

Natural Language Understanding

Lecture 5: n -gram language modeling with feedforward neural networks

Adam Lopez

23 January 2018

School of Informatics
University of Edinburgh
`alopez@inf.ed.ac.uk`

What we have seen so far...

Lecture 3. Given a 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, $w = w_1 \dots w_{|w|}$.

What we have seen so far...

Lecture 3. Given a 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, $w = w_1 \dots w_{|w|}$.

Use the chain rule and a Markov assumption to define this in terms of conditional probabilities $P(w_i \mid w_{i-n+1}, \dots, w_{i-1})$:

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

What we have seen so far...

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

... and sum to one

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

What we have seen so far...

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

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

... and sum to one

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

Lecture 4. multi-layer perceptrons are universal function approximators.

What we have seen so far...

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

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

... and sum to one

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

Lecture 4. multi-layer perceptrons are universal function approximators.

This lecture. Can we use a multilayer perceptron to learn $P(w_i \mid w_1, \dots, w_{i-1})$?

Multilayer perceptrons

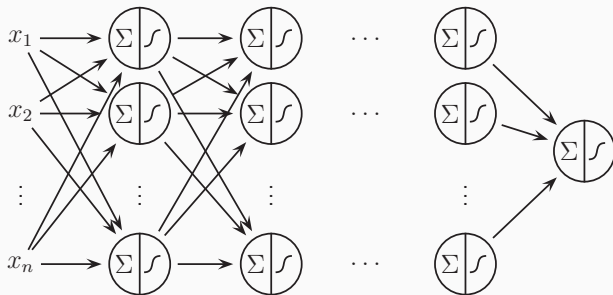
Review of MLPs

Learning: stochastic gradient descent and backpropagation

Language modeling with multilayer perceptrons

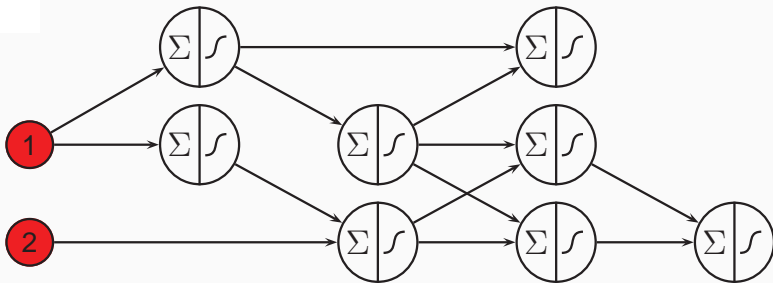
Multilayer perceptrons

Reminder: Multilayer Perceptrons (MLPs)



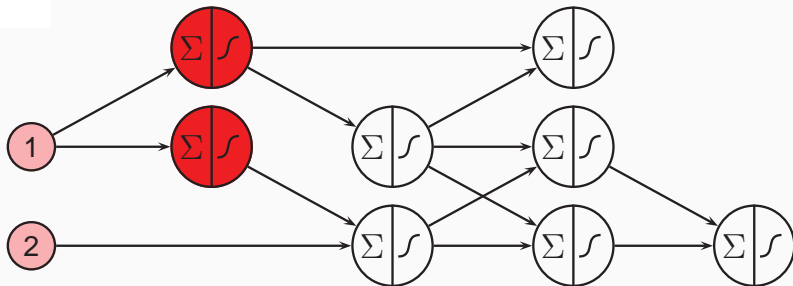
- MLPs are **feed-forward** neural networks, organized in layers.
- One **input** layer, one or more **hidden** layers, one **output** layer.
- Each node in a layer connected to all other nodes in next layer.
- Each connection has a weight (can be zero).
- With one hidden layer, is a universal function approximator!
- ... But can be connected up arbitrarily as needed.

MLP computes a function in a forward pass



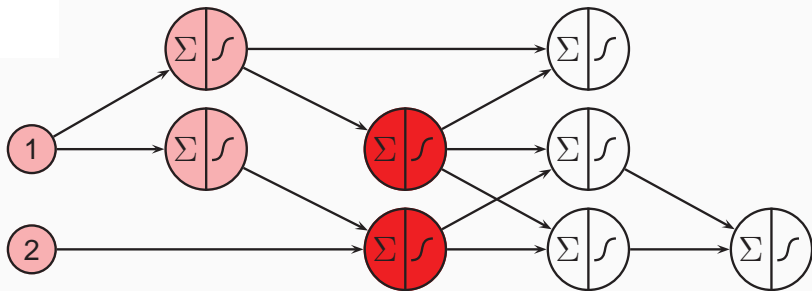
1. Present the pattern at the input layer.

MLP computes a function in a forward pass



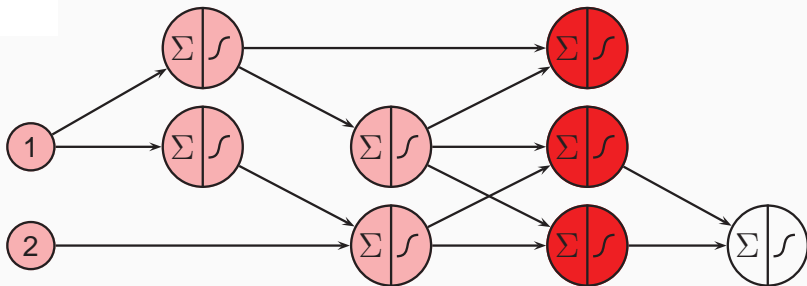
1. Present the pattern at the input layer.
2. Calculate activation of input neurons

MLP computes a function in a forward pass



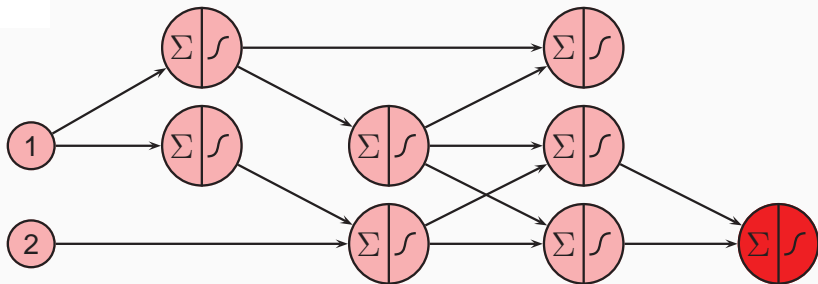
1. Present the pattern at the input layer.
2. Calculate activation of input neurons
3. Propagate forward activations step by step.

MLP computes a function in a forward pass



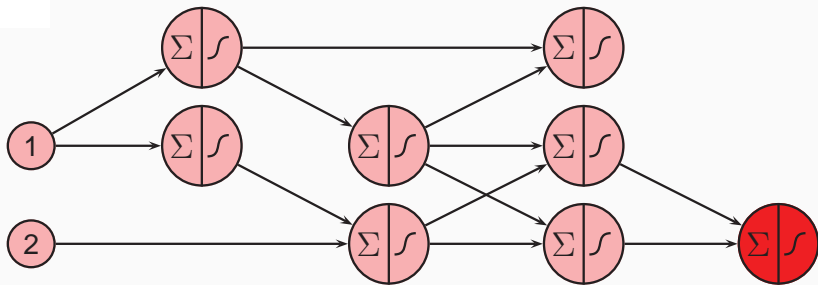
1. Present the pattern at the input layer.
2. Calculate activation of input neurons
3. Propagate forward activations step by step.

MLP computes a function in a forward pass



1. Present the pattern at the input layer.
2. Calculate activation of input neurons
3. Propagate forward activations step by step.

MLP computes a function in a forward pass



1. Present the pattern at the input layer.
2. Calculate activation of input neurons
3. Propagate forward activations step by step.
4. Read the network output from both output neurons.

Learning in multilayer perceptrons

General Idea: same as in a simple perceptron

1. Send the MLP an input pattern, x , from the **training set**.
2. Get the output from the MLP, y .
3. Compare y with the “right answer”, or target t , to get the **error quantity**.
4. Use the error quantity to modify the weights, so next time y will be closer to t .
5. Repeat with another x from the training set.

Learning in multilayer perceptrons

General Idea: same as in a simple perceptron

1. Send the MLP an input pattern, x , from the **training set**.
2. Get the output from the MLP, y .
3. Compare y with the “right answer”, or target t , to get the **error quantity**.
4. Use the error quantity to modify the weights, so next time y will be closer to t .
5. Repeat with another x from the training set.

When updating weights after seeing x , the network doesn't just change the way it deals with x , but other inputs too ...

Inputs it has not seen yet!

Learning in multilayer perceptrons

General Idea: same as in a simple perceptron

1. Send the MLP an input pattern, x , from the **training set**.
2. Get the output from the MLP, y .
3. Compare y with the “right answer”, or target t , to get the **error quantity**.
4. Use the error quantity to modify the weights, so next time y will be closer to t .
5. Repeat with another x from the training set.

When updating weights after seeing x , the network doesn't just change the way it deals with x , but other inputs too ...

Inputs it has not seen yet!

Generalization is the ability to deal accurately with unseen inputs.

Learning as Error Minimization

The perceptron learning rule attempts to reduce the difference between the actual and desired outputs:

$$w_i \leftarrow w_i + \eta(t - o)x_i$$

Generalized to MLPs: Mean Squared Error (MSE)

An **error function** represents such a difference over a set of inputs:

$$E(\vec{w}) = \frac{1}{2N} \sum_{p=1}^N (t^p - o^p)^2$$

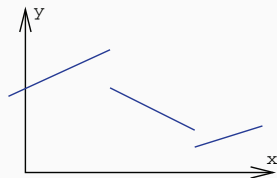
- N is the number of patterns
- t^p is the target output for pattern p
- o^p is the output obtained for pattern p
- the 2 makes little difference, but makes life easier later on!

Minimize error by gradient descent

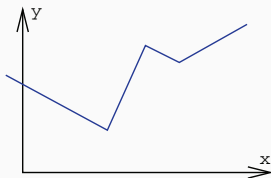
Interpret E just as a mathematical function depending on \vec{w} and forget about its semantics, then we are faced with a problem of mathematical optimization.

$$\underset{\vec{u}}{\text{minimize}} f(\vec{u})$$

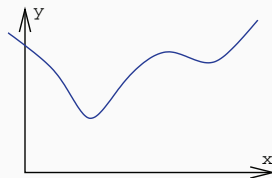
We consider only continuous and differentiable functions.



non continuous function
(disrupted)



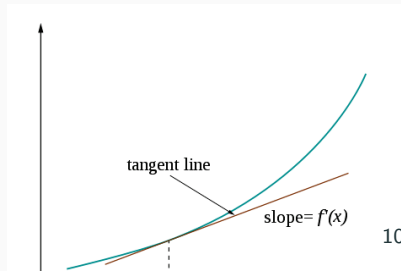
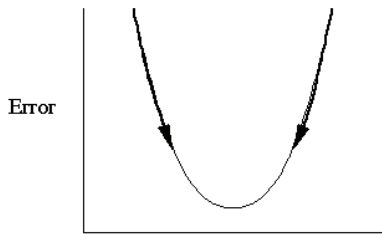
continuous, non differentiable
function (folded)



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

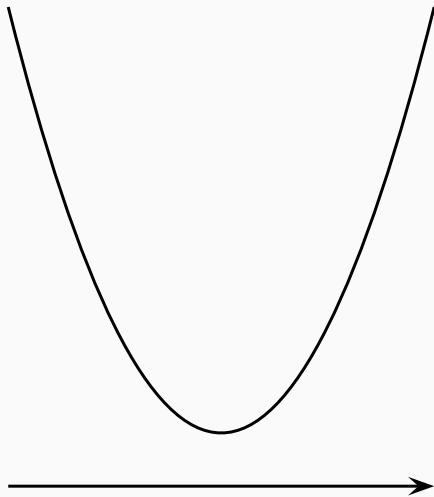
If $y = f(x_1, x_2)$, then we can have $\frac{\partial y}{\partial x_1}$ and $\frac{\partial y}{\partial x_2}$.

Given partial derivatives, update the weights:

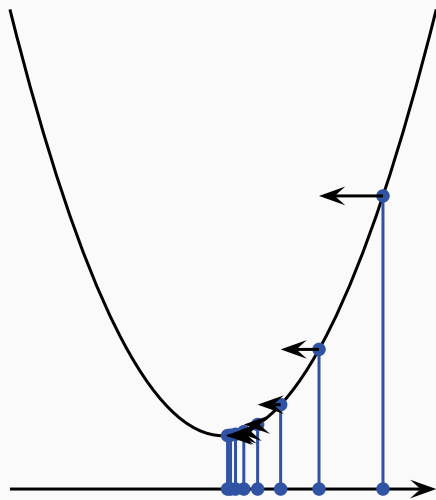
$$w'_{ij} = w_{ij} + \Delta w_{ij}$$

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

Learning Rate

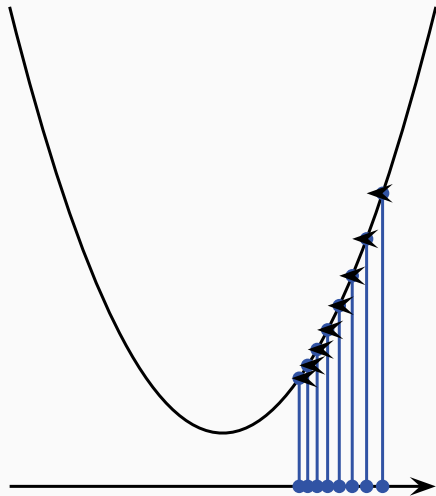


Learning Rate



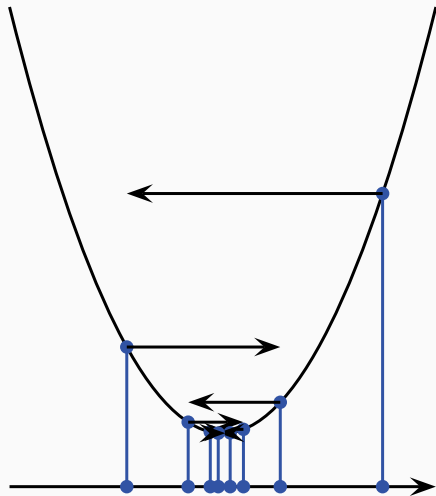
Small η leads to convergence.

Learning Rate



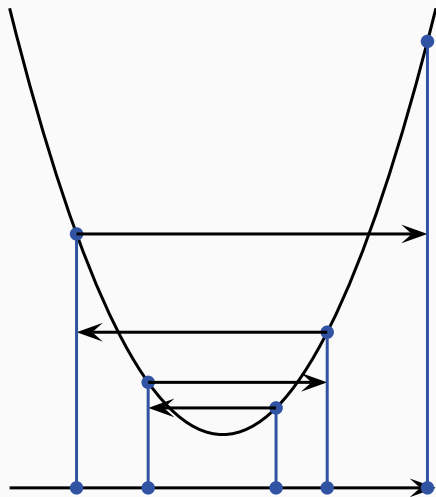
Very small η , convergence may take very long.

Learning Rate



Case of medium size η , also converges.

Learning Rate



Very large η : divergence.

Gradient Descent (cont.)

- Pure gradient descent is a nice theoretical framework but of limited power in practice.
- Finding the right η 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

Summary So Far

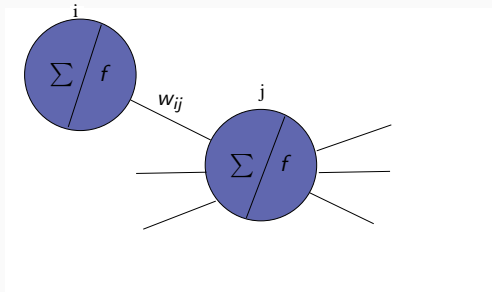
- We learnt what a multilayer perceptron is.
- We know a learning rule for updating weights in order to minimise the error:

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

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

- Δw_{ij} tells us in which **direction** and **how much** we should change each weight to roll down the slope (descend the gradient) of the error function E .
- So, how do we calculate $\frac{\partial E}{\partial w_{ij}}$?

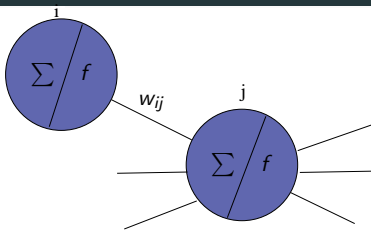
Using Gradient Descent to Minimize the Error



The mean squared error function E , which we want to minimize:

$$E(\vec{w}) = \frac{1}{2N} \sum_{p=1}^N (t^p - o^p)^2$$

Using Gradient Descent to Minimize the Error



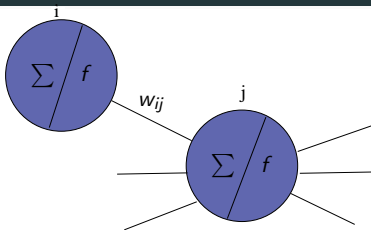
If we use a sigmoid activation function f , then the output of neuron i for pattern p is:

$$o_i^p = f(u_i) = \frac{1}{1 + e^{au_i}}$$

where a is a pre-defined constant and u_i is the result of the input function in neuron i :

$$u_i = \sum_j w_{ij} x_{ij}$$

Using Gradient Descent to Minimize the Error



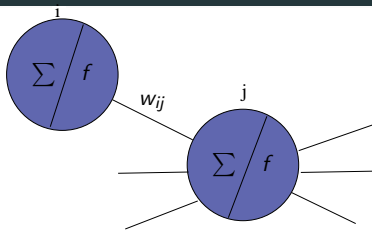
For the p th pattern and the i th neuron, we use gradient descent on the error function:

$$\Delta w_{ij} = -\eta \frac{\partial E_p}{\partial w_{ij}} = \eta(t_i^p - o_i^p)f'(u_i)x_{ij}$$

where $f'(u_i) = \frac{df}{du_i}$ is the derivative of f with respect to u_i .

If f is the sigmoid function, $f'(u_i) = af(u_i)(1 - f(u_i))$.

Using Gradient Descent to Minimize the Error



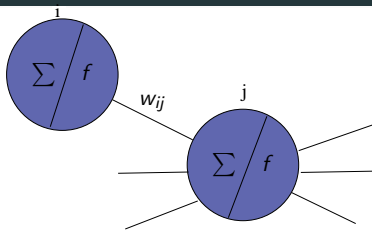
We can update weights after processing each pattern, using rule:

$$\Delta w_{ij} = \eta (t_i^p - o_i^p) f'(u_i) x_{ij}$$

$$\Delta w_{ij} = \eta \delta_i^p x_{ij}$$

- This is known as the **generalized delta rule**.
- We need to use the derivative of the activation function f .
So, f must be differentiable!
- Sigmoid has a derivative which is easy to calculate.

Using Gradient Descent to Minimize the Error



We can update weights after processing each pattern, using rule:

$$\Delta w_{ij} = \eta (t_i^p - o_i^p) f'(u_i) x_{ij}$$

$$\Delta w_{ij} = \eta \delta_i^p x_{ij}$$

- This is known as the **generalized delta rule**.
- We need to use the derivative of the activation function f .
So, f must be differentiable!
- Sigmoid has a derivative which is easy to calculate.

Updating Output vs Hidden Neurons

We can update **output neurons** using the generalized delta rule:

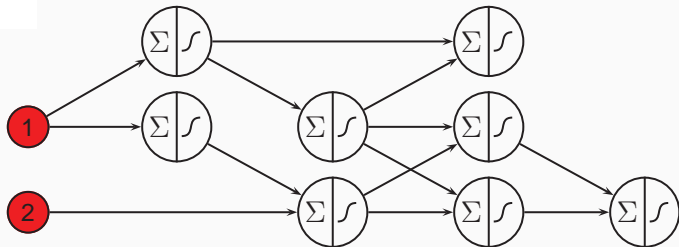
$$\Delta w_{ij} = \eta \delta_i^p x_{ij}$$
$$\delta_i^p = (t_i^p - o_i^p) f'(u_i)$$

This δ_i^p is only good for the **output neurons**, it relies on target outputs. But we don't have target output for the **hidden nodes**!

$$\Delta w_{ki} = \eta \delta_k^p x_{ik} \qquad \delta_k^p = \sum_{j \in I_k} \delta_j^p w_{kj}$$

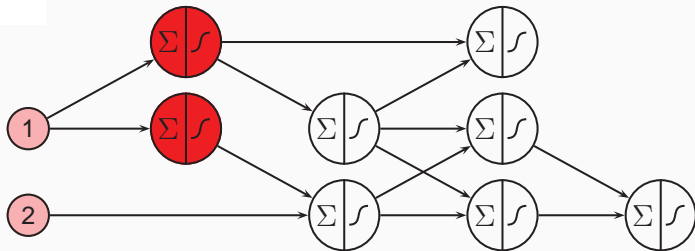
This rule propagates error back from output nodes to hidden nodes. In effect, it **blames hidden nodes** according to how much influence they had. So, now we have rules for updating both output and hidden neurons!

Backpropagation



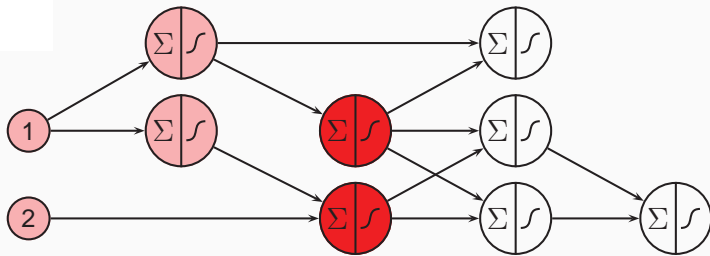
1. Present the pattern at the input layer.

Backpropagation



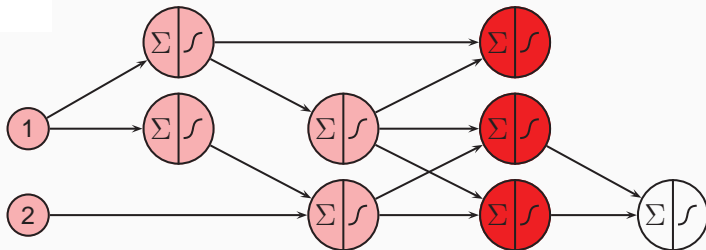
1. Present the pattern at the input layer.
2. Propagate forward activations

Backpropagation



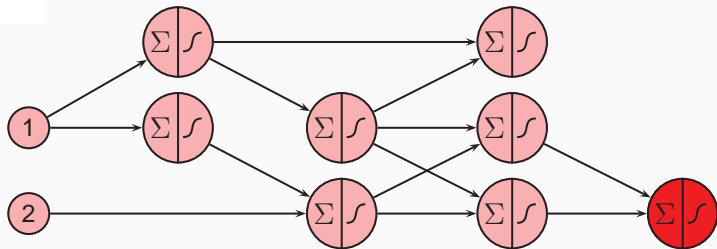
1. Present the pattern at the input layer.
2. Propagate forward activations step

Backpropagation



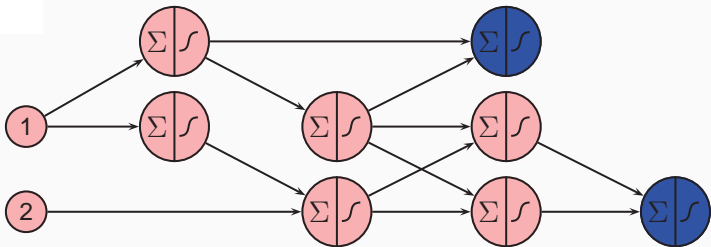
1. Present the pattern at the input layer.
2. Propagate forward activations step by

Backpropagation



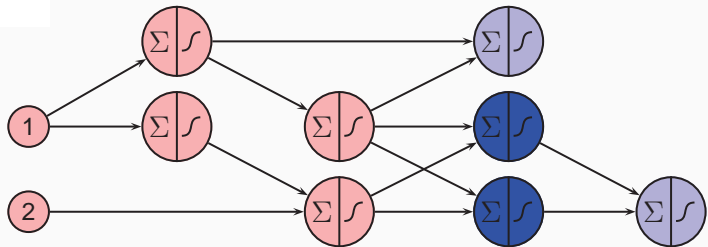
1. Present the pattern at the input layer.
2. Propagate forward activations step by step.

Backpropagation



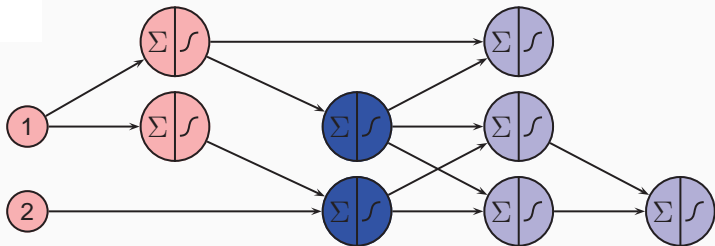
1. Present the pattern at the input layer.
2. Propagate forward activations step by step.
3. Calculate error from both output neurons.

Backpropagation



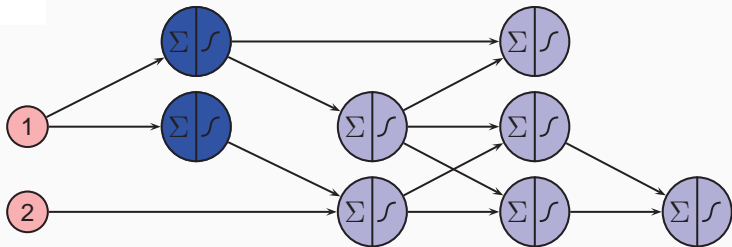
1. Present the pattern at the input layer.
2. Propagate forward activations step by step.
3. Calculate error from both output neurons.
4. Propagate backward error.

Backpropagation



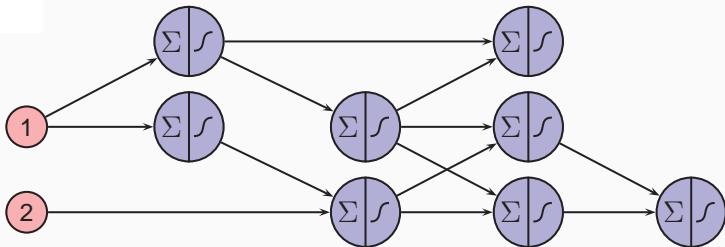
1. Present the pattern at the input layer.
2. Propagate forward activations step by step.
3. Calculate error from both output neurons.
4. Propagate backward error.

Backpropagation



1. Present the pattern at the input layer.
2. Propagate forward activations step by step.
3. Calculate error from both output neurons.
4. Propagate backward error.

Backpropagation



1. Present the pattern at the input layer.
2. Propagate forward activations step by step.
3. Calculate error from both output neurons.
4. Propagate backward error.
5. Calculate $\frac{\partial E}{\partial w_{ij}}$; repeat for all patterns and sum up.

Stochastic (i.e. Online) Gradient Descent

- 1: Initialize all weights to small random values.
- 2: **repeat**
- 3: **for** each training example **do**
- 4: Forward propagate the input features of the example to determine the MLP's outputs.
- 5: Back propagate error to generate Δw_{ij} for all weights w_{ij} .
- 6: Update the weights using Δw_{ij} .
- 7: **end for**
- 8: **until** stopping criteria reached.

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

Summary: multilayer perceptrons, SGD, and backpropagation

Goal has been to give intuition of these topics. This is not a machine learning class!

All of these ideas are very generalizable:

- Backpropagation works on quite arbitrary *computation graphs*.
- Typically don't compute gradients like this by hand (though you will in coursework 1). This is done for you in deep learning toolkits, which use *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!

Language modeling with multilayer perceptrons

Feedforward networks have only one datatype

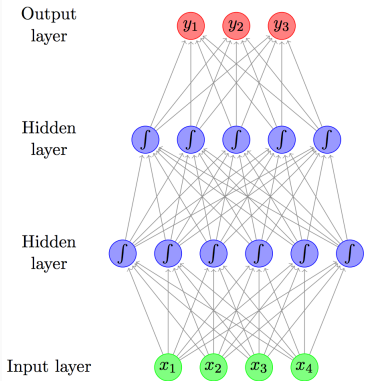
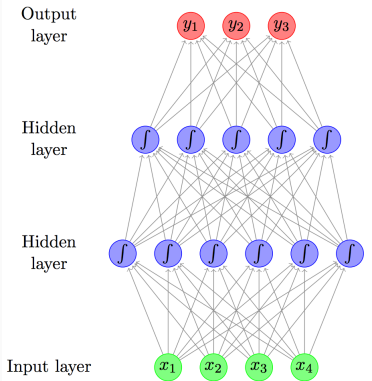


image source: Goldberg, *A Primer on Neural Network Models for Natural Language Processing*

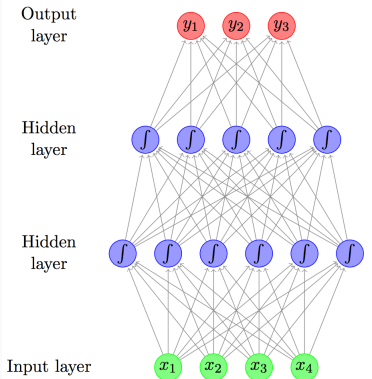
Feedforward networks have only one datatype



Input x : a vector of real numbers.

image source: Goldberg, *A Primer on Neural Network Models for Natural Language Processing*

Feedforward networks have only one datatype

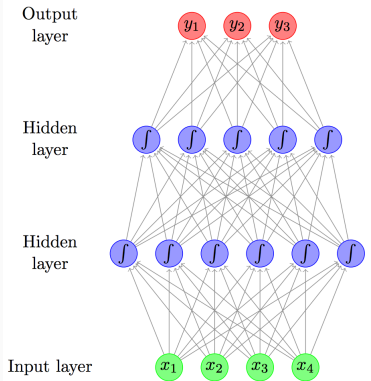


Input x : a vector of real numbers.

Output y : a vector of real numbers.

image source: Goldberg, *A Primer on Neural Network Models for Natural Language Processing*

Feedforward networks have only one datatype



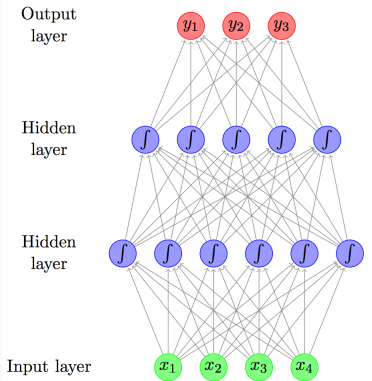
Input x : a vector of real numbers.

Output y : a vector of real numbers.

Hidden layers: vectors of real numbers.

image source: Goldberg, *A Primer on Neural Network Models for Natural Language Processing*

Feedforward networks have only one datatype



Input x : a vector of real numbers.

Output y : a vector of real numbers.

Hidden layers: vectors of real numbers.

How do we use them to model $P(w_i \mid w_{i-n+1}, \dots, w_{i-1})$?

image source: Goldberg, *A Primer on Neural Network Models for Natural Language Processing*



Conditional probability distributions as vectors?

What is the input/ output of $P(w_i \mid w_{i-n+1}, \dots, w_{i-1})$?

Conditional probability distributions as vectors?

What is the input/ output of $P(w_i \mid w_{i-n+1}, \dots, w_{i-1})$?

n -gram view: input is w_{i-n+1}, \dots, w_i , output is a real value. Keys with shared prefix w_{i-n+1}, \dots, w_i must sum to one. Implemented as a lookup table. So, we have a function $f : V^n \rightarrow \mathbb{R}_+$.

Conditional probability distributions as vectors?

What is the input/ output of $P(w_i \mid w_{i-n+1}, \dots, w_{i-1})$?

n -gram view: input is w_{i-n+1}, \dots, w_i , output is a real value. Keys with shared prefix w_{i-n+1}, \dots, w_i must sum to one. Implemented as a lookup table. So, we have a function $f : V^n \rightarrow \mathbb{R}_+$.

Functional view: input is $w_{i-n+1}, \dots, w_{i-1}$, output is a probability distribution over vocabulary. So we have function $f : V^{n-1} \rightarrow (V \rightarrow \mathbb{R}_+)$

Conditional probability distributions as vectors?

What is the input/ output of $P(w_i \mid w_{i-n+1}, \dots, w_{i-1})$?

n -gram view: input is w_{i-n+1}, \dots, w_i , output is a real value. Keys with shared prefix w_{i-n+1}, \dots, w_i must sum to one. Implemented as a lookup table. So, we have a function $f : V^n \rightarrow \mathbb{R}_+$.

Functional view: input is $w_{i-n+1}, \dots, w_{i-1}$, output is a probability distribution over vocabulary. So we have function $f : V^{n-1} \rightarrow (V \rightarrow \mathbb{R}_+)$

How do we represent this function with vectors?

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	Meaning
-------	---------

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	Meaning	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	Meaning	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	Meaning	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	Meaning	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	Meaning	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	Meaning	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	Meaning	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!

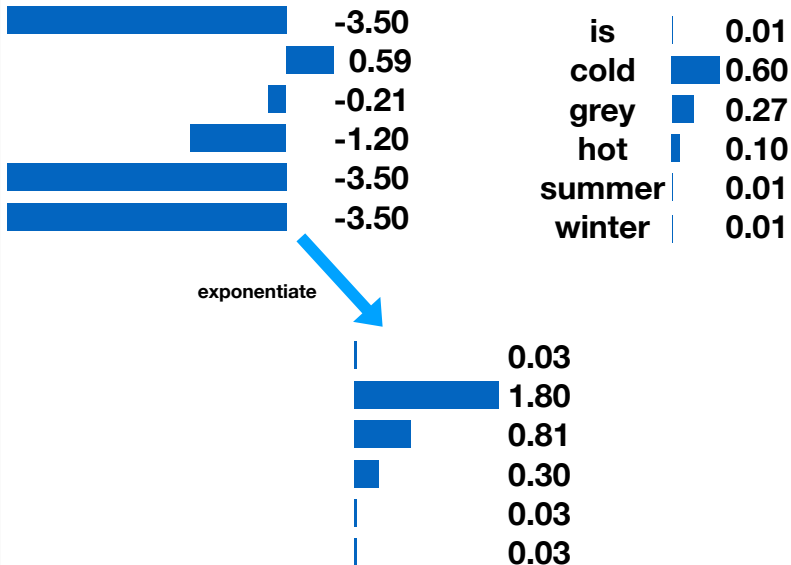
is		0.01
cold		0.60
grey		0.27
hot		0.10
summer		0.01
winter		0.01

Probability distributions are vectors!

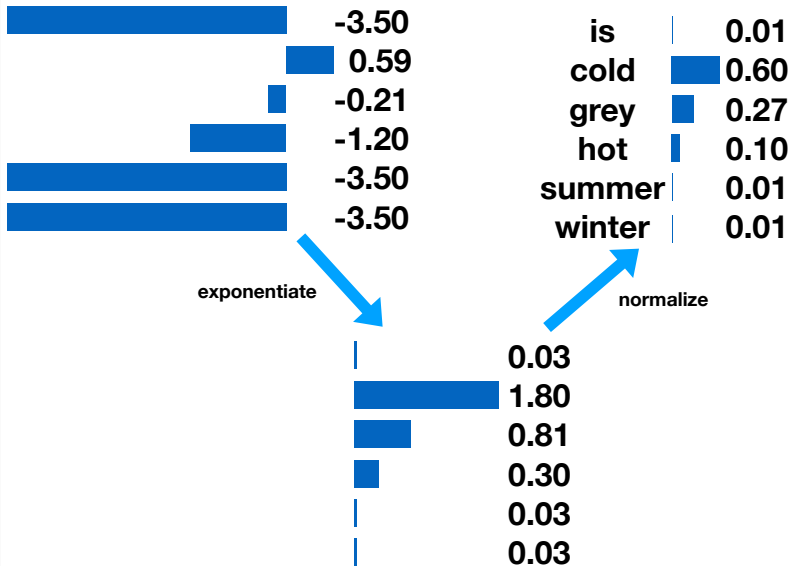


is	0.01
cold	0.60
grey	0.27
hot	0.10
summer	0.01
winter	0.01

Probability distributions are vectors!



Probability distributions are vectors!



Turn any vector into a probability with the softmax function!

$$P(Y = y_i) = \frac{\exp(x_i)}{\sum_{i' \in [1, \dots, |x|]} \exp(x_{i'})}$$

- Softmax is a generalization of the logistic function. Give input vector x , produce output representing probability distribution over elements of x , given by y here.
- Exponentiation makes every value positive—i.e. greater than 0.

Turn any vector into a probability with the softmax function!

$$P(Y = y_i) = \frac{\exp(x_i)}{\sum_{i' \in [1, \dots, |x|]} \exp(x_{i'})}$$

- Softmax is a generalization of the logistic function. Give input vector x , produce output representing probability distribution over elements of x , given by y here.
- Exponentiation makes every value positive—i.e. greater than 0. No need for smoothing!

Turn any vector into a probability with the softmax function!

$$P(Y = y_i) = \frac{\exp(x_i)}{\sum_{i' \in [1, \dots, |x|]} \exp(x_{i'})}$$

- Softmax is a generalization of the logistic function. Give input vector x , produce output representing probability distribution over elements of x , given by y here.
- Exponentiation makes every value positive—i.e. greater than 0. No need for smoothing!
- Normalization makes everything sum to one.

Turn any vector into a probability with the softmax function!

$$P(Y = y_i) = \frac{\exp(x_i)}{\sum_{i' \in [1, \dots, |x|]} \exp(x_{i'})}$$

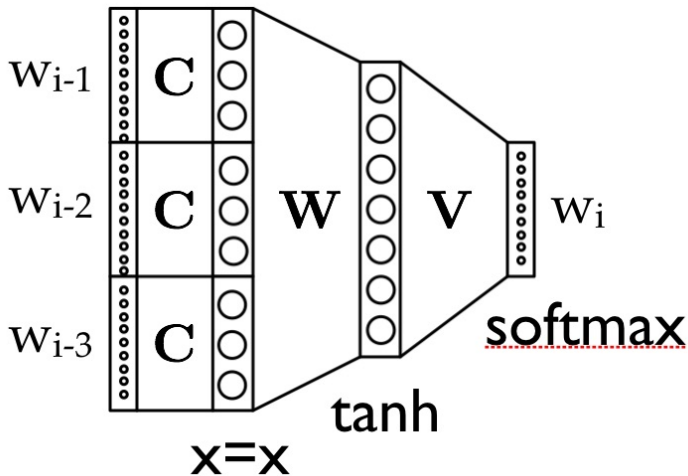
- Softmax is a generalization of the logistic function. Give input vector x , produce output representing probability distribution over elements of x , given by y here.
- Exponentiation makes every value positive—i.e. greater than 0. No need for smoothing!
- Normalization makes everything sum to one.
- You have seen this before: it's just logistic regression.

Turn any vector into a probability with the softmax function!

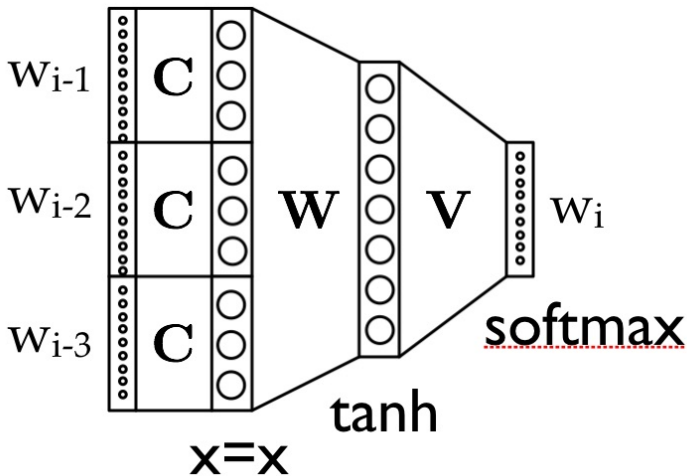
$$P(Y = y_i) = \frac{\exp(x_i)}{\sum_{i' \in [1, \dots, |x|]} \exp(x_{i'})}$$

- Softmax is a generalization of the logistic function. Give input vector x , produce output representing probability distribution over elements of x , given by y here.
- Exponentiation makes every value positive—i.e. greater than 0. No need for smoothing!
- Normalization makes everything sum to one.
- You have seen this before: it's just logistic regression. But in logistic regression, x came from a linear combination of input features and weights. What are the features in our model?

Feedforward LM: function from a vector to a vector



Feedforward LM: function from a vector to a vector



Parameter matrices: C , W , and V . What do these do?

What do the hidden layers do?

- Each hidden layer is a *representation* of its input.
- Multiplying a one-hot vector by C is equivalent to selecting a single row of C —which is a learned parameter matrix.
- So, we say that the i th row of C is the learned representation, or *embedding* of the i th word in the vocabulary.
- Learned representations take the place of hand-engineered *features* in classic logistic regression.
- Because matrix multiplication can produce an output vector of different dimensions than the input, these representations can be of any size. They are often small (dense), rather than the large (sparse) representations of classic logistic regression.
- In this model, the second parameter matrix W produces a representation of the full n -gram history.

Summary of key points (i.e. examinable content)

- We have some intuition about using gradient descent on an error function.
- We know a learning rule for updating weights in order to minimize the error for an MLP.
- We can use this rule to learn an MLP's weights using the **backpropagation algorithm**.
- n -gram models can be parameterized with simple multilayer neural network.
- This model uses **representation learning** to learn a function from concatenated **one-hot encodings** of input variables, and produces probability distributions using the **softmax** function.
- Any conditional probability distribution over discrete variables can be parameterized by an MLP using a similar strategy.

A close-up, black and white photograph of a cat's face. The cat is wearing a pair of round, dark sunglasses. The cat's whiskers are visible, and its eyes are hidden behind the lenses. The background is a plain, light color.

WHAT IF I TOLD YOU

**YOU CAN ENCODE AN ENTIRE
SENTENCE IN A VECTOR**