Learnability of Quantifiers

ML1: past approaches, intro to neural networks

Shane Steinert-Threlkeld & Jakub Szymanik
University of Washington, Linguistics
ILLC
University of Amsterdam

Outline

1. Recap
2. Formal Language Learning of GQs
3. Bayesian Learning
4. Introduction to Neural Networks
   - NNs: Computation
   - NNs: Learning
   - Hands-on Example

Recap

Yesterday:

- How to represent GQs as formal languages
- Relations between logical definability and the Chomsky hierarchy

Today:

- The role of universals in formal language learning
- A Bayesian model of quantifier learning
- Introduction to neural networks, with an eye towards quantifiers

Recap

Yesterday:

- How to represent GQs as formal languages
- Relations between logical definability and the Chomsky hierarchy

Today:

- The role of universals in formal language learning
- A Bayesian model of quantifier learning
- Introduction to neural networks, with an eye towards quantifiers

Outline

Recall...

- $\mathcal{L}_{\text{every}} = \{w \mid \#_0(w) = 0\}$
- $\mathcal{L}_{\text{some}} = \{w \mid \#_1(w) > 0\}$
- $\mathcal{L}_{\text{most}} = \{w \mid \#_1(w) > \#_0(w)\}$

A Model of Language Learning

- *a*
- *aa*
- *aaaaaaa*
- *ab*
- *aaaabbb*
- ...

A Model of Language Learning

- *a*
- *aa*
- *aaaaaaa*
- *ab*
- *aaaabbb*
- ...

A Model of Language Learning

- *a*
- *aa*
- *aaaaaaa*
- *ab*
- *aaaabbb*
- ...

A Model of Language Learning

- *a*
- *aa*
- *aaaaaaa*
- *ab*
- *aaaabbb*
- . . .

A Model of Language Learning

- *a*
- *aa*
- *aaaaaaa*
- *ab*
- *aaaabbb*
- . . .

A Model of Language Learning

- *a*
- *aa*
- *aaaaaaa*
- *ab*
- *aaaabbb*
- . . .

Some Definitions

- A *text* ε over L is an infinite sequence of words from L
- A *learner* ℓ is a function from finite fragments of texts to languages
  E.g.:

$$\ell(\{a, aa, aaaaaa\}) = a^*$$
$$\ell(\{a, aa, aaaaaa, ab\}) = a^* \cup \{ab\}$$

Some Definitions

- A *text* ε over L is an infinite sequence of words from L
- A *learner* ℓ is a function from finite fragments of texts to languages
  E.g.:

$$\ell(\{a, aa, aaaaaa\}) = a^*$$
$$\ell(\{a, aa, aaaaaa, ab\}) = a^* \cup \{ab\}$$

Some Definitions

- A *text* ε over L is an infinite sequence of words from L
- A *learner* ℓ is a function from finite fragments of texts to languages
  E.g.:

$$\ell(\{a, aa, aaaaaa\}) = a^*$$
$$\ell(\{a, aa, aaaaaa, ab\}) = a^* \cup \{ab\}$$

Identifiability

- L is *finitely identifiable* on ε by ℓ if there is an n s.t. $\ell(\varepsilon \upharpoonright n) = L$ and ℓ 'announces the correctness of its verdict'
- L is *identifiable in the limit* on ε by ℓ if there is an n s.t. $\forall m \geq n$, $\ell(\varepsilon \upharpoonright m) = L$
- A class of languages $\mathcal{L}$ is identifiable if there is an ℓ that identifies every $L \in \mathcal{L}$ on every ε

Identifiability

- L is *finitely identifiable* on ε by ℓ if there is an n s.t. $\ell(\varepsilon \restriction n) = L$ and ℓ 'announces the correctness of its verdict'
- L is *identifiable in the limit* on ε by ℓ if there is an n s.t. $\forall m \geq n$, $\ell(\varepsilon \restriction m) = L$
- A class of languages $\mathcal{L}$ is identifiable if there is an ℓ that identifies every $L \in \mathcal{L}$ on every ε

Identifiability

- L is *finitely identifiable* on ε by ℓ if there is an n s.t. $\ell(\varepsilon \restriction n) = L$ and ℓ 'announces the correctness of its verdict'
- L is *identifiable in the limit* on ε by ℓ if there is an n s.t. $\forall m \geq n$, $\ell(\varepsilon \restriction m) = L$
- A class of languages $\mathcal{L}$ is identifiable if there is an ℓ that identifies every $L \in \mathcal{L}$ on every ε

Identifiability and the Chomsky hierarchy

Anomalous text	Recursively enumerable
	Recursive
Informant	Primitive recursive
	Context–sensitive
	Context–free
	Regular
	Superfinite
Text	Finite cardinality languages

Universals and Learnability

Question

Do the universals for quantifiers help any this notion of learnability?
I.e.: what about sub-classes of languages defined by, e.g. monotonicity?

Positive Result

Theorem (Tiede 1999)

The set of first-order definable $\uparrow$MON quantifiers is identifiable in the limit.

Negative Result(s)

Theorem (Tiede 1999)

The set of first-order definable MON$\uparrow$ quantifiers is not identifiable in the limit. (Nor are the remaining two monotonicity profiles.)

Outline

1. Recap
2. Formal Language Learning of GQs
3. Bayesian Learning
4. Introduction to Neural Networks
   - NNs: Computation
   - NNs: Learning
   - Hands-on Example

Universals Restricting the Space

The Big Question

Why do the attested universals hold?

Universals Restricting the Space

The Big Question

Why do the attested universals hold?

Universals Restricting the Space

The Big Question

Why do the attested universals hold?

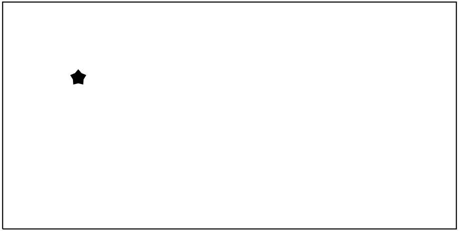
They greatly restrict the search space that a language learner must explore when learning the meanings of expressions. This makes it easier (possible?) for them to learn such meanings from relatively small input.
(Barwise and Cooper 1981; Keenan and Stavi 1986; Szabolsci 2009)

Compare: Poverty of the Stimulus argument for UG.

Universals Restricting the Space

The Big Question

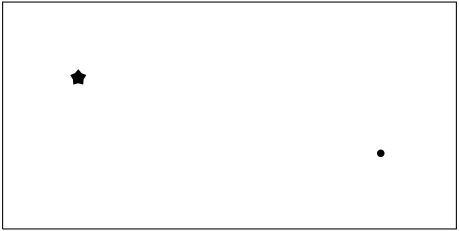
Why do the attested universals hold?



Universals Restricting the Space

The Big Question

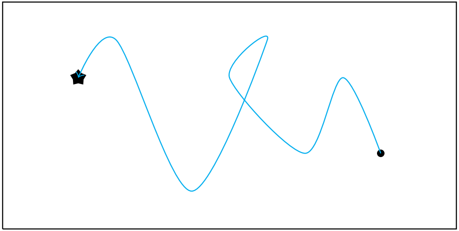
Why do the attested universals hold?



Universals Restricting the Space

The Big Question

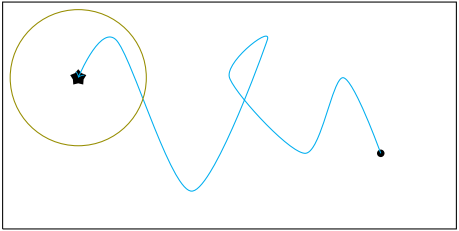
Why do the attested universals hold?



Universals Restricting the Space

The Big Question

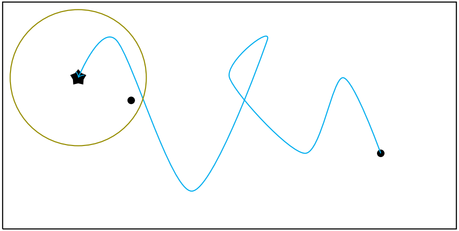
Why do the attested universals hold?



Universals Restricting the Space

The Big Question

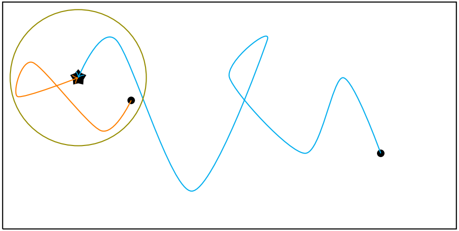
Why do the attested universals hold?



Universals Restricting the Space

The Big Question

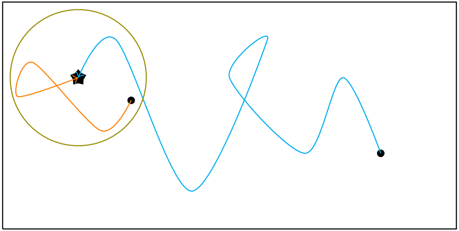
Why do the attested universals hold?



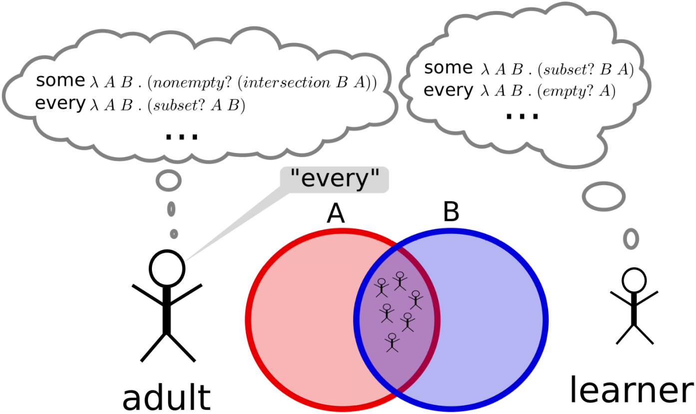
Universals Restricting the Space

The Big Question

Why do the attested universals hold?



Learning Set-up



Piantadosi, Tenenbaum, and Goodman 2013

LoT Grammar

Nonterminal		Expansion	Gloss
START	$\rightarrow$	$\lambda\ A\ B\ .\ BOOL$	Function of A and B
BOOL	$\rightarrow$	*true*	Always true
	$\rightarrow$	*false*	Always false
	$\rightarrow$	(*card>* *SET SET*)	Compare cardinalities (>)
	$\rightarrow$	(*card=* *SET SET*)	Check if cardinalities are equal
	$\rightarrow$	(*subset?* *SET SET*)	Is a subset?
	$\rightarrow$	(*empty?* *SET*)	Is a set empty?
	$\rightarrow$	(*nonempty?* *SET*)	Is a set not empty?
	$\rightarrow$	(*exhaustive?* *SET*)	Is the set the entire set in the context?
	$\rightarrow$	(*singleton?* *SET*)	Contains 1 element?
	$\rightarrow$	(*doubleton?* *SET*)	Contains 2 elements?
	$\rightarrow$	(*tripleton?* *SET*)	Contains 3 elements?
SET	$\rightarrow$	(*union* *SET SET*)	Union of sets
	$\rightarrow$	(*intersection* *SET SET*)	Intersection of sets
	$\rightarrow$	(*set-difference* *SET SET*)	Difference of sets
	$\rightarrow$	A	Argument A
	$\rightarrow$	B	Argument B

Piantadosi, Tenenbaum, and Goodman 2013

Inference

$$P(m|u_1, c_1, u_2, c_2, \ldots u_n, c_n) \propto P(u_1, u_2, \ldots, u_n|m, c_1, \ldots, c_n) \cdot P(m)$$
$$\propto \prod_{i=1}^{n} P(u_i|m, c_i) \cdot P(m)$$

Prior $P(m)$: determined by the LoT grammar; shorter-to-express meanings are preferred.

Likelihood $P(u_i|m, c_i)$: preference for more informative, but with noise.

Inference

$$P(m|u_1, c_1, u_2, c_2, \ldots u_n, c_n) \propto P(u_1, u_2, \ldots, u_n|m, c_1, \ldots, c_n) \cdot P(m)$$
$$\propto \prod_{i=1}^{n} P(u_i|m, c_i) \cdot P(m)$$

Prior $P(m)$: determined by the LoT grammar; shorter-to-express meanings are preferred.

Likelihood $P(u_i|m, c_i)$: preference for more informative, but with noise.

Inference

$$P(m|u_1, c_1, u_2, c_2, \ldots u_n, c_n) \propto P(u_1, u_2, \ldots, u_n|m, c_1, \ldots, c_n) \cdot P(m)$$
$$\propto \prod_{i=1}^{n} P(u_i|m, c_i) \cdot P(m)$$

Prior $P(m)$: determined by the LoT grammar; shorter-to-express meanings are preferred.

Likelihood $P(u_i|m, c_i)$: preference for more informative, but with noise.

Inference

$$P(m|u_1, c_1, u_2, c_2, \ldots u_n, c_n) \propto P(u_1, u_2, \ldots, u_n|m, c_1, \ldots, c_n) \cdot P(m)$$
$$\propto \prod_{i=1}^{n} P(u_i|m, c_i) \cdot P(m)$$

Prior $P(m)$: determined by the LoT grammar; shorter-to-express meanings are preferred.

Likelihood $P(u_i|m, c_i)$: preference for more informative, but with noise.

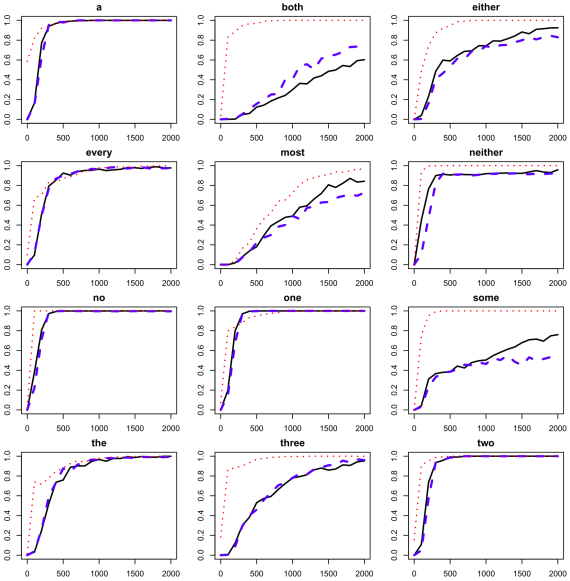
Inference

$$P(m|u_1, c_1, u_2, c_2, \ldots u_n, c_n) \propto P(u_1, u_2, \ldots, u_n|m, c_1, \ldots, c_n) \cdot P(m)$$
$$\propto \prod_{i=1}^{n} P(u_i|m, c_i) \cdot P(m)$$

Prior $P(m)$: determined by the LoT grammar; shorter-to-express meanings are preferred.

Likelihood $P(u_i|m, c_i)$: preference for more informative, but with noise.

Results


a
both
either
every
most
neither
no
one
some
the
three
two
0.0 0.2 0.4 0.6 0.8 1.0
0 500 1000 1500 2000

Conclusion

“Likely, the unrestricted space has many hypotheses which are so implausible, they can be ignored quickly and do not affect learning. The hard part of learning, may be choosing between the plausible competitor meanings, not in weeding out a large space of potential meanings.”

Research Questions

- Does this model predict human learning curves well?
- How sensitive are the model and its results sensitive to various choices (e.g. primitives, weights, shape of likelihood function)?
- Does it say anything general about e.g. monotone and topic-neutral quantifiers?

Research Questions

- Does this model predict human learning curves well?
- How sensitive are the model and its results sensitive to various choices (e.g. primitives, weights, shape of likelihood function)?
- Does it say anything general about e.g. monotone and topic-neutral quantifiers?

Research Questions

- Does this model predict human learning curves well?
- How sensitive are the model and its results sensitive to various choices (e.g. primitives, weights, shape of likelihood function)?
- Does it say anything general about e.g. monotone and topic-neutral quantifiers?

Outline

Today's Plan

1. Neural networks: computation
2. Neural networks: learning
3. Hands-on experiment: learning quantifiers

Goals:

- enough background and material so that you can begin playing around with your own experimental ideas by the end of today
- develop a bit of a map of the field, with pointers to where to go next

Tomorrow:

- Applied to explaining why semantic universals hold
- For quantifiers, but also in other semantic domains
- Connections with complexity and evolution

Today's Plan

1. Neural networks: computation
2. Neural networks: learning
3. Hands-on experiment: learning quantifiers

Goals:

- enough background and material so that you can begin playing around with your own experimental ideas by the end of today
- develop a bit of a map of the field, with pointers to where to go next

Tomorrow:

- Applied to explaining why semantic universals hold
- For quantifiers, but also in other semantic domains
- Connections with complexity and evolution

Today's Plan

1. Neural networks: computation
2. Neural networks: learning
3. Hands-on experiment: learning quantifiers

Goals:

- enough background and material so that you can begin playing around with your own experimental ideas by the end of today
- develop a bit of a map of the field, with pointers to where to go next

Tomorrow:

- Applied to explaining why semantic universals hold
- For quantifiers, but also in other semantic domains
- Connections with complexity and evolution

What I'm Presupposing

Some mathematical notation/concepts from:

- Linear algebra (matrix multiplication, e.g.)
- Multivariable calculus (partial derivatives)

Some programming experience:

- Basics of Python
- Basic syntax in NumPy

But: all concepts and syntax can be explained intuitively, so please ask for clarification at all points!

What I'm Presupposing

Some mathematical notation/concepts from:

- Linear algebra (matrix multiplication, e.g.)
- Multivariable calculus (partial derivatives)

Some programming experience:

- Basics of Python
- Basic syntax in NumPy

But: all concepts and syntax can be explained intuitively, so please ask for clarification at all points!

What I’m Presupposing

Some mathematical notation/concepts from:

- Linear algebra (matrix multiplication, e.g.)
- Multivariable calculus (partial derivatives)

Some programming experience:

- Basics of Python
- Basic syntax in NumPy

But: all concepts and syntax can be explained intuitively, so please ask for clarification at all points!

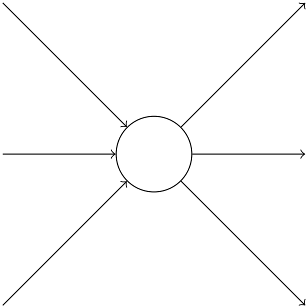
Tutorial

For just these slides, plus some bonus slides, and the Jupyter Notebook:

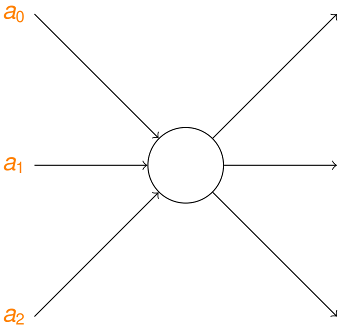
https://github.com/shanest/nn-tutorial

Outline

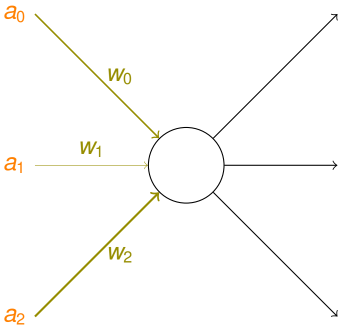
Artificial Neuron



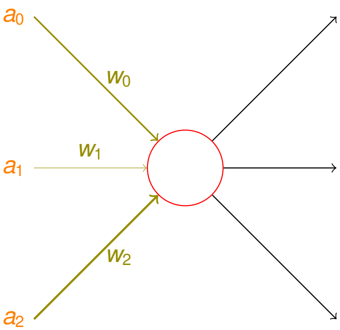
Artificial Neuron



Artificial Neuron

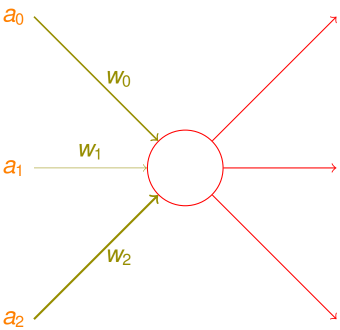


Artificial Neuron



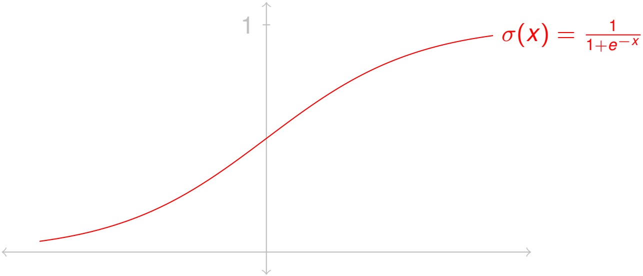
$$a = f(a_0 \cdot w_0 + a_1 \cdot w_1 + a_2 \cdot w_2)$$

Artificial Neuron



$$a = f(a_0 \cdot w_0 + a_1 \cdot w_1 + a_2 \cdot w_2)$$

Activation Function

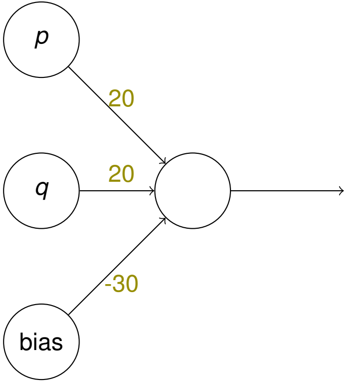


More on choosing activation functions later in the tutorial.

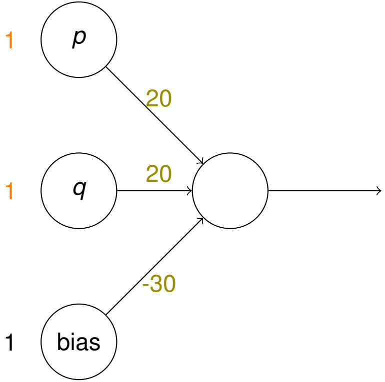
Computing ‘and’

p	q	$p \wedge q$
1	1	1
1	0	0
0	1	0
0	0	0

Computing ‘and’

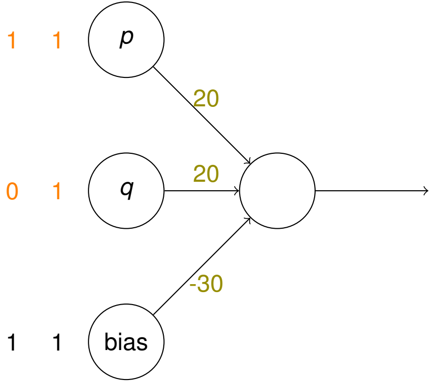


Computing ‘and’



$$a = \sigma(1 \cdot 20 + 1 \cdot 20 + 1 \cdot -30) = \sigma(10) \approx 1$$

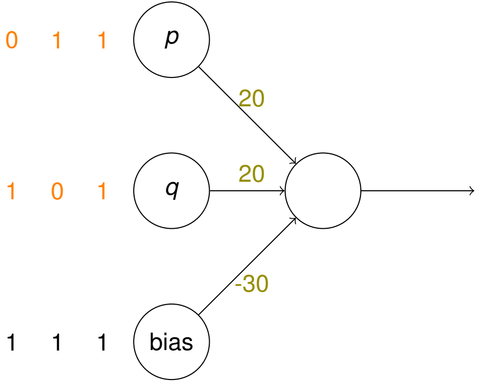
Computing ‘and’



$$a = \sigma(1 \cdot 20 + 1 \cdot 20 + 1 \cdot -30) = \sigma(10) \approx 1$$

$$a = \sigma(1 \cdot 20 + 0 \cdot 20 + 1 \cdot -30) = \sigma(-10) \approx 0$$

Computing ‘and’

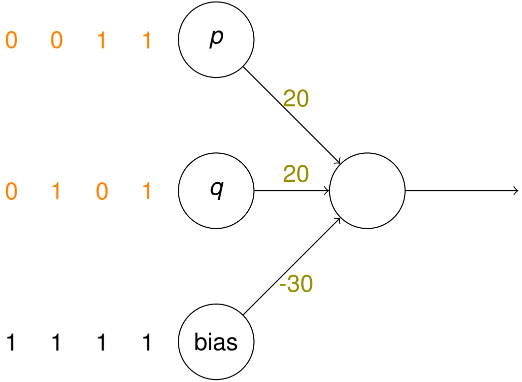


$$a = \sigma(1 \cdot 20 + 1 \cdot 20 + 1 \cdot -30) = \sigma(10) \approx 1$$

$$a = \sigma(1 \cdot 20 + 0 \cdot 20 + 1 \cdot -30) = \sigma(-10) \approx 0$$

$$a = \sigma(0 \cdot 20 + 1 \cdot 20 + 1 \cdot -30) = \sigma(-10) \approx 0$$

Computing 'and'



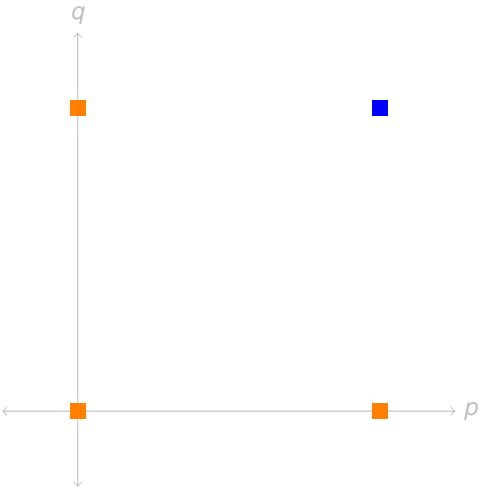
$$a = \sigma(1 \cdot 20 + 1 \cdot 20 + 1 \cdot -30) = \sigma(10) \approx 1$$

$$a = \sigma(1 \cdot 20 + 0 \cdot 20 + 1 \cdot -30) = \sigma(-10) \approx 0$$

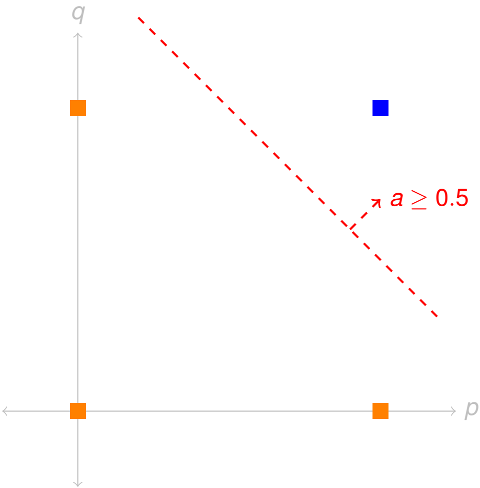
$$a = \sigma(0 \cdot 20 + 1 \cdot 20 + 1 \cdot -30) = \sigma(-10) \approx 0$$

$$a = \sigma(0 \cdot 20 + 0 \cdot 20 + 1 \cdot -30) = \sigma(-30) \approx 0$$

Computing ‘and’



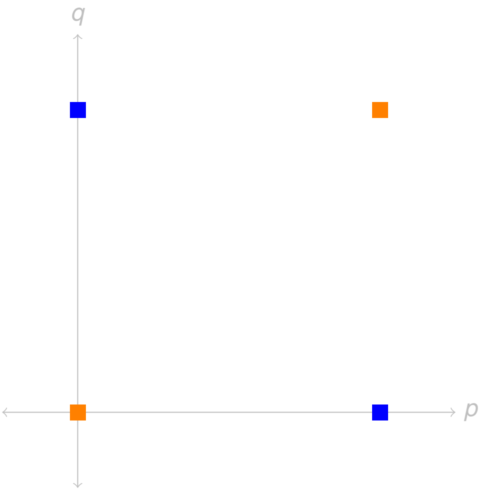
Computing ‘and’



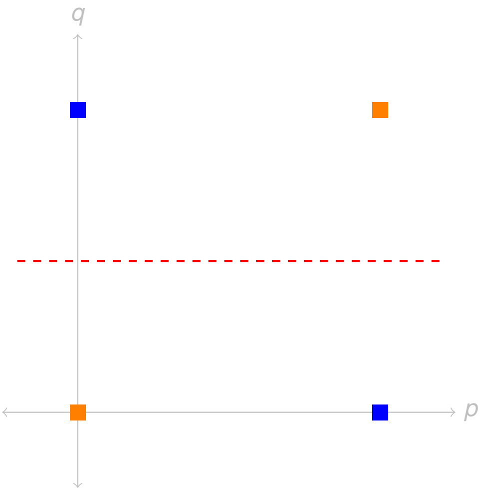
Computing ‘xor’

p	q	p xor q
1	1	0
1	0	1
0	1	1
0	0	0

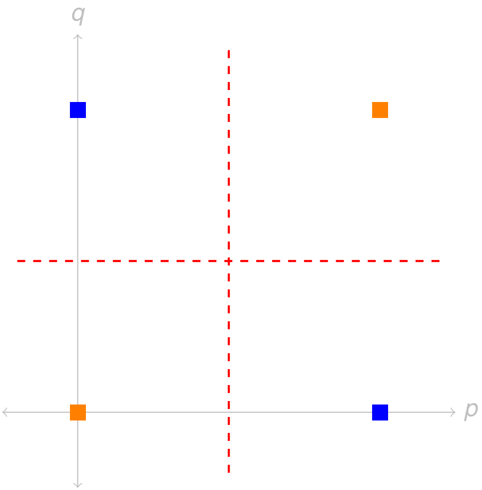
Computing ‘xor’



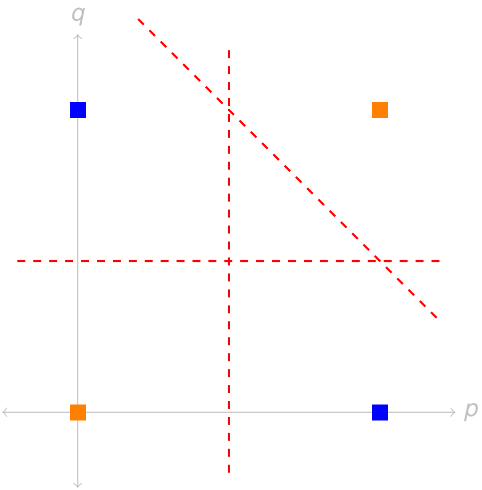
Computing 'xor'



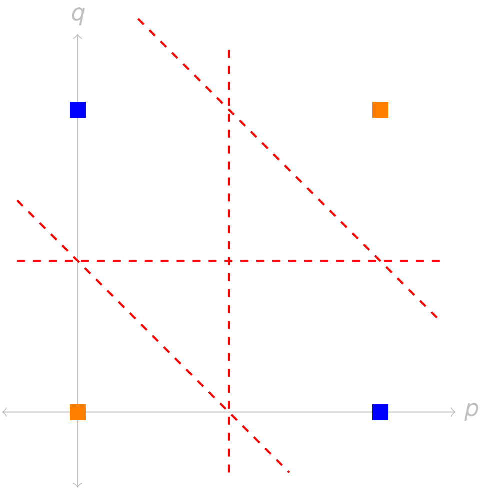
Computing ‘xor’



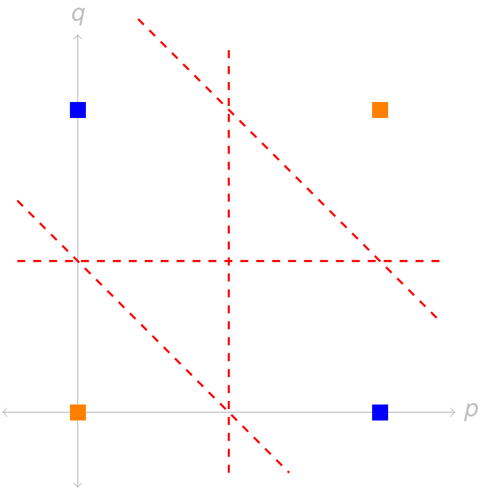
Computing ‘xor’



Computing ‘xor’

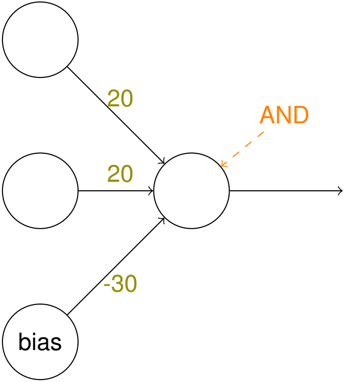


Computing ‘xor’

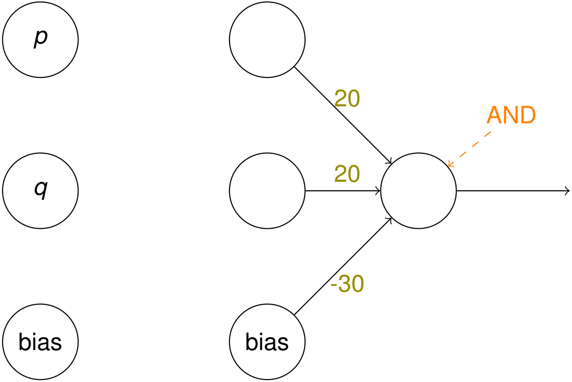


xor is not *linearly separable*

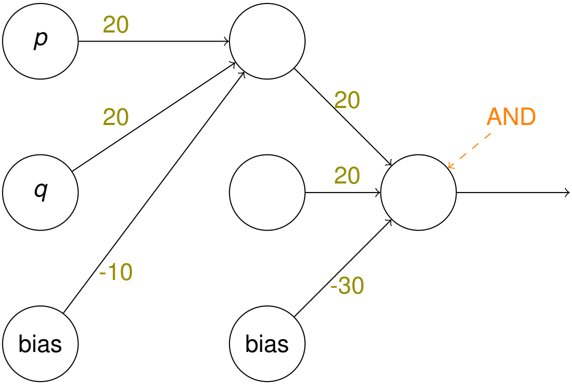
Computing ‘xor’



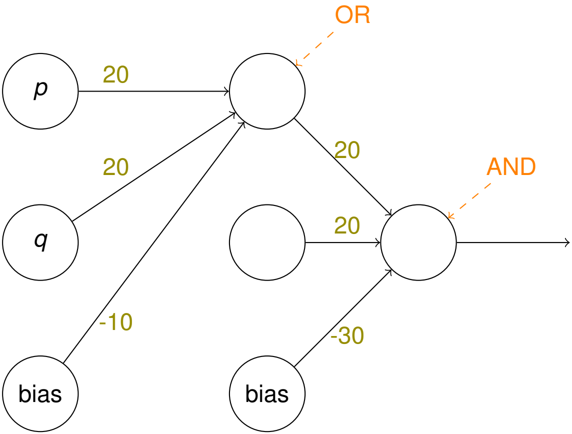
Computing 'xor'



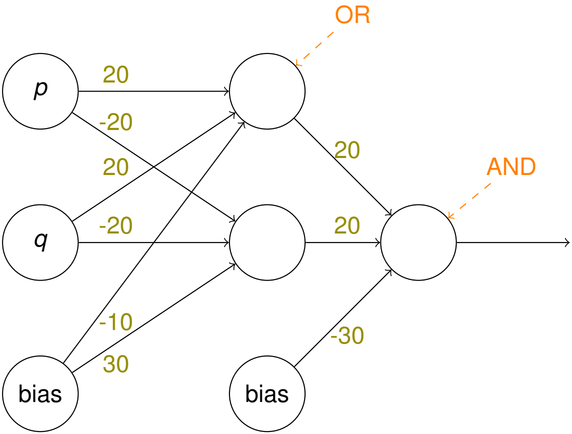
Computing 'xor'



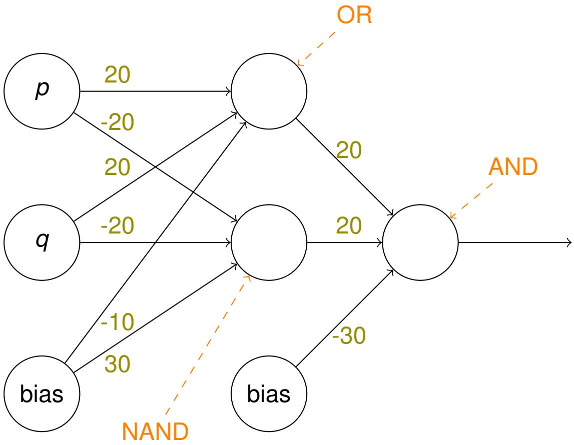
Computing ‘xor’



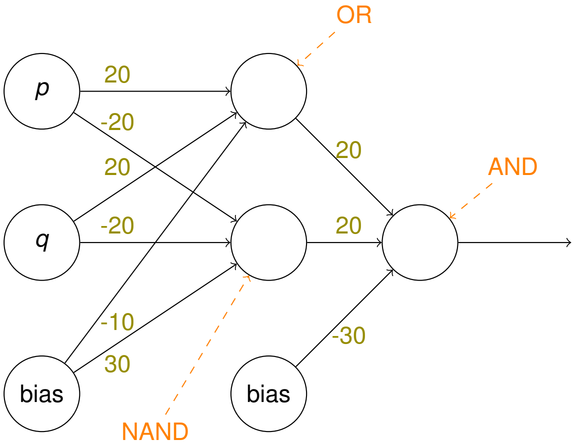
Computing ‘xor’



Computing ‘xor’

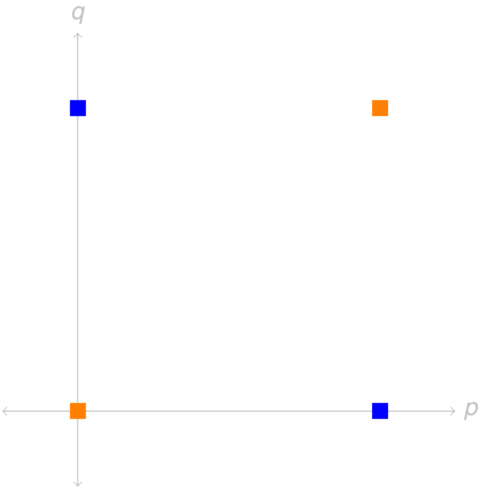


Computing 'xor'

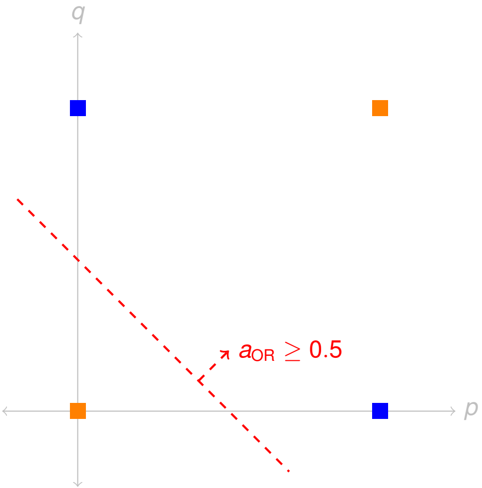


Exercise: show that the hidden units behave as labeled.

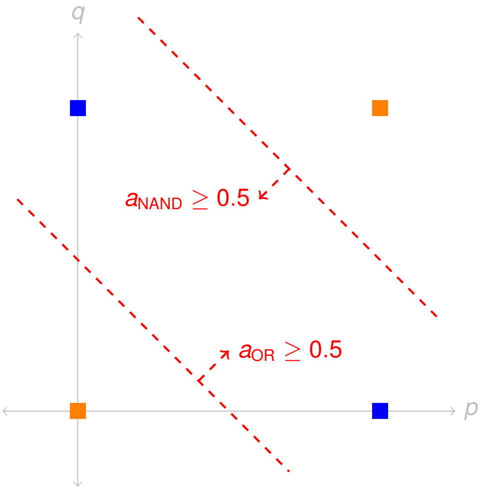
Computing ‘xor’



Computing ‘xor’



Computing ‘xor’



Computing Many Examples

It's often very useful to compute over a 'batch' of inputs at once. The linear combination then turns into a *matrix multiplication*:

$$\vec{a} = f\left(\begin{bmatrix} x_0^0 & x_1^0 & \cdots & x_n^0 \\ x_0^1 & x_1^1 & \cdots & x_n^1 \\ \vdots & \vdots & \ddots & \vdots \\ x_0^m & x_1^m & \cdots & x_n^m \end{bmatrix}\begin{bmatrix} w_0^0 & w_0^1 & \cdots & w_0^l \\ w_1^0 & w_1^1 & \cdots & w_1^l \\ \vdots & \vdots & \ddots & \vdots \\ w_n^0 & w_n^1 & \cdots & w_n^l \end{bmatrix} + \begin{bmatrix} b_0 & b_1 & \cdots & b_l \\ b_0 & b_1 & \cdots & b_l \\ \vdots & \vdots & \ddots & \vdots \\ b_0 & b_1 & \cdots & b_l \end{bmatrix}\right)$$

$$= f(xW + b)$$

- x_j^i: jth feature of input i
- w_k^l: weight from neuron k to neuron l in next layer
- b_m: bias to neuron m in next layer

Exercises:

- write down W^1 and W^2 for the xor network.
- re-write the above as $f(xW)$ by adding a column of 1s to x and a new row to W.

Computing Many Examples

It's often very useful to compute over a 'batch' of inputs at once. The linear combination then turns into a *matrix multiplication*:

$$\vec{a} = f\left(\begin{bmatrix} x_0^0 & x_1^0 & \cdots & x_n^0 \\ x_0^1 & x_1^1 & \cdots & x_n^1 \\ \vdots & \vdots & \ddots & \vdots \\ x_0^m & x_1^m & \cdots & x_n^m \end{bmatrix} \begin{bmatrix} w_0^0 & w_0^1 & \cdots & w_0^l \\ w_1^0 & w_1^1 & \cdots & w_1^l \\ \vdots & \vdots & \ddots & \vdots \\ w_n^0 & w_n^1 & \cdots & w_n^l \end{bmatrix} + \begin{bmatrix} b_0 & b_1 & \cdots & b_l \\ b_0 & b_1 & \cdots & b_l \\ \vdots & \vdots & \ddots & \vdots \\ b_0 & b_1 & \cdots & b_l \end{bmatrix}\right)$$

$$= f(xW + b)$$

- x_j^i: jth feature of input i
- w_k^l: weight from neuron k to neuron l in next layer
- b_m: bias to neuron m in next layer

Exercises:

- write down W^1 and W^2 for the xor network.
- re-write the above as $f(xW)$ by adding a column of 1s to x and a new row to W.

Computing Many Examples

It's often very useful to compute over a 'batch' of inputs at once. The linear combination then turns into a *matrix multiplication*:

$$\vec{a} = f\left(\begin{bmatrix} x_0^0 & x_1^0 & \cdots & x_n^0 \\ x_0^1 & x_1^1 & \cdots & x_n^1 \\ \vdots & \vdots & \ddots & \vdots \\ x_0^m & x_1^m & \cdots & x_n^m \end{bmatrix} \begin{bmatrix} w_0^0 & w_0^1 & \cdots & w_0^l \\ w_1^0 & w_1^1 & \cdots & w_1^l \\ \vdots & \vdots & \ddots & \vdots \\ w_n^0 & w_n^1 & \cdots & w_n^l \end{bmatrix} + \begin{bmatrix} b_0 & b_1 & \cdots & b_l \\ b_0 & b_1 & \cdots & b_l \\ \vdots & \vdots & \ddots & \vdots \\ b_0 & b_1 & \cdots & b_l \end{bmatrix}\right)$$

$$= f(xW + b)$$

- x_j^i: jth feature of input i
- w_k^l: weight from neuron k to neuron l in next layer
- b_m: bias to neuron m in next layer

Exercises:

- write down W^1 and W^2 for the xor network.
- re-write the above as $f(xW)$ by adding a column of 1s to x and a new row to W.

Hidden Representations

Key idea: hidden layers of a neural network can encode high-level/abstract features of the input.

Outline

1. Recap
2. Formal Language Learning of GQs
3. Bayesian Learning
4. Introduction to Neural Networks
   - NNs: Computation
   - NNs: Learning
   - Hands-on Example

(Supervised) Learning

Where do the weights and biases come from? They can be *learned* from data to approximate a function.

Supervised learning (will talk about others later):

- Initialize the network randomly.
- Give the network a bunch of inputs.
- Compare its outputs *to the true outputs*.
- Update the weights and biases to move the network's outputs closer to the true outputs.

The last step is done via *gradient descent* (and refinements thereof).

(Supervised) Learning

Where do the weights and biases come from? They can be *learned* from data to approximate a function.

Supervised learning (will talk about others later):

- Initialize the network randomly.
- Give the network a bunch of inputs.
- Compare its outputs *to the true outputs.*
- Update the weights and biases to move the network's outputs closer to the true outputs.

The last step is done via *gradient descent* (and refinements thereof).

(Supervised) Learning

Where do the weights and biases come from? They can be *learned* from data to approximate a function.

Supervised learning (will talk about others later):

- Initialize the network randomly.
- Give the network a bunch of inputs.
- Compare its outputs *to the true outputs.*
- Update the weights and biases to move the network's outputs closer to the true outputs.

The last step is done via *gradient descent* (and refinements thereof).

(Supervised) Learning

Where do the weights and biases come from? They can be *learned* from data to approximate a function.

Supervised learning (will talk about others later):

- Initialize the network randomly.
- Give the network a bunch of inputs.
- Compare its outputs *to the true outputs*.
- Update the weights and biases to move the network's outputs closer to the true outputs.

The last step is done via *gradient descent* (and refinements thereof).

(Supervised) Learning

Where do the weights and biases come from? They can be *learned* from data to approximate a function.

Supervised learning (will talk about others later):

- Initialize the network randomly.
- Give the network a bunch of inputs.
- Compare its outputs *to the true outputs*.
- Update the weights and biases to move the network's outputs closer to the true outputs.

The last step is done via *gradient descent* (and refinements thereof).

(Supervised) Learning

Where do the weights and biases come from? They can be *learned* from data to approximate a function.

Supervised learning (will talk about others later):

- Initialize the network randomly.
- Give the network a bunch of inputs.
- Compare its outputs *to the true outputs*.
- Update the weights and biases to move the network's outputs closer to the true outputs.

The last step is done via *gradient descent* (and refinements thereof).

Gradient Descent: Example

Task: predict a true value $y = 2$.
"Model": one parameter θ, outputs $\hat{y} = \theta$.
Loss function:

$$\mathcal{L}(\theta, y) = (\hat{y}(\theta) - y)^2$$

Gradient Descent: Example

Task: predict a true value $y = 2$.
"Model": one parameter θ, outputs $\hat{y} = \theta$.
Loss function:

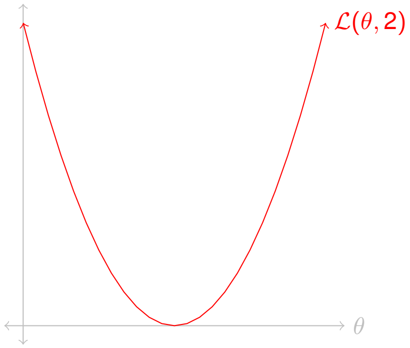
$$\mathcal{L}(\theta, y) = (\hat{y}(\theta) - y)^2$$

Gradient Descent: Example

Task: predict a true value $y = 2$.
"Model": one parameter θ, outputs $\hat{y} = \theta$.
Loss function:

$$\mathcal{L}(\theta, y) = (\hat{y}(\theta) - y)^2$$

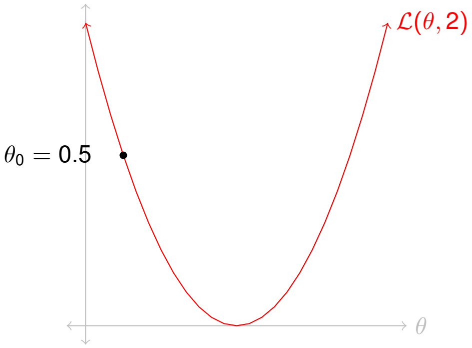
Gradient Descent: Example



$$\frac{\partial}{\partial \theta}\mathcal{L}(\theta, y) = 2(\theta - y)$$

$$\theta_{t+1} = \theta_t - \alpha \cdot \frac{\partial}{\partial \theta}\mathcal{L}(\theta, y)$$

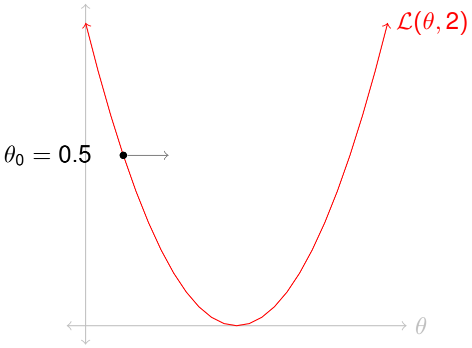
Gradient Descent: Example



$$\frac{\partial}{\partial \theta}\mathcal{L}(\theta, y) = 2(\theta - y)$$

$$\theta_{t+1} = \theta_t - \alpha \cdot \frac{\partial}{\partial \theta}\mathcal{L}(\theta, y)$$

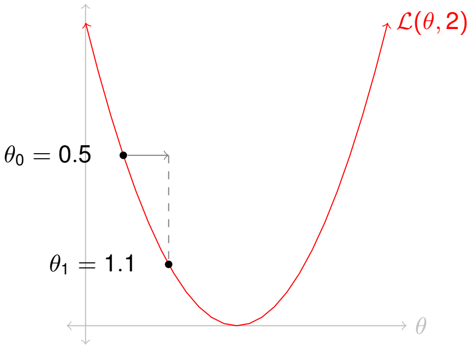
Gradient Descent: Example



$$\frac{\partial}{\partial\theta}\mathcal{L}(\theta, y) = 2(\theta - y)$$

$$\theta_{t+1} = \theta_t - \alpha \cdot \frac{\partial}{\partial\theta}\mathcal{L}(\theta, y)$$

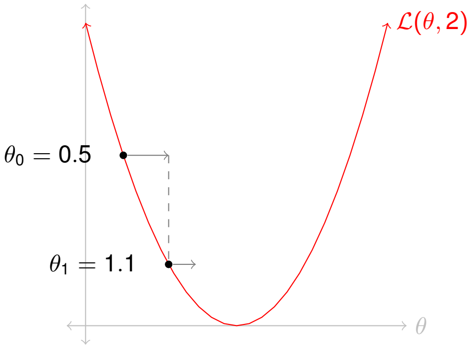
Gradient Descent: Example



$$\frac{\partial}{\partial\theta}\mathcal{L}(\theta, y) = 2(\theta - y)$$

$$\theta_{t+1} = \theta_t - \alpha \cdot \frac{\partial}{\partial\theta}\mathcal{L}(\theta, y)$$

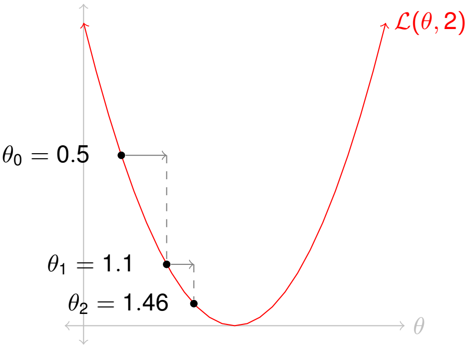
Gradient Descent: Example



$$\frac{\partial}{\partial \theta}\mathcal{L}(\theta, y) = 2(\theta - y)$$

$$\theta_{t+1} = \theta_t - \alpha \cdot \frac{\partial}{\partial \theta}\mathcal{L}(\theta, y)$$

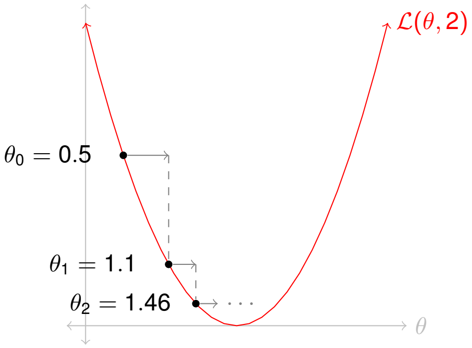
Gradient Descent: Example



$$\frac{\partial}{\partial\theta}\mathcal{L}(\theta, y) = 2(\theta - y)$$

$$\theta_{t+1} = \theta_t - \alpha \cdot \frac{\partial}{\partial\theta}\mathcal{L}(\theta, y)$$

Gradient Descent: Example



$$\frac{\partial}{\partial\theta}\mathcal{L}(\theta, y) = 2(\theta - y)$$

$$\theta_{t+1} = \theta_t - \alpha \cdot \frac{\partial}{\partial\theta}\mathcal{L}(\theta, y)$$

Gradient Descent for NNs

A neural network computes a complex function of its input. For an L-layer feed-forward network:

$$\hat{y}(x) = f_L(f_{L-1}(\cdots f_2(f_1(xW^1 + b^1)W^2 + b^2)\cdots)W^L + b^L)$$

All of the weights and biases form a long vector of parameters θ. So instead of a partial derivative, we take a *gradient*:

$$\nabla_\theta \mathcal{L}(\hat{y}(\theta), y) = \left\langle \frac{\partial}{\partial \theta_1}\mathcal{L}, \ldots, \frac{\partial}{\partial \theta_N}\mathcal{L} \right\rangle$$

The (negative) gradient tells us *which direction in 'parameter space'* to walk in order to make the loss ($\mathcal{L}$) smaller, i.e. to make the network's output closer to the true output.

Gradient Descent for NNs

A neural network computes a complex function of its input. For an *L*-layer feed-forward network:

$$\hat{y}(x) = f_L(f_{L-1}(\cdots f_2(f_1(xW^1 + b^1)W^2 + b^2)\cdots)W^L + b^L)$$

All of the weights and biases form a long vector of parameters θ. So instead of a partial derivative, we take a *gradient*:

$$\nabla_\theta \mathcal{L}(\hat{y}(\theta), y) = \left\langle \frac{\partial}{\partial \theta_1}\mathcal{L}, \ldots, \frac{\partial}{\partial \theta_N}\mathcal{L} \right\rangle$$

The (negative) gradient tells us *which direction in 'parameter space'* to walk in order to make the loss ($\mathcal{L}$) smaller, i.e. to make the network's output closer to the true output.

Gradient Descent for NNs

A neural network computes a complex function of its input. For an *L*-layer feed-forward network:

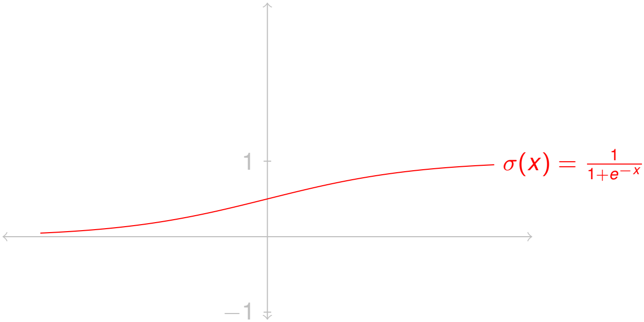
$$\hat{y}(x) = f_L(f_{L-1}(\cdots f_2(f_1(xW^1 + b^1)W^2 + b^2)\cdots)W^L + b^L)$$

All of the weights and biases form a long vector of parameters θ. So instead of a partial derivative, we take a *gradient*:

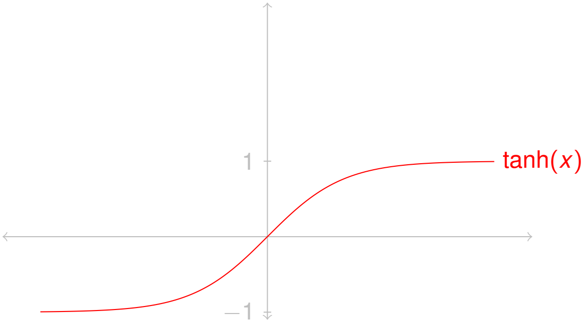
$$\nabla_\theta \mathcal{L}(\hat{y}(\theta), y) = \left\langle \frac{\partial}{\partial \theta_1}\mathcal{L}, \ldots, \frac{\partial}{\partial \theta_N}\mathcal{L} \right\rangle$$

The (negative) gradient tells us *which direction in 'parameter space'* to walk in order to make the loss ($\mathcal{L}$) smaller, i.e. to make the network's output closer to the true output.

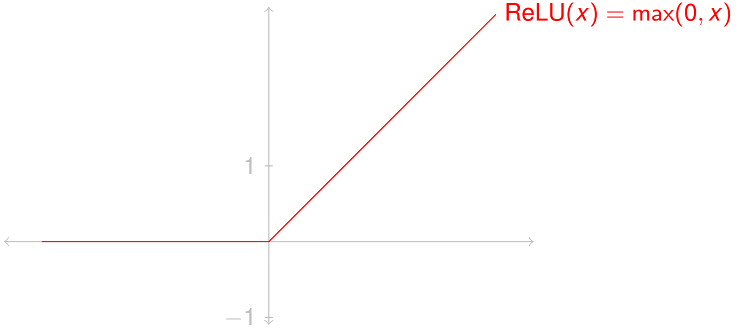
Activation Functions



Activation Functions



Activation Functions



ReLUs are incredibly popular at the moment, as are refinements: softplus, leaky ReLU, exponential linear (ELU), gaussian linear (GLU), . . .

Glorot, Bordes, and Bengio 2011; Hahnioser, Sarpeshkar, Mahowald, Douglas, and Seung 2000

Loss Functions

Regression: different kinds of geometrical distances, e.g.

$$\ell(\hat{y}, y) = (\hat{y} - y)^2$$

Classification: *cross entropy*:

$$\ell(\hat{y}, y) = -\sum y_i \cdot \ln(\hat{y}_i) = -\ln(\hat{y}_{\text{true class}})$$

Loss Functions

Regression: different kinds of geometrical distances, e.g.

$$\ell(\hat{y}, y) = (\hat{y} - y)^2$$

Classification: *cross entropy*:

$$\ell(\hat{y}, y) = -\sum y_i \cdot \ln(\hat{y}_i) = -\ln(\hat{y}_{\text{true class}})$$

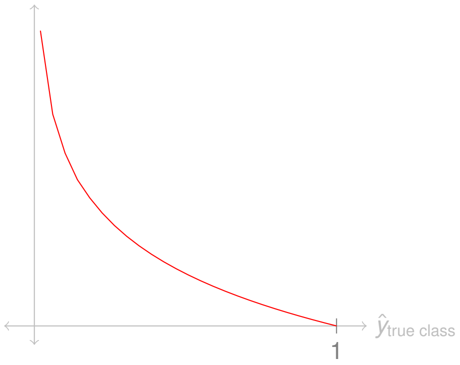
Loss Functions

Regression: different kinds of geometrical distances, e.g.

$$\ell(\hat{y}, y) = (\hat{y} - y)^2$$

Classification: *cross entropy*:

$$\ell(\hat{y}, y) = -\sum y_i \cdot \ln(\hat{y}_i) = -\ln(\hat{y}_{\text{true class}})$$



Example of Learned Hidden Layers

Edges (layer conv2d0) **Textures** (layer mixed3a) **Patterns** (layer mixed4a)

Olah, Mordvintsev, and Schubert 2017; Yosinski, Clune, Bengio, and Lipson 2014
https://distill.pub/2017/feature-visualization/

Learned Representations

Key idea: a neural network can learn *which* high-level/abstract features of the input are useful in helping it solve its task. (Features are learned, instead of engineered by us.)

Outline

Anatomy of a DL Experiment

(1) Specify parameters
(2) Build data input/generation pipeline
  - Train/test split [dev as well; more later]
(3) Build model
(4) Train the model!
  (a) Evaluate at regular intervals
  (b) Measure your variables of interest
  (c) Monitor train/test loss
  (d) Early stopping [later in tutorial]
(5) Analysis
  - quantitative
  - qualitative behavior

NOTE: keep detailed records about what you're doing!
For guidance on keeping records and writing up results, see these lecture notes from Sam Bowman.

Anatomy of a DL Experiment

(1) Specify parameters

(2) Build data input/generation pipeline

- Train/test split [dev as well; more later]

(3) Build model

(4) Train the model!

  (a) Evaluate at regular intervals
  (b) Measure your variables of interest
  (c) Monitor train/test loss
  (d) Early stopping [later in tutorial]

(5) Analysis

- quantitative
- qualitative behavior

NOTE: keep detailed records about what you're doing!
For guidance on keeping records and writing up results, see these lecture notes from Sam Bowman.

Anatomy of a DL Experiment

(1) Specify parameters
(2) Build data input/generation pipeline
  - Train/test split [dev as well; more later]
(3) Build model
(4) Train the model!
  (a) Evaluate at regular intervals
  (b) Measure your variables of interest
  (c) Monitor train/test loss
  (d) Early stopping [later in tutorial]
(5) Analysis
  - quantitative
  - qualitative behavior

NOTE: keep detailed records about what you're doing!
For guidance on keeping records and writing up results, see these lecture notes from Sam Bowman.

Anatomy of a DL Experiment

(1) Specify parameters

(2) Build data input/generation pipeline

- Train/test split [dev as well; more later]

(3) Build model

(4) Train the model!

    (a) Evaluate at regular intervals
    (b) Measure your variables of interest
    (c) Monitor train/test loss
    (d) Early stopping [later in tutorial]

(5) Analysis

- quantitative
- qualitative behavior

NOTE: keep detailed records about what you're doing!
For guidance on keeping records and writing up results, see these lecture notes from Sam Bowman.

Anatomy of a DL Experiment

(1) Specify parameters
(2) Build data input/generation pipeline
    - Train/test split [dev as well; more later]
(3) Build model
(4) Train the model!
    (a) Evaluate at regular intervals
    (b) Measure your variables of interest
    (c) Monitor train/test loss
    (d) Early stopping [later in tutorial]
(5) Analysis
    - quantitative
    - qualitative behavior

NOTE: keep detailed records about what you're doing!
For guidance on keeping records and writing up results, see these lecture notes from Sam Bowman.

Anatomy of a DL Experiment

(1) Specify parameters
(2) Build data input/generation pipeline
    - Train/test split [dev as well; more later]
(3) Build model
(4) Train the model!
    (a) Evaluate at regular intervals
    (b) Measure your variables of interest
    (c) Monitor train/test loss
    (d) Early stopping [later in tutorial]
(5) Analysis
    - quantitative
    - qualitative behavior

NOTE: keep detailed records about what you're doing!
For guidance on keeping records and writing up results, see these lecture notes from Sam Bowman.

To the code!

https://github.com/shanest/nn-tutorial/blob/master/tutorial.ipynb