

# Applied Machine Learning

Multilayer Perceptron

**Reihaneh Rabbany**



**COMP 551 (winter 2020)**

# Learning objectives

multilayer perceptron:

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

# Adaptive bases

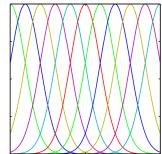
several methods can be classified as *learning these bases adaptively*

$$f(x) = \sum_d w_d \phi_d(x; v_d)$$

- decision trees
- generalized additive models
- boosting
- neural networks



- consider the adaptive bases in a general form (contrast to decision trees)
- use gradient descent to find good parameters (contrast to boosting)
- create more complex adaptive bases by combining simpler bases
  - leads to **deep neural networks**



$$\phi_d(x) = e^{-\frac{(x-\mu_d)^2}{s^2}}$$

Gaussian bases, or radial bases

# Adaptive Radial Bases

## non-adaptive case

model:  $f(x; w) = \sum_d w_d \phi_d(x)$

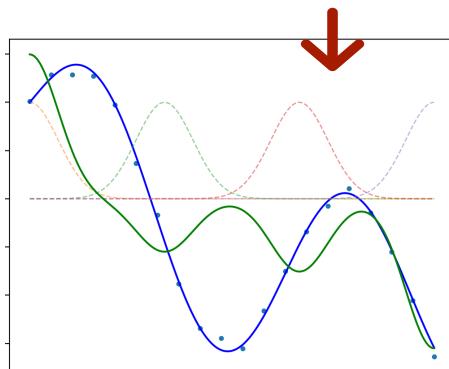
cost:  $J(w) = \frac{1}{2} \sum_n (f(x^{(n)}; w) - y^{(n)})^2$

the center are fixed ←

the model is linear in its parameters

the cost is convex in w (unique minimum)

even has a closed form solution



## adaptive case

we can make the bases adaptive by learning these centers

model:  $f(x; w, \mu) = \sum_d w_d \phi_d(x; \mu_d)$

how to minimize the cost?

not convex in all model parameters

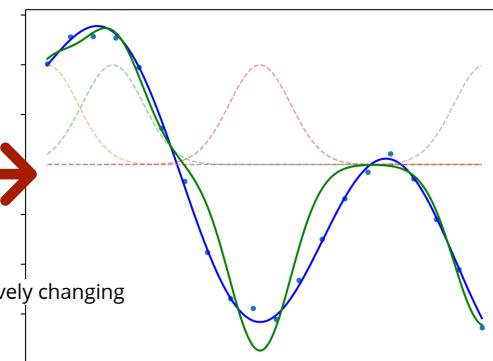
use gradient descent to find a local minimum →

note that the basis centers are adaptively changing

```

1 #x: N
2 #y: N
3 plt.plot(x, y, 'b.')
4 phi = lambda x,mu: np.exp(-(x-mu)**2)
5 mu = np.linspace(0,4,10) #4 Gaussians bases
6 Phi = phi(x[:,None], mu[None,:]) #N x 10
7 w = np.linalg.lstsq(Phi, y)[0]
8 yh = np.dot(Phi,w)
9 plt.plot(x, yh, 'g-')

```





# Sigmoid Bases

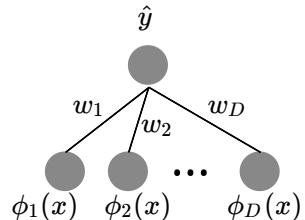
$$\phi_d(x) = \frac{1}{1+e^{-(\frac{x-\mu_d}{s_d})}}$$

using adaptive sigmoid bases gives us a neural network

## non-adaptive case

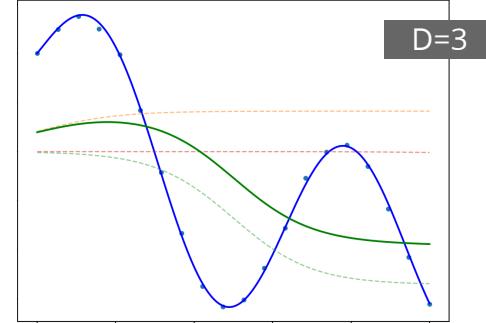
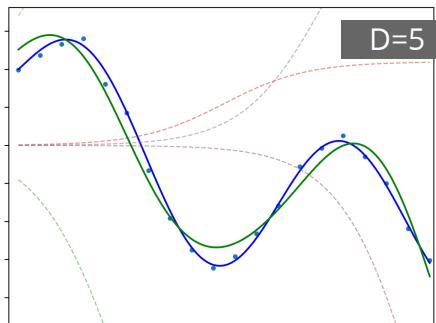
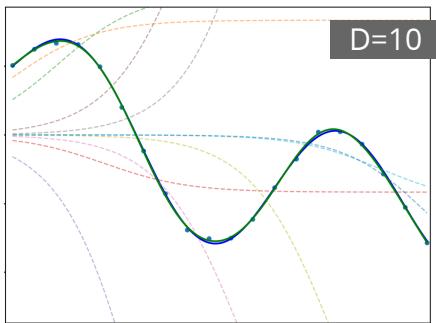
- $\mu_d$  is fixed to D locations
- $s_d = 1$

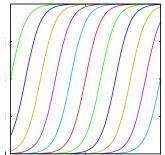
model:  $f(x; w) = \sum_d w_d \phi_d(x)$



```

1 #x: N
2 #y: N
3 plt.plot(x, y, 'b.')
4 phi = lambda x,mu,sigma: 1/(1 + np.exp(-(x - mu)))
5 mu = np.linspace(0,3,10)
6 Phi = phi(x[:,None], mu[None,:]) #N x 10
7 w = np.linalg.lstsq(Phi, y)[0]
8 yh = np.dot(Phi,w)
9 plt.plot(x, yh, 'g-')
  
```





# Adaptive Sigmoid Bases

$$\phi_d(x) = \frac{1}{1+e^{-(\frac{x-\mu_d}{s_d})}}$$

rewrite the sigmoid basis

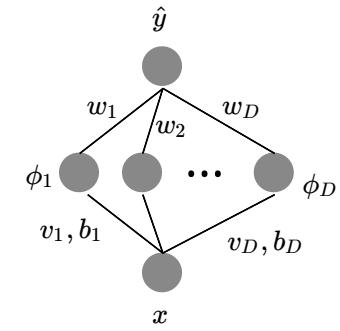
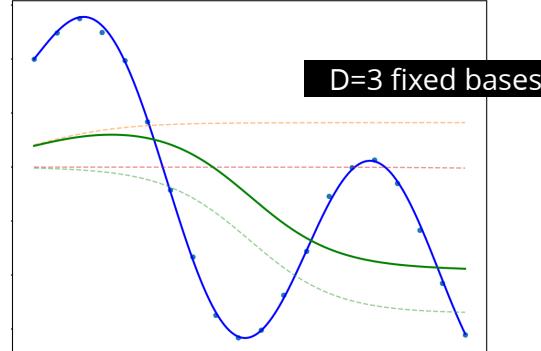
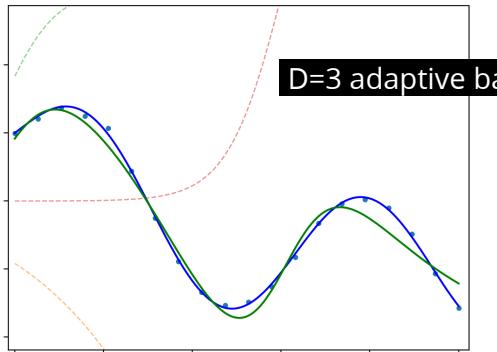
$$\phi_d(x) = \sigma\left(\frac{x-\mu_d}{s_d}\right) = \sigma(v_d x + b_d)$$

each basis is the logistic regression model  $\phi_d(x) = \sigma(v_d^\top x + b_d)$

assuming input is higher than one dimension

model:  $f(x; w, v, b) = \sum_d w_d \sigma(v_d^\top x + b_d)$  this is a **neural network** with two layers →

optimize using gradient descent (find a local optima)



# Multilayer Perceptron (MLP)

suppose we have

- D inputs  $x_1, \dots, x_D$
- K outputs  $\hat{y}_1, \dots, \hat{y}_K$
- M hidden units  $z_1, \dots, z_M$

model

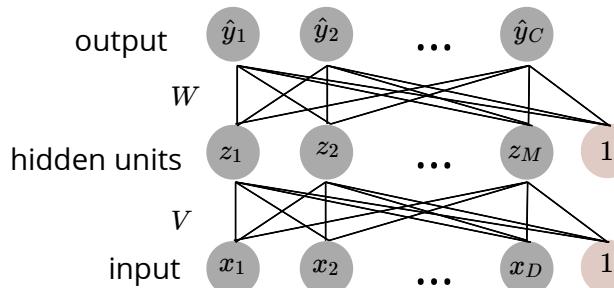
$$\hat{y}_k = g \left( \sum_m W_{k,m} h \left( \sum_d V_{m,d} x_d \right) \right)$$

nonlinearity, activation function: we have different choices

more compressed form

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

non-linearities are applied elementwise



for simplicity we may drop bias terms

# Regression using Neural Networks

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

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

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

we may have one or more output variables

identity function + L2 loss : Gaussian likelihood

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

more generally

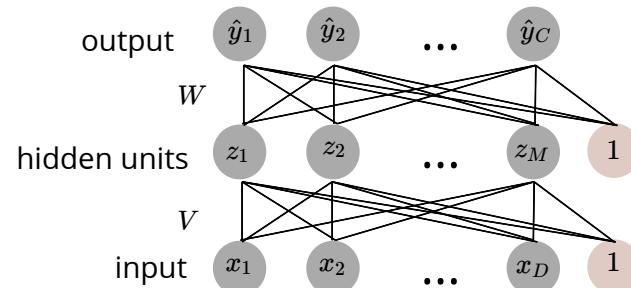
we may explicitly produce a distribution at output - e.g.,

- mean and variance of a Gaussian
- mixture of Gaussians

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*



Recall: Maximum likelihood & linear regression

$$p(y; \hat{y}) = \mathcal{N}(y; \hat{y}, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y-\hat{y})^2}{2\sigma^2}}$$

$$\log(p(y; \hat{y})) = -\frac{1}{2\sigma^2}(y - \hat{y})^2 + \text{constants}$$

# Classification using neural networks

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

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

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

scalar output C=1

logistic sigmoid + CE loss: Bernoulli likelihood

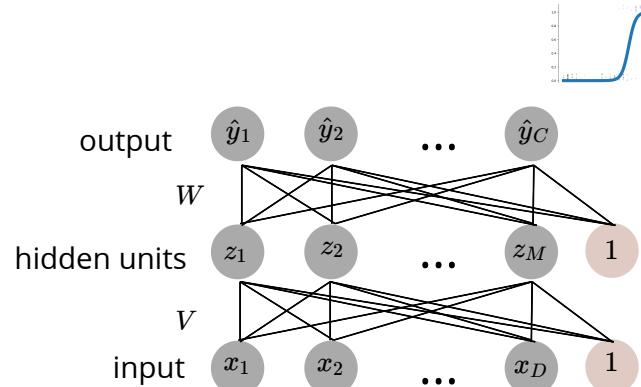
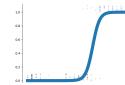
$$L(y, \hat{y}) = y \log \hat{y} + (1 - y) \log(1 - \hat{y}) = \log \text{Bernoulli}(y; \hat{y})$$

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

C is the number of classes

softmax + multi-class CE loss: categorical likelihood

$$L(y, \hat{y}) = \sum_k y_k \log \hat{y}_k = \log \text{Categorical}(y; \hat{y})$$



Recall: Maximum likelihood & logistic regression

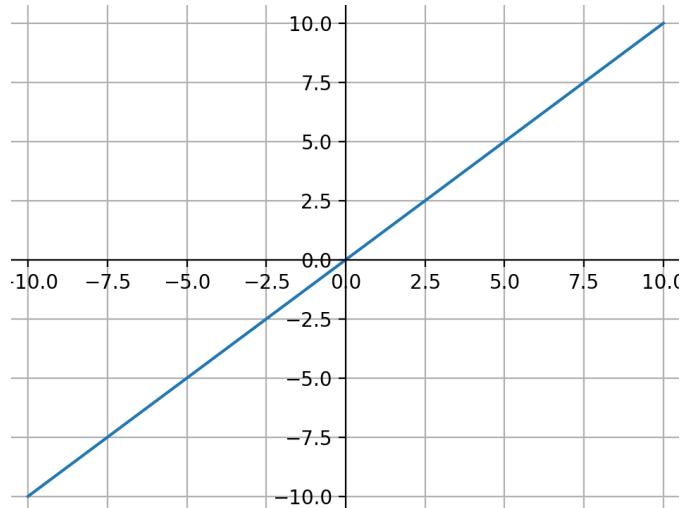
$$p(y; \hat{y}) = \text{Bernoulli}(y; \hat{y}) = \hat{y}^y (1 - \hat{y})^{1-y}$$

$$\log(p(y; \hat{y})) = y \log \hat{y} + (1 - y) \log(1 - \hat{y})$$

$$\text{Categorical}(y; \hat{y}) = \prod_{c=1}^C \hat{y}_c^{\mathbb{I}(y=c)}$$

# Activation function

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



$$h(x) = x \text{ identity (no activation function)}$$

composition of two linear functions is linear

$$\underbrace{WVx}_{W'} = W'x$$

$K \times M \quad M \times D \quad K \times D$

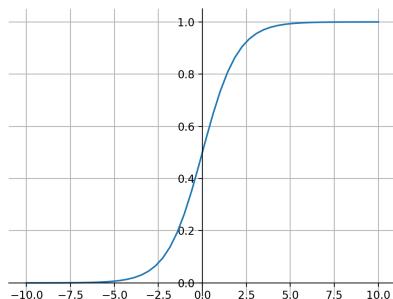
so nothing is gained (in representation power) by stacking linear layers

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

this idea is used in dimensionality reduction (later!)

# Activation function

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



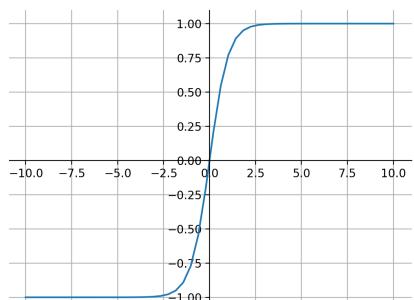
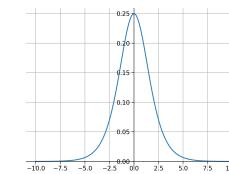
$$h(x) = \sigma(x) = \frac{1}{1+e^{-x}} \quad \text{logistic function}$$

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



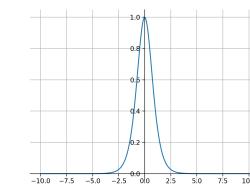
$$h(x) = 2\sigma(x) - 1 = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad \text{hyperbolic tangent}$$

similar to sigmoid, but symmetric

often better for optimization because close to zero it is similar to a linear function

(rather than an affine function when using logistic)

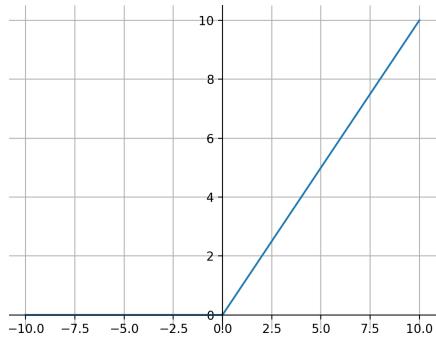
similar problem with vanishing gradient



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

# Activation function

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



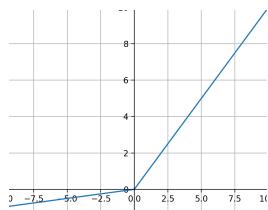
$$h(x) = \max(0, x)$$
 Rectified Linear Unit (**ReLU**)

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  $\gamma$  a learnable parameter

**Softplus** (differentiable everywhere)

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



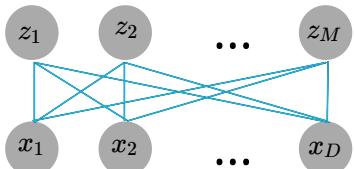
it doesn't perform as well  
in practice

# Network architecture

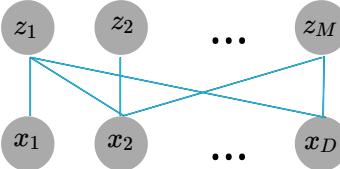
architecture is the overall structure of the network

**feedforward network** (aka multilayer perceptron)

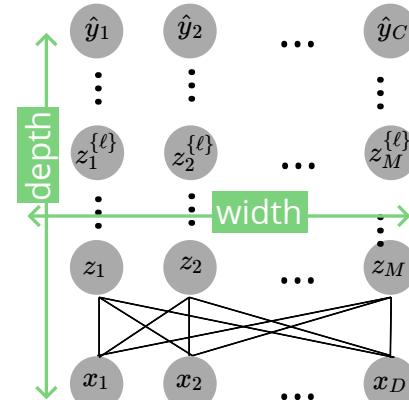
- can have many layers
- # layers is called the **depth** of the network
- each layer can be **fully connected** (dense) or sparse



fully connected



sparsely connected

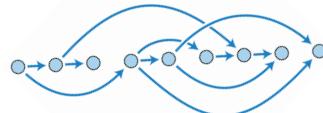


# Network architecture

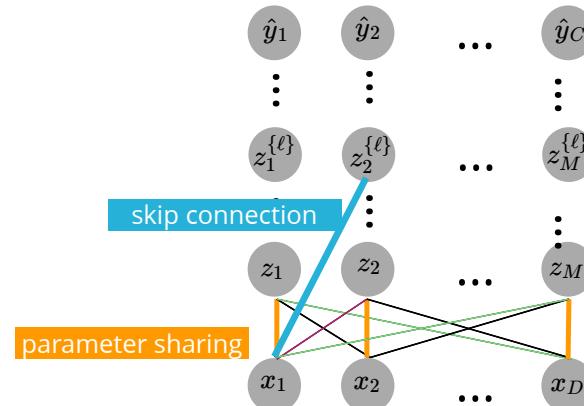
architecture is the overall structure of the network

**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**
  - helps with gradient flow
- units may have different **activations**
- parameters may be shared across units (e.g., in conv-nets)



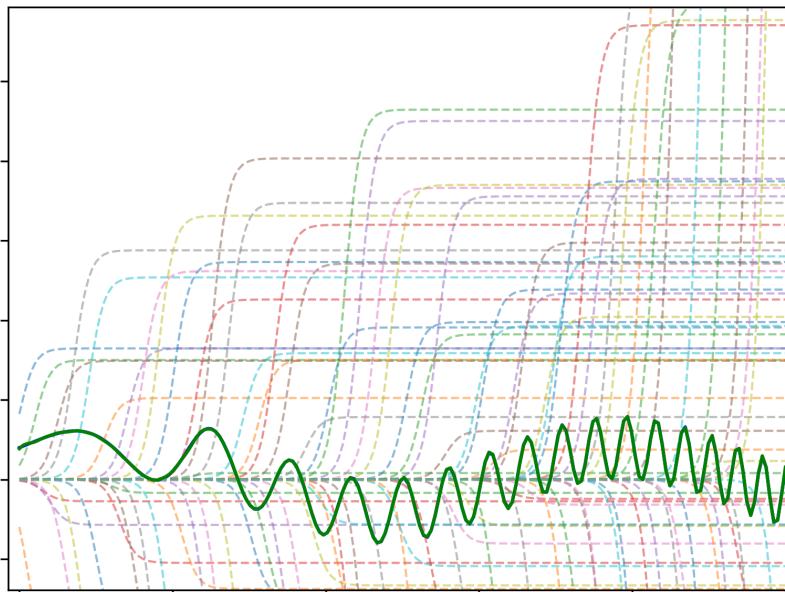
more generally a directed acyclic graph (DAG) expresses the feed-forward architecture



# 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**)

# Depth vs Width

universal approximation theorem

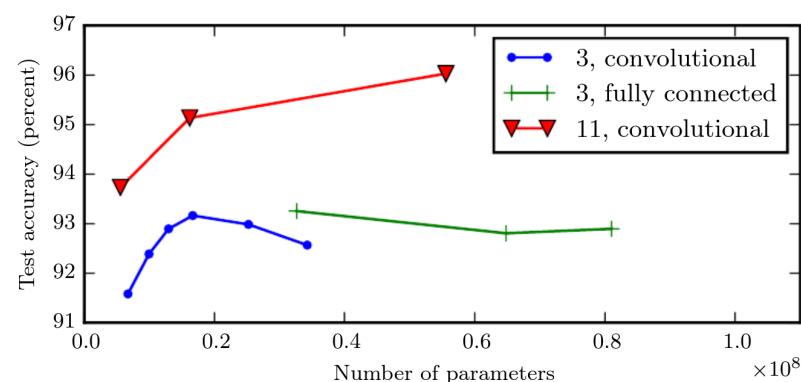
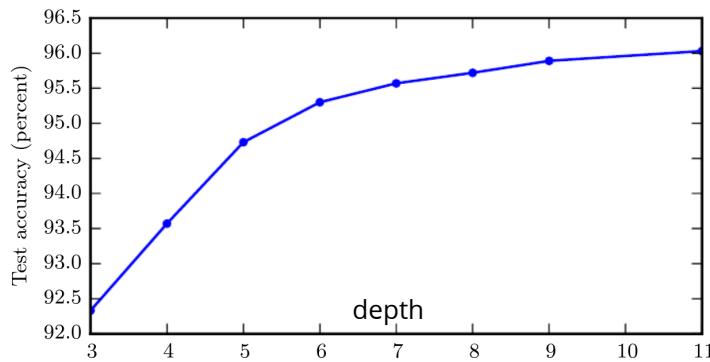
an MLP with single hidden layer can approximate any continuous function on with arbitrary accuracy

## Caveats

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

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

empirically it is observed that increasing depth is often more effective than increasing width (#parameters per layer)  
assuming a compositional functional form (through depth) is a useful inductive bias



# Depth vs Width

universal approximation theorem

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

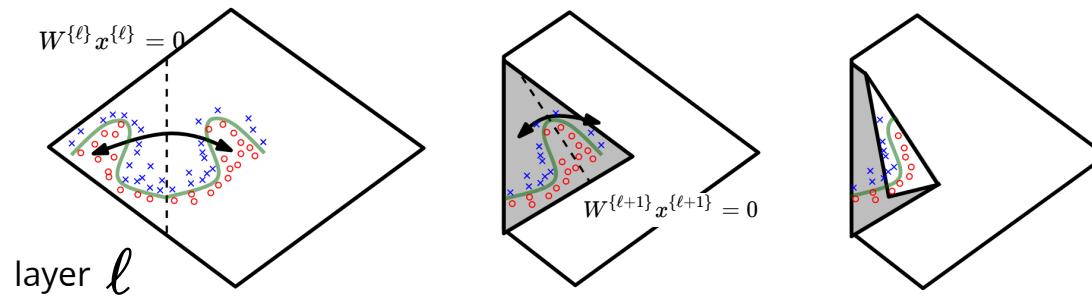
## Caveats

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

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

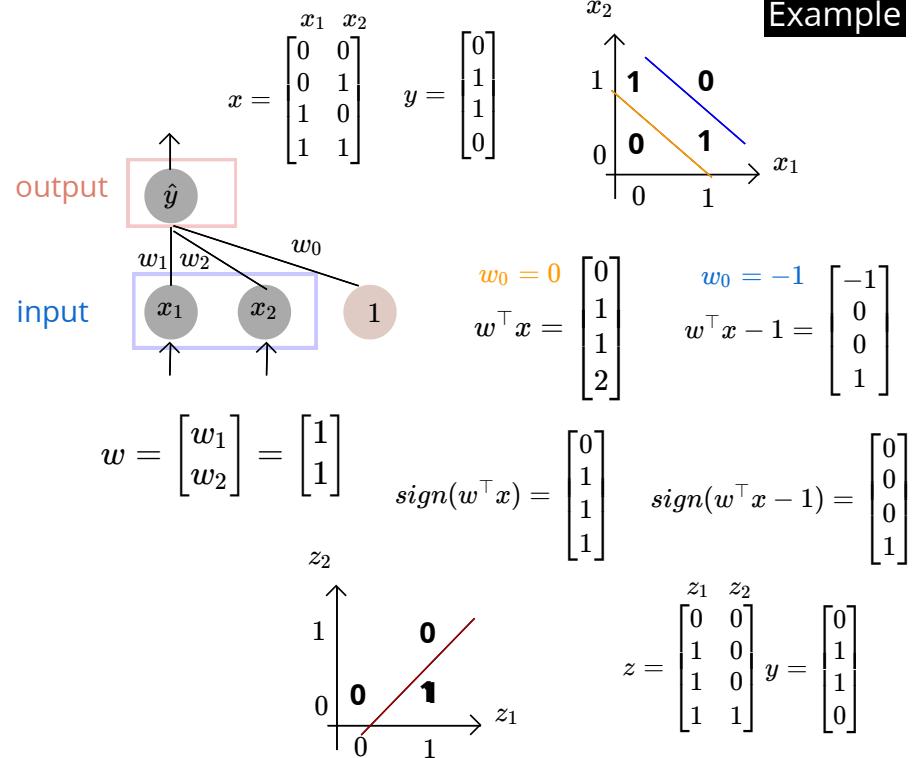
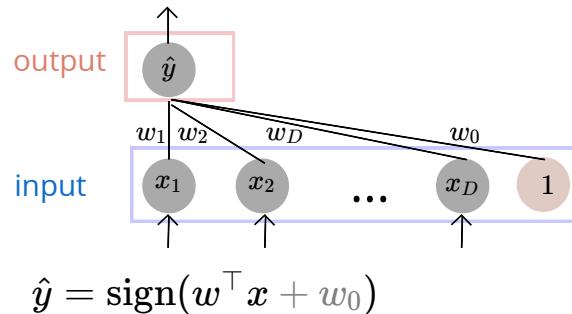
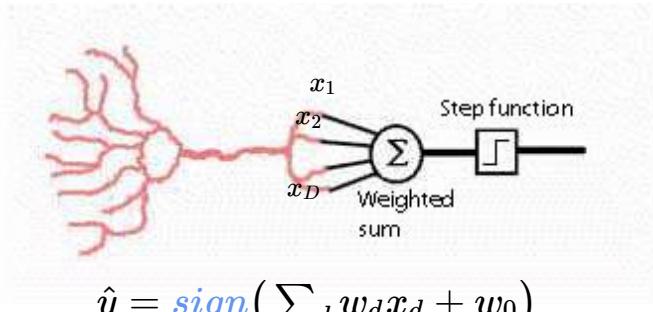


# Multilayer Perceptron

a.k.a. Feed-forward network

recap

## Recall Perceptron

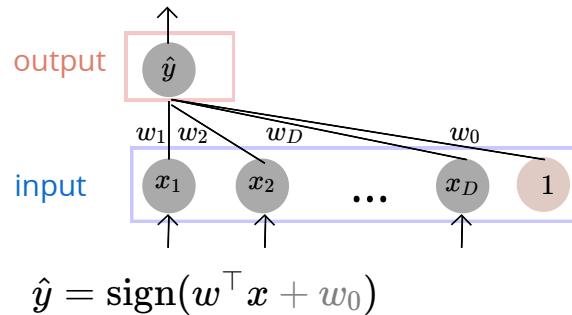
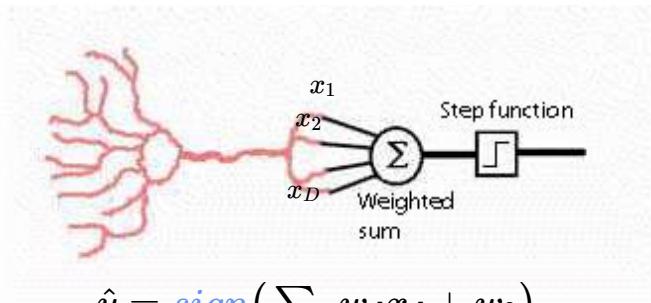


# Multilayer Perceptron

a.k.a. Feed-forward network

recap

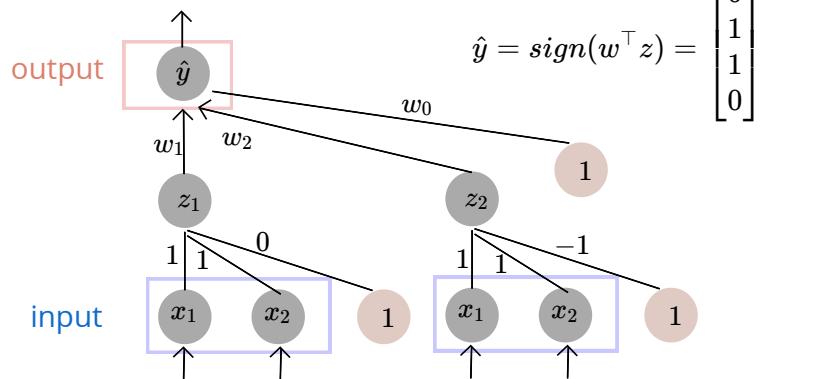
## Recall Perceptron



$$x = \begin{bmatrix} x_1 & x_2 \\ 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \quad y = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Example

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



# Multilayer Perceptron

a.k.a. Feed-forward network

recap

$$V = \begin{bmatrix} 0 & 1 & 1 \\ -1 & 1 & 1 \end{bmatrix}$$

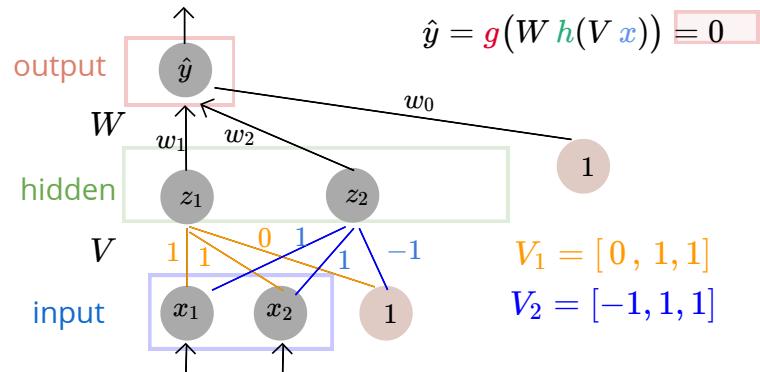
$$\underline{x} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

$$Vx = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

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

$$\begin{aligned} h(Vx) &= \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} z_1 \\ W h(Vx) &= -1 \end{aligned}$$

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



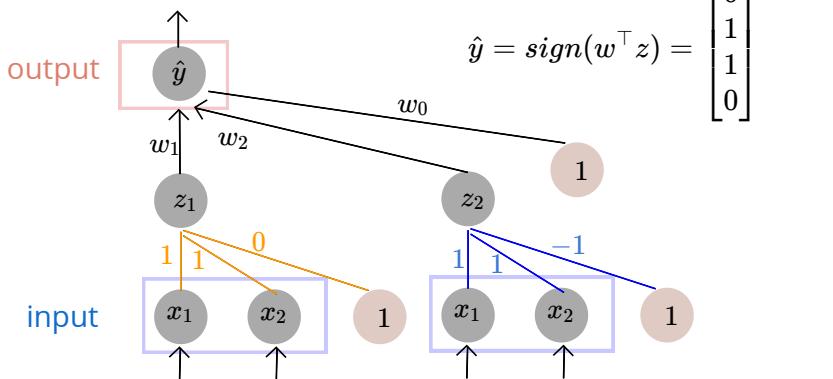
$$x = \begin{bmatrix} x_1 & x_2 \\ 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \quad y = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Example

$$z = \begin{bmatrix} z_1 & z_2 \\ 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$$

$$w = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}$$

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



# Multilayer Perceptron

a.k.a. Feed-forward network

recap

$$V = \begin{bmatrix} 0 & 1 & 1 \\ -1 & 1 & 1 \end{bmatrix}$$

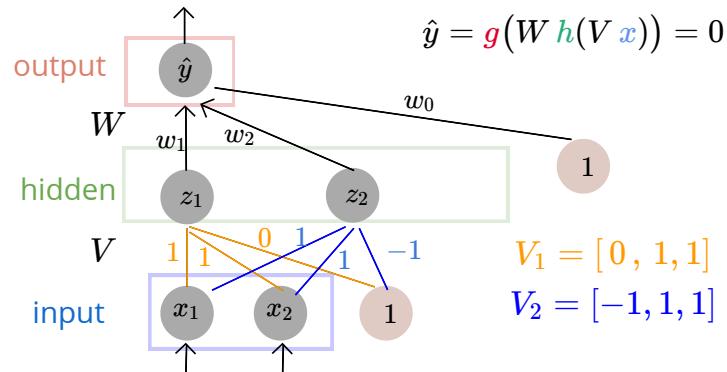
$$\underline{x} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

$$Vx = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

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

$$\hat{y} = \textcolor{red}{g}(W \textcolor{teal}{h}(V \underline{x}))$$

$$\begin{aligned} \textcolor{teal}{h}(Vx) &= \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} z_1 \\ W\textcolor{teal}{h}(Vx) &= -1 \end{aligned}$$

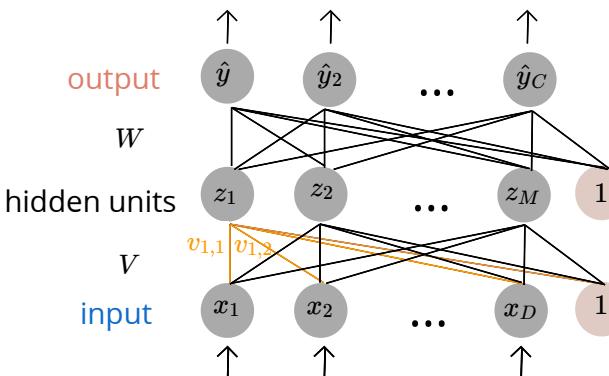


$$\hat{y} = \textcolor{red}{g}(W \textcolor{teal}{h}(V \underline{x})) = 0$$

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

$$z_m = \textcolor{teal}{h}(V_m x) = \textcolor{teal}{h}\left(\sum_d V_{m,d} x_d\right)$$

$$y_k = \textcolor{red}{g}(W_k z) = \textcolor{red}{g}\left(\sum_m W_{k,m} z_m\right)$$

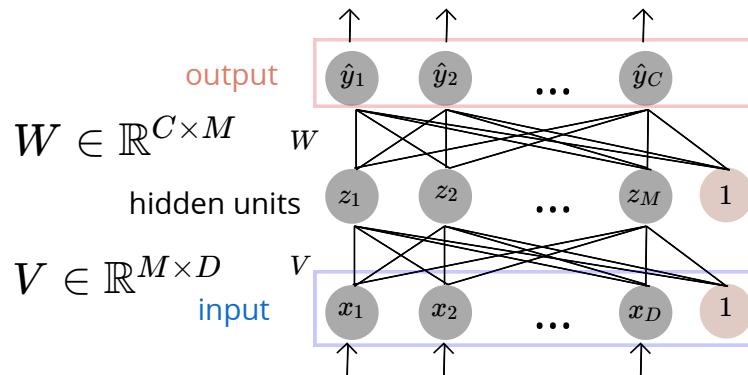


# Multilayer Perceptron

a.k.a. Feed-forward network

recap

2 layers

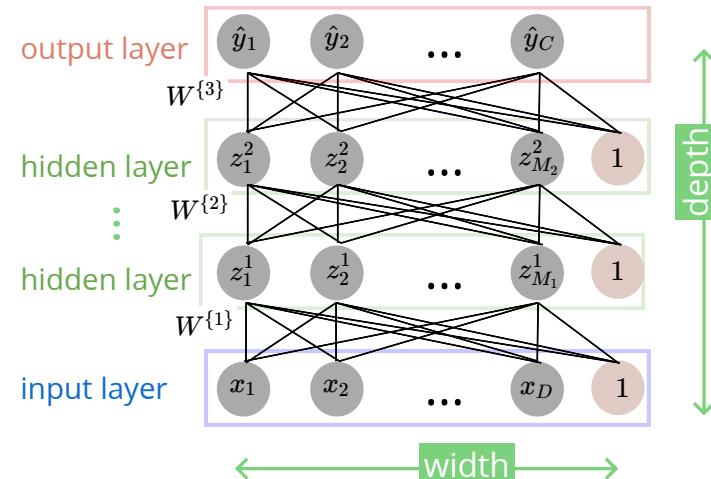


$$\hat{y} = g(W h(V x))$$

universal function approximator

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

Multiple layers

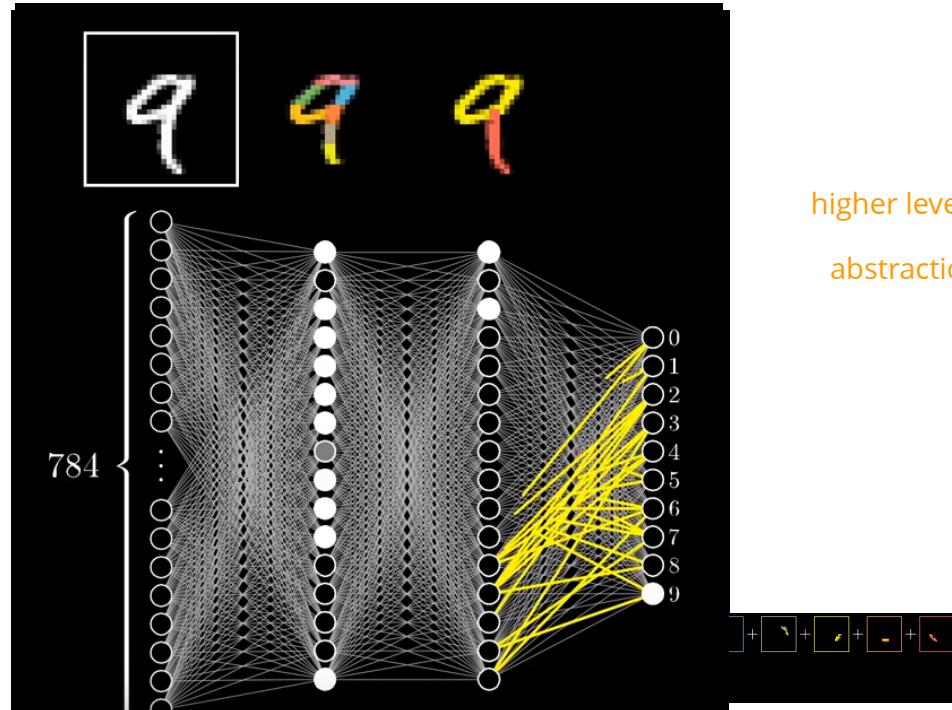
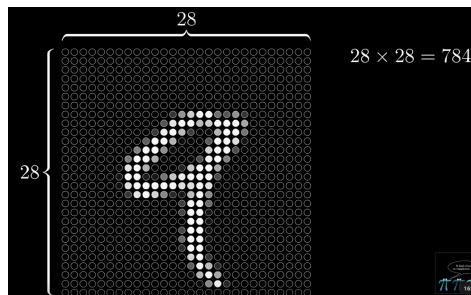


$$z^{(l)} = h(W^{(l)} z^{(l-1)})$$

activation function

# MNIST Example

classifying handwritten digits



see this video for better intuition

[https://www.youtube.com/watch?v=aircAruvnKk&list=PLZHQBObOWTQDNU6R1\\_67000Dx\\_ZCJB-3pi&index=2&t=7s](https://www.youtube.com/watch?v=aircAruvnKk&list=PLZHQBObOWTQDNU6R1_67000Dx_ZCJB-3pi&index=2&t=7s)

# Multilayer Perceptron

a.k.a. Feed-forward network

recap

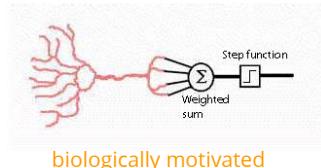
activation function for hidden units

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

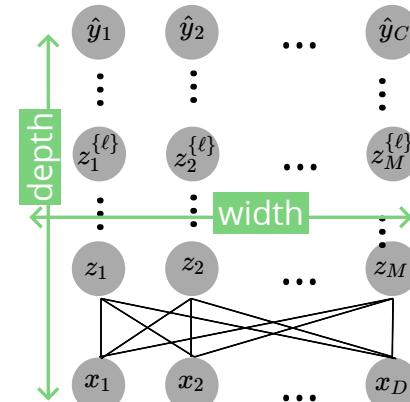
logistic function

$$h(x) = \max(0, x)$$

Rectified Linear Unit (ReLU)



output activation function  
depends on the task



How to train this?

$$W^* = \min_W \sum_n L(y^{(n)}, f(x^{(n)}; W))$$

loss function depends on the task

How to avoid overfitting?

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

output of one layer is input to the next

fully-connected

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

# Regularization strategies

universality of neural networks also means they can overfit  
strategies for variance reduction:

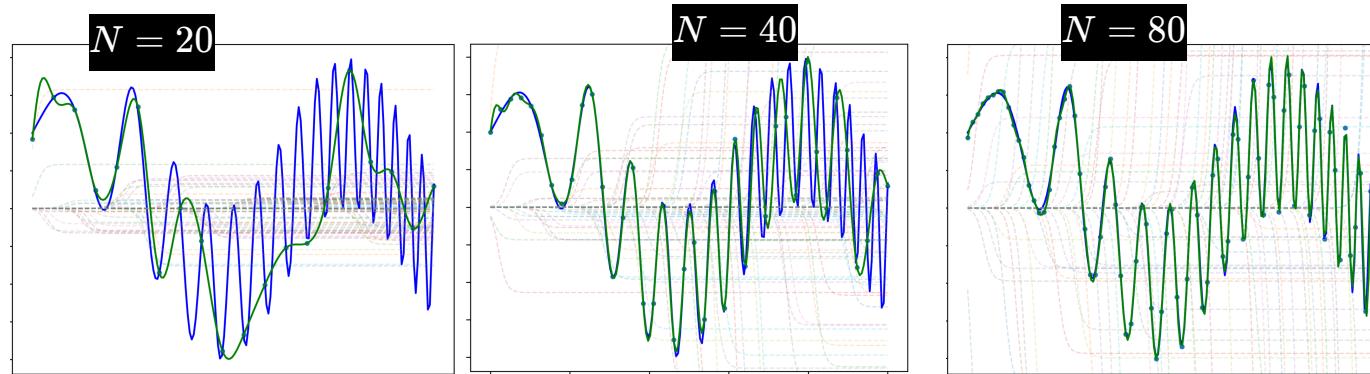
- L1 and L2 regularization (*weight decay*)
- data augmentation
- noise robustness
- early stopping
- bagging and dropout
- sparse representations (e.g., *L1 penalty on hidden unit activations*)
- semi-supervised and multi-task learning
- adversarial training
- parameter-tying

# Data augmentation

a larger dataset results in a better generalization

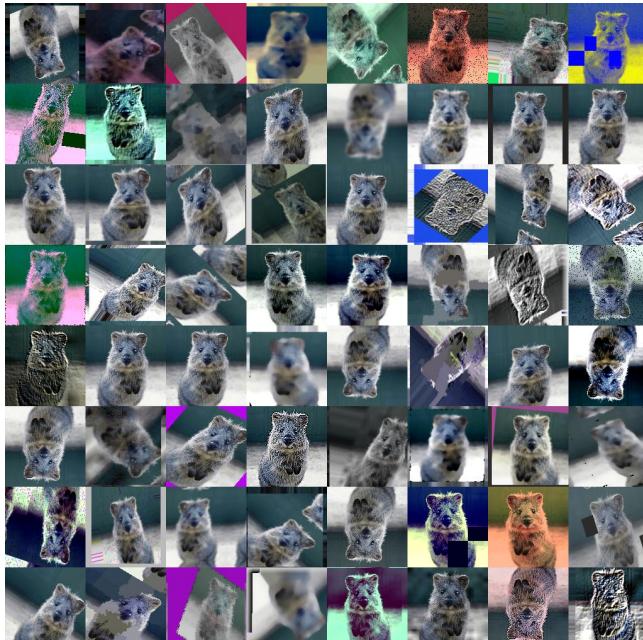
**example:** in all 3 examples below training error is close to zero

however, a larger training dataset leads to better generalization



# Data augmentation

a larger dataset results in a better generalization



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

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

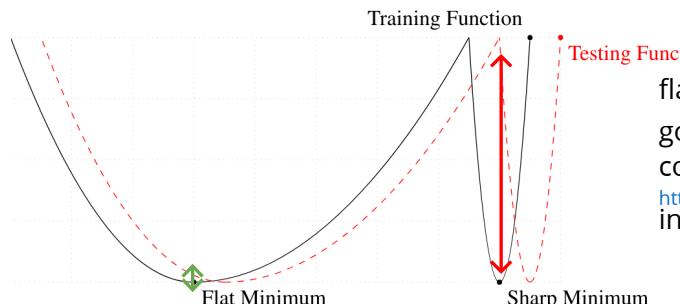
# Noise robustness

make the model robust to noise in

**input** (data augmentation)

**hidden units** (e.g., in dropout)

**weights** the loss 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 into flat minima which generalizes better, train loss closer to test loss

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

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

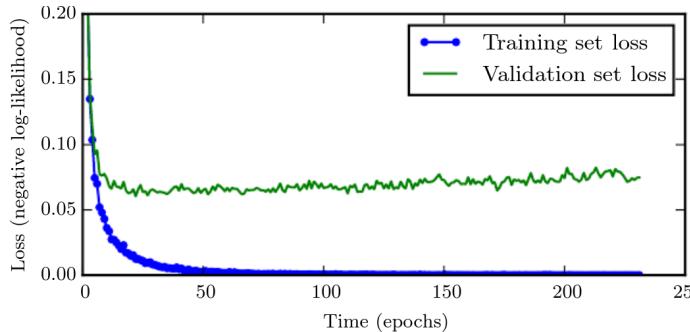
**output** (avoid overfitting, specially to wrong labels)

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

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

image credit: Keshkar et al'17

# Early stopping

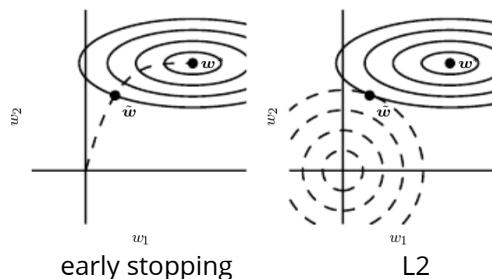


the **test loss-vs-time step** is "often" U-shaped  
use validation for early stopping  
also saves computation!

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  $\lambda$ )  
*we saw a similar phenomena in boosting*



# Bagging

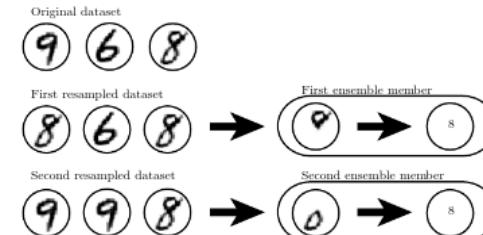
several sources of variance in neural networks, such as

- optimization
  - initialization
  - randomness of SGD
  - learning rate and other hyper-parameters
- choice of architecture
  - number of layers, hidden units, etc.

use bagging or even averaging without bootstrap to reduce variance

**issue:** computationally expensive

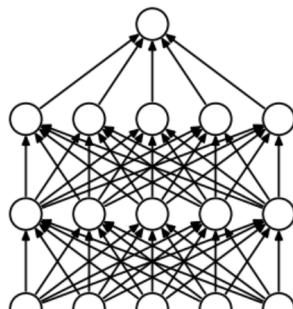
How to make it practical?



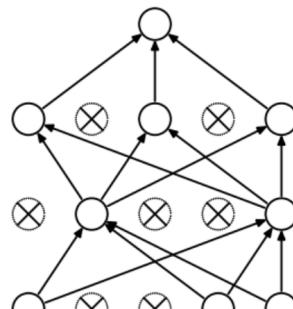
# Dropout

idea

randomly remove a subset of units during training  
as opposed to bagging a single model is trained



(a) Standard Neural Net

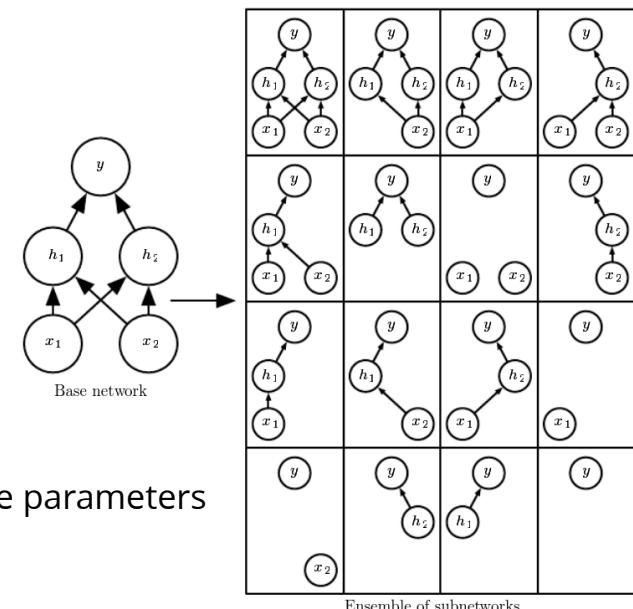


(b) After applying dropout.

can be viewed as exponentially many subnetworks that share parameters  
is one of the most effective regularization schemes for MLPs

How is it different than bagging?

Models are not independent and share parameters

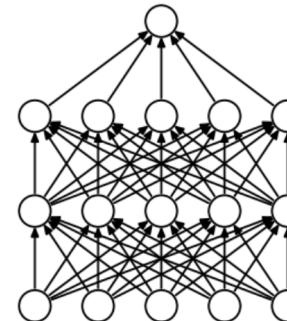


# Dropout

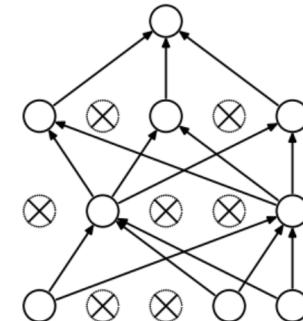
during training

for each instance ( $n$ ):

randomly dropout each unit with probability  $p$  (e.g.,  $p=.5$ )  
only the remaining subnetwork participates in training



(a) Standard Neural Net



(b) After applying dropout.

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

in general this is **not** equivalent to the average prediction of the ensemble

*"the expected total input to a unit at test time is roughly the same as the expected total input to that unit at train time, even though half the units at train time are missing on average"*

# 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