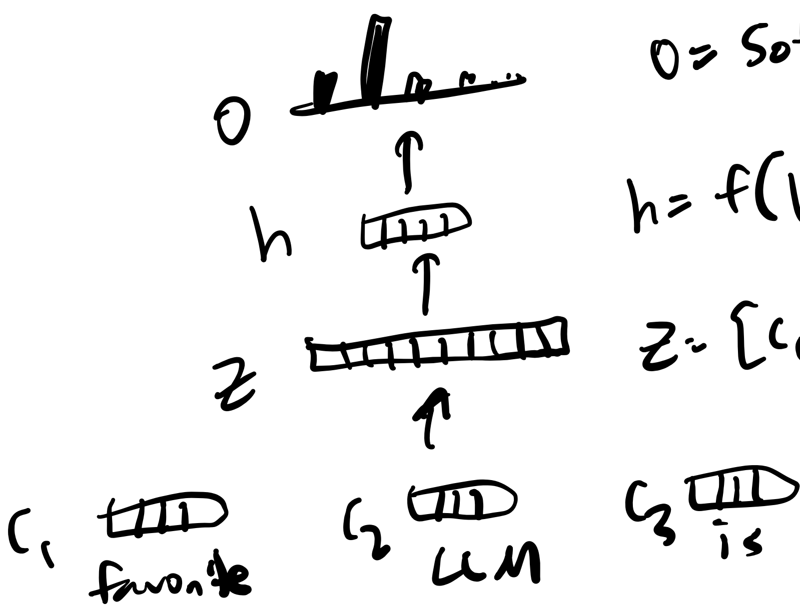
Training neural Language models:

- NLMs contain parameters
  - RNN: $\{W_h, W_c, c_{1,\dots n}, W_o\}$
- all params are randomly initialized
  - $p(w_n \mid w_{1,\dots n-1})$ is also random to start with
- by training the NLM, we adjust its parameters to maximize the likelihood of the training data

---



$o = \text{softmax}(W_o h)$

$h = f(Wz)$

$z = [c_1; c_2; c_3]$

params:

$\theta = \{W, W_o, c_{1 \dots n}\}$

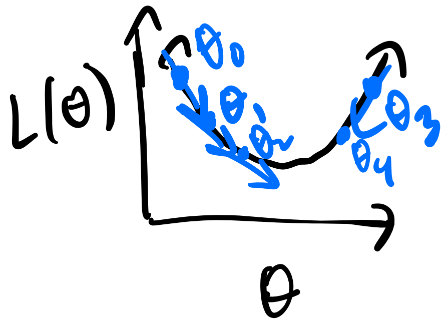
Steps to train NLM:

1. define loss fn $L(\theta)$
   - ↳ tells us how bad model is at predicting next word
   - ↳ smooth, differentiable
2. Given $L(\theta)$, we compute the gradient of L with respect to θ
   - ↳ gradient gives us the direction of steepest ascent
   - ↳ <u>intuition</u>: for each param j in θ, gradient $\frac{dL}{d\theta_j}$ tells us how much L would change if I increase j by a very small amount
3. Given $\frac{dL}{d\theta}$, we take a step in the direction of the negative gradient
   - ↳ minimize L

η = learning rate, hyperparameter "step size"

$$\theta_{new} = \theta_{old} - \eta \frac{dL}{d\theta}$$

← gradient



optimizer

- Stochastic gradient descent (SGD)
- Adam
- Muon

hyperparams of gradient descent:

↳ learning rate η

↳ batch size

↳ how many training examples do we use to estimate $\frac{dL}{d\theta}$ before taking a step

---

Loss function: cross-entropy loss

$\underbrace{\text{favorite LLM is}}_{\text{training prefix}} \Rightarrow \underbrace{\text{ChatGPT}}_{\text{training target}}$, V labels

goal: maximize $p(\text{ChatGPT} \mid \text{"my fav LLM is"})$

↳ minimizing $-\log p(\text{ChatGPT} \mid \ldots)$"

$L = -\log P(\text{ChatGPT} \mid \ldots)$

loss is neg. log prob of correct next word

why "cross entropy loss"?

$$\text{CE loss: } -\sum_{w \in V} \underbrace{p(w)}_{\text{reference}} \underbrace{\log q(w)}_{\text{model prediction}}$$

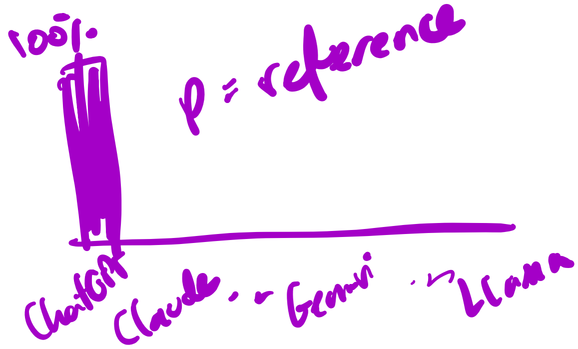
this reduces to neg log likelihood when

$p(w) = 1$ when w is correct next word

$p(w) = 0$ otherwise

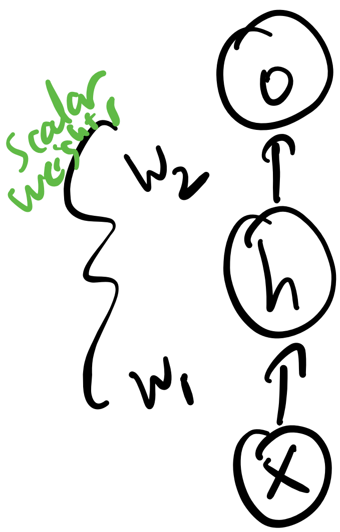
q = model's pred. dist

p = reference



backpropagation: algorithm to compute $\frac{dL}{d\theta}$ in an efficient manner

example w/ scalar inputs/outputs:



output, model's prediction

$o = \tanh(w_2 h)$

hidden

$h = \tanh(w_1 x)$

input

training data:

(x, y) pairs

input → x, target → y

e.g. $(0.3, -0.5)$

what are the params of this network?

$$\theta = \{w_1, w_2\}$$

gradient $$\frac{dL}{d\theta} = \left\{ \frac{dL}{dw_1}, \frac{dL}{dw_2} \right\}$$

Step 1: compute loss L

↳ for this example, instead of NLL, we will use square loss

$$L = \frac{1}{2}(\underbrace{y}_{\text{target}} - \overbrace{o}^{\text{model prediction}})^2$$

Step 2: compute $\frac{dL}{dw_1}$, $\frac{dL}{dw_2}$

↳ we will use the chain rule of calculus

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

↳ we start at the top of the network (i.e. output layer) and work our way down

$$\frac{dL}{dw_2} \left] \begin{aligned} L &= \tfrac{1}{2}(y-o)^2 \\ o &= \tanh(a) \\ a &= w_2 h \end{aligned} \right.$$

useful to define intermediate vars

$a = w_2 h$

$b = w_1 x$

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

$$\frac{dL}{dw_2} = \frac{dL}{do} \cdot \frac{do}{da} \cdot \frac{da}{dw_2}$$

$$\boxed{-(y-o) \cdot (1-o^2) \cdot h}$$

$$\frac{dL}{dw_1}$$

$$L = \frac{1}{2}(y-o)^2$$
$$o = \tanh(a)$$
$$a = w_2 h$$
$$h = \tanh(b)$$
$$b = w_1 x$$

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

$$\frac{dL}{dw_2} = \underbrace{\frac{dL}{do} \cdot \frac{do}{da}}_{} \cdot \frac{da}{dw_2}$$

if we cache this product, we can reuse it when computing $\frac{dL}{dw_1}$

<u>backpropagation</u>: chain rule of calculus + caching prev. computed derivatives

Step 3: update params

$$\theta = \{ w_1, w_2 \}$$

$$w_{1_{new}} = w_{1_{OLD}} - \eta \frac{dL}{dw_1}$$

$$w_{2_{new}} = w_{2_{OLD}} - \eta \frac{dL}{dw_2}$$

Steps 2, 3 super easy in PyTorch:

loss = $-\log P_{model}$(Gemini | favorite LLM is)

loss.backwards() $\Rightarrow$ computes gradient

optimizer.step() $\Rightarrow$ update params