- review of NLMs
- gradient descent
- backpropagation
- single neuron
- backprop of a linear layer

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

\[ h = f(W_1 [C_1; C_2; C_3]) \]
\[ o = \text{Softmax}(W_2 h) \]

\[ \text{NM}: W_1, W_2, C_1, \ldots, W_1 \]
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: \( \frac{dL}{d\theta} \)

\( \triangleright \) the gradient tells us how the loss changes when we modify the model params \( \theta \)

3. take a step in the direction of the negative gradient, minimizing the loss: \( \theta_{\text{new}} = \theta_{\text{old}} - \eta \frac{dL}{d\theta} \)

\( \triangleright \) learning rate, controls step size

\( \triangleright \) hyperparameter, tuned manually

---

**Diagram:**

- Loss \( L(\theta) \) axis
  - Random initialization \( \theta_0 \)
  - Gradient descent path
  - Minimum loss \( \text{GOAL!} \)
  - Learning rate \( \eta \) controls step size

---
Concrete example: (single neuron)

\[
\begin{align*}
\text{input} & \quad w_1 \quad \text{hidden} \quad w_2 \quad \text{output} \\
X & \quad \rightarrow & \quad h \quad \rightarrow & \quad o \\
\text{params?} & \quad w_1, w_2 \\
\text{inputs:} & \quad (x, y) \\
\end{align*}
\]

\[h = \tanh(w_1x) \quad o = \tanh(w_2x)\]

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

**Step 1: Forward propagation**

1. Compute the value of each neuron \(h, o\) using the model equations.
2. Predict the value of \(y\) = 0.
3. Goal: make 0 as close to \(y\) as possible.

**Step 2: Compute loss fn**

- Use square loss in this example. For simplicity, \[L = \frac{1}{2} (y - o)^2\]

**Step 3: Compute gradients WRT \(\theta\)**

\[
\frac{dL}{dw_1}, \quad \frac{dL}{dw_2}
\]
\[ \frac{dL}{dw_2} \]

**Chain Rule (calculus)**

\[ \frac{d}{dx} g(f(x)) = \frac{dg}{df} \cdot \frac{df}{dx} \]

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

\[ o = \tanh(w_2 h) \]

\[ o = \tanh(a) \]

\[ \frac{dL}{dw_2} = \frac{dL}{do} \cdot \frac{do}{da} \cdot \frac{da}{dw_2} \]

\[ -(y - o)(1 - o^2)h \]

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

Intermediate var:

- Let's say \( a = w_2 h \)

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

Intermediate var \( b = w_1 x \)

\( \Rightarrow \) chain rule

\( \Rightarrow \) compute these terms

\( \Rightarrow \) cache \( \frac{dL}{dw_2} \)
Intuition for backprop:

Cache derivatives that appear higher up in the network and use them to compute lower-level derivatives.

Param update:

\[ w_{1,\text{new}} = w_{1,\text{old}} - \eta \frac{\partial L}{\partial w_1} \]
\[ w_{2,\text{new}} = w_{2,\text{old}} - \eta \frac{\partial L}{\partial w_2} \]

Single neurons $\Rightarrow$ multiple neurons

$\Rightarrow$ matrix/vector notation

Linear layer: simple component of NNs

\[ y = f(XW) \]

- Each row of $X$ contains a vector that corresponds to a single training example (a batch)
- $N =$ number of examples (batch size)
- $D =$ dimensionality of embeddings
- $M =$ dimensionality of hidden layers

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

\[ f = \text{identity} \quad f(x) = x \]
\[ X = \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix}, \quad W = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix} \]

\[ Y = \begin{bmatrix} x_{11}w_{11} + x_{12}w_{21} & x_{11}w_{12} + x_{12}w_{22} & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix} \]

Let's assume we're given \( \frac{dL}{dy} \)

We want to compute \( \frac{dL}{dx} \), \( \frac{dL}{dW} \)

\[
\frac{dL}{dx} = \frac{dL}{dy} \left( \frac{dy}{dx} \right)
\]

\( \Rightarrow \) Jacobian matrix \( (N \times M) \times (N \times D) \)

We can compute \( \frac{dL}{dx} \) w/o forming \( \frac{dL}{dy} \)

\[
\frac{dL}{dx} = \begin{bmatrix} \frac{dL}{dx_{11}} & \frac{dL}{dx_{12}} \\ \frac{dL}{dx_{21}} & \frac{dL}{dx_{22}} \end{bmatrix}
\]
\[
\frac{dC}{dy} = \begin{bmatrix}
\frac{dl}{dy} & \frac{dl}{dy} & \frac{dl}{dy}
\end{bmatrix}
\]

\[
\frac{dl}{dx_{11}} \cdot \frac{dy}{dx_{11}} \rightarrow \begin{bmatrix}
w_{11} & w_{12} & w_{13}
\end{bmatrix}
\]

\[
\frac{dl}{dx_{12}} \rightarrow \begin{bmatrix}
w_{21} & w_{22} & w_{23}
\end{bmatrix}
\]

\[
\frac{dl}{dx_{21}} \rightarrow \begin{bmatrix}
w_{11} & w_{12} & w_{13}
\end{bmatrix}
\]

\[
\frac{dl}{dx_{22}} \rightarrow \begin{bmatrix}
w_{21} & w_{22} & w_{23}
\end{bmatrix}
\]

\[
\frac{dl}{dy} = \frac{dl}{dy} + \frac{dl}{dy} + \frac{dl}{dy}
\]

\[
\frac{dl}{dy} + \frac{dl}{dy} + \frac{dl}{dy} = w_{11} + w_{12} + w_{13}
\]
\[
\frac{dl}{dx} = \begin{bmatrix}
\frac{dl}{dy_{11}} + \frac{dl}{dy_{12}} \frac{dl}{w_{12}} + \frac{dl}{dy_{13}} \frac{dl}{w_{13}} \\
\frac{dl}{dy_{21}} \frac{dl}{w_{21}} + \frac{dl}{dy_{22}} \frac{dl}{w_{22}} + \frac{dl}{dy_{23}} \frac{dl}{w_{23}} 
\end{bmatrix}
\]

Rewritten as a matrix product:

\[
\begin{bmatrix}
\frac{dl}{dy_{11}} & \frac{dl}{dy_{12}} & \frac{dl}{dy_{13}} \\
\frac{dl}{dy_{21}} & \frac{dl}{dy_{22}} & \frac{dl}{dy_{23}}
\end{bmatrix}
\begin{bmatrix}
w_{11} & w_{12} \\
w_{12} & w_{22} \\
w_{13} & w_{23}
\end{bmatrix}
\]

\[
\frac{dl}{dx} = \frac{dl}{dy} W^T
\]

In practice, we do this instead of computing the Jacobian:

\[
\frac{dl}{dW} = X^T \frac{dl}{dy}
\]

**Today:** \(
\rightarrow\) Grad. Descent / Backprop

\(
\rightarrow\) You should be able to implement backprop for a single neuron network.
Backprop: - Chain rule
- Caching intermediate derivatives
- Efficient computation to avoid forming Jacobian