

# Natural Language Understanding, Generation, and Machine Translation

## Lecture 4: Feedforward Language Models

---

Alexandra Birch

22 January 2024 (week 2)

School of Informatics  
University of Edinburgh  
[a.birch@ed.ac.uk](mailto:a.birch@ed.ac.uk)

Based on slides by Adam Lopez.

# Overview

Representing  $n$ -gram probabilities with neural networks

Input and output: one-hot vectors and the softmax

Logistic Regression

Learning and Objective Functions

Gradient Descent

Word Embeddings

Reading: Sections 4 and 5 of [Neubig \(2017\)](#).

# What We've Seen So Far, and Agenda for Today

**Lecture 3.** Given finite vocabulary  $V$ , we want to define a probability distribution  $P : V^* \rightarrow \mathbb{R}_+$ .

Let  $w$  be a sequence of words in  $V^*$ . Let  $|w|$  be its length and let  $w_i$  be its  $i$ th word. So,  $w = w_1 \dots w_{|w|}$ . Using the chain rule and Markov assumptions, we get:

$$P(w_1 \dots w_{|w|}) \approx \prod_{i=1}^{|w|+1} P(w_i \mid w_{i-n+1}, \dots, w_{i-1})$$

# What We've Seen So Far, and Agenda for Today

**Lecture 3.** Probability theory requires  $P(w_i \mid w_{i-n+1}, \dots, w_{i-1})$  to obey these constraints:

Probabilities are non-negative

$$P : V \rightarrow \mathbb{R}_+$$

...and sum to one

$$\sum_{w \in V} P(w \mid w_{i-n+1}, \dots, w_{i-1}) = 1$$

*Any function  $P$  that satisfies these constraints is valid!*

# What We've Seen So Far, and Agenda for Today

**Lecture 3.** Probability theory requires  $P(w_i \mid w_{i-n+1}, \dots, w_{i-1})$  to obey these constraints:

Probabilities are non-negative

$$P : V \rightarrow \mathbb{R}_+$$

...and sum to one

$$\sum_{w \in V} P(w \mid w_{i-n+1}, \dots, w_{i-1}) = 1$$

*Any function  $P$  that satisfies these constraints is valid!*

**ML/AML/MLPR/MLP.** The perceptron is a non-linear model with a simple learning algorithm. The multilayer perceptron is a universal function approximator.

# What We've Seen So Far, and Agenda for Today

**Lecture 3.** Probability theory requires  $P(w_i \mid w_{i-n+1}, \dots, w_{i-1})$  to obey these constraints:

Probabilities are non-negative

$$P : V \rightarrow \mathbb{R}_+$$

...and sum to one

$$\sum_{w \in V} P(w \mid w_{i-n+1}, \dots, w_{i-1}) = 1$$

*Any function  $P$  that satisfies these constraints is valid!*

**ML/AML/MLPR/MLP.** The perceptron is a non-linear model with a simple learning algorithm. The multilayer perceptron is a universal function approximator.

**This lecture.** How can we use multilayer perceptrons to learn  $P(w_i \mid w_{i-n+1}, \dots, w_{i-1})$ ?

## Representing $n$ -gram probabilities with neural networks

---

# Think carefully about the form of $P$

Our constraints again:

Probabilities are non-negative

$$P : V \rightarrow \mathbb{R}_+$$

...and sum to one

$$\sum_{w \in V} P(w \mid w_{i-n+1}, \dots, w_{i-1}) = 1$$

This formulation says we have many different functions  $P$ , one for each  $w_{i-n+1} \dots w_{i-1} \in V^{n-1}$ . In an  $n$ -gram model, these functions are tables of parameters, and share nothing (unless we use smoothing or backoff).

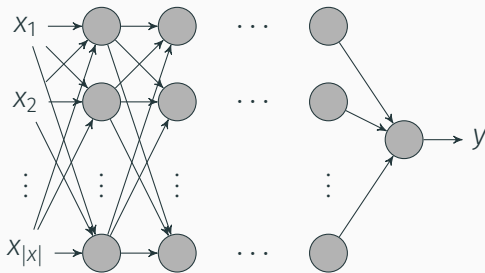
Alternatively, we can view  $P$  as a function from  $n$ -gram histories to distributions over  $V$ . That is,  $P : V^{n-1} \rightarrow (V \rightarrow \mathbb{R}_+)$ .

Constraints still hold!



## What is the input and output of function $P$ ?

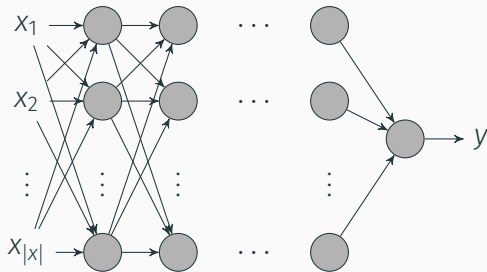
Let's work with  $P : V^{n-1} \rightarrow (V \rightarrow \mathbb{R}_+)$ . How do we implement this using a perceptron? What is the input  $\mathbf{x}$  and the output  $y$ ?





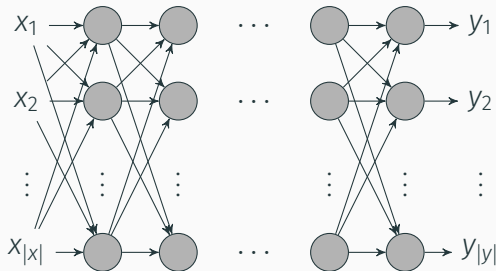
## What is the input and output of function $P$ ?

**Insight.** Multilayer perceptrons aka *neural networks* have *exactly one data type*: vectors. The input  $\mathbf{x}$  is a vector. Each hidden layer  $\mathbf{h}$  is a vector. The output  $y$  is a vector, even if it is a single number



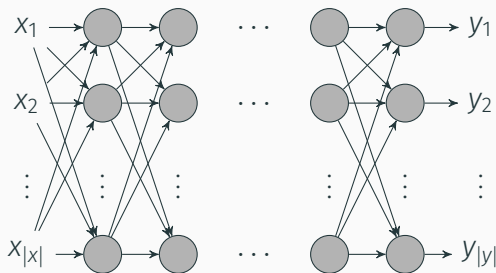
## What is the input and output of function $P$ ?

**Insight.** Multilayer perceptrons aka *neural networks* have *exactly one data type*: vectors. The input  $\mathbf{x}$  is a vector. Each hidden layer  $\mathbf{h}$  is a vector. The output  $y$  is a vector, even if it is a single number ...but it doesn't have to be!



# What is the input and output of function $P$ ?

**Insight.** Multilayer perceptrons aka *neural networks* have *exactly one data type*: vectors. The input  $\mathbf{x}$  is a vector. Each hidden layer  $\mathbf{h}$  is a vector. The output  $\mathbf{y}$  is a vector, even if it is a single number ...but it doesn't have to be!



**Consequence.** If we can describe our input and output as vectors, a neural network can model the mapping between them.

## Discrete symbols from a finite vocabulary are vectors!

If  $V$  is a finite set of (ordered) symbols, and  $w$  its  $i$ th element, then the *one-hot encoding* of  $w$  is a vector with  $|V|$  elements, in which all elements are 0 except for the  $i$ th element, which is 1.

## Discrete symbols from a finite vocabulary are vectors!

If  $V$  is a finite set of (ordered) symbols, and  $w$  its  $i$ th element, then the *one-hot encoding* of  $w$  is a vector with  $|V|$  elements, in which all elements are 0 except for the  $i$ th element, which is 1.

**Example.** Suppose  $V = \{\text{is, cold, grey, hot, summer, winter}\}$

Index  $i$    Symbol  $V_i$

0	is
1	cold
2	grey
3	hot
4	summer
5	winter

## Discrete symbols from a finite vocabulary are vectors!

If  $V$  is a finite set of (ordered) symbols, and  $w$  its  $i$ th element, then the *one-hot encoding* of  $w$  is a vector with  $|V|$  elements, in which all elements are 0 except for the  $i$ th element, which is 1.

**Example.** Suppose  $V = \{\text{is, cold, grey, hot, summer, winter}\}$

Index $i$	Symbol $V_i$	One-hot encoding of...
		summer
0	is	0
1	cold	0
2	grey	0
3	hot	0
4	summer	1
5	winter	0



## Discrete symbols from a finite vocabulary are vectors!

If  $V$  is a finite set of (ordered) symbols, and  $w$  its  $i$ th element, then the *one-hot encoding* of  $w$  is a vector with  $|V|$  elements, in which all elements are 0 except for the  $i$ th element, which is 1.

**Example.** Suppose  $V = \{\text{is, cold, grey, hot, summer, winter}\}$

Index $i$	Symbol $V_i$	One-hot encoding of...	
		summer	is
0	is	0	1
1	cold	0	0
2	grey	0	0
3	hot	0	0
4	summer	1	0
5	winter	0	0

## Discrete symbols from a finite vocabulary are vectors!

If  $V$  is a finite set of (ordered) symbols, and  $w$  its  $i$ th element, then the *one-hot encoding* of  $w$  is a vector with  $|V|$  elements, in which all elements are 0 except for the  $i$ th element, which is 1.

**Example.** Suppose  $V = \{\text{is, cold, grey, hot, summer, winter}\}$

Index $i$	Symbol $V_i$	One-hot encoding of...		
		summer	is	hot
0	is	0	1	0
1	cold	0	0	0
2	grey	0	0	0
3	hot	0	0	1
4	summer	1	0	0
5	winter	0	0	0

## Discrete symbols from a finite vocabulary are vectors!

If  $V$  is a finite set of (ordered) symbols, and  $w$  its  $i$ th element, then the *one-hot encoding* of  $w$  is a vector with  $|V|$  elements, in which all elements are 0 except for the  $i$ th element, which is 1.

**Example.** Suppose  $V = \{\text{is, cold, grey, hot, summer, winter}\}$

Index $i$	Symbol $V_i$	One-hot encoding of...			
		summer	is	hot	winter
0	is	0	1	0	0
1	cold	0	0	0	0
2	grey	0	0	0	0
3	hot	0	0	1	0
4	summer	1	0	0	0
5	winter	0	0	0	1

## Discrete symbols from a finite vocabulary are vectors!

If  $V$  is a finite set of (ordered) symbols, and  $w$  its  $i$ th element, then the *one-hot encoding* of  $w$  is a vector with  $|V|$  elements, in which all elements are 0 except for the  $i$ th element, which is 1.

**Example.** Suppose  $V = \{\text{is, cold, grey, hot, summer, winter}\}$

Index $i$	Symbol $V_i$	One-hot encoding of...				
		summer	is	hot	winter	is
0	is	0	1	0	0	1
1	cold	0	0	0	0	0
2	grey	0	0	0	0	0
3	hot	0	0	1	0	0
4	summer	1	0	0	0	0
5	winter	0	0	0	1	0

## Discrete symbols from a finite vocabulary are vectors!

If  $V$  is a finite set of (ordered) symbols, and  $w$  its  $i$ th element, then the *one-hot encoding* of  $w$  is a vector with  $|V|$  elements, in which all elements are 0 except for the  $i$ th element, which is 1.

**Example.** Suppose  $V = \{\text{is, cold, grey, hot, summer, winter}\}$

Index $i$	Symbol $V_i$	One-hot encoding of...				
		summer	is	hot	winter	is
0	is	0	1	0	0	1
1	cold	0	0	0	0	0
2	grey	0	0	0	0	0
3	hot	0	0	1	0	0
4	summer	1	0	0	0	0
5	winter	0	0	0	1	0

## Discrete symbols from a finite vocabulary are vectors!



If  $V$  is a finite set of (ordered) symbols, and  $w$  its  $i$ th element, then the *one-hot encoding* of  $w$  is a vector with  $|V|$  elements, in which all elements are 0 except for the  $i$ th element, which is 1.

**Example.** Suppose  $V = \{\text{is, cold, grey, hot, summer, winter}\}$

Index $i$	Symbol $V_i$	One-hot encoding of...				
		summer	is	hot	winter	is
0	is	0	1	0	0	1
1	cold	0	0	0	0	0
2	grey	0	0	0	0	0
3	hot	0	0	1	0	0
4	summer	1	0	0	0	0
5	winter	0	0	0	1	0

To get “winter is”, concatenate:  $[0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0]$

## Probability distributions are vectors!

is		0.01
cold		0.60
grey		0.27
hot		0.10
summer		0.01
winter		0.01

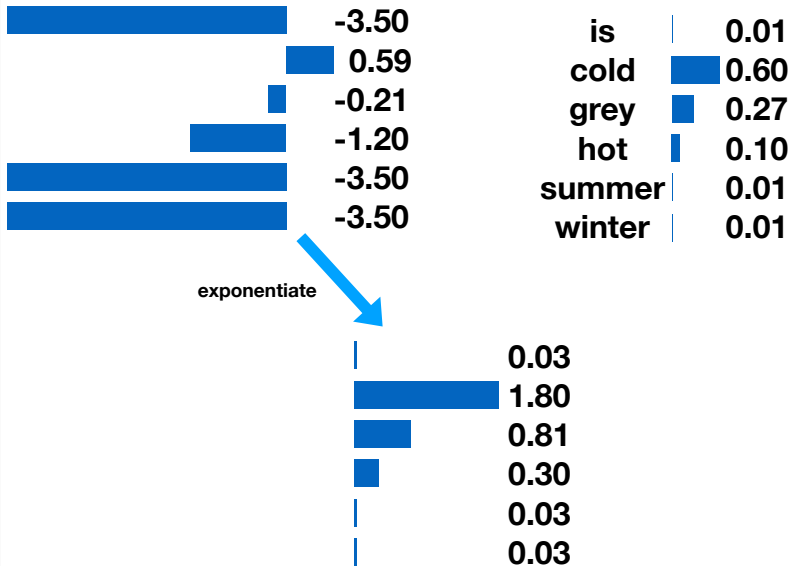
# Probability distributions are vectors!



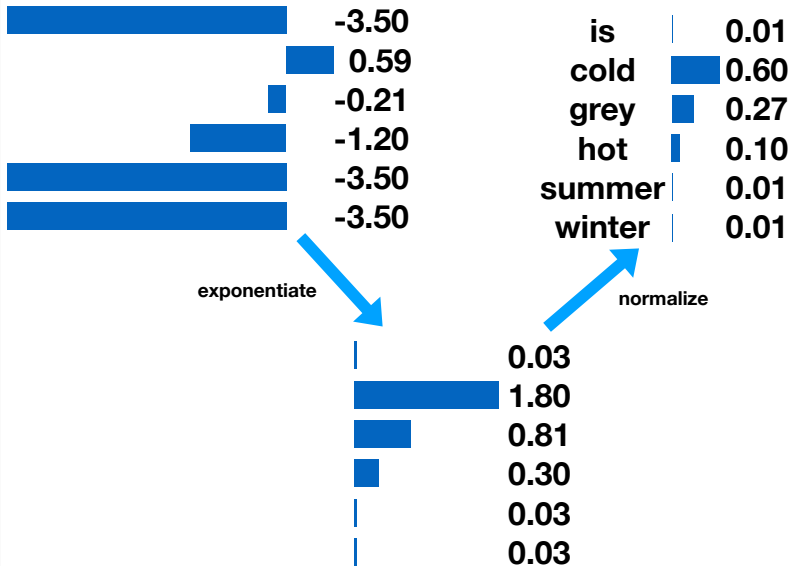
is	0.01
cold	0.60
grey	0.27
hot	0.10
summer	0.01
winter	0.01



# Probability distributions are vectors!



# Probability distributions are vectors!



# Softmax turns any vector into a probability distribution

More formally, here's how to convert a vector into a probability distribution using the *softmax* function:

**Input.** A vector  $\mathbf{v}$  of length  $|\mathbf{v}|$ .

**Output.**  $\text{softmax}(\mathbf{v})$  is a distribution  $\mathbf{u}$  over  $|\mathbf{v}|$ , where:

$$u_i = \frac{\exp v_i}{\sum_{j=1}^{|\mathbf{v}|} \exp v_j}$$

# Softmax turns any vector into a probability distribution

More formally, here's how to convert a vector into a probability distribution using the *softmax* function:

**Input.** A vector  $\mathbf{v}$  of length  $|\mathbf{v}|$ .

**Output.**  $\text{softmax}(\mathbf{v})$  is a distribution  $\mathbf{u}$  over  $|\mathbf{v}|$ , where:

$$u_i = \frac{\exp v_i}{\sum_{j=1}^{|\mathbf{v}|} \exp v_j}$$

Have you seen this function before?

# Logistic regression is softmax over a linear layer

Now that we know how to represent input and output, we have our first neural  $n$ -gram model: *logistic regression!*

**Input.** To get  $\mathbf{x}$ , concatenate the one-hot encodings of the history words  $w_{i-n+1}, \dots, w_{i-1}$ , and binary variables for any other *features* we can think of, e.g. does the word end in “able”? Most words that do are adjectives, indicating a noun will follow. So, we can explicitly engineer this function as part of the input *representation*, and it will most likely improve the model.

**Output.** A distribution  $P : V \rightarrow \mathbb{R}_+$ .

$$\mathbf{y} = \text{softmax}(\mathbf{W}\mathbf{x} + \mathbf{b})$$

# Interpreting logistic regression

$$\mathbf{y} = \text{softmax}(\mathbf{W}\mathbf{x} + \mathbf{b})$$

- Matrix  $\mathbf{W}$  and vector  $\mathbf{b}$  are *parameters* of the model.
- Input  $\mathbf{x}$  and output  $\mathbf{y}$  can have different dimension. These determine the dimension of  $\mathbf{W}$  and  $\mathbf{b}$ :
  - $\mathbf{W}$  must be a  $|\mathbf{y}| \times |\mathbf{x}|$  matrix. Element  $W_{ij}$  of  $\mathbf{W}$  relates the  $j$ th feature in  $\mathbf{x}$  to the  $i$ th outcome in  $\mathbf{y}$ .
  - $\mathbf{b}$ , the *bias*, must be a  $|\mathbf{y}|$ -dimensional vector. The  $i$ th element  $b_i$  of  $\mathbf{b}$  allows the model to express an input-independent (dis)preference for the  $i$ th element of  $\mathbf{y}$ .

# Learning and Objective Functions

---

## There are multiple views of learning

Suppose we see the trigram “winter is cold” in our training data. What does learning look like? How do we update the model?



# There are multiple views of learning

Suppose we see the trigram “winter is cold” in our training data. What does learning look like? How do we update the model?

**View 1.** Choose parameters to *maximize likelihood*, as when estimating an  $n$ -gram model.

# There are multiple views of learning

Suppose we see the trigram “winter is cold” in our training data. What does learning look like? How do we update the model?

**View 1.** Choose parameters to *maximize likelihood*, as when estimating an  $n$ -gram model.

**View 2.** Update parameters to *minimize error*, as in the perceptron.

Index $i$	Symbol $V_i$	$x_1 \dots x_6$	$x_7 \dots x_{12}$	target $t$	output $o$
0	is	0	1	0	0.01
1	cold	0	0	1	0.60
2	grey	0	0	0	0.27
3	hot	0	0	0	0.10
4	summer	0	0	0	0.01
5	winter	1	0	0	0.01

## Both views yield the same objective function!

The maximum likelihood view attempts to put as much probability as possible on the target output.

The neural network view minimizes the *cross-entropy loss*, which penalizes the model for the proportion of the output probability mass that it does not assign to the target output.

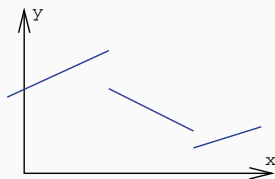
These do the same thing!

# Minimize error by gradient descent

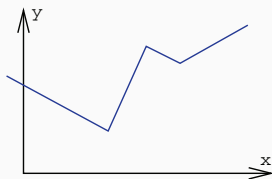
Interpret the error  $E$  just as a mathematical function depending on  $\mathbf{W}$  and  $\mathbf{b}$  and forget about its semantics. We are faced with a problem of mathematical optimization:

$$\underset{\mathbf{W}, \mathbf{b}}{\text{minimize}} E(\mathbf{W}, \mathbf{b})$$

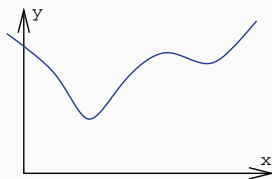
Conveniently, our functions are continuous and differentiable.



non continuous function  
(disrupted)



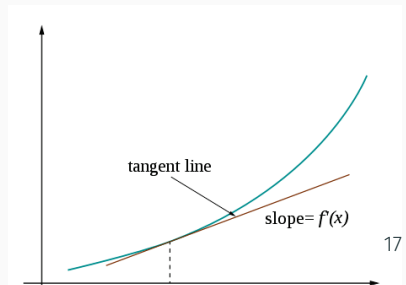
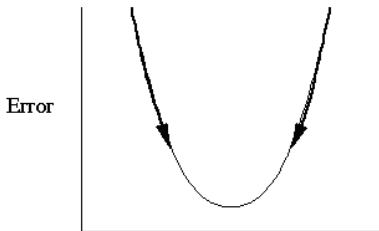
continuous, non differentiable  
function (folded)



differentiable function  
(smooth)

# Gradient and Derivatives: The Idea

- **Gradient descent** can be used for minimizing functions.
- The derivative is **a measure of the rate of change of a function**, as its input changes;
- For function  $y = f(x)$ , the derivative  $\frac{dy}{dx}$  indicates how much  $y$  changes in response to changes in  $x$ .
- If  $x$  and  $y$  are real numbers, and if the graph of  $y$  is plotted against  $x$ , the derivative measures the **slope** or **gradient** of the line at each point, i.e., it describes the steepness or incline



# Gradient and Derivatives: The Idea

- So, we know how to use derivatives to **adjust one input** value.
- But we have **several weights** to adjust!
- We need to use **partial derivatives**.
- A partial derivative of a function of several variables is its derivative with respect to one of those variables, with the others held constant.

## Example

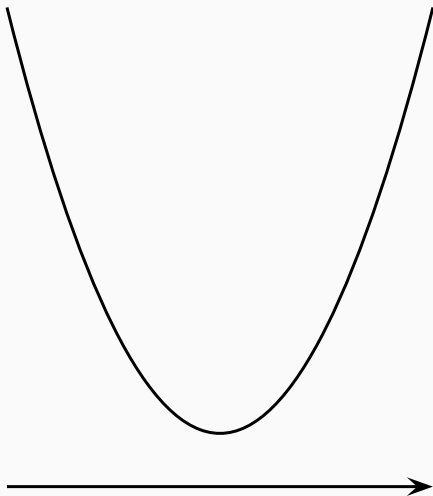
If  $y = f(x_1, x_2)$ , then we can have  $\frac{\partial y}{\partial x_1}$  and  $\frac{\partial y}{\partial x_2}$ .

Given partial derivatives, update the weights:

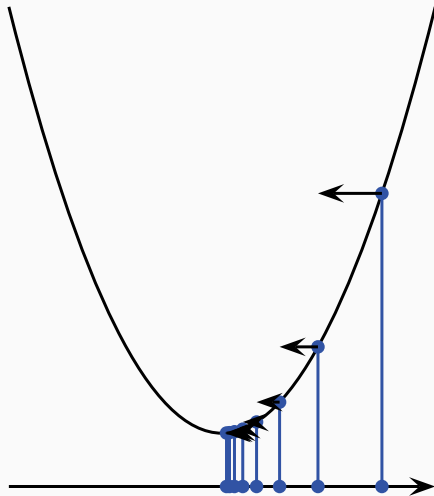
$$w'_{ij} = w_{ij} + \Delta w_{ij}$$

$$\text{where } \Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}}$$

The learning rate must be set carefully



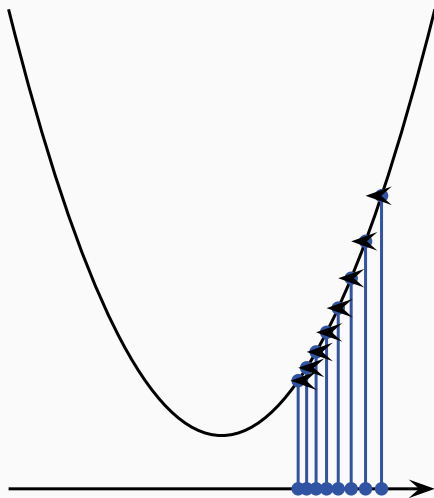
## The learning rate must be set carefully



Small  $\eta$  leads to convergence.

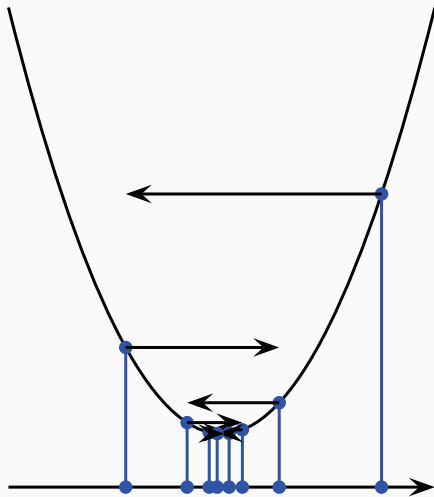


## The learning rate must be set carefully



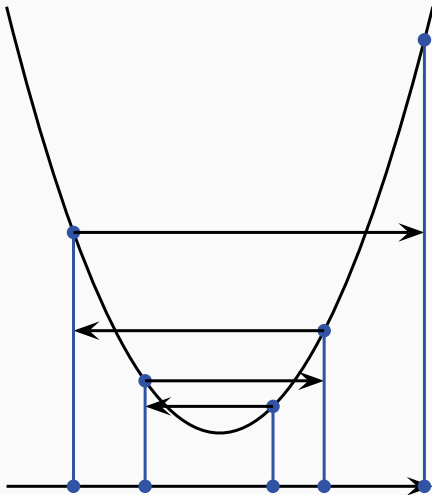
Very small  $\eta$ , convergence may take very long.

## The learning rate must be set carefully



Case of medium size  $\eta$ , also converges.

## The learning rate must be set carefully



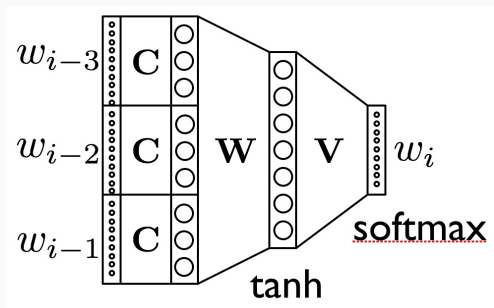
Very large  $\eta$ : divergence.

# Gradient descent is the baseline for many learning algorithms

- Pure gradient descent is a nice theoretical framework but of limited power in practice.
- Finding the right  $\eta$  is annoying. Approaching the minimum is time consuming.
- Heuristics to overcome problems of gradient descent:
  - gradient descent with momentum
  - individual learning rates for each dimension
  - adaptive learning rates
  - decoupling step length from partial derivatives

In logistic regression, it conveniently finds the minimum, because error surface is convex. However ...

## $n$ -grams with a multilayer neural network



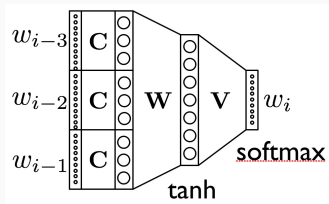
$$P(w_i \mid w_{i-3}, w_{i-2}, w_{i-1}) = \text{softmax}(\mathbf{V}\mathbf{h}_2 + \mathbf{b}_2)$$

$$\mathbf{h}_2 = \tanh(\mathbf{W}\mathbf{h}_1 + \mathbf{b}_1)$$

$$\mathbf{h}_1 = \mathbf{C}w_{i-3}; \mathbf{C}w_{i-2}; \mathbf{C}w_{i-1}$$

$$\mathbf{w}_i = \text{onehot}(w_i)$$

# Interpreting the feedforward $n$ -gram model



- Each hidden layer is a *representation* of its input.
- Multiplying one-hot by  $C$  selects a row of  $C$  (a vector).
- We call  $i$ th row of  $C$  the *embedding* of  $i$ th word in  $V$ .
- Learned *word embeddings* replace hand-engineered features of logistic regression.
- Output dimension of  $C$  (embedding size) can be small (dense), unlike the large (sparse) representations of logistic regression.
- Output of  $W$  represents the full  $n$ -gram history.

# Learning in feedforward neural networks

**General Idea:** same as in a simple perceptron

1. Take the first input pattern  $x$  from the **training set**.
2. Get the output  $y$ .
3. Compare  $y$  with the “right answer” (target  $t$ ) to get the error and compute the **weight updates**.
4. Repeat with the next  $x$  from the training set.
5. Sum up the weight updates for all patterns in the training set and **modify the weights of the network**.
6. Repeat from step 1 until the overall error is sufficiently small.

# Learning in feedforward neural networks

**General Idea:** same as in a simple perceptron

1. Take the first input pattern  $x$  from the **training set**.
2. Get the output  $y$ .
3. Compare  $y$  with the “right answer” (target  $t$ ) to get the error and compute the **weight updates**.
4. Repeat with the next  $x$  from the training set.
5. Sum up the weight updates for all patterns in the training set and **modify the weights of the network**.
6. Repeat from step 1 until the overall error is sufficiently small.

We need to process the whole training set before making and update. This often means convergence is slow.



# Stochastic (i.e. Online) Gradient Descent

```
1: Initialize all weights to small random values.  
2: repeat  
3:   for each training example do  
4:     Compute error gradients to yield update  
        $\Delta w_{ij}$  for all weights  $w_{ij}$ .  
5:     Update the weights using  $\Delta w_{ij}$ .  
6:   end for  
7: until stopping criteria reached.
```

- Common variant: update weights for *minibatches* of examples.
- Automatic differentiation is a subroutine of SGD.

## Summary: neural networks, SGD, and learning

This class is about NLP, not ML, so lectures focus on intuition. Details are in the **required reading**, and there are many alternative tutorials—find one that works for you. Take time to understand learning, but course will mostly focus on model design, assuming learning is automated.

Ideas that generalize to many models:

- Learning requires gradients, but you rarely need to compute them by hand (you will in coursework 1). Deep learning toolkits compute them for you with *automatic differentiation*.
- SGD has a very large number of variants, also in toolkits.
- In general: can simply specify model, let toolkits deal with gradient computation and learning. Very powerful!

# Summary

- We can represent  $n$ -gram probabilities with a neural network, using *one-hot* vectors for input words and the *softmax* function to guarantee that output is a probability distribution.
- We can learn the model using *stochastic gradient descent* to minimize *cross-entropy loss*, equivalent to maximum likelihood for this model.
- Hidden layers of the model learn representations of the input words: *word embeddings*.
- Any conditional probability distribution over discrete variables can be parameterized using a similar strategy.

**Next lecture.** We will use neural networks to remove Markov independence assumptions of language models!

Neubig, G. (2017). Neural machine translation and sequence-to-sequence models: A tutorial.