Demo of Bias-Variance Tradeoff

Introduction to Data Mining

CS 145

Lecture 3:

Logistic Regression, MLP

& Backpropagation

Recap: Basic Supervised Learning

- Training Data: $S = \{(x_i, y_i)\}_{i=1}^{N}$ $x \in R^D$ $y \in \{-1, +1\}$

  Sometimes we need a pre-defined **feature engineer** to get proper x (e.g. bag-of-word) from raw data

- Model Class: $f(x \mid w, b) = w^T x - b$ **Linear Models**

- Loss Function: $L\big(y_i, f(x_i \mid w, b)\big)$ **Squared Loss**

- Learning Objective: $\underset{w,b}{\operatorname{argmin}} \sum_{i=1}^{N} L\big(y_i, f(x_i \mid w, b)\big)$

  Optimization Problem

Example of House Price

Living Area (sqft)	# of Beds	Has pool	Price (1000$)
2104	3	Yes	400
1600	3	No	330
2400	3	No	369
1416	2	No	232
3000	4	Yes	540

$\boldsymbol{x}=(x_1, x_2, x_3)'$

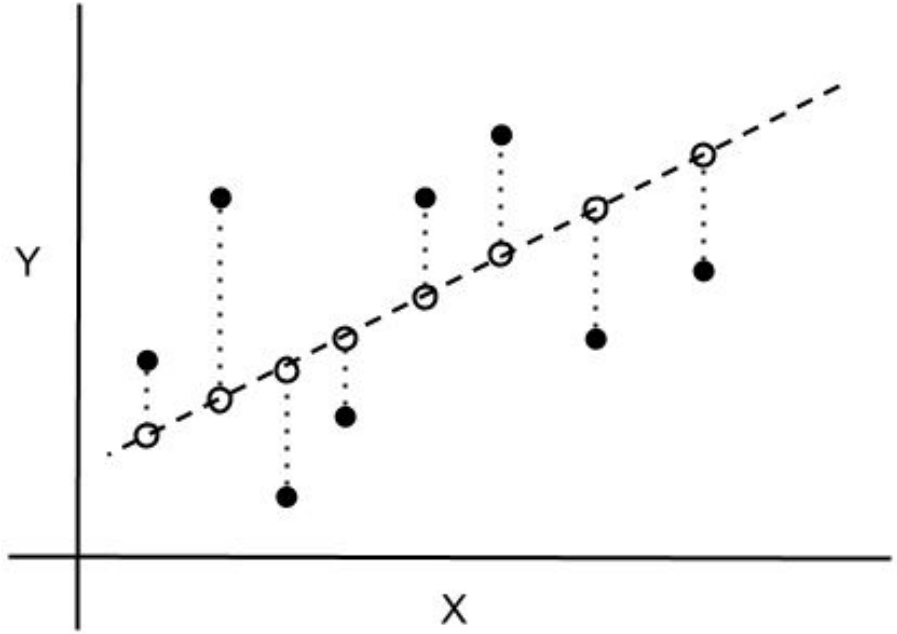
y

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3$$

Least Square Estimation

Loss function (Mean Square Error):

$$L(\boldsymbol{\beta}) = \frac{1}{2}\sum_i \left(\boldsymbol{x}_i^T\boldsymbol{\beta} - y_i\right)^2 / n$$



1. **Analytical solution** (you can directly calculate without optimization):

$$\widehat{\boldsymbol{\beta}} = (X^T X)^{-1} X^T y$$

2. For those problem that is hard to calculate it (large n & d, low-rank so non-invertible), we do **gradient-descent**

Gradient = dL / d (beta)

Bias Variance TradeOff

For each Dataset, **running model for multiple times can lead to different results** (with different observational noise, train-test split, etc)

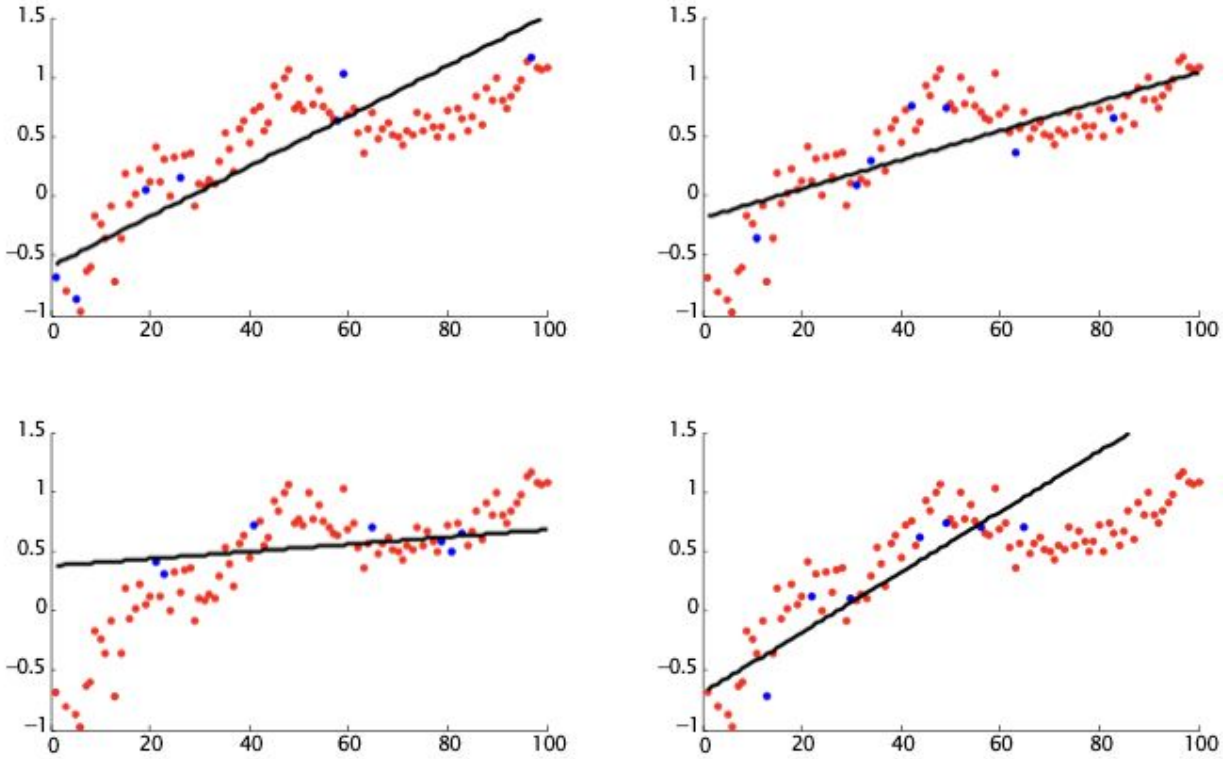
$$\mathrm{E}_{D,\varepsilon}\left[\left(y-\hat{f}\left(x;D\right)\right)^2\right]=\left(\mathrm{Bias}_D\left[\hat{f}\left(x;D\right)\right]\right)^2+\mathrm{Var}_D\left[\hat{f}\left(x;D\right)\right]+\sigma^2$$

where

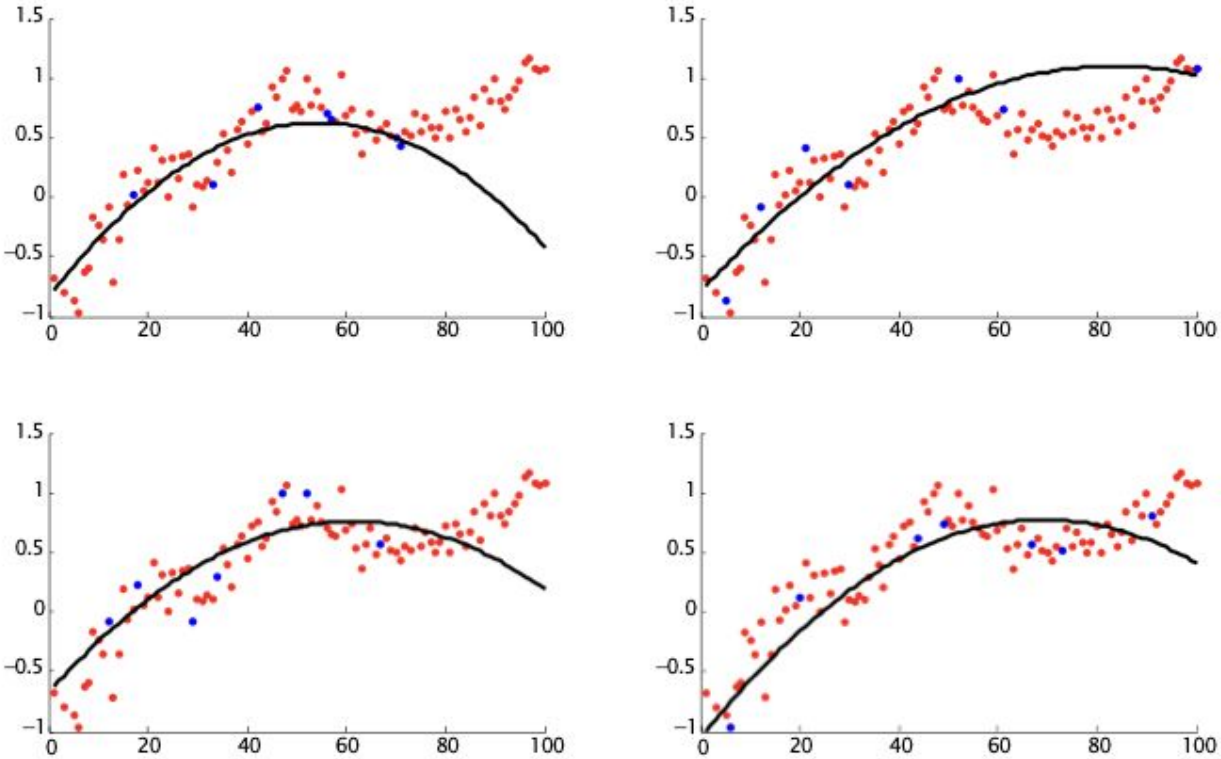
$$\mathrm{Bias}_D\left[\hat{f}\left(x;D\right)\right]=\mathrm{E}_D\left[\hat{f}\left(x;D\right)-f(x)\right]=\mathrm{E}_D\left[\hat{f}\left(x;D\right)\right]-\mathrm{E}_{y|x}\left[y(x)\right],$$

$$\mathrm{Var}_D\left[\hat{f}\left(x;D\right)\right]=\mathrm{E}_D[\left(\mathrm{E}_D[\hat{f}\left(x;D\right)]-\hat{f}\left(x;D\right)\right)^2].$$

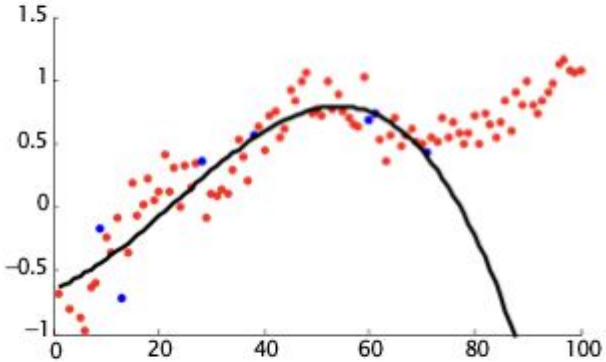
$h_S(x)$ Linear

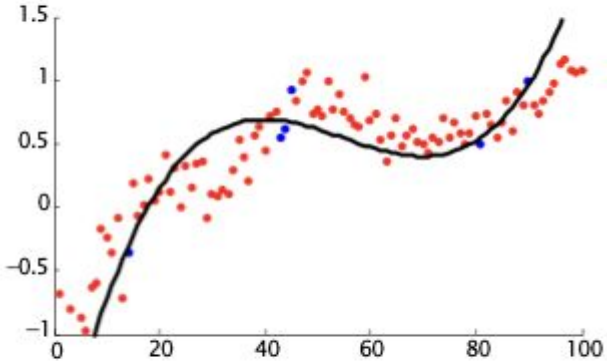


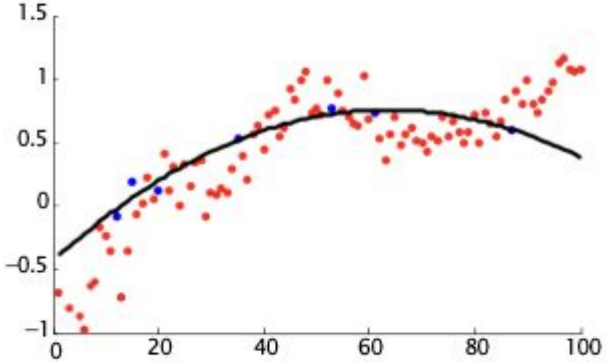
$h_S(x)$ Quadratic

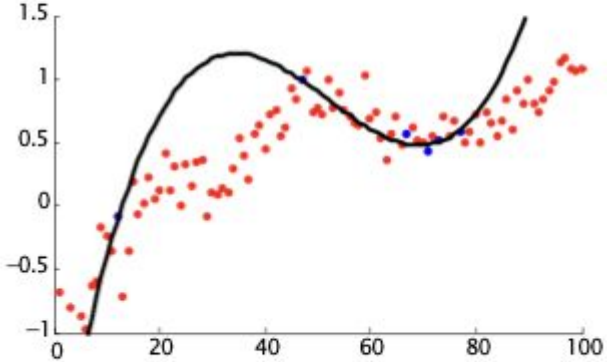


$h_S(x)$ Cubic

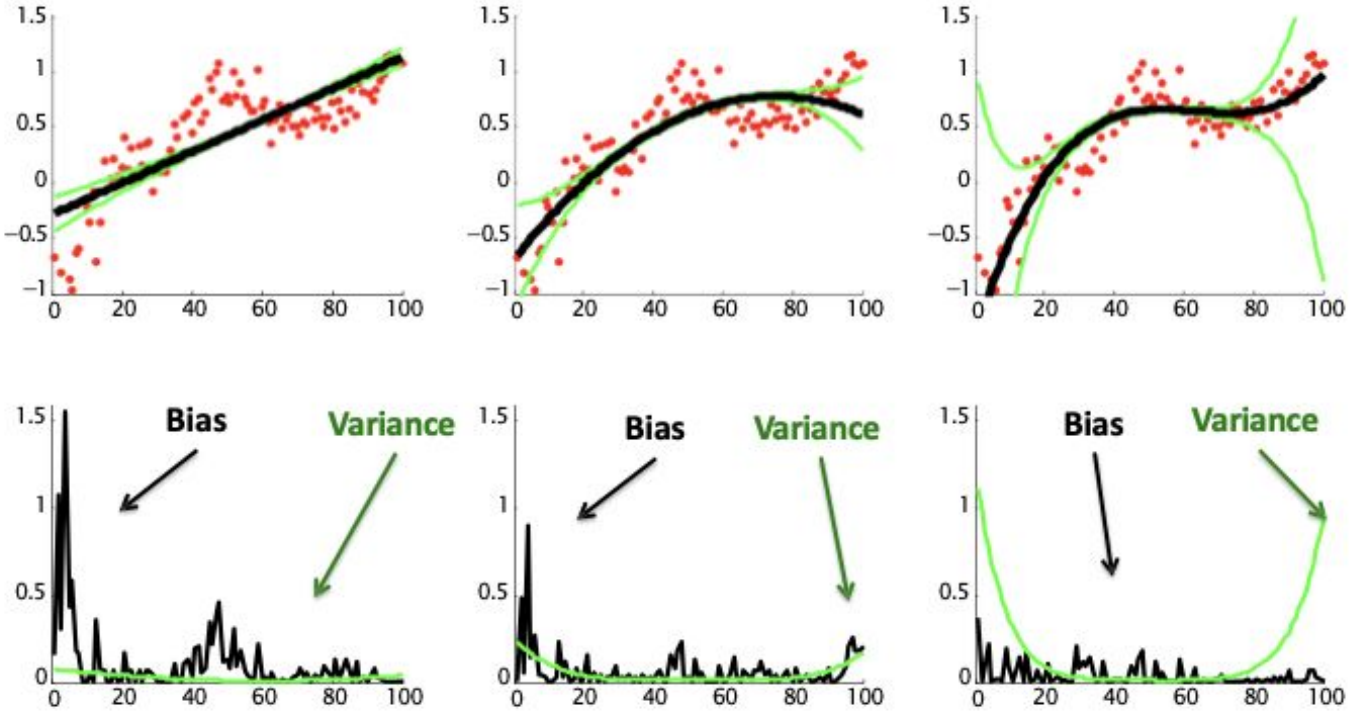

1.5
1
0.5
0
−0.5
−1
0
20
40
60
80
100


1.5
1
0.5
0
−0.5
−1
0
20
40
60
80
100

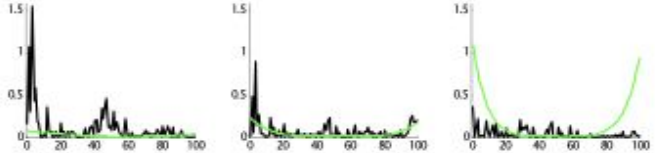

1.5
1
0.5
0
−0.5
−1
0
20
40
60
80
100


1.5
1
0.5
0
−0.5
−1
0
20
40
60
80
100

Bias-Variance Trade-off



Overfitting vs Underfitting



- High variance implies **overfitting**
  - Model class unstable
  - Variance increases with model complexity
  - Variance reduces with more training data.
- High bias implies **underfitting**
  - Even with no variance, model class has high error
  - Bias decreases with model complexity
  - Independent of training data size

Demo of Bias-Variance Tradeoff

$$\mathrm{MSE} \triangleq \mathrm{E}\left[(y - \hat{f})^2\right] = \mathrm{E}\left[y^2 - 2y\hat{f} + \hat{f}^2\right] = \mathrm{E}\left[y^2\right] - 2\,\mathrm{E}\left[y\hat{f}\right] + \mathrm{E}\left[\hat{f}^2\right]$$

Firstly, since we model $y = f + \varepsilon$, we show that

$$\begin{aligned}
\mathrm{E}\left[y^2\right] &= \mathrm{E}\left[(f+\varepsilon)^2\right] \\
&= \mathrm{E}[f^2] + 2\,\mathrm{E}[f\varepsilon] + \mathrm{E}[\varepsilon^2] && \text{by linearity of } \mathrm{E} \\
&= f^2 + 2f\,\mathrm{E}[\varepsilon] + \mathrm{E}[\varepsilon^2] && \text{since } f \text{ does not depend on the data} \\
&= f^2 + 2f \cdot 0 + \sigma^2 && \text{since } \varepsilon \text{ has zero mean and variance } \sigma^2
\end{aligned}$$

Secondly,

$$\begin{aligned}
\mathrm{E}\left[y\hat{f}\right] &= \mathrm{E}\left[(f+\varepsilon)\hat{f}\right] \\
&= \mathrm{E}[f\hat{f}] + \mathrm{E}[\varepsilon\hat{f}] && \text{by linearity of } \mathrm{E} \\
&= \mathrm{E}[f\hat{f}] + \mathrm{E}[\varepsilon]\,\mathrm{E}[\hat{f}] && \text{since } \hat{f} \text{ and } \varepsilon \text{ are independent} \\
&= f\,\mathrm{E}[\hat{f}] && \text{since } \mathrm{E}[\varepsilon] = 0
\end{aligned}$$

Lastly,

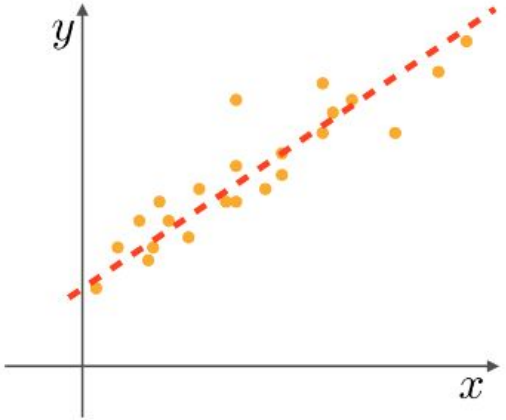
$$\mathrm{E}\left[\hat{f}^2\right] = \mathrm{Var}(\hat{f}) + \mathrm{E}[\hat{f}]^2 \qquad \text{since } \mathrm{Var}[X] \triangleq \mathrm{E}\left[(X - \mathrm{E}[X])^2\right] = \mathrm{E}[X^2] - \mathrm{E}[X]^2 \text{ for any random variable } X$$

Eventually, we plug these 3 formulas in our previous derivation of MSE and thus show that:

$$\begin{aligned}
\mathrm{MSE} &= f^2 + \sigma^2 - 2f\,\mathrm{E}[\hat{f}] + \mathrm{Var}[\hat{f}] + \mathrm{E}[\hat{f}]^2 \\
&= (f - \mathrm{E}[\hat{f}])^2 + \sigma^2 + \mathrm{Var}\left[\hat{f}\right] \\
&= \mathrm{Bias}[\hat{f}]^2 + \sigma^2 + \mathrm{Var}\left[\hat{f}\right]
\end{aligned}$$

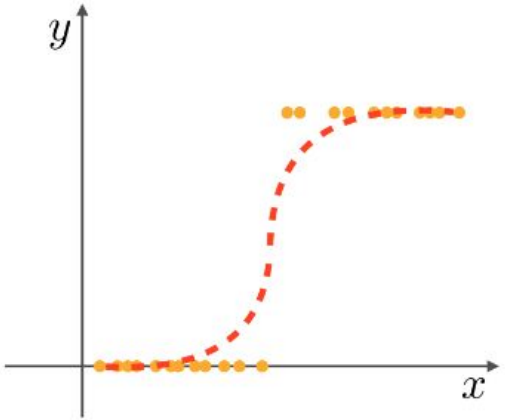
Logistic Regression (Classification)

y is continuous



regression

y is binary or categorical



classification

Linear Regression VS. Logistic Regression

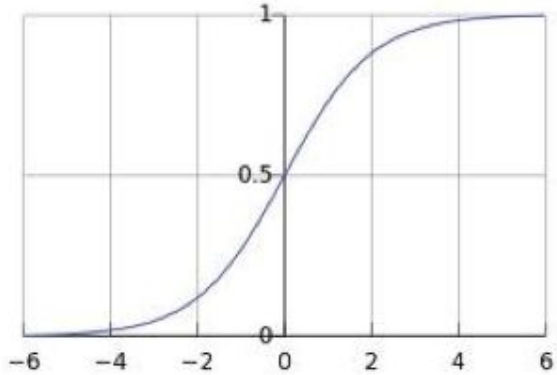
- Linear Regression (prediction)
  - Y: *continuous value* $(-\infty, +\infty)$
    - $Y = \boldsymbol{x}^T\boldsymbol{\beta} = \beta_0 + x_1\beta_1 + x_2\beta_2 + \cdots + x_p\beta_p$
    - $Y|\boldsymbol{x}, \beta \sim N(\boldsymbol{x}^T\beta, \sigma^2)$

- Logistic Regression (classification)
  - Y: *discrete value from m classes*
    - $p(Y = C_j|\boldsymbol{x}, \beta) \in [0,1]$ *and* $\sum_j p(Y = C_j|\boldsymbol{x}, \beta) = 1$

Logistic Function

- Logistic Function / sigmoid function:

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



$$Note: \sigma'(x) = \sigma(x)(1-\sigma(x))$$

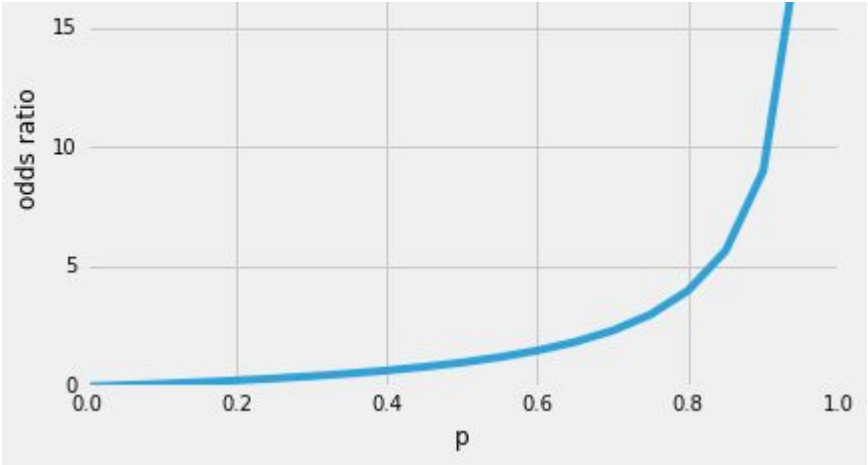
Modeling Probabilities of Two Classes

- $P(Y = 1|\boldsymbol{x}, \beta) = \sigma(\boldsymbol{x}^T\beta) = \frac{1}{1+\exp\{-\boldsymbol{x}^T\beta\}} = \frac{\exp\{\boldsymbol{x}^T\beta\}}{1+\exp\{\boldsymbol{x}^T\beta\}}$
- $P(Y = 0|\boldsymbol{x}, \beta) = 1 - \sigma(\boldsymbol{x}^T\beta) = \frac{\exp\{-\boldsymbol{x}^T\beta\}}{1+\exp\{-\boldsymbol{x}^T\beta\}} = \frac{1}{1+\exp\{\boldsymbol{x}^T\beta\}}$

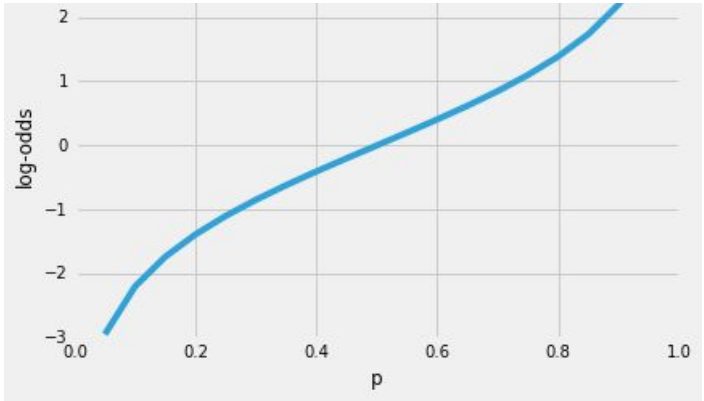
$$\beta = \begin{pmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_p \end{pmatrix}$$

- In other words
  - $y|\boldsymbol{x}, \beta \sim Bernoulli(\sigma(\boldsymbol{x}^T\beta))$

Why Sigmoid?



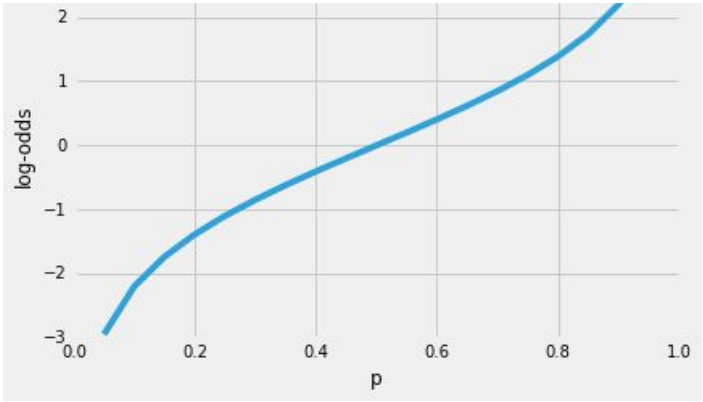
“odds ratio” *p / (1 - p)*: describes the ratio between the probability that a event occurs to the probability that it doesn’t occur

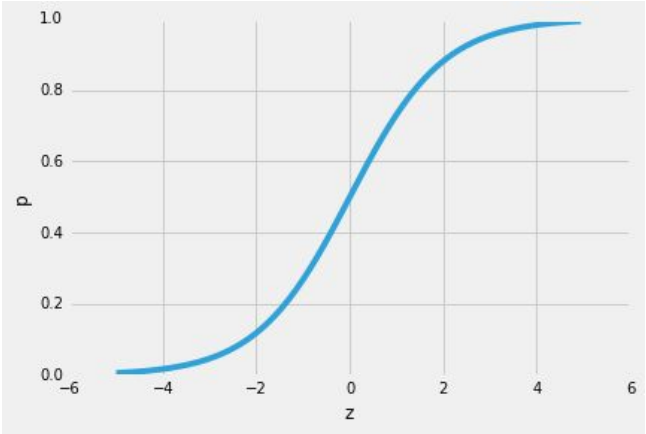


Take log to odd become logit:

$$logit(p(y=1|\mathbf{x})) = log\left(\frac{p}{1-p}\right) = log(p) - log(1-p)$$

Why Sigmoid?





Take log to odd become logit:

$$logit(p(y=1|\mathbf{x})) = log\left(\frac{p}{1-p}\right) = log(p) - log(1-p)$$

Use Linear Transformation to model logit

$$logit(p(y=1|\mathbf{x})) = log\left(\frac{p}{1-p}\right) = log(p) - log(1-p) = b + w_1x_1 + \ldots + w_mx_m$$

After some transformation we have:

$$\log\frac{p}{1-p} = f(x) = W^T x$$

$$\frac{p}{1-p} = e^{f(x)}$$

$$\frac{1}{p} - 1 = e^{-f(x)}$$

$$p = \frac{1}{1+e^{-f(x)}} = \text{sigmoid}(f(x))$$

Maximum Likelihood Estimation

- Given a dataset D, *with* N *data points*
- For a single data object with attributes $\boldsymbol{x}_i$, class label y_i
  - Let $p_i = p(y_i = 1|\boldsymbol{x}_i, \beta)$, *the prob. of* i *in class* 1
  - The probability of observing y_i would be
    - If $y_i = 1$, *then* p_i
    - If $y_i = 0$, *then* $1 - p_i$

$$\text{maximize Likelihood}(\beta) = \prod_{y_i=1} p(x_i) \cdot \prod_{y_j=0} (1 - p(x_j))$$

$$= \prod_{(x_i, y_i) \in \mathcal{D}} p(x_i)^{y_i} \cdot (1 - p(x_i))^{1-y_i}$$

Maximum Likelihood Estimation

$$\text{maximize Likelihood}(\beta) = \prod_{y_i=1} p(x_i) \cdot \prod_{y_j=0} (1 - p(x_j))$$

$$= \prod_{(x_i,y_i)\in\mathcal{D}} p(x_i)^{y_i} \cdot (1 - p(x_i))^{1-y_i}$$

$$\text{log-Likelihood}(\beta) = \sum_{(x_i,y_i)\in\mathcal{D}} y_i \cdot \log p(x_i) + (1 - y_i) \cdot \log(1 - p(x_i))$$

Also known as (binary) **cross-entropy**

Maximum Likelihood Estimation

Put in $p = \frac{1}{1+e^{-f(x)}} = \text{sigmoid}(f(x))$

$$\log\text{-Likelihood}(\beta) = \sum_{(x_i,y_i)\in\mathcal{D}} y_i \cdot \log p(x_i) + (1-y_i)\cdot\log(1-p(x_i)) \tag{7}$$

$$= \sum_{(x_i,y_i)\in\mathcal{D}} y_i \cdot \log \frac{1}{1+e^{-f(x)}} + (1-y_i)\cdot\log\frac{e^{-f(x)}}{1+e^{-f(x)}} \tag{8}$$

$$= \sum_{(x_i,y_i)\in\mathcal{D}} y_i \cdot \log\left(\frac{1}{1+e^{-f(x)}}\cdot\frac{1+e^{-f(x)}}{e^{-f(x)}}\right) + \log\frac{e^{-f(x)}}{1+e^{-f(x)}}$$

$$= \sum_{(x_i,y_i)\in\mathcal{D}} y_i \cdot f(x) - \log(1+e^{f(x)}) \tag{9}$$

What about Multiclass Classification?

- It is easy to handle under logistic regression, say M classes, using softmax function

$$p(y = k; \beta) = \text{Softmax}(x)_k = \frac{e^{f_k(x)}}{\sum_j e^{f_j(x)}} \quad , \text{where } k \in [1, 2, \cdots, K]$$

Easy to find that sigmoid is a special extension of softmax when K = 2

$$p(y = 1; \beta) = \frac{e^{f(x)}}{e^{f(x)} + e^{g(x)}} = \frac{1}{1 + e^{\{g(x) - f(x)\}}} = \text{sigmoid}(g(x) - f(x))$$

Maximum Likelihood Estimation

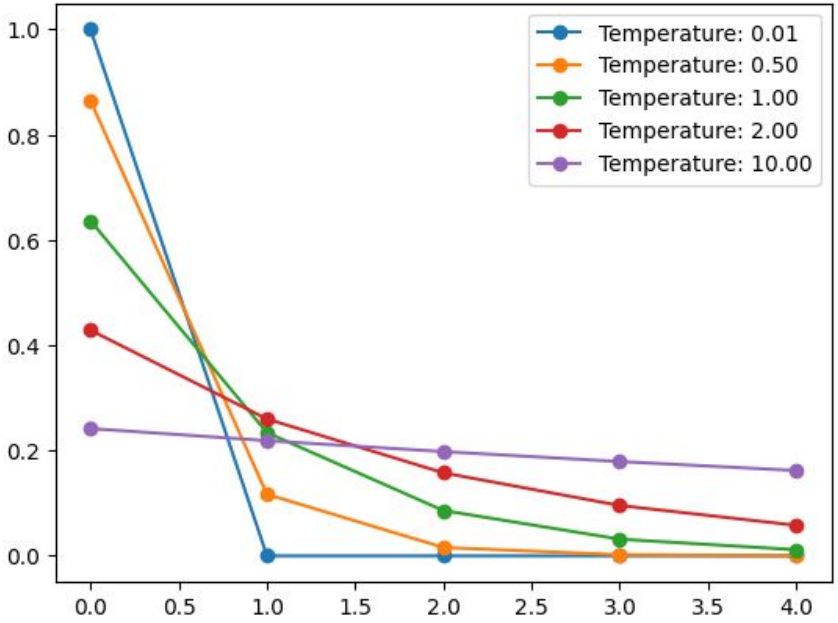
$$\text{maximize Likelihood}(\beta) = \prod_{(x_i, y_i) \in \mathcal{D}} p(y_i = k; \beta)^{\mathcal{I}(y_i = k)}$$

$$\log\text{-Likelihood}(\beta) = \sum_{(x_i, y_i) \in \mathcal{D}} \log p(y_i = k; \beta) \cdot \mathcal{I}(y_i = k)$$

Treat ground-truth label y's probability as q
Sum [log (p) * q] = E_q[log p] is also known as **cross-entropy** **H(q; p)**

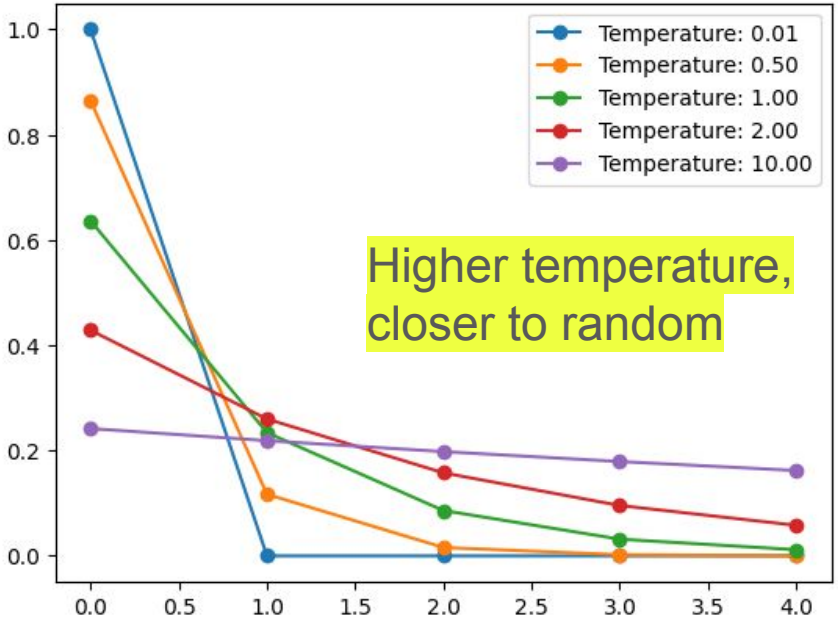
Softmax with different temperature

$$p(y=k;\beta)=\frac{e^{[f_k(x)/\mathbf{temperature}]}}{\sum_j e^{[f_j(x)/\mathbf{temperature}]}}=\frac{e^{[f_k(x)/\tau]}}{\sum_j e^{[f_j(x)/\tau]}}$$



Softmax with different temperature

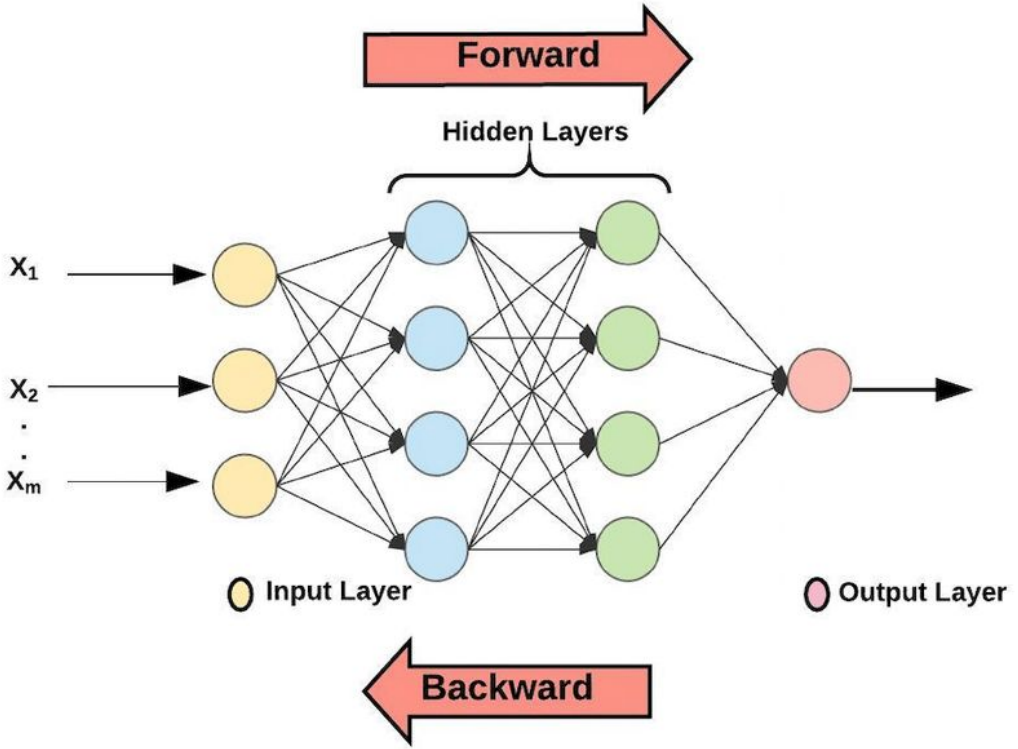
$$p(y=k;\beta)=\frac{e^{[f_k(x)/\textbf{temperature}]}}{\sum_j e^{[f_j(x)/\textbf{temperature}]}}=\frac{e^{[f_k(x)/\tau]}}{\sum_j e^{[f_j(x)/\tau]}}$$



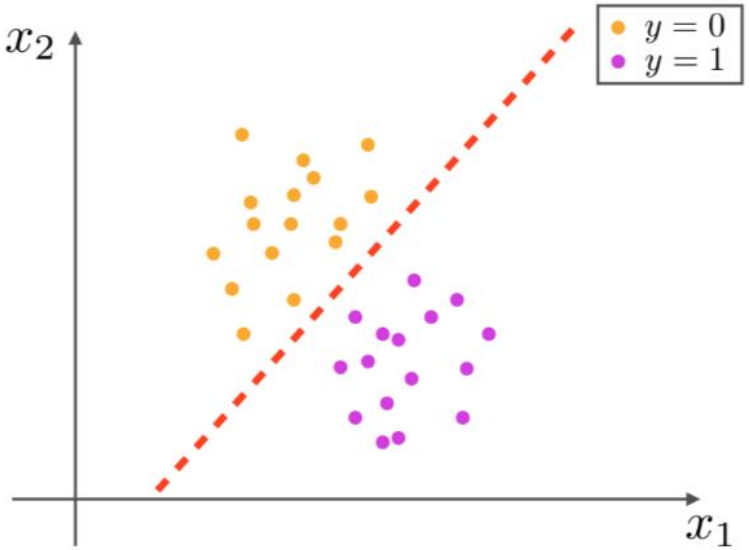
Why we need to minus max?

```python
import numpy as

def softmax_with_temperature(z, temperature=1.0):
    z_temp = z / temperature
    e_z_temp = np.exp(z_temp - np.max(z_temp))   # for n
    return e_z_temp / e_z_temp.sum(axis=0)

z = np.array([2.0, 1.0, 0.1, 0, -2])
for temp in [0.01, 0.5, 1, 2, 10]:
  out = softmax_with_temperature(z, temperature = temp)
  plt.plot(out, label='Temperature: %.2f' % temp, marke
plt.legend()
plt.show()
```

Multi-Layer Perceptron (MLP)



classification example



$$\mathbf{x} = (x_1, x_2)$$

logistic regression

regress to the logistic transform

linear decision boundary

$$\log \frac{p(y=1|\mathbf{x})}{1-p(y=1|\mathbf{x})} = \mathbf{w}^\intercal \mathbf{x} + b$$

$$\rightarrow \quad p(y=1|\mathbf{x}) = \frac{1}{1+e^{-(\mathbf{w}^\intercal \mathbf{x}+b)}}$$

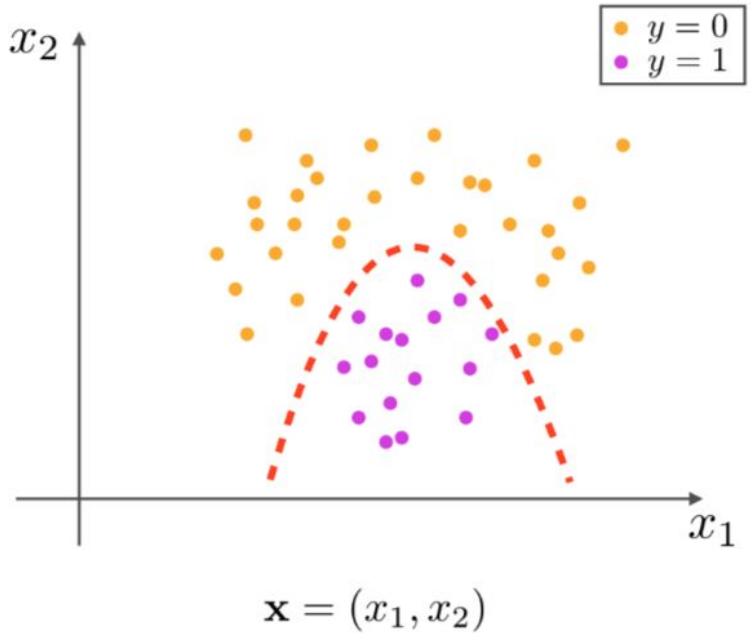
minimize the *binary cross entropy* loss function $\mathcal{L}$ to find the optimal $\mathbf{w}$ and b.

gradient descent

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla_{\mathbf{w}} \mathcal{L}$$

$$b \leftarrow b - \alpha \frac{\partial \mathcal{L}}{\partial b}$$

classification example



we need a ***non-linear*** decision boundary

option 1: use non-linear terms,

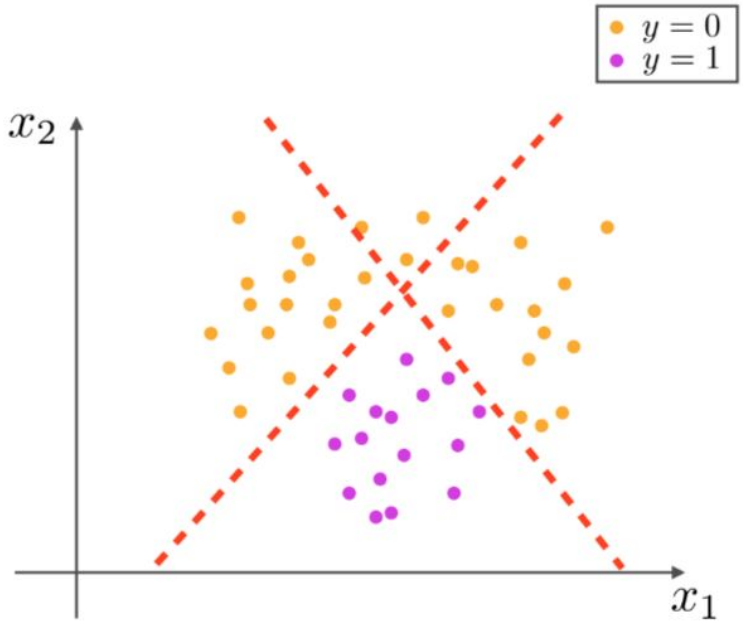
expand $\mathbf{x}$ and $\mathbf{w}$

$$(x_1, x_2) \rightarrow (x_1^2, x_2^2, x_1 x_2, x_1, x_2)$$

More complex feature, higher chance of overfitting;

Can we automatically learn the best feature for each task?

classification example



$$\mathbf{x} = (x_1, x_2)$$

we need a ***non-linear*** decision boundary

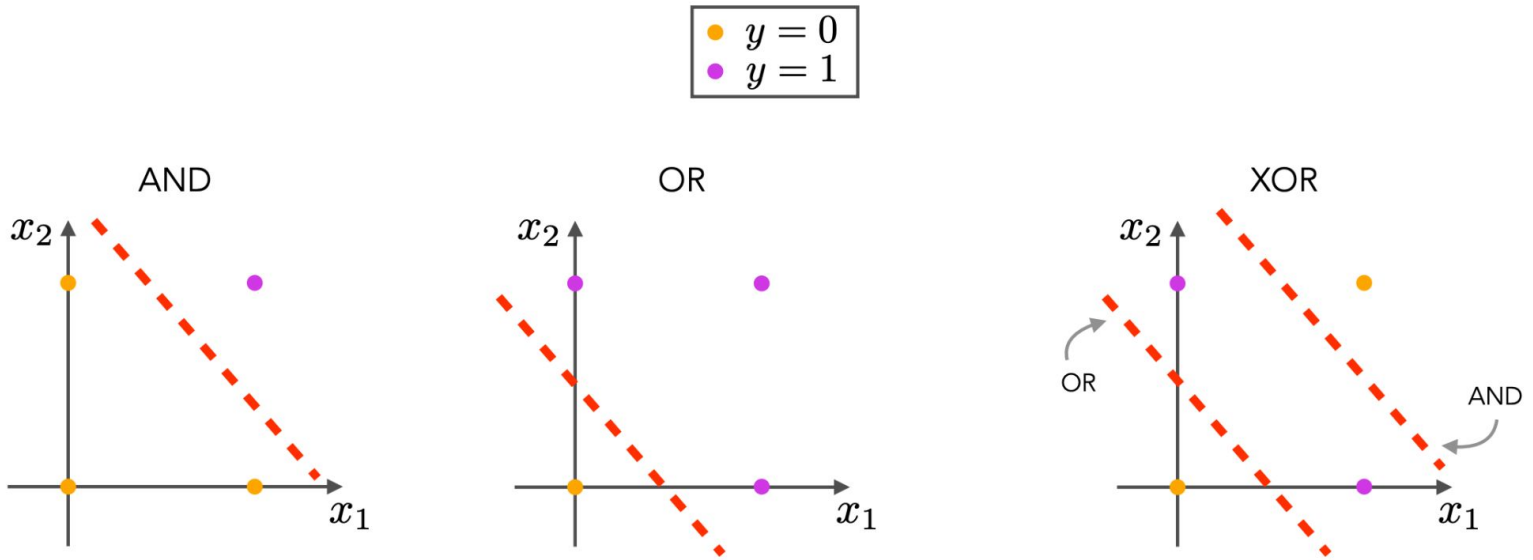
option 1: use non-linear terms, expand $\mathbf{x}$ and $\mathbf{w}$

$$(x_1, x_2) \rightarrow (x_1^2, x_2^2, x_1 x_2, x_1, x_2)$$

option 2: use multiple linear decision boundaries to compose a non-linear boundary

in both cases, transform the data into a representation that is linearly separable

boolean operations



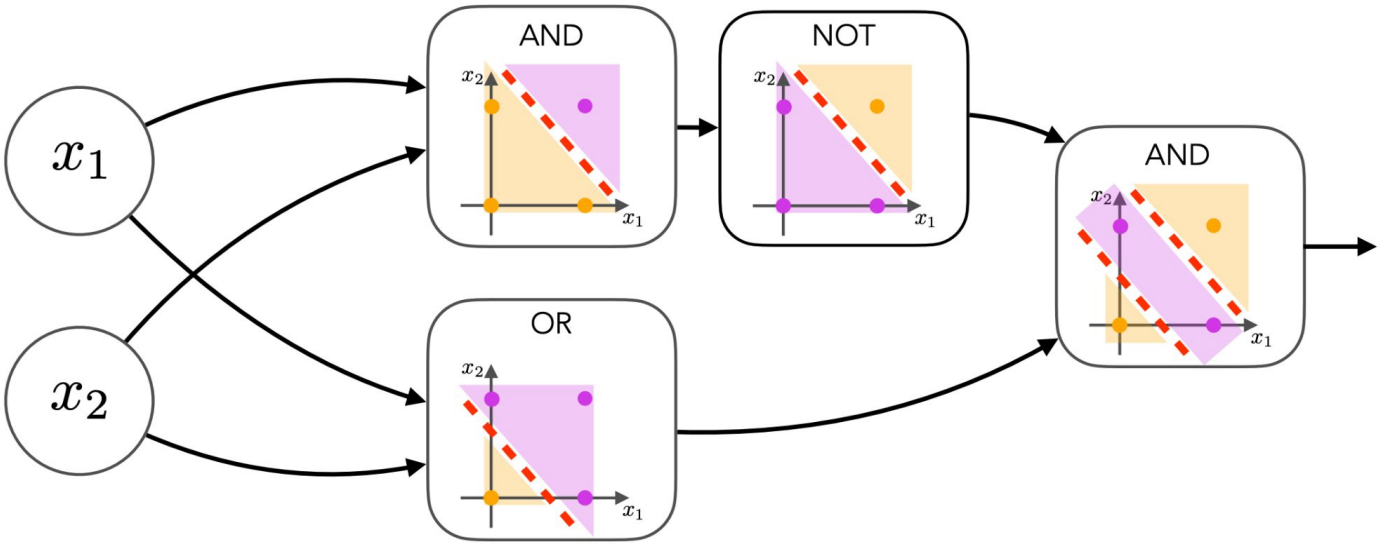
AND and OR are both linearly separable

XOR is not linearly separable, but can be separated using AND and OR

boolean operations

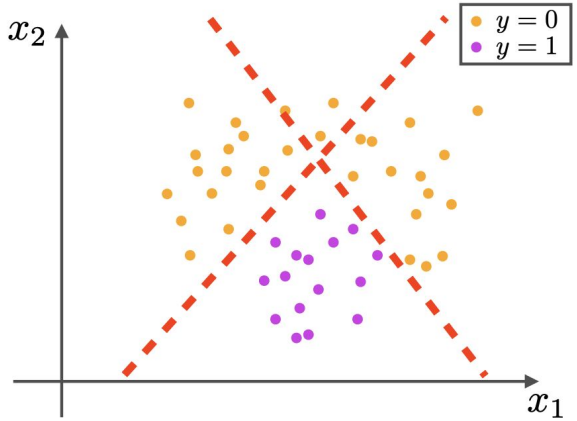
building XOR from AND and OR

*composing **non-linear** boundaries from **linear** boundaries*



recapitulation

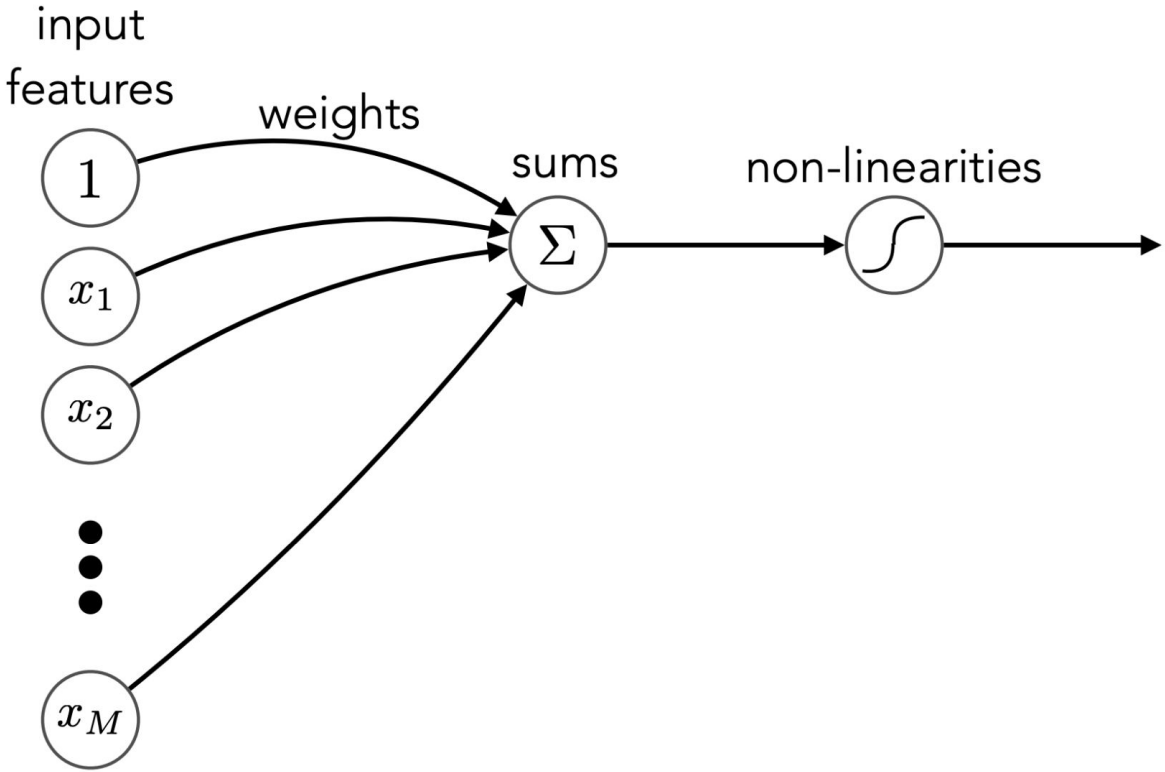
to fit more complex data, we need more expressive ***non-linear*** functions

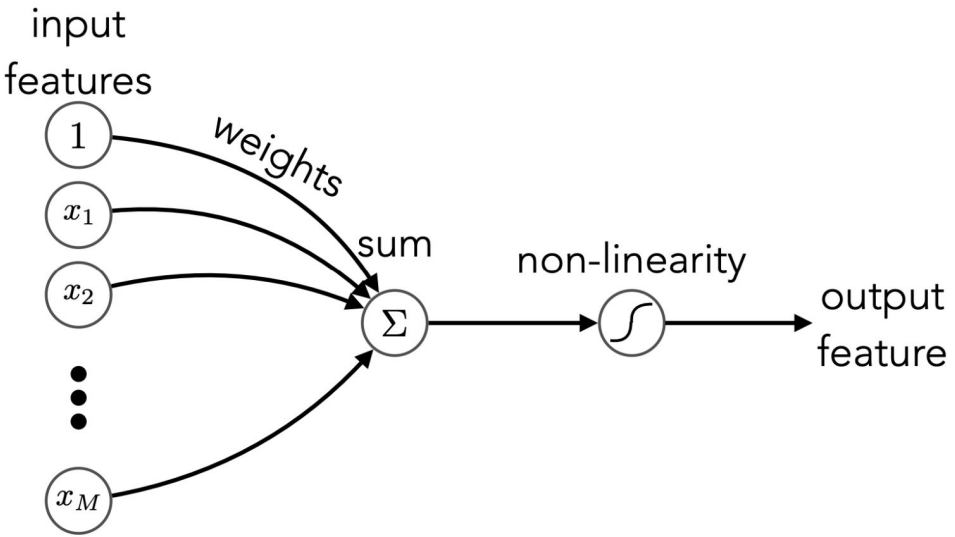


we can form non-linear functions by composing stages of processing

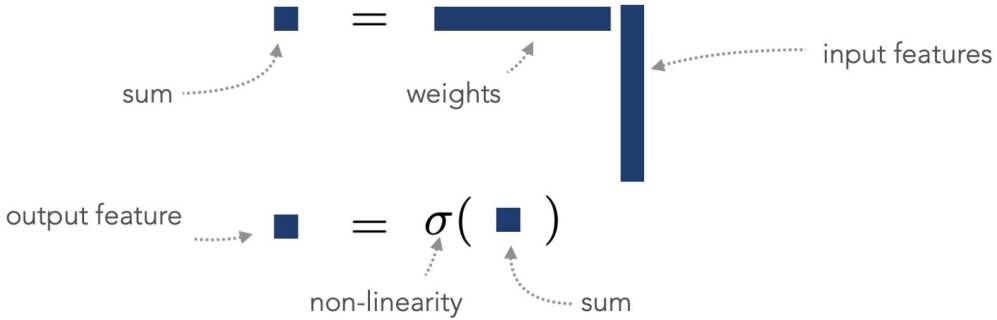
depth: the number of stages of processing

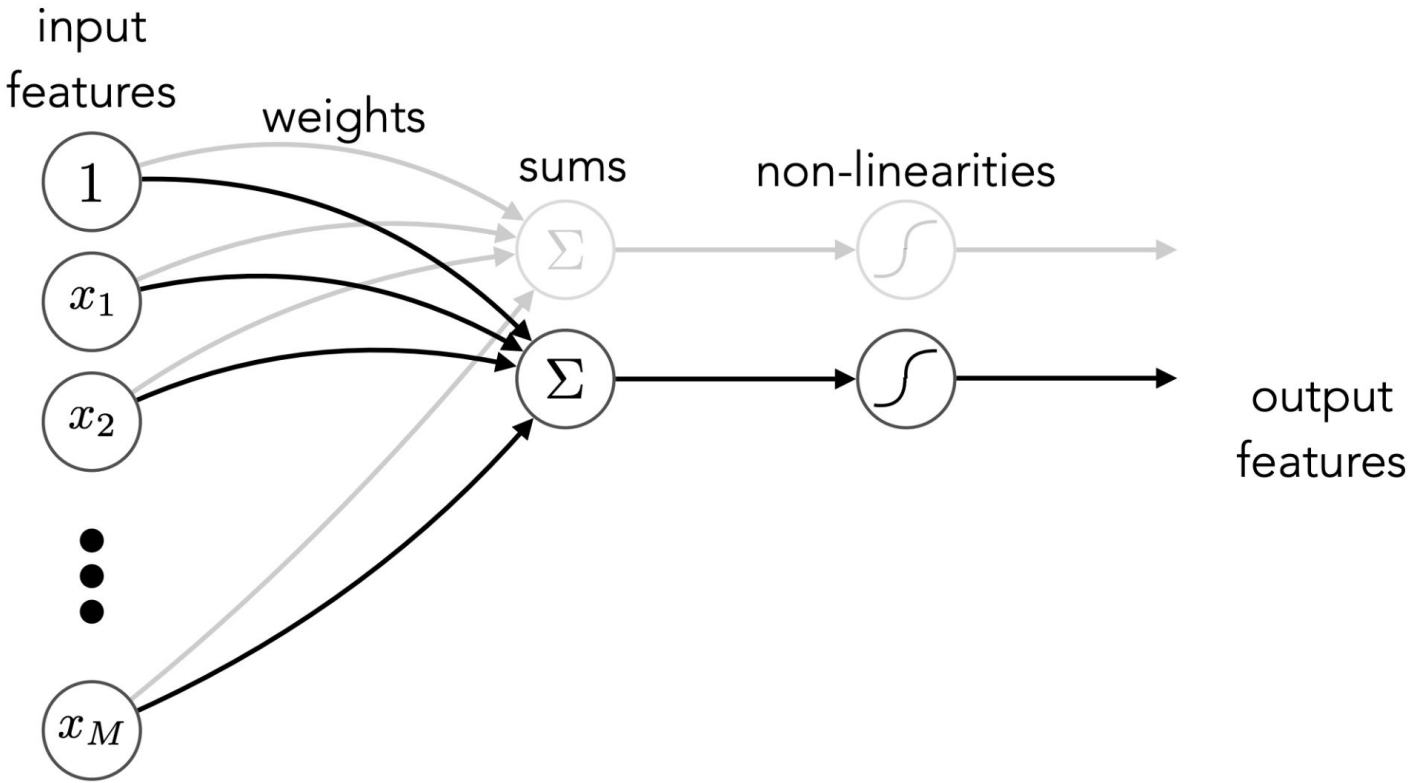
deep learning: learning functions with multiple stages of processing

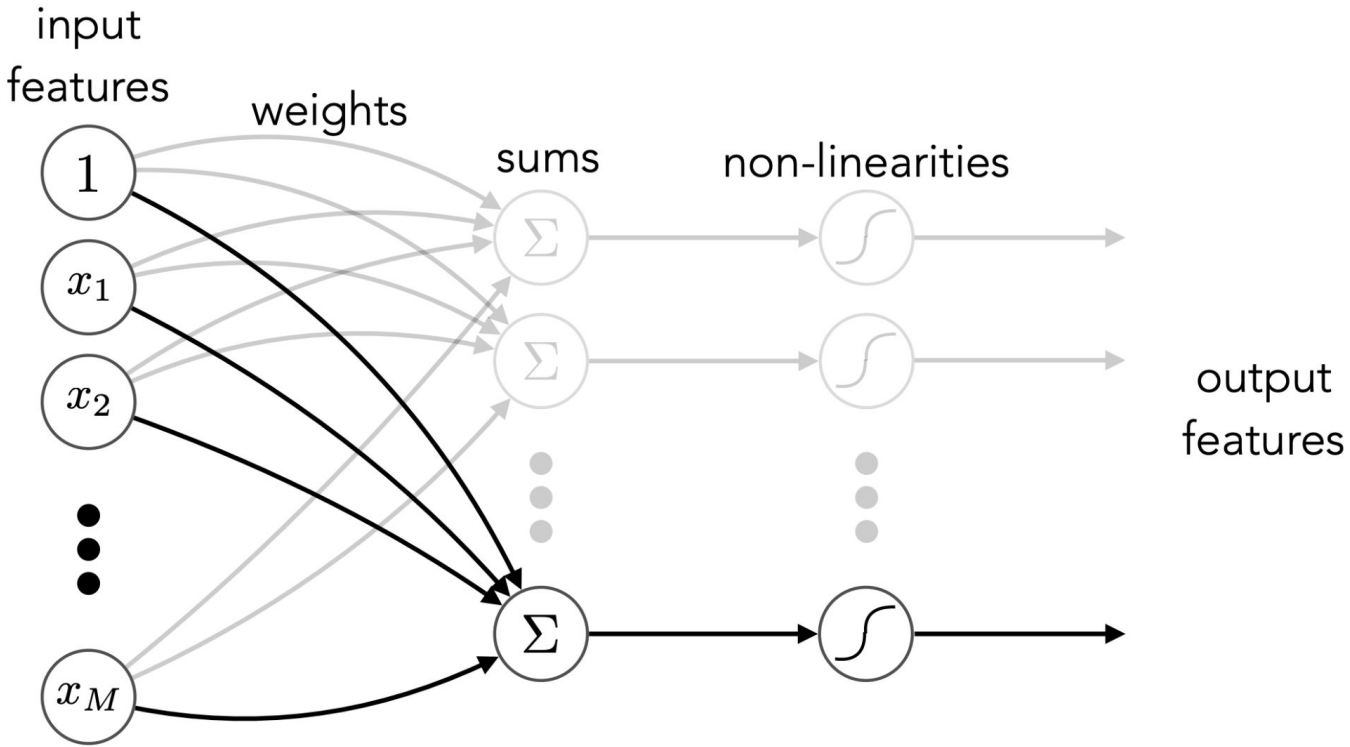

input
features
weights
sums
non-linearities
1
x_1
x_2
x_M
Σ



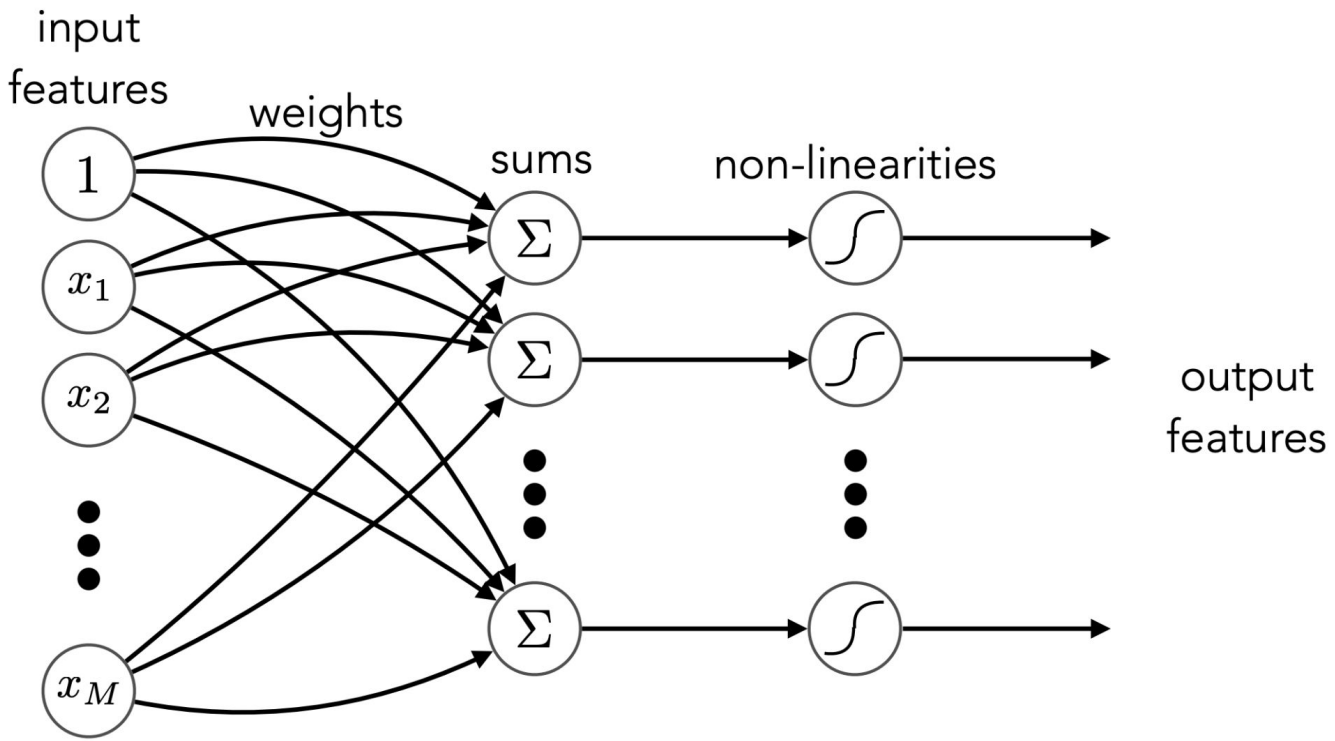
artificial neuron: *weighted sum and non-linearity*

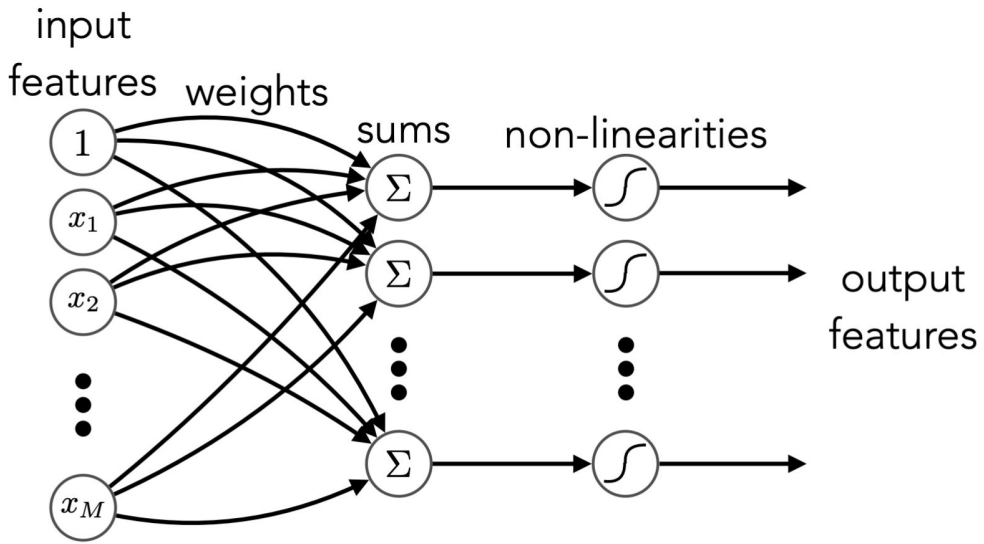



input
features
weights
sums
non-linearities
1
x_1
x_2
x_M
Σ
Σ
output
features

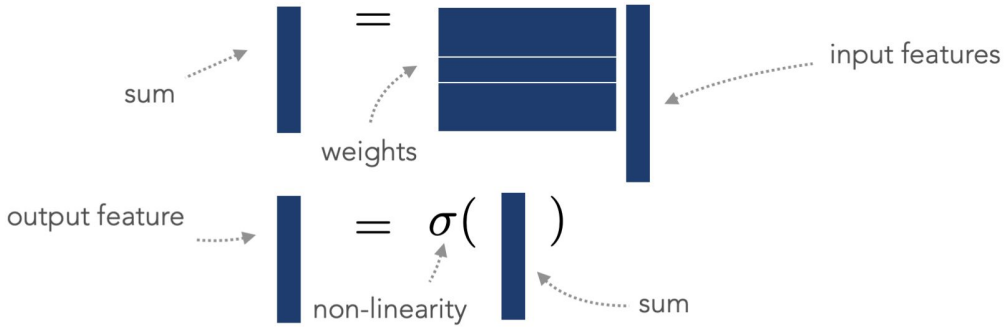

input
features
weights
sums
non-linearities
output
features
1
x_1
x_2
x_M
Σ
Σ
Σ
$\int$
$\int$
$\int$

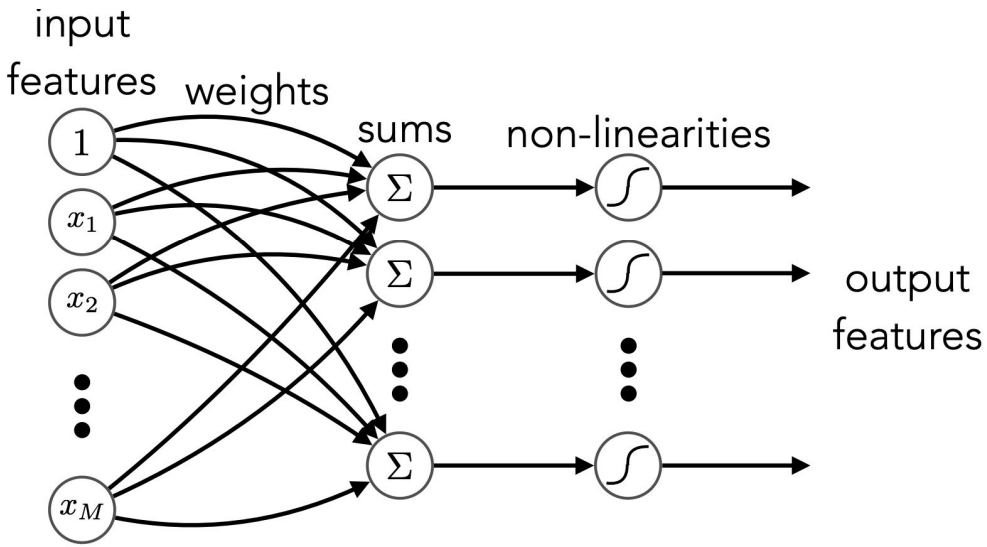
multiple neurons form a **layer**





layer: *parallelized weighted sum and non-linearity*

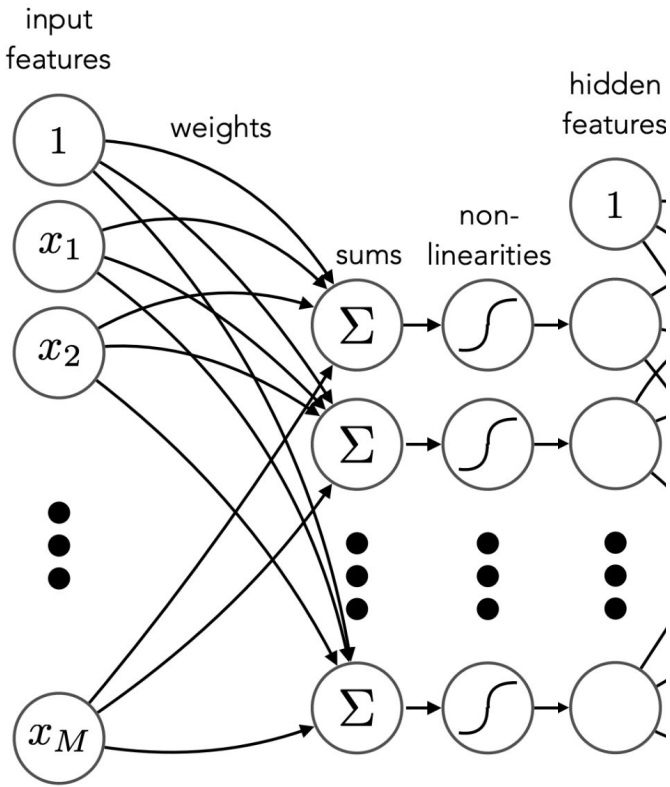




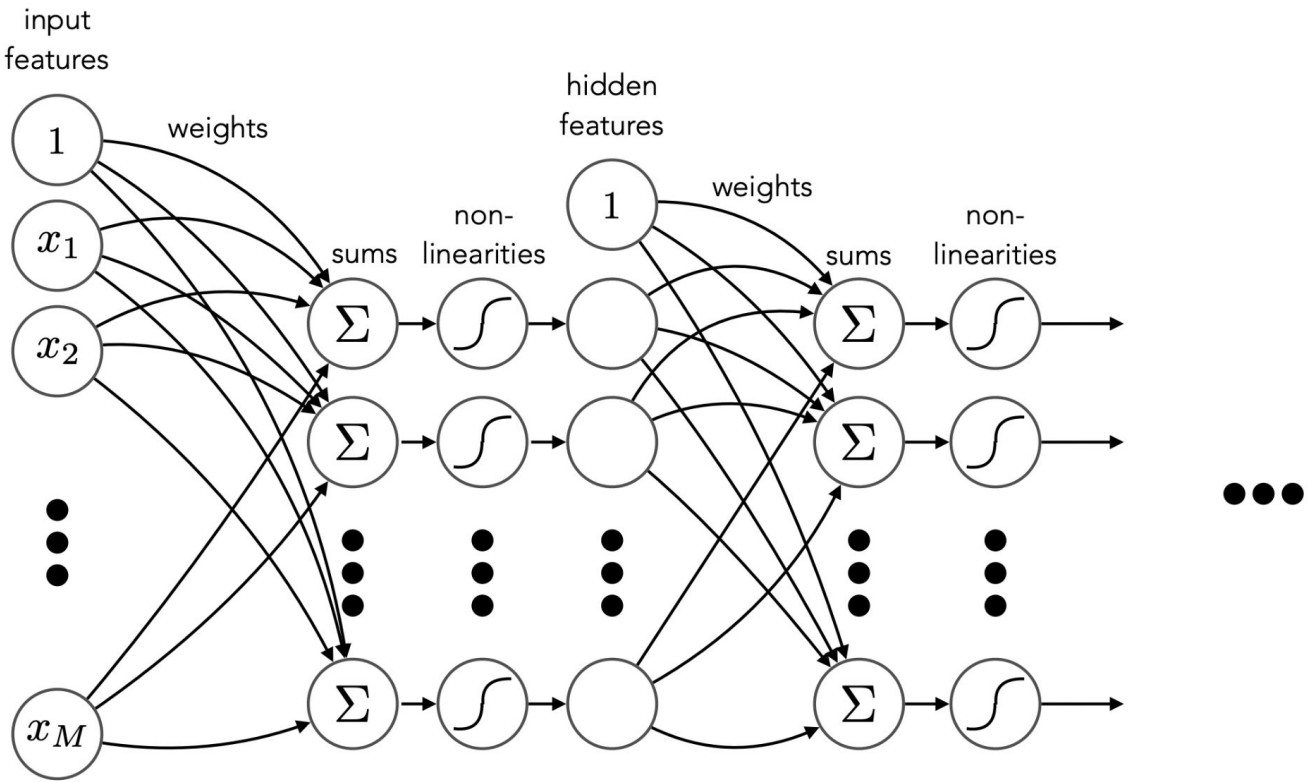
layer: *parallelized weighted sum and non-linearity*

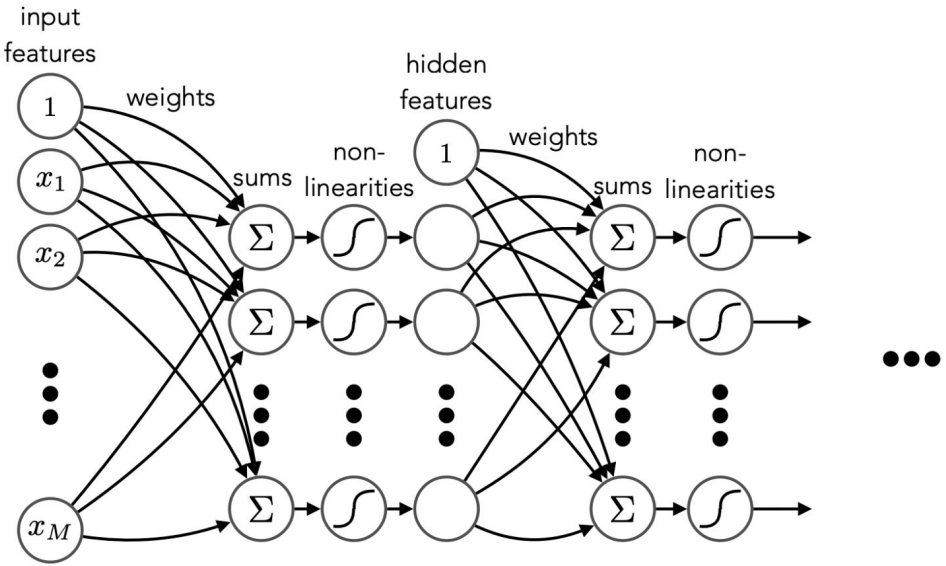
one sum per weight *vector* $s_j = \mathbf{w}_j^\intercal \mathbf{x}$ $\longrightarrow$ $\mathbf{s} = \mathbf{W}^\intercal \mathbf{x}$ vector of sums from weight *matrix*

$$\mathbf{h} = \sigma(\mathbf{s})$$


input
features
weights
hidden
features
1
x_1
x_2
x_M
sums
non-
linearities
Σ
Σ
Σ
1

multiple layers form a **network**





network: *sequence of parallelized weighted sums and non-linearities*

DEFINE $\mathbf{x}^{(0)} \equiv \mathbf{x}$, $\mathbf{x}^{(1)} \equiv \mathbf{h}$, ETC.

1st layer

$$\mathbf{s}^{(1)} = \mathbf{W}^{(1)\intercal}\mathbf{x}^{(0)}$$

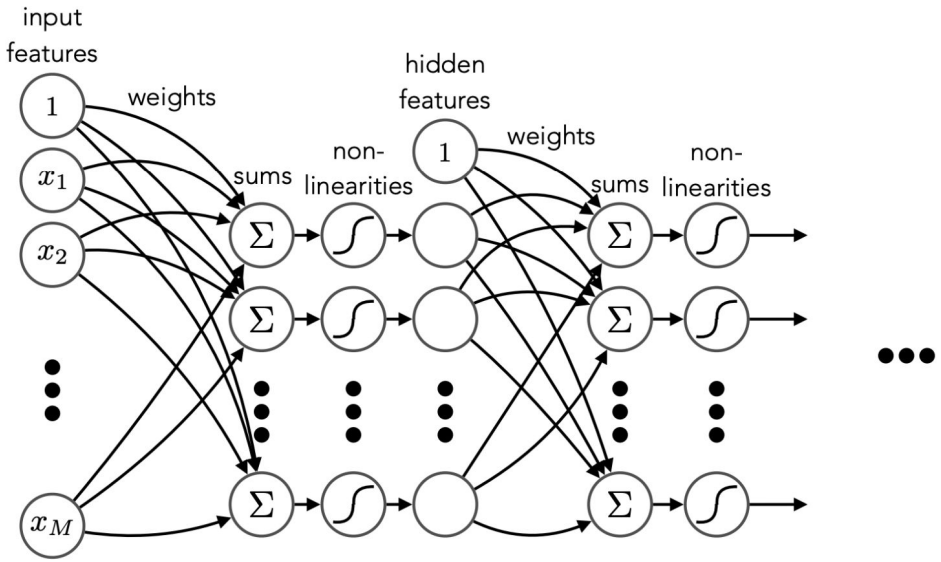
$$\mathbf{x}^{(1)} = \sigma(\mathbf{s}^{(1)})$$

2nd layer

$$\mathbf{s}^{(2)} = \mathbf{W}^{(2)\intercal}\mathbf{x}^{(1)}$$

$$\mathbf{x}^{(2)} = \sigma(\mathbf{s}^{(2)})$$

...



network: *sequence of parallelized weighted sums and non-linearities*

$$= \sigma(\cdots \sigma(\quad \sigma(\quad)) \cdots)$$

output

2nd weights

1st weights

input

big picture

neural networks are functions / function approximators

their *nested* (deep) structure enables a broader set of functions

$$\begin{aligned}\text{output} &= \text{NN}(\text{inputs}) \\ &= \text{Layer}_L(\text{Layer}_{L-1}(\ldots \text{Layer}_1(\text{inputs})\ldots))\end{aligned}$$

deep networks are *universal function approximators* *(Hornik, 1991)*

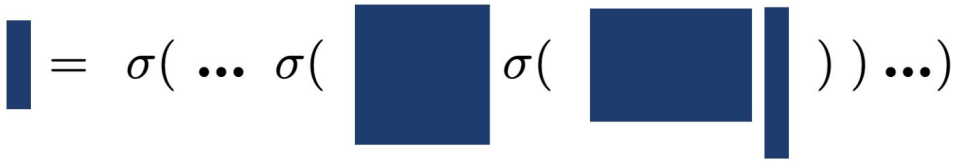
→ with enough units & layers, can approximate *any* function

Therefore, each intermediate representation h can be treated as automatically “learned” feature h = f(x)

non-linearities

the non-linearities are ***essential***

without them, the network collapses to a linear function

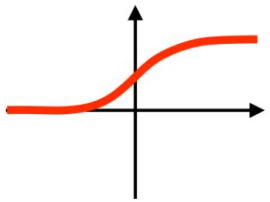


different non-linearities result in different functions and optimization surfaces

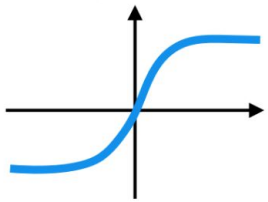
non-linearities

"old school"

logistic sigmoid



hyperbolic tangent (tanh)

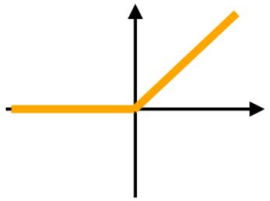


saturating

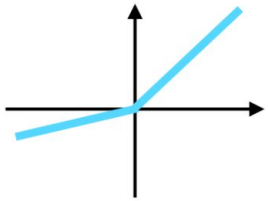
derivative goes to zero at +∞ and -∞

"new school"

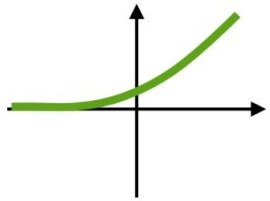
rectified linear unit (ReLU)



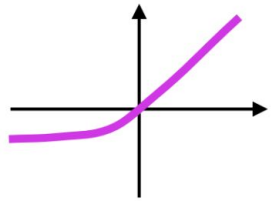
leaky ReLU



softplus



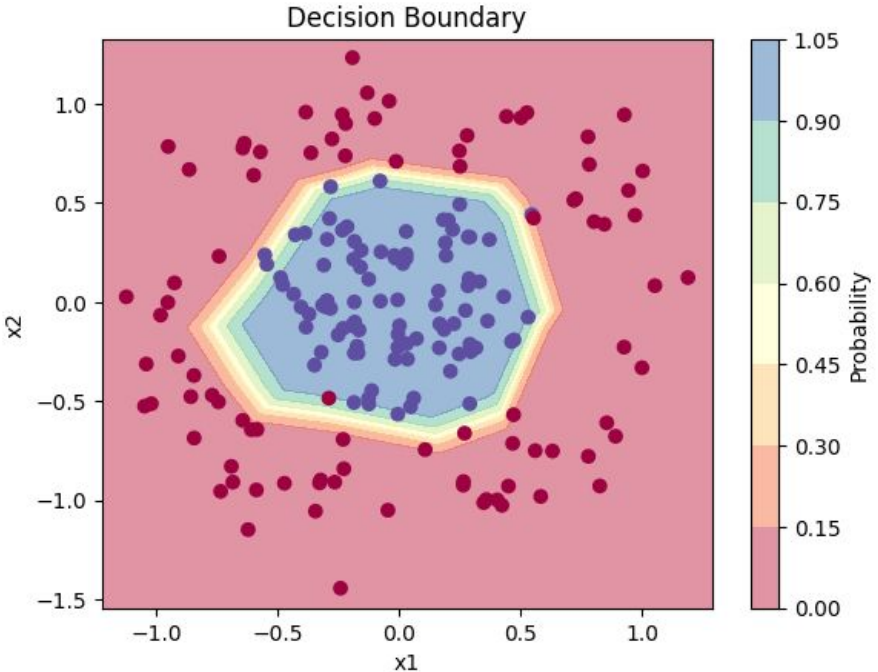
exponential linear unit (ELU)

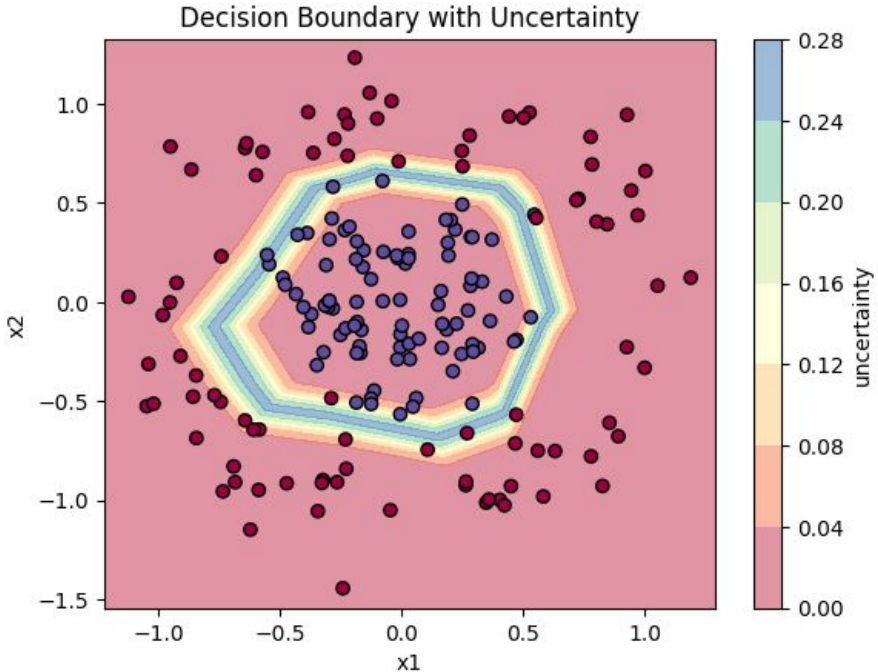


non-saturating

non-zero derivative at +∞ and/or -∞

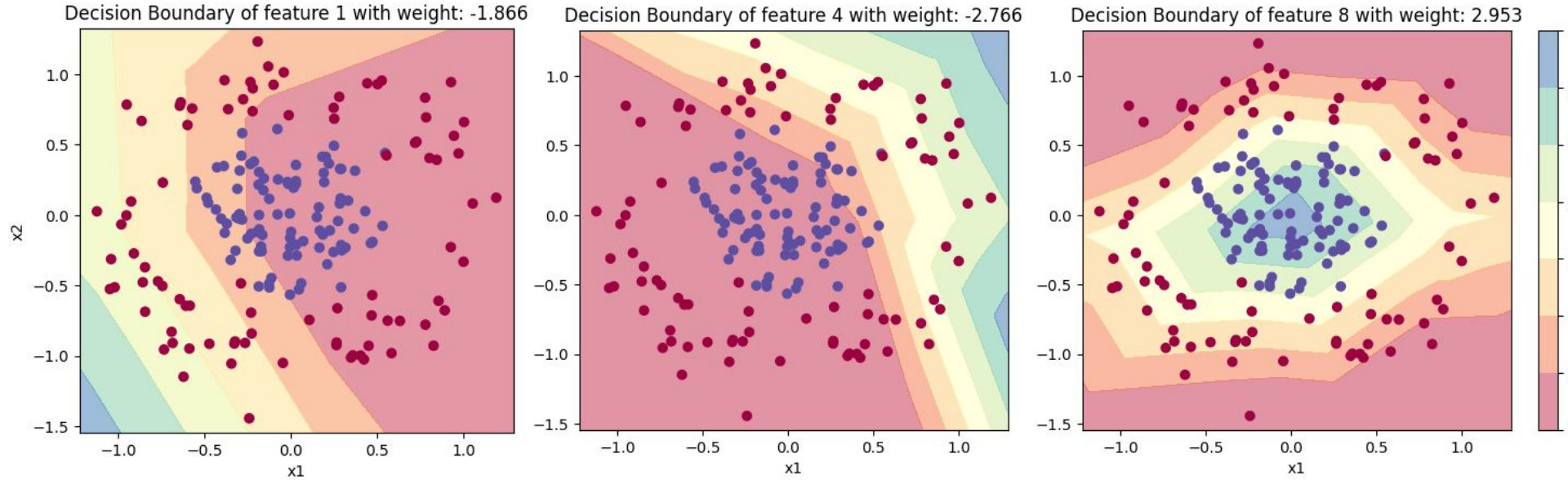
```
model = Network(layers =
  [
    Linear(2, 10, name="input"),
    Activation(ReLU(), name="rely1"),
    Linear(10, 10, name="middle"),
    Activation(ReLU(), name="relu2"),
    Linear(10, 1, name="output"),
    Activation(Sigmoid(), name="sigmoid")
  ]
)
```





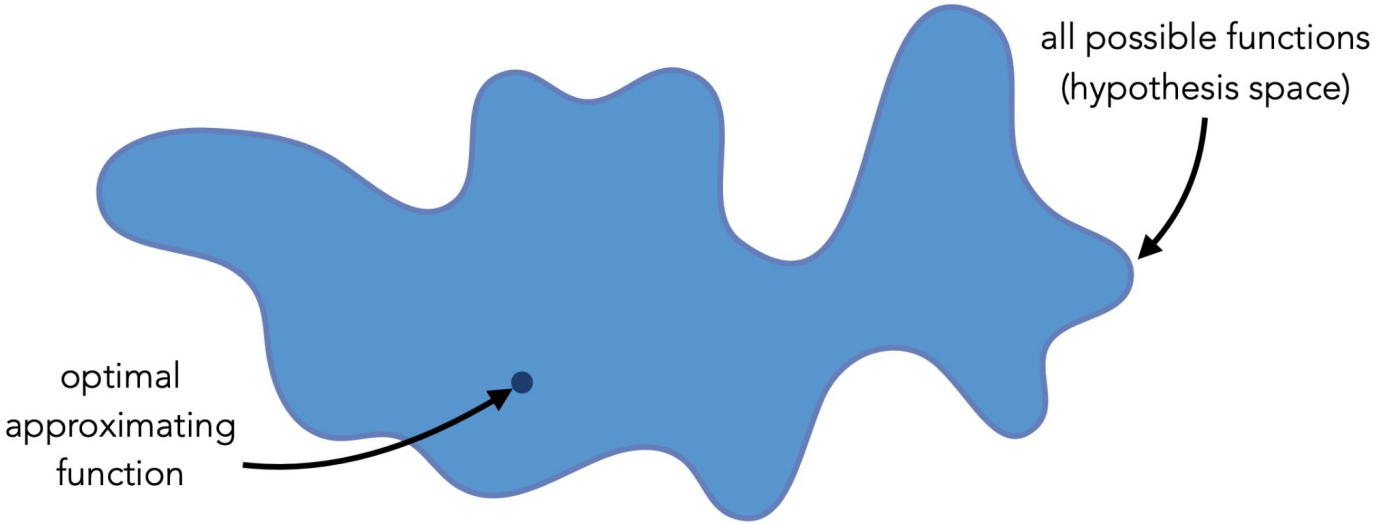
```
model = Network(layers =
  [
    Linear(2, 10, name="input"),
    Activation(ReLU(), name="rely1"),
    Linear(10, 10, name="middle"),
    Activation(ReLU(), name="relu2"),
    Linear(10, 1, name="output"),
    Activation(Sigmoid(), name="sigmoid")
  ]
)
```

What does this layer learn?



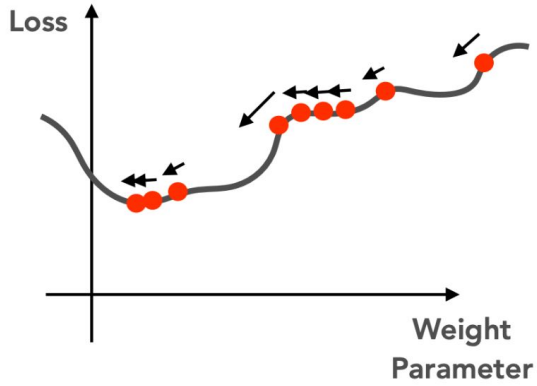
BACKPROPAGATION

neural networks are universal function approximators,
but we still must find an optimal approximating function



we do so by adjusting the weights

learning as *optimization*



to learn the weights, we need the **derivative** of the loss w.r.t. the weight

i.e. "*how should the weight be updated to decrease the loss?*"

$$w = w - \alpha \frac{\partial \mathcal{L}}{\partial w}$$

with multiple weights, we need the **gradient** of the loss w.r.t. the weights

$$\mathbf{w} = \mathbf{w} - \alpha \nabla_{\mathbf{w}} \mathcal{L}$$

backpropagation

a neural network defines a function of composed operations

$$f_L(\mathbf{w}_L, f_{L-1}(\mathbf{w}_{L-1}, \dots f_1(\mathbf{w}_1, \mathbf{x}) \dots))$$

and the loss $\mathcal{L}$ is a function of the network output

→ use chain rule to calculate gradients

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx},$$

and

$$\left.\frac{dz}{dx}\right|_x = \left.\frac{dz}{dy}\right|_{y(x)} \cdot \left.\frac{dy}{dx}\right|_x,$$

backpropagation

a neural network defines a function of composed operations

$$f_L(\mathbf{w}_L, f_{L-1}(\mathbf{w}_{L-1}, \dots f_1(\mathbf{w}_1, \mathbf{x}) \dots))$$

and the loss $\mathcal{L}$ is a function of the network output

⟶ use chain rule to calculate gradients

chain rule example

$$y = w_2 e^{w_1 x}$$

input x output y parameters w_1, w_2

evaluate parameter derivatives: $\frac{\partial y}{\partial w_1}, \frac{\partial y}{\partial w_2}$

define

$$v \equiv e^{w_1 x} \longrightarrow y = w_2 v$$

$$u \equiv w_1 x \longrightarrow v = e^u$$

then $\frac{\partial y}{\partial w_2} = v = e^{w_1 x}$

chain rule

$$\frac{\partial y}{\partial w_1} = \boxed{\frac{\partial y}{\partial v}\frac{\partial v}{\partial u}\frac{\partial u}{\partial w_1}} = w_2 \cdot e^{w_1 x} \cdot x$$

backpropagation

recall

1st layer

$$\mathbf{s}^{(1)} = \mathbf{W}^{(1)\intercal}\mathbf{x}^{(0)}$$
$$\mathbf{x}^{(1)} = \sigma(\mathbf{s}^{(1)})$$

2nd layer

$$\mathbf{s}^{(2)} = \mathbf{W}^{(2)\intercal}\mathbf{x}^{(1)}$$
$$\mathbf{x}^{(2)} = \sigma(\mathbf{s}^{(2)})$$

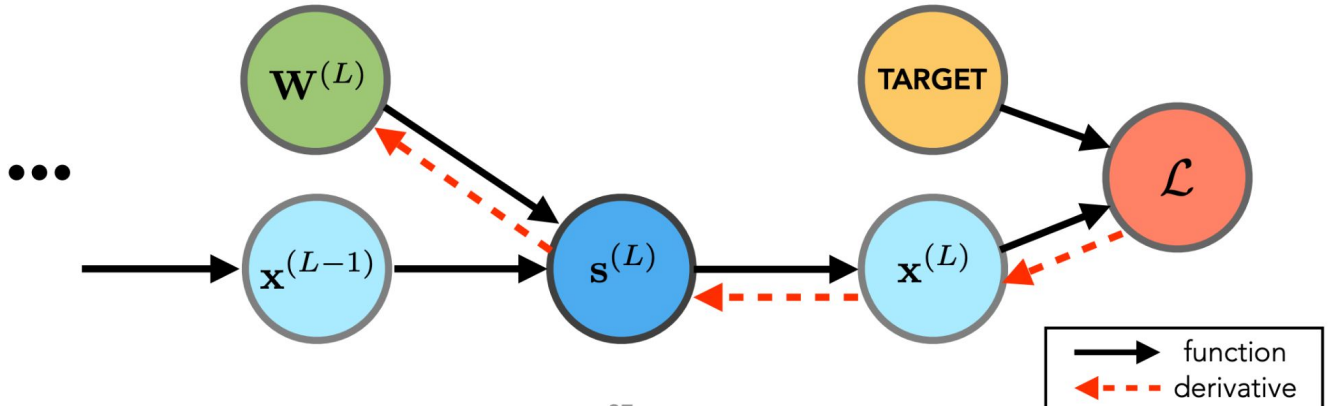
$\bullet\bullet\bullet$

Loss

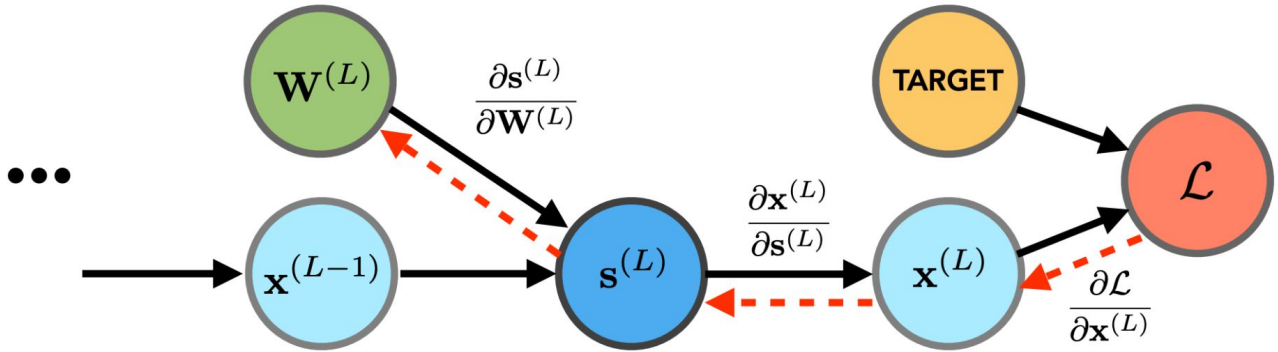
$\mathcal{L}$

calculate $\nabla_{W^{(1)}}\mathcal{L}, \nabla_{W^{(2)}}\mathcal{L}, \ldots$ let's start with the final layer: $\nabla_{W^{(L)}}\mathcal{L}$

to determine the chain rule ordering, we'll draw the dependency graph



backpropagation



$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(L)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(L)}} \frac{\partial \mathbf{x}^{(L)}}{\partial \mathbf{s}^{(L)}} \frac{\partial \mathbf{s}^{(L)}}{\partial \mathbf{W}^{(L)}}$$

depends on the form of the loss

derivative of the non-linearity

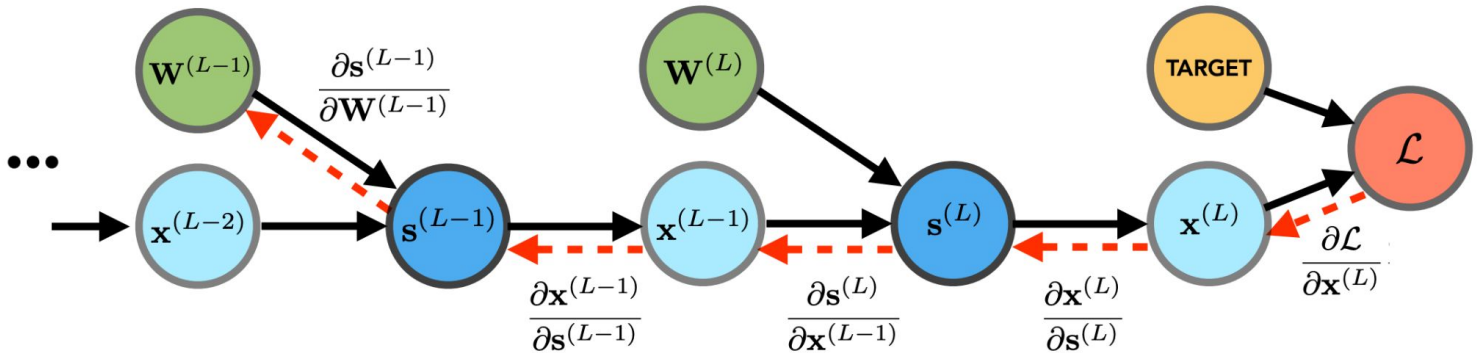
$$\frac{\partial}{\partial \mathbf{W}^{(L)}} (\mathbf{W}^{(L)\intercal} \mathbf{x}^{(L-1)}) = \mathbf{x}^{(L-1)\intercal}$$

note $\nabla_{\mathbf{W}^{(L)}} \mathcal{L} \equiv \frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(L)}}$ is notational convention

backpropagation

now let's go back one more layer…

again we'll draw the dependency graph:

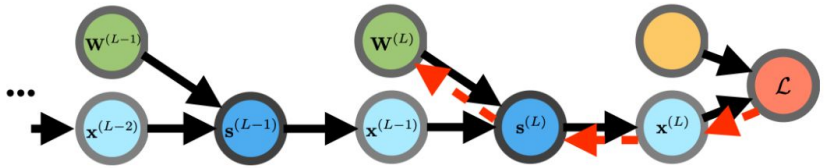


$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(L-1)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(L)}} \frac{\partial \mathbf{x}^{(L)}}{\partial \mathbf{s}^{(L)}} \frac{\partial \mathbf{s}^{(L)}}{\partial \mathbf{x}^{(L-1)}} \frac{\partial \mathbf{x}^{(L-1)}}{\partial \mathbf{s}^{(L-1)}} \frac{\partial \mathbf{s}^{(L-1)}}{\partial \mathbf{W}^{(L-1)}}$$

backpropagation

BACKPROPAGATION ALGORITHM

calculate $\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(L)}}$



backpropagation

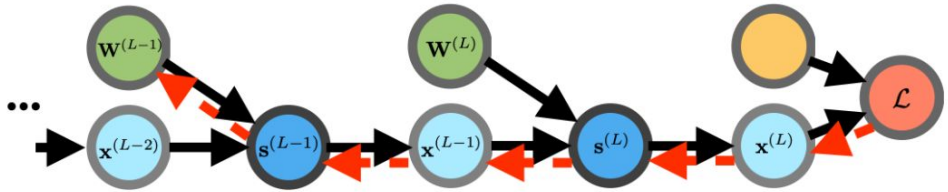
BACKPROPAGATION ALGORITHM

calculate $\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(L)}}$

store $\frac{\partial \mathcal{L}}{\partial \mathbf{s}^{(L)}}$

for $\ell = [L-1, \ldots, 1]$

use $\frac{\partial \mathcal{L}}{\partial \mathbf{s}^{(\ell+1)}}$ to calculate $\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(\ell)}}$



backpropagation

BACKPROPAGATION ALGORITHM

calculate $\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(L)}}$

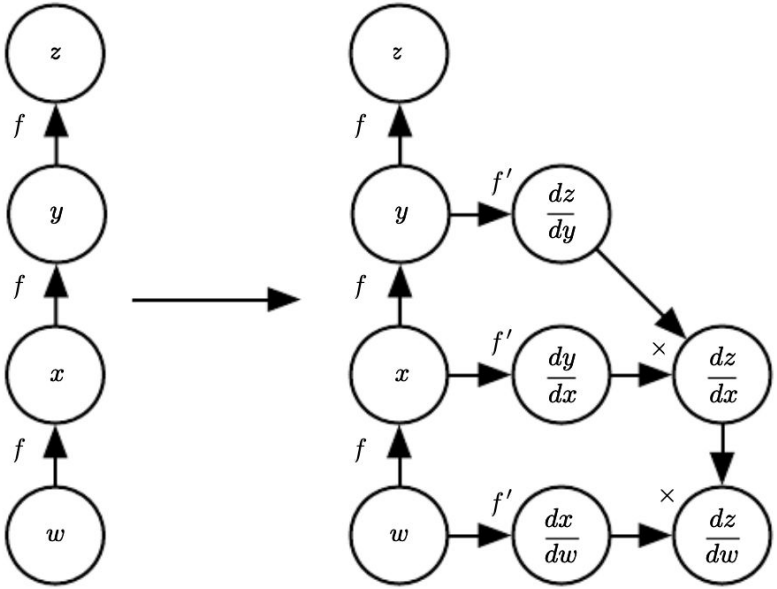
store $\frac{\partial \mathcal{L}}{\partial \mathbf{s}^{(L)}}$

for $\ell = [L-1, \ldots, 1]$

use $\frac{\partial \mathcal{L}}{\partial \mathbf{s}^{(\ell+1)}}$ to calculate $\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(\ell)}}$

store $\frac{\partial \mathcal{L}}{\partial \mathbf{s}^{(\ell)}}$

return $\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(1)}}, \ldots, \frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(L)}}$

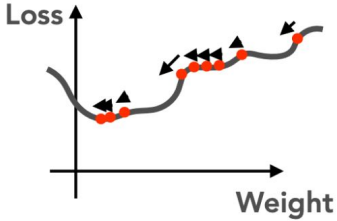


$$
\begin{aligned}
&\frac{\partial z}{\partial w} \\
=&\frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w} \\
=&f'(y)f'(x)f'(w) \\
=&f'(f(f(w)))f'(f(w))f'(w).
\end{aligned}
$$

Figure 6.10: An example of the symbol-to-symbol approach to computing derivatives. In this approach, the back-propagation algorithm does not need to ever access any actual specific numeric values. Instead, it adds nodes to a computational graph describing how to compute these derivatives. A generic graph evaluation engine can later compute the derivatives for any specific numeric values. *(Left)* In this example, we begin with a graph representing $z = f(f(f(w)))$. *(Right)* We run the back-propagation algorithm, instructing it to construct the graph for the expression corresponding to $\frac{dz}{dw}$. In this example, we do not explain how the back-propagation algorithm works. The purpose is only to illustrate what the desired result is: a computational graph with a symbolic description of the derivative.

recapitulation

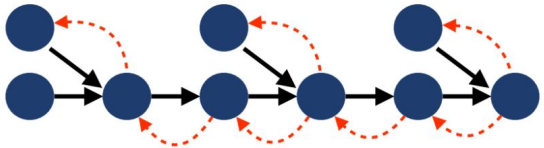
update weights using gradient of loss



backpropagation calculates the loss gradients w.r.t. internal weights

→ *"credit assignment"* via chain rule

gradient is *propagated backward* through the network



most deep learning software libraries automatically calculate gradients

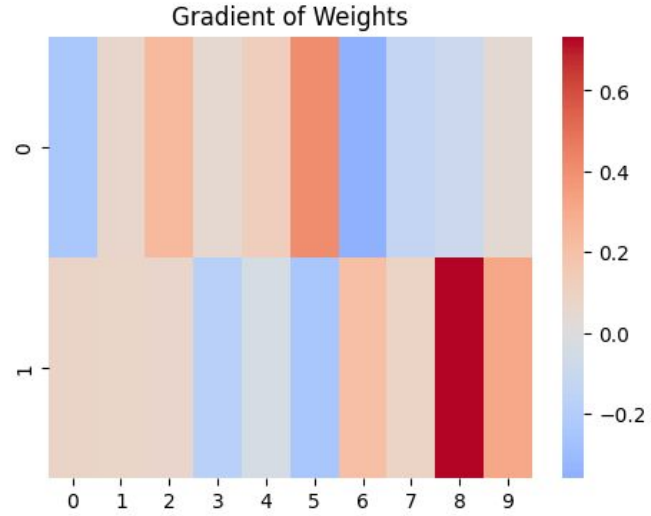
→ *"automatic differentiation"* or *"auto-diff"*

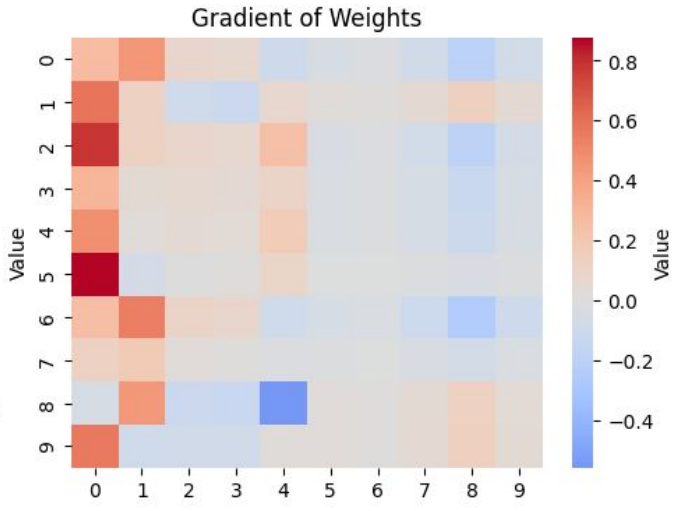
→ can calculate gradients for any differentiable operation

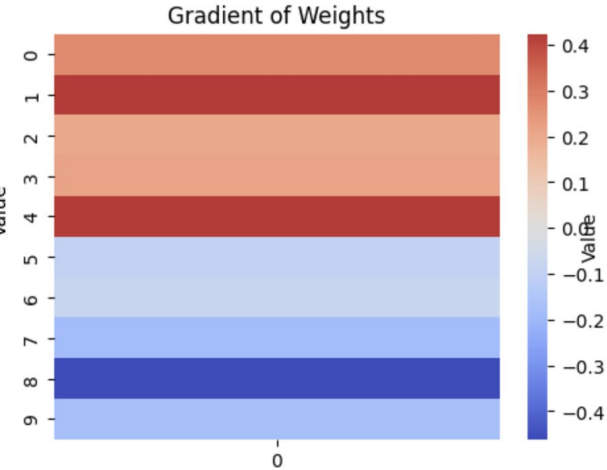
```
model = Network(layers =
  [
    Linear(2, 10, name="input"),
    Activation(ReLU(), name="rely1"),
    Linear(10, 10, name="middle"),
    Activation(ReLU(), name="relu2"),
    Linear(10, 1, name="output"),
    Activation(Sigmoid(), name="sigmoid")
  ]
)
```

```
class Network():
    def backward(self, loss_grad):
        for layer in reversed(self.layers):
            loss_grad = layer.backward(loss_grad)
```

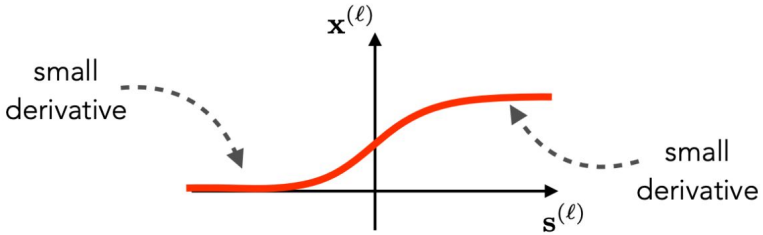
```
class Linear():
    def backward(self, error):
        input_error = np.dot(error, self.W.T)
        self.gradient_w = np.dot(self.input.T, error)
        return input_error
```



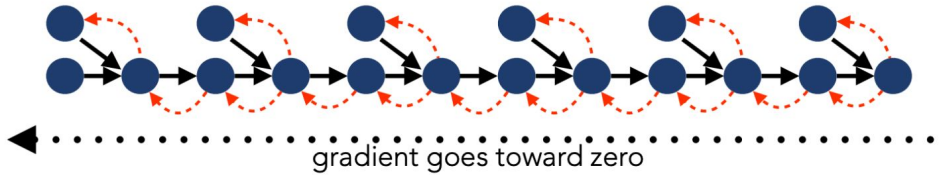




vanishing gradients



saturating non-linearities have *small* derivatives almost everywhere

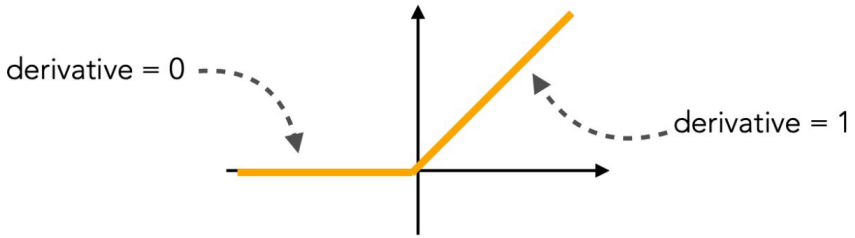


in backprop, the product of many small terms (i.e. $\frac{\partial \mathbf{x}^{(\ell)}}{\partial \mathbf{s}^{(\ell)}}$) goes to zero

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(\ell)}} = \cdots \frac{\partial \mathbf{x}^{(L)}}{\partial \mathbf{s}^{(L)}} \cdots \frac{\partial \mathbf{x}^{(L-1)}}{\partial \mathbf{s}^{(L-1)}} \cdots \frac{\partial \mathbf{x}^{(\ell+1)}}{\partial \mathbf{s}^{(\ell+1)}} \cdots \frac{\partial \mathbf{x}^{(\ell)}}{\partial \mathbf{s}^{(\ell)}} \frac{\partial \mathbf{s}^{(\ell)}}{\partial \mathbf{W}^{(\ell)}}$$

difficult to train very deep networks with saturating non-linearities

ReLU



$$\mathrm{ReLU}(x) = \max(x, 0)$$

in the positive region, ReLU does not saturate,
preventing gradients from vanishing in deep networks

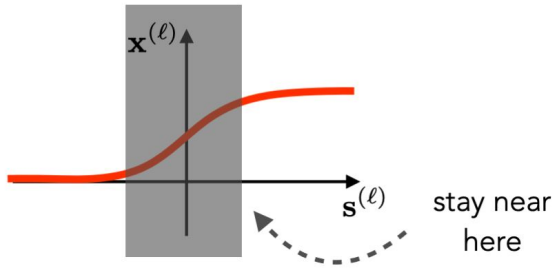
in the negative region, ReLU saturates at zero,
resulting in 'dead units' where the gradient is zero

How to avoid value all be negative?

normalization

can we prevent the gradients from saturating non-linearities from becoming too small?

→ keep the inputs within the dynamic range of the non-linearity



we can ***normalize*** the activations before applying the non-linearity

$$\mathbf{s} \leftarrow \frac{\mathbf{s} - \text{shift}}{\text{scale}}$$

batch normalization

batch norm. normalizes each layer's activations according to the statistics of the batch

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

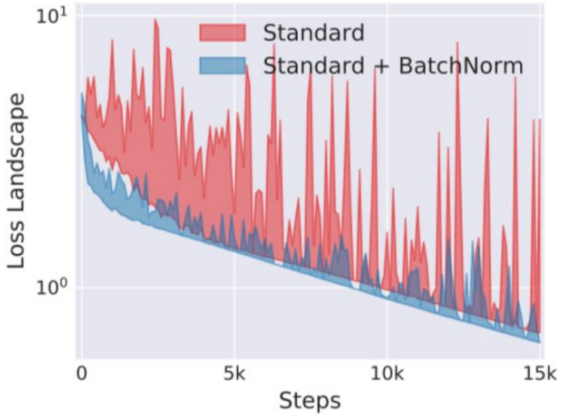
$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

keeps internal activations in similar range, speeding up training

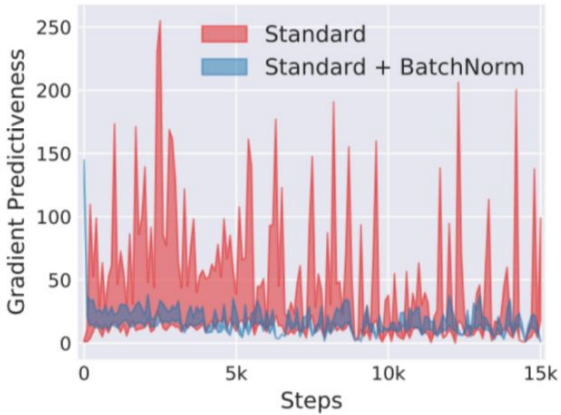
adds stochasticity, improves generalization

Batch Normalization, Szegedy & Ioffe, 2015

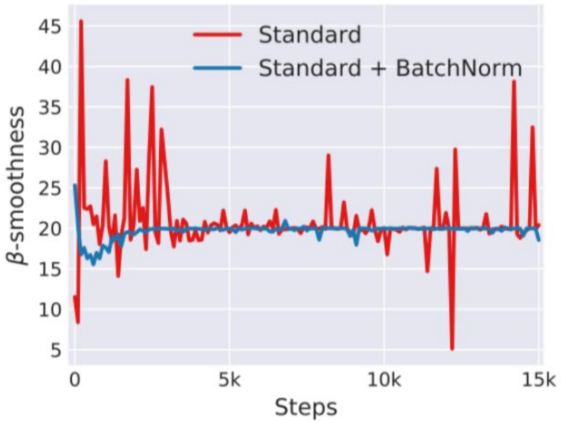
BatchNorm can stabilize Training and and smooths the loss landscape



(a) loss landscape



(b) gradient predictiveness



(c) "effective" β-smoothness

More Regularization and Normalization & why they work will be covered **next lecture!**