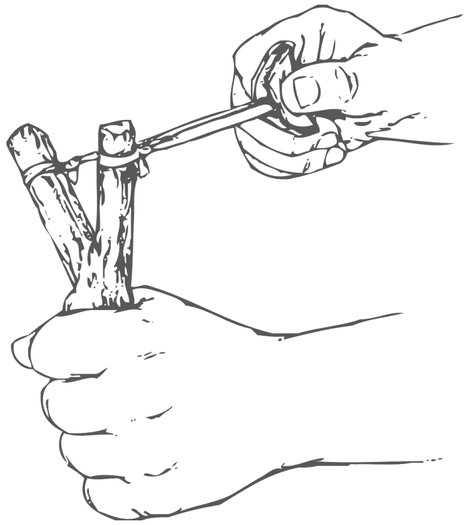
Understanding Large Language Models

Carsten Eickhoff, Michael Franke and Polina Tsvilodub

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

Main learning goals



1. PyTorch

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

2. Optimization (via Backpropagation)

- basic concepts: loss function, gradients, brackpropagation, 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

PyTorch

- 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

```
nTrainingSteps= 10000
for i in range(nTrainingSteps):
    pred =  torch.distributions.Normal(loc=location,
                                       scale=1.0)
    loss = -torch.sum(prediction.log_prob(trainData))
    loss.backward()
    if (i+1) % 500 == 0:
        opt.step()
        opt.zero_grad()
```


Patzke et al. (2019) “PyTorch: An Imperative Style, High-Performance Deep Learning Library”

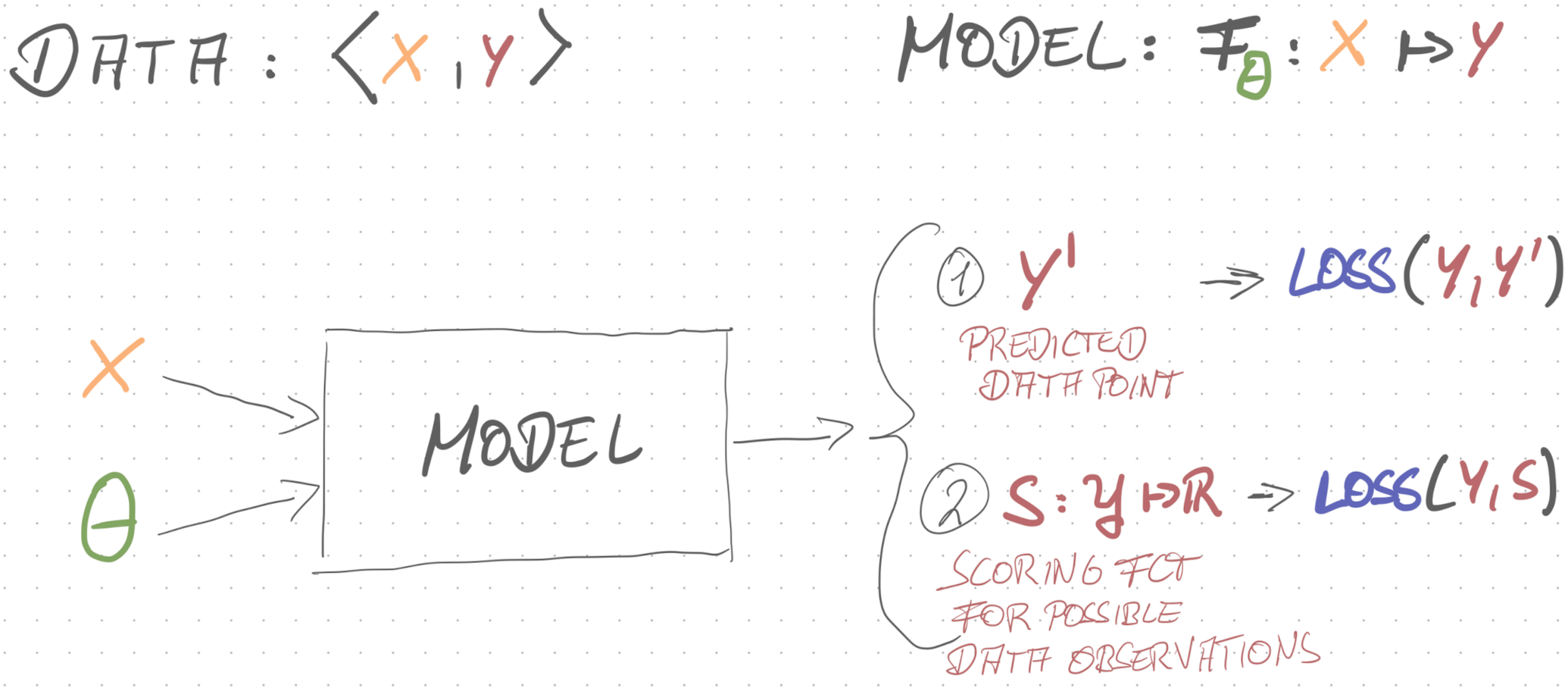
demo
PyTorch

demo 01: PyTorch essentials

Optimization

Models, parameters, predictions & loss

Map x into probability prediction of y



Optimization
for probabilistic models

- given:
  - data $D = \langle X, Y \rangle$
  - probabilistic model $M\colon \Theta, X \to \Delta(Y)$
  - loss function $L\colon \Theta, X, Y \to \mathbb{R}$
    - most commonly used is negative log likelihood:

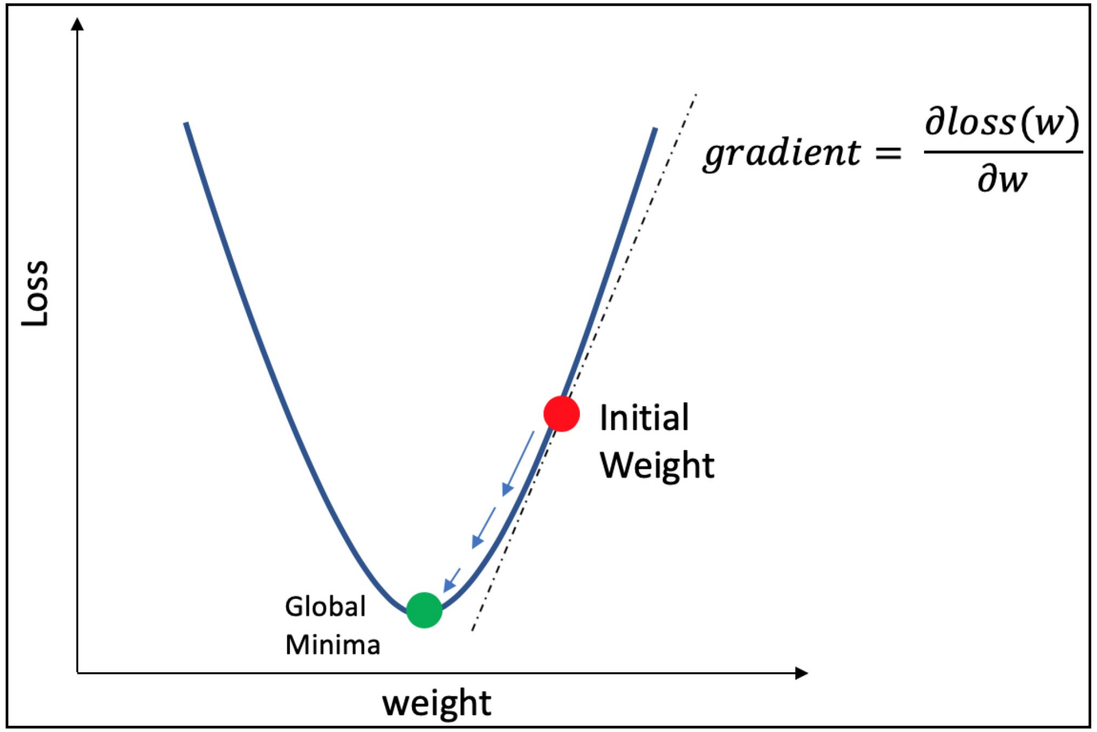
$$L\left(\theta, x, y\right) = -\log P_{M(\theta,x)}\left(y\right)$$

- find parameters that minimize loss for data:

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

From initial red dot we want to go to green dot finally, which is the minima.

We need software to do automatic differentiation, and there PyTorch.



sgb gradient descent (?), a function in pytorch that does the work.

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*

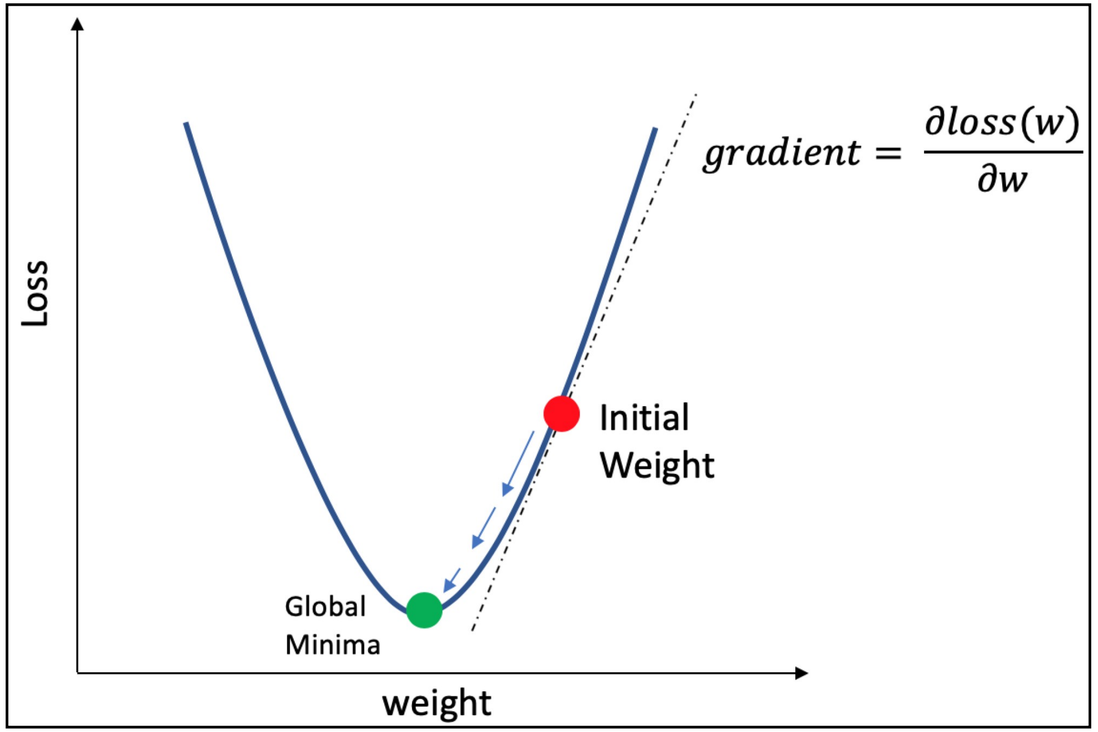
$\theta_t \leftarrow \theta_{t-1} + \gamma g_t$

else

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

return θ_t

from PyTorch Documentation of SGD



from this paper

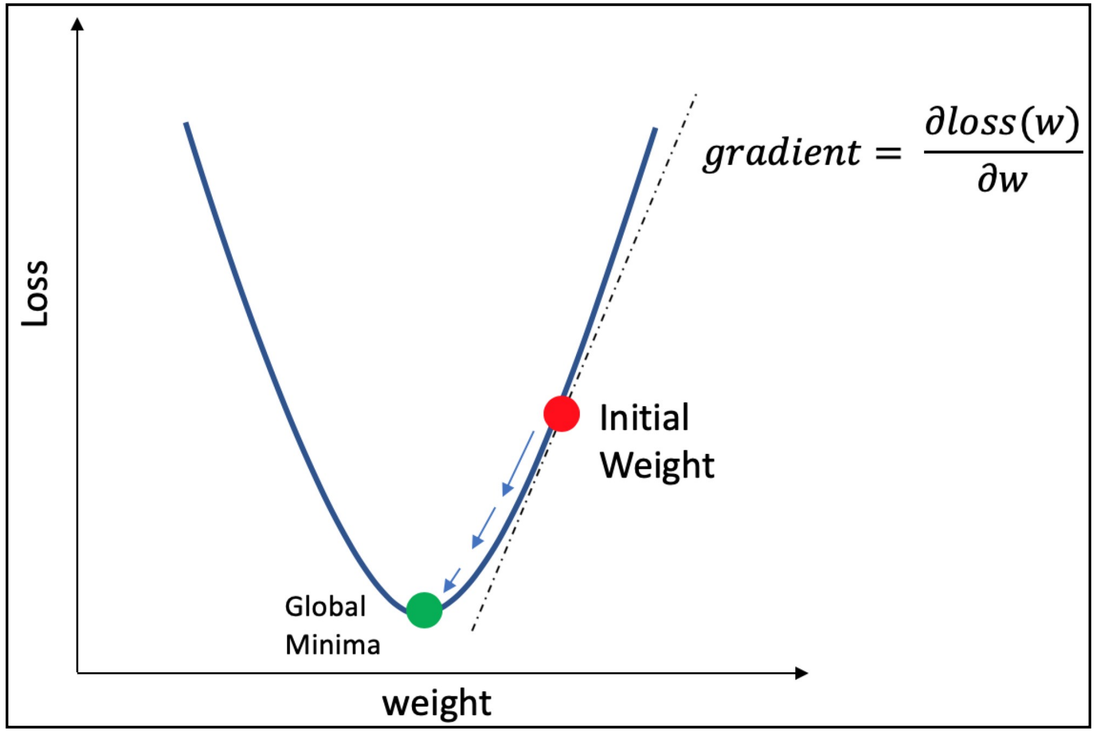
Stochastic gradient descent

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

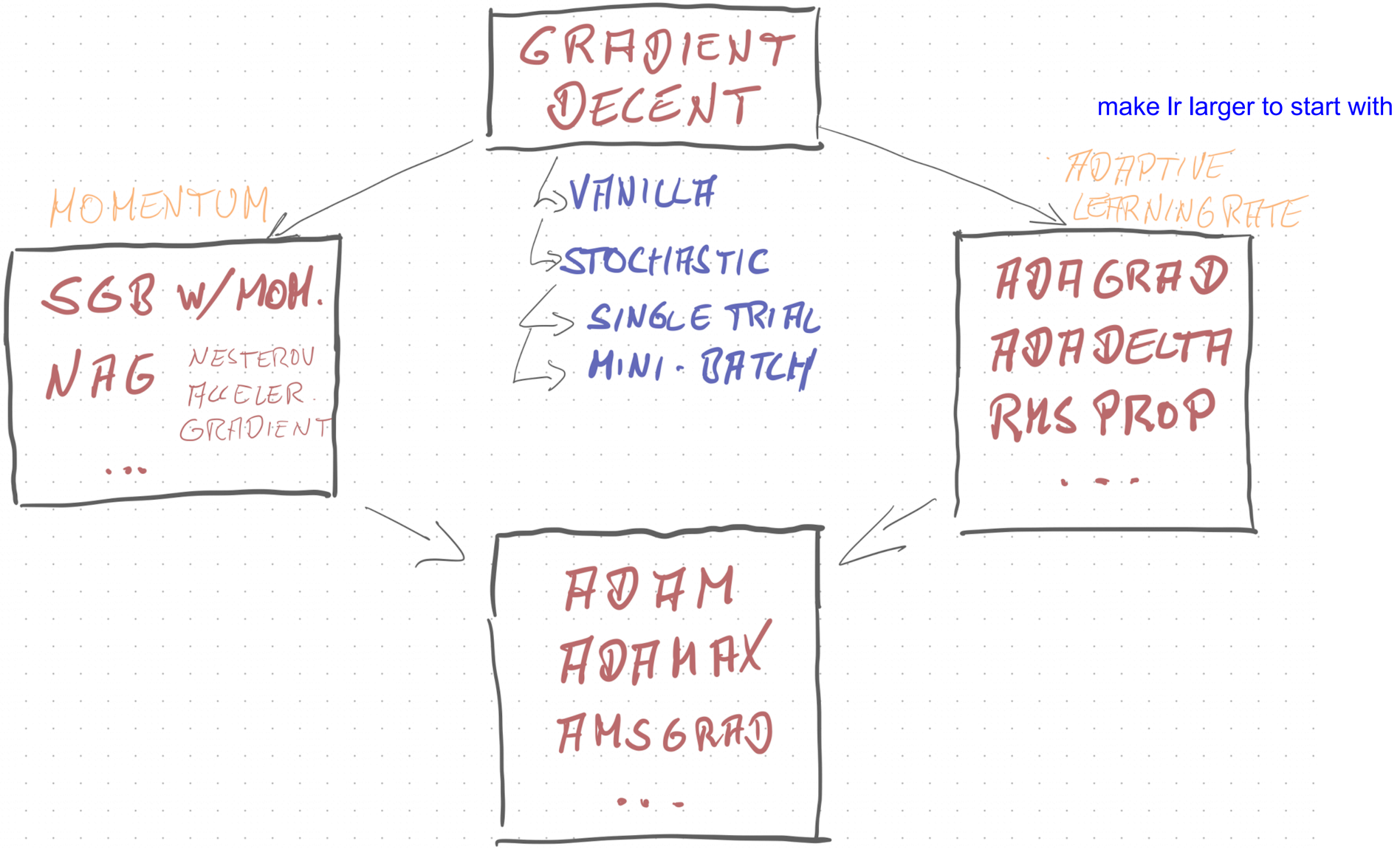
return θ_t

from PyTorch Documentation of SGD



from this paper

Common optimization algorithms



Structure

also learning step

Anatomy of a training step

CSP-Subheading

1. compute predictions

take the para as it is and based on current para to make predictions, confirm the predict or see what is predicted

   - what do we predict in the current state?

2. compute the loss

Tell torch this is what we predicted, this is the loss

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

3. backpropagate the error

An associated info about the gradient

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

4. update the parameters

Based on gradient info, call for optimisation to update

   - 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

```
nTrainingSteps= 10000
for i in range(nTrainingSteps):
    pred =  torch.distributions.Normal(loc=location,
                                       scale=1.0)
    loss = -torch.sum(prediction.log_prob(trainData))
    loss.backward()
    if (i+1) % 500 == 0:
        opt.step()
        opt.zero_grad()
```

demo
optimization

demo 02: Maximum Likelihood Estimation (in PyTorch)

Artificial Neural Networks

Units neurons

- input vector:

  $\mathbf{x} = [x_1, \ldots, x_n]^\intercal$

- weight vector:

  $\mathbf{w} = [w_1, \ldots, w_n]^\intercal$

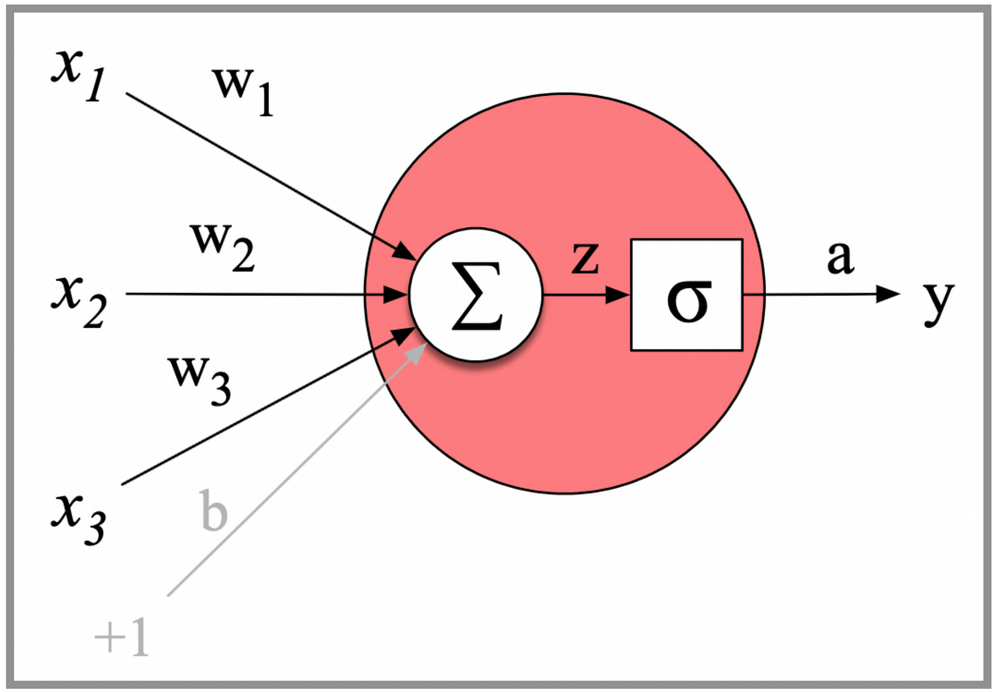
- bias:

  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:

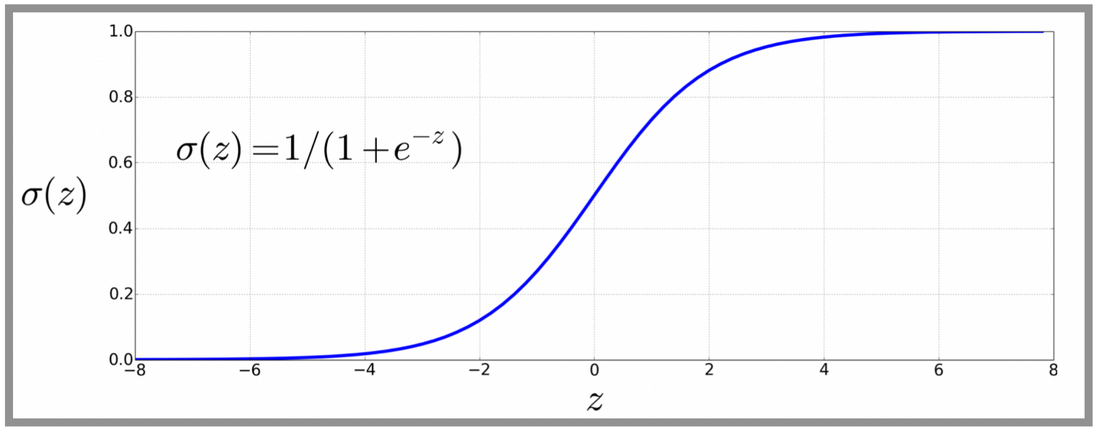
$$f(z) = \sigma(z) = \frac{1}{1 + \exp(-z)}$$

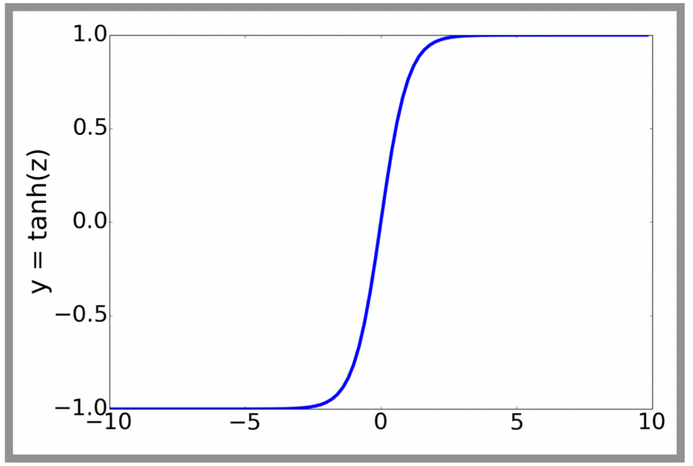
- hyperbolic tangent:

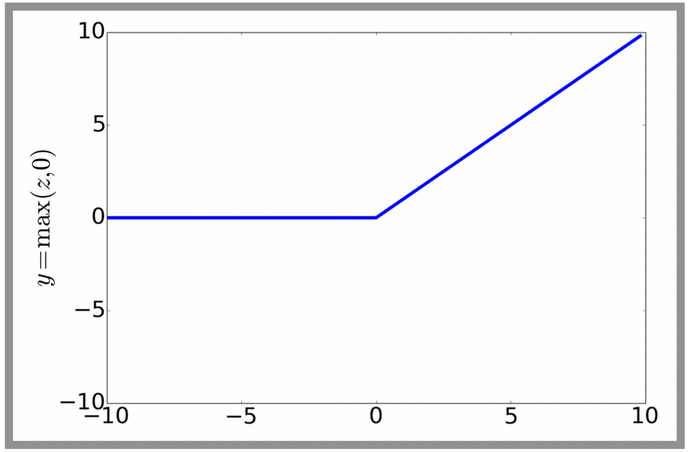
$$f(z) = \tanh(z) = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)}$$

- rectified linear unit:

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







Matrix multiplication

recap

row x column:

A x B –> Arow x Bcolum

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

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

2 x 4 4 x 3 2 x 3

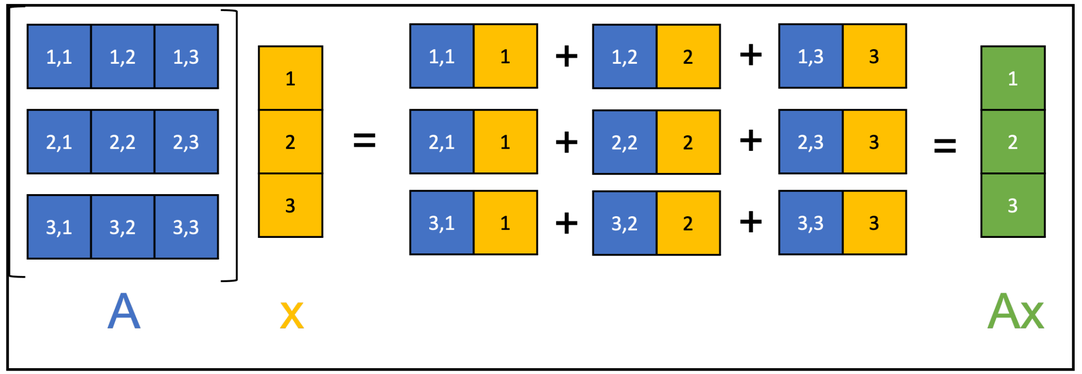
$$c_{22} = a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} + a_{24}b_{42}$$

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

Matrix-vector multiplication

recap

final length corresponding to the columns



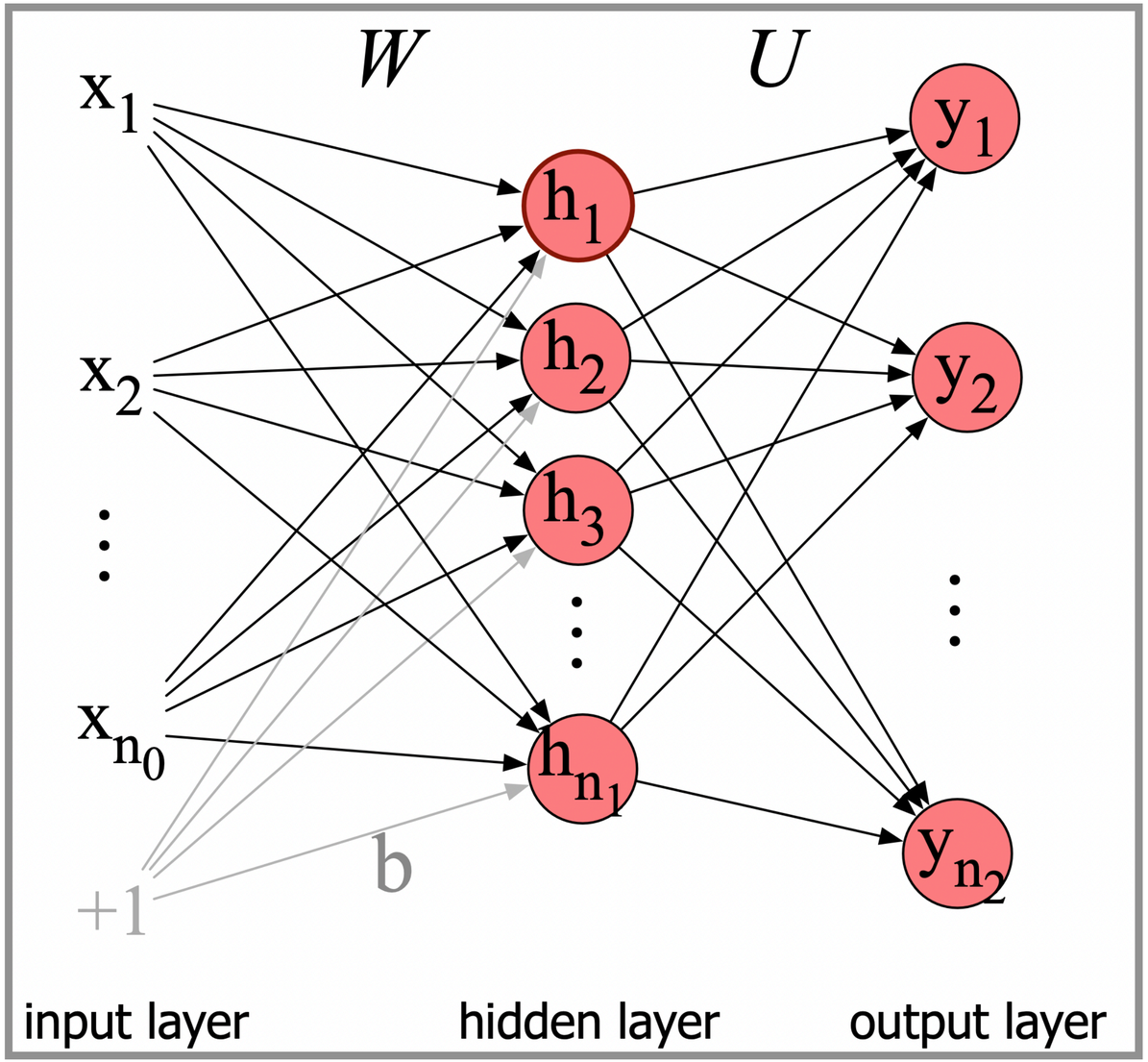
- think of matrix **A** with dimensions (n, m) as a **linear mapping** $f_{\mathbf{A}} : \mathbb{R}^n \rightarrow \mathbb{R}^m$ from vectors of length m to vectors for length n, so that with $\mathbf{x} = [x_1, \ldots, x_m]$:

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

Feed-forward neural network

one layer

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



Feed-forward neural network

multi-layer perceptron

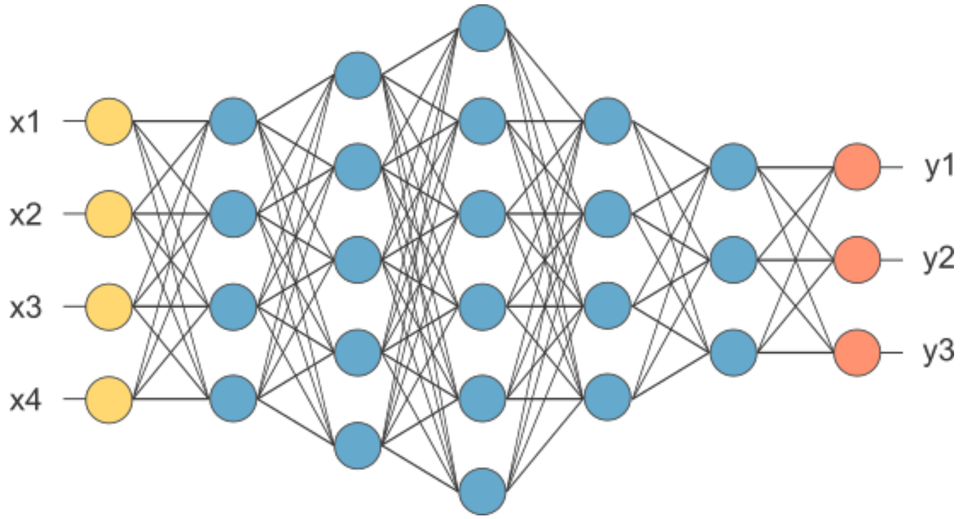
- anchoring in input:

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

- activation at layer n:

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

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



forward: take input, calculate output. Important to get the function, have the nn make preditions
Bw: from the loss,

demo
MLP

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

Language Models

Language model

sequence prediction model

high-level definition

- let $\mathcal{V}$ be a (finite) **vocabulary**, a set of tokens
  - tokens can be characters, sub-words, phrases, …
- let $w_{1:n} = \langle w_1, \ldots, w_n \rangle$ be a finite sequence of tokens
  - it is still common to use w (reminiscent of "words") for tokens
- let S be a 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 $\theta \in \Theta$:

S for sequence

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

  - 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

or better wording: left-to-right LM, as not so related to causation

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

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

  - we write $P_{LM}(w_{n+1} \mid w_{1:n})$ for the **next-token probability**
  - the **surprisal** of w_{n+1} after sequence $w_{1:n}$ is $-\log\left(P_{LM}(w_{n+1} \mid w_{1:n})\right)$

Suprisal: the uncertainty/unpredictbility of an event.
Used in NLP to evaluate LM predictbility and the ability to understand texts.
Commonly use PPL to measure this surprisal

- the **sequence probability** follows from the chain rule:

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

- measures of **goodness of fit** for observed sequence $w_{1:n}$:
  - **perplexity**: Goal is to minimise it

$$\mathrm{PP}_{LM}(w_{1:n}) = P_{LM}(w_{1:n})^{-\frac{1}{n}}$$

  - **average surprisal**:

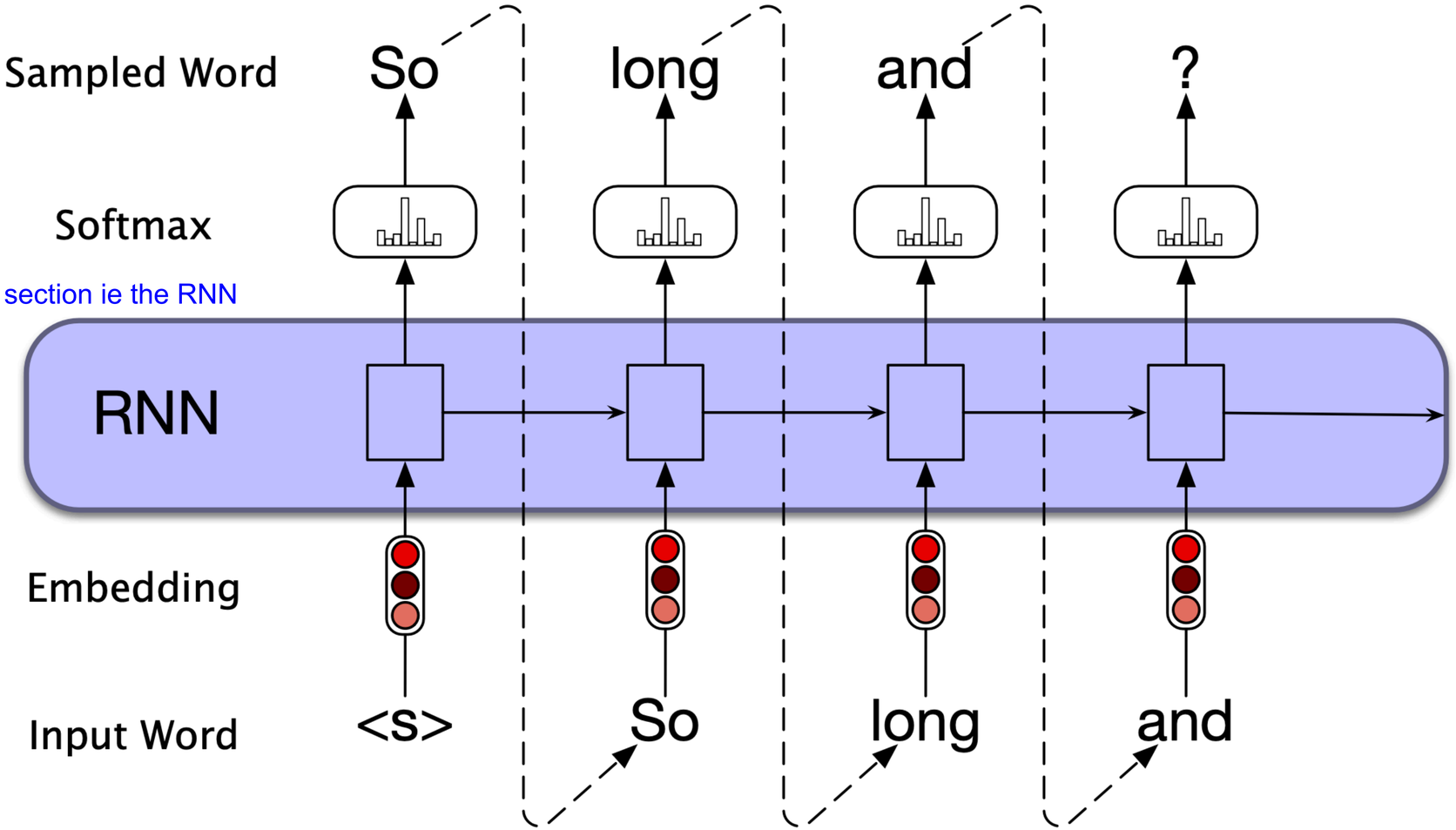
$$\text{Avg-Surprisal}_{LM}(w_{1:n}) = -\frac{1}{n}\log P_{LM}(w_{1:n})$$

$$\log \mathrm{PP}_{M}(w_{1:n}) = \text{Avg-Surprisal}_{M}(w_{1:n})$$

Autoregressive generation

left-to-right / causal model

Predict given previous sequences.



how to train paras in the blank section ie the RNN

Negative log likelihood (NLL)?

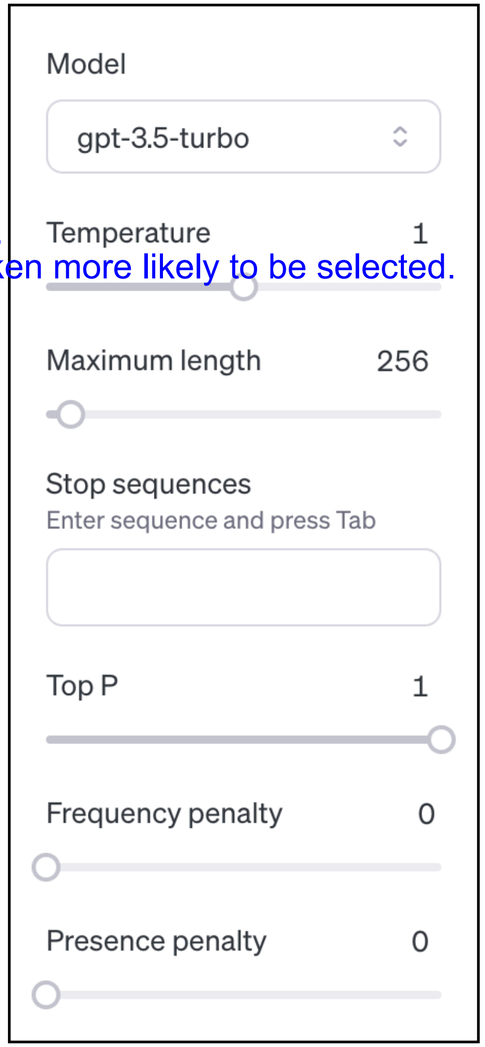
Language model objectives

Predictions from different decoding schemes

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

- **pure sampling** NTP: next token prob
  - next token is sampled from NTP distribution: $w_{i+1} \sim P(\cdot \mid w_{1:i})$
- **greedy decoding** only sample the best word temperature to 0
  - next token is the one with the highest NTP: $w_{i+1} = \arg\max_{w'} P(w' \mid w_{1:i})$
- **softmax sampling**
  - next token is sampled from softmax of NTP distribution: $w_{i+1} \sim SM_{\alpha}\left(P(\cdot \mid w_{1:i})\right)$
- **top-k sampling** only sample k best words, but might not work well when the best is less than k
  - next token is sampled from NTP distribution after restricting to the **k** most likely words
- **top-p sampling** So try top-p
  - 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

pd stays original.
>1 higher prb token more likely to be selected.
< 1 lower.

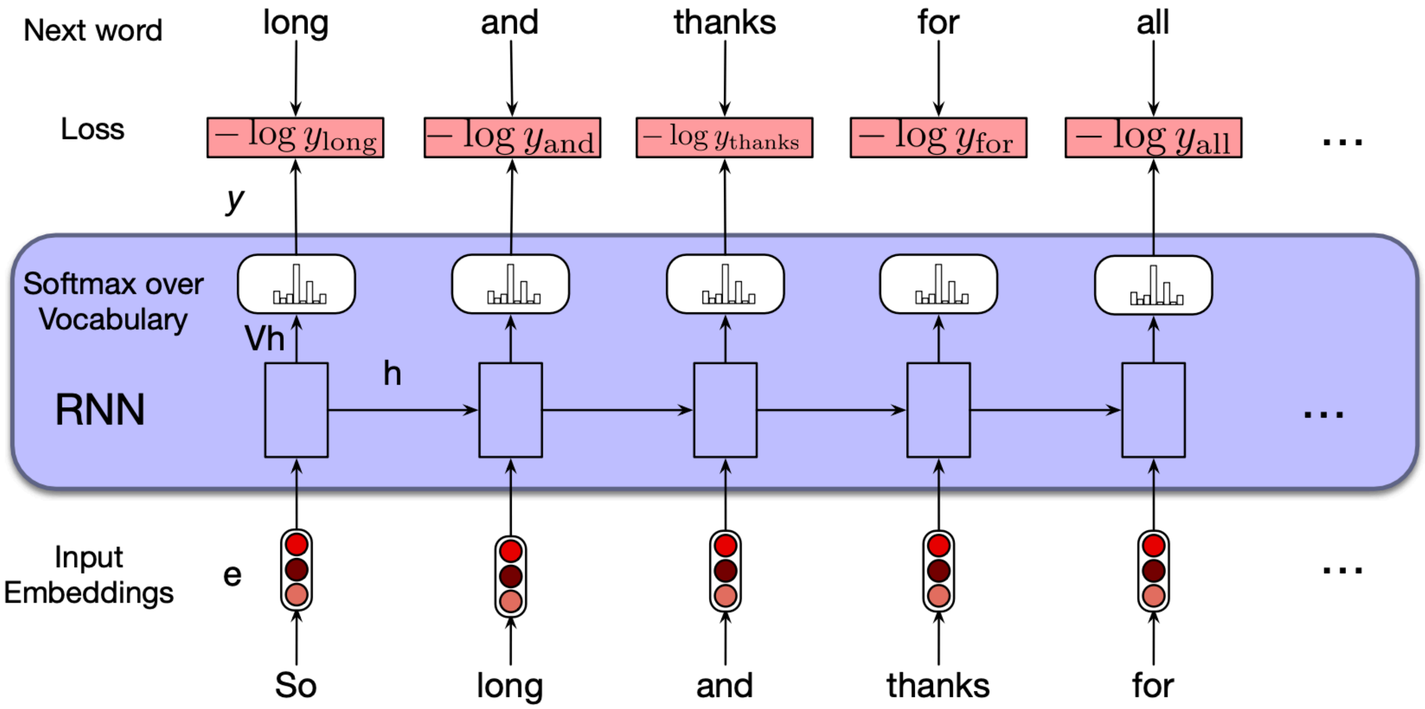


from OpenAI's Playground

Training RNNs

using teacher forcing & next-word surprisal

- teacher forcing giving the model the whole sequence up to the point and ask it to predict ntp.
  - predict each next token given the preceding input (not the model-generated sequence)
- next-work surprisal
  - loss function is (average) next-token surprisal



Target: So long and thanks
Labels: So long and thanks [EOS]
input: [BOS] So long and thanks

In practice, tell the beginning, the BOS
tell the model when see this BOS, to predict 'so'

pass through transformation softmax, log esp. not just raw
We want to have a log prob of 0, log(1) = 0.
argmin - logP(x)

More often used is the cross-entropy loss.
just use the scores and feed into loss function instead of applying log softmax

P is the target, Q is the predicted

Shift labels and inputs for 1.

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

Plain causal LMs in a nutshel

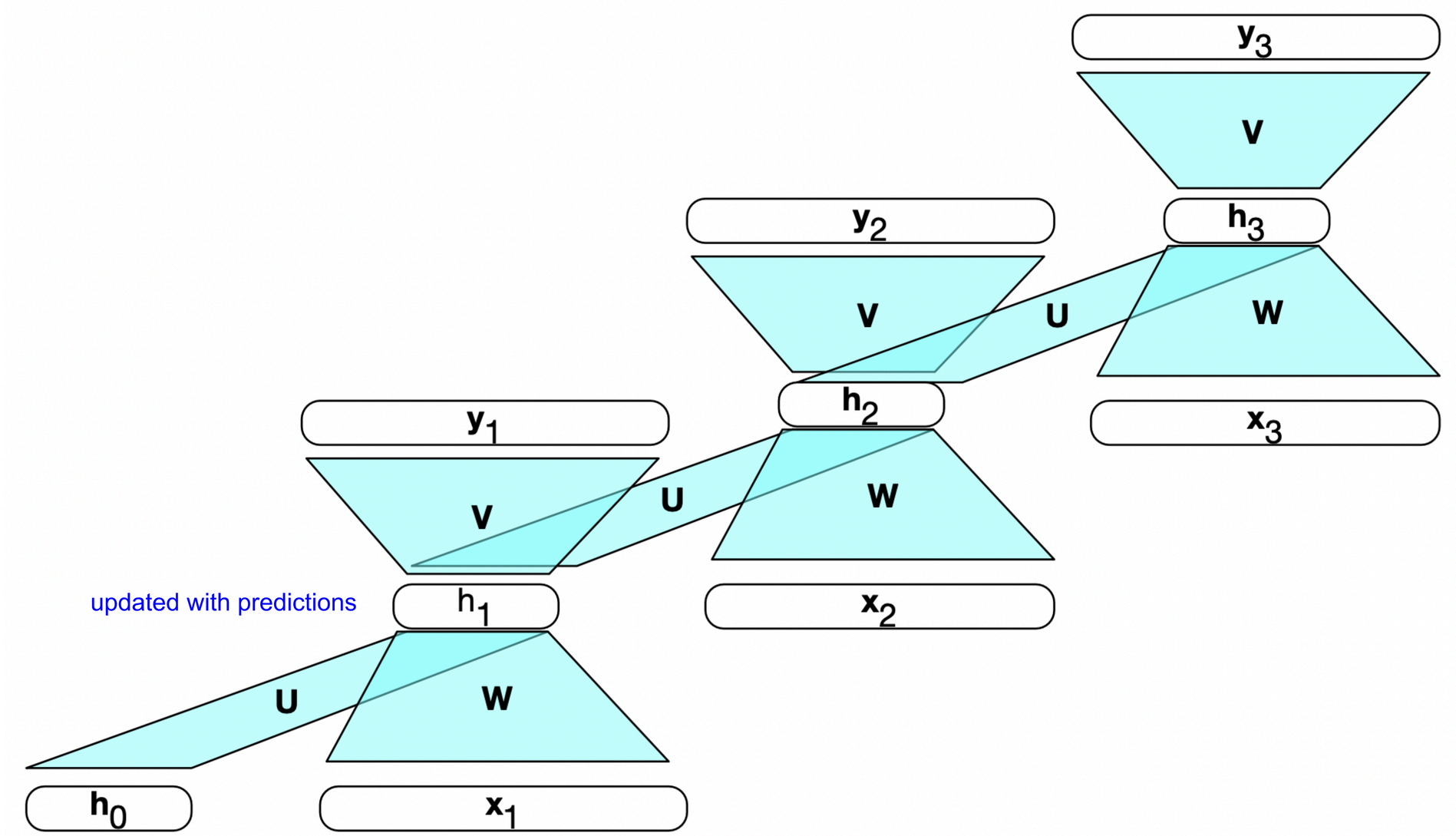
- **definition**
  - sequence probabilities given by product of next-word probabilities
- **training**
  - minimize next-word surprisal
- **prediction**
  - sample auto regressively, using next-word probabilities
- **evaluation**
  - perplexity or average surprisal
- **consistent def-train-pred-eval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



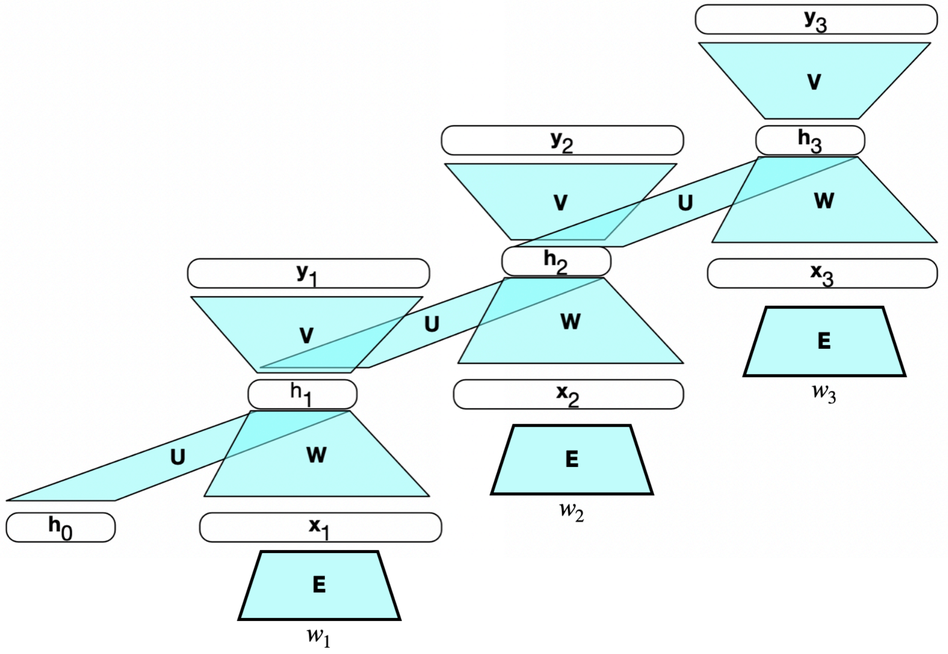
RNN-based language model

one of many similar architectures

RNN, compared to perceptron and such, we have to take care of the hidden units

- 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 $\mathbf{h}_0 = 0$)
  - $\mathbf{y}_t \in \Delta(\mathscr{V})$: probability distribution over tokens
  - $f \in \{\sigma, \texttt{tanh}, \dots\}$: 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):
  - $\mathbf{x}_t = \mathbf{E}\mathbf{w}_t$
  - $\mathbf{h}_t = f\left[\mathbf{U}\mathbf{h}_t + \mathbf{W}\mathbf{x}_t\right]$
  - $\mathbf{y}_t = \texttt{softmax}(\mathbf{V}\mathbf{h}_t)$



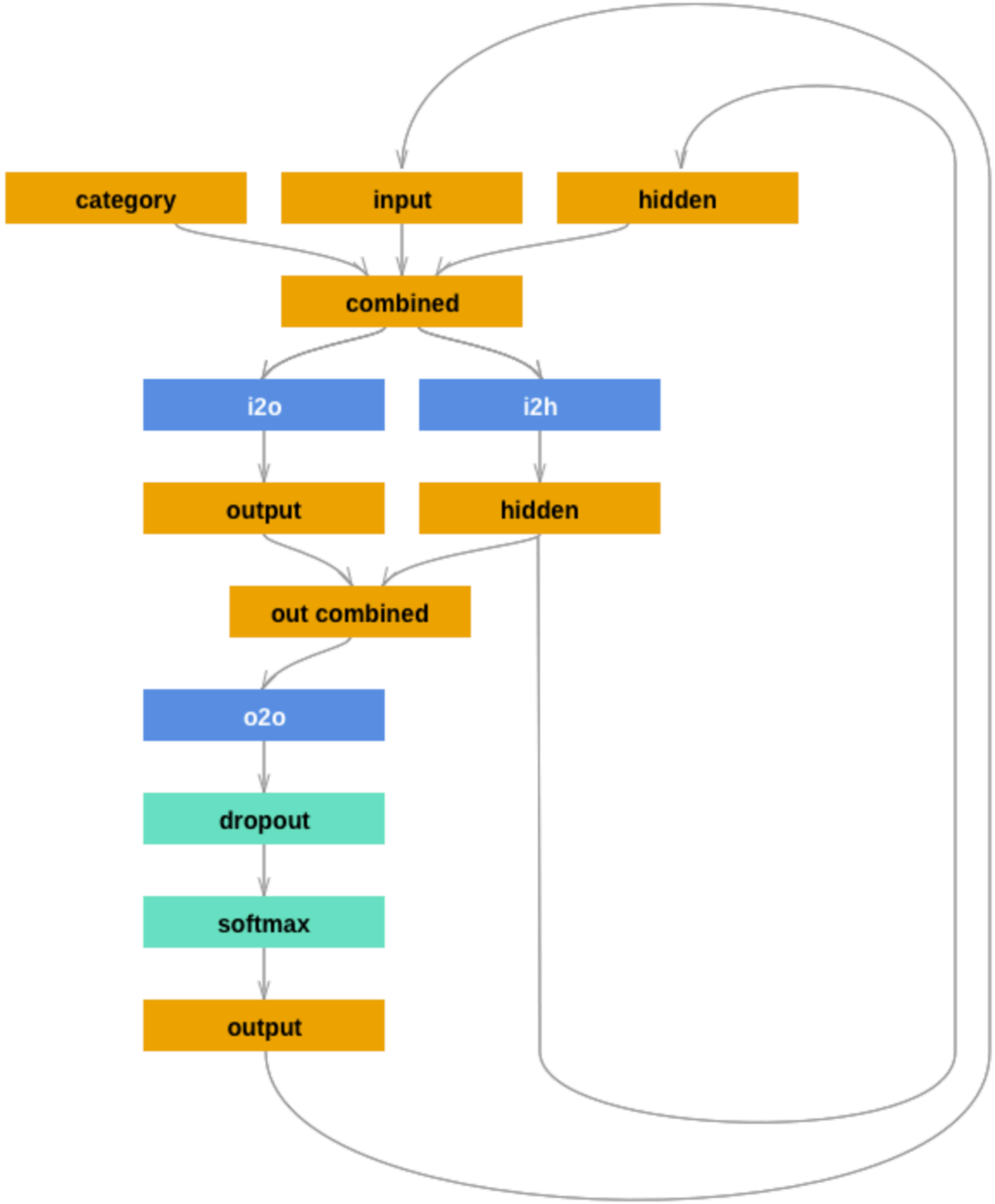
based on Jurafsky & Martin “NLP” book draft

Custom RNN

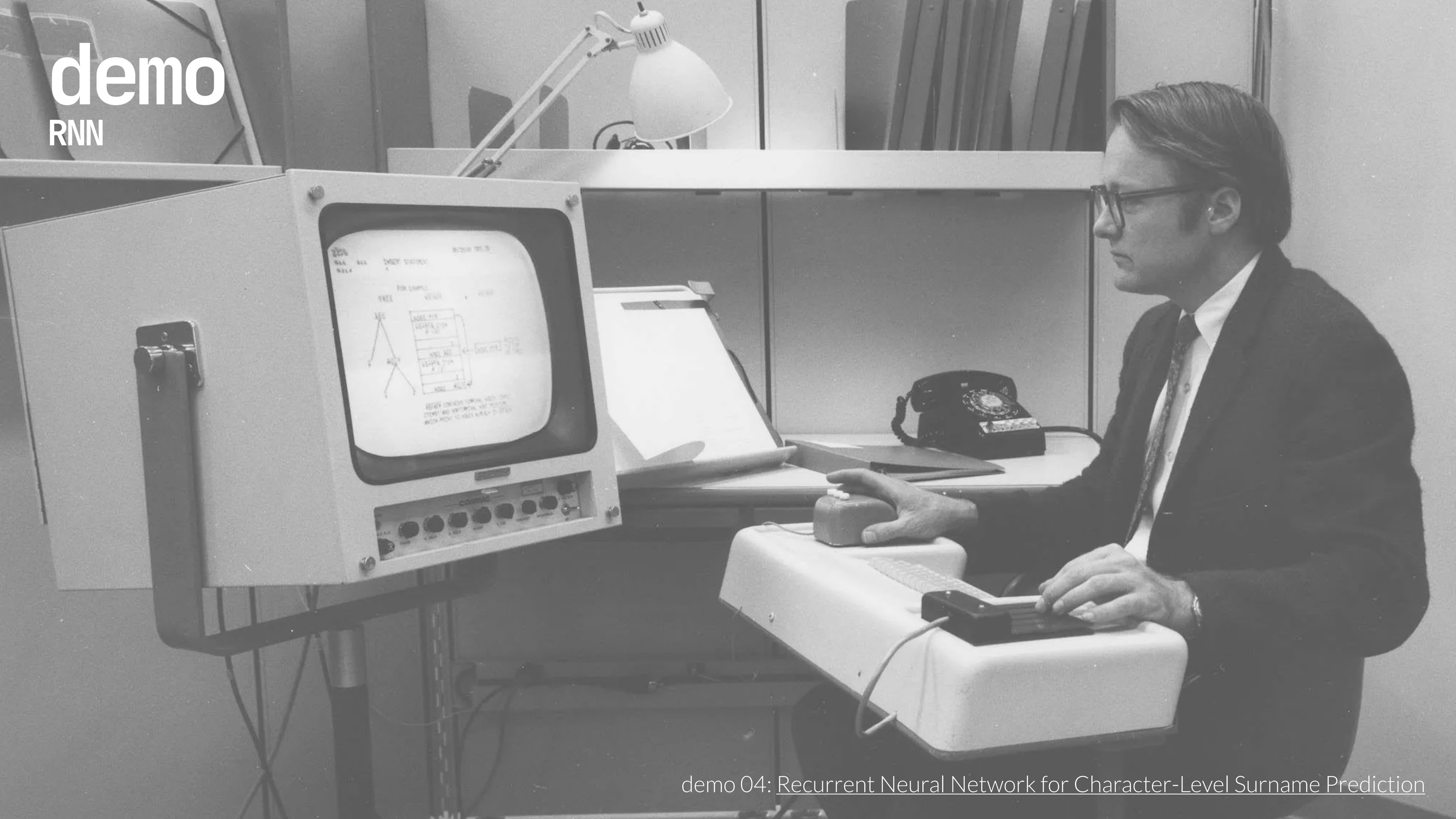
```
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNN, self).__init__()
        self.hidden_size = hidden_size
        self.i2h = nn.Linear(n_categories + input_size + hidden_size,
                             hidden_size)
        self.i2o = nn.Linear(n_categories + 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