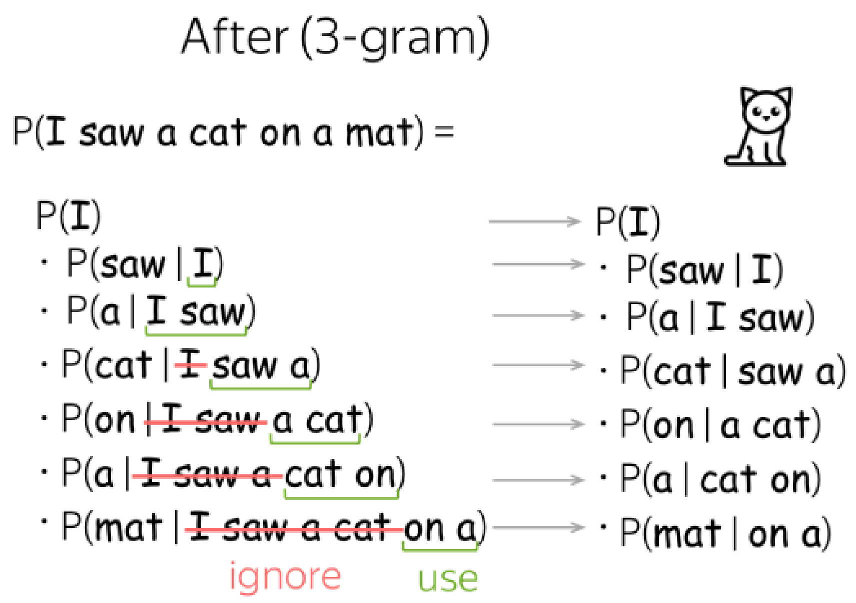
FOLT Lecture 3, NNs for NLP

Intro

- A language model (LM) measures the probability of a text sequence using the chain rule

$$P(w_1 w_2 \dots w_T) = \prod_{i}^{T} P(w_i | w_{<i})$$



$$\underbrace{P(w_T | w_1 w_2 \dots w_{T-1})}_{\text{all previous words}} \approx P(w_T | \underbrace{w_{T-n+1} \dots w_{T-1}}_{n-1 \text{ previous words}})$$

$$P(w_T | w_{<T}) \approx \frac{N(w_{T-n+1} \dots w_{T-1} w_T)}{N(w_{T-n+1} \dots w_{T-1})} = \frac{N(n\text{-gram})}{N((n-1)\text{-gram})}$$

We usually measure the performance of LMs with the intrinsic evaluation method: Perplexity
exp: How hard is recognizing (30,000) names at random? Perplexity = 30,000
We previously talked about N-gram language models some of their Pros

- Easy to understand
- Cheap (with modern hardware)
- Fine in some applications and when training data is scarce
  And some of their cons :
- Assume the vocabulary is known
- Sparsity problems
- Zero count
- **Cannot use long context (large n(-gram) leads to higher sparsity)**
  this last point is the challenge to when we have to model **Semantic similarity**

Semantic word Similarity

Two words with $N(w_1) >> N(w_2)$ ($N(cat) >> N(kitten)$)
Can $P(cat|\text{saw a})$ give us information about $P(kitten|\text{saw a})$?
Not in N-gram models
But in Neural LMs Yes

PART1: Neural Network Overview

State-of-the-Art NLP Methods are :

- Deep learning approaches(• (often) free from linguistic features • very large neural models • pre-training over large raw text • works well with unstructured data • deliver high-quality results)
- Neural network architectures(• LSTM (Long Short-Term Memory) • GRU (Gated Recurrent Unit) • CNN (Convolutional Neural Networks) • Transformers)
  Neural networks: originally inspired by modelling biological neural systems

Neural networks and Deep Learning Pros and Cons

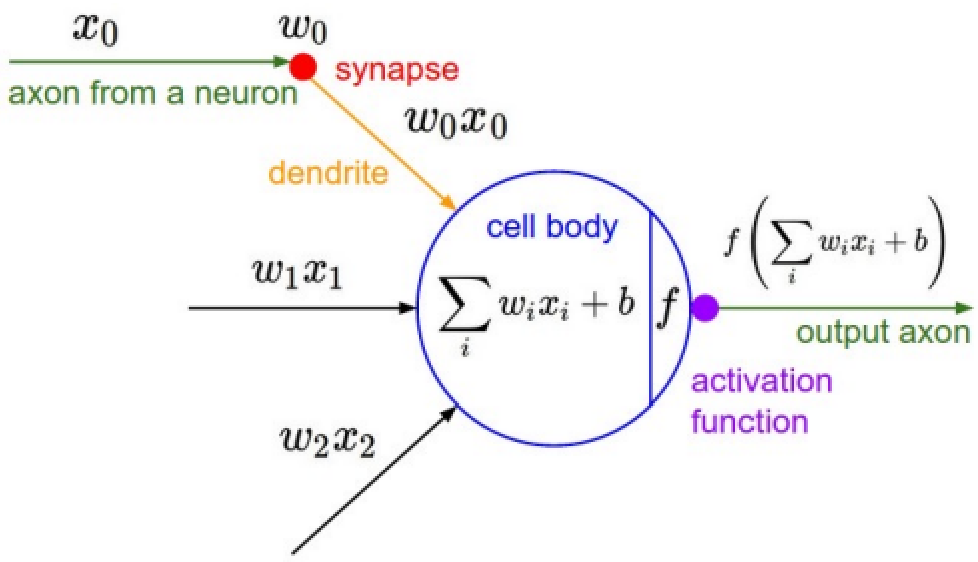
Pros

● State-of-the-art performance (current best performance) ● Architecture easily adaptable to multiple problems, e.g., Transformers ● Reduce need for feature engineering, e.g., counting n-grams

Cons

- Require a lot of cleaned training data and computational resources (E.g., Reinforcement learning from human feedback (RLHF) à ChatGPT, GPT-4)
- Long training time with a lot of Hardware and multiple rounds of human feedback
- Interpretability is a challenge (DL is a Blackbox)

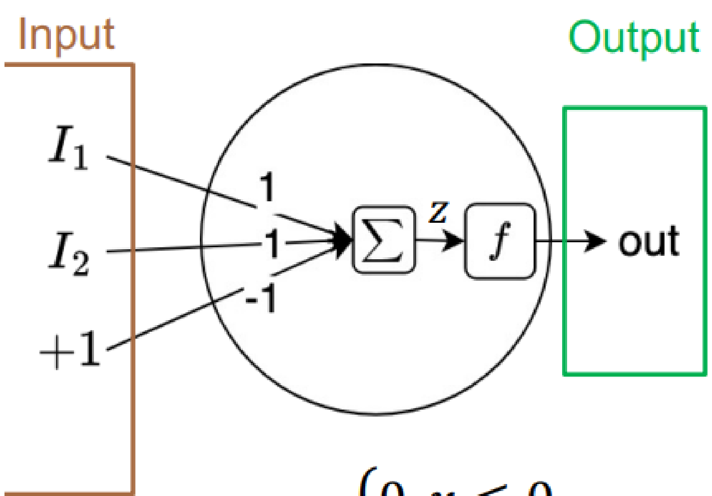
PART2: Perceptrons



1) Let's try to compute AND

AND		
I_1	I_2	out
0	0	0
0	1	0
1	0	0
1	1	1

Input Output

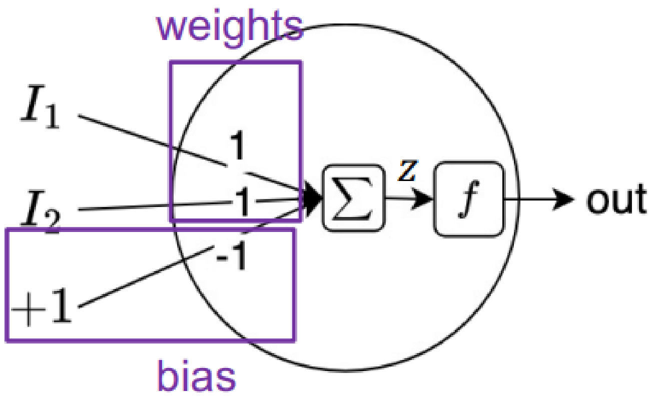


$$f(x) = \begin{cases} 0, x \le 0 \\ 1, x > 0 \end{cases}$$

AND		
I_1	I_2	out
0	0	0
0	1	0
1	0	0
1	1	1

Predict output using a perceptron

- We feed input into the perceptron
- Try to predict output



$$f(x) = \begin{cases} 0, x \leq 0 \\ 1, x > 0 \end{cases}$$

AND		
I_1	I_2	out
0	0	0
0	1	0
1	0	0
1	1	1

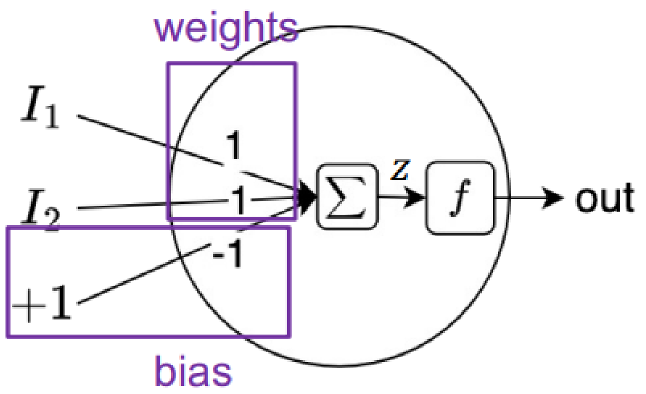
Predict output using a perceptron:

Parameters θ of a perceptron:

weights and biases

- We multiply input by weights
- Take sum and add the bias

$$z = I_1 \times 1 + I_2 \times 1 + 1 \times -1$$
$$= 0 \times 1 + 0 \times 1 + 1 \times -1$$
$$= -1$$



$$f(z) = \begin{cases} 0, z \leq 0 \\ 1, z > 0 \end{cases}$$

activation function

AND		
I_1	I_2	out
0	0	0
0	1	0
1	0	0
1	1	1

Predict output using a perceptron:

Parameters θ of a perceptron:

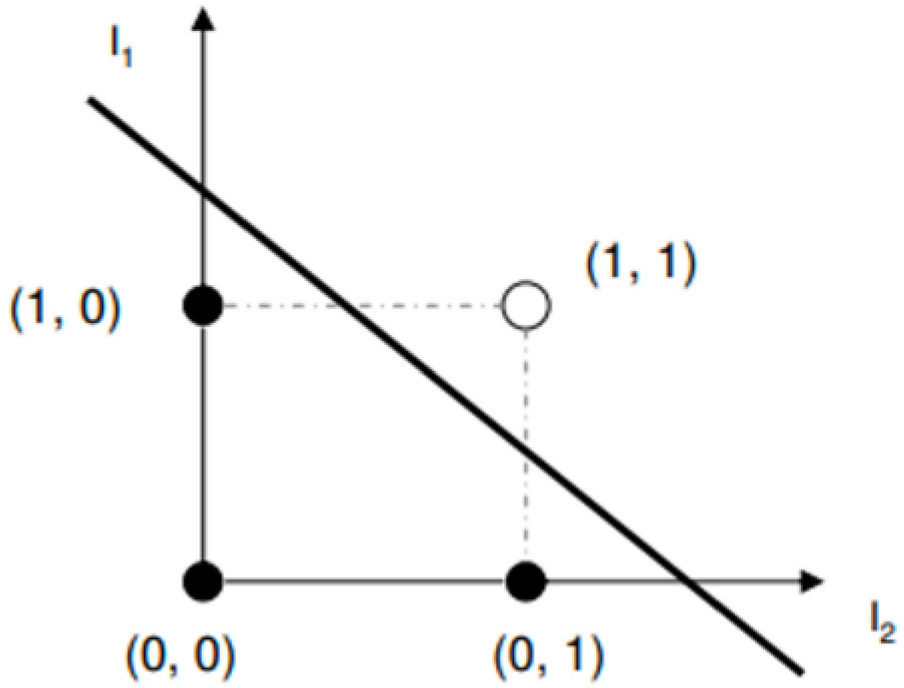
weights and biases

- We get $z = -1$
- Then we feed z into the activation function $f(z)$
- Because $z = -1 < 0, f(z) = 0 =$ out

AND		
I_1	I_2	out
0	0	0
0	1	0
1	0	0
1	1	1

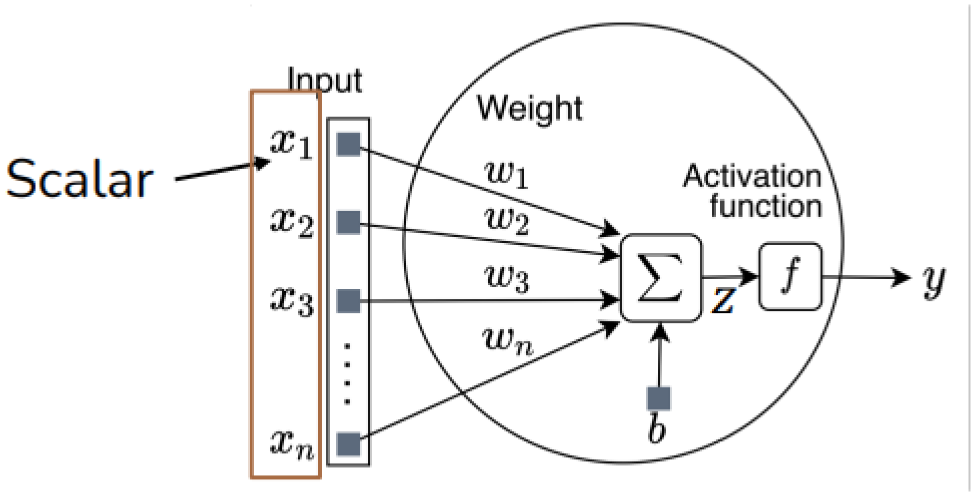
$$f(z) = f(I_1 \times 1 + I_2 \times 1 + 1 \times -1)$$
$$= f(0 \times 1 + 1 \times 1 + 1 \times -1)$$
$$= f(0) = 0 = \text{out}$$

rest of input output combinations computed similarly and we obtain:



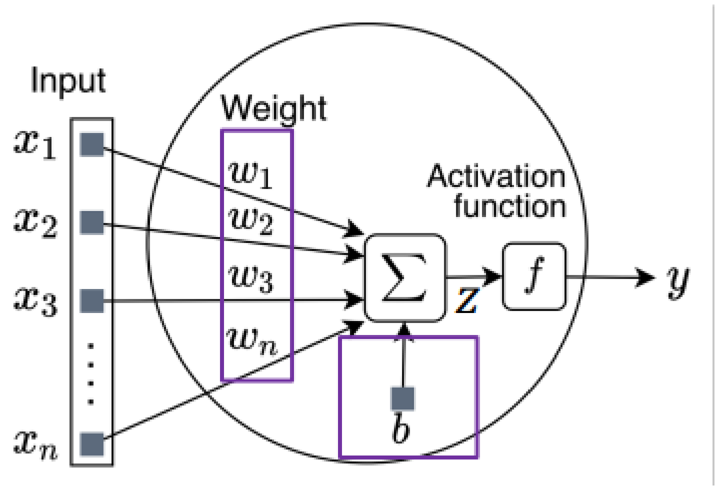
2) Perceptrons

o Classic form



• Input x: usually a real value vector

Each element is a scalar = real number • E.g., input is a vector with n dimensions



• Parameters θ :

-Weights $w = w_1, w_2 . . . , w_n$(please don't confuse with words input x is our "words") Each weight associated with each input indicate each input's “importance” • E.g., n scalar inputs, n weight values or weight vector of n

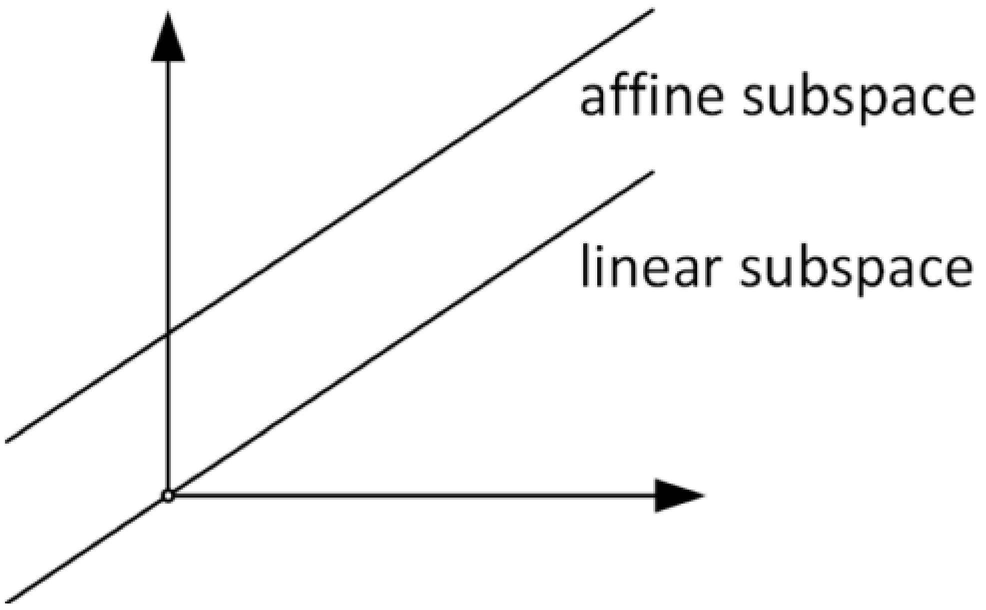
-Bias b : a scalar

- Intermediate output $z = (\sum_i^n w_i x_i) + b$:
  - **Weighted** summation of the inputs
    - E.g., $z = I_1 \times 1 + I_2 \times 1$
  - With an additional **bias**
    - E.g., $z = I_1 \times 1 + I_2 \times 1 + 1 \times -1$

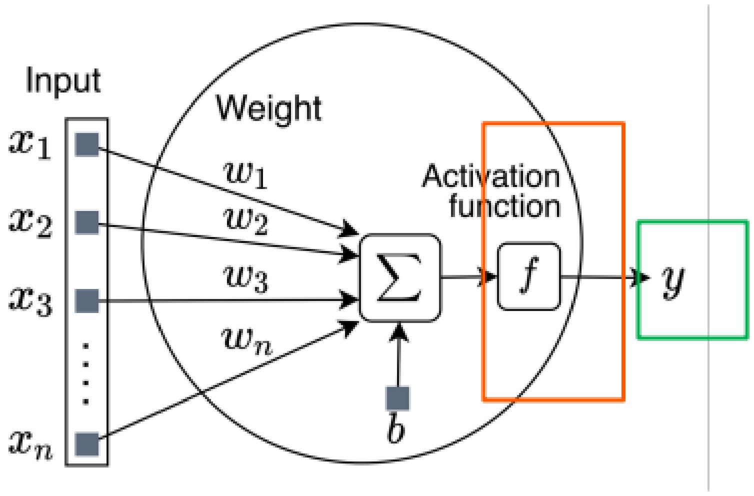
Why Using Bias?

o A neuron with weights only must fix the origin to zero.

o The bias allows for modelling the set of **affine** functions, which is a superset of **linear** functions.



- Then, pass z through an activation function f
  - E.g., $f(z) = \begin{cases} 0, z \leq 0 \\ 1, z > 0 \end{cases}$
  - Controls the value range of the output y

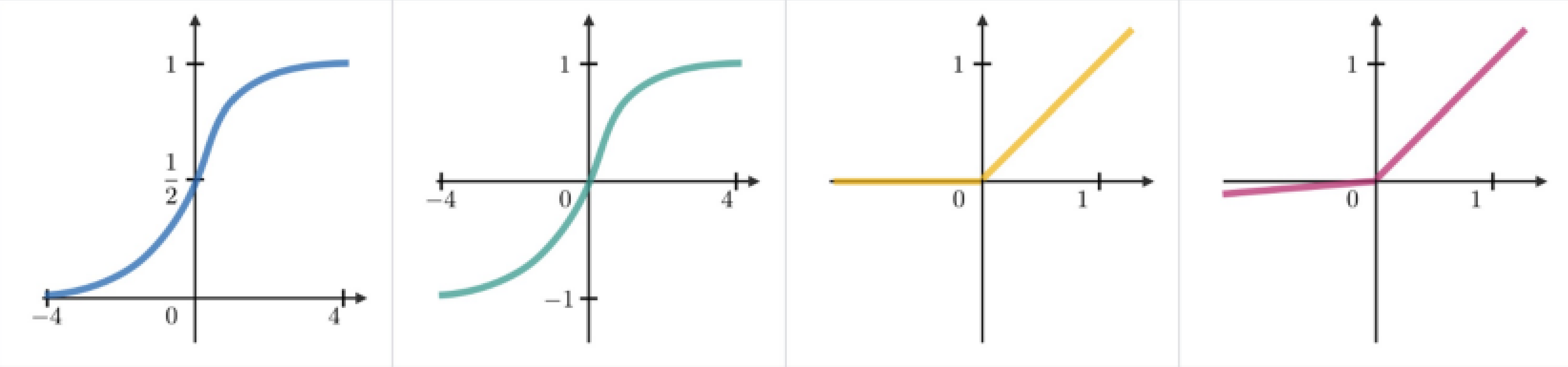


3)Activation function $f(.)$

o Identity (linear function): $f(z) = z$

o Activation functions are used introduce non-linear complexities to a model

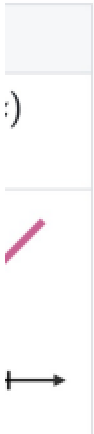
Sigmoid	Tanh	ReLU	Leaky ReLU
$g(z) = \dfrac{1}{1+e^{-z}}$	$g(z) = \dfrac{e^z - e^{-z}}{e^z + e^{-z}}$	$g(z) = \max(0, z)$	$g(z) = \max(\epsilon z, z)$ with $\epsilon \ll 1$



Example

Compute "AND"

$$\boxed{f(z) = \begin{cases} 0, z \le 0 \\ 1, z > 0 \end{cases}}$$



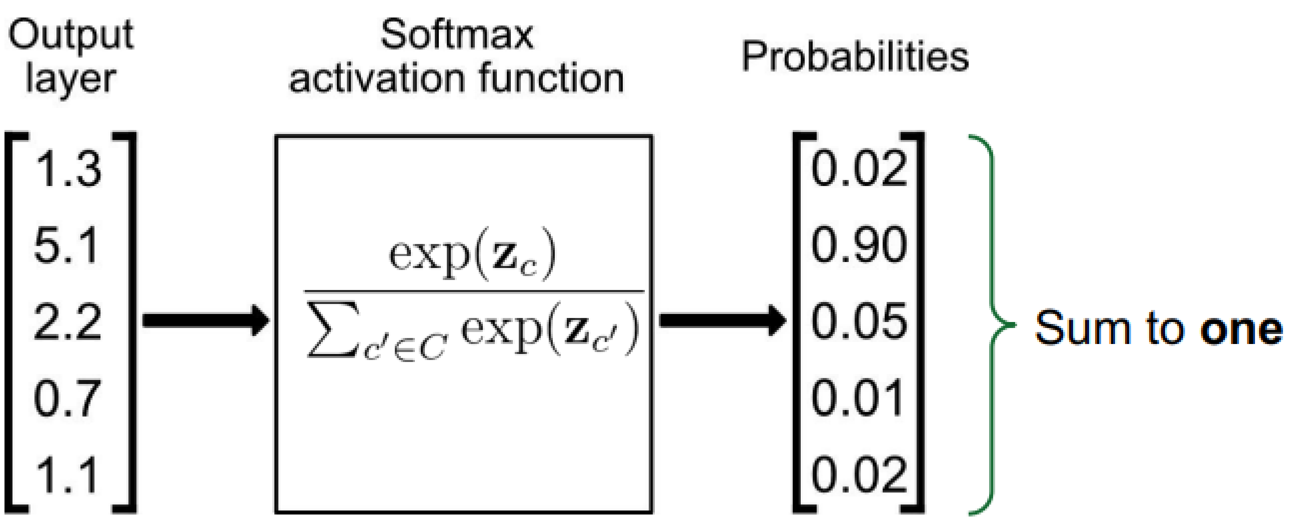
$$\boxed{\begin{aligned} f(z) &= \begin{cases} 0, z \le 0 \\ z, z > 0 \end{cases} \\ &= \max(0, z) \\ &= \text{ReLU} \end{aligned}}$$

1a)Softmax

Take a vector and compute a probability distribution out of the vector

○ Each score resides in [0, 1]

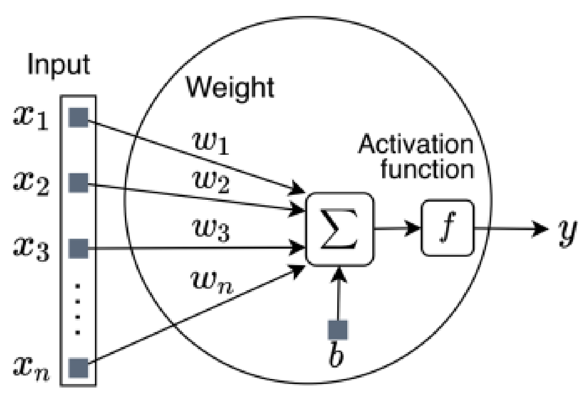
● Act on a layer, not individual node



Vector form of a perceptron

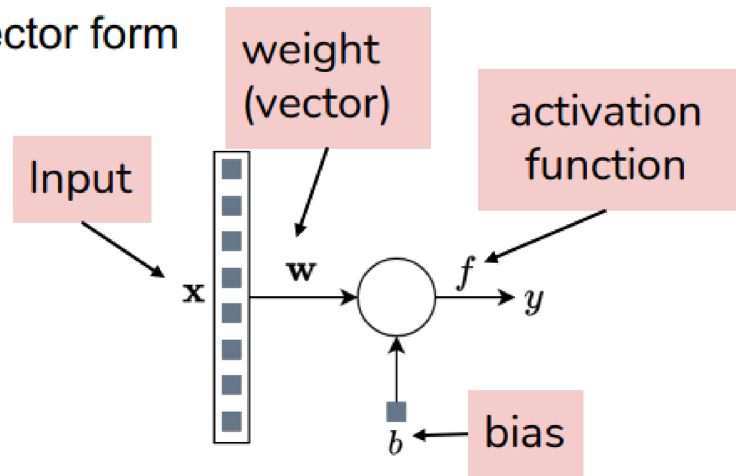
**

- Classic form



$$y = f\left(\left(\sum_{i}^{n} w_i x_i\right) + b\right)$$

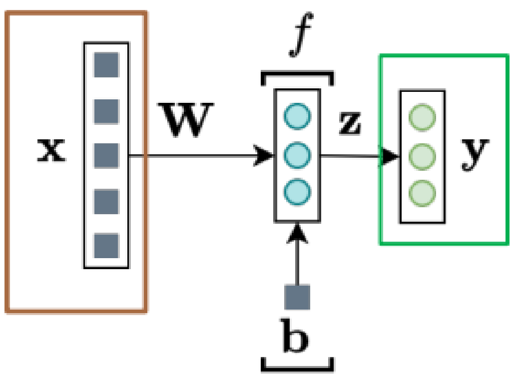
- Vector form



$$y = f(\mathbf{w}^{\mathsf{T}}\mathbf{x} + b)$$

Input: $\mathbf{x} \in \mathbb{R}^n$, Output: $y \in \mathbb{R}$
Parameters θ: Weights: $\mathbf{w} \in \mathbb{R}^n$, bias: $b \in \mathbb{R}$

2) Perceptron : multiple Outputs

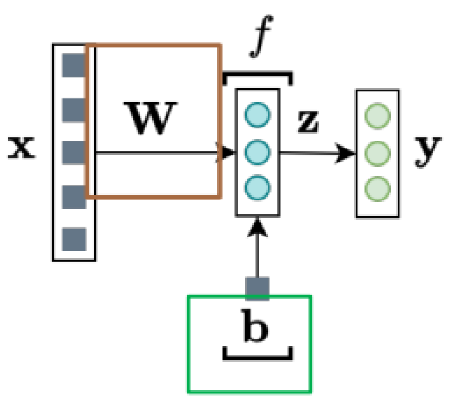


One-layer perceptron

or one-layer neural network

$$\boxed{\mathbf{y} =} \qquad \boxed{\mathbf{x}}$$

- Input: $\mathbf{x} \in \mathbb{R}^n$ → vector
- Output: $\mathbf{y} \in \mathbb{R}^m$ → vector



One-layer perceptron

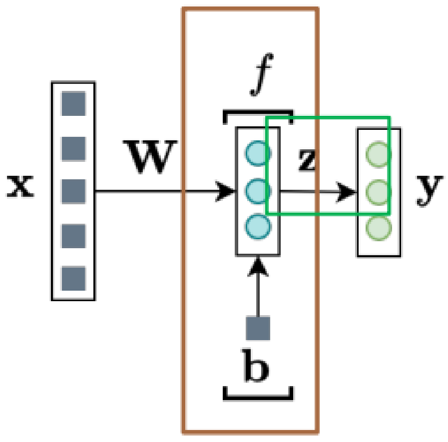
or one-layer neural network

$$\mathbf{y} = \boxed{\mathbf{x}\mathbf{W}} + \boxed{\mathbf{b}}$$

- Input: $\mathbf{x} \in \mathbb{R}^n$ → vector
- Output: $\mathbf{y} \in \mathbb{R}^m$ → vector

Parameters θ:

- Weights: $\boxed{\mathbf{W} \in \mathbb{R}^{n \times m}}$ → matrix
- bias: $\boxed{\mathbf{b} \in \mathbb{R}^m}$ → vector



One-layer perceptron

or one-layer neural network

$$\mathbf{y} = f(\mathbf{z}) = f(\mathbf{x}\mathbf{W} + \mathbf{b})$$

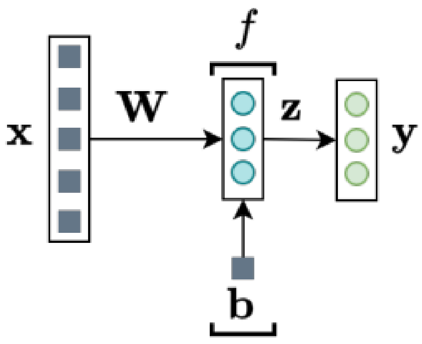
- Input: $\mathbf{x} \in \mathbb{R}^n$ → vector
- Output: $\mathbf{y} \in \mathbb{R}^m$ → vector

Parameters θ:

- Weights: $\mathbf{W} \in \mathbb{R}^{n \times m}$ → matrix
- bias: $\mathbf{b} \in \mathbb{R}^m$ → vector

Activation function $f(\cdot)$: element-wise

- Apply to each element of its input
- E.g., each element in the intermediate output $\mathbf{z} \in \mathbb{R}^m$



One-layer perceptron

or one-layer neural network

$$\mathbf{y} = f(\mathbf{z}) = f(\mathbf{x}\mathbf{W} + \mathbf{b})$$

$n = 3$ $m = 2$

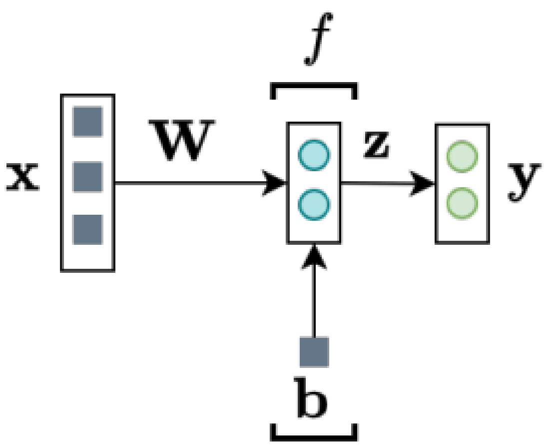
Example: $\boldsymbol{y} = \boldsymbol{x}\boldsymbol{W} + \boldsymbol{b}$, $d_{in} = 3$, $d_{out} = 2$

$$\begin{pmatrix} \boldsymbol{x}_{[1]} & \boldsymbol{x}_{[2]} & \boldsymbol{x}_{[3]} \end{pmatrix} \begin{pmatrix} \boldsymbol{W}_{[1,1]} & \boldsymbol{W}_{[1,2]} \\ \boldsymbol{W}_{[2,1]} & \boldsymbol{W}_{[2,2]} \\ \boldsymbol{W}_{[3,1]} & \boldsymbol{W}_{[3,2]} \end{pmatrix} + \begin{pmatrix} \boldsymbol{b}_{[1]} & \boldsymbol{b}_{[2]} \end{pmatrix} = \begin{pmatrix} \boldsymbol{y}_{[1]} & \boldsymbol{y}_{[2]} \end{pmatrix}$$

Ivan H., Deep learning for NLP, TUDarmstadt

Example:**

$$\mathbf{y} = f(\mathbf{z}) = f(\mathbf{x}\mathbf{W} + \mathbf{b})$$



- Input: $\mathbf{x}$=[1, 0.5, -2] $\in \mathbb{R}^3$ → vector with 3 dimensions
- Output: $\mathbf{y} \in \mathbb{R}^2$ → vector

Parameters θ:

- Weights: $\mathbf{W} = \begin{bmatrix} 1 & 0 \\ -1 & -1 \\ 10 & -9 \end{bmatrix} \in \mathbb{R}^{3 \times 2}$ → matrix
- bias: $\mathbf{b} = [0, 0] \in \mathbb{R}^2$ → vector zeros → ignore

Activation function $f(\cdot)$: element-wise

- Assume $f(\cdot)$ is identity → ignore

$$\mathbf{y} = \mathbf{x}\mathbf{W}$$

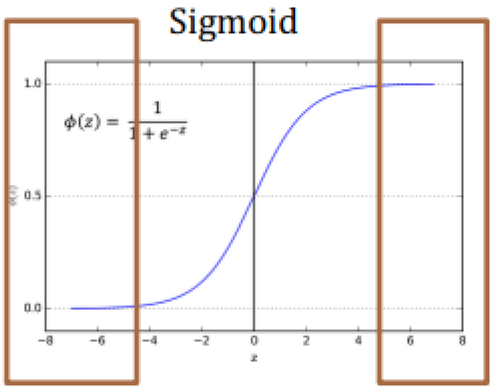
$$= [1, 0.5, -2]\begin{bmatrix} 1 & 0 \\ -1 & -1 \\ 10 & -9 \end{bmatrix}$$

=[1×1 + 0.5×-1 + -2×10, 1×0 + 0.5×-1 + -2×-9]

=[-19.5,17.5]

If activation function is $\text{Sigmoid} = \frac{\mathbf{1}}{\mathbf{1+e^{-x}}}$

$$\mathbf{y} = f(\mathbf{x}\mathbf{W})$$



$$= f([1, 0.5, -2]\begin{bmatrix} 1 & 0 \\ -1 & -1 \\ 10 & -9 \end{bmatrix})$$

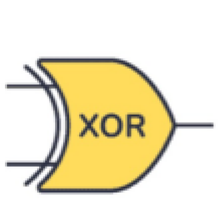
=f([1×1 + 0.5×-1 + -2×10, 1×0 + 0.5×-1 + -2×-9])

=f([-19.5,17.5])=Sigmoid([$\boxed{-19.5}$ $\boxed{17.5}$])=[3.39e-09, 0.99] ≈ [0, 1]

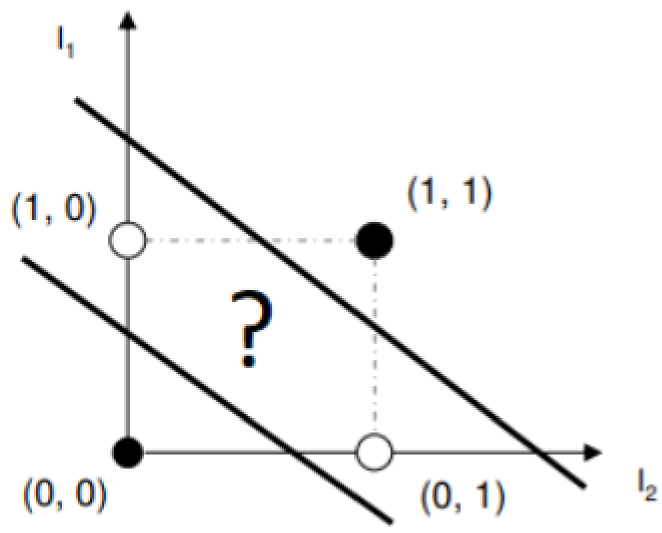
4) Multi-layer Perceptron

Non-linearly Separable Problems

- Single-layer perceptron cannot solve non-linearly separable problems
  - Such as the most famous simple example is the boolean XOR operator



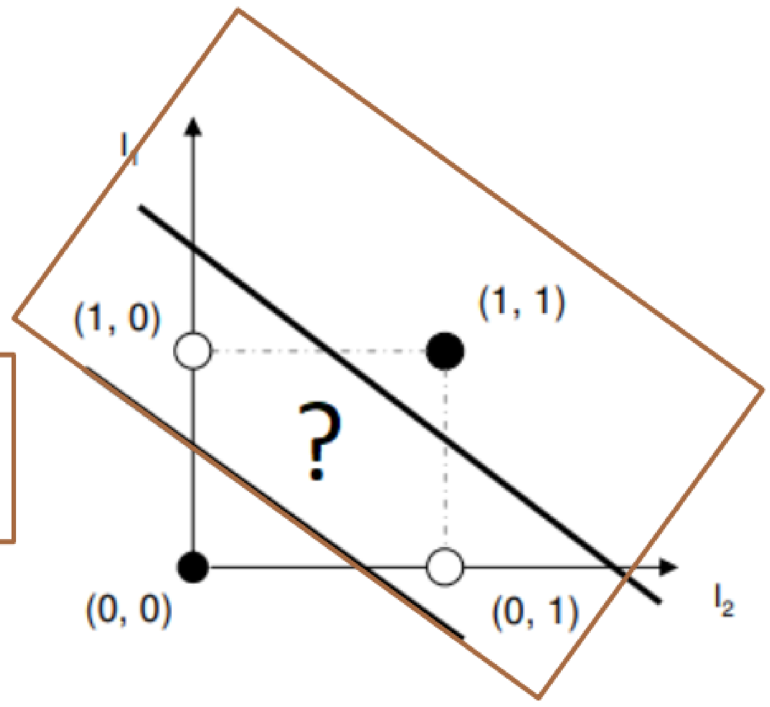
XOR		
I_1	I_2	out
0	0	0
0	1	1
1	0	1
1	1	0



1)XOR from OR and NOT AND

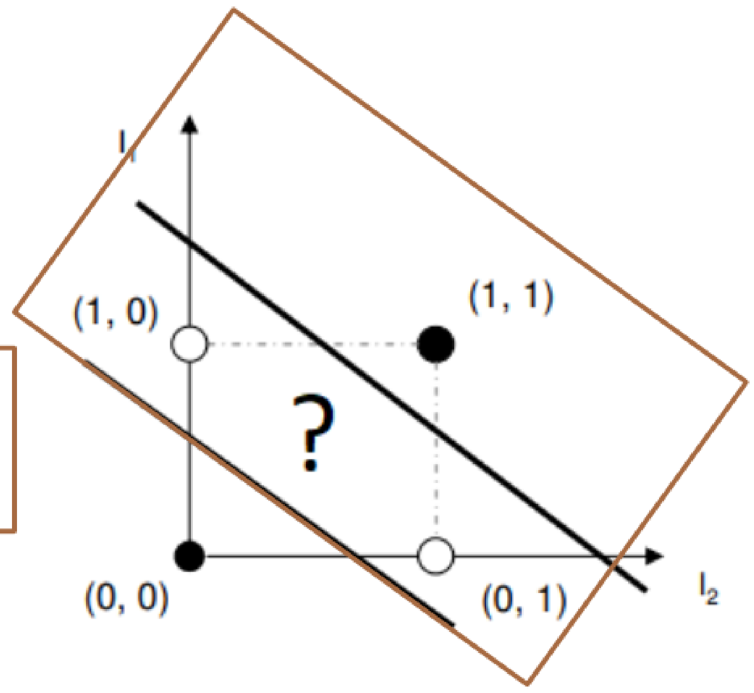
Compute OR

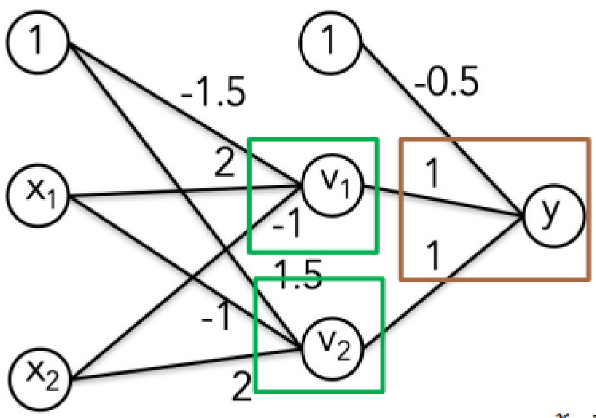
XOR		
I_1	I_2	out
0	0	0
0	1	1
1	0	1
1	1	0



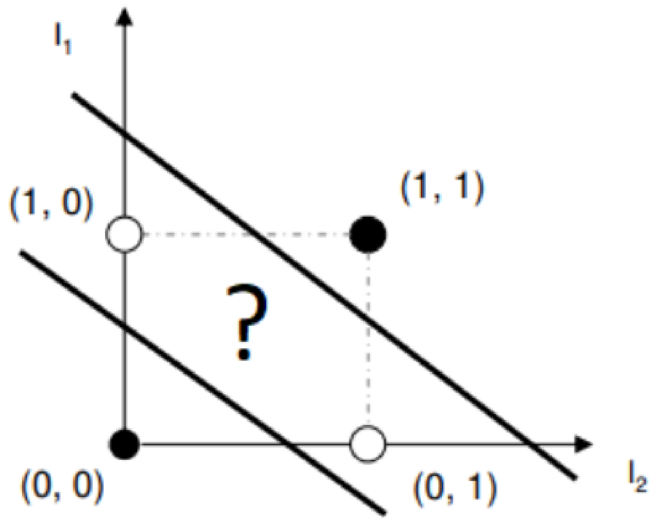
Compute OR

XOR		
I_1	I_2	out
0	0	0
0	1	1
1	0	1
1	1	0





XOR		
I_1	I_2	out
0	0	0
0	1	1
1	0	1
1	1	0



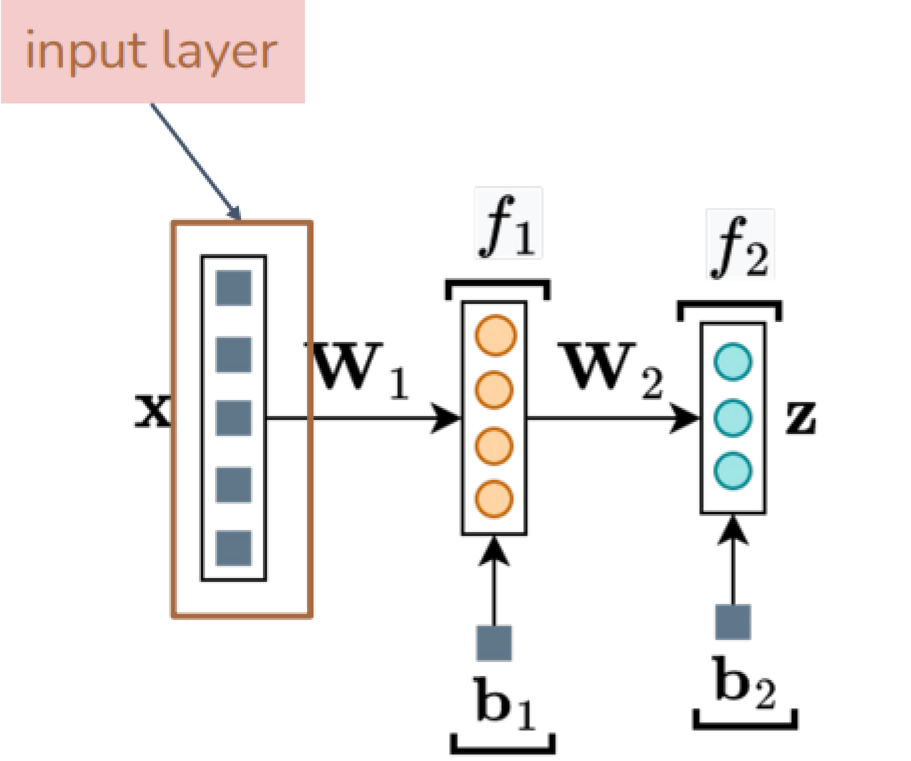
$x_1 = I_1,$
$x_2 = I_2$

$$v_1 = f(-1.5 + 2x_1 - x_2)$$
$$v_2 = f(-1.5 + 2x_2 - x_1)$$
$$f(z) = \begin{cases} 0, z \le 0 \\ 1, z > 0 \end{cases}$$
$$y = -0.5 + v_1 + v_2$$

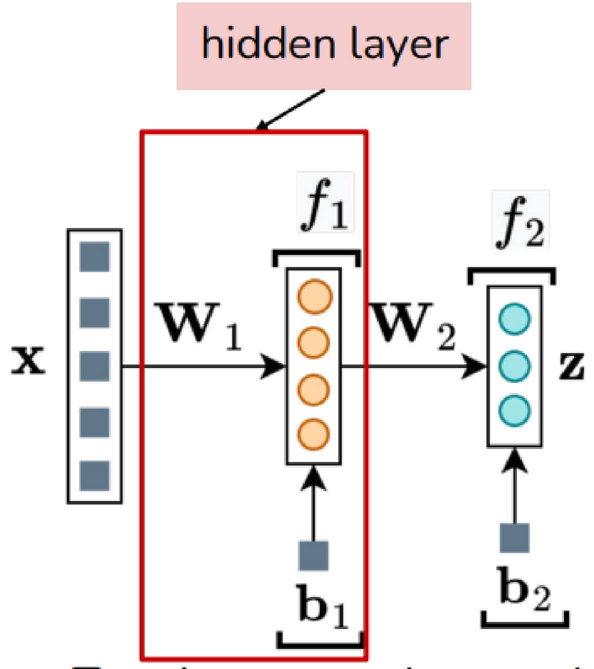
2)Multi-layer Perceptron

input layer

$$\mathbf{z}_0 = \boxed{\mathbf{x}}$$



Two-layer neural network
or
One hidden layer neural network

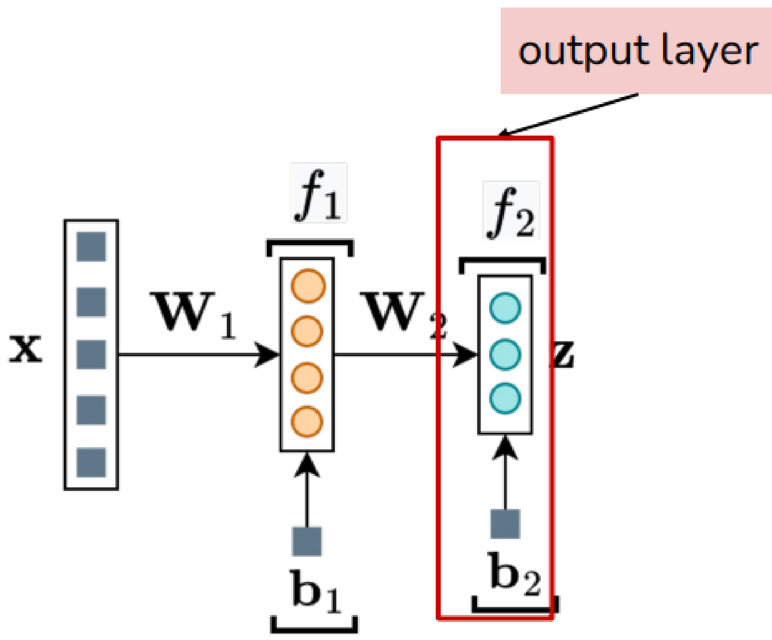


Two-layer neural network
or
One hidden layer neural network

$$\mathbf{z}_0 = \mathbf{x}$$

$$\boxed{\mathbf{z}_1 = \boxed{f_1}(\mathbf{z}_0\mathbf{W}_1 + \mathbf{b}_1)}$$

Activation function

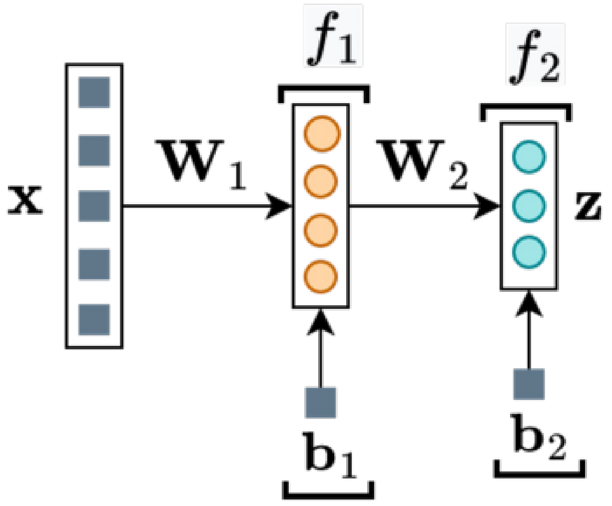


Two-layer neural network
or
One hidden layer neural network

$$\mathbf{z}_0 = \mathbf{x}$$

$$\mathbf{z}_1 = f_1(\mathbf{z}_0\mathbf{W}_1 + \mathbf{b}_1)$$

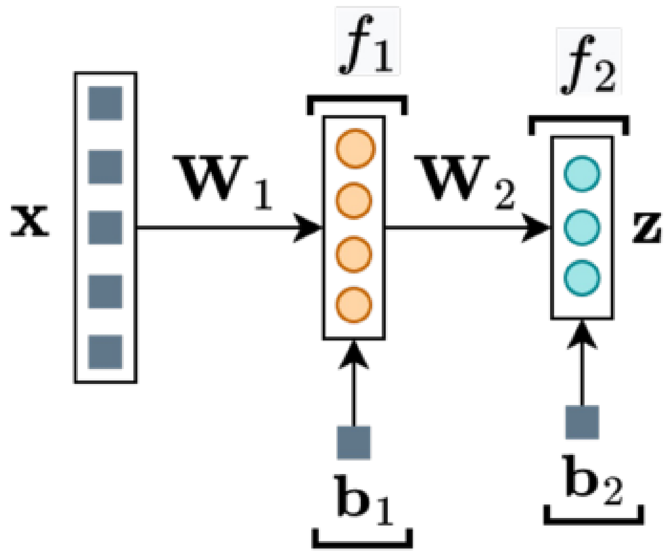
$$\boxed{\mathbf{z}_2 = f_2(\mathbf{z}_1\mathbf{W}_2 + \mathbf{b}_2)}$$



Two-layer neural network
or
One hidden layer neural network

$$
\begin{aligned}
\mathbf{z}_0 &= \mathbf{x} \\
\mathbf{z}_1 &= f_1(\mathbf{z}_0\mathbf{W}_1 + \mathbf{b}_1) \\
\mathbf{z}_2 &= f_2(\mathbf{z}_1\mathbf{W}_2 + \mathbf{b}_2)
\end{aligned}
$$

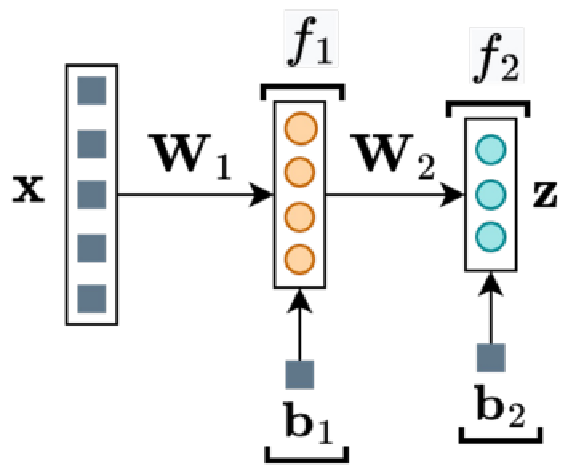
output of each layer



Two-layer neural network
or
One hidden layer neural network

$$
\begin{aligned}
\mathbf{z}_0 &= \mathbf{x} \\
\mathbf{z}_1 &= f_1(\mathbf{z}_0\mathbf{W}_1 + \mathbf{b}_1) \\
\mathbf{z}_2 &= f_2(\mathbf{z}_1\mathbf{W}_2 + \mathbf{b}_2)
\end{aligned}
$$

input of each layer
=
output of previous layer

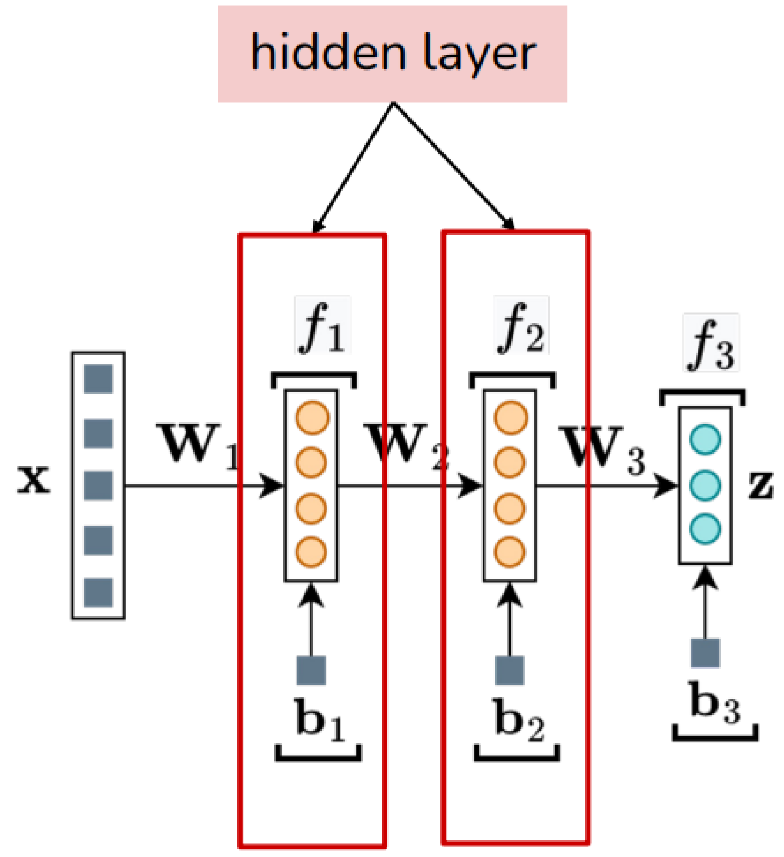


Two-layer neural network
or
One hidden layer neural network

$$\mathbf{z}_0 = \mathbf{x}$$
$$\mathbf{z}_1 = f_1(\mathbf{z}_0\mathbf{W}_1 + \mathbf{b}_1)$$
$$\mathbf{z}_2 = f_2(\mathbf{z}_1\mathbf{W}_2 + \mathbf{b}_2)$$

input of each layer
=
output of previous
layer

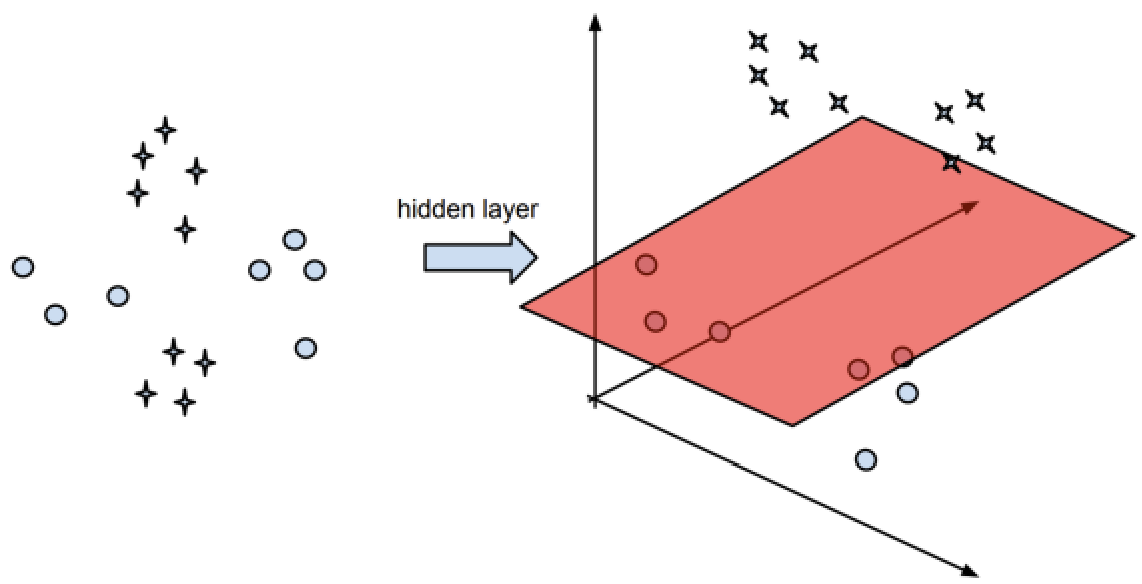
$$\mathbf{z}_2 = f_2(f_1(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2)$$



Three-layer neural network
or
Two hidden layer neural network

Hidden Layers

o Decides the expressiveness of the whole neuron network
o Hidden layer can project the data onto another vector space in which they are now linearly separable.



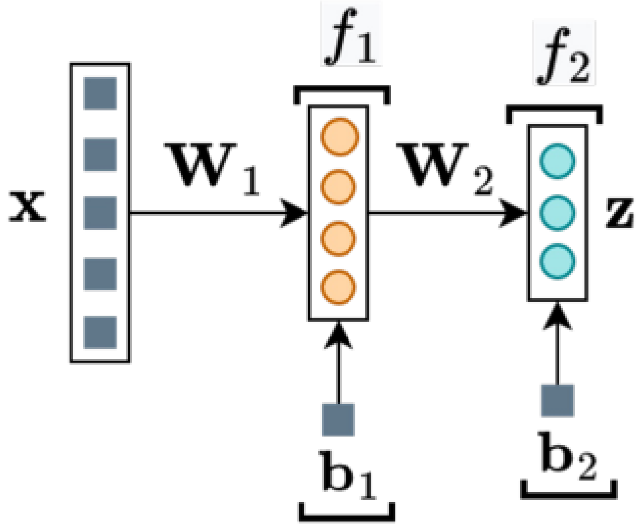
Do We Need More Hidden Layers?

● In theory, two-layer neural networks (one hidden layer) can approximate any functions if we increase the number of hidden units and cover all examples .

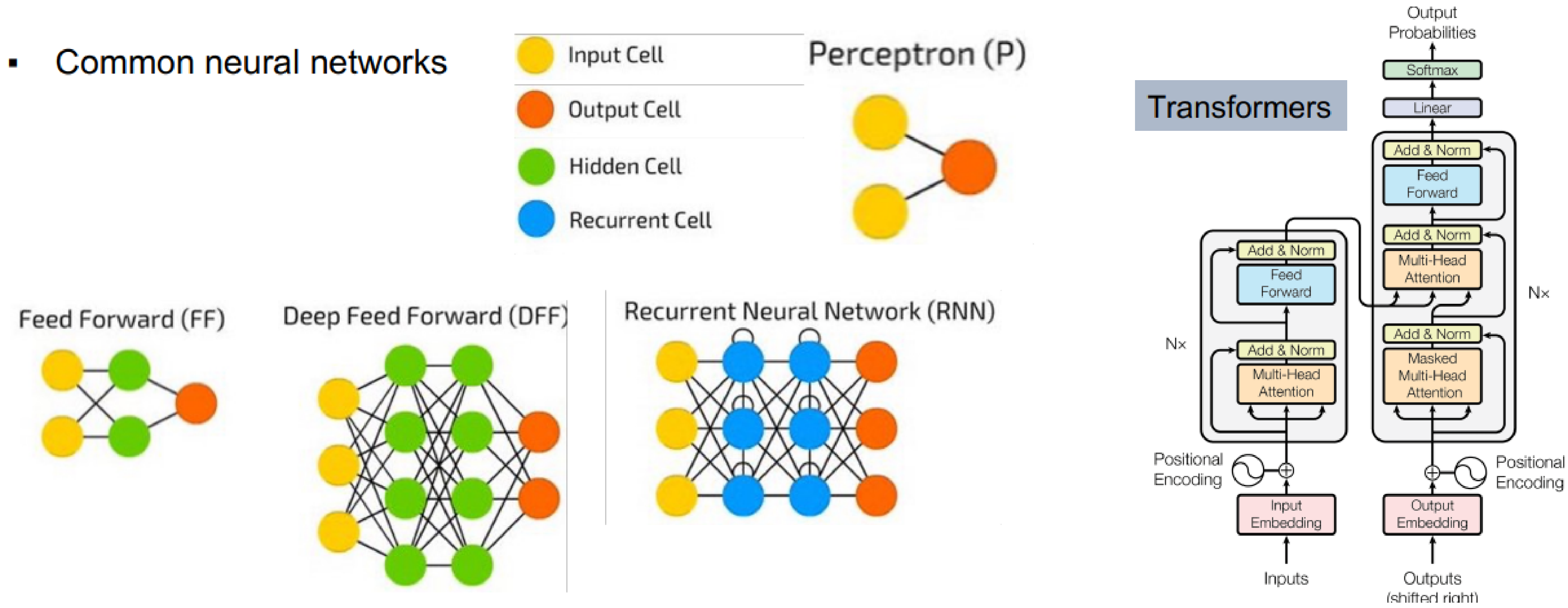
● In practice, it is impossible to include all examples, the network can fit any training data but not generalise to new data points by reasoning from the observed data. So, we usually add more layers to increase the complexity

● To prevent linearity → approximate complex functions

$$
\begin{aligned}
\mathbf{z} &= f_2(f_1(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2) \\
&= ((\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2) \\
&= (\mathbf{W}_1\mathbf{x})\mathbf{W}_2 + (\mathbf{b}_1\mathbf{W}_2 + \mathbf{b}_2) \\
&= \mathbf{W}'\mathbf{x} + \mathbf{b}'
\end{aligned}
$$



Artificial Neural Networks

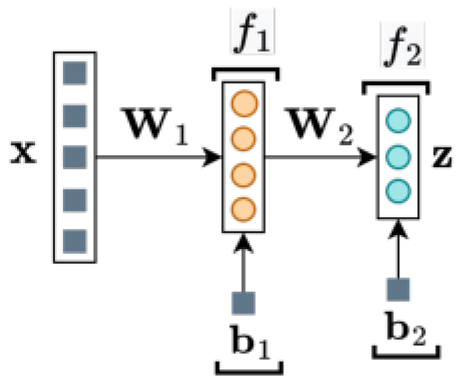


PART3:Training a Neural Network

o How does the model get the parameters?
Training on the training data.
o Use stochastic gradient descent algorithms (more details in DL courses) to optimize
-Maximum likelihood estimate (MLE) (Compute AND, maximize the likelihood of output 1 given input (1, 1) , Compute XOR, maximize the likelihood of output 1 given input (0, 1) or (1, 0))
- I.e., minimize the error between the prediction of the model and the “correct” output

1) Optimize Parameters

○ Step 0. Initialize model
○ Step 1. Take a batch of training data
○ Step 2. Perform forward pass (inference) to obtain the prediction
o Forward pass (inference)
o Feed the input into neural networks
o Compute from input to output
o Get the prediction / output



$$\mathbf{z}_2 = f_2(f_1(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2)$$

○ Step 3. Backward pass (training) from MLE
o Backward pass (training)
o Compute error of the predictions

o Compute changes to parameters in each layer, backward from output to input
o Add the changes to parameters → update

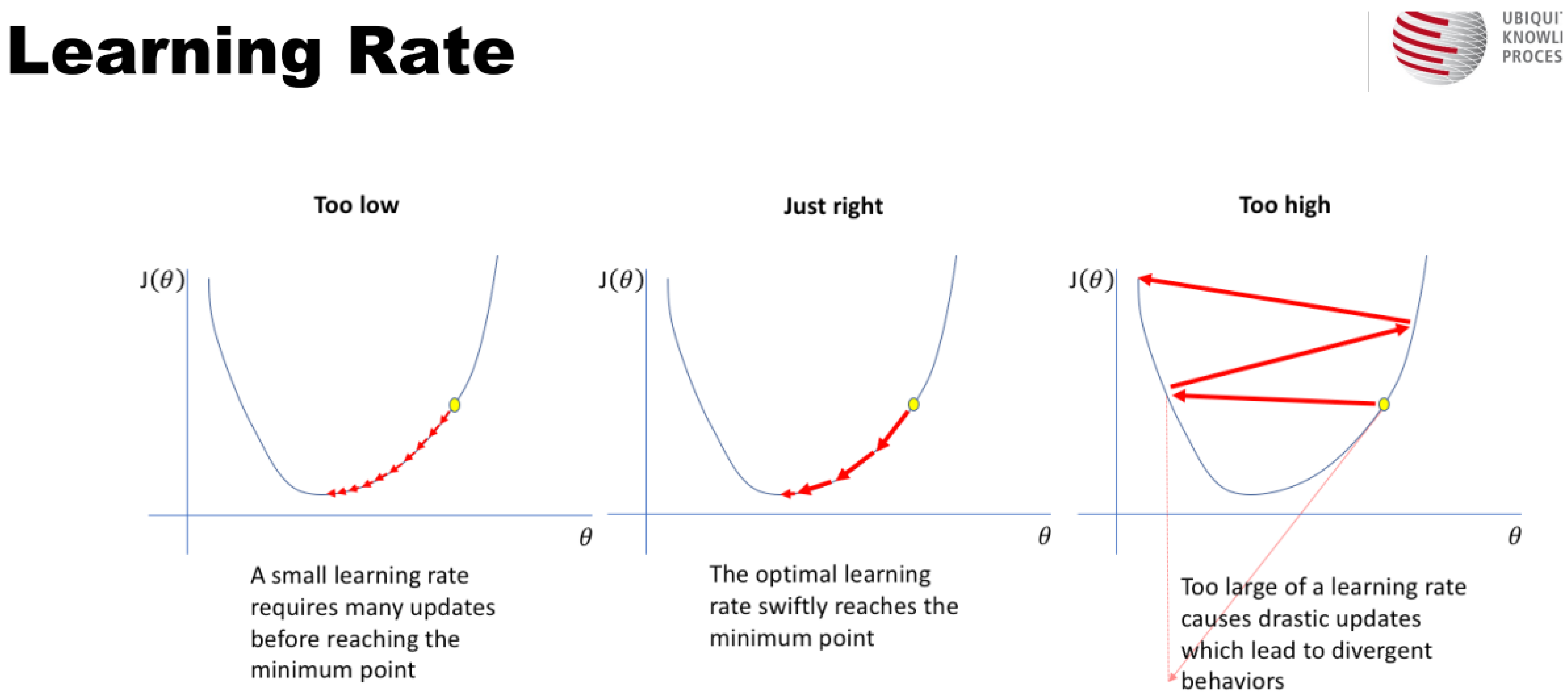
○ Step 4. Update parameters

When to Stop Training?

- **Epoch** or **parameter updating steps**. In the context of training, one epoch refers to one time when the model sees the entire training examples
  - Set a maximum number of epoch
- **Validation performance**. Performance evaluation on the validation set.
  - Early stopping: If the validation performance does not increase after k (parameter-)updating steps
    - k is called patience

2)Hyperparameters

- There are "micro-decisions" to be made for each ML (NLP) methods
  - N-gram LMs
    - N in n-gram, which can be 1, 2, 3, 4, …
    - α in Add-α smoothing, $\alpha < 1$ or $\alpha = 1$ (Laplace smoothing)
  - Neural networks:
    - Types of neural layers: most studies recently use Transformers
    - Number of neural layers, e.g., Llama-2 7B uses 32 layers
    - Activation functions, e.g. ReLU is one of the most used
    - Learning rate



o How to choose hyperparameters?
Try different values, and choose one using a validation dataset
Try a systematic and replicable procedure for finding the best values; report this procedure

PART4:Neural Language Models

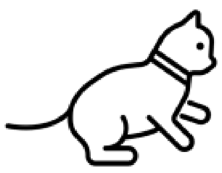
Semantic Word Similarity

- Two words with $N(w_1) \gg N(w_2)$
  - where $N(w_i)$ is count of word i, $\gg$ denotes *much larger than*
  - E.g., $N(\text{cat}) \gg N(\text{kitten})$
- Can $P(\text{cat}|\text{saw a})$ give us information about $P(\text{kitten}|\text{saw a})$?
- N-gram language models: no
- Neural language models: yes
  - Can be extremely time-consuming (hours-days)
  - More complex / less transparent
  - Robust over longer contexts
  - Usually lower perplexity than n-gram

Recap: Language Models

- A language model (LM) $\boxed{\text{measures the probability of a text sequence.}}$ Using the chain rule, we get:

$$P(w_1 w_2 \dots w_T) = P(w_1)P(w_2|w_1) \dots P(w_T|w_1 w_2 \dots w_{T-1}) = \prod_i^T P(w_i|w_1 \dots w_{i-1}) = \prod_i^T P(w_i|w_{<i})$$



P(I saw a cat on a mat) =

P(I)
· P(saw | I)
· P(a | I saw)
· P(cat | I saw a)
· P(on | I saw a cat)
· P(a | I saw a cat on)
· P(mat | I saw a cat on a)

$$\boxed{P(w_T|w_{<T}) = P(w_T|w_1 w_2 \dots w_{T-1}) = \frac{N(w_1 \dots w_{T-1} w_T)}{N(w_1 \dots w_{T-1})}}$$

Instead of looking up the count of a word or n-gram, neural language models do:

o encoding the previous words

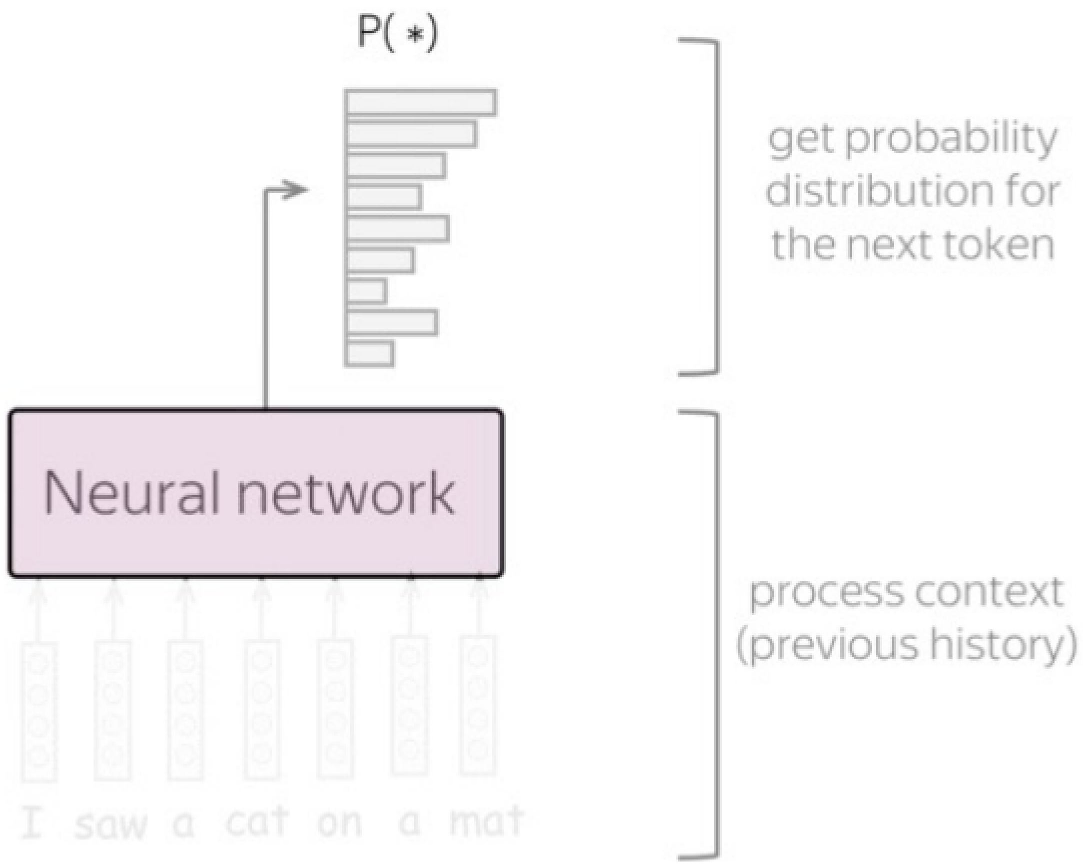
model-specific

Idea: get a vector representation for the previous words (computers work with numbers)

We can use different model architecture, e.g., RNN, CNN, Transformers

o transforming it into a distribution over the next word

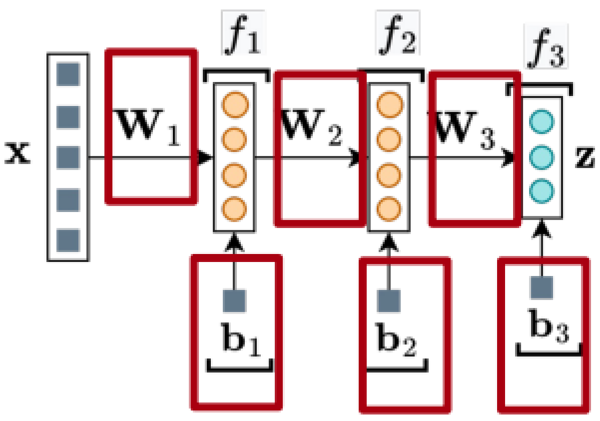
Idea: get the probability distribution for the next token



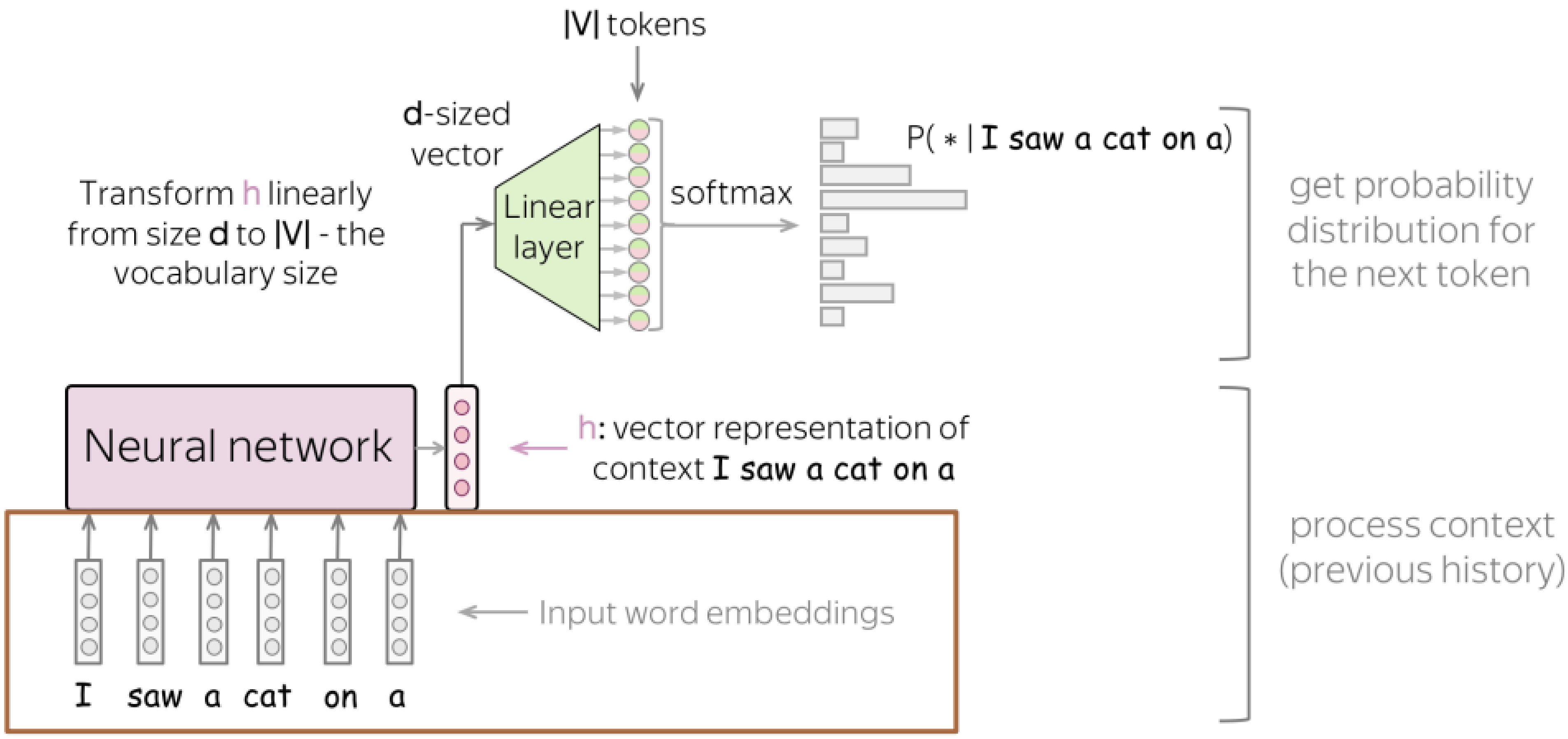
- Formally, we compute $P(w_T|w_{<T})$ using neural network
  - Input $w_{<T}$: previous words
  - Expected output $P(w_T|w_{<T})$: a probability distribution for the next word w_T

$$P(w_T|w_{<T}) = \boxed{\mathrm{NN}(\mathrm{enc}(w_{<T});\ \theta)}$$

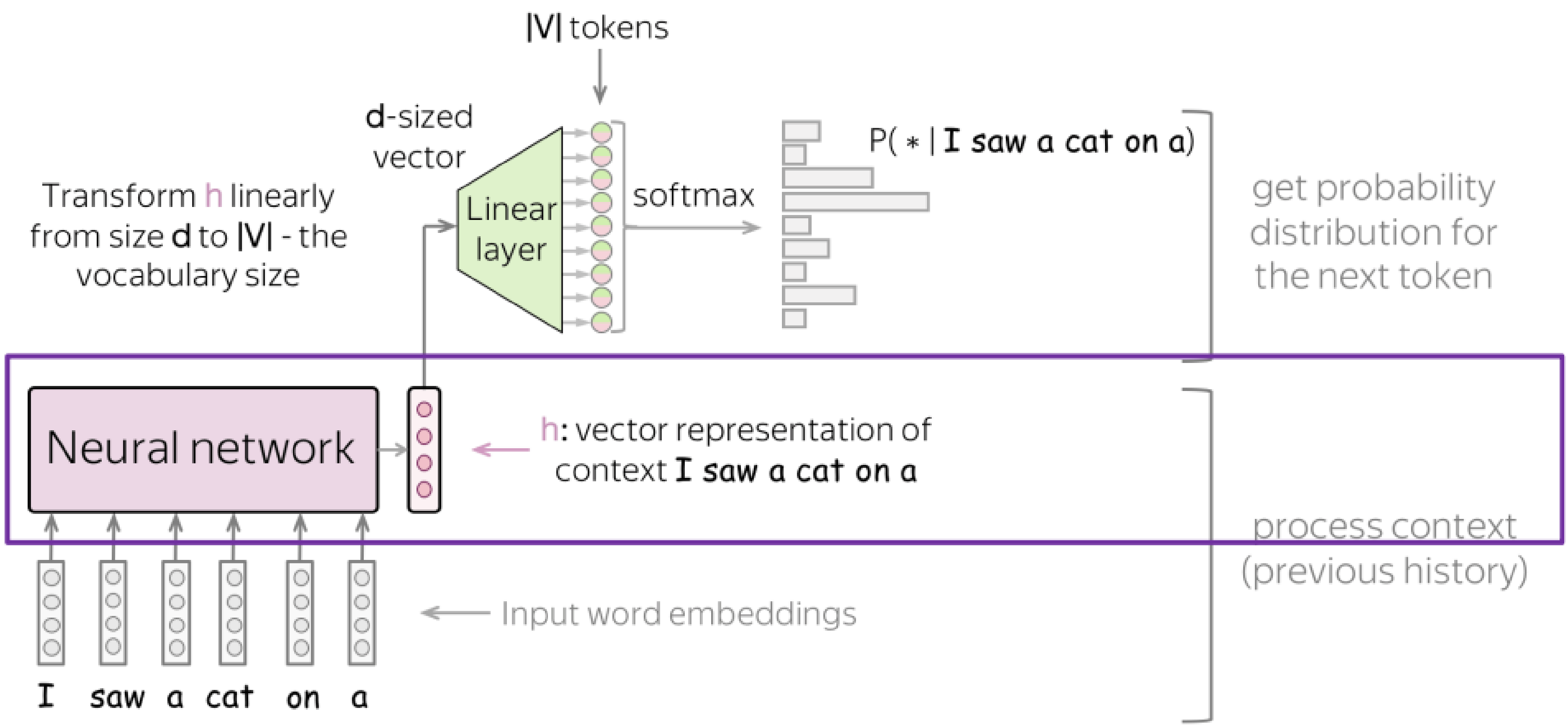
- where θ (trained parameters or weights of a model) do
  - encoding the previous words – $\mathrm{enc}(\cdot)$
  - transforming it into a distribution for the next word – $\mathrm{NN}(\cdot)$



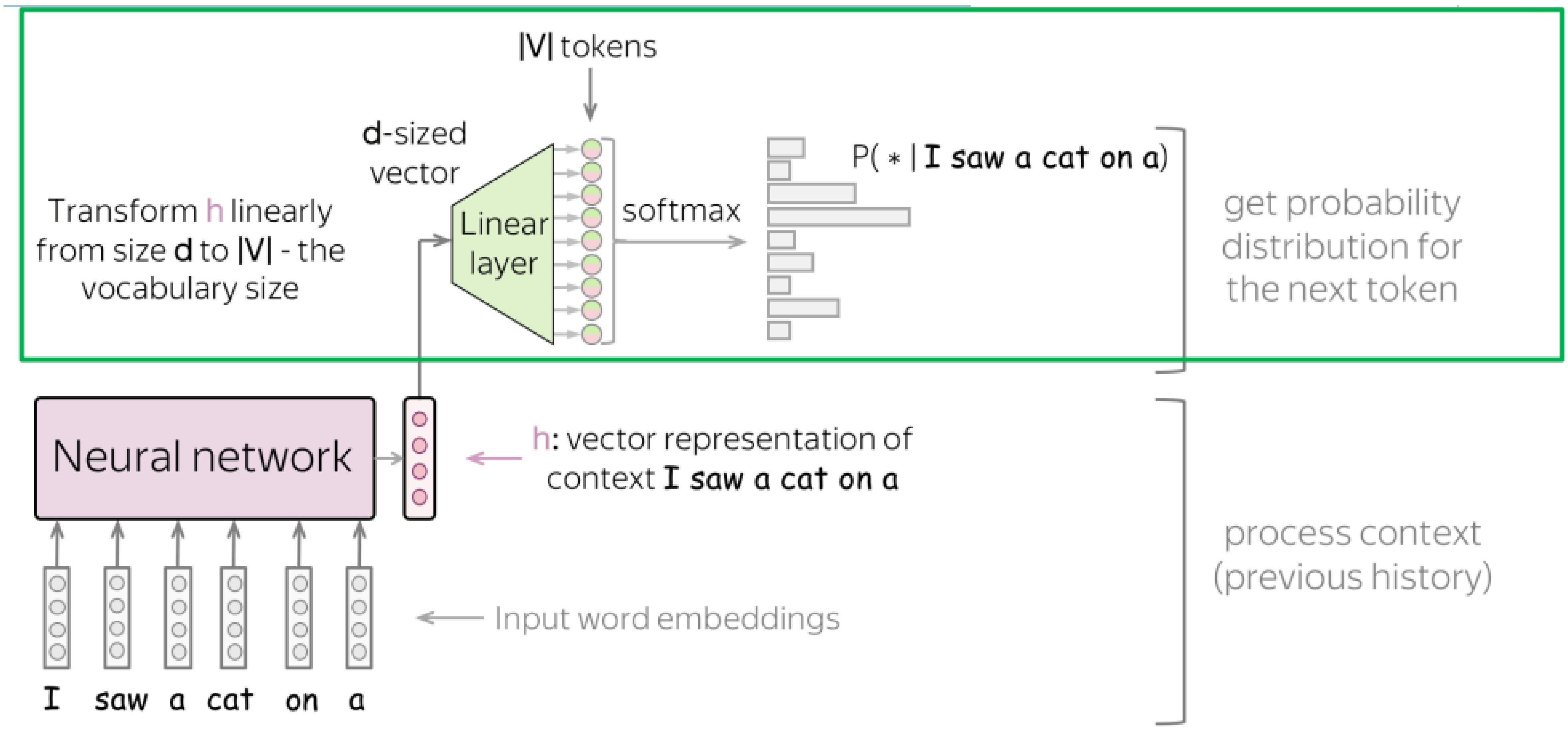
1) Get Word Embeddings(later in course)



2)Get vector representation of context



3)Predict Next Word (Token)

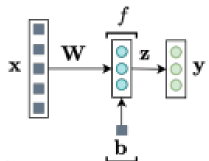


We have:

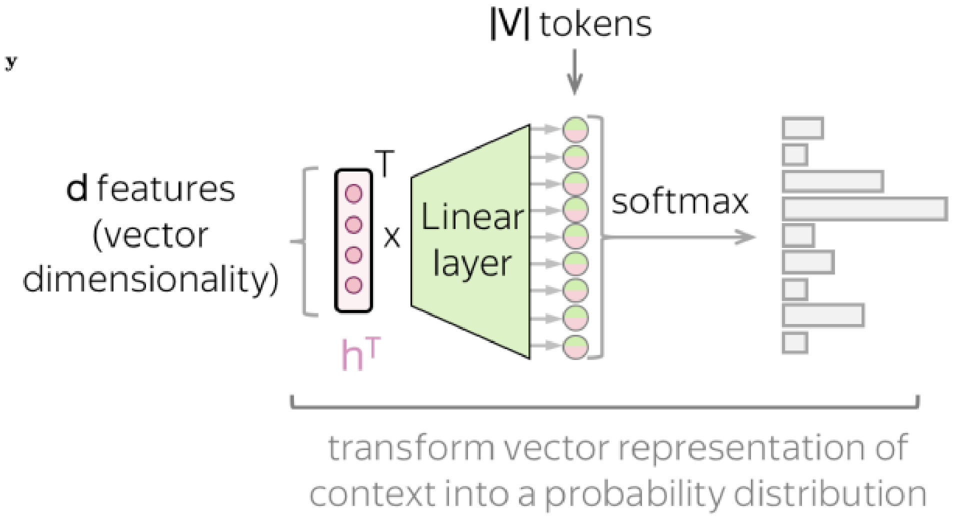
- h - vector of size d

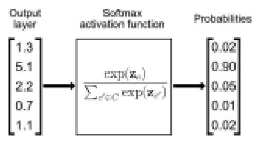
We need:

- vector of size |V| – probabilities for all tokens in the vocabulary



Transform linearly from size d to size |V|





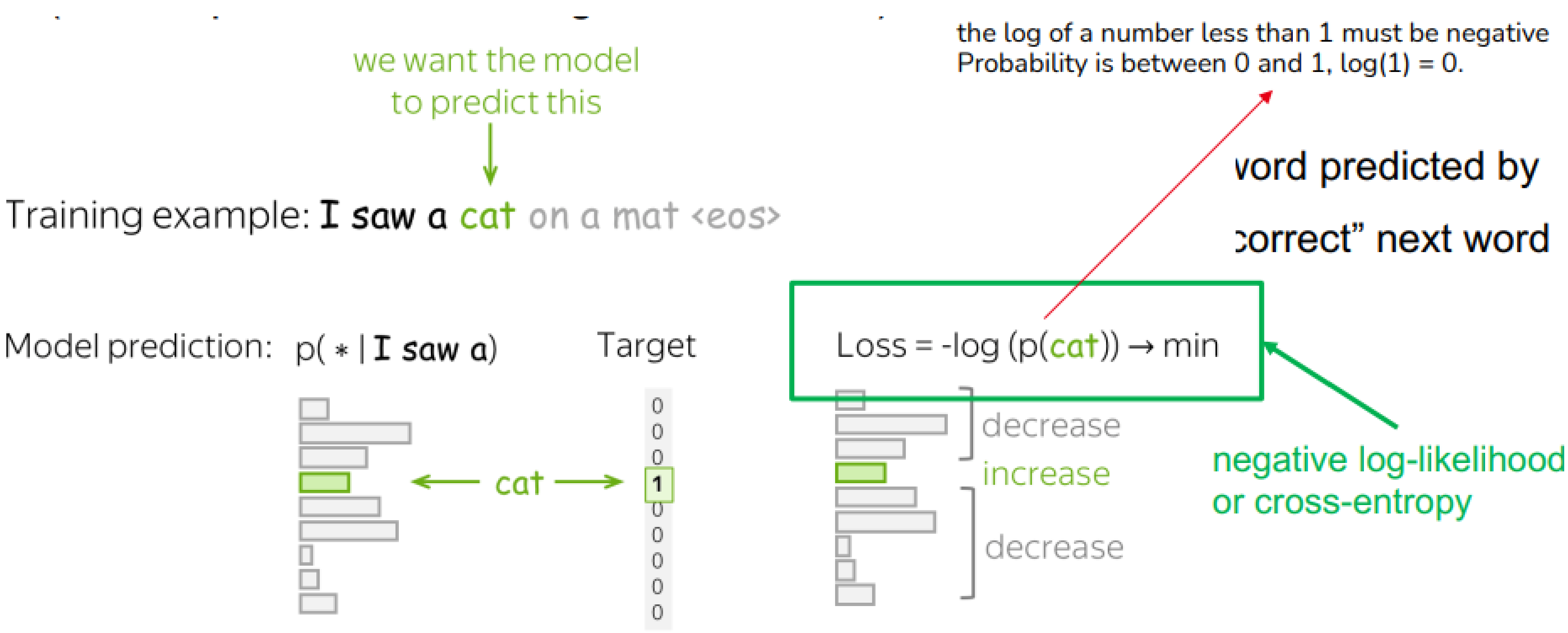
$$\mathbf{y} = \mathrm{softmax}(\mathbf{hW} + \mathbf{b})$$

$$\boldsymbol{h} \in \mathbb{R}^d, \mathbf{W} \in \mathbb{R}^{d \times |V|}, \mathbf{b} \in \mathbb{R}^{|V|}$$

Training

$$P(w_T | w_{<T}) = \mathrm{NN}(\mathrm{enc}(w_{<T});\ \theta)$$

- Training: updating θ (trained parameters or weights of a model)
  - Using *stochastic gradient descent* algorithms to maximize likelihood
  - **Maximize likelihood estimate** (MLE): maximize the probability of the "correct" next word predicted by the model, i.e., **minimize error** between the model's predicted probabilities and the "correct" next word



Inference

- Inference: use the trained model (trained parameters/weights θ) to compute $P(w_T|w_{<T})$