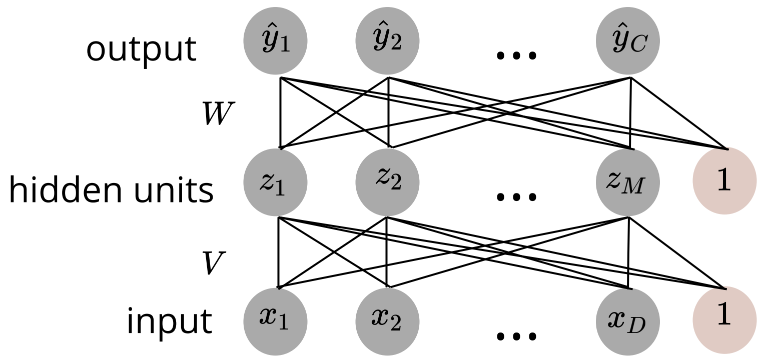
the choice of **activation function** in the final layer depends on the task

model $\hat{y} = g\big(W\,h(V\,x)\big)$

regression $\hat{y} = g(\,W z\,) = W z$

- we may have one or more output variables
- no activation (identity function)
- L2 loss = Gaussian likelihood

$$L(y,\hat{y}) = \tfrac{1}{2}||y-\hat{y}||_2^2 = -\log \mathcal{N}(y;\hat{y},\mathbf{I}) + \text{constant}$$



more generally

we may explicitly produce a distribution at output - *e.g.*,

- mean and variance of a Gaussian
- the loss will be the log-likelihood of the data under our model

$$L(y,\hat{y}) = \log p(y; f(x))$$

neural network outputs the parameters of a distribution

Classification using neural networks

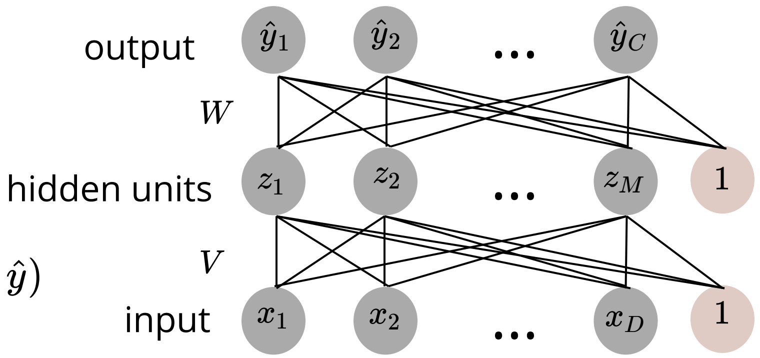
the choice of activation function in the final layer depends on the task

model $\hat{y} = g\big(W\, h(V\, x)\big)$

binary classification $\hat{y} = g(Wz) = \frac{1}{1+e^{-Wz}}$

- scalar output C=1
- activation function is logistic sigmoid
- CE loss = Bernoulli likelihood

$$L(y, \hat{y}) = -y \log \hat{y} - (1-y)\log(1-\hat{y}) = -\log \mathrm{Bernoulli}(y; \hat{y})$$



multiclass classification $\hat{y} = g(Wz) = \mathrm{softmax}(Wz)$

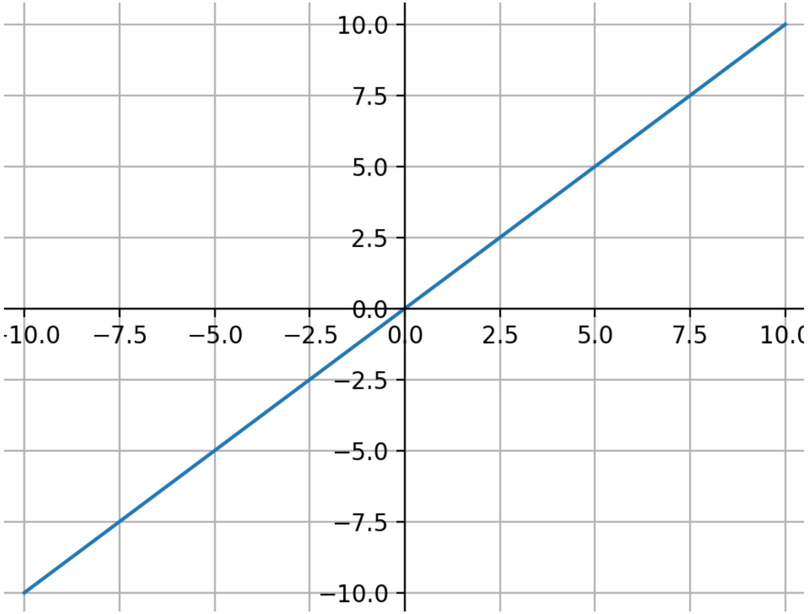
C is the number of classes

softmax activation

multi-class cross entropy loss = categorical likelihood $L(y, \hat{y}) = -\sum_k y_k \log \hat{y}_k = -\log \mathrm{Categorical}(y; \hat{y})$

Activation function

for **middle layer(s)** there is more freedom in the choice of activation function



$h(x) = x$ **identity** (no activation function)

composition of two linear functions is linear

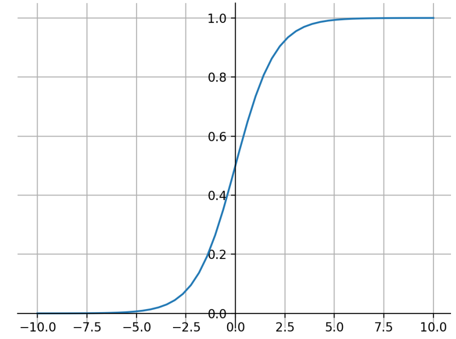
$$\underbrace{\overset{C \times M}{W} \overset{M \times D}{V}}_{W'} x = \overset{C \times D}{W'} x$$

so nothing is gained (in representation power) by stacking linear layers

exception: if $M < \min(D, C)$ then the hidden layer is compressing the data (W' is low-rank)

Activation function

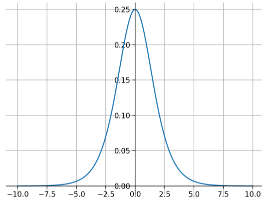
for **middle layer(s)** there is more freedom in the choice of activation function

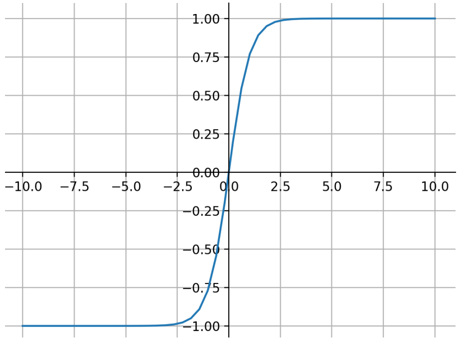


$$h(x) = \sigma(x) = \frac{1}{1+e^{-x}}$$ **logistic function**

the same function used in logistic regression
used to be the function of choice in neural networks
away from zero it changes slowly, so the derivative is small (leads to vanishing gradient)
its derivative is easy to remember

$$\frac{\partial}{\partial x}\sigma(x) = \sigma(x)(1-\sigma(x))$$

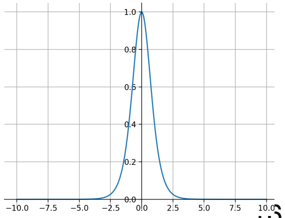




$$h(x) = 2\sigma(x) - 1 = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$ **hyperbolic tangent**

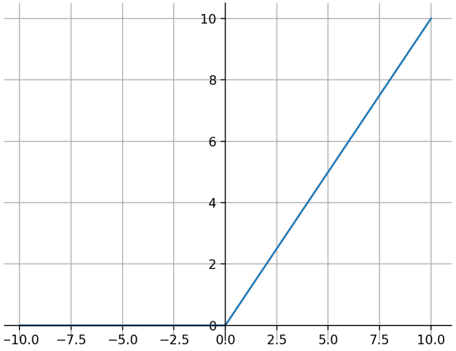
similar to sigmoid, but symmetric
often better for optimization because close to zero it
similar to a linear function (rather than an affine function when using logistic)
similar problem with vanishing gradient

$$\frac{\partial}{\partial x}\tanh(x) = 1 - \tanh(x)^2$$



Activation function

for **middle layer(s)** there is more freedom in the choice of activation function

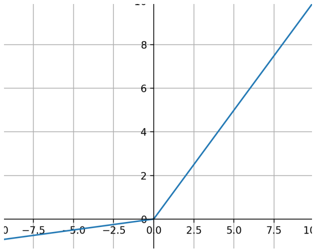


$h(x) = \max(0, x)$ Rectified Linear Unit (**ReLU**)

replacing logistic with ReLU significantly improves the training of deep networks

zero derivative if the unit is "inactive"
initialization should ensure active units at the beginning of optimization

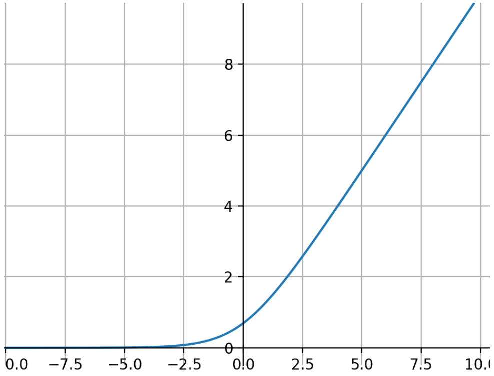
leaky ReLU $h(x) = \max(0, x) + \gamma \min(0, x)$



fixes the zero-gradient problem

parameteric ReLU:
make γ a learnable parameter

Softplus (differentiable everywhere) $h(x) = \log(1 + e^x)$



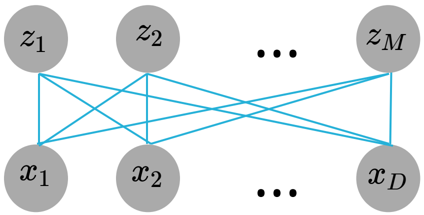
it doesn't perform as well in practice

Network architecture

architecture is the overall structure of the network

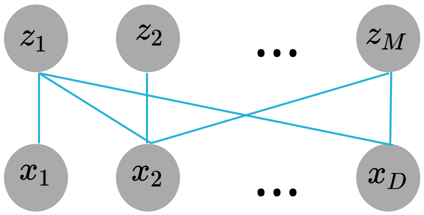
feedforward network (aka multilayer perceptron)

- can have many layers
- # layers is called the **depth** of the network
- each layer can be **fully connected** (dense) or sparse

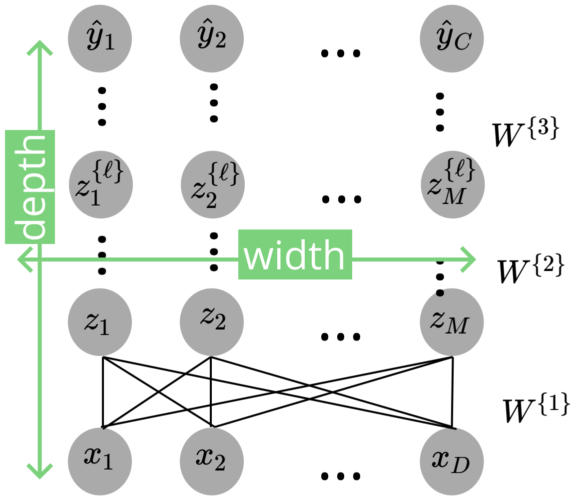


fully connected

all outputs of one layer's units are input to **all** the next units



sparsely connected



$$z^{\{l\}} = h\left(W^{\{l\}} z^{\{l-1\}}\right)$$

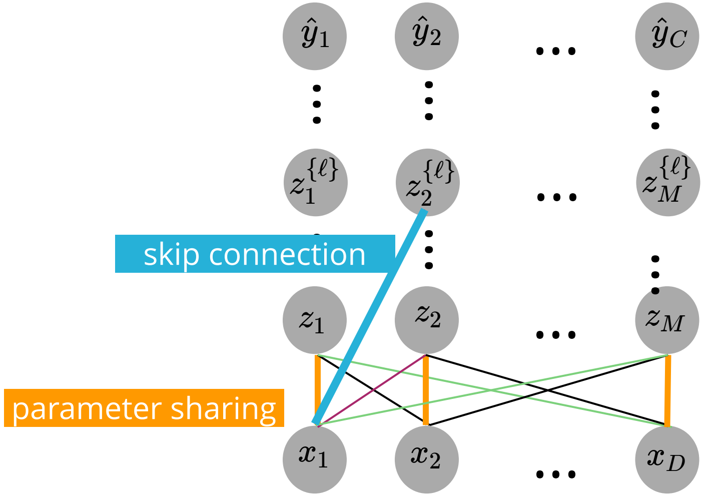
output of one layer is input to the next

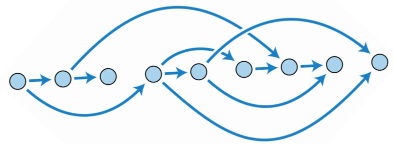
Network architecture

architecture is the overall structure of the network

feed-forward network (aka multilayer perceptron**)**

- can have many layers
- # layers is called the **depth** of the network
- each layer can be **fully connected** (dense) or sparse
- layers may have **skip layer connections**
- units may have different **activations**
- parameters may be shared across units (*e.g.*, in conv-nets)



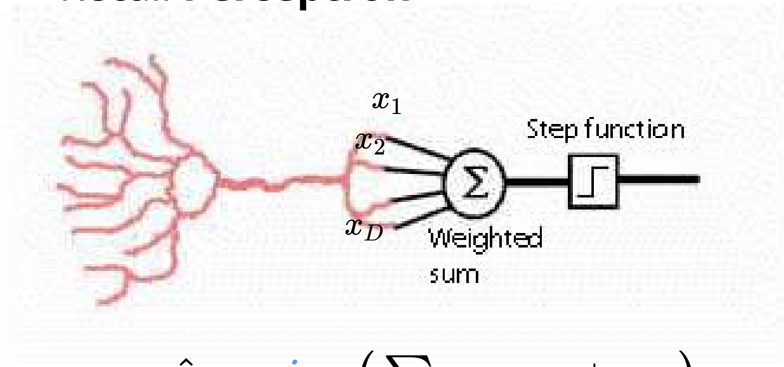


more generally a directed acyclic graph (DAG) expresses the feed-forward architecture

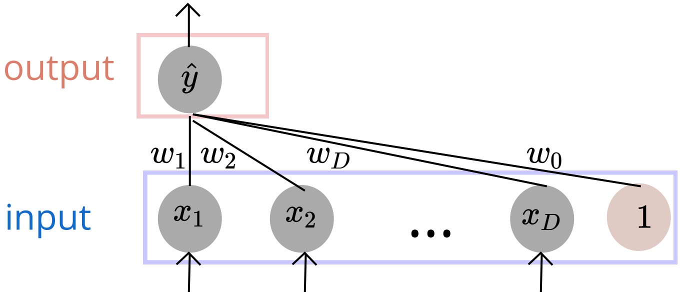
Multilayer Perceptron

Example

Recall **Perceptron**

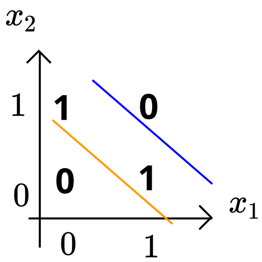


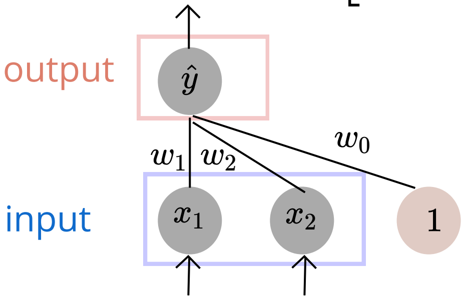
$$\hat{y} = sign\left(\sum_d w_d x_d + w_0\right)$$



$$\hat{y} = \text{sign}(w^\top x + w_0)$$

$$X = \begin{bmatrix} -x^{(1)\top}- \\ -x^{(2)\top}- \\ -x^{(3)\top}- \\ -x^{(4)\top}- \end{bmatrix} = \begin{array}{cc} x_1 & x_2 \\ \end{array} \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \quad y = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$





$w_0 = 0$

$$w^\top x^{(i)}_{i\in[1..4]**} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 2 \end{bmatrix}$$

$w_0 = -1$

$$w^\top x - 1 = \begin{bmatrix} -1 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

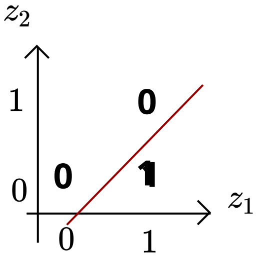
$$w = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$$sign^h(w^\top x) = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \end{bmatrix} \quad sign^h(w^\top x - 1) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

$$sign^h(x) = \mathbb{I}(x > 0)$$

Heaviside sign function, which is 0 for 0 and negative values

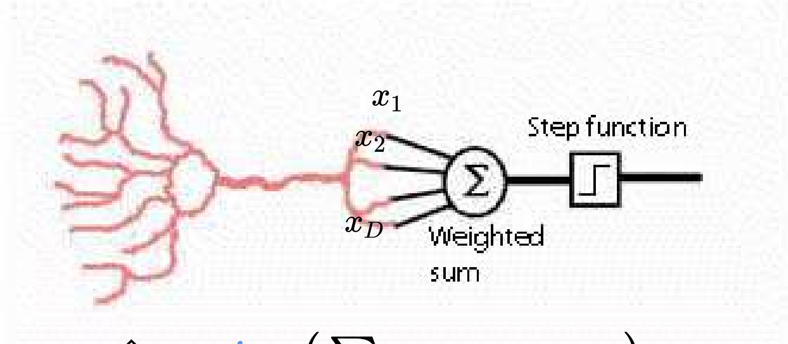
** we drop this for simplicity, it is similar to $X^\top W$, since $w^\top x$ is for one instance, however we use them interchangably to show an affine function of input instances



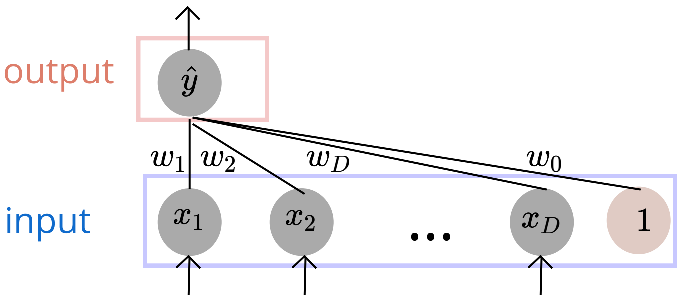
$$z = \begin{array}{cc} z_1 & z_2 \\ \end{array} \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \quad y = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Text

Multilayer Perceptron

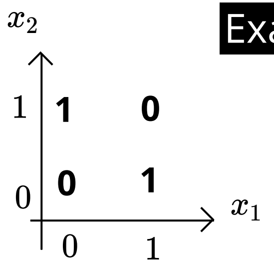


$$\hat{y} = sign\left(\sum_d w_d x_d + w_0\right)$$



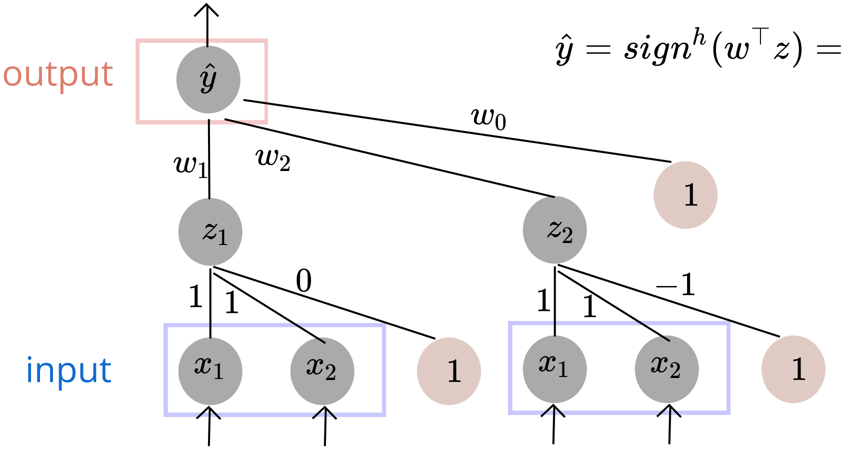
$$\hat{y} = \mathrm{sign}(w^\top x + w_0)$$

Example



$$z = \begin{array}{cc} z_1 & z_2 \\ \multicolumn{2}{c}{\begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}} \end{array} \qquad w = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = \begin{bmatrix} 1 \\ -2 \end{bmatrix} \qquad w^\top z = \begin{bmatrix} 0 \\ 1 \\ 1 \\ -1 \end{bmatrix}$$

$$\hat{y} = sign^h(w^\top z) = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$



Multilayer Perceptron

Example

$$V = \begin{bmatrix} 0 & 1 & 1 \\ -1 & 1 & 1 \end{bmatrix} \qquad x = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \qquad Vx = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

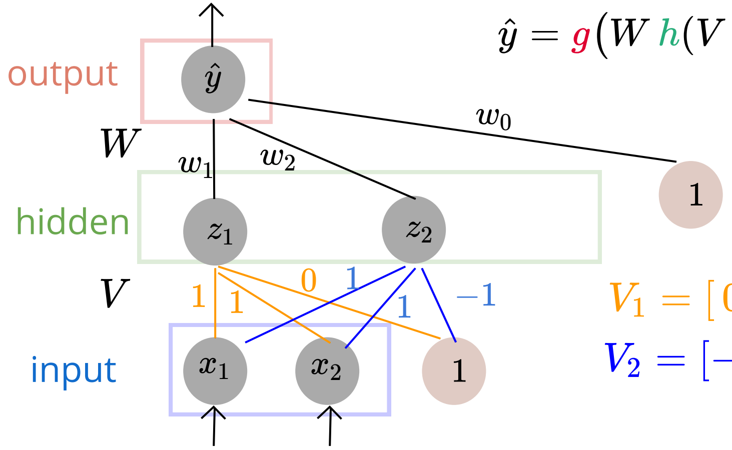
$$W = [0, 1, -2]$$

$$h(Vx) = \begin{array}{cc} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} & \begin{array}{c} \\ z_1 \\ z_2 \end{array} \end{array}$$

$$\hat{y} = g\big(W\, h(V\, x)\big)$$

$$W h(Vx) = -1$$

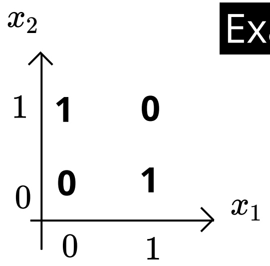
$$\hat{y} = g\big(W\, h(V\, x)\big) = 0$$



$$V_1 = [0, 1, 1]$$

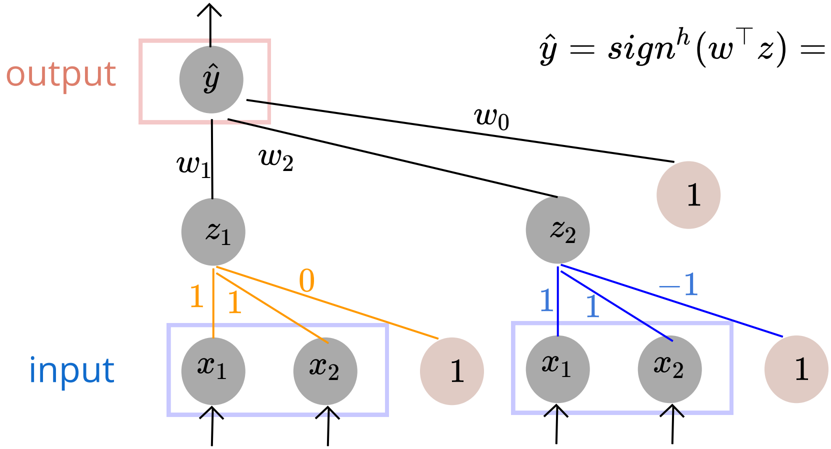
$$V_2 = [-1, 1, 1]$$

$$\begin{bmatrix} -x^{(1)\top}- \\ -x^{(2)\top}- \\ -x^{(3)\top}- \\ -x^{(4)\top}- \end{bmatrix} = \begin{array}{c} \begin{array}{cc} x_1 & x_2 \end{array} \\ \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \end{array} \qquad y = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$



$$z = \begin{array}{c} \begin{array}{cc} z_1 & z_2 \end{array} \\ \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \end{array} \qquad w = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = \begin{bmatrix} 1 \\ -2 \end{bmatrix} \qquad w^\top z = \begin{bmatrix} 0 \\ 1 \\ 1 \\ -1 \end{bmatrix}$$

$$\hat{y} = sign^h(w^\top z) = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$



Multilayer Perceptron

Example

$$\hat{y} = g\big(W\, h(V\, x)\big)$$

$$V = \begin{bmatrix} 0 & 1 & 1 \\ -1 & 1 & 1 \end{bmatrix} \qquad x = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \qquad Vx = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

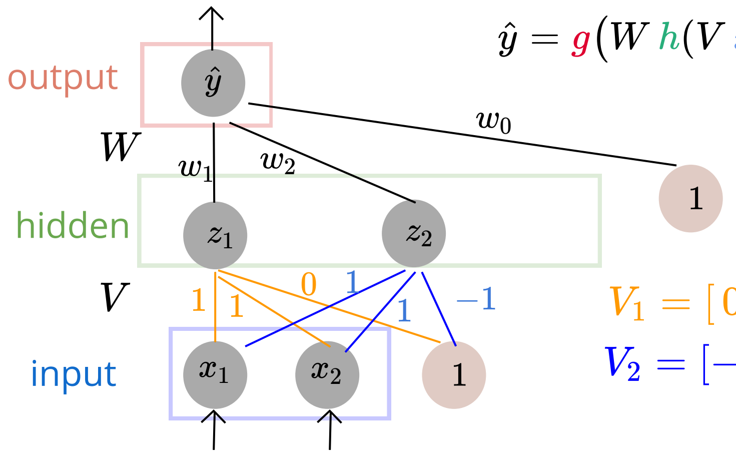
$$W = [0, 1, -2]$$

$$h(Vx) = \begin{array}{cc} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} & \begin{array}{c} \\ z_1 \\ z_2 \end{array} \end{array}$$

$$\hat{y} = g\big(W\, h(V\, x)\big)$$

$$W h(Vx) = -1$$

$$\hat{y} = g\big(W\, h(V\, x)\big) = 0$$

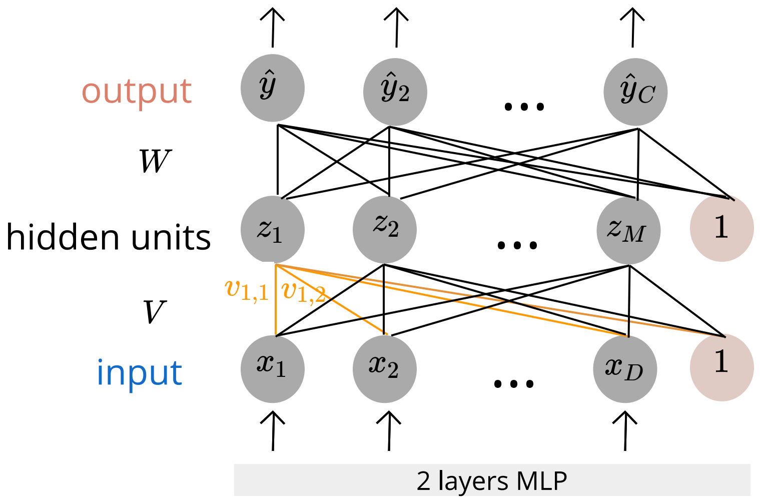


$$V_1 = [\,0\,,\,1,1]$$
$$V_2 = [-1,1,1]$$

$$V \in \mathbb{R}^{M \times \hat{D}} \qquad W \in \mathbb{R}^{C \times \hat{M}}$$

$$z_m = h\big(V_m x\big) = h\big(\textstyle\sum_d V_{m,d}\, x_d\big)$$

$$\hat{y}_k = g\big(W_k z\big) = g\big(\textstyle\sum_m W_{k,m}\, z_m\big)$$

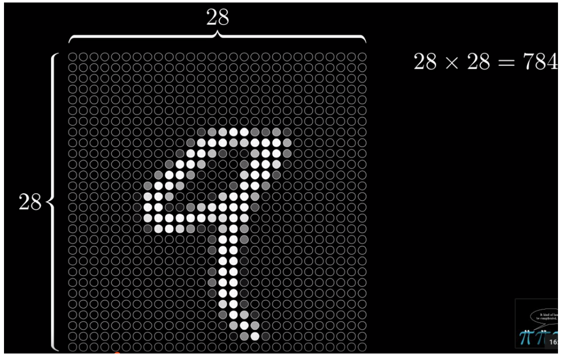


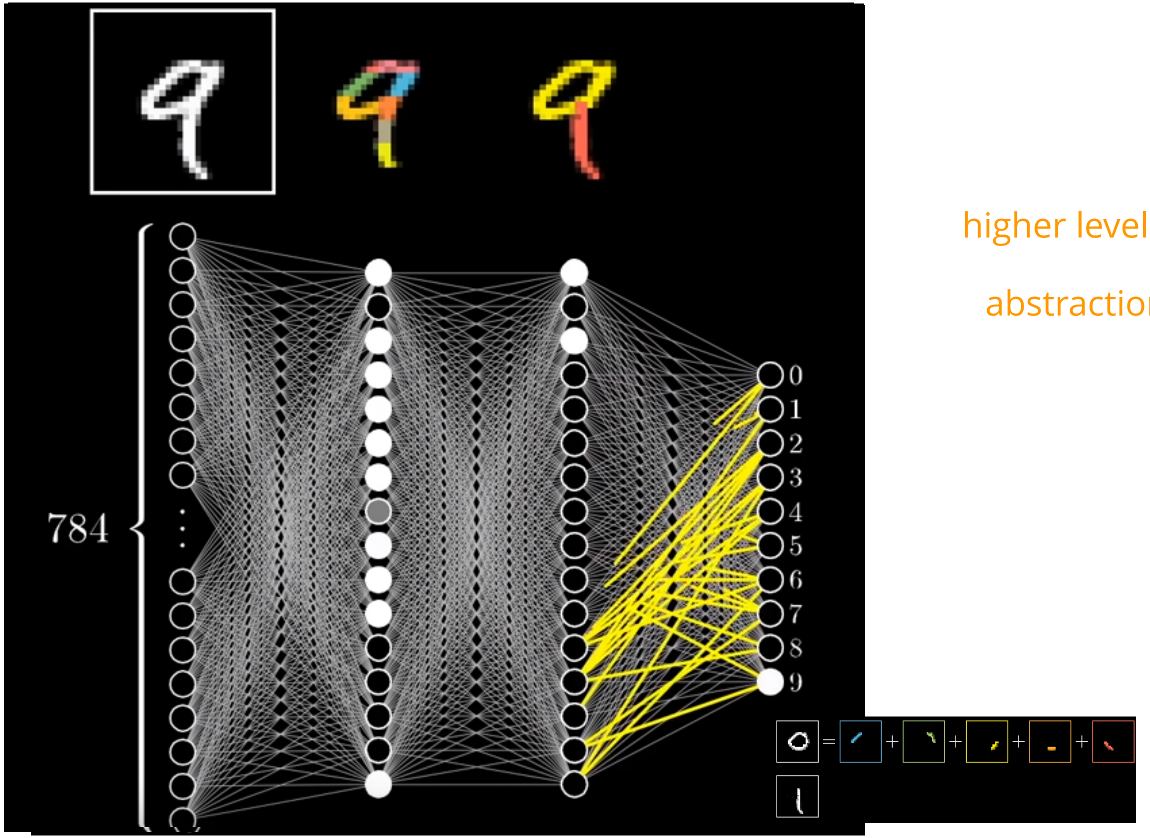
universal function approximator

model any suitably smooth function, given enough hidden units, to any desired level of accuracy

MNIST Example

classifying handwritten digits





higher level of

abstraction

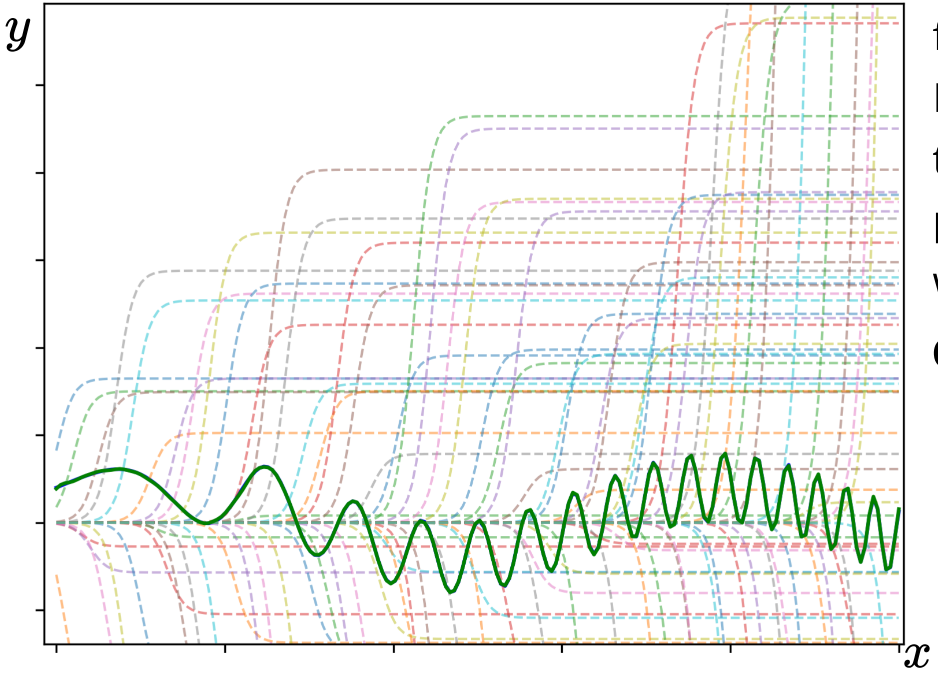
see this video for better intuition

https://www.youtube.com/watch?v=aircAruvnKk&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi&index=2&t=7s

Expressive power

universal approximation theorem

an MLP with single hidden layer can approximate any continuous function with arbitrary accuracy



for 1D input we can see this even with **fixed bases**

M = 100 in this example

the fit is good (hard to see the blue line)

however # bases (M) should grow exponentially with D (**curse of dimensionality**)

Caveats of the universality

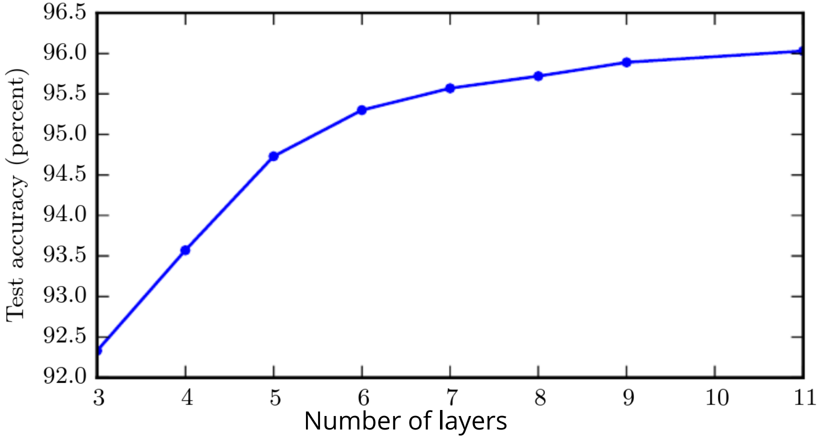
- we may need a very wide network (large M)
- this is only about training error, we care about test error

Depth vs Width

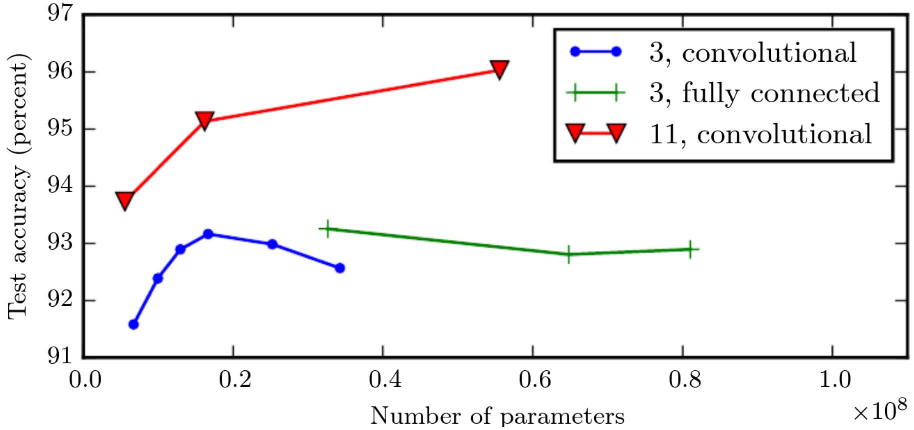
Deep networks (with ReLU activation) of bounded width are also shown to be universal

- empirically, increasing the depth is often more effective than increasing the width (#parameters per layer)
- compositional functional form through depth is a useful inductive bias

increasing depth in image recognition



increasing the width (# parameters)



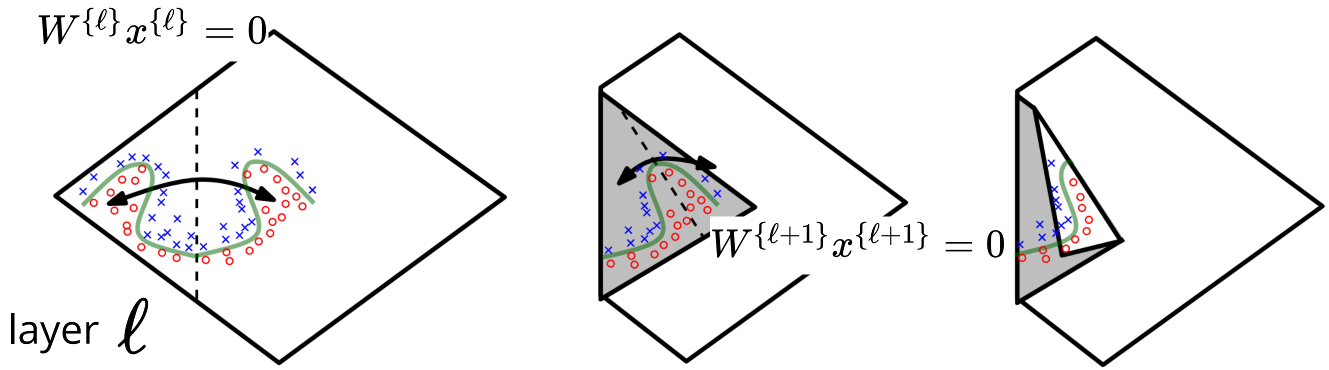
optional

Depth vs Width

Deep networks (with ReLU activation) of bounded width are also shown to be universal

number of regions (in which the network is linear) grows exponentially with depth

simplified demonstration $h(W^{\{\ell\}}x) = |W^{\{\ell\}}x|$



Regularization strategies

universality of neural networks also means they can overfit

strategies for variance reduction:

- L1 and L2 regularization *(weight decay)*
- data augmentation
- noise robustness
- early stopping
- dropout
- bagging
- sparse representations *(e.g., L1 penalty on hidden unit activations)*
- semi-supervised and multi-task learning
- adversarial training
- parameter-tying

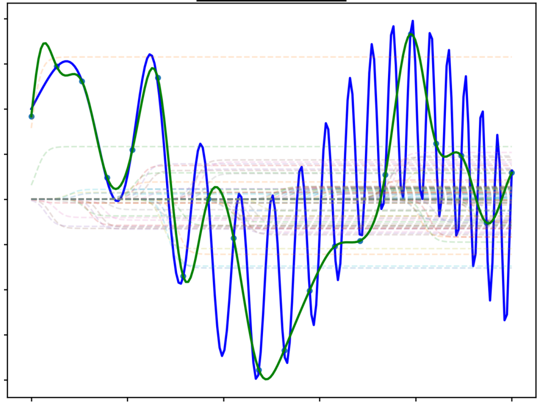
Regularization using **Data augmentation**

a larger dataset results in a better generalization

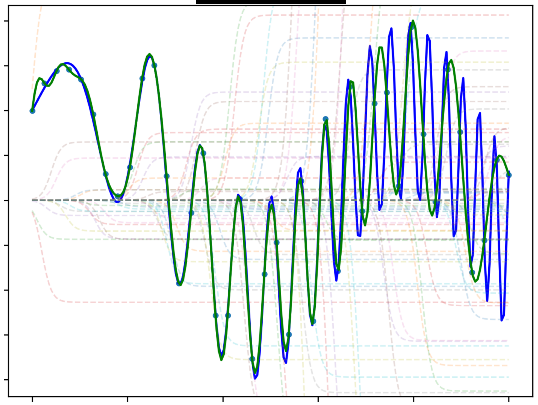
example: in all 3 examples below training error is close to zero

however, a larger training dataset leads to better generalization

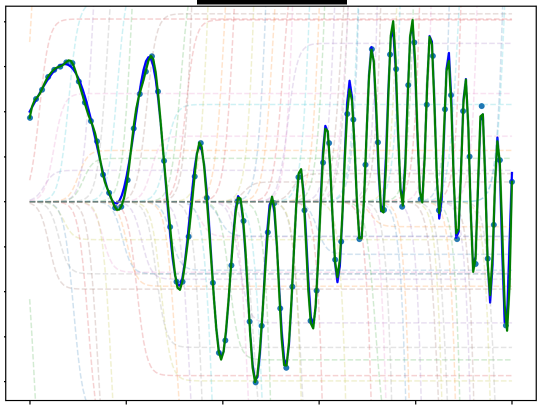
$N = 20$



$N = 40$



$N = 80$



Regularization using **Data augmentation**

a larger dataset results in a better generalization

image: https://github.com/aleju/imgaug/blob/master/README.md

idea

increase the size of dataset by adding reasonable transformations $\tau(x)$
that change the label in predictable ways; e.g., $f(\tau(x)) = f(x)$

special approaches to data-augmentation

- adding noise to the input
- adding noise to hidden units
  - noise in higher level of abstraction
- learn a **generative model** $\hat{p}(x, y)$ of the data
  - use $x^{(n')}, y^{(n')} \sim \hat{p}$ for training

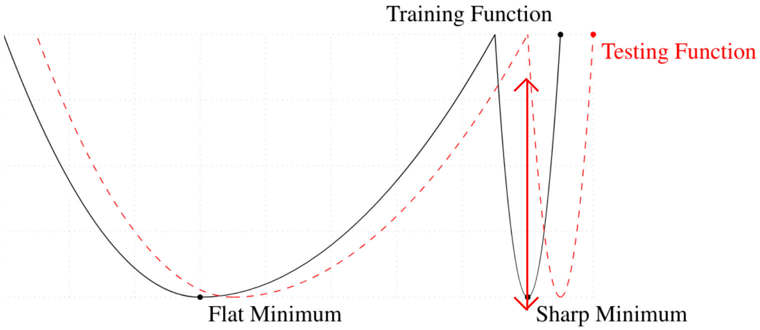
sometimes we can achieve the same goal by designing the models that are **invariant** to a given set of transformations

Regularization using **Noise** robustness

1. input (data augmentation)

2. hidden units (*e.g.,* in dropout as we see soon)

3. weights the cost is not sensitive to small changes in the weight (**flat minima**)



flat minima generalize better

good performance of SGD using small minibatch is attributed to converging to flat minima which generalizes better (train loss closer to test loss)

in this case, SGD regularizes the model due to **gradient noise**

https://arxiv.org/pdf/1609.04836.pdf

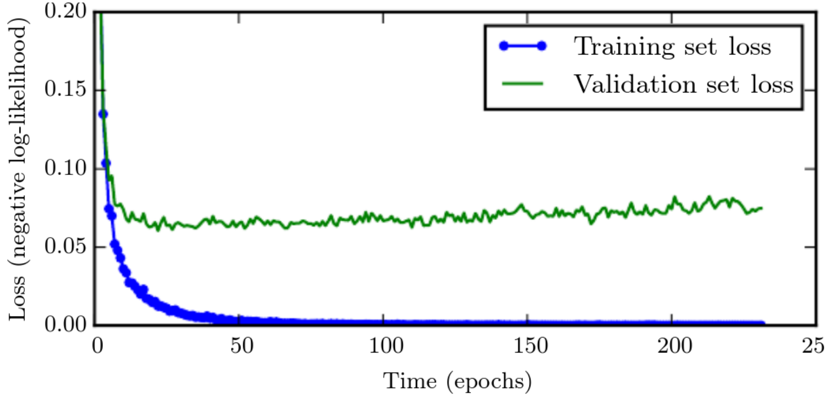
4. output (avoid overfitting, specially to wrong labels)

a heuristic is to replace hard labels with "soft-labels" label smoothing

e.g., $[0, 0, 1, 0] \rightarrow [\frac{\epsilon}{3}, \frac{\epsilon}{3}, 1-\epsilon, \frac{\epsilon}{3}]$

image credit: Keshkar et al'17

Regularization using **Early stopping**



the **test loss**-vs-**time step** is "often" U-shaped

use validation for early stopping

also saves computation!

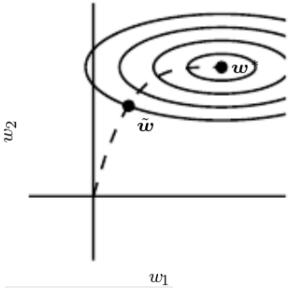
early stopping bounds the region of the parameter-space that is reachable in T time-steps

assuming

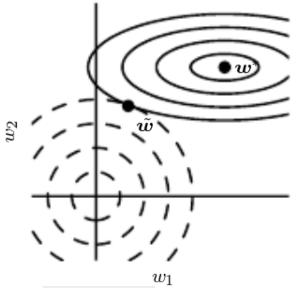
- *bounded gradient*
- *starting with a small w*

it has an effect similar to L2 regularization

we get the regularization path (various λ)



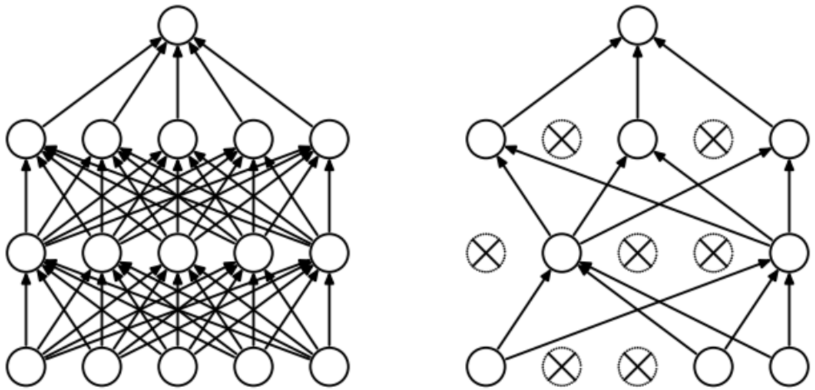
early stopping



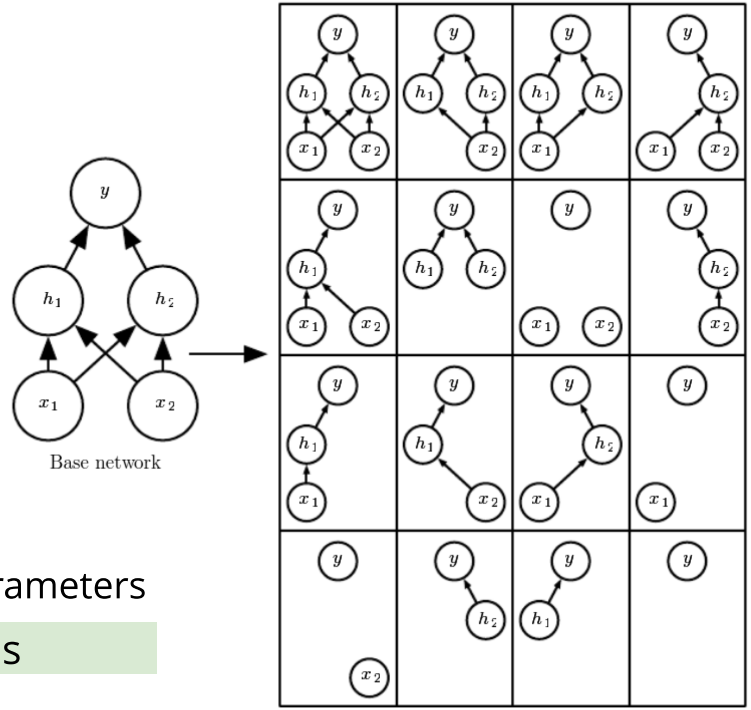
L2

Regularization using **Dropout**

randomly remove a subset of units during training



(a) Standard Neural Net (b) After applying dropout.



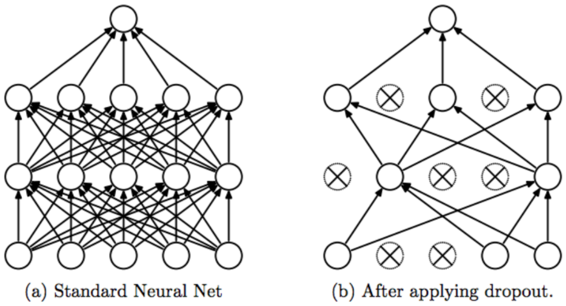
Base network

Ensemble of subnetworks

can be viewed as exponentially many subnetworks that share parameters

is one of the most effective regularization schemes for MLPs

Regularization using Dropout



during training

for each instance (n):

randomly dropout each unit with probability p (e.g., p=.5)

only the remaining subnetwork participates in training

at test time

ideally we want to average over the prediction of all possible sub-networks
this is computationally infeasible, instead:

1) Monte Carlo dropout: average the prediction of several feed-forward passes using dropout

2) weight scaling: scale the weights by **p** to compensate for dropout

e.g., for 50% dropout, scale by a factor of 2

either multiply by 2 in training or divide by 2 at the end of training

Summary

Deep feed-forward networks learn **adaptive bases**

more complex bases at higher layers

increasing **depth** is often preferable to width

various choices of **activation function** and **architecture**

universal approximation power

their expressive power often necessitates using **regularization** schemes