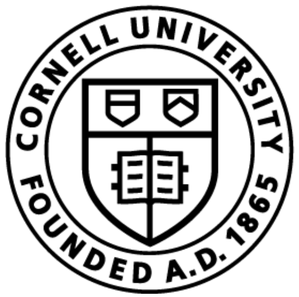
Lecture 4: FFNNs



Cornell Bowers CIS
Computer Science

Tanya Goyal

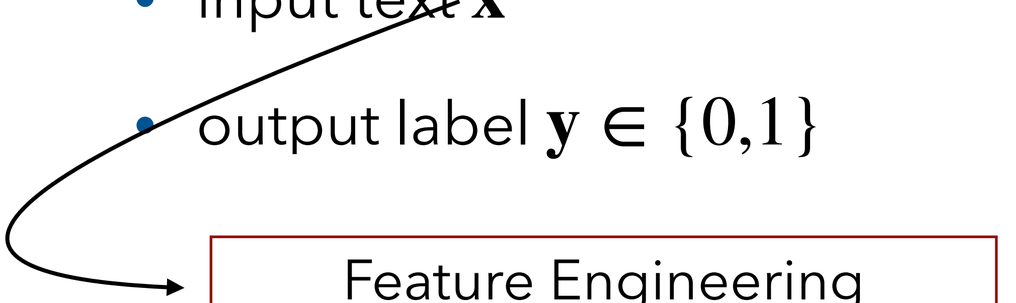
CS 4740 (and crosslists): Introduction to Natural Language Processing

Today

- Recap: Logistic Regression
- Feed Forward Neural Networks

Recap: Binary Logistic Regression

- **Training Data**
  - input text $\mathbf{x}$
  - output label $\mathbf{y} \in \{0,1\}$



$x_0 = 1 \longrightarrow w_0$

$x_1 = \#\text{words} \longrightarrow w_1$

$x_2 = \#\text{“great”} \longrightarrow w_2$

$x_3 = \#\text{ positive words} \longrightarrow w_3$

$x_4 = \#\text{ negative words} \longrightarrow w_4$

$$P(\mathbf{y}=1 \mid \mathbf{x}) = \frac{e^{\sum_i w_i x_i}}{1 + e^{\sum_i w_i x_i}}$$

$$P(\mathbf{y}=0 \mid \mathbf{x}) = \frac{1}{1 + e^{\sum_i w_i x_i}}$$

Goal: Learn Weights $\mathbf{w} = [w_0, w_1 \ldots w_K]$

Recap: Binary Logistic Regression

$$P(\mathbf{y} = 1 \,|\, \mathbf{x}) = \frac{e^{w.x}}{1 + e^{w.x}}$$

$$P(\mathbf{y} = 0 \,|\, \mathbf{x}) = \frac{1}{1 + e^{w.f}}$$

Goal: Learn Weights $\mathbf{w} = [w_0, w_1 \ldots w_K]$

Minimize Loss using **stochastic gradient descent.**

$$\text{Loss} = \sum_{j=1}^{N} -\log P(\hat{y} = y^j \,|\, x^j)$$

j = Index of datapoint.

Recap: Multinomial Logistic Regression

- Multinomial Logistic Regression: $P(\hat{y} = y^j) = \frac{e^{w_i x^j}}{\sum_{k=1}^{L} e^{w_i x^j}}$
- Loss: $\mathscr{L} = -\sum_j \log P(\hat{y} = y^j \mid x^j)$
- Matrix multiplication view?

j = Index of datapoint.
i = Index of label.

Recap: Word Embeddings

- In NLP, we represent word types with **vectors**.

$$\mathbf{x}^{\text{Cornell}} = [x_1, x_2, x_3 \cdots, x_d]$$

d-dimension vector, d is fixed.

- How do we learn these word embeddings?
  - Sparse Embeddings?
  - Dense Embeddings?

Recap: Skip-gram Model

- Idea:
  - Train a binary classifier on a **prediction** task:

    Is word w *likely* to occur near word ``cherry''?

    $P(+|w,c)$ ← c is a context word of w

    $P(-|w,c) = 1 - P(-|w,c)$ ← c is not a context word of w
- Operationalization:
  - Randomly initialize C and W matrices that consist of word and context embeddings for all |V| words in the vocabulary.
  - Using Logistic Regression, update these matrices to maximize $P(y|w,c)$ ← here y is either + or - depending on whether w, c are nearby words or not.

Recap: Skip-gram Model

$P(+|w,c)$ ← c is a context word of w. $P(-|w,c) = 1 - P(+|w,c)$ ← c is not a context word of w

$P(+|w,c) \approx \mathbf{c} \,.\, \mathbf{w}$

Word vector for word w

Context vector for word c

This is not a probability.

$$P(+|w,c) = \sigma(\mathbf{c} \,.\, \mathbf{w}) = \frac{1}{1 + \exp(-\mathbf{c} \,.\, \mathbf{w})}$$

- Classification model. What is our objective? Maximize log likelihood of the data.

$$\sum_{(w,c_{pos}) \in +} \log P(+|w,c_{pos}) + \sum_{(w,c_{neg}) \in -} \log P(-|w,c_{neg})$$

Recap: Skip-gram Model

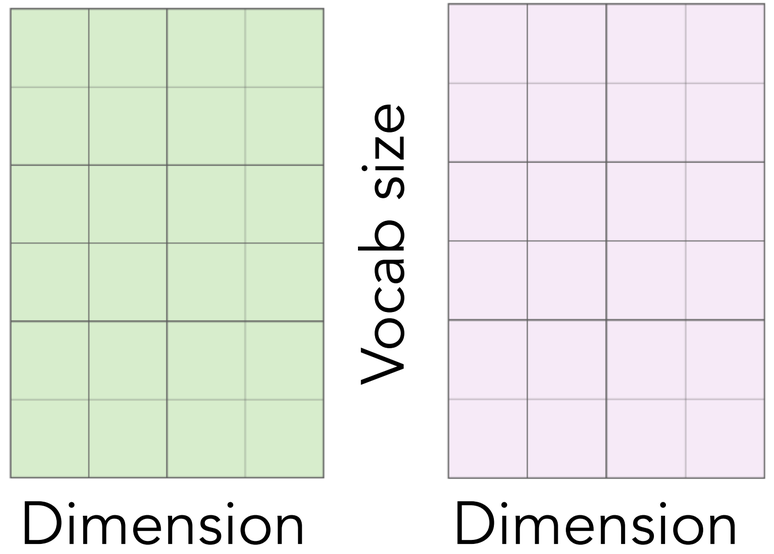
- Focusing on one target word (log likelihood)

$$L(\theta) = \log P(+|w, c_{\text{pos}}) + \log P(-|w, c_{\text{neg}})$$

$$= \log \frac{\exp(\mathbf{c}_{pos} \cdot \mathbf{w})}{1 + \exp(\mathbf{c}_{pos} \cdot \mathbf{w})} + \log \frac{1}{1 + \exp(\mathbf{c}_{neg} \cdot \mathbf{w})}$$

$$P(+|w, c) = \frac{\exp(\mathbf{c} \cdot \mathbf{w})}{1 + \exp(\mathbf{c} \cdot \mathbf{w})}$$

$$P(-|w, c) = \frac{1}{1 + \exp(\mathbf{c} \cdot \mathbf{w})}$$



Recap: Skip-gram Model

- Initialize C^o, W^0
- For each training sample:

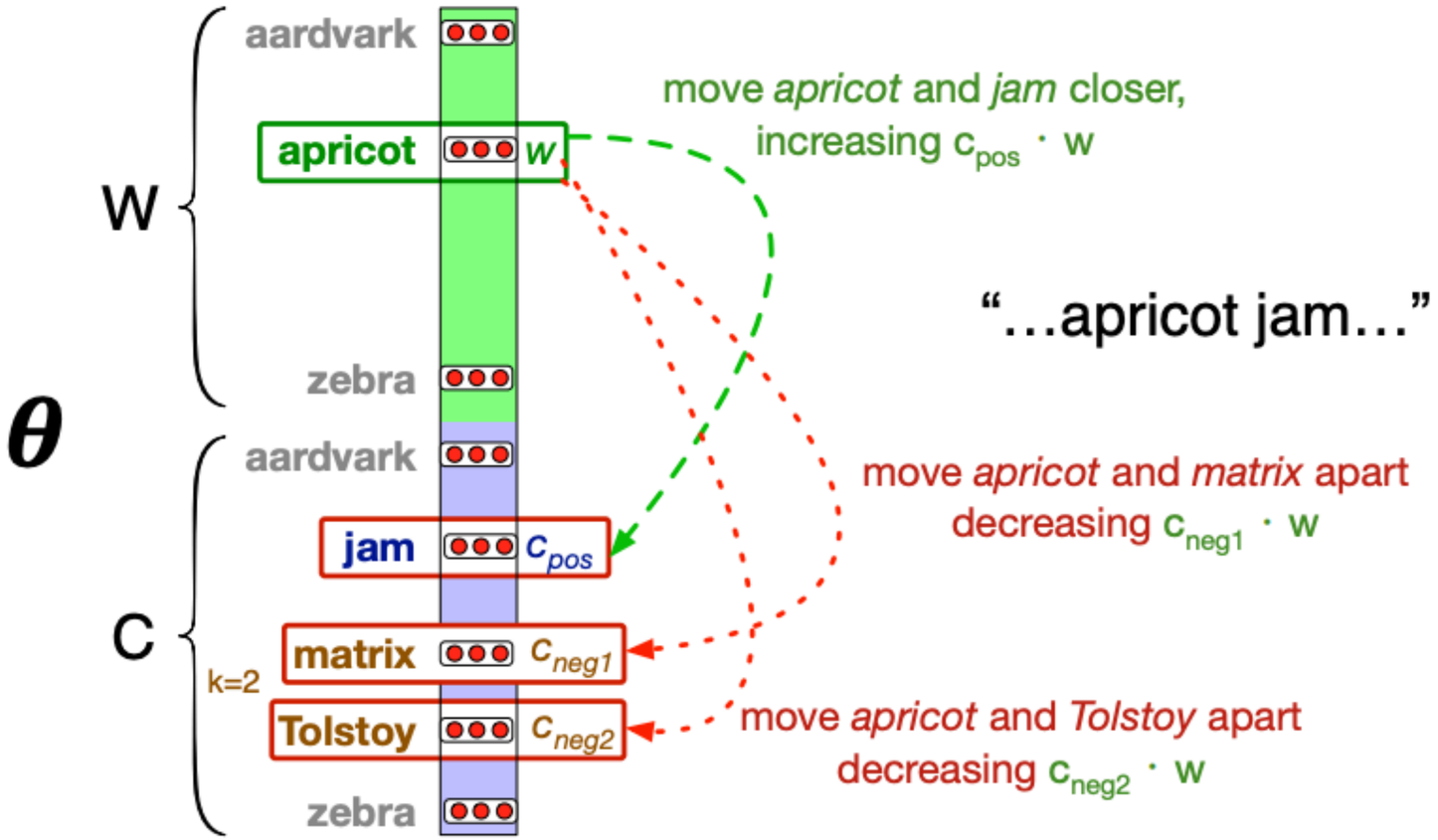
$$\frac{\partial L}{\partial \mathbf{c}_{\text{pos}}} = [\sigma(\mathbf{c}_{\text{pos}} \cdot \mathbf{w}) - 1]\ \mathbf{w}$$

$$\frac{\partial L}{\partial \mathbf{c}_{\text{neg}}} = [\sigma(\mathbf{c}_{\text{neg}} \cdot \mathbf{w})]\ \mathbf{w}$$

$$\frac{\partial L}{\partial \mathbf{w}} = [\sigma(\mathbf{c}_{\text{pos}} \cdot \mathbf{w}) - 1]\ \mathbf{c}_{\text{pos}} + [\sigma(\mathbf{c}_{\text{neg}} \cdot \mathbf{w})]\ \mathbf{c}_{\text{neg}}$$

- Gradient update!

Recap: Skip-gram Model



- To represent word w, we can
  - Concatenate $\mathbf{c}^i$ and $\mathbf{w}^i$
  - Keep $\mathbf{w}^i$

Let’s get back to logistic regression

- Goal: learn a weight vector W.
- To learn this, the first step is feature engineering. These are human designed features that rely on human intuitions of what might be discriminative between different features.
- **Representation learning** attempts to automatically learn good features and representations.
- **Deep learning** attempts to learn multiple levels of representations of increasing complexity/abstraction.

Slide Credit: Alane Suhr

The problem: generic world

- Feed-forward neural networks
  - As a mechanism for representation learning
  - As another approach to classification problems for text/language

For word embeddings

- Let's consider the task of Named Entity Recognition as our running example.
- For each word in our sentence, we want to classify it as either {PER, LOC, NULL}

[PER] [LOC]
London lives in Ithaca.

The problem: generic word vectors might not be optimal

- For a specific task, such as NE tagging, we could learn better representations.
  - Example: it'd be nice if the vectors for LOC words were separate from the vectors for PER words.

[PER] [LOC]
London lives in Ithaca.

The solution we're building up to today: feed-forward neural networks (FFNNs) for classification

Given a input representation

$$\mathbf{x} = [x_1, \cdots x_i, \cdots x_d]$$

New vector representation

$$\mathbf{h} = [h_1, \cdots h_i, \cdots h_{d_1}]$$

Predict: probability distribution over the labels

$$\mathbf{y} = [y_1 = P(\text{LOC}\,|\,\mathbf{x}), P(\text{PER}\,|\,\mathbf{x}), P(\text{NULL}\,|\,\mathbf{x})]$$

FFNNs will acquire the ability to perform this task by learning to derive better representations of the word vectors.

The solution we're building up to today: feed-forward neural networks (FFNNs) for classification

Given a input representation

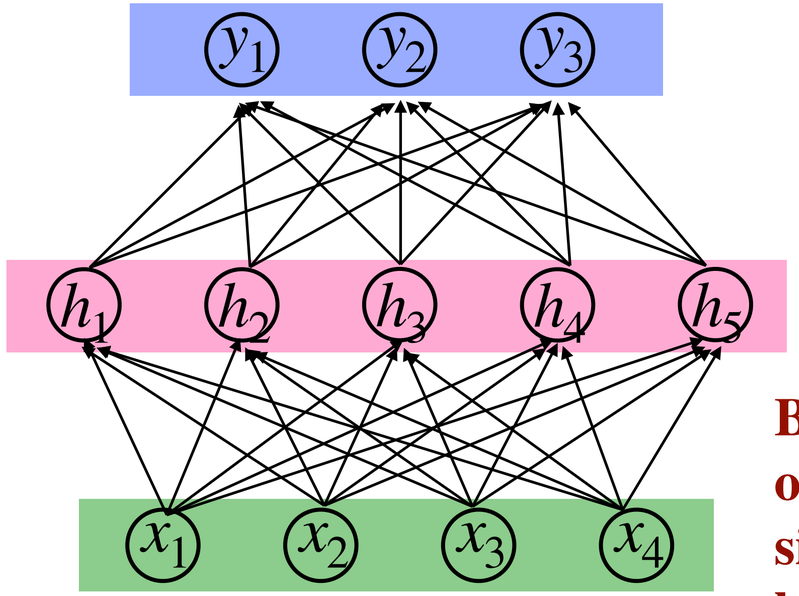
$\mathbf{x} = [x_1, \cdots x_i, \cdots x_d]$

New vector representation

$\mathbf{h} = [h_1, \cdots h_i, \cdots h_{d_1}]$

Predict: probability distribution over the labels

$\mathbf{y} = [y_1 = P(\text{LOC}\,|\,\mathbf{x}), P(\text{PER}\,|\,\mathbf{x}), P(\text{NULL}\,|\,\mathbf{x})]$

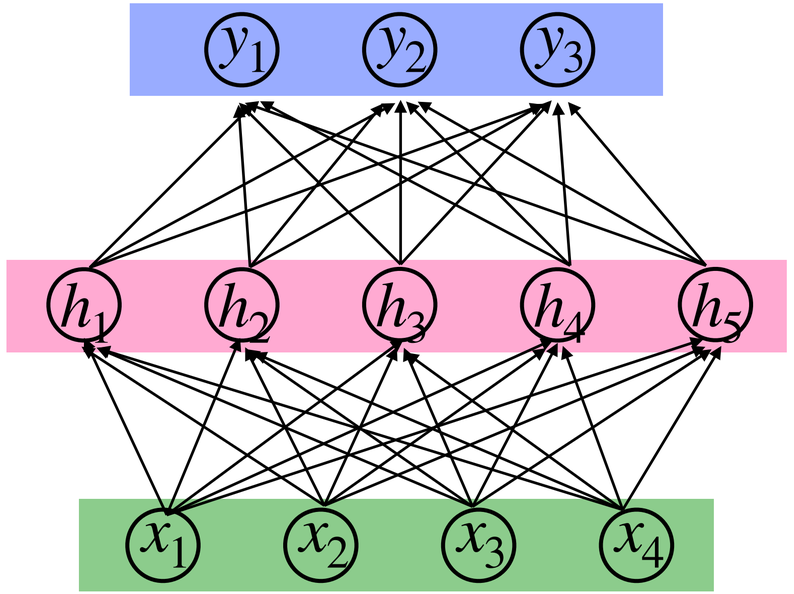


Bias term omitted for simplicity, let's come back to this.

Intuition: FFNNs learn how to better represent **x** (i.e, as **h**) so as to be able to accurately predict **y**.

The solution we're building up to today: feed-forward neural networks (FFNNs) for classification

- **Fully connected (FC) layers**
  - Each unit of one layer is connected to each unit of the next layer.
- We assume we already know how to represent **x** $\in R^d$. (Here d = 4).
  - Simple when **x** is a word.
- We'll see how to represent this when it's a sequence of words later.

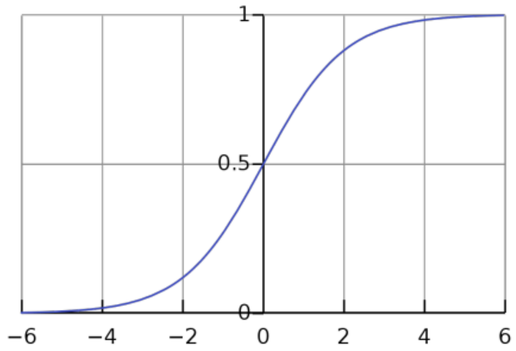


Intuition: FFNNs learn how to better represent **x** (i.e, as **h**) so as to be able to accurately predict **y**.

Activation functions

sigmoid

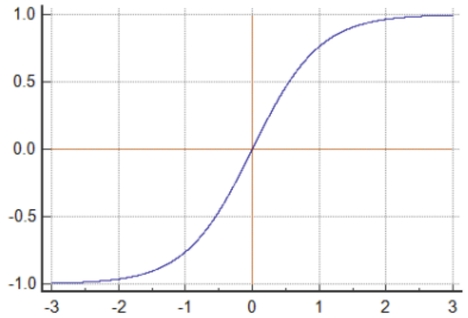
$$f(z) = \frac{1}{1 + e^{-z}}$$



$$f'(z) = f(z) \times (1 - f(z))$$

tanh

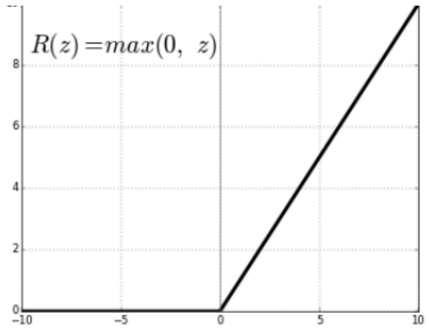
$$f(z) = \frac{e^{2z} - 1}{e^{2z} + 1}$$



$$f'(z) = 1 - f(z)^2$$

ReLU
(rectified linear unit)

$$f(z) = \max(0, z)$$



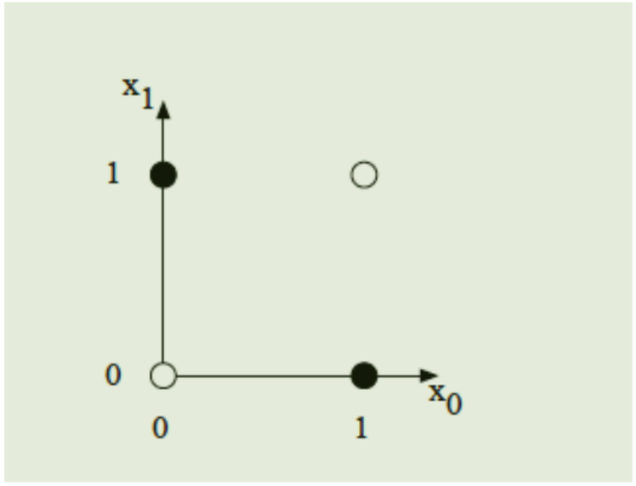
$$f'(z) = \begin{cases} 1 & z > 0 \\ 0 & z < 0 \end{cases}$$

- Applied element-wise: $f([z_1, z_2, \ldots z_n]) = [f(z_1), f(z_2) \ldots f(z_n)]$

Changing the original x's into new h's where items with different labels are separable

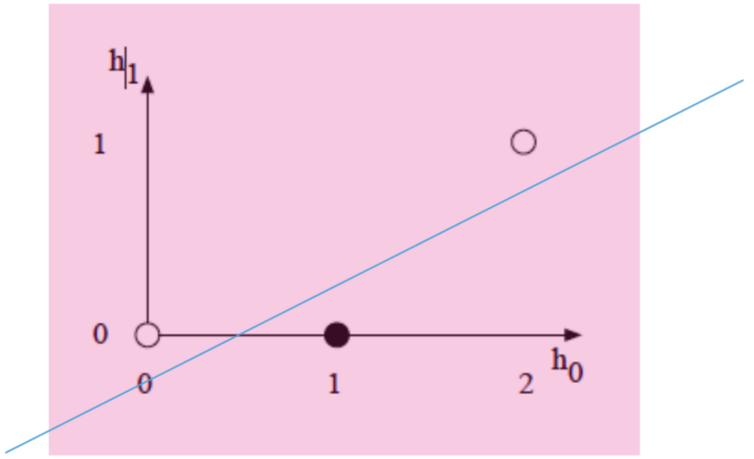
Our LOC/PER example

Suppose that ● = PER, and ○ = LOC, and our w2v vectors looked like this:



a) The original x space

There's a transform whereby the PERs can be separated from the LOCs:



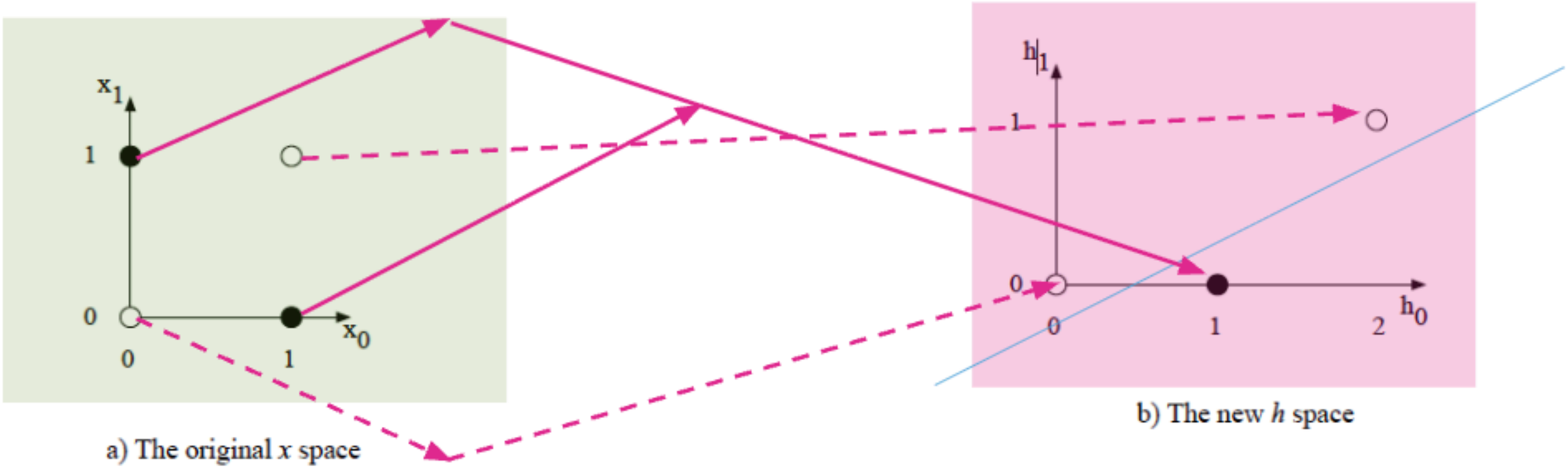
b) The new h space

Changing the original x's into new h's where items with different labels are separable

Our LOC/PER example

Suppose that ● = PER, and ○ = LOC, and our w2v vectors looked like this:

There's a transform whereby the PERs can be separated from the LOCs:



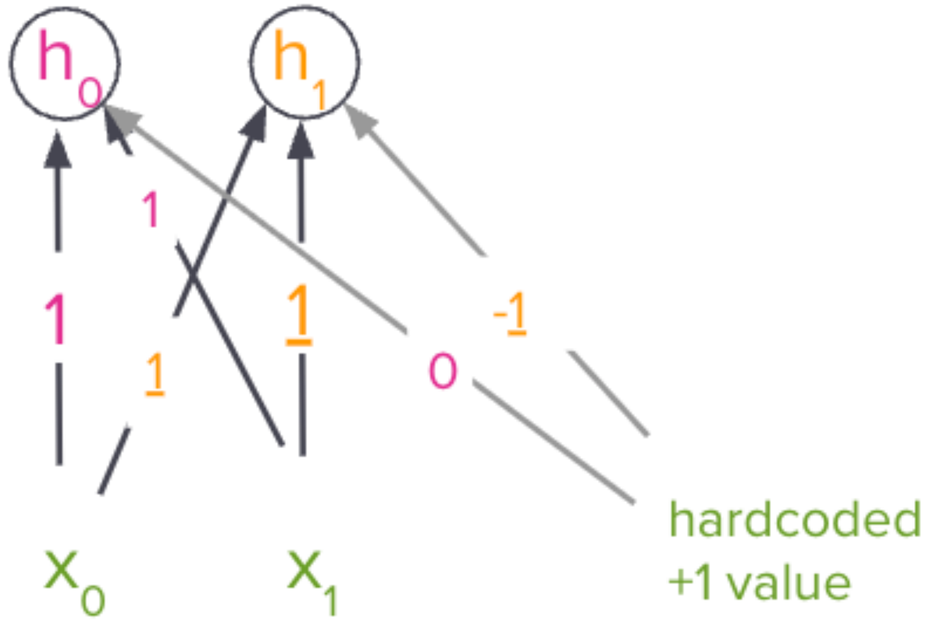
a) The original *x* space

b) The new *h* space

Changing the original x's into new h's where items with different labels are separable

- Consider a 2-d case. **Our LOC/PER example**

g = the ReLU activation function,
g(x) = max(x, 0)



Computation for $x = [0, 0]^T$

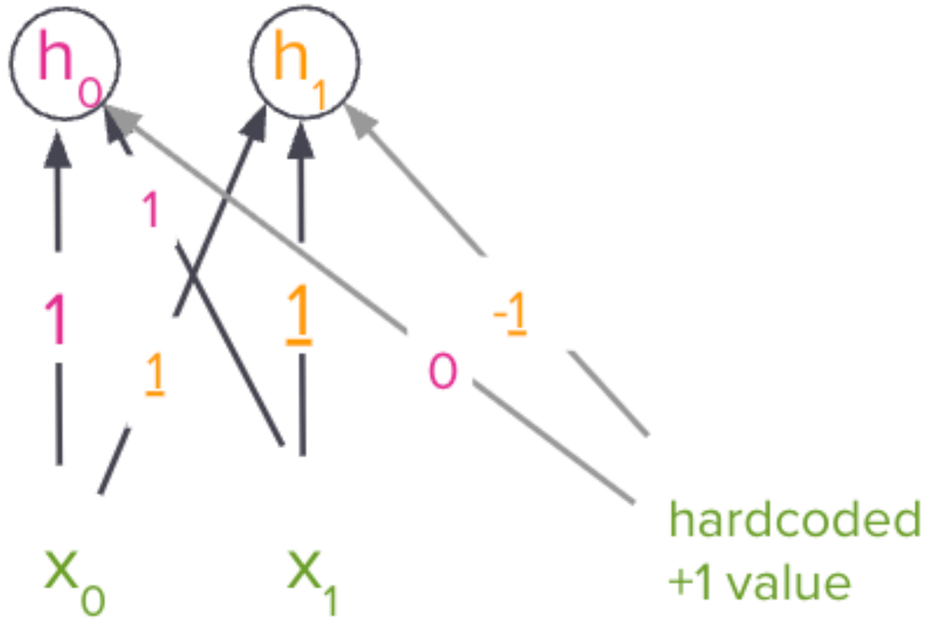
$h_0 =$

$h_1 =$

Changing the original x's into new h's where items with different labels are separable

- Consider a 2-d case. **Our LOC/PER example**

g = the ReLU activation function,
g(x) = max(x, 0)

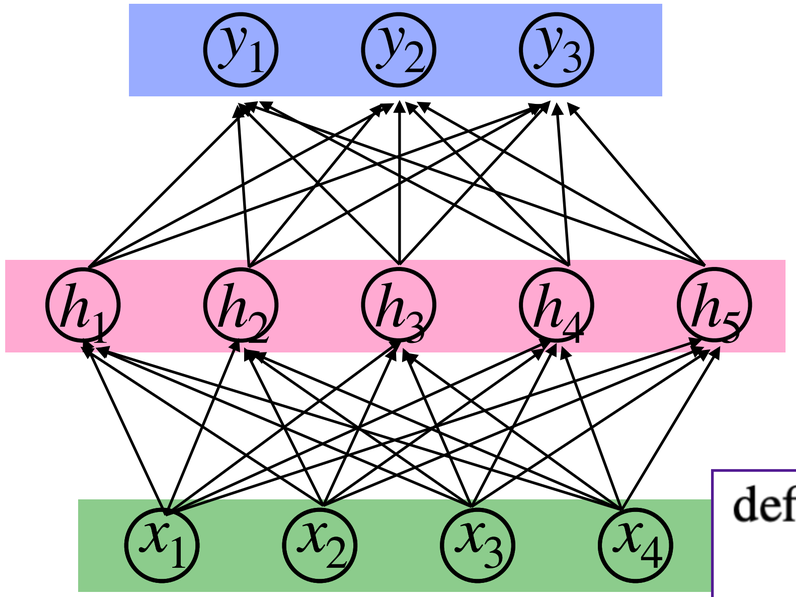


Computation for $x = [0, 0]^T$

$h_0 =$

$h_1 =$

Feed-forward neural networks (FFNNs), Matrix version



- Input layer $x = [x_1 x_2 \ldots x_4]^T$
- Hidden Layer:
  - $h = f(Wx) \quad \in R^5$
  - $W \in \mathbb{R}^?$

defined as:

$$\text{softmax}(\mathbf{z}_i) = \frac{\exp(\mathbf{z}_i)}{\sum_{j=1}^{d} \exp(\mathbf{z}_j)} \quad 1 \le i \le d \tag{7.9}$$

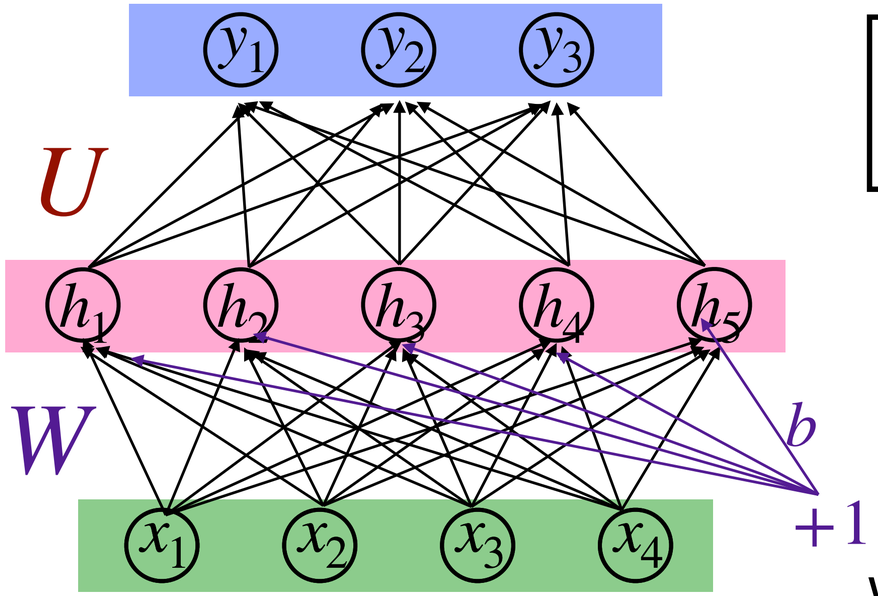
Thus for example given a vector

$$\mathbf{z} = [0.6, 1.1, -1.5, 1.2, 3.2, -1.1], \tag{7.10}$$

the softmax function will normalize it to a probability distribution (shown rounded):

$$\text{softmax}(\mathbf{z}) = [0.055, 0.090, 0.0067, 0.10, 0.74, 0.010] \tag{7.11}$$

Feed-forward neural networks (FFNNs), matrix version



U and W are weights that are learnt during training.

What is this ***b*** term and the extra input value that is always a 1?

Number of layers = Number of learnable layers (2 in this figure)

Quiz 1: Dimensionality

My input x has dimension 100. I want to classify it as one of 4 labels. I designed a 2 layer FFNN.

Q1. Is this dimension information complete to construct the FFNN?

Q2: What is the dimensions of the weight matrices and the bias matrices?

Q3: Which weights are “learnable”?

Putting it together: FFNNs

- Computation proceeds iteratively from one layer of units to the next.
- A feedforward network is a multilayer feedforward network in which the units are connected with no cycles i.e., the outputs from units in each layer are passed to units in the next higher layer, and no outputs are passed back to lower layers.
- Sometimes called multi-layer perceptrons (MLPs); however, units in modern multilayer networks aren't perceptrons (which are linear classifiers)

- Deep learning when the network has many layers
- Logistic regression is a really basic FFNN
- Unlike LR, no feature engineering is needed for FFNNs!!!
- Take dense word embeddings as input, induce the necessary representations as part of learning to classify.
- Downside: need a large amount of training data

Big question: how do we learn the weights?

- In the previous slides, we saw how to derive the final output $\hat{y}$ given an input x and the weights (W and b) for all layers.
- How do we learn these weights?
- Similar strategy to how we learned weights for logistic regression:

Initialize W and b randomly.

Repeat

For each training datapoint (x, y)

1. Feedforward pass to compute output $\hat{y}$
2. Compute error, i.e. loss(y, $\hat{y}$)
3. Update **weights in the network to reduce error.**

Until a stopping criterion is reached.

Return the learned W and b

Big question: how do we learn the weights?

Initialize W and b randomly.

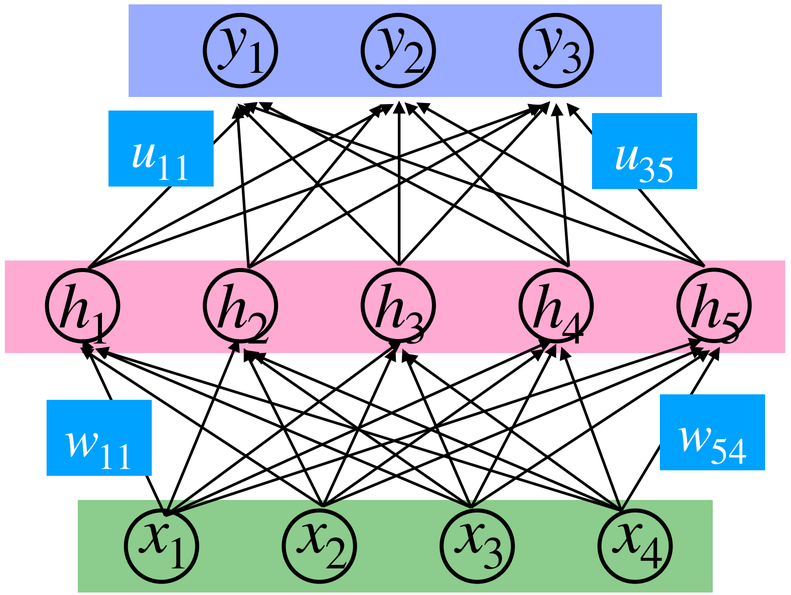
Repeat

For each training datapoint (x, y)

1. Feedforward pass to compute output $\hat{y}$
2. Compute error, i.e. loss(y, $\hat{y}$)
3. Update **weights in the network to reduce error.**

Until a stopping criterion is reached.

Return the learned W and b



- We want to learn weight updates to all "parameters" like $w_{11}, \ldots w_{54} \ldots u_{11} \ldots u_{35}, b_1 \ldots b_n$
- We need to know $\frac{\partial L}{\partial w_{11}}$, etc. to update the weights.

Challenge: We need to compute this derivative wrt weight parameters that appear in very early layers of the network, even though loss is computed at the very end.

Solution: Backpropagation

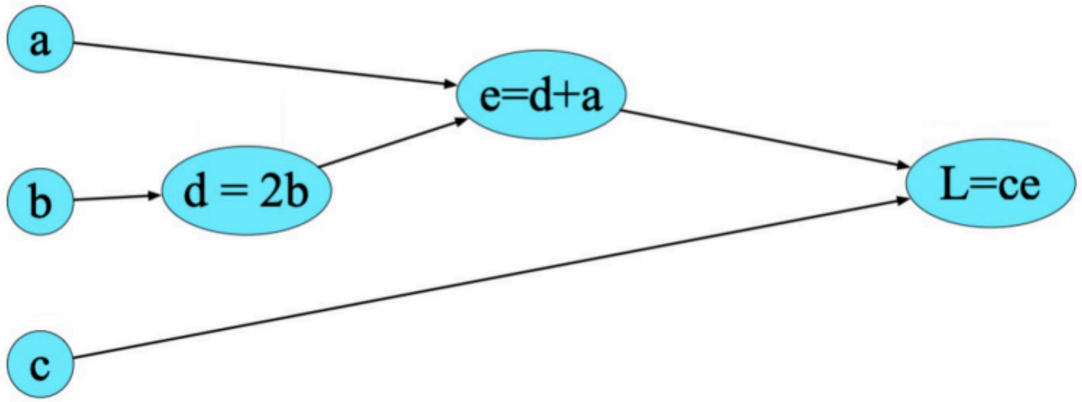
- Computationally efficient algorithm (automatic differentiation)
- Implemented using **computation graphs.**
    - A representation of the process of computing a mathematical expression.
    - Each operation is modeled as a node in a graph.
    - Useful to think of computation graphs as modeling data dependencies

Computational Graphs

Computational Graph Construction

$$L(a,b,c) = c(a+2b)$$

$$\begin{aligned} d &= 2*b \\ e &= a+d \\ L &= c*e \end{aligned}$$



Computational Graphs

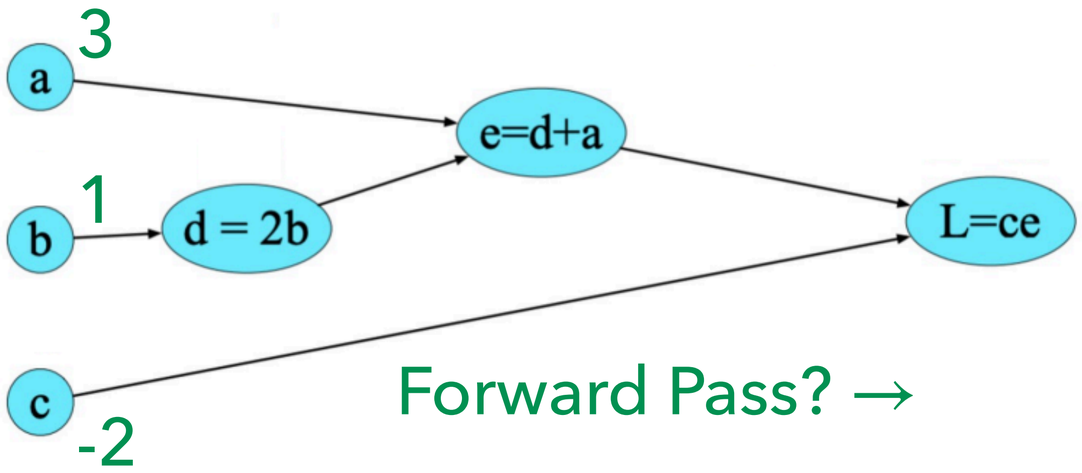
Computational Graph Construction

$$L(a,b,c) = c(a+2b)$$

$$d = 2*b$$

$$e = a+d$$

$$L = c*e$$



Computational Graphs

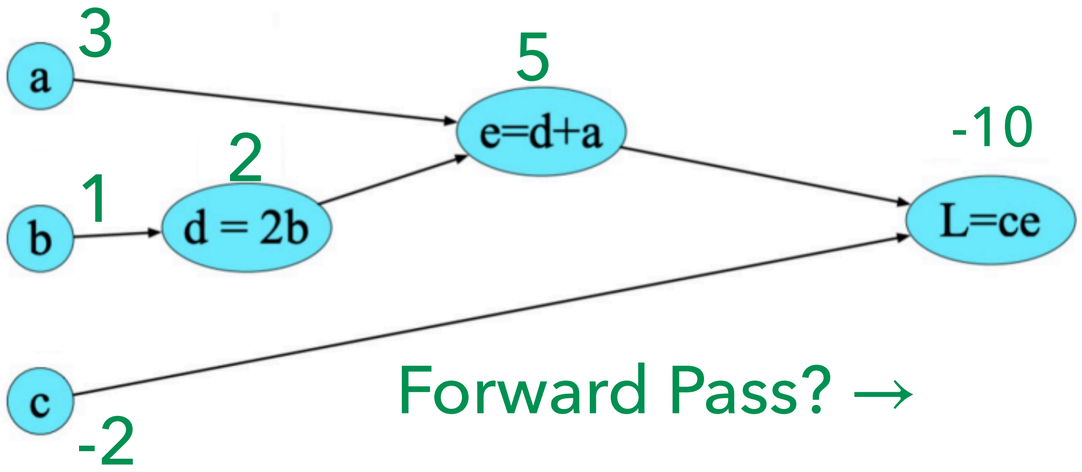
Computational Graph Construction

$$L(a,b,c) = c(a+2b)$$

$$d = 2*b$$

$$e = a+d$$

$$L = c*e$$



How do we compute derivative of L wrt a, b, c?

Main Idea: use chain rule. If $f(x) = u(v(x))$, then $\frac{\partial f}{\partial x} = \frac{\partial u}{\partial v}\frac{\partial v}{\partial x}$

$$\frac{\partial L}{\partial c} = e$$

$$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial a}$$

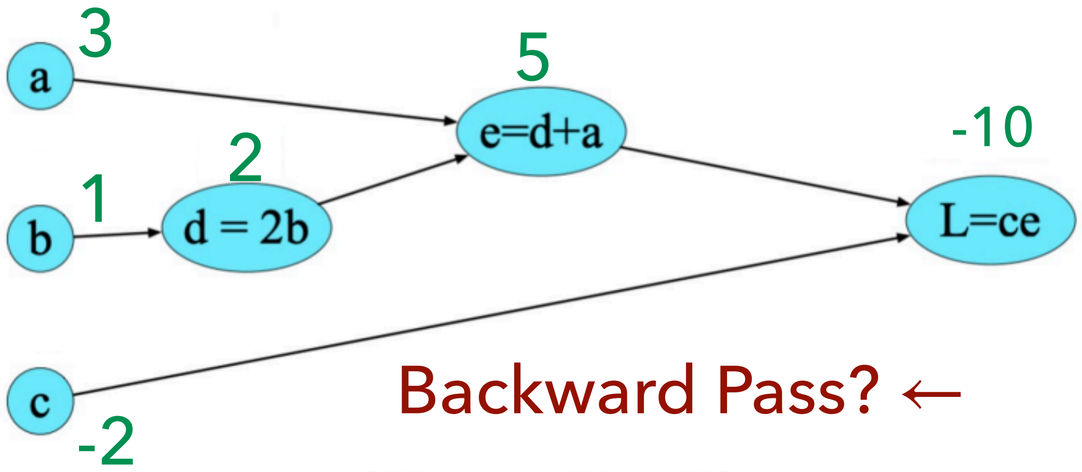
$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial b}$$

Computational Graphs

Computational Graph Construction

$$L(a,b,c) = c(a+2b)$$

$$\begin{aligned} d &= 2*b \\ e &= a+d \\ L &= c*e \end{aligned}$$



Backward Pass? ←

$$\frac{\partial L}{\partial c} = e$$

$$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial a}$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial b}$$

1 layer network for binary classification ~ LR

Backpropagation for a 3 layer NN

Backpropagation in PyTorch

```python
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(784, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 10)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

```python
import torch.optim as optim

net = Net()
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

```python
outputs = net(inputs)
loss = criterion(outputs, labels)
loss.backward()
optimizer.step()
```

PyTorch did back-propagation for you in this one line of code!

Slide Acknowledgements

- Earlier versions of this course offerings including materials from Claire Cardie, Marten van Schijndel, Lillian Lee.
- CS 288 course by Alane Suhr (UC Berkeley).