Tutorial Introduction to Neural Networks with an eye towards linguistic applications

Shane Steinert-Threlkeld

January 25, 2019

European Research Council
Established by the European Commission

Today's Plan

1. Neural Networks: computation
2. Neural Networks: learning
3. Hands-on experiment: learning quantifiers
4. Some practical tips
5. Further Topics + Resources

Today's Plan

1. Neural Networks: computation
2. Neural Networks: learning
3. Hands-on experiment: learning quantifiers
4. Some practical tips
5. Further Topics + Resources

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

What I'm Presupposing

Some mathematical notation/concepts from:

- Linear algebra (matrix multiplication, e.g.)
- Multivariate calculus (partial derivatives)
- Logic / formal semantics of natural language

What I'm Presupposing

Some mathematical notation/concepts from:

- Linear algebra (matrix multiplication, e.g.)
- Multivariate calculus (partial derivatives)
- Logic / formal semantics of natural language

Some programming experience:

- Basics of Python
- Basic syntax in NumPy

What I'm Presupposing

Some mathematical notation/concepts from:

- Linear algebra (matrix multiplication, e.g.)
- Multivariate calculus (partial derivatives)
- Logic / formal semantics of natural language

Some programming experience:

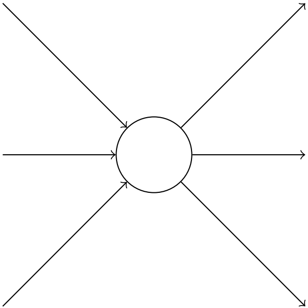
- Basics of Python
- Basic syntax in NumPy

But: no formal requirements; all concepts and syntax can be explained intuitively, so please ask for clarification at all points!

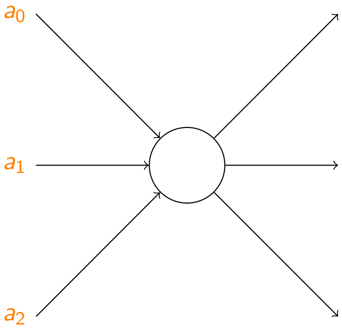
Materials: Slides + Jupyter Notebook

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

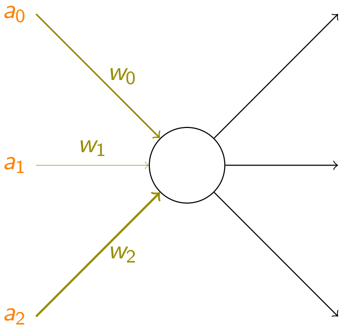
Artificial Neuron



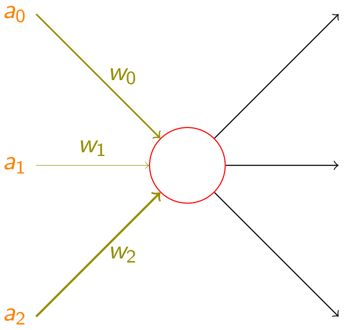
Artificial Neuron



Artificial Neuron

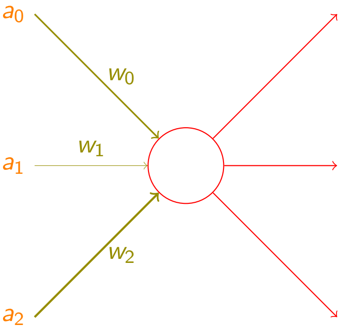


Artificial Neuron



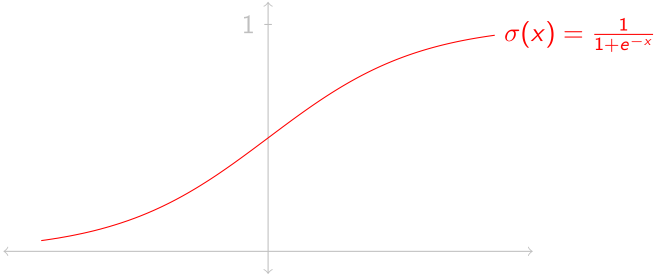
$$a = f\left(a_0 \cdot w_0 + a_1 \cdot w_1 + a_2 \cdot w_2\right)$$

Artificial Neuron



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

Activation Function

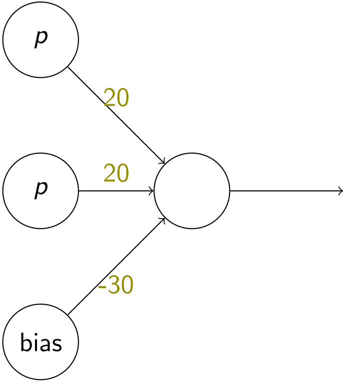


More on choosing activation functions later in the tutorial.

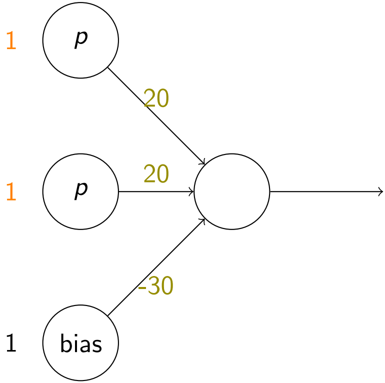
Computing ‘and’

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

Computing ‘and’

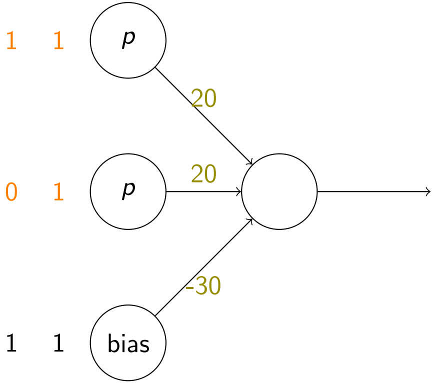


Computing ‘and’



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

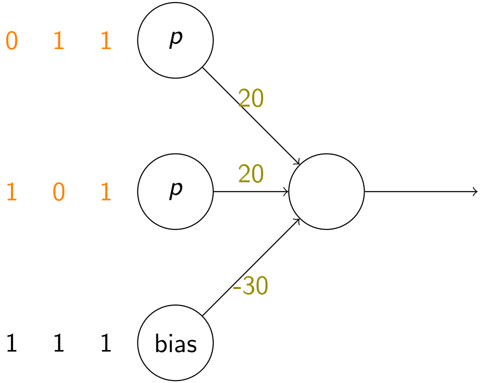
Computing ‘and’



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

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

Computing ‘and’

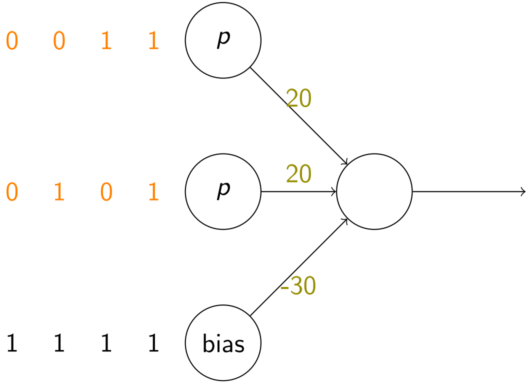


$$a = \sigma(1 \cdot 20 + 1 \cdot 20 + 1 \cdot -30) = \sigma(10) \approx 1$$

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

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

Computing ‘and’



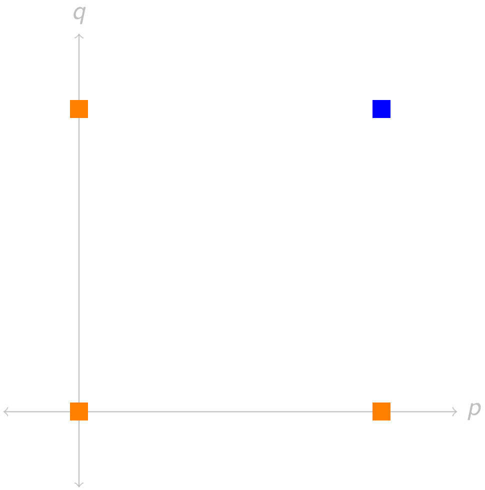
$$a = \sigma(1 \cdot 20 + 1 \cdot 20 + 1 \cdot -30) = \sigma(10) \approx 1$$

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

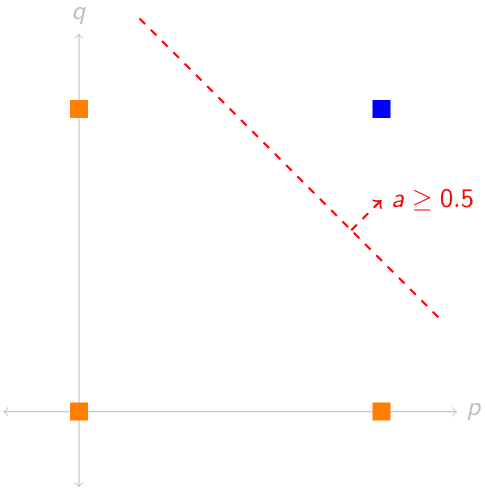
$$a = \sigma(0 \cdot 20 + 1 \cdot 20 + 1 \cdot -30) = \sigma(-10) \approx 0$$

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

Computing ‘and’



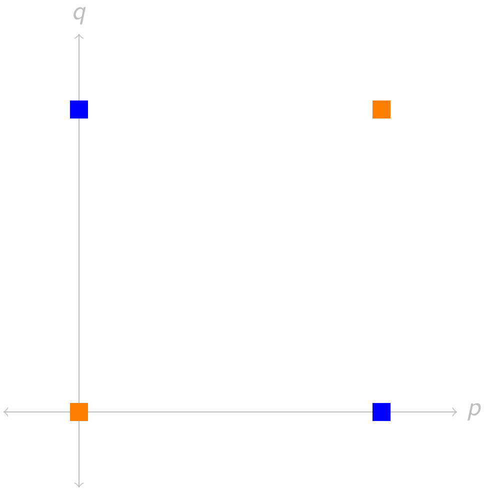
Computing 'and'



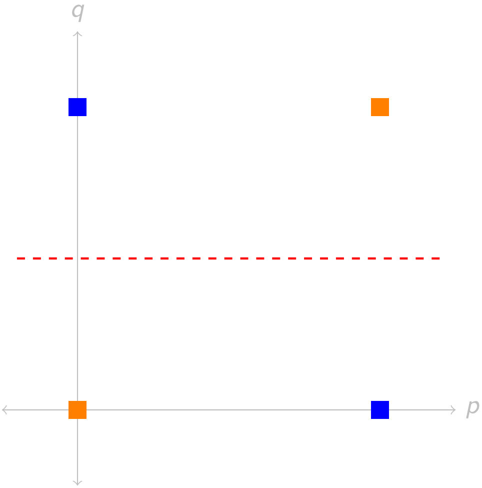
Computing 'xor'

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

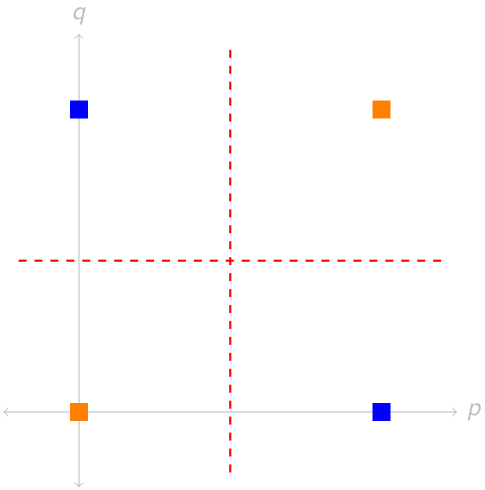
Computing 'xor'



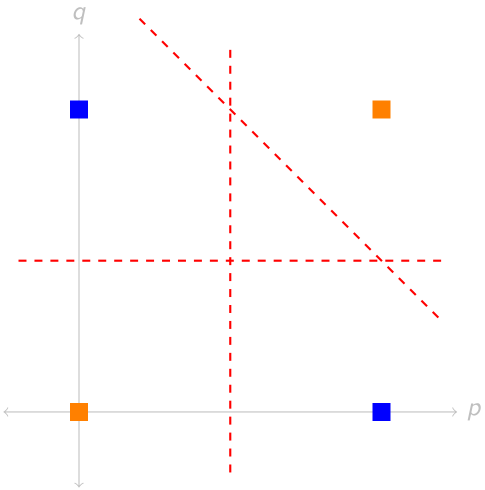
Computing ‘xor’



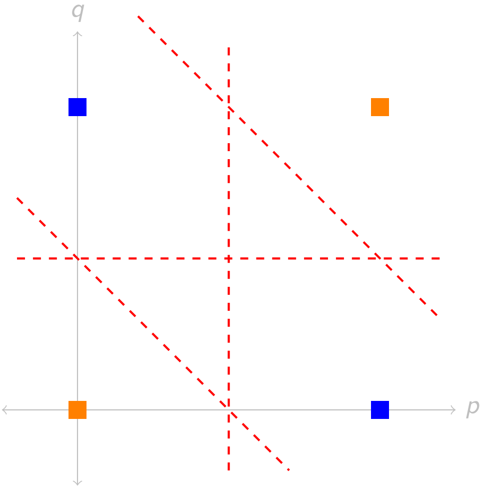
Computing ‘xor’



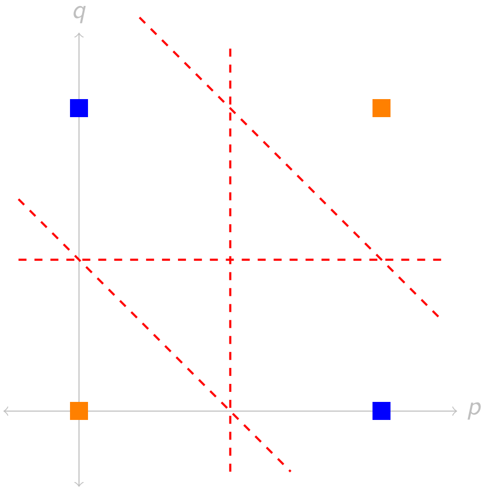
Computing 'xor'



Computing ‘xor’

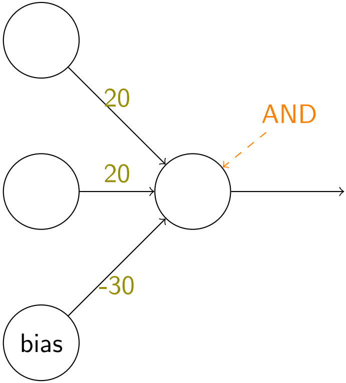


Computing 'xor'

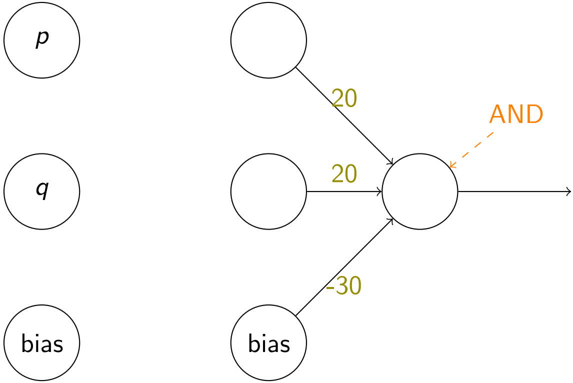


xor is not *linearly separable*

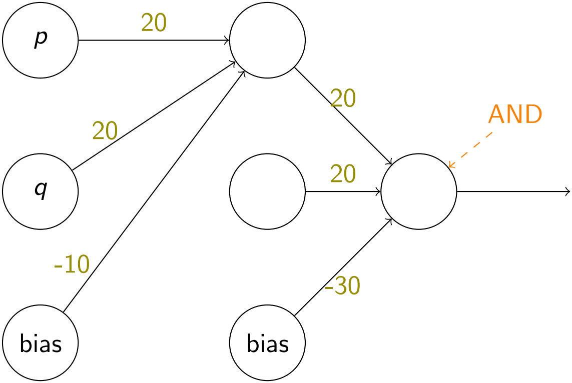
Computing ‘xor’



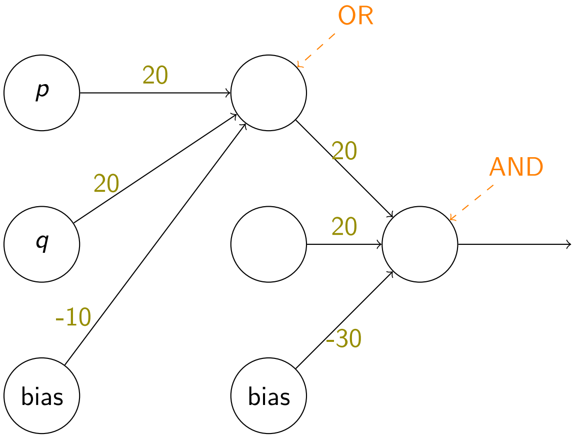
Computing 'xor'



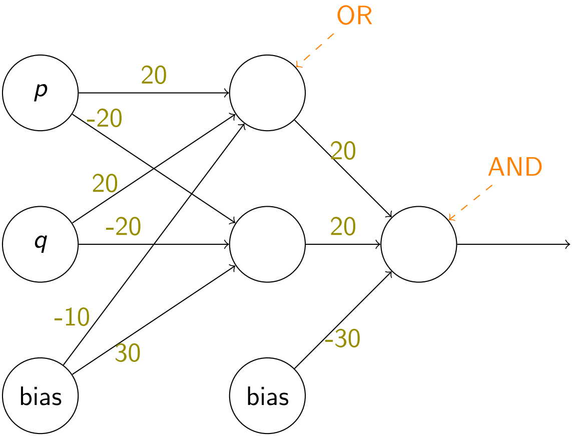
Computing 'xor'



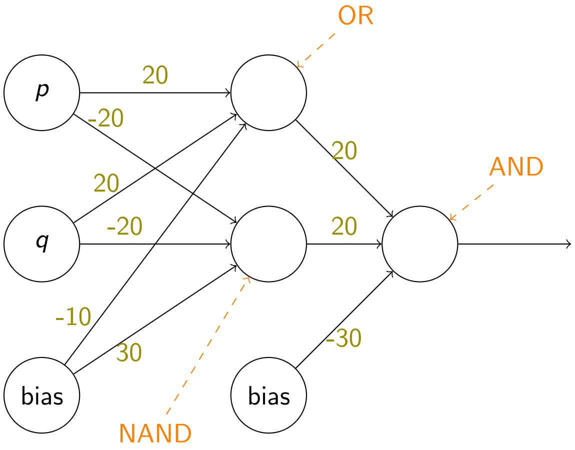
Computing ‘xor’



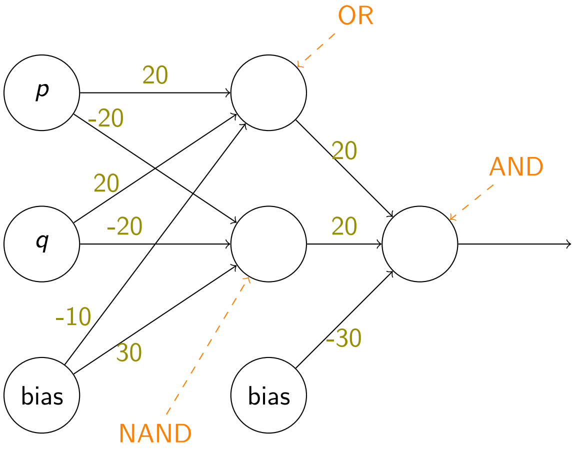
Computing 'xor'



Computing 'xor'

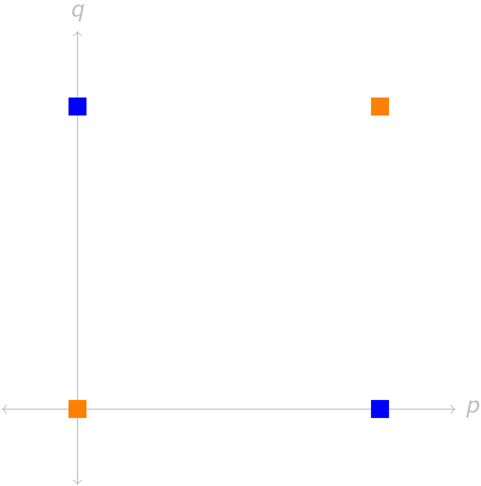


Computing 'xor'

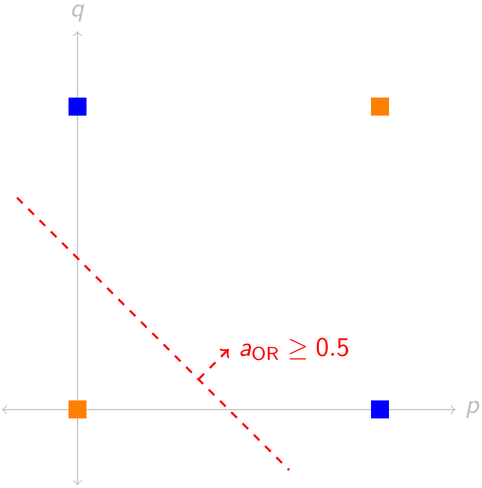


Exercise: show that the hidden units behave as labeled.

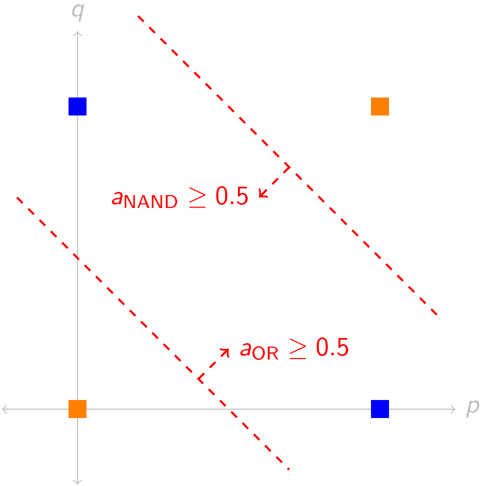
Computing ‘xor’



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*:

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

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

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.

(Supervised) Learning

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

(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.

(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.

(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*.

(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.

(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 $y = 2$.

Gradient Descent: Example

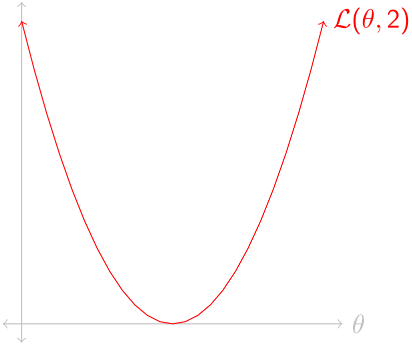
Task: predict a true value $y = 2$.
"Model": one parameter θ, outputs $\hat{y} = \theta$.

Gradient Descent: Example

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

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

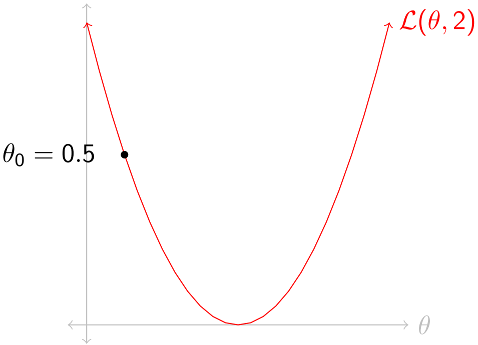
Gradient Descent: Example



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

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

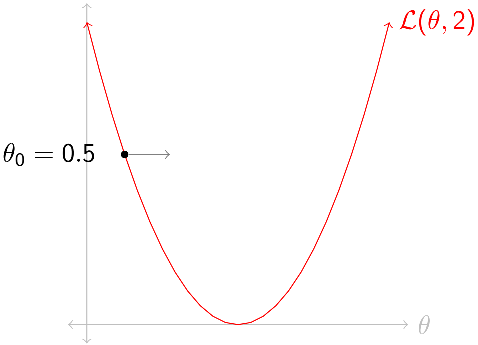
Gradient Descent: Example



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

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

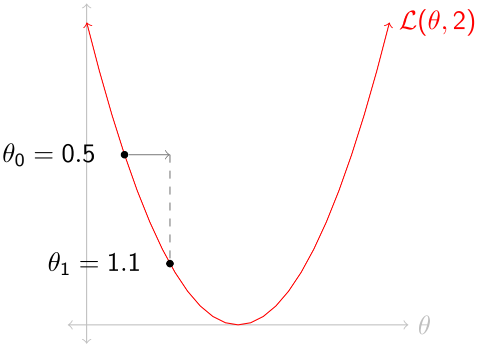
Gradient Descent: Example



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

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

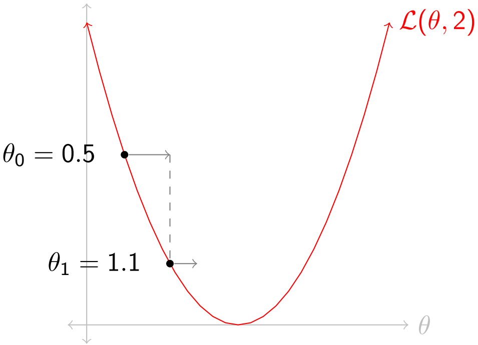
Gradient Descent: Example



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

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

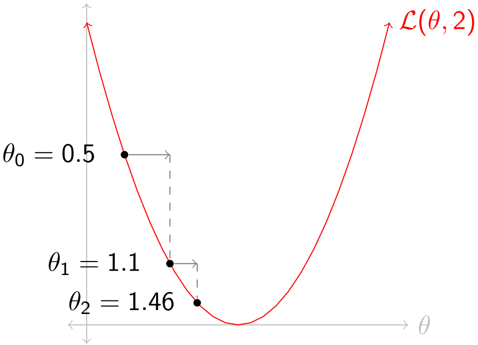
Gradient Descent: Example



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

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

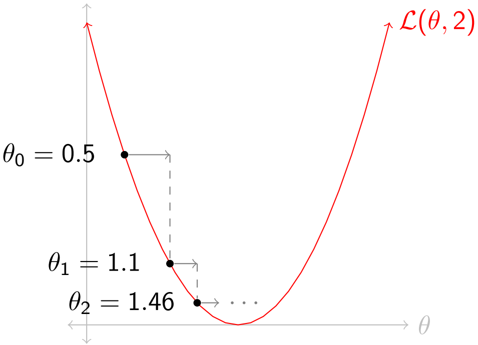
Gradient Descent: Example



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

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

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:

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

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*:

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

Gradient Descent for NNs

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

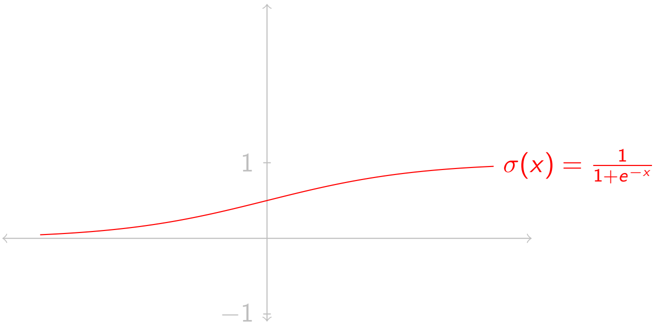
$$\hat{y}(x) = f_L(f_{L-1}(\cdots f_2(f_1(xW^1 + b^1)W^2 + b^2)\cdots)W^L + b^L)$$

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

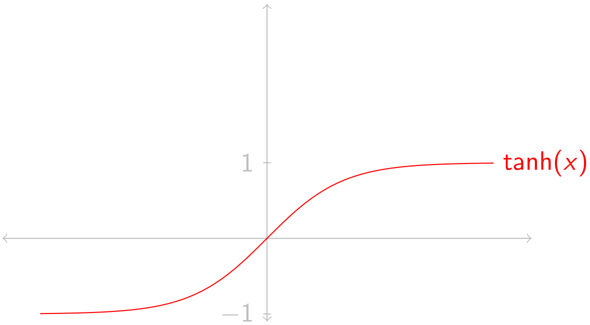
$$\nabla_\theta \mathcal{L}(\hat{y}(\theta), y) = \left\langle \frac{\partial}{\partial \theta_1}\mathcal{L}, \ldots, \frac{\partial}{\partial \theta_N}\mathcal{L} \right\rangle$$

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.

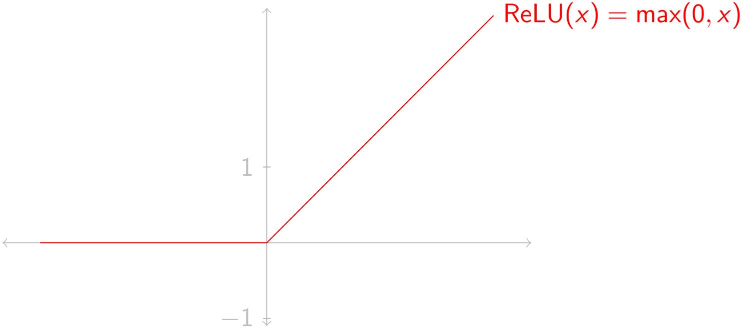
Activation Functions



Activation Functions



Activation Functions



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

Hahnioser et al. 2000; Glorot, Bordes, and Bengio 2011

Loss Functions

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

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

Loss Functions

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

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

Classification: *cross entropy*:

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

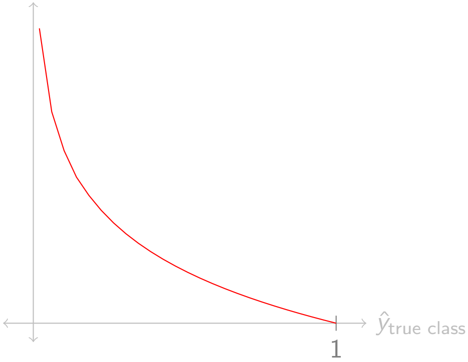
Loss Functions

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 et al. 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.)

Anatomy of a DL Experiment

1. Specify parameters

Anatomy of a DL Experiment

1. Specify parameters
2. Build data input/generation pipeline

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

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!

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!
    1. Evaluate at regular intervals
    2. Measure your variables of interest
    3. Monitor train/test loss
    4. Early stopping [later in tutorial]
5. Analyze
    - quantitative
    - qualitative behavior

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!
    1. Evaluate at regular intervals
    2. Measure your variables of interest
    3. Monitor train/test loss
    4. Early stopping [later in tutorial]
5. Analyze
    - 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

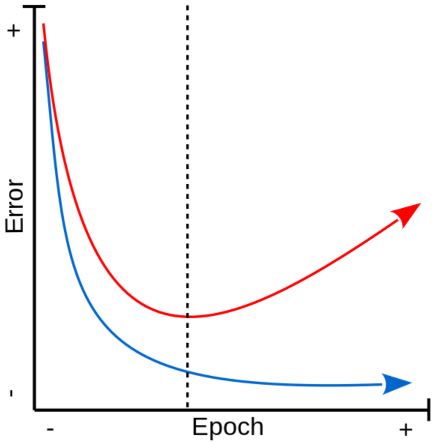
Balancing the Data

For classification tasks, it's very important to *balance the training data*, so that it contains (roughly) equal numbers of examples for each class. Otherwise, a network can very quickly get stuck in a local minimum, where it uniformly guesses the most-frequent class.
Two methods:

1. Downsampling: randomly sample as many examples from each class as in your least-frequent class
2. Upsampling: repeat examples from your less-frequent classes until you have as many as the most-frequent

Over-fitting: detection

Neural networks, especially very large, deep ones, run the risk of *over-fitting* ("memorizing") their training data.
Evidence: training loss continues to go down, while testing loss rises.

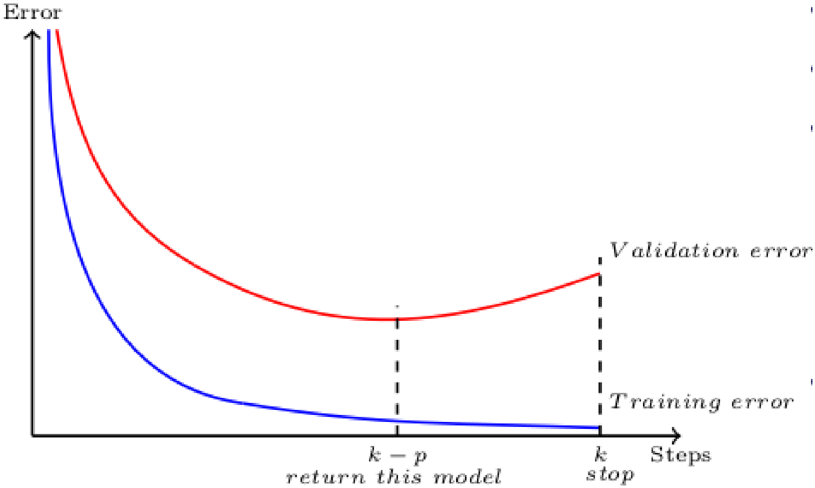


Early Stopping

How do you know how many epochs to train for? One common method: A LOT, but put in a condition for *early stopping*.

Early Stopping

How do you know how many epochs to train for? One common method: A LOT, but put in a condition for *early stopping*.



Early Stopping

How do you know how many epochs to train for? One common method: A LOT, but put in a condition for *early stopping*.

- Record test performance every N epochs
- Save best performing model.
- If performance doesn't improve after P epochs, quit training. [Note: P for 'patience']

Over-fitting: avoidance (regularization)

Two very popular and successful *regularization* techniques to combat over-fitting (and generally make your life better):

- L2 regularization: add a term $\lambda||\theta||_2^2$ to loss function.
- Dropout (Srivastava et al. 2014): randomly 'turn off' (set to 0) a certain percentage of input nodes to this layer
- Batch normalization (Ioffe and Szegedy 2015): scale the inputs to the layer so that they're roughly averaged around 0 and normally distributed.
  (I've found BN *very* useful in my own research.)

Note: `.eval()` and `.train()` on a PyTorch `nn.Module` turn these sorts of methods off/on based on whether you are in training or 'inference' / prediction mode.

Hyper-parameter Tuning

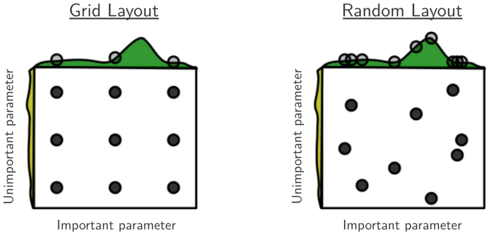
How do you decide how to set the parameters of your experiment? There are so many knobs to turn! (Listed roughly in order of importance of turning them.)

1. Network architecture: depth (number of layers) and width (size of layers)
   [or even different network types altogether]
2. Activation functions
3. Optimizers
   - Learning rates + other parameters here
4. ...

Hyper-parameter Tuning (cont.)

Main idea: try a whole bunch of settings! Choose the setting that performs best. Some notes:

- This requires a third set, a *development (dev)* set, in addition to training/testing.
  You choose the best hyper-parameters based on best performance (however measured) on the *dev* set. Then you evaluate that model on the *test* set.
- *Random* search in parameter space better than 'grid' search:



Bergstra and Bengio 2012

Scaling Up

Once models and datasets become non-trivially sized, your personal computer likely won't suffice to run experiments.

A common paradigm: prototype locally, experiment externally.

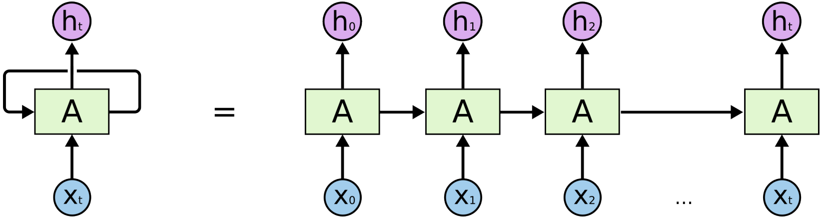
Two most critical components:

- Large RAM for storing big models and datasets
- GPUs (Graphics Processing Units)!
  These are made to massively parallelize linear algebra of the kind computed by networks, so can dramatically speed up training and inference.

If you can't access these things via your institution, you can buy compute time on appropriate machines via Amazon Web Services or Google Cloud Platform.

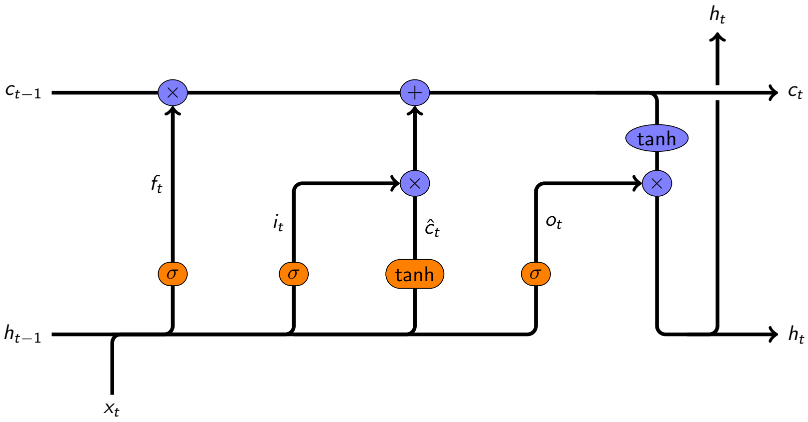
Processing Sequences: Recurrent Neural Networks

Processing sequences (of variable length) is crucial for applying neural networks to text!



http://karpathy.github.io/2015/05/21/rnn-effectiveness/
http://colah.github.io/posts/2015-08-Understanding-LSTMs/

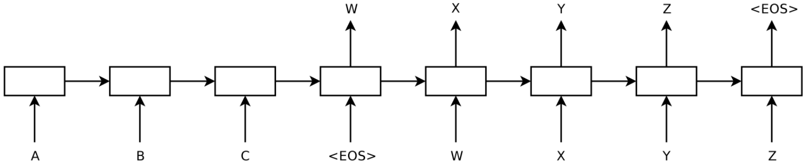
Long Short-Term Memory Network



Hochreiter and Schmidhuber 1997

Sequence-to-Sequence Models

Widely used in machine translation, dependency parsing, semantic parsing, . . .



Sutskever, Vinyals, and Le 2014

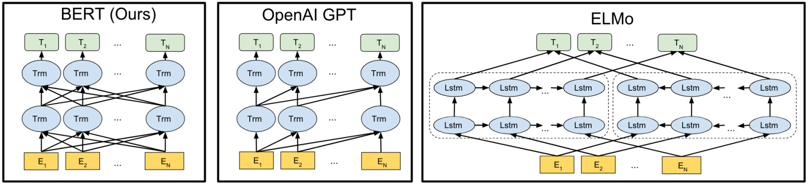
Semi-Supervised + Transfer Learning

- (Un-/Semi-)supervised: no 'true output', predict some piece of the input from some other piece.
  E.g.: *language modeling*, predict $P(w_t | w_{t-k}, \ldots, w_{t-1})$.

Semi-Supervised + Transfer Learning

- (Un-/Semi-)supervised: no 'true output', predict some piece of the input from some other piece.
  E.g.: *language modeling*, predict $P(w_t|w_{t-k}, \ldots, w_{t-1})$.
- Transfer learning: train a model on one task, 'fine tune' it on the task you care about.
  Very successful in computer vision.
  "NLP's ImageNet moment has arrived."
  (https://thegradient.pub/nlp-imagenet/)

Transfer Learning in NLP



Devlin et al. 2018; Radford et al. 2018; Peters et al. 2018
http://jalammar.github.io/illustrated-bert/

Transfer Learning in NLP

	Rank	Name	Model	URL	Score	CoLA	SST-2	MRPC	STS-B	QQP	MNLI-m	MNLI-mm	QNLI	RTE	WNLI	AX
	1	Microsoft D365 AI & MSR AI	BIGBIRD		81.9	61.5	94.5	90.0/86.7	88.3/87.7	71.0/89.2	86.8	86.0	97.6	75.2	65.1	39.7
+	2	Jacob Devlin	BERT: 24-layers, 1024-hidden, 16-heads		80.4	60.5	94.9	89.3/85.4	87.6/86.5	72.1/89.3	86.7	85.9	91.1	70.1	65.1	39.6
	3	Jason Phang	GPT on STILTs		76.9	47.2	93.1	87.7/83.7	85.3/84.8	70.1/88.1	80.7	80.6	87.2	69.1	65.1	29.4
	4	Alec Radford	Singletask Pretrain Transformer		72.8	45.4	91.3	82.3/75.7	82.0/80.0	70.3/88.5	82.1	81.4	88.1	56.0	53.4	29.8
+	5	Samuel Bowman	BiLSTM+ELMo+Attn		70.5	36.0	90.4	84.9/77.9	75.1/73.3	64.8/84.7	76.4	76.1	79.9	56.8	65.1	26.5
	6	GLUE Baselines	BiLSTM+ELMo+Attn		68.9	18.9	91.6	83.5/77.3	72.8/71.1	63.3/83.5	75.6	75.9	81.7	61.2	65.1	22.6

https://gluebenchmark.com/

Further Resources

General references:

- Nielsen 2015
- Goodfellow, Bengio, and Courville 2016
- http://3blue1brown.net/neural-networks

Further Resources

General references:

- Nielsen 2015
- Goodfellow, Bengio, and Courville 2016
- http://3blue1brown.net/neural-networks

Bleeding edge papers:

- arXiv [cs.CL, cs.LG, cs.AI]
- Conference proceedings: ACL, EMNLP, NAACL, NeurIPS, ICML, ICLR

Further Resources

General references:

- Nielsen 2015
- Goodfellow, Bengio, and Courville 2016
- http://3blue1brown.net/neural-networks

Bleeding edge papers:

- arXiv [cs.CL, cs.LG, cs.AI]
- Conference proceedings: ACL, EMNLP, NAACL, NeurIPS, ICML, ICLR

Helpful, often interactive, articles: Distill (https://distill.pub), The Gradient (https://thegradient.pub)

Further Resources

General references:

- Nielsen 2015
- Goodfellow, Bengio, and Courville 2016
- http://3blue1brown.net/neural-networks

Bleeding edge papers:

- arXiv [cs.CL, cs.LG, cs.AI]
- Conference proceedings: ACL, EMNLP, NAACL, NeurIPS, ICML, ICLR

Helpful, often interactive, articles: Distill (https://distill.pub), The Gradient (https://thegradient.pub)

Google is your friend (and GitHub too)! Blog posts (incl. from places like OpenAI, DeepMind, Google Brain, AllenAI) are very popular in the DL world, often including break-downs and sample implementations of models from papers.

References I

- Bergstra, James and Yoshua Bengio (2012). "Random Search for Hyper-Parameter Optimization". In: *Journal of Machine Learning Research* 13, pp. 281–305.
- Devlin, Jacob et al. (2018). "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding".
- Glorot, Xavier, Antoine Bordes, and Yoshua Bengio (2011). "Deep Sparse Rectifier Neural Networks". In: *14th International Conference on Artificial Intelligence and Statistics (AISTATS)*, pp. 315–323.
- Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*. The MIT Press. URL: https://www.deeplearningbook.org/.
- Hahnioser, Richard H.R. et al. (2000). "Digital selection and analogue amplification coexist in a cortex- inspired silicon circuit". In: *Nature* 405.6789, pp. 947–951. ISSN: 00280836. DOI: 10.1038/35016072.

References II

Hochreiter, Sepp and Jürgen Schmidhuber (1997). "Long Short-Term Memory". In: *Neural Computation* 9.8, pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735.

Ioffe, Sergey and Christian Szegedy (2015). "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: arXiv: 1502.03167. URL: http://arxiv.org/abs/1502.03167.

Nielsen, Michael A (2015). *Neural Networks and Deep Learning*. Determination Press. URL: http://neuralnetworksanddeeplearning.com/.

Olah, Chris, Alexander Mordvintsev, and Ludwig Schubert (2017). "Feature Visualization: How neural networks build up their understanding of images". In: *Distill*.

References III

- Peters, Matthew E. et al. (2018). "Deep contextualized word representations". In: *Proceedings of North American Association for Computational Linguistics (NAACL)*. arXiv: 1802.05365. URL: http://arxiv.org/abs/1802.05365.
- Radford, Alec et al. (2018). "Improving Language Understanding by Generative Pre-Training".
- Srivastava, Nitish et al. (2014). "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15, pp. 1929–1958. DOI: 10.1214/12-AOS1000.
- Sutskever, Ilya, Oriol Vinyals, and Quoc V. Le (2014). "Sequence to Sequence Learning with Neural Networks". In: *Advances in Neural Information Processing Systems (NIPS 27)*. Ed. by Zoubin Ghahramani et al. arXiv: 1409.3215. URL: https://arxiv.org/abs/1409.3215.
- Yosinski, Jason et al. (2014). "How transferable are features in deep neural networks?". In: *Advances in Neural Information Processing Systems 27 (NIPS 2014)*. arXiv: 1411.1792v1.