Outline

- Recap and neural networks motivation
- Neural Networks
  - A single neuron
  - A single layer
  - Many layers
    - Design choices (activation functions, loss functions choices)
- Forward pass
- Backward pass (back-propagation)
e.g. linear regression represented as a computation graph

![Computation Graph for Linear Regression]

Each data point incurs a loss of 

\[ \sum (w^T x^{(i)} + w_0 - y^{(i)})^2 \]

Repeat for each data point, sum up the individual losses

Gradient of the total loss gives us the "signal" on how to optimize for \(w, w_0\)

learnable parameters (weights)
e.g. linear logistic regression (linear classification) represented as a computation graph

- Each data point incurs a loss of $- (y^{(i)} \log g^{(i)} + (1 - y^{(i)}) \log (1 - g^{(i)}))$
- Repeat for each data point, sum up the individual losses
- Gradient of the total loss gives us the "signal" on how to optimize for $w, w_0$
We saw that, one way of getting complex input-output behavior is to leverage nonlinear transformations

\[
\phi \left( [x_1, x_2]^\top \right) = [1, x_1, x_2, x_1^2, x_1x_2, x_2^2]^\top
\]

e.g. use for decision boundary
\[
\text{sign}(0 + 0x_1 + 0x_2 + 0x_1^2 + 4x_1x_2 + 0x_2^2 + 0)
\]

👆 importantly, linear in \( \phi \), non-linear in \( x \)
Today (2nd cool idea): "stacking" helps too!

\[ z = w^T x \]
\[ y = \text{sign}(z) \]
So, two epiphanies:

- nonlinearity empowers linear tools
- stacking helps

👋 heads-up: all neural network graphs focus on a single data point for simple illustration.)
Outline

- Recap and neural networks motivation
- Neural Networks
  - A single neuron
  - A single layer
  - Many layers
  - Design choices (activation functions, loss functions choices)
- Forward pass
- Backward pass (back-propogation)
A single neuron is

- the basic operating "unit" in a neural network.
- the basic "node" when a neural network is viewed as computational graph.

- \( x \): \( m \)-dimensional input (a single data point)
- \( w \): weights (i.e. parameters)
- \( z \): pre-activation scalar output
- \( f \): activation function
- \( a \): post-activation scalar output

**Diagram:**

- Neuron, a function, maps a vector input \( x \in \mathbb{R}^m \) to a scalar output
- Inside the neuron, circles do function evaluation/computation
- \( f \): we engineers choose
- \( w \): learnable parameters
A single layer is

- made of many individual neurons.
- (# of neurons) = (layer output dimension).
- typically, all neurons in one layer use the same activation $f$ (if not; uglier/messier algebra)
- typically, no "cross-wire" between neurons. e.g. $z_1$ doesn't influence $a_2$. in other words, a layer has the same activation applied element-wise. (softmax is an exception to this, details later.)
- typically, fully connected. i.e. there's an edge connecting $x_i$ to $z_j$, for all $i \in \{1, 2, 3, \ldots, m\}$; $j \in \{1, 2, \ldots, n\}$. in other words, all $x_i$ influence all $a_j$. 

![Diagram of a single layer](image)
A (feed-forward) neural network is

\[ \left( \sum f_1(\cdot) \right) \left( \sum f_1(\cdot) \right) \left( \sum f_2(\cdot) \right) \ldots \]

\( x_1 \quad x_2 \quad \ldots \quad x_m \)

\( W_1 \quad W_2 \)

input

linear combo

learnable weights

activations

layer

input

linear combo

learnable weights

activations

layer
Activation function $f$ choices

- $\sigma$ used to be popular
- firing rate of neuron
- $\sigma'(z) = \sigma(z) \cdot (1 - \sigma(z))$
ReLU is the de-facto activation choice nowadays

\[
\text{ReLU}(z) = \begin{cases} 
0 & \text{if } z < 0 \\
(z) & \text{otherwise}
\end{cases}
= \max(0, z)
\]

- **Default** choice in **hidden** layers.
- **Pro**: very efficient to implement, choose to let the gradient be:

\[
\frac{\partial \text{ReLU}(z)}{\partial z} := \begin{cases} 
0 & \text{if } z < 0 \\
1 & \text{otherwise}
\end{cases}
\]

- **Drawback**: if strongly in negative region, unit can be "dead" (no gradient).
- Inspired variants like elu, leaky-relu.
The last layer, the output layer, is special

- activation and loss depends on problem at hand
- we've seen e.g. regression (one unit in last layer, squared loss).

More complicated example: predict one class out of $K$ possibilities
then last layer: $K$ neurons, softmax activation
e.g., say $K = 5$ classes

$$g = A^L = f^L(Z^L) = \text{softmax}(Z^L) = \begin{bmatrix} \exp(z_1)/\sum_i \exp(z_i) \\ \vdots \\ \exp(z_K)/\sum_i \exp(z_i) \end{bmatrix}$$
\[ L(g, y) = \log(g) \]

**Ground truth label** \( y \)

**Loss**

\[
\mathcal{L}_{	ext{nllm}}(g, y) = -\sum_{k=1}^{K} y_k \cdot \log(g_k)
\]

### Class Predictions

- Dolphin
- Cat
- Grizzly bear
- Angel fish
- Chameleon
- Clown fish
- Iguana
- Elephant

### Log Probabilities

- \(-\infty\)
- Log prob
- \(0\)

### Ground Truth

- Dolphin
- Cat
- Grizzly bear
- Angel fish
- Chameleon
- Clown fish
- Iguana
- Elephant

### Probabilities

- \(0\)
- Prob
- \(-\infty\)

### Loss

- \(-\infty\)
- Loss
- \(0\)
Outline

- Recap and neural networks motivation
- Neural Networks
  - A single neuron
  - A single layer
  - Many layers
    - Design choices (activation functions, loss functions choices)
- Forward pass
- Backward pass (back-propogation)
How do we optimize

\[ J(W) = \sum_{i=1}^{L} \mathcal{L} \left( f_L \left( \ldots f_2 \left( f_1 \left( x^{(i)}, w_1 \right), w_2 \right), \ldots w_L \right), y^{(i)} \right) \]
Forward propagation to obtain the output (model's guess)

\[ X = A_0^0, \quad W_1^1, \quad Z_1^1 \xrightarrow{\frac{\partial \text{loss}}{\partial Z_1^1}} f_1, \quad A_1^1, \quad W_2^2, \quad Z_2^2 \xrightarrow{\frac{\partial \text{loss}}{\partial Z_2^2}} f_2, \quad \ldots, \quad A_1^{L-1}, \quad W_L^L, \quad Z_L^L \xrightarrow{\frac{\partial \text{loss}}{\partial Z_L^L}} f_L, \quad A_L^L, \quad \text{Loss} \]

Backpropagation to obtain gradients with respect to the loss
Backprop = gradient descent & the chain rule

Recall that, the chain rule says:
For the composed function: \( h(x) = f(g(x)) \), its derivative is: 
\[
    h'(x) = f'(g(x))g'(x)
\]

Here, our loss depends on the final output,
and the final output \( A^L \) comes from a chain of composition of functions
Backprop = gradient descent & the chain rule
Backprop = gradient descent & the chain rule

\[ \frac{\partial \text{loss}}{\partial Z^{(\ell)}} = \frac{\partial A^{(\ell)}}{\partial Z^{(\ell)}} \frac{\partial Z^{(\ell+1)}}{\partial A^{(\ell)}} \frac{\partial A^{(\ell+1)}}{\partial Z^{(\ell+1)}} \cdots \frac{\partial A^{(L-1)}}{\partial Z^{(L-1)}} \frac{\partial Z^{(L)}}{\partial A^{(L-1)}} \frac{\partial A^{(L)}}{\partial Z^{(L)}} \frac{\partial \text{loss}}{\partial A^{(L)}} \]

\[ n^\ell \times 1 \quad n^\ell \times n^\ell \quad n^\ell \times n^{\ell+1} \quad n^{\ell+1} \times n^{\ell+1} \quad \cdots \quad n^{L-1} \times n^{L-1} \quad n^{L-1} \times n^L \quad n^L \times n^L \quad n^L \times 1 \]
(The demo won't embed in PDF. But the direct link below works.)

https://playground.tensorflow.org/
Two different ways to represent a function
Two different ways to represent a function
Data transformations for a variety of neural net layers

\[ x_{\text{out}} = 2x_{\text{in}} \]

\[ x_{\text{out}} = \frac{x_{\text{in}} + 1}{2} \]

\[ x_{\text{out}} = \text{relu}(x_{\text{in}}) \]

\[ x_{\text{out}} = \text{sigmoid}(x_{\text{in}}) \]
### Wiring Graph

**Linear**

- Input node: $x_{in}$
- Weight node: $W$
- Bias node: $b$
- Output node: $x_{out}$

Equation:

$$x_{out} = Wx_{in} + b$$

**ReLU**

- Input nodes: $x_{in}$
- Output nodes: $x_{out}$

Equation:

$$x_{out} = \max(x_{in}, 0)$$

### Mapping 1D

- Input: $x_{in}$
- Output: $x_{out}$
- Mapping: $x_{out} = Wx_{in} + b$

### Mapping 2D

- Input: $x_{in}$
- Output: $x_{out}$
- Mapping: $x_{out} = \max(x_{in}, 0)$
Optimizing parameters versus optimizing inputs

\[ \frac{\partial J}{\partial \theta} \quad \text{How much the total cost is increased or decreased by changing the parameters.} \]
Optimizing parameters versus optimizing inputs

\[ \frac{\partial y_j}{\partial x} \]  How much the “cat” score is increased or decreased by changing the image pixels.
Adversarial attacks

\[
\frac{\partial y_j}{\partial r} \quad \text{What adversarial signal } r \text{ should we add to change the output label?}
\]

Adversarial attacks

\[
\begin{align*}
\arg\max_r p(y = \text{ostrich}|x + r) \quad \text{subject to} \quad \|r\| < \epsilon
\end{align*}
\]

Summary

- We saw last week that introducing non-linear transformations of the inputs can substantially increase the power of linear regression and classification hypotheses.
- We also saw that it’s kind of difficult to select a good transformation by hand.
- Multi-layer neural networks are a way to make (S)GD find good transformations for us!
- Fundamental idea is easy: specify a hypothesis class and loss function so that $d \frac{\text{Loss}}{\text{theta}}$ is well behaved, then do gradient descent.
- Standard feed-forward NNs (sometimes called multi-layer perceptrons which is actually kind of wrong) are organized into layers that alternate between parametrized linear transformations and fixed non-linear transforms (but many other designs are possible!)
- Typical non-linearities include sigmoid, tanh, relu, but mostly people use relu
- Typical output transformations for classification are as we have seen: sigmoid and/or softmax
- There’s a systematic way to compute $d \frac{\text{Loss}}{\text{theta}}$ via backpropagation
We'd love it for you to share some lecture feedback.

Thanks!