Natural Language Processing
DSECL ZG565

Dr. Chetana Gavankar, Ph.D,
IIT Bombay-Monash University Australia
Chetana.gavankar@pilani.bits-pilani.ac.in

BITS Pilani
Pilani Campus

BITS Pilani
Pilani Campus

Grateful Acknowledgment: Source of this slide deck

Jurafsky and Martin: https://web.stanford.edu/~jurafsky/slp3/

https://web.stanford.edu/~jurafsky/slp3/slides/7_NN_Apr_28_2021.pptx

Session Content

Neural Networks and Neural Language Models

(Chapter 7 Jurafsky and Martin Book)

- Units in Neural Networks
- The XOR problem
- Feedforward Neural Networks
- Applying Feedforward Neural Networks to NLP tasks
- Training Neural Networks: Overview
- Computation Graphs & Backward Differentiation
- Neural Language Models

This is in your brain

By BruceBlaus - Own work, CC BY 3.0, https://commons.wikimedia.org/w/index.php?curid=28761830

Neural Network Unit

This is not in your brain

Output value

Non-linear transform

Weighted sum

Weights

Input layer

Neural unit

- Take weighted sum of inputs, plus a bias

$$z = b + \sum_i w_i x_i$$

$$z = w \cdot x + b$$

- Instead of just using z, we'll apply a nonlinear activation function f:

$$y = a = f(z)$$

Non-Linear Activation Functions

We've already seen the sigmoid for logistic regression:

Sigmoid

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

Final function the unit is computing

$$y = s(w \cdot x + b) = \frac{1}{1 + \exp(-(w \cdot x + b))}$$

Final unit again

Output value

Non-linear activation function

Weighted sum

Weights

Input layer

An example

- *Suppose a unit has:*
- w = `[0.2,0.3,0.9]`
- b = `0.5`
- What happens with input x:
- x = `[0.5,0.6,0.1]`

$$y = s(w \cdot x + b) =$$

An example

- *Suppose a unit has:*
  - $w = [0.2, 0.3, 0.9]$
  - $b = 0.5$
- What happens with the following input x?
- $x = [0.5, 0.6, 0.1]$

$$y = s(w \cdot x + b) = \frac{1}{1 + e^{-(w \cdot x + b)}} =$$

An example

- *Suppose a unit has:*
- $w = [0.2, 0.3, 0.9]$
- $b = 0.5$
- What happens with input x:
- $x = [0.5, 0.6, 0.1]$

$$y = s(w \cdot x + b) = \frac{1}{1 + e^{-(w \cdot x + b)}} =$$

$$\frac{1}{1 + e^{-(.5 \times .2 + .6 \times .3 + .1 \times .9 + .5)}} =$$

An example

- *Suppose a unit has:*
- $w = [0.2, 0.3, 0.9]$
- $b = 0.5$
- What happens with input x:
- $x = [0.5, 0.6, 0.1]$

$$y = s(w \cdot x + b) = \frac{1}{1 + e^{-(w \cdot x + b)}} =$$

$$\frac{1}{1 + e^{-(.5 \times .2 + .6 \times .3 + .1 \times .9 + .5)}} = \frac{1}{1 + e^{-0.87}} = .70$$

Non-Linear Activation Functions besides sigmoid

tanh

Most Common:

ReLU

Rectified Linear Unit

The XOR problem

Minsky and Papert (1969)

- Can neural units compute simple functions of input?

AND			OR			XOR		
x1	x2	y	x1	x2	y	x1	x2	y
0	0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	1	1
1	0	0	1	0	1	1	0	1
1	1	1	1	1	1	1	1	0

Perceptrons

- A very simple neural unit
- Binary output (0 or 1)
- No non-linear activation function

$$y = \begin{cases} 0, & \text{if } w \cdot x + b \leq 0 \\ 1, & \text{if } w \cdot x + b > 0 \end{cases}$$

Easy to build AND or OR with perceptrons

$$y = \begin{cases} 0, & \text{if } w \cdot x + b \leq 0 \\ 1, & \text{if } w \cdot x + b > 0 \end{cases}$$

AND

AND

x1	x2	y
0	0	0
0	1	0
1	0	0
1	1	1

OR

OR

x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	1

Easy to build AND or OR with perceptrons

$$y = \begin{cases} 0, & \text{if } w \cdot x + b \leq 0 \\ 1, & \text{if } w \cdot x + b > 0 \end{cases}$$

AND

AND

x1	x2	y
0	0	0
0	1	0
1	0	0
1	1	1

OR

OR

x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	1

Easy to build AND or OR with perceptrons

$$y = \begin{cases} 0, & \text{if } w \cdot x + b \leq 0 \\ 1, & \text{if } w \cdot x + b > 0 \end{cases}$$

AND

AND

x1	x2	y
0	0	0
0	1	0
1	0	0
1	1	1

OR

OR

x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	1

Not possible to capture XOR with perceptrons

- Pause the lecture and try for yourself!

Why? Perceptrons are linear classifiers

- Perceptron equation given x_1 and x_2, is the equation of a line
- $w_1x_1 + w_2x_2 + b = 0$
- (in standard linear format: $x_2 = (-w_1/w_2)x_1 + (-b/w_2)$)
- This line acts as a **decision boundary**
- 0 if input is on one side of the line
- 1 if on the other side of the line

Decision boundaries

a) x_1 AND x_2 b) x_1 OR x_2 c) x_1 XOR x_2

XOR is not a **linearly separable** function!

Solution to the XOR problem

- XOR **can't** be calculated by a single perceptron
- XOR **can** be calculated by a layered network of units.

XOR

x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	0

Solution to the XOR problem

- XOR **can't** be calculated by a single perceptron
- XOR **can** be calculated by a layered network of units.

XOR

x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	0

The hidden representation h

a) The original *x* space

b) The new (linearly separable) *h* space

(With learning: hidden layers will learn to form useful representations)

Feedforward Neural Networks

- Can also be called **multi-layer perceptrons** (or **MLPs**) for historical reasons

Binary Logistic Regression as a 1-layer Network

(we don't count the input layer in counting layers!)

Output layer
(σ node)

$$y = \sigma(w \cdot x + b)$$

(y is a scalar)

Input layer
vector x

Fully connected single layer network

Output layer
(softmax nodes)

$$y = \text{softmax}(Wx + b)$$

y is a vector

W is a matrix

b is a vector

Input layer
scalars

softmax: a generalization of sigmoid

- For a vector *z* of dimensionality *k*, the softmax is:

$$\text{softmax}(z) = \left[\frac{\exp(z_1)}{\sum_{i=1}^{k}\exp(z_i)}, \frac{\exp(z_2)}{\sum_{i=1}^{k}\exp(z_i)}, \ldots, \frac{\exp(z_k)}{\sum_{i=1}^{k}\exp(z_i)}\right]$$

$$\text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_{j=1}^{k}\exp(z_j)} \quad 1 \leq i \leq k$$

- Example:

$$z = [0.6, 1.1, -1.5, 1.2, 3.2, -1.1]$$

$$\text{softmax}(z) = [0.055, 0.090, 0.006, 0.099, 0.74, 0.010]$$

Two-Layer Network with scalar output

Output layer
(σ node)

hidden units
(σ node)

Input layer
(vector)

$y = \sigma(z)$ y is a scalar

$z = Uh$

$h = \sigma(Wx + b)$

Could be ReLU
Or tanh

Two-Layer Network with scalar output

Output layer
(σ node)

hidden units
(σ node)

Input layer
(vector)

$y = \sigma(z)$ y is a scalar

$z = Uh$

$h = \sigma(Wx + b)$

b vector

Two-Layer Network with scalar output

Output layer
(σ node)

hidden units
(σ node)

Input layer
(vector)

$y = \sigma(z)$ y is a scalar

$z = Uh$

$h = \sigma(Wx + b)$

Could be ReLU
Or tanh

Two-Layer Network with softmax output

Output layer
(σ node)

hidden units
(σ node)

Input layer
(vector)

$$y = \text{softmax}(z)$$
$$z = Uh$$

y is a vector

$$h = \sigma(Wx + b)$$

Could be ReLU
Or tanh

Multi-layer Notation

$$y = a^{[2]}$$

$$a^{[2]} = g^{[2]}(z^{[2]})$$ sigmoid or softmax

$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$

$$a^{[1]} = g^{[1]}(z^{[1]})$$ ReLU

$$z^{[1]} = W^{[1]}a^{[0]} + b^{[1]}$$

$a^{[0]}$

Multi Layer Notation

$$
\begin{aligned}
z^{[1]} &= W^{[1]}a^{[0]} + b^{[1]} \\
a^{[1]} &= g^{[1]}(z^{[1]}) \\
z^{[2]} &= W^{[2]}a^{[1]} + b^{[2]} \\
a^{[2]} &= g^{[2]}(z^{[2]}) \\
\hat{y} &= a^{[2]}
\end{aligned}
$$

for i **in** 1..n

$$
\begin{aligned}
z^{[i]} &= W^{[i]}\, a^{[i-1]} + b^{[i]} \\
a^{[i]} &= g^{[i]}(z^{[i]}) \\
\hat{y} &= a^{[n]}
\end{aligned}
$$

Replacing the bias unit

- Let's switch to a notation without the bias unit
- Just a notational change

1. Add a dummy node $a_0=1$ to each layer
2. Its weight w_0 will be the bias
3. So input layer $a^{[0]}_0=1$,
   - And $a^{[1]}_0=1$, $a^{[2]}_0=1$,…

Replacing the bias unit

- Instead of:

$$x = x_1, x_2, \ldots, x_{n0}$$

$$h = \sigma(Wx + b)$$

$$h_j = \sigma\left(\sum_{i=1}^{n_0} W_{ji} x_i + b_j\right)$$

We'll do this:

$$x = x_0, x_1, x_2, \ldots, x_{n0}$$

$$h = \sigma(Wx)$$

$$\sigma\left(\sum_{i=0}^{n_0} W_{ji} x_i\right)$$

Replacing the bias unit

Instead of:

We'll do this:

Use cases for feedforward networks

- Let's consider 2 (simplified) sample tasks:

1. Text classification
2. Language modeling

•State of the art systems use more powerful neural architectures, but simple models are useful to consider!

Classification: Sentiment Analysis

- We could do exactly what we did with logistic regression
- Input layer are binary features as before
- Output layer is 0 or 1

Sentiment Features

Var	Definition
x_1	$\text{count}(\text{positive lexicon}) \in \text{doc})$
x_2	$\text{count}(\text{negative lexicon}) \in \text{doc})$
x_3	$\begin{cases} 1 & \text{if } \text{“no”} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$
x_4	$\text{count}(\text{1st and 2nd pronouns} \in \text{doc})$
x_5	$\begin{cases} 1 & \text{if } \text{“!”} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$
x_6	$\log(\text{word count of doc})$

Feedforward nets for simple classification

Logistic Regression

2-layer feedforward network

- Just adding a hidden layer to logistic regression
- allows the network to use non-linear interactions between features
- which may (or may not) improve performance.

Even better: representation learning

- The real power of deep learning comes from the ability to **learn** features from the data
- Instead of using hand-built human-engineered features for classification
- Use learned representations like embeddings!

Neural Net Classification with embeddings as input features!

Issue: texts come in different sizes

- This assumes a fixed size length (3)!
- Kind of unrealistic.
- Some simple solutions (more sophisticated solutions later)

1. Make the input the length of the longest review
   - If shorter then pad with zero embeddings
   - Truncate if you get longer reviews at test time
2. Create a single "sentence embedding" (the same dimensionality as a word) to represent all the words
   - Take the mean of all the word embeddings
   - Take the element-wise max of all the word embeddings
     - For each dimension, pick the max value from all words

Reminder: Multiclass Outputs

- What if you have more than two output classes?
  - Add more output units (one for each class)
  - And use a “softmax layer”

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}} \quad 1 \le i \le D$$

Neural Language Models (LMs)

- **Language Modeling**: Calculating the probability of the next word in a sequence given some history.
- We've seen N-gram based LMs
- But neural network LMs far outperform n-gram language models

•State-of-the-art neural LMs are based on more powerful neural network technology like Transformers

•But **simple feedforward LMs** can do almost as well!

Simple feedforward Neural Language Models

- **Task**: predict next word w_t
- given prior words w_{t-1}, w_{t-2}, w_{t-3}, …
- **Problem**: Now we're dealing with sequences of arbitrary length.
- **Solution**: Sliding windows (of fixed length)

$$P(w_t | w_1^{t-1}) \approx P(w_t | w_{t-N+1}^{t-1})$$

Neural Language Model

Why Neural LMs work better than N-gram LMs

- **Training data:**
- We've seen: I have to make sure that the cat gets fed.
- Never seen: dog gets fed
- **Test data:**
- I forgot to make sure that the dog gets ___
- N-gram LM can't predict "fed"!
- Neural LM can use similarity of "cat" and "dog" embeddings to generalize and predict “fed” after dog

Intuition: training a 2-layer Network

Intuition: Training a 2-layer network

- For every training tuple (x, y)
  - Run *forward* computation to find our estimate $\hat{y}$
  - Run *backward* computation to update weights:
    - For every output node
    - Compute loss L between true y and the estimated $\hat{y}$
    - For every weight w from hidden layer to the output layer
      - » Update the weight
    - For every hidden node
    - Assess how much blame it deserves for the current answer
    - For every weight w from input layer to the hidden layer
      - » Update the weight

Reminder: Loss Function for binary logistic regression

- A measure for how far off the current answer is to the right answer
- Cross entropy loss for logistic regression:

$$L_{\mathrm{CE}}(\hat{y}, y) = -\log p(y|x) \;=\; -[y \log \hat{y} + (1-y)\log(1-\hat{y})]$$

$$= \; -[y \log \sigma(w \cdot x + b) + (1-y)\log(1-\sigma(w \cdot x + b))]$$

Reminder: gradient descent for weight updates

- Use the derivative of the loss function with respect to weights
  $\frac{d}{dw} L(f(x;w), y)$
- To tell us how to adjust weights for each training item
  - Move them in the opposite direction of the gradient

$$w^{t+1} = w^{t} - h \frac{d}{dw} L(f(x;w), y)$$

  - For logistic regression

$$\frac{\partial L_{\mathrm{CE}}(\hat{y}, y)}{\partial w_j} = [\sigma(w \cdot x + b) - y] x_j$$

Where did that derivative come from?

- Using the chain rule! $f(x) = u(v(x))$
- Intuition (see the text for details)

$$\frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dx}$$

Derivative of the weighted sum

Derivative of the Activation

Derivative of the Loss

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial z}\frac{\partial z}{\partial w_i}$$

How can I find that gradient for every weight in the network?

- These derivatives on the prior slide only give the updates for one weight layer: the last one!
- What about deeper networks?
- Lots of layers, different activation functions?
- Solution in the next lecture:
- Even more use of the chain rule!!
- Computation graphs and backward differentiation!

Summary

- For training, we need the derivative of the loss with respect to weights in early layers of the network
- But loss is computed only at the very end of the network!
- Solution: **backward differentiation**
- Given a computation graph and the derivatives of all the functions in it we can automatically compute the derivative of the loss with respect to these early weights.

Neural Language Model

Probabilistic Language Models

- Probability of a sequence of words: $P(W) = P(w_1, w_2, \ldots, w_{t-1}, w_T)$
- **Conditional probability** of an upcoming word: $P(w_T | w_1, w_2, \ldots, w_{t-1})$
- Chain rule of probability: $P(w_1, w_2, \ldots, w_{t-1}, w_T) = P(w_1)P(w_2 | w_1)P(w_3 | w_1, w_2)\ldots P(w_T | w_1, w_2, \ldots, w_T)$
- $(n\text{-}1)^{\text{th}}$ order Markov assumption $P(w_1, w_2, \ldots, w_{t-1}, w_T) = \prod_{t=1}^{T} P(w_t | w_1, w_2, \ldots, w_{t-1})$
- Each $p(w_i | w_{i-4}, w_{i-3}, w_{i-2}, w_{i-1})$ may not have enough statistics to estimate
  - we back off to $p(w_i | w_{i-3}, w_{i-2}, w_{i-1})$, $p(w_i | w_{i-2}, w_{i-1})$, etc., all the way to $p(w_i)$

Neural Probabilistic Language Models

- Instead of treating words as tokens, exploit semantic similarity
  - Learn a distributed representation of words that will allow sentences like these to be seen as similar

    The cat is walking in the bedroom.
    A dog was walking in the room.
    The cat is running in a room.
    The dog was running in the bedroom.
    etc.
  - Use a neural net to represent the conditional probability function $\boldsymbol{P(w_t | w_{t-n}, w_{t-n+1}, \dots, w_{t-1})}$
  - Learn the word representation and the probability function simultaneously

Neural Language Models (LMs)

- **Language Modeling**: Calculating the probability of the next word in a sequence given some history.
- We've seen N-gram based LMs
- But neural network LMs far outperform n-gram language models
- State-of-the-art neural LMs are based on more powerful neural network technology like **Transformers**
- But **simple feedforward LMs** can do almost as well!

Simple feedforward Neural Language Models

- **Task**: predict next word w_t

  given prior words w_{t-1}, w_{t-2}, w_{t-3}, ...
- **Output** : probability distribution over possible next words.

$$P(w_t|w_1,\ldots,w_{t-1}) \approx P(w_t|w_{t-N+1},\ldots,w_{t-1})$$

- **Problem**: Now we're dealing with sequences of arbitrary length.
- **Solution**: Sliding windows (of fixed length)

Word Embeddings

- one-hot vector representation , e.g., dog = (0,0,0,0,1,0,0,0,0,....), cat = (0,0,0,0,0,0,0,1,0,....)
- Represent each of the N previous words as a one-hot vector of length |V| one-hot vector , i.e., with one dimension for each word in the vocabulary
- word "toothpaste", supposing it is V_5, i.e., index 5 in the vocabulary, $x_5 = 1$, and $x_i = 0$ i $\forall \neq 5$,

```
[0 0 0 0 1 0 0 ... 0 0 0 0]
 1 2 3 4 5 6 7 ...  ... |V|
```

Figure 7.12 Selecting the embedding vector for word V_5 by multiplying the embedding matrix **E** with a one-hot vector with a 1 in index 5.

Why Neural LMs work better than N-gram LMs

embeddings

- Neural language models represent words in this prior context by their embeddings, rather than just by their word identity as used in n-gram language models.
- Using embeddings allows neural language models to generalize better to unseen data.

Training data:

- We've seen: I have to make sure that the cat gets fed.
- Never seen: dog gets fed

Test data:

- I forgot to make sure that the dog gets ___
- N-gram LM can't predict "fed"!
- Neural LM can use similarity of "cat" and "dog" embeddings to generalize and predict “fed” after dog

Neural Language Model

Equations:

$$
\begin{aligned}
\mathbf{e} &= [\mathbf{E}\mathbf{x}_{t-3}; \mathbf{E}\mathbf{x}_{t-2}; \mathbf{E}\mathbf{x}_{t-1}] \\
\mathbf{h} &= \sigma(\mathbf{W}\mathbf{e} + \mathbf{b}) \\
\mathbf{z} &= \mathbf{U}\mathbf{h} \\
\hat{\mathbf{y}} &= \mathrm{softmax}(\mathbf{z})
\end{aligned}
$$

concatenate 3 embeddings for the 3 context words to produce the embedding layer e

Training the neural language model

- Two ways:
- Freeze the embedding layer E with initial word2vec values.
  - Freezing means we use **word2vec or some other pretraining algorithm** to compute the initial embedding matrix E, and then hold it constant while we only Modify W, U, and b, i.e., we don't update E during language model training
- Learn the embeddings simultaneously with training the network.
  - Useful when the task the network is designed for (like sentiment classification, translation, or parsing) places strong constraints on what makes a good representation for words.

Training the neural language model

Figure 7.18 Learning all the way back to embeddings. Again, the embedding matrix **E** is

θ= E,W,U, b.

one single embedding dictionary E that's shared among one-hot vectors

Training the parameters to minimize loss will result both in an algorithm for
language modeling (a word predictor)
but also a **new set of embeddings E**
that can be used as word representations
for other tasks.

N-gram vs NLM

- Compared to n-gram models, NLM can
  - handle much longer histories
  - generalize better over contexts of similar words,
  - Are more accurate at word-prediction.
- NLM are
  - much more complex,
  - slower
  - need more energy to train,
  - less interpretable than n-gram models, so for many (especially smaller) tasks an n-gram language model is still the right tool.

References

CH-7 - Speech and Language Processing by Daniel Jurafsky

https://www.youtube.com/watch?v=BtmsIy0j_dY&t=5s

Why Computation Graphs

- For training, we need the derivative of the loss with respect to each weight in every layer of the network
- But the loss is computed only at the very end of the network!
- Solution: **error backpropagation** (Rumelhart, Hinton, Williams, 1986)
- **Backprop** is a special case of **backward differentiation**
- Which relies on **computation graphs**.

Computation Graphs

- A computation graph represents the process of computing a mathematical expression

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

Computations:

$$d = 2*b$$
$$e = a+d$$
$$L = c*e$$

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

Computations:

$$d = 2*b$$
$$e = a+d$$
$$L = c*e$$

Backwards differentiation in computation graphs

- The importance of the computation graph comes from the backward pass
- This is used to compute the derivatives that we'll need for the weight update.

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

$$d = 2*b$$

$$e = a+d$$

$$L = c*e$$

We want: $\frac{\partial L}{\partial a}$, $\frac{\partial L}{\partial b}$, and $\frac{\partial L}{\partial c}$

The derivative $\frac{\partial L}{\partial a}$, tells us how much a small change in *a* affects *L*.

The chain rule

- Computing the derivative of a composite function:

- $f(x) = u(v(x))$ $$\frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dx}$$

- $f(x) = u(v(w(x)))$ $$\frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dw} \cdot \frac{dw}{dx}$$
-

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

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

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

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

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

Example

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

$$d = 2*b$$

$$e = a+d$$

$$L = c*e$$

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

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

$$L = ce \ : \quad \frac{\partial L}{\partial e} = c, \frac{\partial L}{\partial c} = e$$

$$e = a+d \ : \quad \frac{\partial e}{\partial a} = 1, \frac{\partial e}{\partial d} = 1$$

$$d = 2b \ : \quad \frac{\partial d}{\partial b} = 2$$

Example

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

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

$$L = ce \ : \quad \frac{\partial L}{\partial e} = c, \frac{\partial L}{\partial c} = e$$

$$e = a + d \ : \quad \frac{\partial e}{\partial a} = 1, \frac{\partial e}{\partial d} = 1$$

$$d = 2b \ : \quad \frac{\partial d}{\partial b} = 2$$

Example

a=3
a
∂L/∂a=-2
∂e/∂a=1
e=5
e=d+a
b=1
b
d=2
d = 2b
∂e/∂d =1
∂L/∂e=-2
∂L/∂e=-2
L=-10
L=ce
∂L/∂b=-4
∂d/∂b=2
∂L/∂d=-2
∂L/∂c =5
c=-2
c
backward pass
∂L/∂c=5

Backward differentiation on a two layer network

$$
\begin{aligned}
z^{[1]} &= W^{[1]}\mathbf{x} + b^{[1]} \\
a^{[1]} &= \mathrm{ReLU}(z^{[1]}) \\
z^{[2]} &= W^{[2]}a^{[1]} + b^{[2]} \\
a^{[2]} &= \boldsymbol{\sigma}(z^{[2]}) \\
\hat{y} &= a^{[2]}
\end{aligned}
$$

Backward differentiation on a two layer network

$$
\begin{aligned}
z^{[1]} &= W^{[1]}\mathbf{x} + b^{[1]} \\
a^{[1]} &= \mathrm{ReLU}(z^{[1]}) \\
z^{[2]} &= W^{[2]}a^{[1]} + b^{[2]} \\
a^{[2]} &= \sigma(z^{[2]}) \\
\hat{y} &= a^{[2]}
\end{aligned}
$$

$$\frac{d\,\mathrm{ReLU}(z)}{dz} = \begin{cases} 0 & for \quad z < 0 \\ 1 & for \quad z \geq 0 \end{cases}$$

$$\frac{d\sigma(z)}{dz} = \sigma(z)(1 - \sigma(z))$$

Backward differentiation on a 2-layer network

$$\frac{d\,\mathrm{ReLU}(z)}{dz} = \begin{cases} 0 & for \quad z < 0 \\ 1 & for \quad z \geq 0 \end{cases}$$

$$\frac{d\sigma(z)}{dz} = \sigma(z)(1 - \sigma(z))$$

$$\frac{d\,\mathrm{ReLU}(z)}{dz} = \begin{cases} 0 & for \quad z < 0 \\ 1 & for \quad z \geq 0 \end{cases}$$

$$z^{[1]} = W^{[1]}\mathbf{x} + b^{[1]}$$
$$a^{[1]} = \mathrm{ReLU}(z^{[1]})$$
$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$
$$a^{[2]} = \sigma(z^{[2]})$$
$$\hat{y} = a^{[2]}$$

Starting off the backward pass: $\frac{\partial L}{\partial z}$

(I'll write a for $a^{[2]}$ and z for $z^{[2]}$)

$$L(\hat{y}, y) = -(\mathrm{y}\log(\hat{y}) + (1-y)\log(1-\hat{y}))$$

$$L(a, y) = -(\mathrm{y}\log a + (1-y)\log(1-a))$$

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

$$\frac{\partial L}{\partial a} = -\left(\left(y\frac{\partial \log(a)}{\partial a}\right) + (1-\mathrm{y})\frac{\partial \log(1-a)}{\partial a}\right)$$

$$= -\left(\left(y\frac{1}{a}\right) + (1-\mathrm{y})\frac{1}{1-a}(-1)\right) = -\left(\frac{y}{a} + \frac{y-1}{1-a}\right)$$

$$\frac{\partial a}{\partial z} = a(1-a) \qquad \frac{\partial L}{\partial z} = -\left(\frac{y}{a} + \frac{y-1}{1-a}\right)a(1-a) = \mathrm{a} - \mathrm{y}$$