

Understanding Large Language Models

Carsten Eickhoff, Michael Franke and Polina Tsvilodub

Session 02: PyTorch, Optimization, ANNs, LMs & RNNs

Main learning goals

1. PyTorch

- simple optimization problems with stochastic gradient descent (SGD)
- basic usage of `nn.Module` and `Dataset` classes

2. Optimization (via Backpropagation)

- basic concepts: loss function, gradients, backpropagation, SGD
- anatomy of update step & training loop (in PyTorch)

3. Artificial Neural Networks (specifically: Multi-Layer Perceptrons)

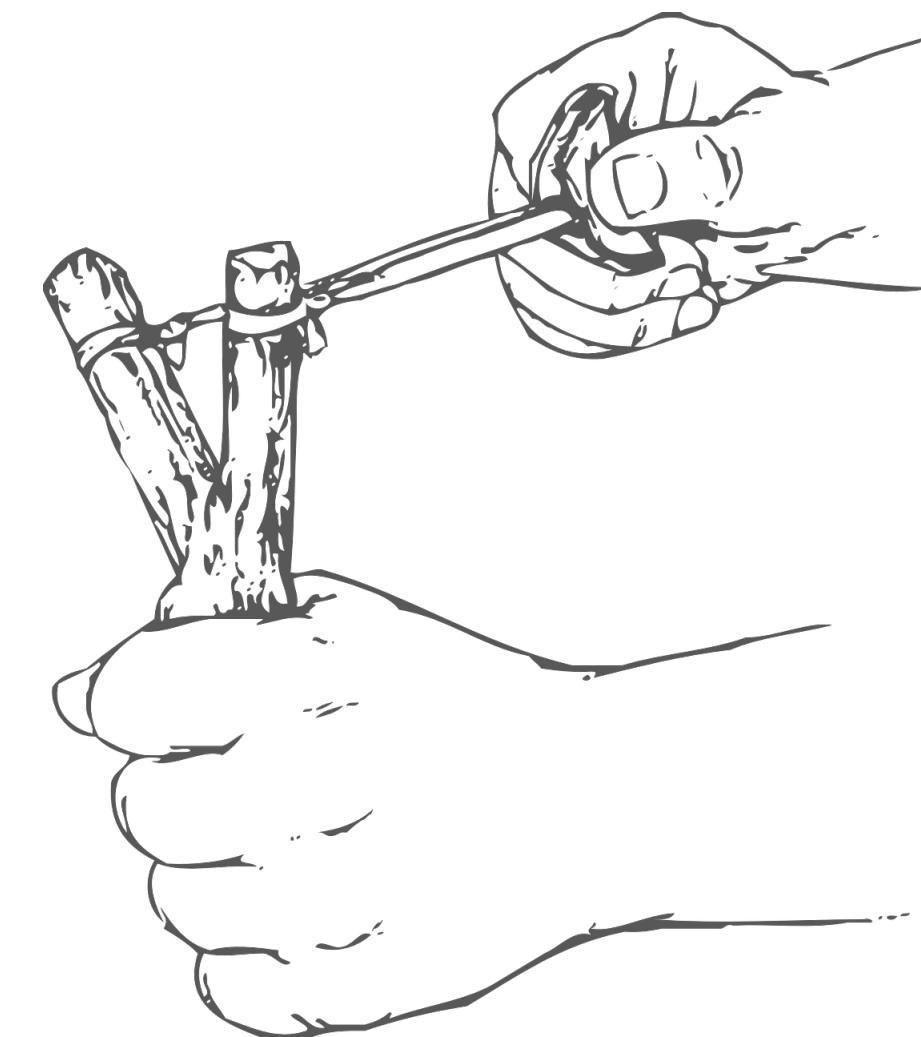
- definitions of ANNs & MLPs, mathematical notation in matrix-vector form
- concepts: weights & biases (slopes & intercepts), activation (function), hidden layers, score, prediction (sample, probability)

4. Language Models

- definition of (autoregressive) language models, loss functions, decoding

5. Recurrent Neural Networks

- definition & example (character-level RNN for surname generation)





PyTorch

Key features



- ▶ **high-level framework for ML**
 - especially for artificial neural networks
- ▶ **efficient tensor algebra**
 - ability to run on GPUs
- ▶ **pre-defined building blocks for ANNs**
 - standard layers, data handling etc.
- ▶ **automatic differentiation**
 - enables efficient optimization

```
for i in range(n_training_steps):
    prediction = torch.distributions.Normal(loc=location, scale=1.0)
    loss = -torch.sum(prediction.log_prob(train_data))
    loss.backward()
    opt.step()
    opt.zero_grad()
```

demo PyTorch



demo 01: PyTorch essentials

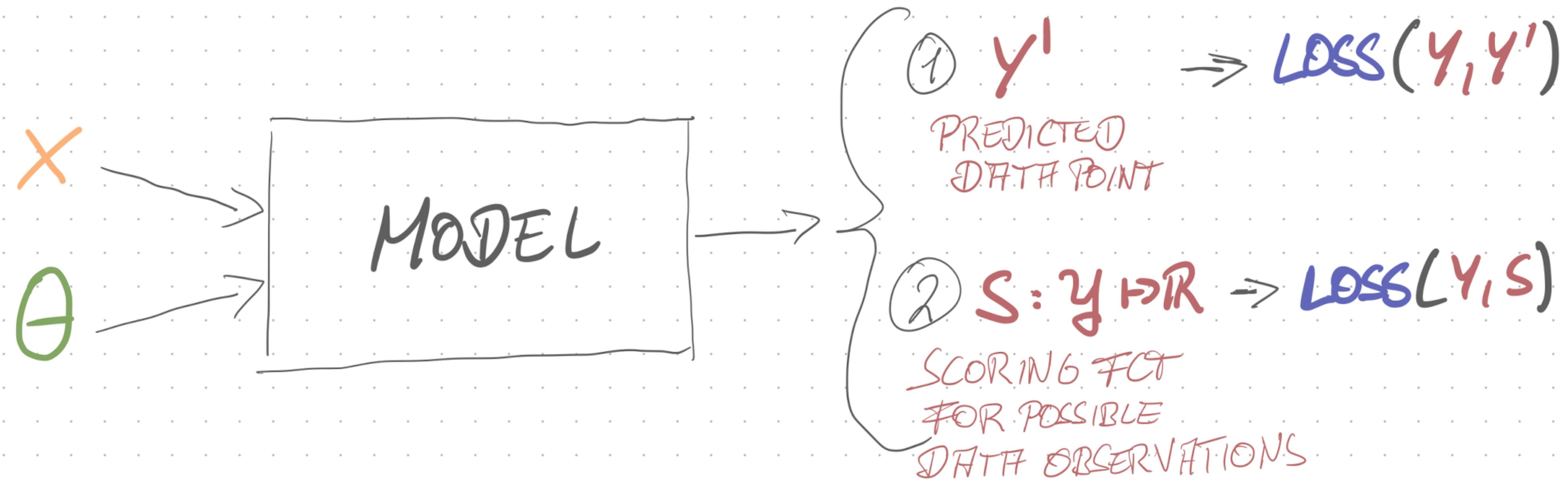


Optimization

Models, parameters, predictions & loss

① DATA : $\langle x, y \rangle$

MODEL: $f_\theta: x \mapsto y$

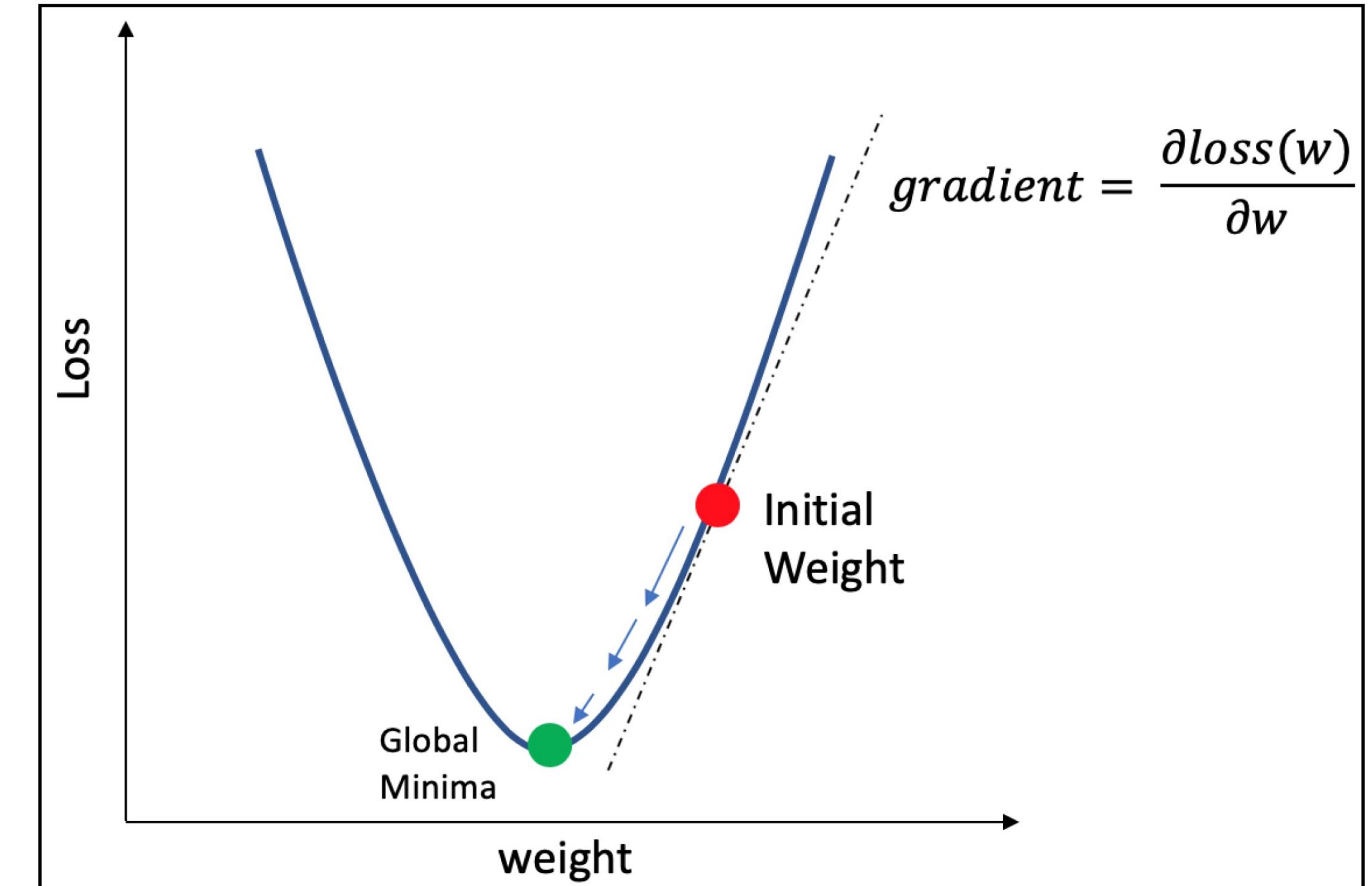


Optimization

for probabilistic models

- ▶ given:
 - data $D = \langle X, Y \rangle$
 - probabilistic model $M: \Theta, X \rightarrow \Delta(Y)$
 - loss function $L: \Theta, X, Y \rightarrow \mathbb{R}$
 - most commonly used is negative log likelihood:
$$L(\theta, x, y) = -\log P_{M(\theta, x)}(y)$$
- ▶ find parameters that minimize loss for data:

$$\hat{\theta} = \arg \min_{\theta \in \Theta} = \sum_{x \in X, y \in Y} L(\theta, x, y)$$



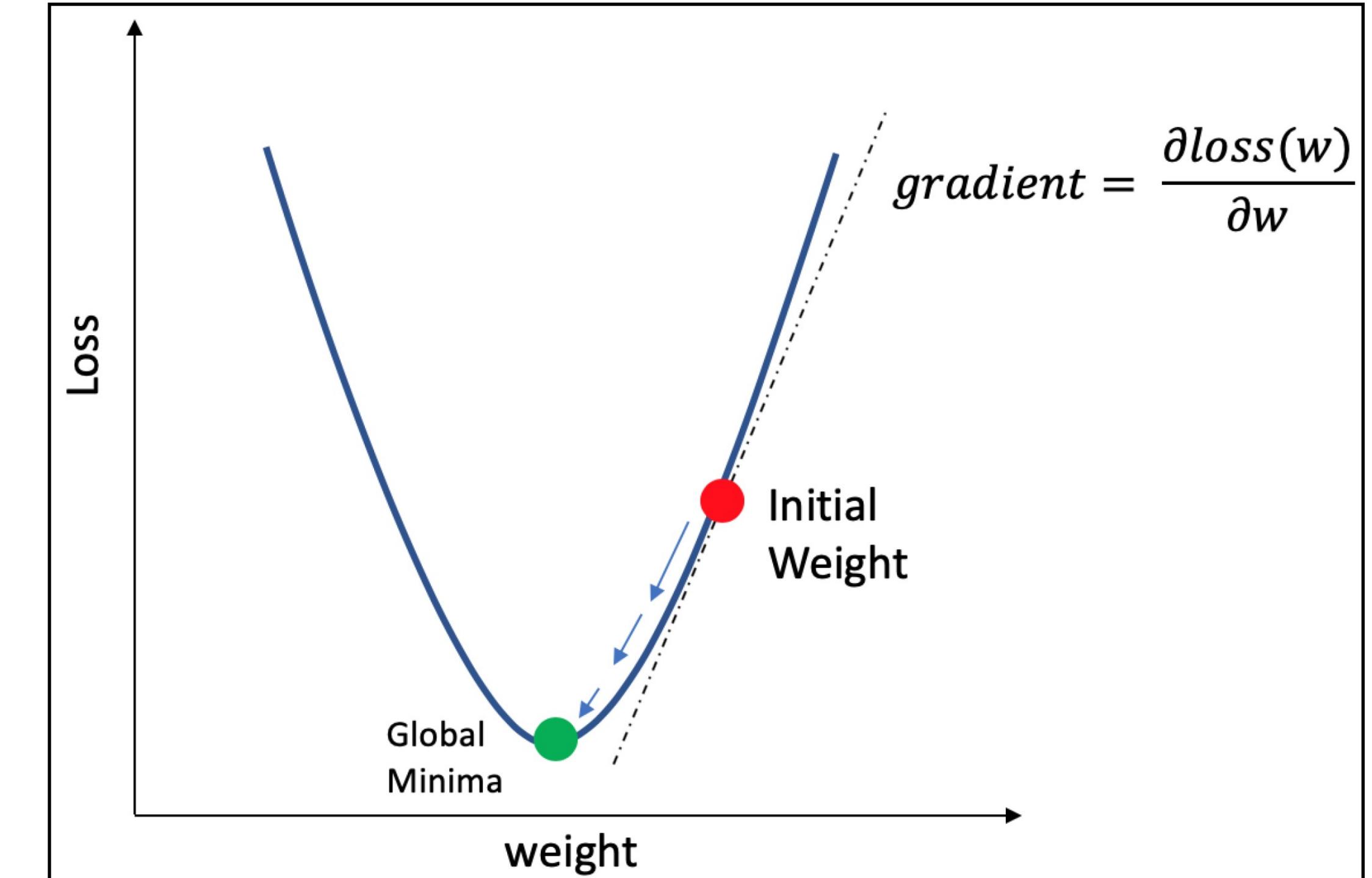
Stochastic gradient descent

input : γ (lr), θ_0 (params), $f(\theta)$ (objective), λ (weight decay),
 μ (momentum), τ (dampening), nesterov, maximize

for $t = 1$ **to** ... **do**

```
     $g_t \leftarrow \nabla_{\theta} f_t(\theta_{t-1})$ 
    if  $\lambda \neq 0$ 
         $g_t \leftarrow g_t + \lambda \theta_{t-1}$ 
    if  $\mu \neq 0$ 
        if  $t > 1$ 
             $b_t \leftarrow \mu b_{t-1} + (1 - \tau) g_t$ 
        else
             $b_t \leftarrow g_t$ 
        if nesterov
             $g_t \leftarrow g_t + \mu b_t$ 
        else
             $g_t \leftarrow b_t$ 
        if maximize
             $\theta_t \leftarrow \theta_{t-1} + \gamma g_t$ 
        else
             $\theta_t \leftarrow \theta_{t-1} - \gamma g_t$ 
```

return θ_t



from [this paper](#)

Stochastic gradient descent

input : γ (lr), θ_0 (params), $f(\theta)$ (objective), λ (weight decay),
 μ (momentum), τ (dampening), *nesterov*, *maximize*

for $t = 1$ **to** ... **do**

$g_t \leftarrow \nabla_{\theta} f_t(\theta_{t-1})$

if $\lambda \neq 0$

$g_t \leftarrow g_t + \lambda \theta_{t-1}$

if $\mu \neq 0$

if $t > 1$

$\mathbf{b}_t \leftarrow \mu \mathbf{b}_{t-1} + (1 - \tau) g_t$

else

$\mathbf{b}_t \leftarrow g_t$

if *nesterov*

$g_t \leftarrow g_t + \mu \mathbf{b}_t$

else

$g_t \leftarrow \mathbf{b}_t$

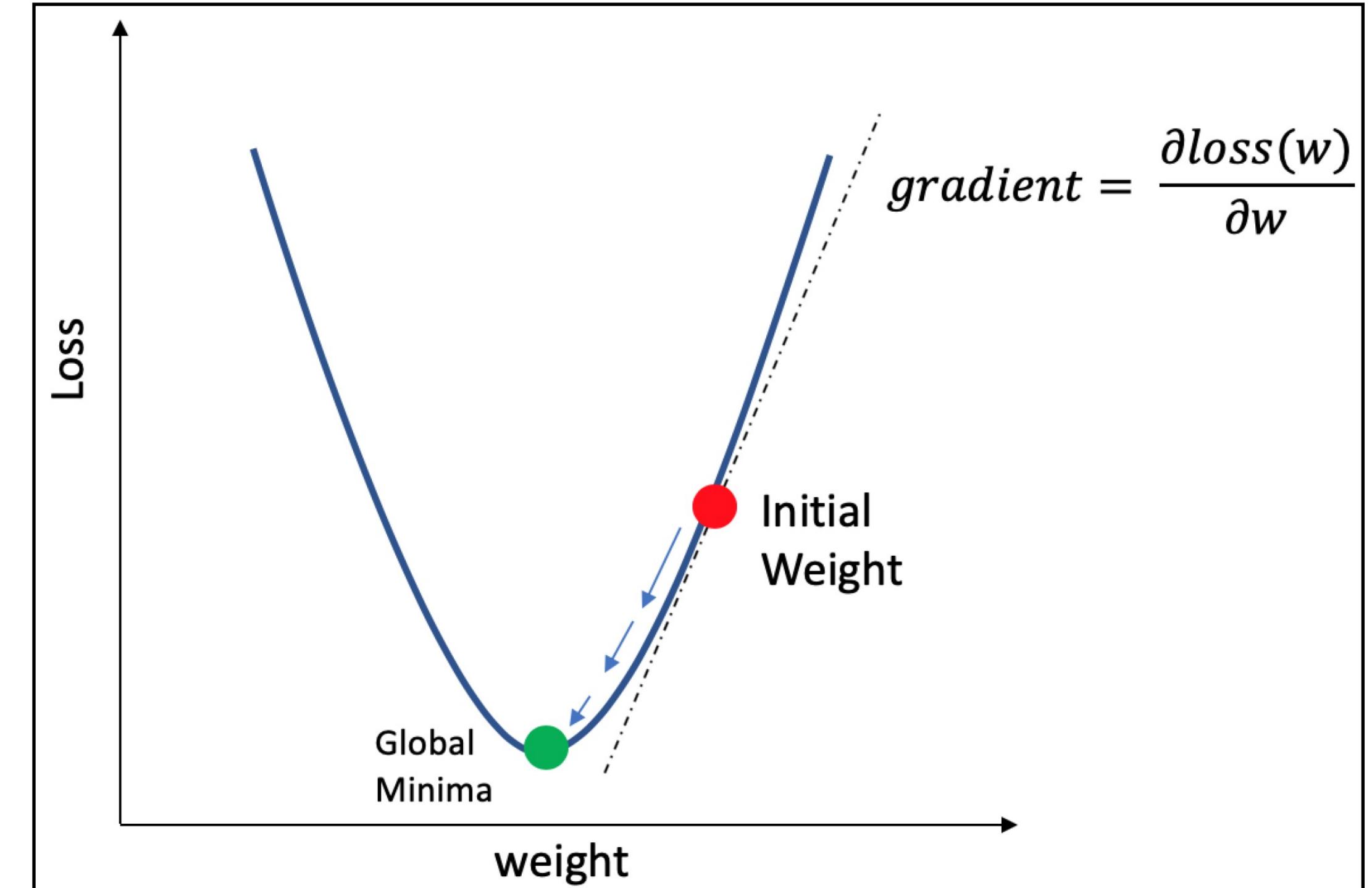
if *maximize*

$\theta_t \leftarrow \theta_{t-1} + \gamma g_t$

else

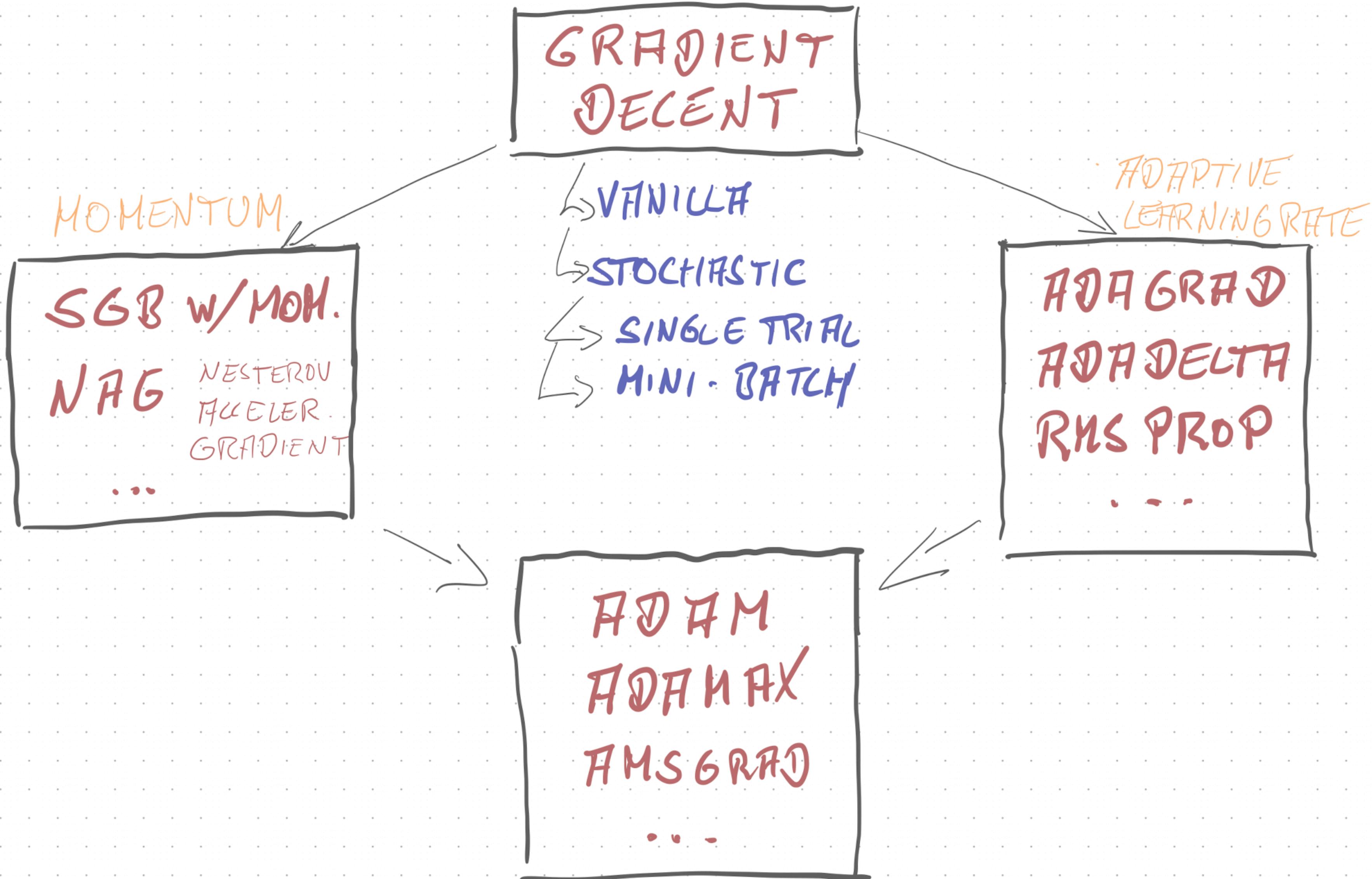
$\theta_t \leftarrow \theta_{t-1} - \gamma g_t$

return θ_t



from [this paper](#)

Common optimization algorithms



Anatomy of a training step

1. compute predictions

- what do we predict in the current state?

2. compute the loss

- how good is this prediction (for the training data)?

3. backpropagate the error

- in which direction would we need to change the relevant parameters to make the prediction better?

4. update the parameters

- change the parameters (to a certain degree, the so-called learning rate) in the direction that should make them better

5. zero the gradients

- reset the information about “which direction to tune” for the next training step

```
for i in range(n_training_steps):
    prediction = torch.distributions.Normal(loc=location, scale=1.0)
    loss = -torch.sum(prediction.log_prob(train_data))
    loss.backward()
    opt.step()
    opt.zero_grad()
```

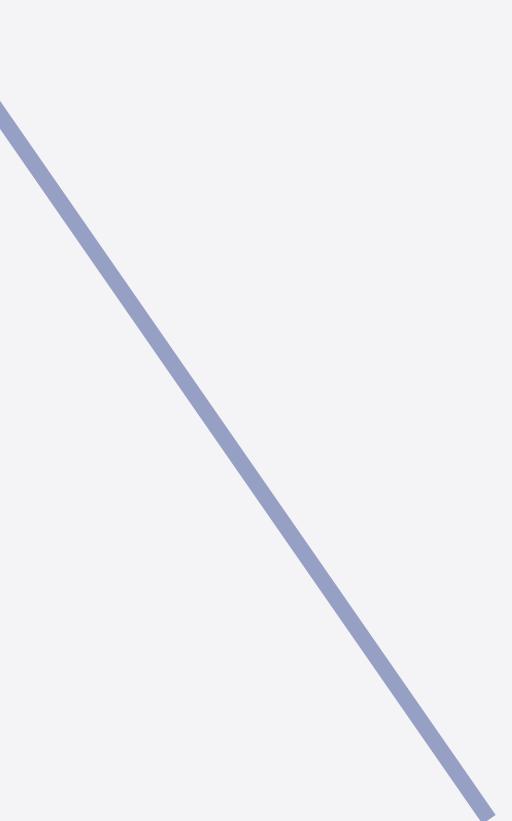
demo optimization



demo 02: Maximum Likelihood Estimation (in PyTorch)



Artificial Neural Networks



Units neurons

- ▶ input vector:

$$\mathbf{x} = [x_1, \dots, x_n]^T$$

- ▶ weight vector:

$$\mathbf{w} = [w_1, \dots, w_n]^T$$

- ▶ bias:

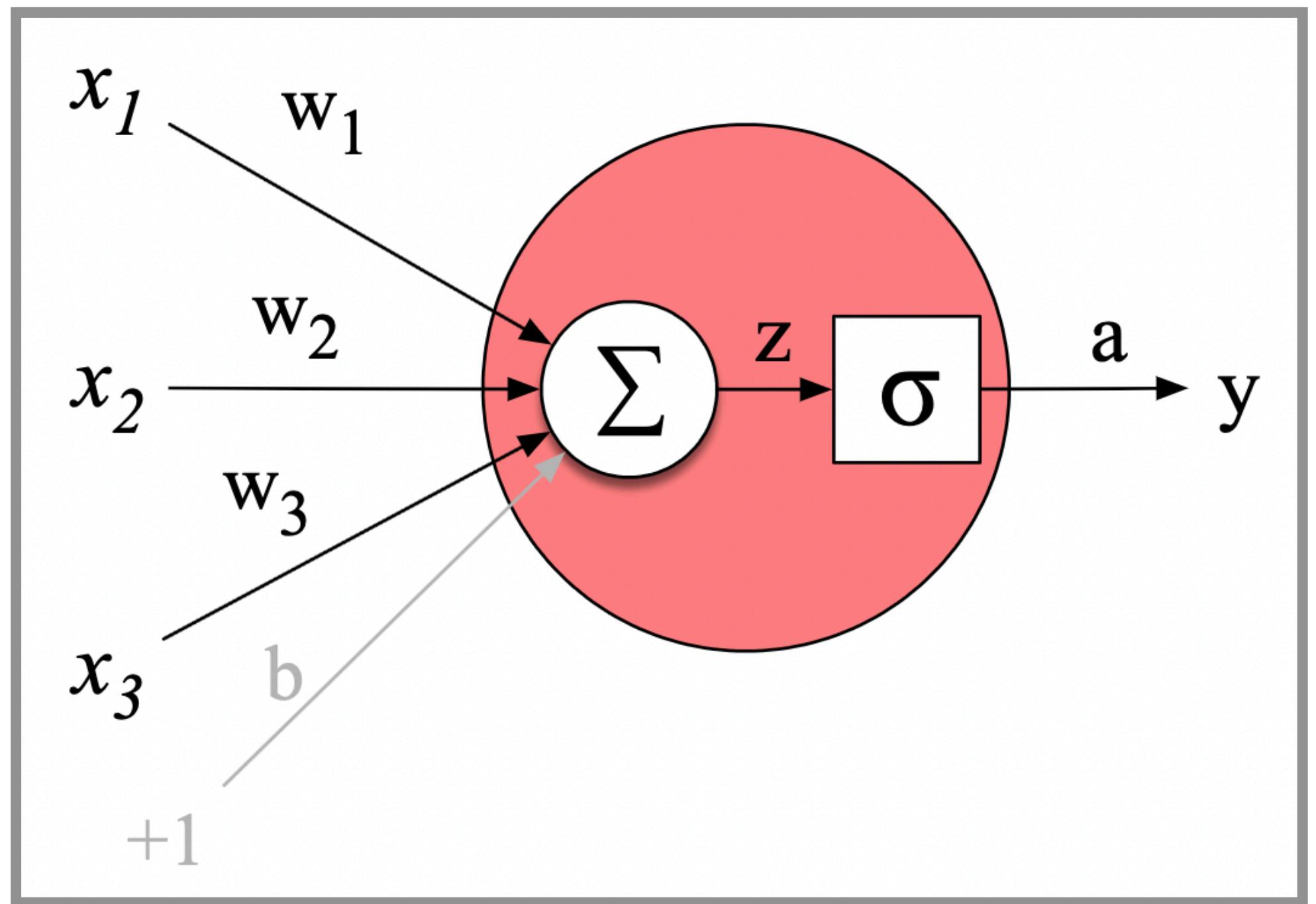
$$b$$

- ▶ score:

$$z = b + \sum_{j=1}^n w_j x_j = b + \mathbf{w} \cdot \mathbf{x}$$

- ▶ activation level:

- $a = f(z)$, where f is the **activation function**



Common activation functions

- ▶ perceptron:

$$f(z) = \delta_{z>0}$$

- ▶ sigmoid:

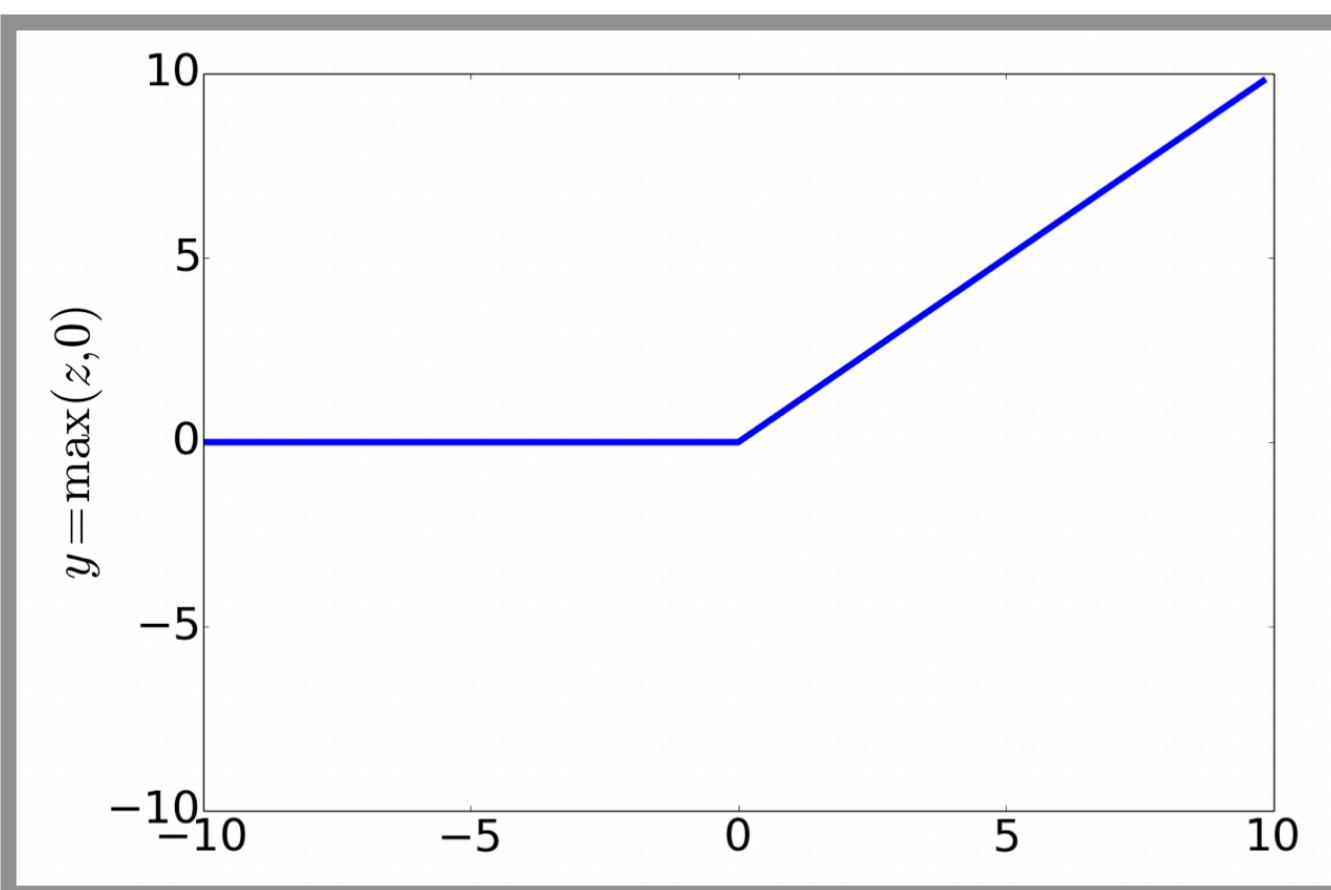
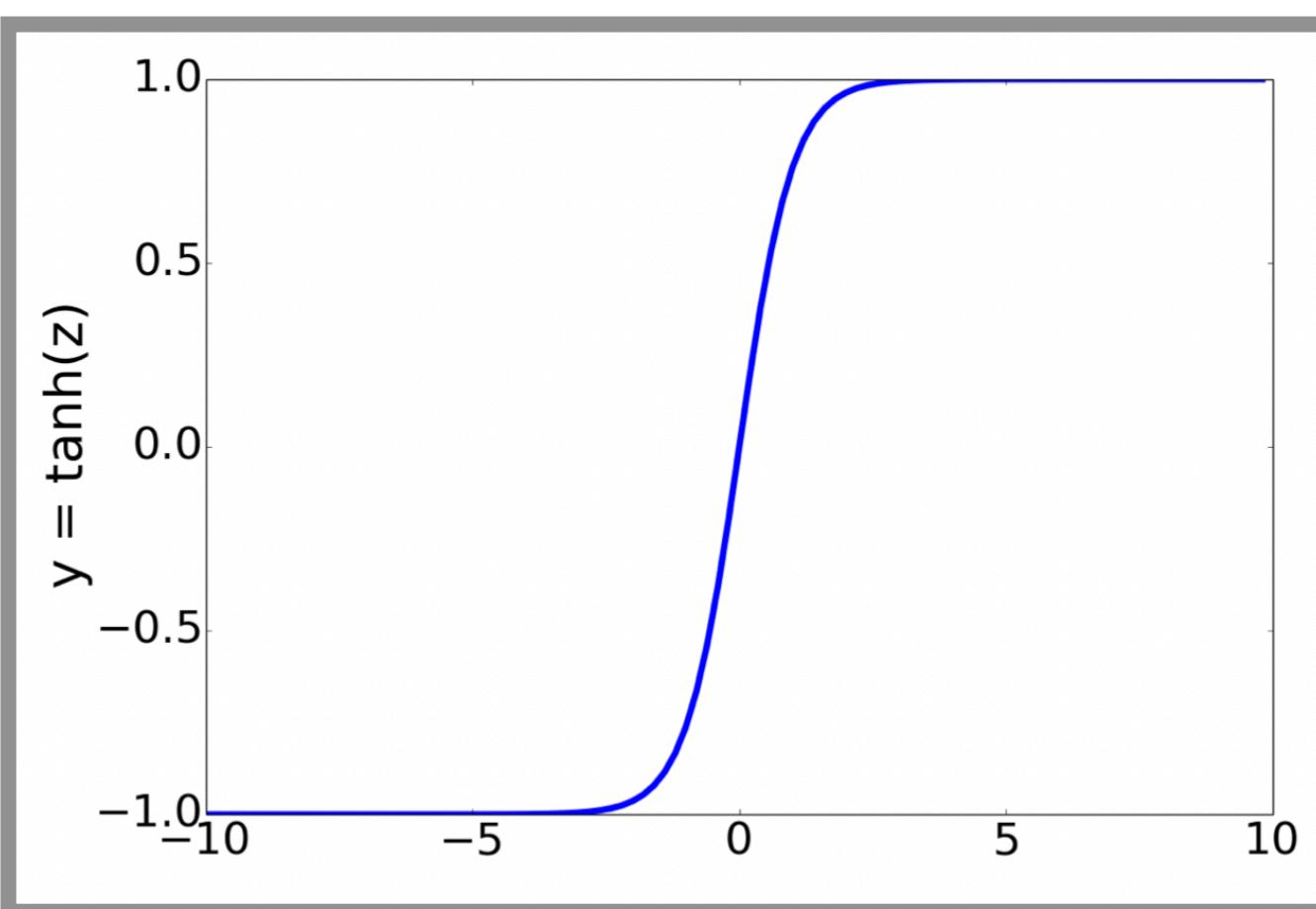
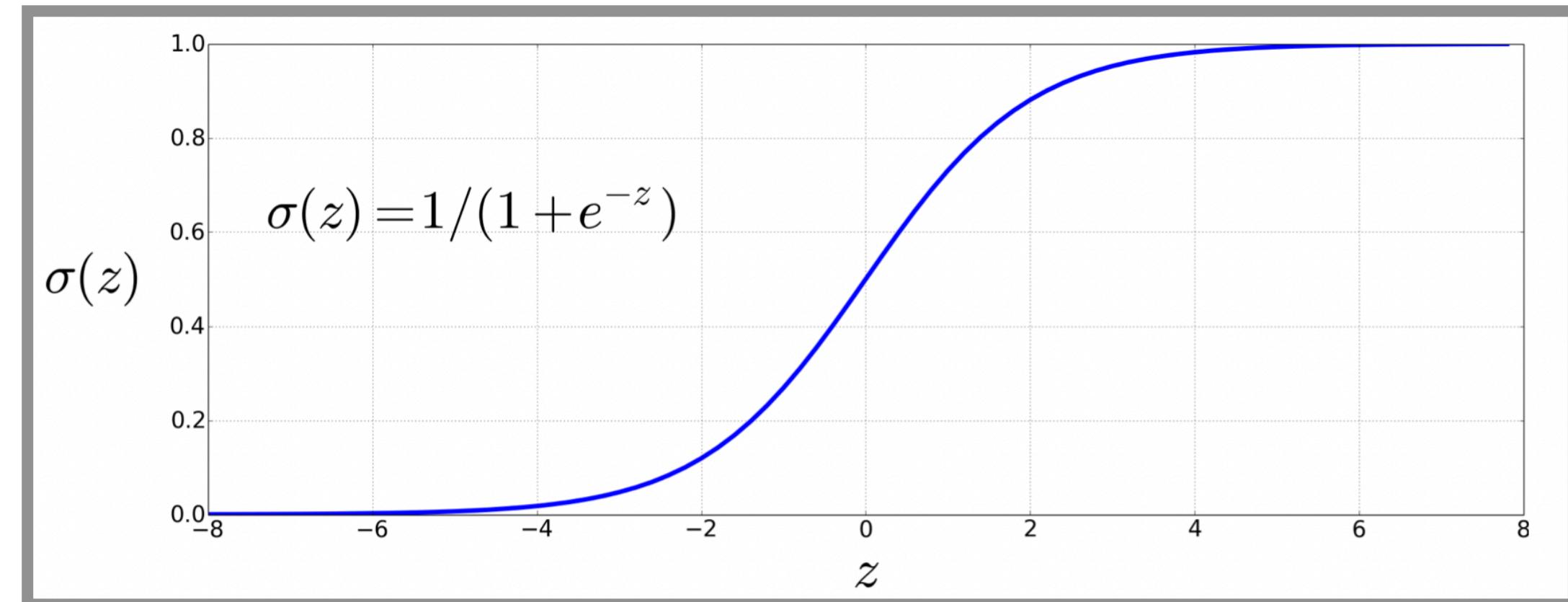
$$f(z) = \sigma(z) = \frac{1}{1 + \exp(-z)}$$

- ▶ hyperbolic tangent:

$$f(z) = \tanh(z) = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)}$$

- ▶ rectified linear unit:

$$f(z) = \text{ReLU}(z) = \max(z, 0)$$



Matrix multiplication

recap

$$c_{11} = a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} + a_{14}b_{41}$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \\ b_{41} & b_{42} & b_{43} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \end{bmatrix}$$

2×4 4×3 2×3

$$c_{22} = a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} + a_{24}b_{42}$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \\ b_{41} & b_{42} & b_{43} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \end{bmatrix}$$

Matrix-vector multiplication

recap

$$\begin{array}{c} \begin{bmatrix} 1,1 & 1,2 & 1,3 \\ 2,1 & 2,2 & 2,3 \\ 3,1 & 3,2 & 3,3 \end{bmatrix} \\ A \end{array} \begin{matrix} \times \\ X \end{matrix} \begin{array}{c} \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \\ = \end{array} \begin{array}{c} \begin{bmatrix} 1,1 & 1 \\ 2,1 & 1 \\ 3,1 & 1 \end{bmatrix} + \begin{bmatrix} 1,2 & 2 \\ 2,2 & 2 \\ 3,2 & 2 \end{bmatrix} + \begin{bmatrix} 1,3 & 3 \\ 2,3 & 3 \\ 3,3 & 3 \end{bmatrix} \\ = \end{array} \begin{array}{c} \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \\ Ax \end{array}$$

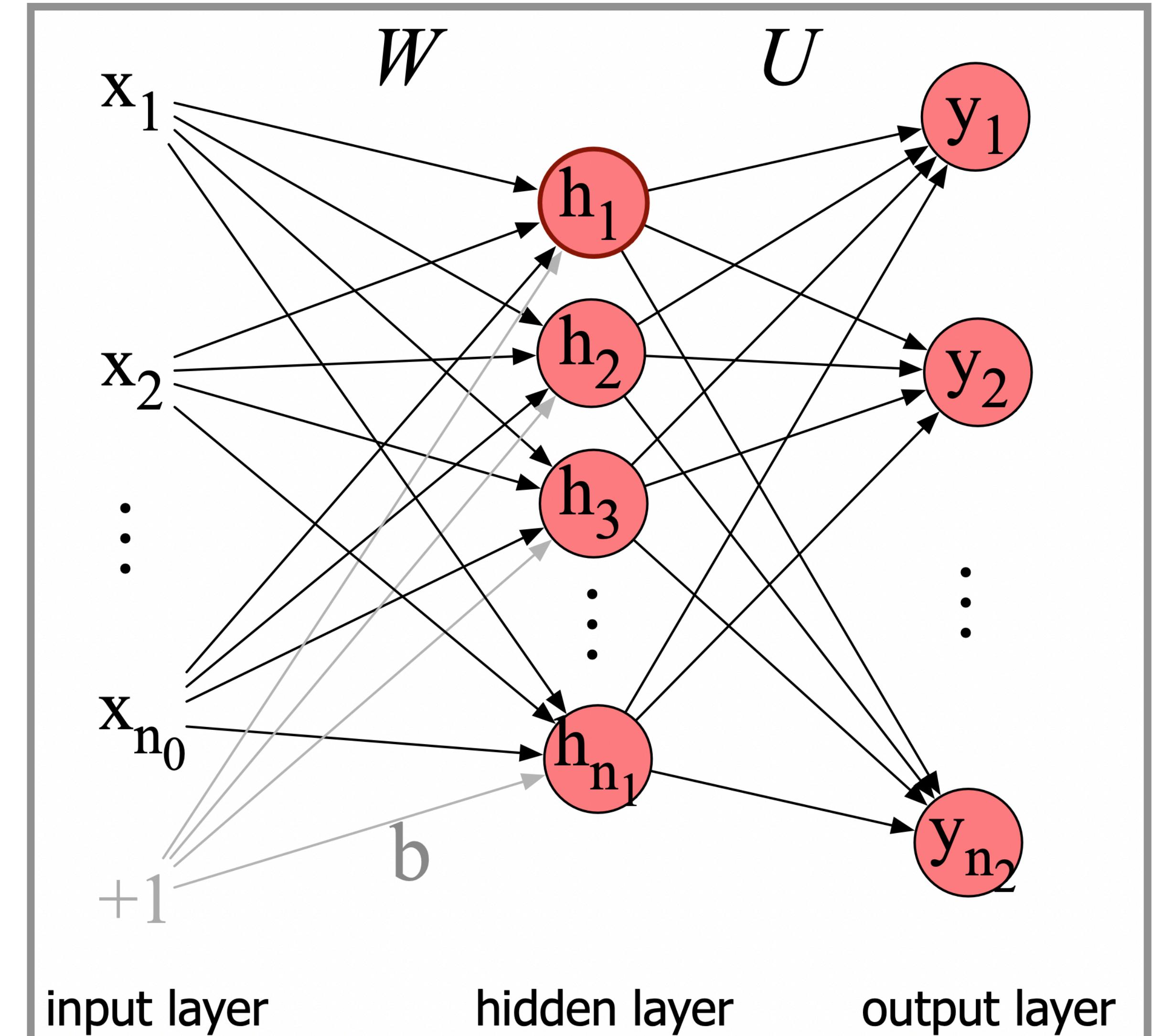
- think of matrix \mathbf{A} with dimensions (n, m) as a **linear mapping** $f_{\mathbf{A}}: \mathbb{R}^n \rightarrow \mathbb{R}^m$ from vectors of length m to vectors of length n , so that with $\mathbf{x} = [x_1, \dots, x_m]$:

$$f_{\mathbf{A}}(\mathbf{x}) = \mathbf{Ax}$$

Feed-forward neural network

one layer

- ▶ input:
 $\mathbf{x} = [x_1, \dots, x_{n_x}]^T$
- ▶ weight matrix:
 $\mathbf{W} \in \mathbb{R}^{n_k \times n_x}$
- ▶ bias vector:
 $\mathbf{b} = [b_1, \dots, b_{n_k}]^T$
- ▶ activation vector hidden layer:
 $\mathbf{h} = f(\mathbf{W}\mathbf{x} + \mathbf{b})$, with $f \in \{\sigma, \tanh, \dots\}$
- ▶ weight matrix:
 $\mathbf{U} \in \mathbb{R}^{n_y \times n_k}$
- ▶ prediction vector:
 $\mathbf{y} = g(\mathbf{U}\mathbf{h})$, with $g \in \{\sigma, \text{soft-max}, \dots\}$



Feed-forward neural network

multi-layer perceptron

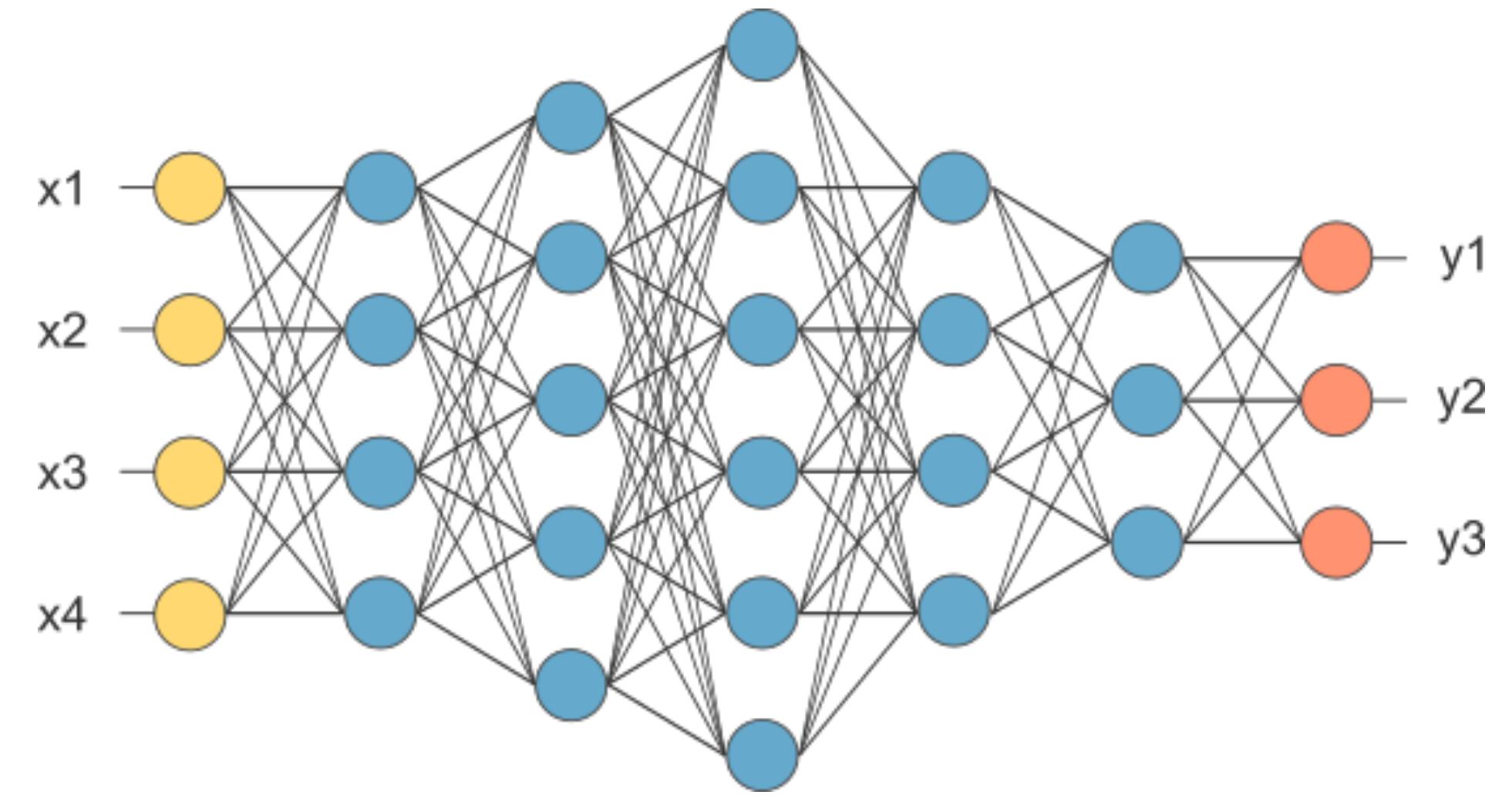
- ▶ anchoring in input:

$$\mathbf{a}^{[0]} = \mathbf{x} = [x_1, \dots, x_{n_x}]^T$$

- ▶ activation at layer n :

$$\mathbf{a}^{[n]} = f^{[n]} (\mathbf{W}^{[n]} \mathbf{a}^{[n-1]} + \mathbf{b}^{[n]})$$

- with $f^{[n]} \in \{\sigma, \tanh, \dots\}$ if n is a hidden layer, or
- with $f^{[n]} \in \{\sigma, \text{soft-max}, \dots\}$ if n is the output layer



demo
MLP



demo 03: Multi-Layer Perceptron for Non-Linear Regression



Language Models

Language model

high-level definition

- ▶ let \mathcal{V} be a (finite) **vocabulary**, a set of tokens
 - tokens can be characters, sub-words, phrases, ...
- ▶ let $w_{1:n} = \langle w_1, \dots, w_n \rangle$ be a finite sequence of tokens
 - ▶ it is still common to use w (reminiscent of “words”) for tokens
- ▶ let S be the set of all (finite) sequences of tokens
- ▶ let X be a set of input conditions
 - e.g., images, text in a different language ...
- ▶ a **language model** LM is function that assigns to each input X a probability distribution over S , given parameters $\theta \in \Theta$:

$$LM_\theta : X \mapsto \Delta(S)$$

- if there is only one input in set X , the LM is just a probability distribution over all sequences of words
- LMs originally meant to capture the true occurrence frequency
- a **neural language model** is an LM realized as a neural network
- in the following we skip the dependence on X

Language model

left-to-right / autoregressive / causal model

- ▶ a **causal language model** is defined as a **function** that maps an initial token sequence to a **next-token distribution**:

$$LM : w_{1:n} \mapsto \Delta(\mathcal{V})$$

- we write $P_{LM}(w_{n+1} | w_{1:n})$ for the **next-token probability**
- the **surprisal** of w_{n+1} after sequence $w_{1:n}$ is
$$-\log(P_{LM}(w_{n+1} | w_{1:n}))$$
- ▶ the **sequence probability** follows from the chain rule:

$$P_{LM}(w_{1:n}) = \prod_{i=1}^n P_{LM}(w_i | w_{1:i-1})$$

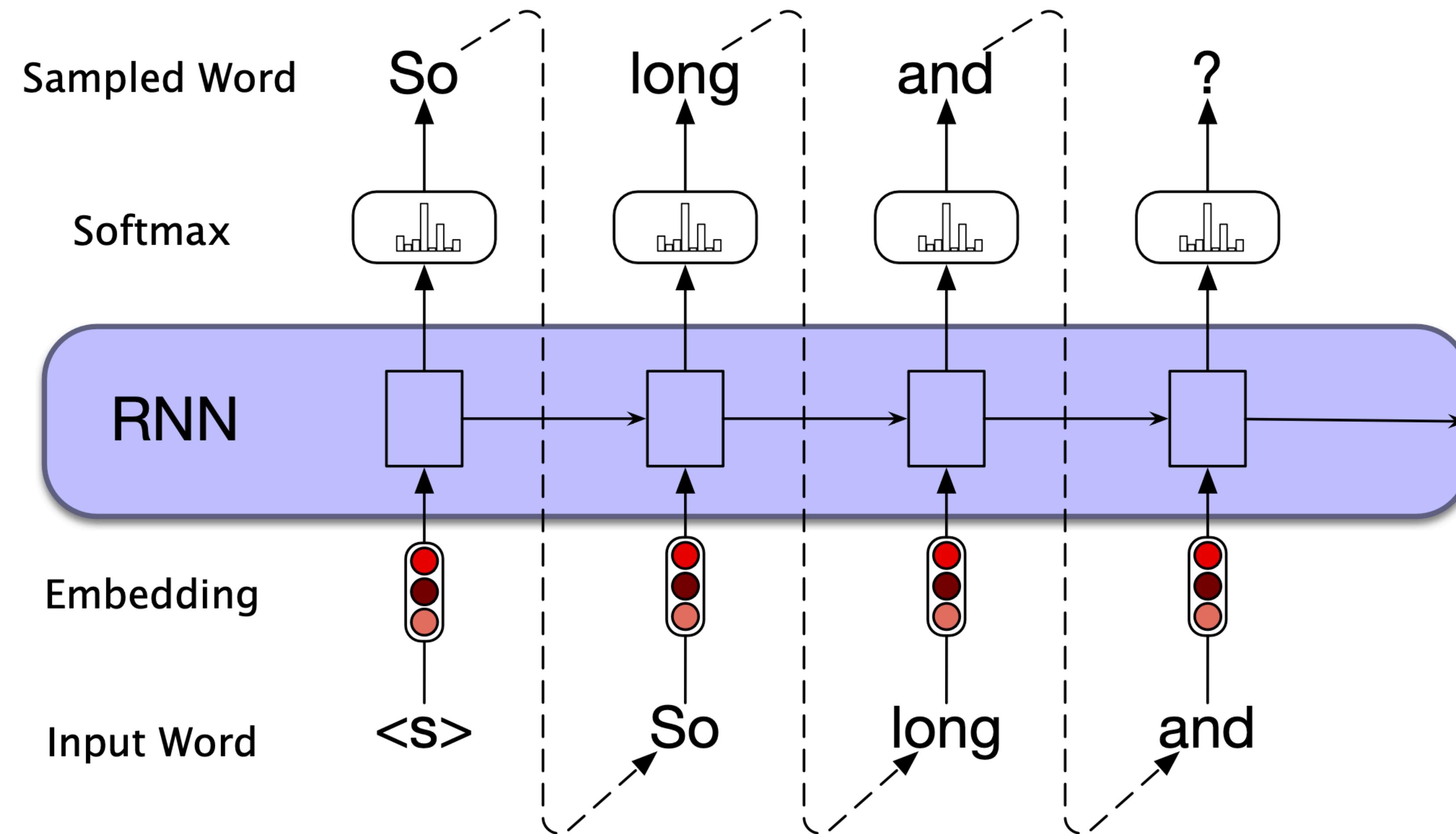
- ▶ measures of **goodness of fit** for observed sequence $w_{1:n}$:
 - **perplexity**:
$$\text{PP}_{LM}(w_{1:n}) = P_{LM}(w_{1:n})^{-\frac{1}{n}}$$
 - **average surprisal**:

$$\text{Avg-Surprisal}_{LM}(w_{1:n}) = -\frac{1}{n} \log P_{LM}(w_{1:n})$$

$$\begin{aligned}\log \text{PP}_M(w_{1:n}) &= \\ \text{Avg-Surprisal}_M(w_{1:n})\end{aligned}$$

Autoregressive generation

left-to-right / causal model



Predictions from different decoding schemes

based on next-token probability $P(w_{i+1} | w_{1:i})$

▶ pure sampling

- next token is sampled from NTP distribution: $w_{i+1} \sim P(\cdot | w_{1:i})$

▶ greedy decoding

- next token is the one with the highest NTP: $w_{i+1} = \arg \max_{w'} P(w' | w_{1:i})$

▶ softmax sampling

- next token is sampled from softmax of NTP distribution:

$$w_{i+1} \sim \text{SM}_\alpha(P(\cdot | w_{1:i}))$$

▶ top-k sampling

- next token is sampled from NTP distribution after restricting to the k most likely words

▶ top-p sampling

- next token is sampled from NTP distribution after restricting to the smallest set of the most likely tokens which together comprise at least NTP p

▶ beam search

- frequently use, if relevant we will cover it later

Model: gpt-3.5-turbo

Temperature: 1

Maximum length: 256

Stop sequences: Enter sequence and press Tab

Top P: 1

Frequency penalty: 0

Presence penalty: 0

from OpenAI's Playground

Training RNNs

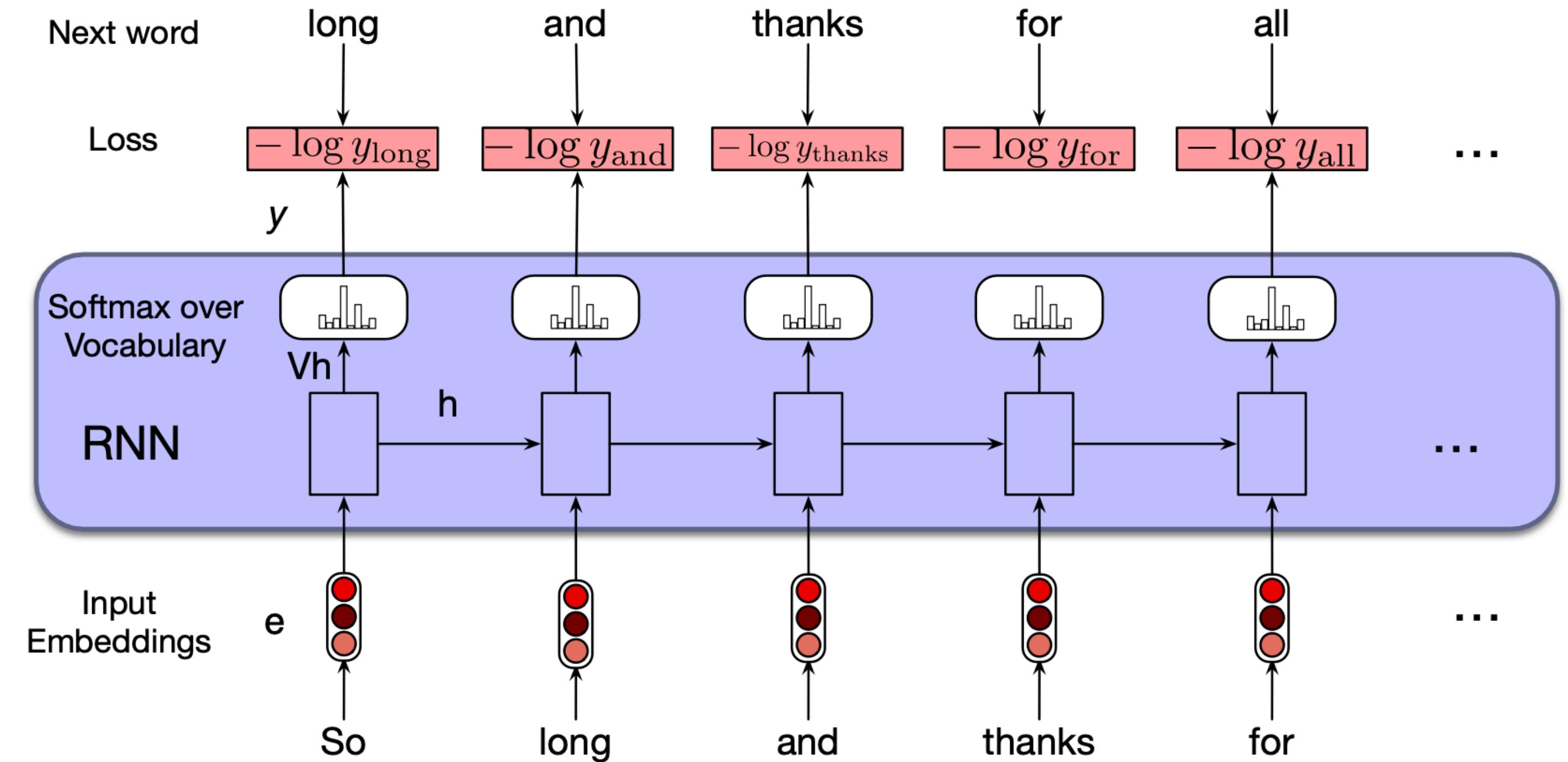
using teacher forcing & next-word surprisal

▶ teacher forcing

- predict each next token given the preceding input (not the model-generated sequence)

▶ next-word surprisal

- loss function is (average) next-token surprisal



Excursion: Different training regimes

- ▶ **teacher forcing**

- LM is fed true word sequence
- training signal is next-word assigned to true word

- ▶ **autoregressive training** (aka free-running mode)

- LM autoregressively generates a sequence
- training signal is next-word probability assigned to true word

- ▶ **curriculum learning** (aka scheduled sampling)

- combine teacher-forced and autoregressive training
- start with mostly teacher forcing, then increase amount of autoregressive training

- ▶ **professor forcing**

- combines teacher forcing with adversarial training
- generative adversarial network GAN is trained to discriminate (autoregressive) predictions from actual data
- LM is trained to minimize this discriminability

- ▶ **decoding-based**

- use prediction function (decoding scheme) to optimize based on *actual* output

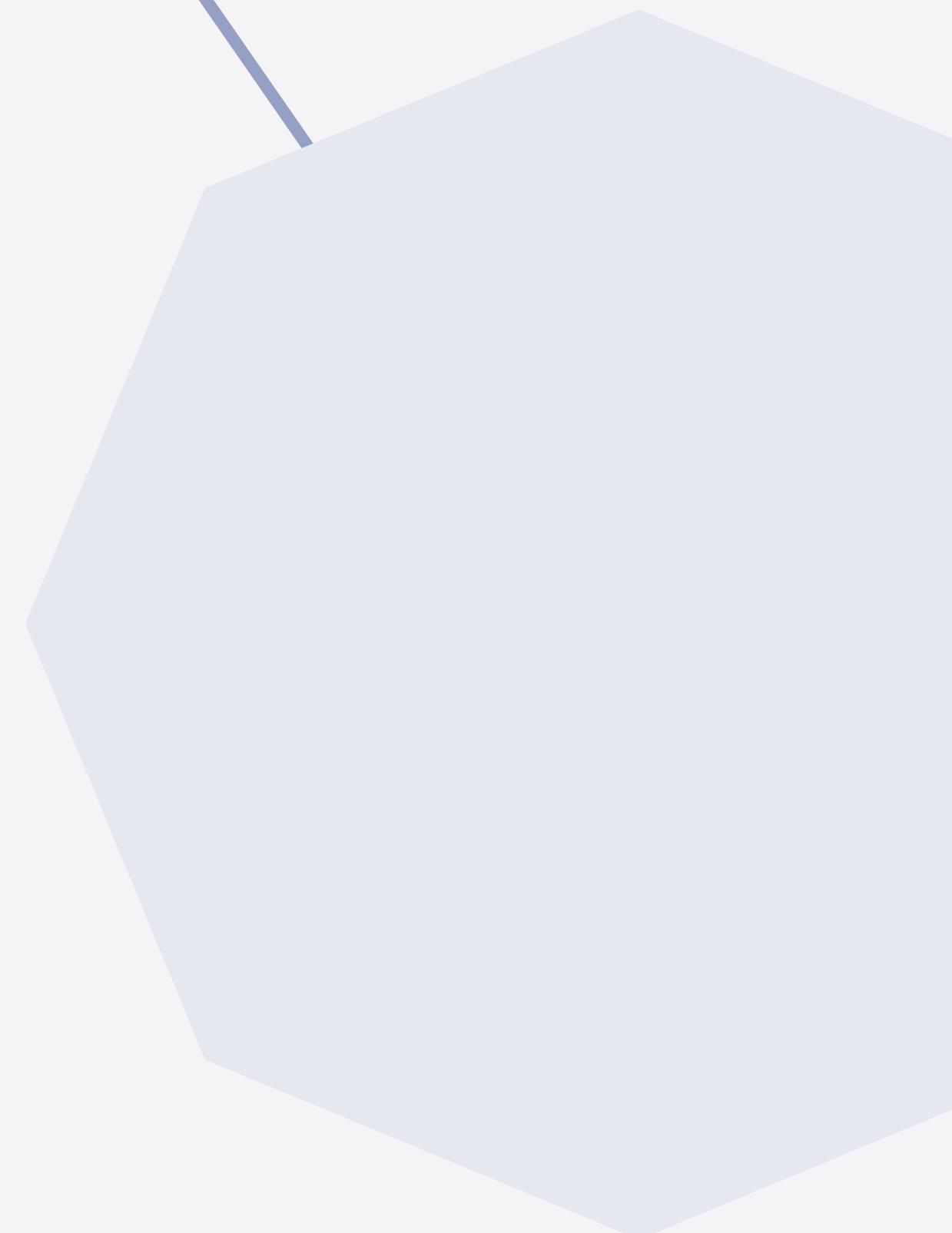
Plain causal LMs in a nutshell

- ▶ **definition**
 - sequence probabilities given by product of next-word probabilities
- ▶ **training**
 - minimize next-word surprisal
- ▶ **prediction**
 - sample auto regressively, using next-word probabilities
- ▶ **evaluation**
 - perplexity or average surprisal
- ▶ **consistent def-train-pred-eval scheme**

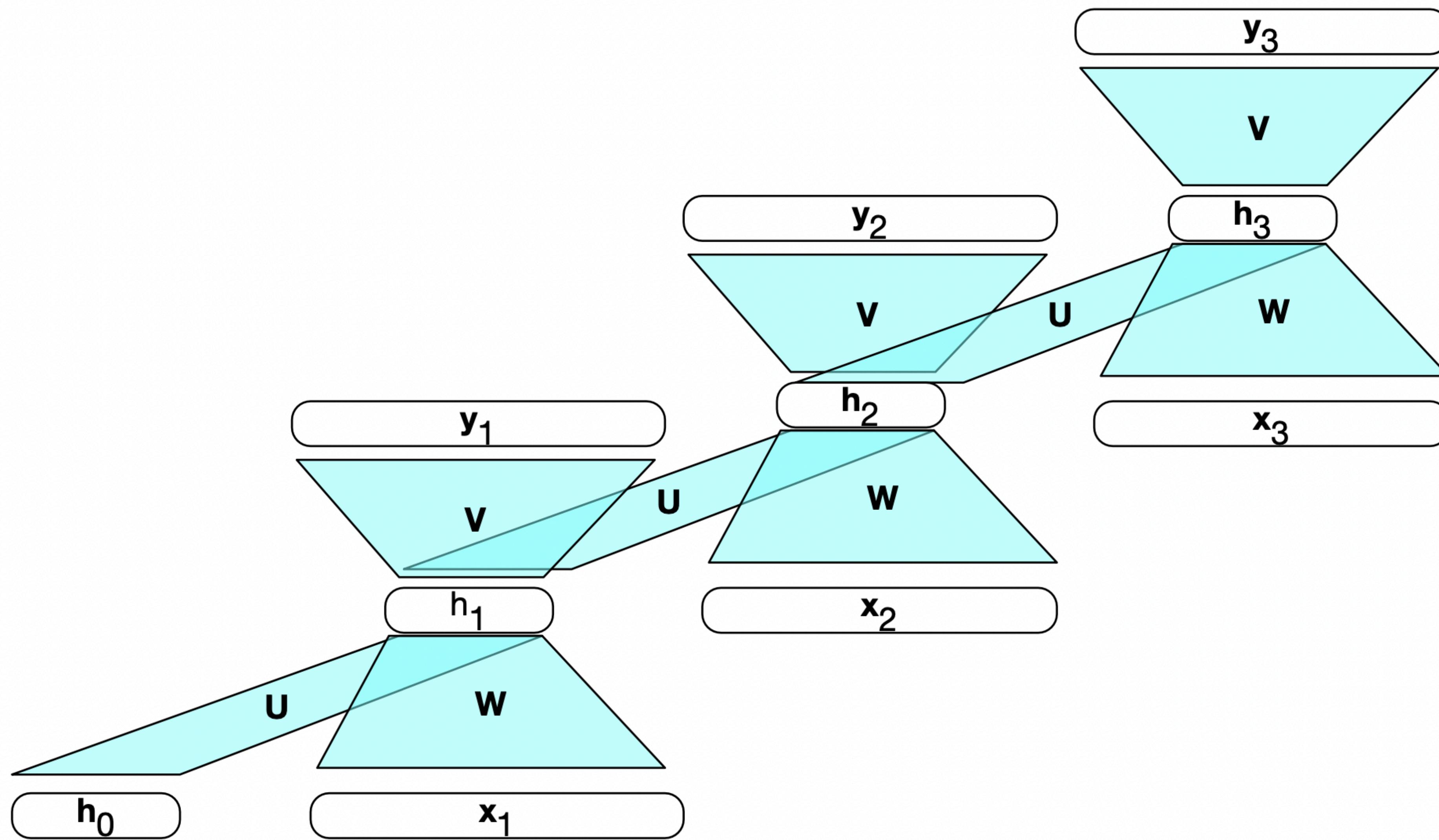
Dirty reality

- ▶ **definition**
 - usually only implicit, often unclear
 - task-dependent
- ▶ **training**
 - usually based on next-word surprisal
 - other (mixed) **training regimes** exist
- ▶ **prediction**
 - whole battery of **decoding strategies**
- ▶ **evaluation**
 - baseline: perplexity or average surprisal
 - additional measure of text quality
- ▶ **possibly inconsistent**

Recurrent Neural Networks



Recurrent neural networks



RNN-based language model

one of many similar architectures

► dimensions:

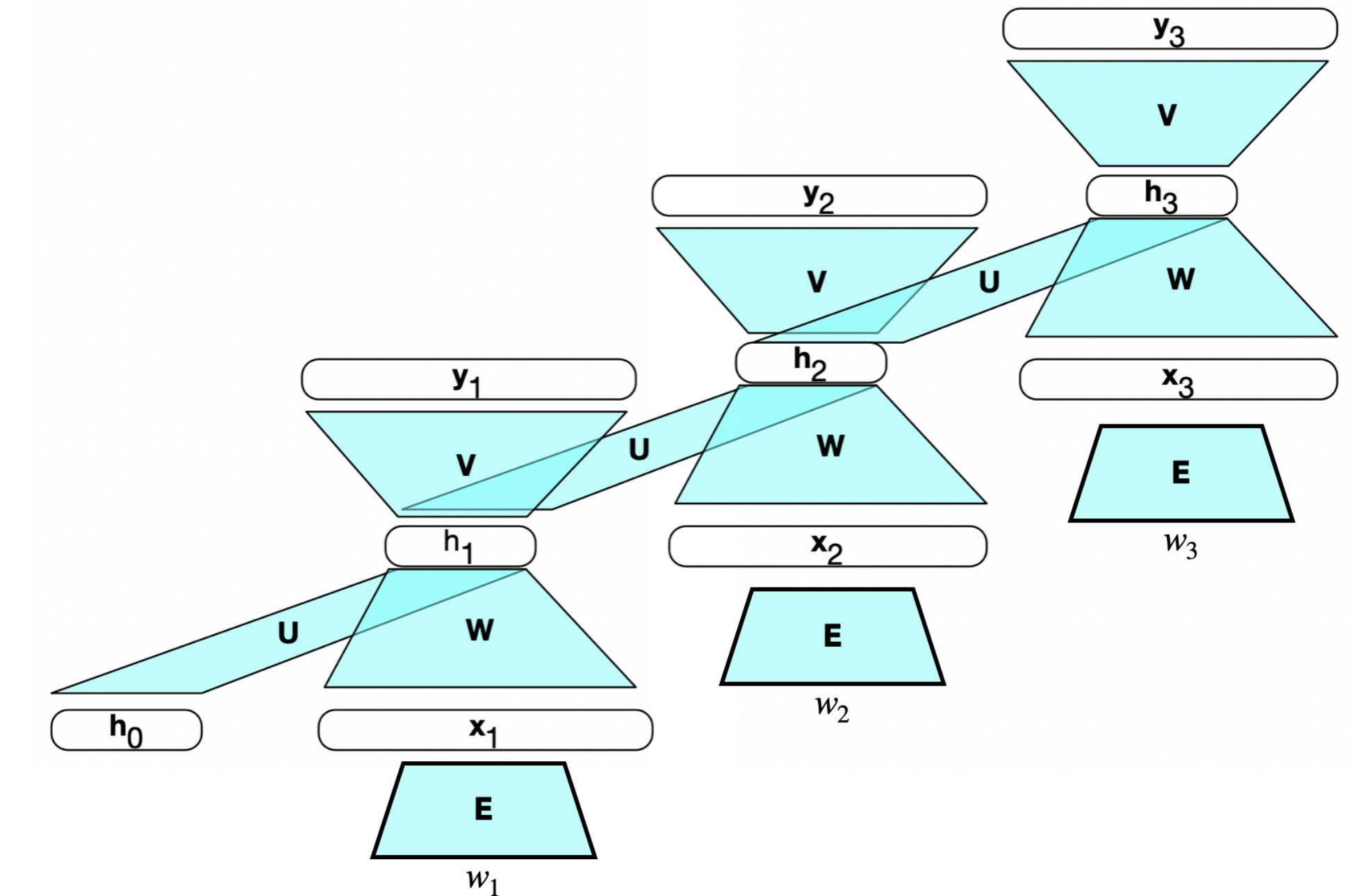
- n_V : # of tokens in vocabulary
- n_h : # units in hidden layer
- n_x : length of input \mathbf{x} (token embedding)

► what is what?

- $\mathbf{w}_t \in \mathbb{R}^{n_V}$: one-hot vector representing token \mathbf{w}_t
- $\mathbf{x}_t \in \mathbb{R}^{n_x}$: word embedding of token \mathbf{w}_t
- $\mathbf{h}_t \in \mathbb{R}^{n_h}$: hidden layer activation at time t (with $\mathbf{h}_0 = 0$)
- $\mathbf{y}_t \in \Delta(\mathcal{V})$: probability distribution over tokens
- $f \in \{\sigma, \tanh, \dots\}$: activation function (as usual)
- $\mathbf{U} \in \mathbb{R}^{n_h \times n_h}$: mapping hidden-to-hidden
- $\mathbf{V} \in \mathbb{R}^{n_V \times n_h}$: mapping hidden-to-word
- $\mathbf{E} \in \mathbb{R}^{n_x \times n_V}$: mapping word-to-embedding
- $\mathbf{W} \in \mathbb{R}^{n_h \times n_x}$: mapping embedding-to-hidden

► definition (forward pass):

- $\mathbf{x}_t = \mathbf{E}\mathbf{w}_t$
- $\mathbf{h}_t = f[\mathbf{U}\mathbf{h}_t + \mathbf{W}\mathbf{x}_t]$
- $\mathbf{y}_t = \text{softmax}(\mathbf{V}\mathbf{h}_t)$



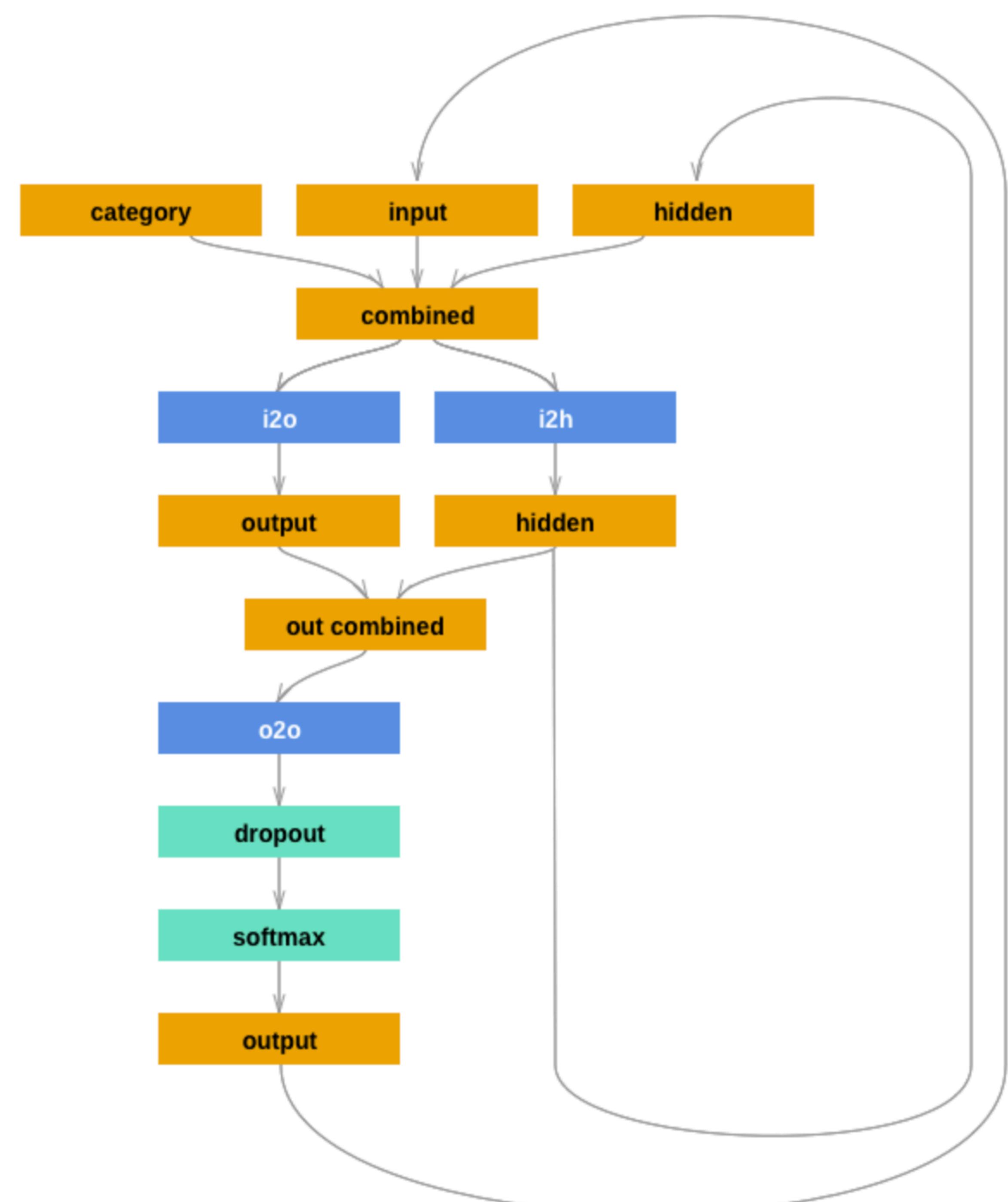
based on Jurafsky & Martin "NLP" book draft

Custom RNN

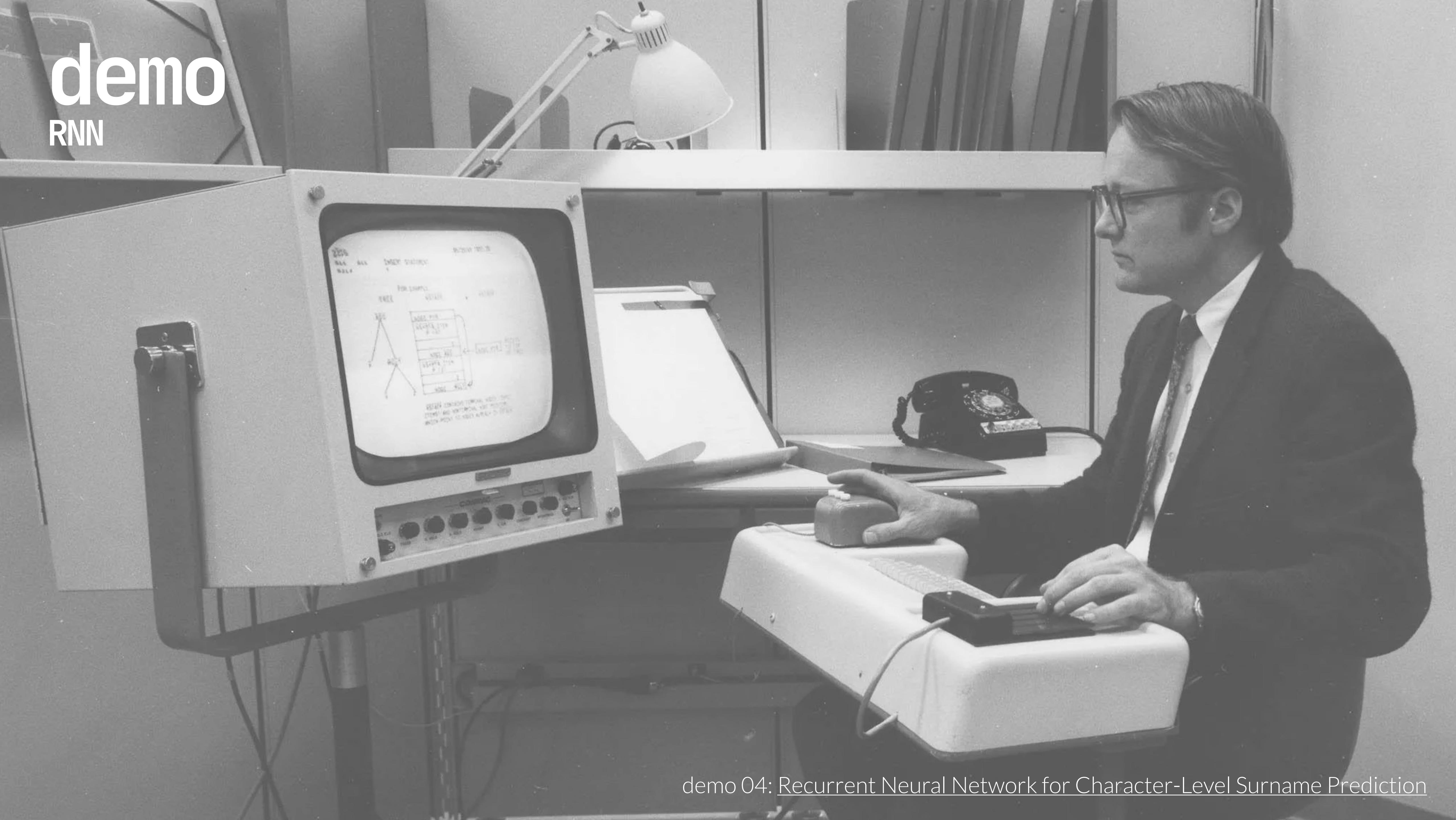
```
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNN, self).__init__()
        self.hidden_size = hidden_size
        self.i2h = nn.Linear(n_categories + input_size + hidden_size,
                            hidden_size)
        self.i2o = nn.Linear(n_categories + input_size + hidden_size,
                            output_size)
        self.o2o = nn.Linear(hidden_size + output_size,
                            output_size)
        self.dropout = nn.Dropout(0.1)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, category, input, hidden):
        input_combined = torch.cat((category, input, hidden), 1)
        hidden = self.i2h(input_combined)
        output = self.i2o(input_combined)
        output_combined = torch.cat((hidden, output), 1)
        output = self.o2o(output_combined)
        output = self.dropout(output)
        output = self.softmax(output)
        return output, hidden

    def initHidden(self):
        return torch.zeros(1, self.hidden_size)
```



demo
RNN



demo 04: Recurrent Neural Network for Character-Level Surname Prediction

Homework for next week

- ▶ solve exercises in worksheets from section 2
 - just for yourself; no submission, no grading
- ▶ ask questions
 - moodle, tutorial, class

