Neural Networks and Deep Learning: A Quick Overview

- Slides (./files/deep_learning by Zico.pdf) from Prof. Zico Kolter (http://www.zicokolter.com/) at CMU
- Lectures from Prof. Seungjin Choi (http://mlg.postech.ac.kr/) at POSTECH

Collected by Prof. Seungchul Lee
iSystems Design Lab
UNIST
http://isystems.unist.ac.kr/

Table of Contents

1. Neural Networks

1.1. Recall supervised learning setup

- Input features $x^{(i)} \in \mathbb{R}^n$
- Ouput $y^{(i)}$
- Model parameters $\theta \in \mathbb{R}^k$
- Hypothesis function $h_\theta : \mathbb{R}^n \to y$
- Loss function $\ell : y \times y \to \mathbb{R}_+$

- Machine learning optimization problem

$$\min_\theta \sum_{i=1}^m \ell \left(h_\theta \left(x^{(i)} \right), y^{(i)} \right)$$

(possibly plus some additional regularization)

We mainly considered the *linear* hypothesis class

$$h_\theta \left(x^{(i)} \right) = \theta^T \phi(x^{(i)})$$

for some set of *non-linear* feature $\phi : \mathbb{R}^n \to \mathbb{R}^k$

1.2. Challenges with linear models

- Linear models crucially depend on choosing "good" features
- Some "stardard" choices: polynomial features, radial basis functions, random features (srprisingly effective)

- But, many specialized domains required highly engineered special features

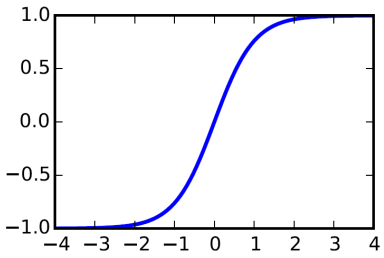
1.3. Neural networks

Neural networks are a simply a machine learning algorithm with a more complex hypothesis class, directly incorporating non-linearity (in the parameters)

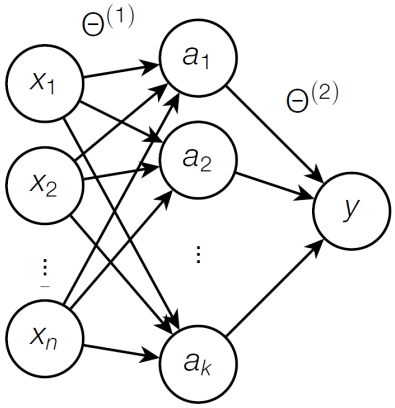
Example: neural network with one hidden layer

$$h_\theta(x) = \Theta^{(2)} f\left(\Theta^{(1)} x\right)$$

where $\Theta^{(1)} \in \mathbb{R}^{k \times n}$, $\Theta^{(2)} \in \mathbb{R}^{1 \times k}$ and f is some non-linear function applied elementwise to a vector (common choice is "tanh" function $\tanh(x) = \frac{1-e^{-2x}}{1+e^{-2x}}$)



Architectures are often shown graphically



Middle layer a is referred to as the hidden layer, there is nothing in the data that prescribes what values these should take, left up to the algorithm to decide

Viewed another way: neural networks are like linear classifiers where the features themselves are also learned

Pros

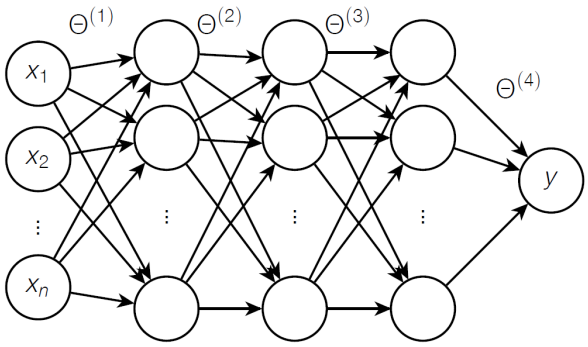
- No need to manually engineer good features, just let the machine learning algorithm handle this part
- It turns out that a 3-layer network is a universal function approximator, any non-linear function can be represented with a 3-layer network with a large enough hidden layer

Cons

- Minimizing loss on training data is no longer a convex optimization problem in parameters θ
- Still need to engineer a good architecture (more on this shortly)

1.4. Deep learning

"Deep" neural networks typically refer to networks with multiple hidden layers



Note: original term "deep learning" referred to any machine learning architecture with multiple layers, including several probabilistic models, etc, but most work these days focuses on neural networks

Motivation from neurobiology: brain appears to use multiple levels of interconnected neurons to process information (but careful,neurons in brain are not just non-linear functions)

In practive: works better for many domains

2. Training Neural Networks

2.1. Optimizing neural network parameters

How do we optimize the parameters for the machine learning loss minimization problem with a neural network

$$\min_{\theta} \sum_{i=1}^{m} \ell\left(h_\theta\left(x^{(i)}\right), y^{(i)}\right)$$

now is this problem non-convex?

Just do exactly what we did before: initialize with random weights and run stochastic gradient descent

Now have the possibility of local optima, and function can be harder to optimize, but we will not worry about all that because the resulting model still often perform better than linear models

Stochastic gradient descent, repeat:

- Select some example i

$$\theta := \theta - \alpha \nabla_\theta \left(h_\theta\left(x^{(i)}\right), y^{(i)}\right)$$

So how do we compute the gradient with respect to all the parameters in a neural network (*i.e.*, all weights $\Theta^{(1)}, \Theta^{(2)}, \cdots$) ?

2.2. Backpropagation

Backpropagation is a method for computing all the necessary gradients using one "forward pass" (just computing all the values at layers), and one "backward pass" (computing gradients backwards in the network)

The equations sometimes look complex, it is just an application of the chain rule of calculus

First, some notation that will make things a bit easier:

- Activations: $a^{(i)}$ valuesat hidden layer i (with convention that $a^{(1)} = x$)
- Linear activation: $z^{(i)} = \Theta^{(i)} a^{(i)}$, so $a^{(i+1)} = f(z^{(i)})$

Let's treat everything as scalars for now, and consider a network with two hidden layers:

$$\begin{aligned}
\frac{\partial}{\partial \Theta^{(1)}} \ell\left(h_\theta(x), y\right) &= \frac{\partial}{\partial \Theta^{(1)}} \ell\left(\Theta^{(3)} f(\Theta^{(2)} f(\Theta^{(1)} x)), y\right) \\
&= \ell'\left(z^{(3)}, y\right) \Theta^{(3)} \frac{\partial}{\partial \Theta^{(1)}} f(\Theta^{(2)} f(\Theta^{(1)} x)) \\
&= \ell'\left(z^{(3)}, y\right) \Theta^{(3)} f'\left(z^{(2)}\right) \Theta^{(2)} \frac{\partial}{\partial \Theta^{(1)}} f(\Theta^{(1)} x) \\
&= \ell'\left(z^{(3)}, y\right) \Theta^{(3)} f'\left(z^{(2)}\right) \Theta^{(2)} f'\left(z^{(1)}\right) a^{(1)}
\end{aligned}$$

By the same procedure

$$\frac{\partial}{\partial \Theta^{(2)}} \ell\left(h_\theta(x), y\right) = \ell'\left(z^{(3)}, y\right) \Theta^{(3)} f'\left(z^{(2)}\right) a^{(2)}$$

If you want to compute gradients with respect to all the parameters $\Theta^{(1)}, \cdots, \Theta^{(L)}$, we can "reuse" parts of this computation

Let

$$g^{(L)} = \ell' \left(z^{(L)} \right) \Theta^{(L)}$$
$$g^{(i)} = g^{(i+1)} f' \left(z^{(i)} \right) \Theta^{(i)}$$

then

$$\frac{\partial}{\partial \Theta^{(i)}} \ell \left(h_\theta(x), y \right) = g^{(i+1)} f' \left(z^{(i)} \right) a^{(i)}$$

It takes just slightly more advanced calculus, but it turns out the general matrix/vector case is exactly the same, just being carful with the ordering/size of matrix multiplication

The full backpropagation algorithm:

$$g^{(L)} = {\Theta^{(L)}}^T \ell' \left(z^{(L)} \right)$$
$$g^{(i)} = {\Theta^{(i)}}^T g^{(i+1)} f' \left(z^{(i)} \right)$$
$$\nabla_{\Theta^{(i)}} \ell \left(h_\theta(x), y \right) = \left(g^{(i+1)} \cdot f' \left(z^{(i)} \right) \right) {a^{(i)}}^T$$

where $\cdot$ denotes elementwise multiplication

Gradients can get somewhat tedious to derive by hand, especially for the more complex models

Fortunately, a lot of this work has already been done for us

Tools like Theano, Torch, Caffe, TensorFlow all let us specify the network structure and then automatically compute all gradients (and use GPUs to do so)

In [1]:
```
%%html
<iframe src="https://www.youtube.com/embed/uXt8qF2Zzfo"
 width="560" height="315" frameborder="0" allowfullscreen></iframe>
```

3. Deep Learning

3.1. What is changed since the 80s?

All these algorithms (and most of the extensions in later slides), were developed in the 80s or 90s

So why are these just becoming more popular in the last few years?

- more data
- faster computers
- (some) better optimization techniques

Unsupervised pre-training (Hinton et al., 2006): "Pre-train" the network have the hidden layers recreate their input, one layer at a time, in an unsupervised fashion

- This paper was partly responsible for re-igniting the interest in deep neural networks, but the general feeling now is that it does not help much

Dropout (Hinton et al., 2012): During training and computation of gradients, randomly set about half the hidden units to zero

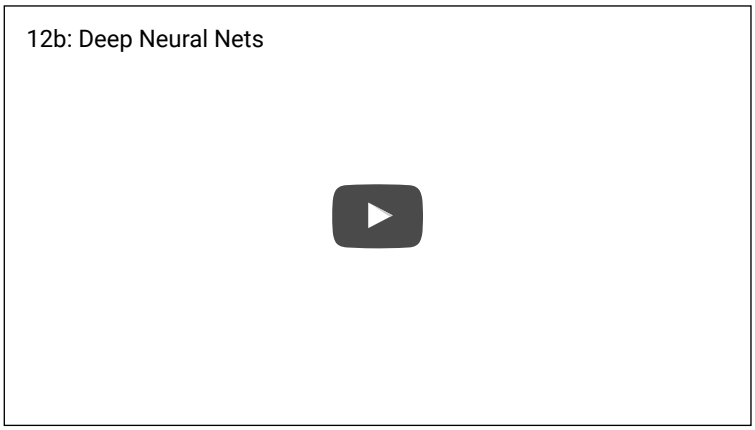
- Acts like regularization, prevents the parameters for overfitting to particular exampels

Different non-linear fuctions (Nair and Hinton, 2010): Use non-linearity $f(x) = \max\{0, x\}$ instead of $f(x) = \tanh(x)$

3.2. Again, why successful?

- Pre-training: Restricted Boltzmann machine (RBM), Autoencoder, nonnegative matrix factorization (NMF)
- Training: Dropout
- Rectified linear units: No vanishing gradient, sparse activation

In [2]:
```
%%html
<iframe src="https://www.youtube.com/embed/VrMHA3yX_QI"
width="560" height="315" frameborder="0" allowfullscreen></iframe>
```



4. Advanced Models and Architectures in Deep Learning

- Convolutional neural networks
- Recurrent neural networks
- Deep reinforcement learning

4.1. Convolutional neural networks

One of the biggest successes for neural networks has come in computer vision, using convolution neural networks.

In traditional neural networks, images are treated as unstructured vectors $x \in \mathbb{R}^{W \cdot H}$, and we learn arbitrary transformation *i.e.*, $f(\Theta x)$

But this does not seem like a good model, slightly shifting/scaling image winds up with a very different input vector, so we need to learn all these invariances in our parameters.

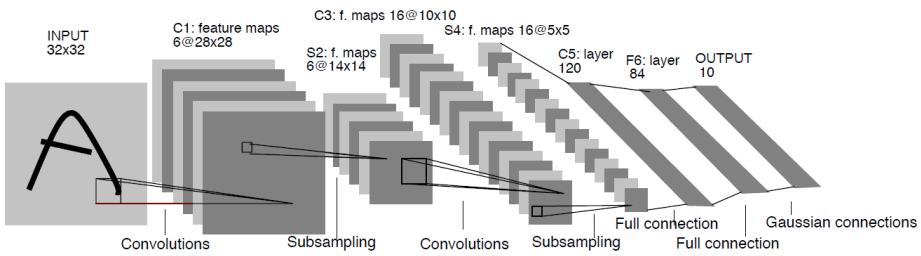
Basic idea of convolutional neural networks: parameters are elements of a (set of) convolutional filters applied to the image

$$a^{(i+1)} = f\left(a^{(i)} * \Theta^{(i)}\right)$$

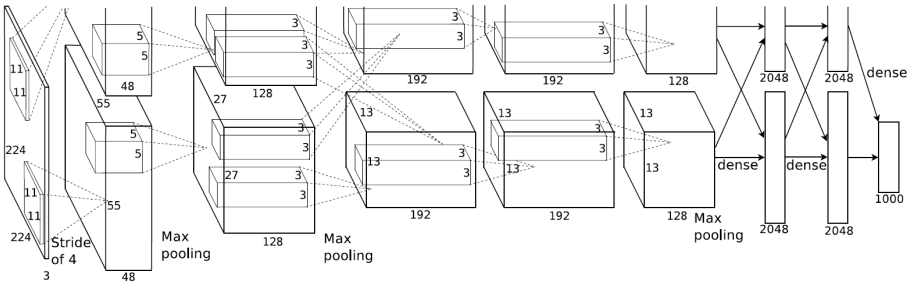
The function f also does downsampling "max-pooling" to produce lower dimensional images and translation invariance at later layers.

ImageNet Large Scale Visual Recognition Challenge (ILSVRC)

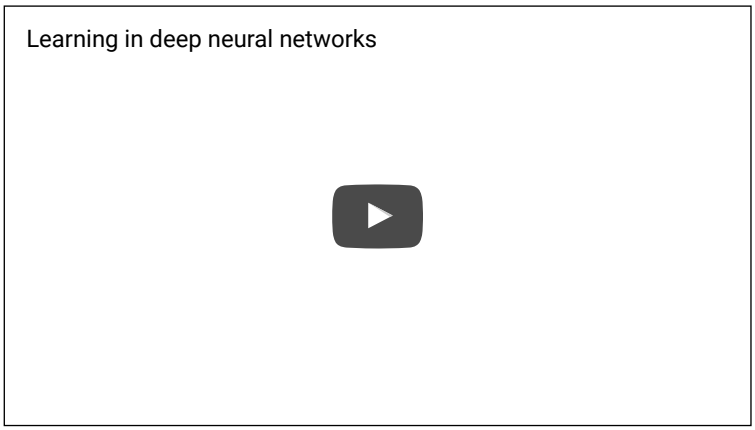
- Lenet-5 (LeCun et al., 1998) architecture, 1% error on MNIST classification (compare to 10% for linear classifier)



- "AlexNet" (Krizhevsky et al., 2012), work ImageNet 2012 competition with a Top-5 error rate of 15.3% (next best system with highly engineered features based upon SIFT got 26.1% error)



In [3]:
```
%%html
<iframe src="https://www.youtube.com/embed/bL1Zymz1b7g"
width="560" height="315" frameborder="0" allowfullscreen></iframe>
```



Slides available (./files/deep_learning_tutorial_2015.pdf) (click to open in a new page)

- By Phillip Isola
- Tutorial Series in Computational topics for BCS

4.2. Recurrent neural networks

Framework for dealing with *sequence* data: hidden units for elements in sequence are fed into hidden units for subsequent elements

- difference equation
- time series analysis
- hidden Markov model (HMM)

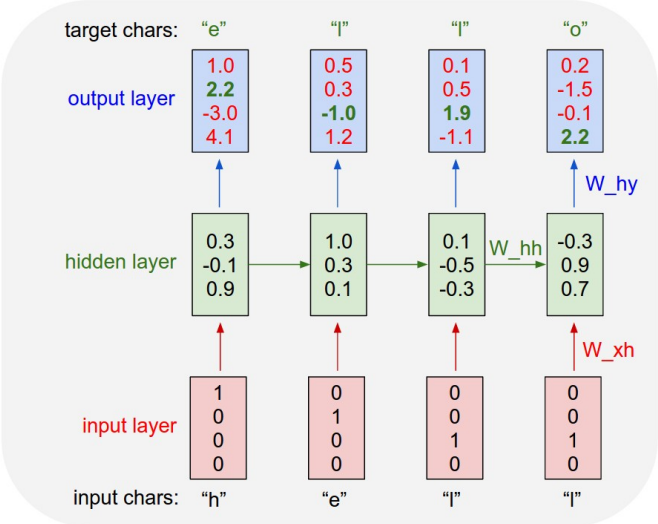


Figure from Karpathy, 2015 (http://karpathy.github.io/2015/05/21/rnn-effectiveness/)

4.3. Deep reinforcement learning

Instead of maintaining a separate Q-value for each state/action pair, we can use a deep neural network (or any other class of functions) to represent the Q function

Q-Learning update rule becomes

$$\theta := \theta - \alpha \left(R + \gamma \max_{a'} Q(s', a'; \theta) - Q(s, a; \theta) \right) \nabla_\theta Q(s, a; \theta)$$

where θ are the parameters (*i.e.*, network weights) that specify our reprentation of the Q function

Google DeepMind paper shows deep learning to play Atari video games: Pong, Breakout, Space Invaders, Seaquest, Beam Rider (Mihn et al., 2013)

- Markov decision process (MDP)
- Bellman equations
- Reinforcement learning
- Q-learning

In [4]:
```
%%javascript
$.getScript('https://kmahelona.github.io/ipython_notebook_goodies/ipython_notebook_toc.js')
```