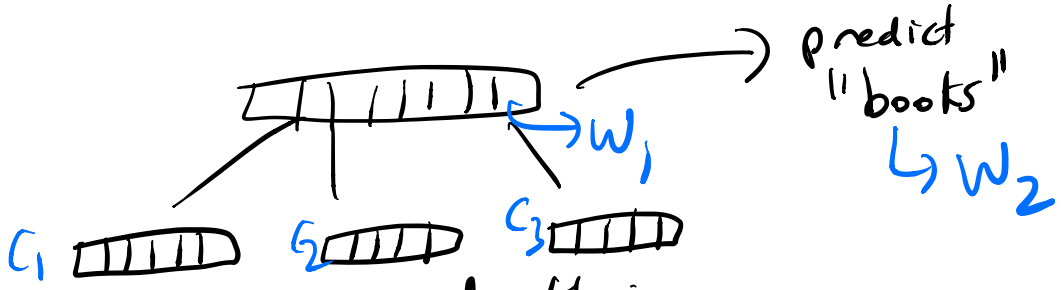
- review of NLMs
- gradient descent
- backpropagation

} train NNs

↳ single neuron

↳ backprop of a linear layer



$$h = f(W_1 [c_1; c_2; c_3])$$

$$o = \text{Softmax}(W_2 h)$$

} h, o are intermediate variables, not parameters

---

we've got our model params, now how do we train them to yield good predictions?

a: gradient descent

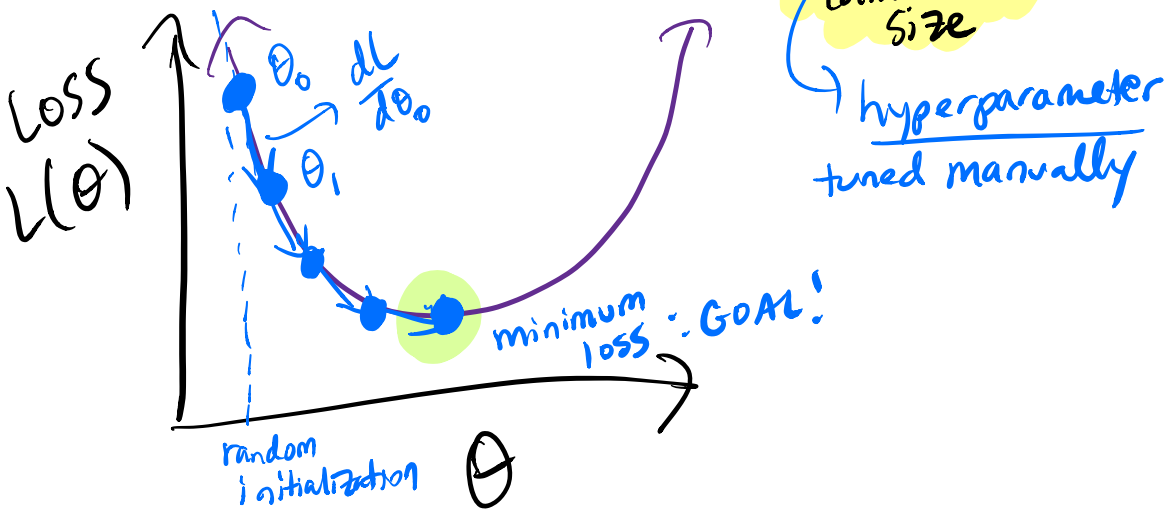
NLM: $W_1, W_2, c, \ldots |V|$

1. define a loss function $L(\theta)$ that measures how bad the current model is at predicting the next word

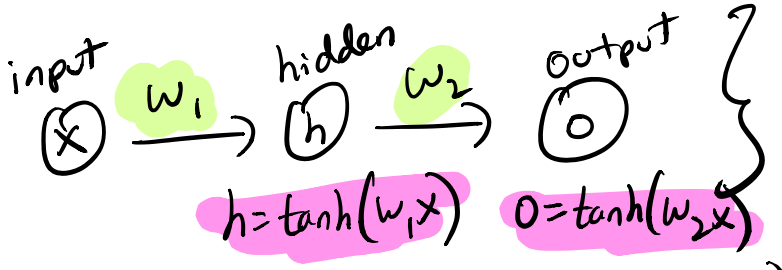
2. calculate the gradient of $L(\theta)$ WRT the model params: $\frac{dL}{d\theta}$
   ↳ the gradient tells us how the loss changes when we modify the model params θ

direction of steepest ascent

3. take a step in the direction of the negative gradient, minimizing the loss: $\theta_{NEW} = \theta_{OLD} - \eta \frac{dL}{d\theta}$
   ↳ learning rate, controls step size
   ↳ hyperparameter tuned manually



Concrete example: (single neuron)



assume our inputs are (x, y) scalar pairs, and we want to predict y from x

STEP 1: forward propagation

↳ compute the value of each neuron h, o using the model eqns

↳ predict the value of y: o

↳ GOAL: make o as close to y as possible

STEP 2: compute loss fn

↳ use square loss in this ex. for simplicity $L = \frac{1}{2}(y - o)^2$ ↳ prediction, ↳ target

STEP 3: compute gradients WRT θ

$$\frac{dL}{dw_1}, \frac{dL}{dw_2}$$

start w/ $\frac{dL}{dw_2}$

chain rule (calculus) $\frac{d}{dx} g(f(x)) = \frac{dg}{df} \cdot \frac{df}{dx}$

↳ partial deriv WRT x

$L = \frac{1}{2}(y-o)^2$

$o = \tanh(w_2 h)$ → intermediate var: let's say $a = w_2 h$

$o = \tanh(a)$

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

$$\frac{dL}{dw_2} = \boxed{\frac{dL}{do}\frac{do}{da}}\frac{da}{dw_2} \Longrightarrow \text{chain rule}$$

$$-(y-o)(1-o^2)\,h$$

$$\frac{dL}{dw_1} = \boxed{\frac{dL}{do}\frac{do}{da}}\frac{da}{dh}\frac{dh}{db}\frac{db}{dw_1}$$

intermediate var $b = w_1 x$

↳ cache from $\frac{dL}{dw_2}$

↳ compute these terms

intuition for backprop:
cache derivatives that appear higher up in the network and use them to compute lower-level derivs.

param update: $W_{1_{NEW}} = W_{1_{OLD}} - \eta \frac{dL}{dW_1}$

$W_{2_{NEW}} = W_{2_{OLD}} - \eta \frac{dL}{dW_2}$

---

single neurons ⇒ multiple neurons
⇒ matrix/vector notation

linear layer: simple component of NNs

$y = f(XW)$

- W: weight matrix
- X: input matrix

- each row of X contains a vector that corresponds to a single training example (a batch)

toy example: $N=2, D=2, M=3$

- N = number of examples (batch size)
- D = dimensionality of embeddings
- M = dimensionality of hiddens

f: identity, $f(x) = x$

$$X = \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix}, \quad W = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix}$$

$$Y = \begin{bmatrix} x_{11}w_{11} + x_{12}w_{21} & x_{11}w_{12} + x_{12}w_{22} & \cdots \\ \vdots & \ddots & x_{21}w_{13} + x_{22}w_{23} \end{bmatrix}$$

let's assume we're given $\frac{dL}{dY}$

we want to compute $\frac{dL}{dX}, \frac{dL}{dW}$

$$\frac{dL}{dx} = \frac{dL}{dY}\frac{dY}{dx}$$

→ Jacobian matrix

$(N \times M) \times (N \times D)$

We can compute $\frac{dL}{dx}$ w/o forming $\frac{dY}{dx}$

$$\frac{dL}{dx} = \begin{bmatrix} \frac{dL}{dx_{11}} & \frac{dL}{dx_{12}} \\ \frac{dL}{dx_{21}} & \frac{dL}{dx_{22}} \end{bmatrix}$$

$$\frac{dL}{dx_{11}} = \frac{dL}{dy} \cdot \frac{dy}{dx_{11}} \rightarrow \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ 0 & 0 & 0 \end{bmatrix}$$

$$\frac{dL}{dx_{12}} = \frac{dL}{dy} \cdot \frac{dy}{dx_{12}} \rightarrow \begin{bmatrix} w_{21} & w_{22} & w_{23} \\ 0 & 0 & 0 \end{bmatrix}$$

$$\frac{dL}{dx_{21}} = \frac{dL}{dy} \cdot \frac{dy}{dx_{21}} \rightarrow \begin{bmatrix} 0 & 0 & 0 \\ w_{11} & w_{12} & w_{13} \end{bmatrix}$$

$$\frac{dL}{dx_{22}} = \frac{dL}{dy} \cdot \frac{dy}{dx_{22}} \rightarrow \begin{bmatrix} 0 & 0 & 0 \\ w_{21} & w_{22} & w_{23} \end{bmatrix}$$

$$\frac{dL}{dy} = \begin{bmatrix} \frac{dL}{dy_{11}} & \frac{dL}{dy_{12}} & \frac{dL}{dy_{13}} \\ \frac{dL}{dy_{21}} & \frac{dL}{dy_{22}} & \frac{dL}{dy_{23}} \end{bmatrix}$$

$$\frac{dL}{dy} \cdot \frac{dy}{dx_{11}} = \frac{dL}{dy_{11}} w_{11} + \frac{dL}{dy_{12}} w_{12} + \frac{dL}{dy_{13}} w_{13}$$

$$\frac{dL}{dx} = \begin{bmatrix} \frac{dL}{dy_{11}} w_{11} + \frac{dL}{dy_{12}} w_{12} + \frac{dL}{dy_{13}} w_{13} & \frac{dL}{dy_{11}} w_{21} + \frac{dL}{dy_{12}} w_{22} + \frac{dL}{dy_{13}} w_{23} \\ \cdots & \frac{dL}{dy_{21}} w_{21} + \frac{dL}{dy_{22}} w_{22} + \frac{dL}{dy_{23}} w_{23} \end{bmatrix}$$

↳ rewritten as the matrix product

$$\begin{bmatrix} \frac{dL}{dy_{11}} & \frac{dL}{dy_{12}} & \frac{dL}{dy_{13}} \\ \frac{dL}{dy_{21}} & \frac{dL}{dy_{22}} & \frac{dL}{dy_{23}} \end{bmatrix} \begin{bmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \\ w_{13} & w_{23} \end{bmatrix}$$

$$\frac{dL}{dx} = \frac{dL}{dy} \cdot W^T$$

} in practice, we do this instead of computing the Jacobian

$$\frac{dL}{dW} = X^T \frac{dL}{dy}$$

TODAY: → GRAD. DESCENT / BACKPROP

↳ You should be able to implement backprop for a single neuron network

↳ BACKprop:-chain rule
- caching intermediate derivatives
- efficient computations to avoid forming Jacobian