Machine Learning for NLP
Lecture 7: Neural networks

UNIVERSITY OF GOTHENBURG

Richard Johansson

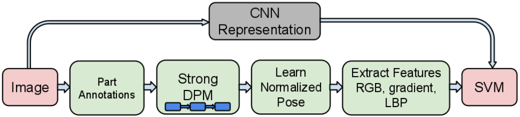
October 8, 2015

the "deep learning tsunami"

- in several fields, such as speech and image processing, neural network or "deep learning" models have led to dramatic improvements
- Manning: "*2015 seems like the year when the full force of the [deep learning] tsunami hit the major NLP conferences*"
- out of the machine learning community: "*NLP is kind of like a rabbit in the headlights of the deep learning machine, waiting to be flattened*"
- so, what's the hype about?

overview

- neural networks (NNs) are systems that learn to form useful abstractions automatically
  - learn to form larger units from small pieces
- appealing because it can reduce the feature engineering effort
  - image borrowed from Josephine Sullivan:



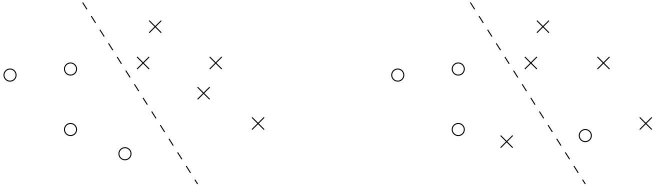
- NNs are excellent for “noisy” problems such as speech and image processing
- while powerful, they can be cumbersome to train and tend to require quite a bit of tweaking

causes of the NN resurgence

- NNs seem to have a hype cycle of about 20 years
- there are a number of reasons for the one we're currently in
- the most important is increasing computational capacity
  - for instance, the famous "cat paper" by Stanford/Google required 1,000 machines (16,000 CPUs)
    - Le et al: *Building high-level features using large scale unsupervised learning*, ICML 2011.
  - much of the recent research is coming out of Google (DeepMind), Microsoft, Facebook, etc.
  - using GPUs from graphics cards can speed up training
- also, a number of new methods proposed recently

recap: linear separability

- some datasets can't be modeled with a linear classifier!



- a dataset is **linearly separable** if there exists a $\boldsymbol{w}$ that gives us perfect classification

example: XOR dataset

```
X = numpy.array([[1, 1],
                 [1, 0],
                 [0, 1],
                 [0, 0]])
Y = ['no', 'yes', 'yes', 'no']

clf = LinearSVC()
clf.fit(X, Y)

# linear inseparability, so we get less than 100% accuracy
print(accuracy_score(Y, clf.predict(X)))
```

"abstraction" by forming feature combinations

- recall from last lecture: we may add "useful combinations" of features to make the dataset separable:

very good	very-good	Positive
very bad	very-bad	Negative
not good	not-good	Negative
not bad	not-bad	Positive

example: XOR dataset with a combination feature

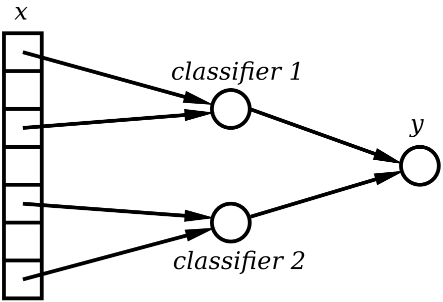
```
# feature1, feature2, feature1&feature2
X = numpy.array([[1, 1, 1],
                 [1, 0, 0],
                 [0, 1, 0],
                 [0, 0, 0]])
Y = ['no', 'yes', 'yes', 'no']

clf = LinearSVC()
clf.fit(X, Y)

# now we have linear separability, so we get 100%
print(accuracy_score(Y, clf.predict(X)))
```

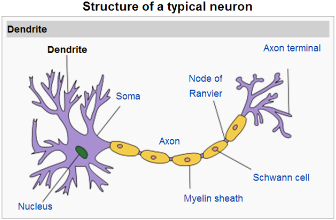
expressing feature combinations as "sub-classifiers"

- instead of defining a rule, such as $x_3 = x_1$ AND x_2, we could imagine that the combination feature x_3 would be computed by a separate classifier, for instance LR
- we could train a classifier using the output of "sub-classifiers"



"neurons"

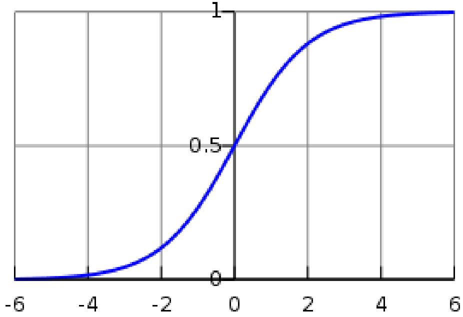
- historically, NNs were inspired by how biological neural systems work – hence the name
- as far as I know, modern NNs and modern neuroscience don't have much in common



- Andrew Ng: "*A single neuron in the brain is an incredibly complex machine that even today we don't understand. A single 'neuron' in a neural network is an incredibly simple mathematical function that captures a minuscule fraction of the complexity of a biological neuron. So to say neural networks mimic the brain, that is true at the level of loose inspiration, but really artificial neural networks are nothing like what the biological brain does.*"

recap: the logistic or sigmoid function

```
def logistic(scores):
    return 1 / (1 + numpy.exp(-scores))
```



a multilayered classifier

- a **feedforward neural network** or **multilayer perceptron** consists of connected layers of “classifiers”
  - the intermediate classifiers are called **hidden units**
  - the final classifier is called the **output unit**
- let’s assume two layers for now
- each hidden unit h_i computes its output based on its own weight vector $\boldsymbol{w}_{h_i}$:

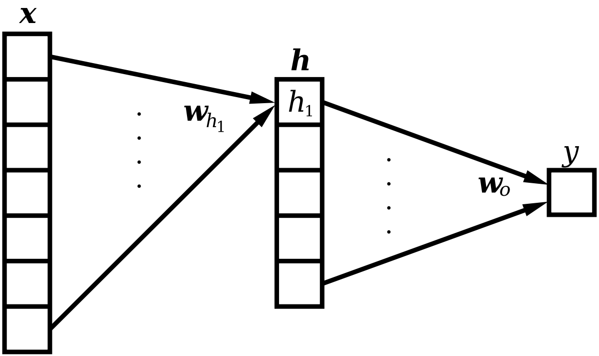
$$h_i = f(\boldsymbol{w}_{h_i} \cdot \boldsymbol{x})$$

- and then the output is computed from the hidden units:

$$y = f(\boldsymbol{w}_o \cdot \boldsymbol{h})$$

- the function f is called the **activation**
  - in this lecture, we’ll assume that f is the logistic function, so the hidden units and output unit can be seen as LR classifiers

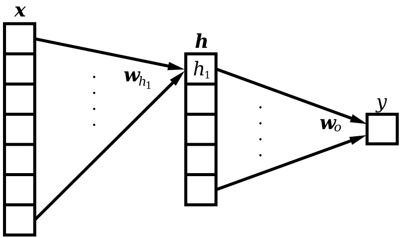
two-layered feedforward NN: figure



implementation in NumPy

- recall that a sequence of dot products can be seen as a matrix multiplication
- in NumPy, the NN can be expressed compactly with matrix multiplication

```
h = logistic(Wh.dot(x))
y = logistic(Wo.dot(h))
```



expressivity of feedforward NNs

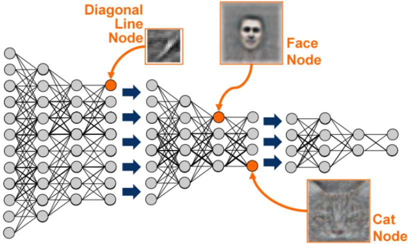
- Hornik's **universal approximation theorem** shows that feedforward NNs can approximate any (bounded) mathematical function
  - Hornik (1991). *Approximation capabilities of multilayer feedforward networks*. Neural Networks, 4(2), 251–257.
- and this is true even with a single hidden layer!

expressivity of feedforward NNs

- Hornik's **universal approximation theorem** shows that feedforward NNs can approximate any (bounded) mathematical function
  - Hornik (1991). *Approximation capabilities of multilayer feedforward networks*. Neural Networks, 4(2), 251–257.
- and this is true even with a single hidden layer!
- however, this is mainly of theoretical interest
  - the theorem does not say how many hidden units we need
  - and it doesn't say how the network should be trained

"deep learning"

- why the "deep" in "deep learning"?
- although a single hidden layer is sufficient in theory, in practice it can be better to have several hidden layers



- previously, it was computationally hard to train models with many hidden layers
- but a number of recently developed algorithmic tricks (and again, better hardware) has made this more feasible

training feedforward neural networks

- training a NN consists of finding the weights in the layers
- so how do we find those weights?

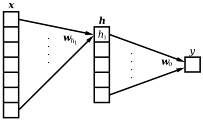
training feedforward neural networks

- training a NN consists of finding the weights in the layers
- so how do we find those weights?
- exactly as we did for the SVM and LR!
- state an **objective function** with a **loss**
  - log loss, hinge loss, etc
- and then tweak the weights to make that loss small
  - again, we can use (stochastic) **gradient descent** to minimize the loss

example

- let's use two layers with logistic units, and then the log loss

$$
\begin{aligned}
\boldsymbol{h} &= \sigma(\boldsymbol{W}_h \cdot \boldsymbol{x}) \\
y &= \sigma(\boldsymbol{W}_o \cdot \boldsymbol{h}) \\
\text{loss} &= -\log(y)
\end{aligned}
$$



- so the whole thing becomes

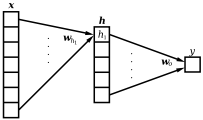
$$\text{loss} = -\log \sigma(\boldsymbol{W}_o \cdot \sigma(\boldsymbol{W}_h \cdot \boldsymbol{x}))$$

- now, to do gradient descent, we need to compute gradients w.r.t. the weights $\boldsymbol{W_h}$ and $\boldsymbol{W_o}$

example

- let's use two layers with logistic units, and then the log loss

$$
\begin{aligned}
\boldsymbol{h} &= \sigma(\boldsymbol{W}_h \cdot \boldsymbol{x}) \\
y &= \sigma(\boldsymbol{W}_o \cdot \boldsymbol{h}) \\
\text{loss} &= -\log(y)
\end{aligned}
$$



- so the whole thing becomes

$$\text{loss} = -\log \sigma(\boldsymbol{W}_o \cdot \sigma(\boldsymbol{W}_h \cdot \boldsymbol{x}))$$

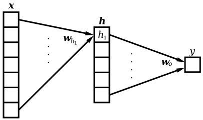
- now, to do gradient descent, we need to compute gradients w.r.t. the weights $\boldsymbol{W_h}$ and $\boldsymbol{W_o}$
- **ouch!** it looks completely unwieldy!

the chain rule of derivatives/gradients

- NNs consist of functions applied to the output of other functions
- the **chain rule** is a useful trick from calculus that can be used in such situations
- assume that we apply the function f to the output of g
- then the chain rule says how we can compute the gradient of the combination:

$$\text{gradient of } f(g(x)) = \text{gradient of } f(g) \cdot \text{gradient of } g(x)$$

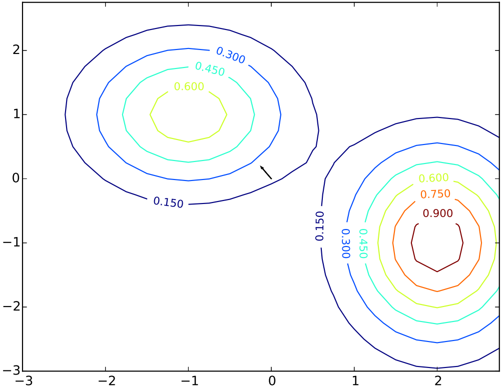
the general recipe: backpropagation



- using the chain rule, the gradients of the weights in each layer can be computed from the gradients of the layers after it
- this trick is called **backpropagation**
- it's not difficult, but involves a lot of book-keeping
- fortunately, there are computer programs that can do the algebra for us!
  - in NN software, we usually just declare the network and the loss, then the gradients are computed under the hood

optimizing NNs

- unlike the linear classifiers we studied previously, NNs have non-convex objective functions with a lot of local minima
- so the end result depends on initialization



training efficiency of NNs

- our previous classifiers took seconds or minutes to train
- NNs tend to take minutes, hours, days, weeks . . .
  - depending on the complexity of the network and the amount of training data
- NNs use a lot of linear algebra (matrix multiplications) so it can be useful to work to speed up the math
  - parallelize as much as possible
  - use optimized math libraries
  - use a GPU

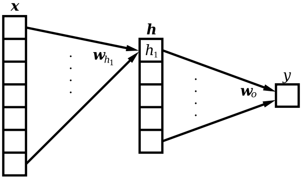
neural network software: Python

- scikit-learn has very limited support for NNs
- the main NN software in the Python world is **Theano**
  - developed by Yoshua Bengio's group in Montréal
  - `http://deeplearning.net/software/theano`
- Theano does a lot of useful math stuff, and integrates nicely with the GPU, but it can be a bit low-level
- so there are a few libraries that package Theano in a more user-friendly way, similar to scikit-learn
  - **pylearn2**: `http://deeplearning.net/software/pylearn2`
  - **Keras**: `https://github.com/fchollet/keras`

other neural network software

- **Caffe**: `http://caffe.berkeleyvision.org/`
- **Torch**: `http://torch.ch/`

coding example with Keras




```
keras_model = Sequential()

n_hidden = 3
keras_model.add(Dense(input_dim=X.shape[1],
                      output_dim=n_hidden))
keras_model.add(Activation("sigmoid"))

keras_model.add(Dense(input_dim=n_hidden,
                      output_dim=1))
keras_model.add(Activation("sigmoid"))

keras_model.compile(loss='binary_crossentropy',
                    optimizer='rmsprop')

keras_model.fit(X, Y)
```

representing words in NNs

- NN implementations tend to prefer dense vectors
- this can be a problem if we are using word-based features
- recall the way we code word features as sparse vectors:

$$\begin{aligned} \textit{tomato} &\rightarrow [0,0,1,0,0,\ldots,0,0,0] \\ \textit{carrot} &\rightarrow [0,0,0,0,0,\ldots,0,1,0] \end{aligned}$$

- the solution: represent words with low-dimensional vectors, in a way so that words with similar meaning have similar vectors

$$\begin{aligned} \textit{tomato} &\rightarrow [0.10,-0.20,0.45,1.2,-0.92,0.71,0.05] \\ \textit{carrot} &\rightarrow [0.08,-0.21,0.38,1.3,-0.91,0.82,0.09] \end{aligned}$$

- in the NN community, the word vectors are called **embeddings**

building the word representations

- the word vectors can be trained directly inside a NN, but often they are produced separately
  - a large corpus is needed to get good vectors
  - but the corpus doesn't have to be annotated
- many methods and software packages, here are just two examples:
  - **word2vec** is based on a method similar to LR
  - **gensim** has a Python-based reimplementation of word2vec
  - demo: `http://rare-technologies.com/word2vec-tutorial/`
- these methods are connected to the ideas of classical **distributional semantics**
- more about this in Yuri's seminar on the 16th

going beyond classification

- for "noisy" problems, NNs are excellent classifiers
  - recognizing a hand-written digit
  - recognizing a face in a photo
  - ...
- for problems that are more symbolic in nature, and if we have good features, NNs are usually not worth the effort
- but the recent enthusiasm about NNs and NLP isn't so much about classification...
- much recent research tends to focus on end-to-end tasks such as speech recognition and translation

NNs for sequences: recurrent NNs

- in a **recurrent** NN, the hidden units can be seen as a representation of a **state**
- in each step, the new state is computed from the input and the previous state
- they can be used for sequence tagging problems
- recurrent NN make excellent language models
- Mikolov et al. (2010): Recurrent neural network based language model, Interspeech.

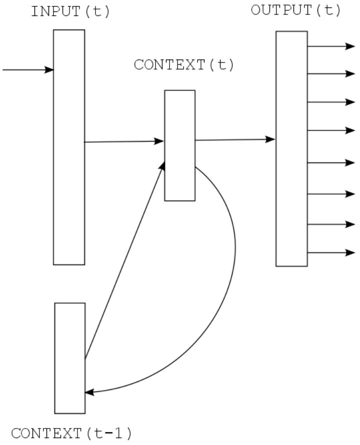


image by Mikolov et al.

translation with NNs: sequence-to-sequence learning

- recently, a team at Google proposed a NN model termed sequence-to-sequence learning, used in machine translation
  - either to rerank outputs generated by a standard SMT system
  - or to generate the output directly!

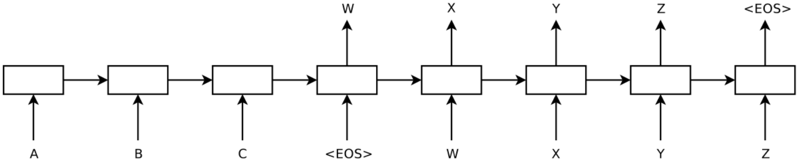


image by Sutskever et al.

- see Sutskever el al. (2014): *Sequence to sequence learning with neural networks*, NIPS.
- they used a model called **long short-term memory**, an extension of recurrent NNs

outlook

- there has been much creative NN/NLP research lately
  - and a number of the leading NN researchers consider NLP the most interesting unexplored research territory
- but so far, we haven't yet seen the dramatic improvements that have disrupted other fields
- the most wide-spread development so far is probably the use of vector representations as features
  - Turian et al. (2010): *Word Representations: A Simple and General Method for Semi-Supervised Learning*, ACL.
  - Toni will speak about this in his seminar on the 19th
- but what happens if "deep learning" will dominate? will it lead to a conentration of NLP research to the tech giants?