CS21si: AI for Social Good

Lecture 2: Basics of Neural Networks

Intro

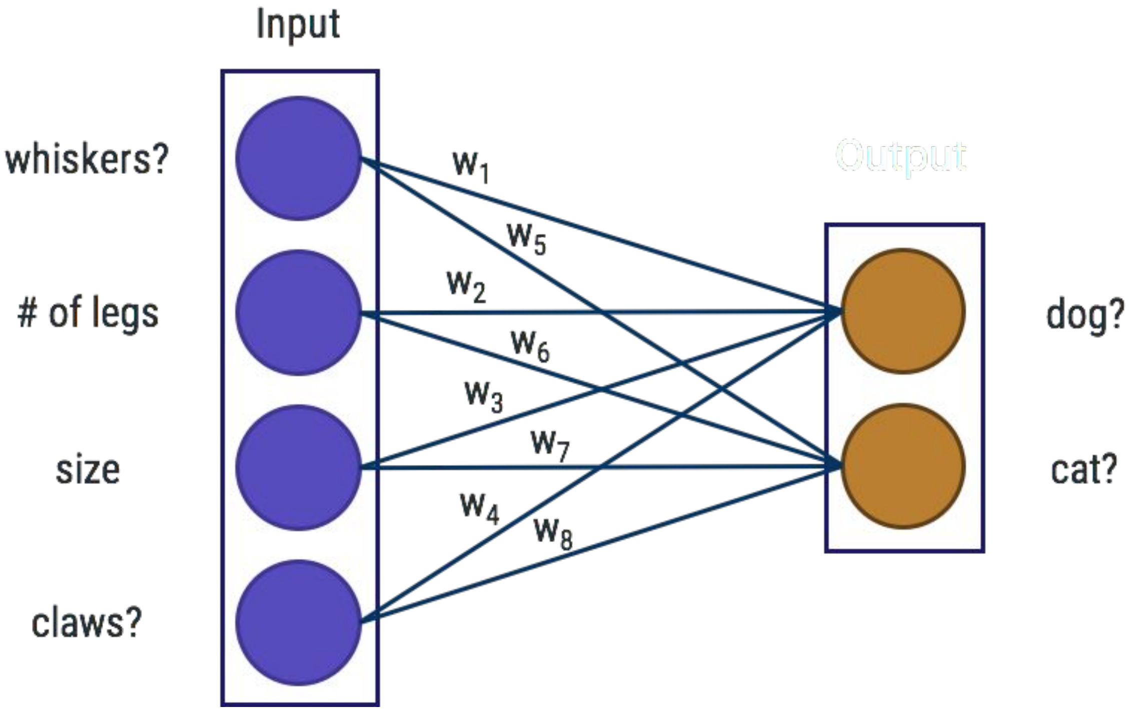
- We want to generate a prediction from some input
- Output should give some probability of all our expected outputs
- Network should give highest probability to correct class most of the time

Forward Pass

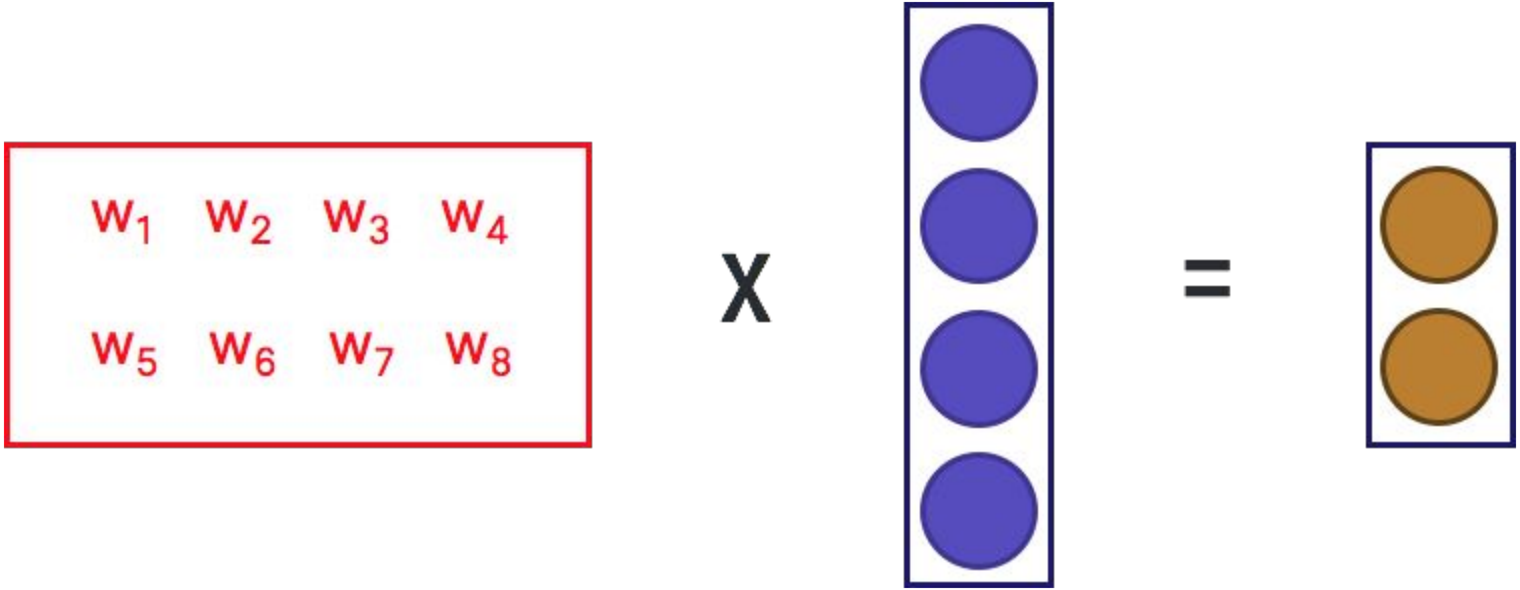
Fully-Connected Layers

- Fully-connected layers are computations in which every input element is connected to every output elements
- We transform the input by using weights to produce an output
- Output may have different shape than input

Fully-Connected Network



FC-Net as a Matrix Multiplication



Probability Distribution

- The output of our network is arbitrarily scaled
- We want to turn the arbitrary scale into a proper probability distribution
- We do so by using the softmax calculation:

$$p_i = \frac{e^{o_i}}{\sum_j e^{o_j}}$$

Data Splitting

- We are interested in how our model does on unseen data above all
- We can split data into training and test—but if we optimize on test, test becomes “indirectly” seen
- Instead, we split into train, validation, and test; we run test only once

Jupyter Notebook Exercises: Part 1

You'll need:

```
np.random.randn(size)

np.zeros(size)

np.matmul(a, b)
```

Loss Value

- The network needs an “objective” to work towards—being accurate
- We use the targets to figure out how “incorrect” the network was
- The loss metric we use is cross-entropy loss:

$$L = -\log(p_c)$$

Cross-Entropy Layer

Softmax

Loss

$$L = -\log(0.731) = 0.313$$

Jupyter Notebook Exercises: Part 2

You'll need:

```
np.exp(x)

np.sum(x, axis=n)

np.log(x)
```

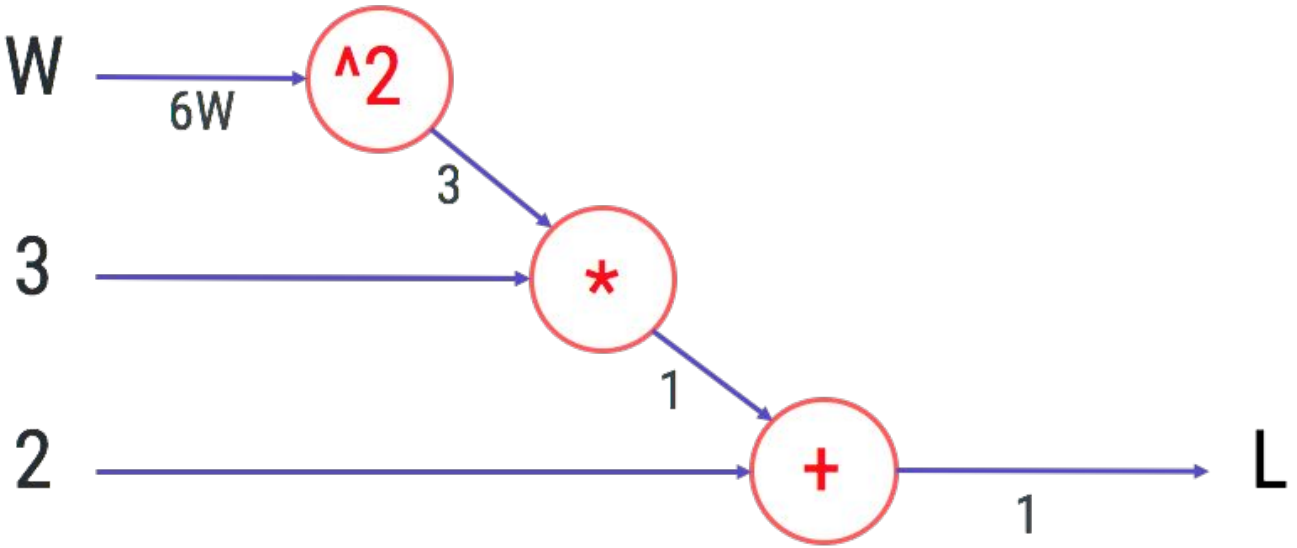
Backward Pass

Backpropagation

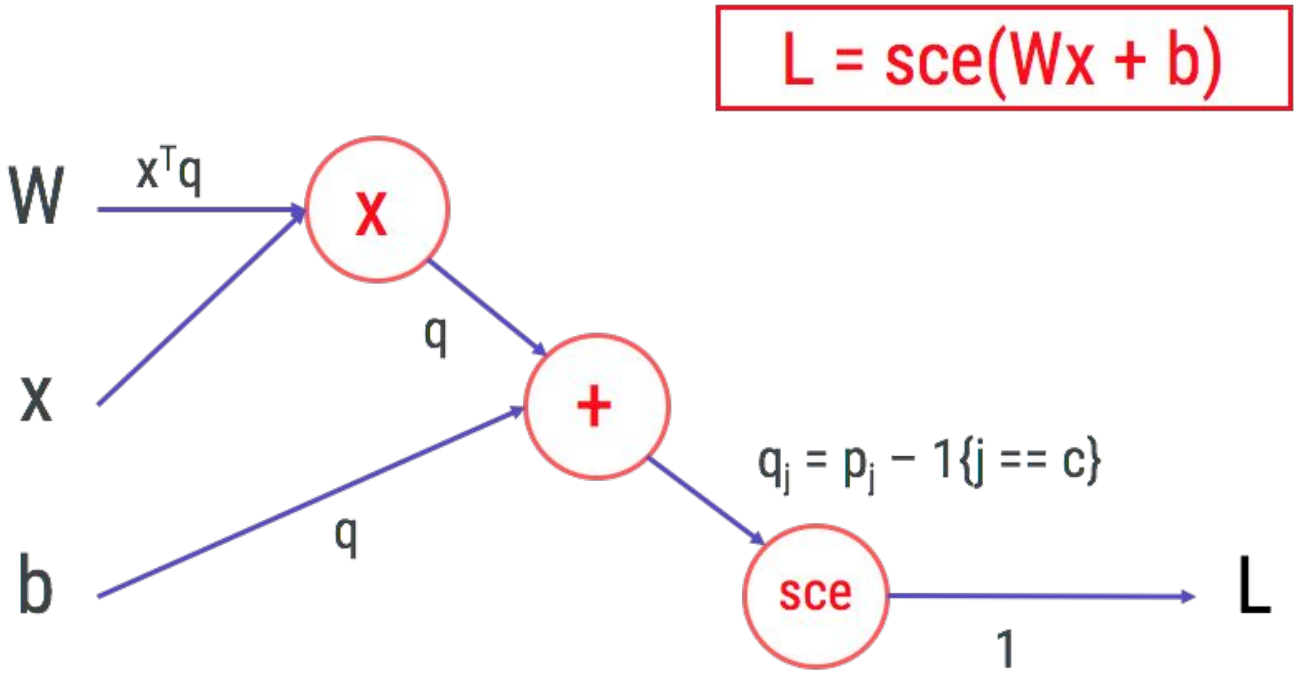
- Each computation is represented as a node in a computational graph
- Gradients are computed at each node of the graph
- Chain rule is applied recursively to get total gradients on each weight

Simple Backprop

$$L = 3W^2 + 2$$



Multidimensional Backprop



Jupyter Notebook Exercises: Part 3

You'll need:

```
np.sum(x, axis=n)

np.matmul(a, b)
```

Training

Weight Update

- Once we find gradients, we must change each weight in the opposite direction
- We can use an update rule to accomplish this; the simplest is SGD:

$$W_i := Wi - \alpha \left(\frac{\partial L}{\partial W_i} \right)$$

Jupyter Notebook Exercises: Part 4

You'll need:

```
np.random.choice(x, n)
```

Training Word Vectors

One-Hot Vectors

- Represent a word as sparse vector of mostly 0’s and a single 1

Input Data

Source Text	Training Samples
The quick brown fox jumps over the lazy dog.	(the, quick) (the, brown)
The quick brown fox jumps over the lazy dog.	(quick, the) (quick, brown) (quick, fox)
The quick brown fox jumps over the lazy dog.	(brown, the) (brown, quick) (brown, fox) (brown, jumps)
The quick brown fox jumps over the lazy dog.	(fox, quick) (fox, brown) (fox, jumps) (fox, over)

The Model

- 2-hidden layer neural network
  - Layer 1 maps N to D
  - Layer 2 maps D to N
- No non-linear activation between layer 1 and 2

Word Vector

- The weight matrix of layer 1 has dimension NxD
- We use the rows of this matrix as word vectors!

In-Class Homework Time

Take this time to get started on the homework! We'll be around to answer questions.

What we’ve learned...

- Neural networks are simple way to model complex data
- We can stack layers to increase complexity
- word2vec is based on simple NN ideas
- It’s super easy to accidentally make racist word vectors!