Neural Networks and Deep Learning

Joong-Ho Won

Seoul National University

August 25, 2016

Outline

- Introduction
- Convolutional neural networks (CNN)
- Recurrent neural networks (RNN)
- A statistical view of deep learning
- Open source tools

Outline

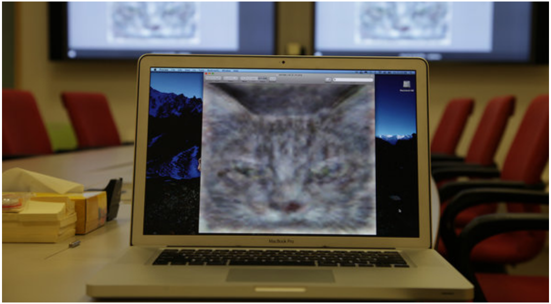
- Introduction
    - Recent highlights on deep learning
    - Deep feedforward neural networks
    - A brief history
- Convolutional neural networks (CNN)
- Recurrent neural networks (RNN)
- A statistical view of deep learning
- Open source tools

Outline

- Introduction
  - Recent highlights on deep learning
  - Deep feedforward neural networks
  - A brief history
- Convolutional neural networks (CNN)
- Recurrent neural networks (RNN)
- A statistical view of deep learning
- Open source tools

Deep learning hype on media

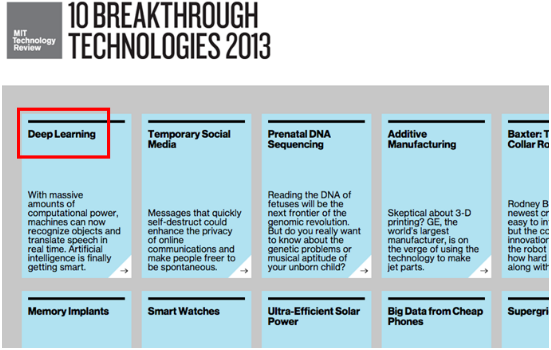
- New York Times (2012)
  - Google's artificial brain identifies a cat from YouTube videos without any labels [Le, Building high-level features using large scale unsupervised learning, 2012]



An image of a cat that a neural network taught itself to recognize
©: Jim Wilson/The New York Times

Deep learning hype on media (cont'd)

- MIT Technology Review (2013)
  - One of top 10 most promising breakthrough techs



MIT Technology Review

10 BREAKTHROUGH TECHNOLOGIES 2013

Deep Learning

With massive amounts of computational power, machines can now recognize objects and translate speech in real time. Artificial intelligence is finally getting smart. →

Temporary Social Media

Messages that quickly self-destruct could enhance the privacy of online communications and make people freer to be spontaneous. →

Prenatal DNA Sequencing

Reading the DNA of fetuses will be the next frontier of the genomic revolution. But do you really want to know about the genetic problems or musical aptitude of your unborn child? →

Additive Manufacturing

Skeptical about 3-D printing? GE, the world's largest manufacturer, is on the verge of using the technology to make jet parts. →

Baxter: T Collar Ro

Rodney B newest cr easy to in but the co innovatior the robot how hard along with

Memory Implants

Smart Watches

Ultra-Efficient Solar Power

Big Data from Cheap Phones

Supergri

10 Breakthrough Technologies 2013 - MIT Technology Review

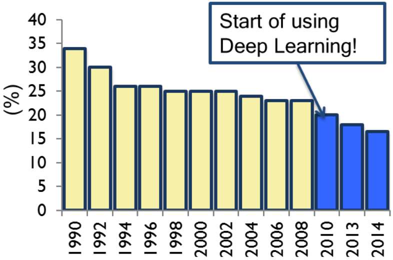
©: http://www.computescotland.com/deep-learning-dc-power-grids-bc-robots-6121.php

Recent impacts

- Real industry impacts!

Speech Recognition

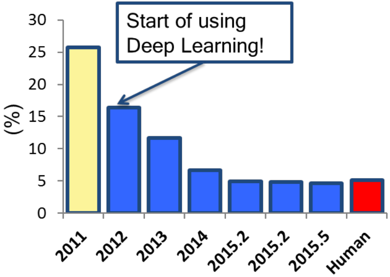
- TIMIT Phone Error Rate (PER)



Deep Learning has been used for Apple Siri, Google Voice Search, Samsung S-Voice etc.

Image Recognition

- Top-5 Error on ImageNet



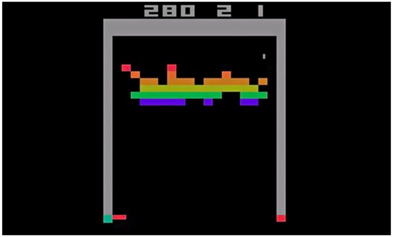
Major companies (Google, Yahoo, Facebook, Microsoft, etc.) use the Deep Learning-based image recognition systems

Recent impacts (cont'd)

- Getting scary . . .

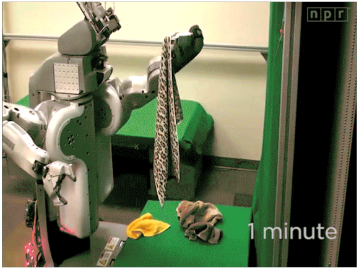
Playing video games

- Playing Atari video games



[Mnih et.al., Human-level control through deep reinforcement learning, 2015]

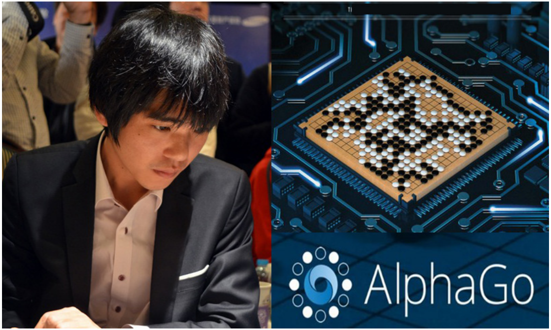
Teaching robots



[Levine et. al., End-to-end training of deep visuomotor policies, 2015]

Recent impacts (cont'd)

- Google DeepMind Challenge Match (2016)



Sedol Lee vs. AlphaGo. 1 : 4

©: http://www.koreaittimes.com/story/58635/lessons-lee-sedol-vs-alphago-match

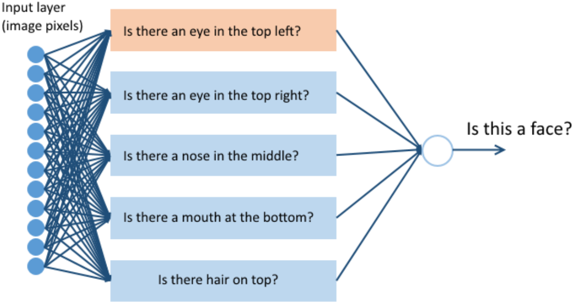
Outline

- Introduction
  - Recent highlights on deep learning
  - Deep feedforward neural networks
  - A brief history
- Convolutional neural networks (CNN)
- Recurrent neural networks (RNN)
- A statistical view of deep learning
- Open source tools

What is deep learning?

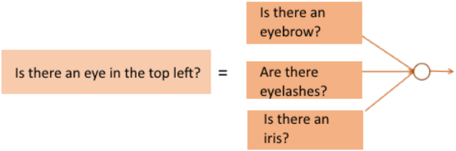
- Suppose we want to determine whether each image shows a human face or not

- Then our questions are, for example:


Input layer
(image pixels)
Is there an eye in the top left?
Is there an eye in the top right?
Is there a nose in the middle?
Is there a mouth at the bottom?
Is there hair on top?
Is this a face?

What is deep learning? (cont'd)

- Each question can be broke down into sub-questions:



Is there an eye in the top left? =

Is there an eyebrow?

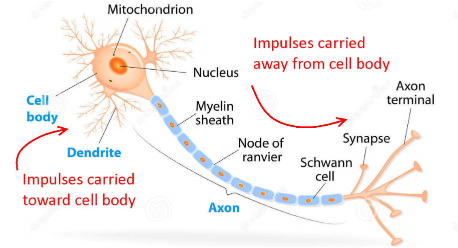
Are there eyelashes?

Is there an iris?

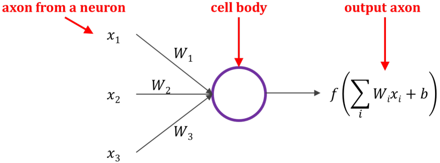
- Similar to a human's perception process
    - Abstractions from the low level representation to the high level representation
    - The higher layer builds new abstarctions on the top of the previous layer
- Main idea: learn representations (= features) of data using "multiple processing layers with non-linear transformations" a.k.a., "artificial neural networks (ANN)"
- Inspired by biological neural networks in a human brain

Artificial neuron

- Motivated from the way that biological impulses transfer between cells



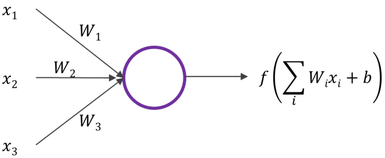
Biological neuron



Artificial neuron

Artificial neuron (cont'd)

- A computational unit (also known as "perceptron")
  - Affine transformation + non-linearity
  - Weights (+ biases) are the parameters to learn



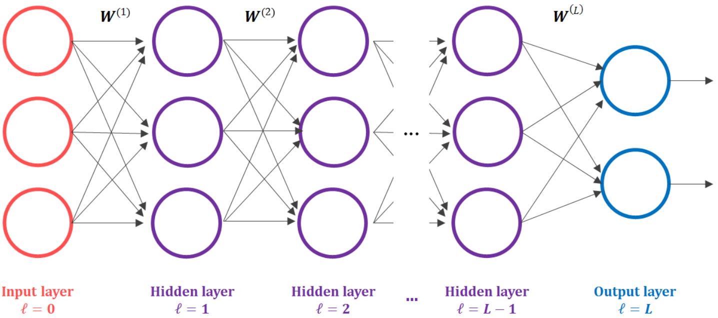
Artificial neuron with three inputs

$$\mathbf{y} = f(\mathbf{W}\mathbf{x} + \mathbf{b}) = f(\sum_i W_i x_i + b)$$

where f is called an activation function with non-linearity

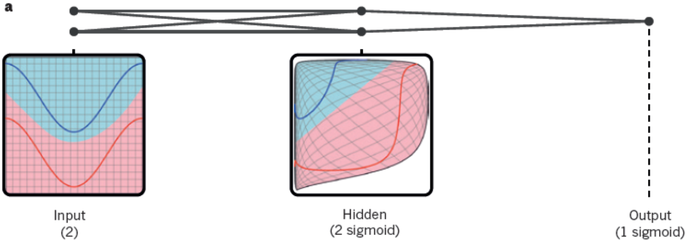
Deep feedforward neural networks

- Architecture
  - An input layer, hidden layer(s), and an output layer consisting of neurons
  - Each neuron's ouput can be an input of another neuron at the next layer
  - No connections between neurons at the same layer, no feedback connections $\longrightarrow$ "feedforward"
  - Hidden layers can go "deep", i.e., multiple layers of multiple neurons



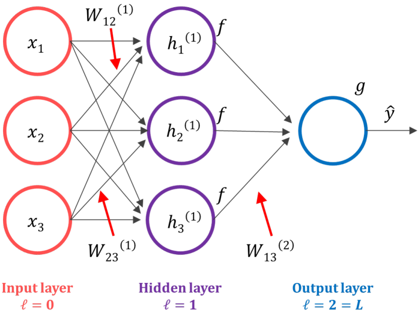
Why "deep", i.e., multi-layer neural networks?

- Shallow NNs may require a huge number of computational units to model highly varying functions
  - e.g., checker board function
- Deep NNs can nonlinearly distort the input space
  - In results, a simple classifier can easily separate classes
  - Deep NNs can learn right transformations for a given learning task



©: [LeCun, Bengio, and Hinton, Deep learning, 2015]

Architecture



$$
\begin{aligned}
\mathbf{h}^{(0)} &= \mathbf{x} \\
\mathbf{a}^{(\ell)} &= \mathbf{W}^{(\ell)}\mathbf{h}^{(\ell-1)} + \mathbf{b}^{(\ell)} \\
\mathbf{h}^{(\ell)} &= f(\mathbf{a}^{(\ell)}) \text{ for } \ell = 1, 2, \dots, L-1 \\
\widehat{\mathbf{y}} = \mathbf{h}^{(L)} &= g(\mathbf{a}^{(L)})
\end{aligned}
$$

L: network depth
$\mathbf{x}$: input vector
$\widehat{\mathbf{y}}$: output vector
$\mathbf{W}^{(\ell)}$: weight matrix at layer ℓ
$\mathbf{b}^{(\ell)}$: bias vector at layer ℓ
f: activation function for hidden units
g: activation function for output units

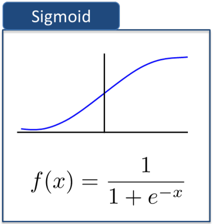
Architecture (cont'd)

- For example, if the network consists of three inputs ($= x_1, x_2, x_3$), one output ($= \hat{y}$), and one hidden layer with three neurons:

$$\begin{aligned}
h_1^{(1)} &= f(a_1^{(1)}) = f(W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{13}^{(1)}x_3 + b_1^{(1)}) \\
h_2^{(1)} &= f(a_2^{(1)}) = f(W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{23}^{(1)}x_3 + b_2^{(1)}) \\
h_3^{(1)} &= f(a_3^{(1)}) = f(W_{31}^{(1)}x_1 + W_{32}^{(1)}x_2 + W_{33}^{(1)}x_3 + b_3^{(1)}) \\
\hat{y} &= g(a_1^{(2)}) = g(W_{11}^{(2)}h_1^{(1)} + W_{12}^{(2)}h_2^{(1)} + W_{13}^{(2)}h_3^{(1)} + b_1^{(2)})
\end{aligned}$$

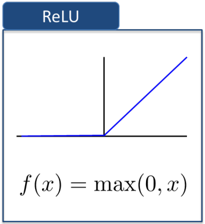
Activation functions

- Nonlinear distortion of the input
- Common choices
  - For hidden units
    - Sigmoid, tanh, ReLU (rectified linear unit), maxout
    - ReLU has become a de facto standard recently
  - For output units for classification: multilogit transform $g_j(\mathbf{t}) = \frac{exp(t_j)}{\sum_j exp(t_j)}$
- Typically applied to vectors in an element-wise fashion
  - e.g., $f : \mathbb{R}^3 \mapsto \mathbb{R}^3$, $f([x_1, x_2, x_3]) = [f(x_1), f(x_2), f(x_3)]$



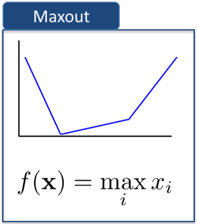
Sigmoid

$$f(x) = \frac{1}{1 + e^{-x}}$$



ReLU

$$f(x) = \max(0, x)$$



Maxout

$$f(\mathbf{x}) = \max_i x_i$$

Training deep feedforward neural networks

- Objective: given $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^N$, minimize the cost function

$$J(\boldsymbol{\theta}) = \sum_{n=1}^{N} J_i(\boldsymbol{\theta}) = \sum_{n=1}^{N} \mathcal{L}(\mathbf{y}_i, \hat{\mathbf{y}}_i)$$

  where $\boldsymbol{\theta} = (\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \mathbf{W}^{(2)}, \mathbf{b}^{(2)}, \dots, \mathbf{W}^{(L)}, \mathbf{b}^{(L)})$

- Weights (+ biases) are learned using the gradient descent method:

$$\boldsymbol{\theta}^{(t+1)} := \boldsymbol{\theta}^{(t)} - \eta^{(t)} \frac{\partial J}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}^{(t)})$$

$\longrightarrow$

$$W_{km}^{(\ell)} \longleftarrow W_{km}^{(\ell)} - \eta \sum_{i=1}^{N} \frac{\partial J_i}{\partial W_{km}^{(\ell)}}$$

$$b_k^{(\ell)} \longleftarrow b_k^{(\ell)} - \eta \sum_{i=1}^{N} \frac{\partial J_i}{\partial b_k^{(\ell)}}$$

Back-propagation algorithm

- Back-propagation algorithm ("backprop") enables us to compute the gradients very efficiently
- Recall $J(\boldsymbol{\theta}) = \sum_{n=1}^{N} J_i(\boldsymbol{\theta}) = \sum_{n=1}^{N} \mathcal{L}(\mathbf{y}_i, \hat{\mathbf{y}}_i)$

$$
\begin{aligned}
\hat{y}_{ik} &= h_{ik}^{(L)} = g_k\left(\mathbf{a}_i^{(L)}\right) && k = 1, \ldots, K \\
\mathbf{a}_i^{(L)} &= \mathbf{W}^{(L)}\mathbf{h}_i^{(L-1)} + \mathbf{b}^{(L)} && \in \mathbb{R}^K \\
h_{im_1}^{(L-1)} &= f\left(a_{im_1}^{(L-1)}\right) && m_1 = 1, \ldots, M_1 \\
&\vdots \\
h_{im_L}^{(0)} &= x_{im_L} && m_L = 1, \ldots, M_L = p
\end{aligned}
$$

Back-propagation algorithm (cont'd)

$$
\begin{aligned}
\frac{\partial J_i}{\partial W_{km_1}^{(L)}} &= \underbrace{\left(\frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}_i}\right)^\top}_{1\times K} \underbrace{\left(\frac{\partial \hat{\mathbf{y}}_i}{\partial \mathbf{a}_i^{(L)}}\right)}_{K\times K} \underbrace{\left(\frac{\partial \mathbf{a}_i^{(L)}}{\partial W_{km_1}^{(L)}}\right)}_{K\times 1} \\
&= \sum_{k'=1}^{K} \left[\left(\frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}_i}\right)^\top \left(\frac{\partial \hat{\mathbf{y}}_i}{\partial \mathbf{a}_i^{(L)}}\right)\right]_{k'} \cdot \left(h_{im_1}^{(L-1)} \delta_{k'k}\right) \\
&= \left[\left(\frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}_i}\right)^\top \left(\frac{\partial \hat{\mathbf{y}}_i}{\partial \mathbf{a}_i^{(L)}}\right)\right]_{k} \cdot h_{im_1}^{(L-1)} \\
&= s_{ik}^{(L)} \cdot h_{im_1}^{(L-1)}
\end{aligned}
$$

Back-propagation algorithm (cont'd)

$$
\begin{aligned}
\frac{\partial J_i}{\partial W^{(L-1)}_{m_1 m_2}} &= \underbrace{\left(\frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}_i}\right)^{\top}}_{1\times K} \underbrace{\left(\frac{\partial \hat{\mathbf{y}}_i}{\partial \mathbf{a}^{(L)}_i}\right)}_{K\times K} \underbrace{\left(\frac{\partial \mathbf{a}^{(L)}_i}{\partial \mathbf{h}^{(L-1)}_i}\right)}_{K\times M_1} \underbrace{\left(\frac{\partial \mathbf{h}^{(L-1)}_i}{\partial \mathbf{a}^{(L-1)}_i}\right)}_{M_1\times M_2} \underbrace{\left(\frac{\partial \mathbf{a}^{(L-1)}_i}{\partial W^{(L-1)}_{m_1 m_2}}\right)}_{M_1\times 1} \\
&= \sum_{m_1'=1}^{M_1} \Big[\underbrace{\mathbf{s}^{(L)}_i}_{1\times K} \cdot \underbrace{\mathbf{W}^{(L)}}_{K\times M_1} \cdot \underbrace{f'(a^{(L-1)}_{im_1'})\mathbf{I}_{M_1}}_{M_1\times M_1} \Big]_{m_1'} \cdot h^{(L-2)}_{im_2} \delta_{m_1' m_1} \\
&= \left(f'(a^{(L-1)}_{im_1}) \sum_{k=1}^{K} s^{(L)}_{ik} W^{(L)}_{km_1} \right) \cdot h^{(L-2)}_{im_2} \\
&= s^{(L-1)}_{im_1} \cdot h^{(L-2)}_{im_2}
\end{aligned}
$$

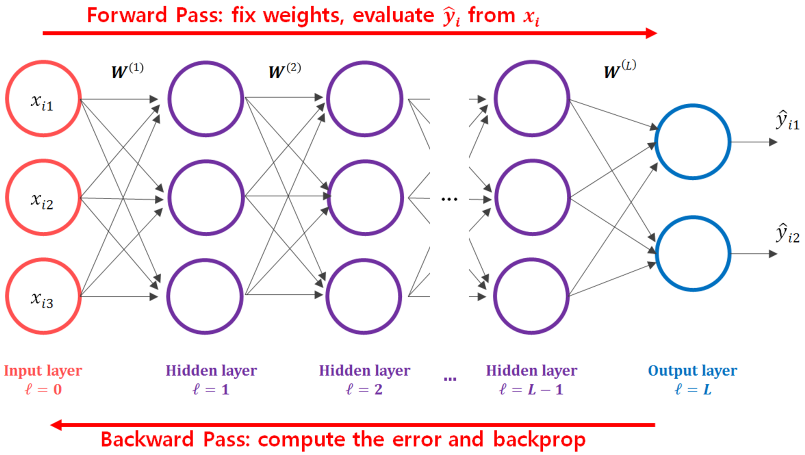
Back-propagation algorithm (cont'd)

- In other words,

$$
\begin{aligned}
\frac{\partial J_i}{\partial W_{km_1}^{(L)}} &= s_{ik}^{(L)} \cdot h_{im_1}^{(L-1)}, & s_{ik}^{(L)} &= \left[\left(\frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}_i}\right)^{\top}\left(\frac{\partial \hat{\mathbf{y}}_i}{\partial \mathbf{a}_i^{(L)}}\right)\right]_k \quad \text{"error"} \\
\frac{\partial J_i}{\partial W_{m_1 m_2}^{(L-1)}} &= s_{im_1}^{(L-1)} \cdot h_{im_2}^{(L-2)}, & s_{im_1}^{(L-1)} &= f'(a_{im_1}^{(L-1)}) \sum_{k=1}^{K} s_{ik}^{(L)} W_{km_1}^{(L)} \\
\frac{\partial J_i}{\partial W_{m_2 m_3}^{(L-2)}} &= s_{im_2}^{(L-2)} \cdot h_{im_3}^{(L-3)}, & s_{im_2}^{(L-2)} &= f'(a_{im_2}^{(L-2)}) \sum_{m_1=1}^{M_1} s_{im_1}^{(L-1)} W_{m_1 m_2}^{(L-1)} \\
&\vdots & &\vdots
\end{aligned}
$$

Back-propagation algorithm (cont'd)

- This suggests a two-pass algorithm



Forward Pass: fix weights, evaluate $\hat{y}_i$ from x_i

Input layer $\ell = 0$

Hidden layer $\ell = 1$

Hidden layer $\ell = 2$

...

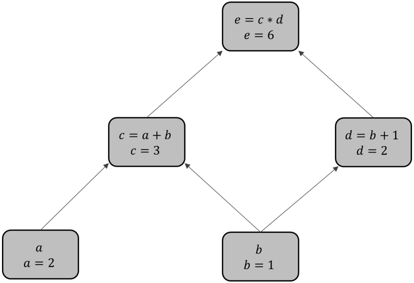
Hidden layer $\ell = L - 1$

Output layer $\ell = L$

Backward Pass: compute the error and backprop

Back-propagation algorithm: computation

- Backprop is merely an exercise of the chain rule for gradient descent
- But quickly becomes very complicated with complex neural network architectures (e.g., for CNNs, $\mathbf{W}^{(\ell)}$ is a tensor)
- Computational graph
  - Re-organize the mathematical expression as operations and nodes
  - Evaluate the expression by computing up subexpressions through the graph



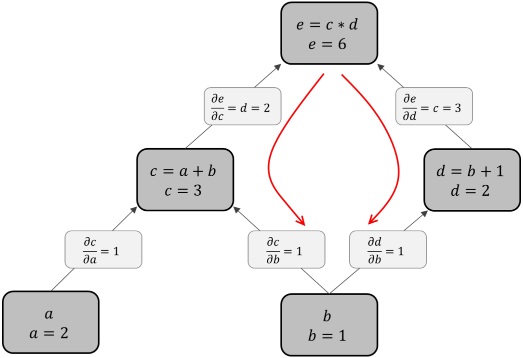
e.g., $e = (a + b) * (b + 1) \Rightarrow$

$$e = c * d$$
$$c = a + b$$
$$d = b + 1$$

⟶ can evaluate each node when a and b are given (e.g., $a = 2, b = 1$)

Computing the gradient between two nodes

- Backward differentiation with the chain rule
  - Sum over all possible paths from one node to the other, multiplying the derivatives on each edge of the path together
- e.g., $\frac{\partial e}{\partial b}$?
  - b affects e through c and d $\longrightarrow \frac{\partial e}{\partial b} = \frac{\partial e}{\partial c}\frac{\partial c}{\partial b} + \frac{\partial e}{\partial d}\frac{\partial d}{\partial b} = 2 * 1 + 3 * 1$

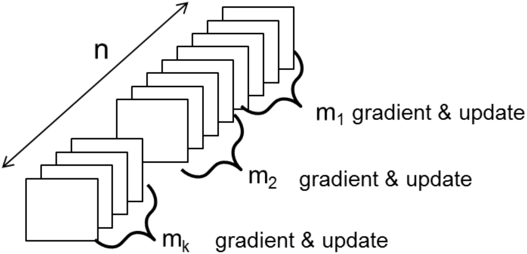

$e = c * d$
$e = 6$
$\frac{\partial e}{\partial c} = d = 2$
$\frac{\partial e}{\partial d} = c = 3$
$c = a + b$
$c = 3$
$d = b + 1$
$d = 2$
$\frac{\partial c}{\partial a} = 1$
$\frac{\partial c}{\partial b} = 1$
$\frac{\partial d}{\partial b} = 1$
a
$a = 2$
b
$b = 1$

Advantages of backward differentiation

- Locality
  - At each node, we can compute the local gradient of its inputs w.r.t. its output value without being aware of any of the details of the full networks $\longrightarrow$ can be implemented efficiently on a parallel machine
- Online learning capability
  - $\boldsymbol{\theta}$ can be updated in a sample-by-sample fashion (SGD)
  - Training epoch - one sweep through the entire training set

The *mini-batch* SGD

- Stochastic gradient descent (SGD)
  - Instead of the batch data, work with *mini-batch* of *m* examples
  - SGD is an unbiased estimate of GD when we average the gradient on mini-batches drawn i.i.d from the data generating distribution



The *mini-batch* SGD (cont'd)

```
Algorithm 1 SGD update at training iteration $k$
Require: Learning rate $\epsilon_k$
Require: Initial parameter $\boldsymbol{\theta}$
    while stopping criterion not met do
        Sample a mini-batch of $m$ exmples from the training set $\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}\}$ with corresponding targets $\mathbf{y}^{(i)}$
        Compute gradient estimate $\hat{\mathbf{g}} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i \mathcal{L}(\mathbf{y}^{(i)}, f(\mathbf{x}^{(i)}; \boldsymbol{\theta}))$
        Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon_k \hat{\mathbf{g}}$
    end while
```

The *mini-batch* SGD (cont'd)

- SGD gradient estimator introduces a source of noise (the random sampling of m training examples) that does not vanish even when we arrive at a mininum
  - The learning rate ϵ_k is crucial for SGD to work well
  - A sufficient condition to guarantee convergence:

$$\sum_k^{\infty} \epsilon_k = \infty \text{ and } \sum_k^{\infty} {\epsilon_k}^2 < \infty$$

  - In practice, it is common to decay ϵ_k linearly until iteration τ:

$$\epsilon_k \longleftarrow (1-\alpha)\epsilon_0 + \alpha\epsilon_k$$

with $\alpha = k/\tau$. After interation τ, it is common to leave ϵ constant.

Outline

- Introduction
  - Recent highlights on deep learning
  - Deep feedforward neural networks
  - A brief history
- Convolutional neural networks (CNN)
- Recurrent neural networks (RNN)
- A statistical view of deep learning
- Open source tools

A brief history of neural networks

- Early age (1940s – 1960s)
  - McCulloch and Pitts (1943) – biologically inspired, not data-adaptive
  - Perceptron (Rosenblatt, 1958) – learns linearly separable distributions
  - Widrow and Hoff (1960) – LMS (SGD) algorithm (linear logistic regression)
  - The "XOR problem" (Minsky and Papert, 1969)

——— Criticisms on biologically inspired learning ———
(dormant period of NNs)

A brief history of neural networks (cont'd)

- Second wave (mid 1980s – mid 1990s)
  - Backprop (Rumelhart et al., 1986)
  - Universal approximation theorems (Cybenko, 1989; Hornik et al., 1989; Leshno et al., 1993)
  - Distributed representation (Hinton et al., 1986)
  - Convolutional neural networks (LeCun et al., 1990)
  - Recurrent neural networks (Bengio et al., 1994)
  - LSTM (Hochreiter and Schmidhuber, 1997)

——— Vanshing gradient problem ———

——— Difficulties in interpretation ———

——— Other methods (SVM, kernel machines) surpass NNs ———

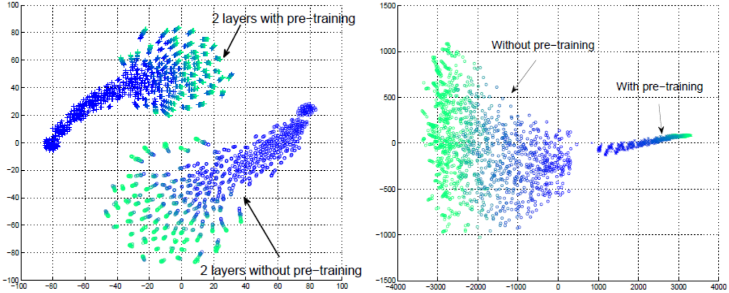
(Dark age of NNs)

A brief history of neural networks (cont'd)

- Third wave (2006 –)
    - Deep belief networks (Hinton and Salakhutdinov, 2006)
    - Unsupervised, layer-wise pretraining (Hinton and Salakhutdinov, 2006; Bengio et al., 2007)
    - Google Youtube ("cat") (Le, 2012)

Renaissance - reason 1: unsupervised pre-training

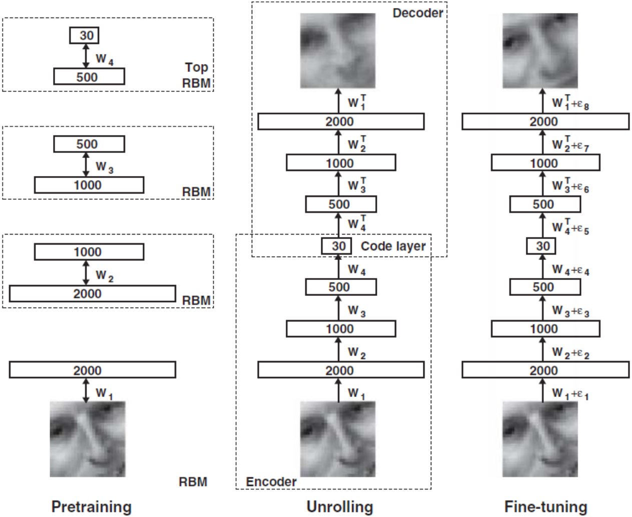
- With/Without pre-training



2D visualizations with tSNE (left) and ISOMAP (right) of the functions represented by 50 networks with and 50 networks without pretraining, as supervised training proceeds over the MNIST dataset. Color from dark blue to cyan indicates a progression in training iterations.

©: [Erhan et al., Why does unsupervised pre-training help deep learning? (2010)]

Deep belief networks (DBNs): architecture

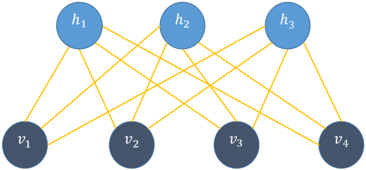


©: [Hinton and Salakhutdinov, Reducing dimensionality of data with neural networks, 2006]

Deep belief networks (DBNs): architecture (cont'd)

- A generative graphical model
    - Specify the value of some of the neurons and then "run the network backward", generating values for the input activations
- Unsupervised and semi-supervised learning
    - Find the hidden structure in unlabeled data
- Main idea (two phases)
    - Pre-training: unsupervised learning *layer-by-layer*
        - Pre-training consists of learning a stack of "restricted Boltzmann machines (RBMs)", each having only one layer of feature detectors
        - The learned feature activations of one RBM are used as input data for training the next RBM in the stack
    - Fine-tuning: backprop the whole network
        - The stacked RBMs are unrolled to create a deep architecture (a deep "autoencoder" in the original paper), and then backprop is applied to find more accurate weights

Deep belief networks (DBNs): RBM



- A undirected graphical model with a bipartite graph structure
  - Input layer $\mathbf{v} = \{v_1, v_2, ..., v_m\}^\top$, hidden layer $\mathbf{h} = \{h_1, h_2, ..., h_n\}^\top$
- Probability distribution it represents:

$$P(\mathbf{v}, \mathbf{h}) = \frac{1}{Z} e^{-E(\mathbf{v},\mathbf{h})}$$

  where Z is the partition function to make $\sum P = 1$
- The energy function $E(\mathbf{v}, \mathbf{h})$ is given as

$$E(\mathbf{v}, \mathbf{h}) = -\mathbf{b}^\top \mathbf{v} - \mathbf{c}^\top \mathbf{h} - \mathbf{h}^\top \mathbf{W} \mathbf{v}$$

Deep belief networks (DBNs): RBM (cont'd)

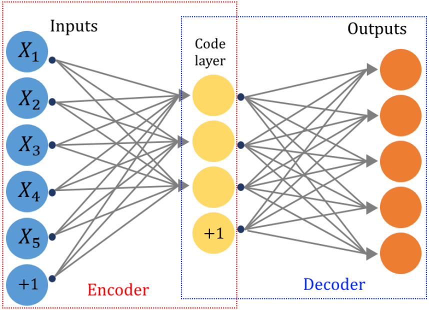
- The expected value of a hidden node:

$$P(h_i = 1|\mathbf{v}) = \sigma(\sum_{j=1}^{m} W_{ij} v_j + c_i)$$

- The expected value of a visible node can be modeled in a similar way, in the opposite direction
- Each direction of a RBM can be seen as a feedforward NN ($\mathbf{v} \longrightarrow \mathbf{h}$, $\mathbf{h} \longrightarrow \mathbf{v}$) with the sigmoid activation function

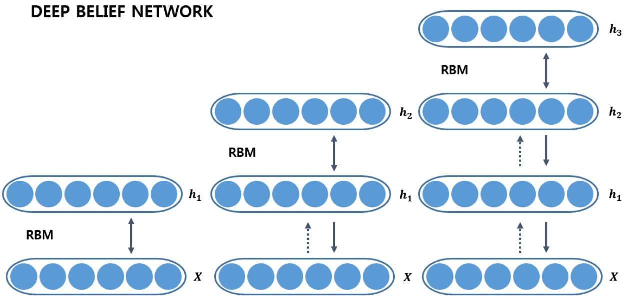
Deep belief networks (DBNs): autoencoder

- What it does: dimensionality reduction (nonlinear PCA)
  - Encoder network: transforms the high-dimensional input data into a low-dimensional code
  - Decoder network: recovers the data from the code
- Each network (encoder/decoder) can be seen as a feedforward NN



Training a DBN

1. Train the 1st layer as an RBM seeing the raw input as its visible layer
2. Use the 1st layer's hidden layer as the 2nd layer's visible layer
3. Train the 2nd layer as an RBM, taking the transformed data (e.g., samples or mean activations) of the 1st layer's output as its input
4. Iterate 2 and 3 for the desired number of layers
5. Unroll RBMs to create a deep architecture and fine-tune all parameters of the whole network using backprop

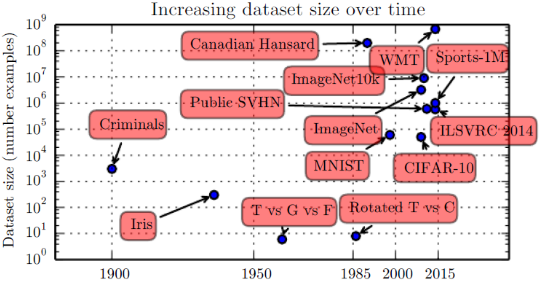


Modern deep learning (2012 –)

- DBNs and layer-wise unsupervised pretraining are no longer widely used
- Backprop struck back!
  - ReLUs replaced sigmoids – facilitated backprop (Glorot et al., 2011)
  - "Using a rectifying nonlinearity is the single most important factor in improving the performance of a recognition system" (Jarrett et al., 2009)
- Training modern DNNs $=$ ReLU $+$ backprop

Renaissance - reason 2: going big

- Training DNNs has been regarded as an art rather than a science
- With more training data, the “art” part diminishes

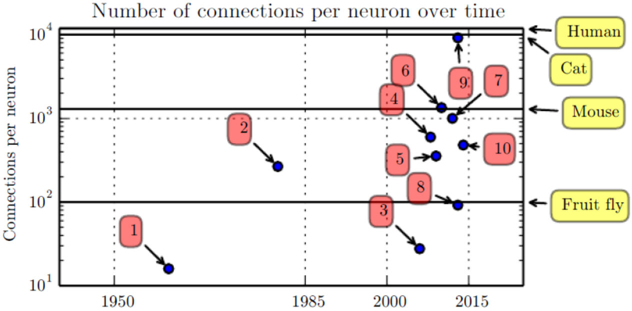


©: [Goodfellow et al., Deep learning, 2016]

- “The most important new development is that today we can provide these algorithms (ReLU + backprop) with the resources they need to succeed” (Goodfellow et al., 2016)
- Made possible by the “Big Data” era

Renaissance - reason 3: computational resources

- Computers can run much larger models than those of 80s



©: [Goodfellow et al., Deep learning, 2016]

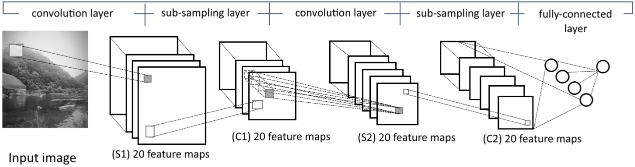
1. Adaptive linear element (Widrow and Hoff, 1960)
2. Neocognitron (Fukushima, 1980)
3. GPU-accelerated CNN (Chellapilla et al., 2006)
4. Deep Boltzmann machine (Salakhutdinov and Hinton, 2009a)
5. Unsupervised CNN (Jarrett et al., 2009)
6. GPU-accelerated multilayer perceptron (Ciresan et al., 2010)
7. Distributed autoencoder (Le et al., 2012)
8. Multi-GPU CNN (Krizhevsky et al., 2012)
9. COTS HPC unsupervised CNN (Coates et al., 2013)
10. GoogLeNet (Szegedy et al., 2014a)

Renaissance - reason 3: computational resources (cont'd)

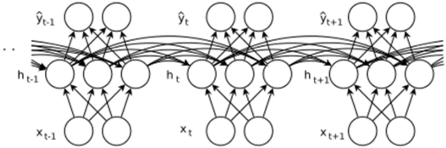
- Largely due to the advent of general purpose GPUs
  - Massive amount of independent computations (multiplying many vectors with the <u>same</u> matrix)
  - No branching
  - high degree of parallelism, high memory bandwidth
- Fit computational requirements for DNN algorithms
  - Locality of backprop $\longrightarrow$ no branching; parallel update
  - Large and numerous buffers of parameters updated every iteration $\longrightarrow$ require high memory bandwith
- GPUs are cheaper
  - Other choices: fast CPUs + high-speed network = expensive

Two most successful deep architectures

- Convolutional Neural Networks (CNN)
    - Excellent for image data



- Recurrent Neural Networks (RNN)
    - Excellent for sequential data



Outline

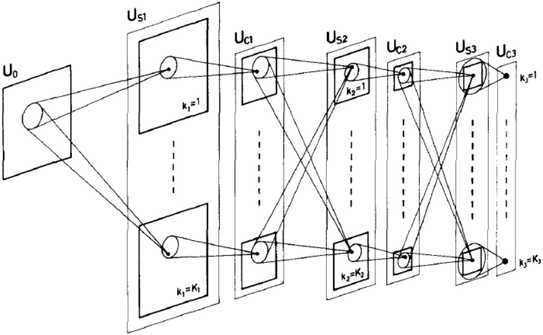
- Introduction
- Convolutional neural networks (CNN)
    - A brief history
    - Architecture and properties
    - How to train
    - Applications
- Recurrent neural networks (RNN)
- A statistical view of deep learning
- Open source tools

Outline

- Introduction
- Convolutional neural networks (CNN)
  - A brief history
  - Architecture and properties
  - How to train
  - Applications
- Recurrent neural networks (RNN)
- A statistical view of deep learning
- Open source tools

A brief history of CNN

- Neocognitron (Fukushima, 1980)
    - Hierarchical & shift invariant model for vision problem
    - Lacks supervised training algorithm



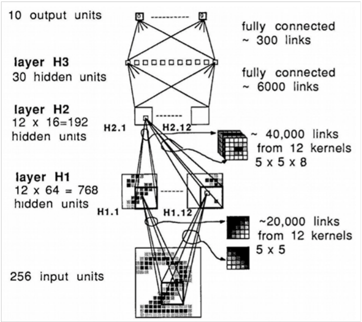
Interconnections between layers in the neocognitron

©: [K. Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position, 1980]

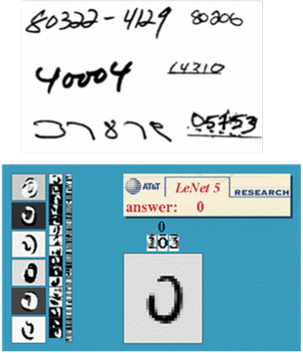
A brief history of CNN (cont'd)

- LeNet (LeCun, 1989)
  - Supervised training for CNN (SGD, back-propagation)

First CNN by LeCun



Hand-digit recognition



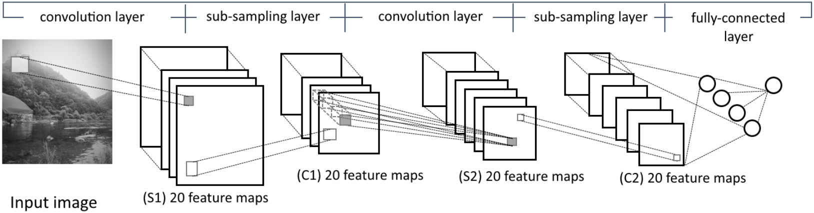
©: [LeCun et. al., Backpropagation applied to handwritten zip code recognition, 1989]

Outline

- Introduction
- Convolutional neural networks (CNN)
    - A brief history
    - Architecture and properties
    - How to train
    - Applications
- Recurrent neural networks (RNN)
- A statistical view of deep learning
- Open source tools

Architecture

- (Convolution + pooling) × k layers + fully-connected layer



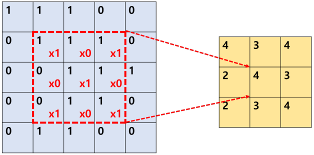
Properties

- Very well-suited to image recognition
- Sparse connections and parameter sharing at the convolution layer
- Powerful regularization techniques (notably “dropout”) to reduce overfitting at the fully-connected layer
- Using ReLUs instead of sigmoid functions

⟶ We will cover the detail while seeing each layer

Convolution layer

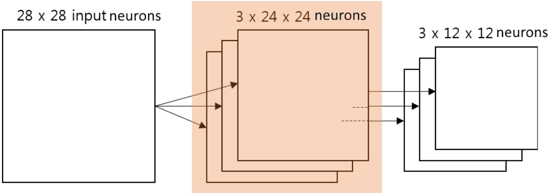
- Extracts feature maps by repeatly applying a kernel (= "convolution" + nonlinearity) across a finite support of the input image called "receptive fields"
    - To form a richer representation of the data, each hidden layer is composed of multiple feature maps, $\{\mathbf{h}^k, k = 0, \dots, K\}$
    - The k^{th} feature map at a given layer: $h^k_{ij} = f((\mathbf{W}^k * \mathbf{x})_{ij} + b_k)$
        - $*$: convolution operator
        - f: nonlinear function (e.g., ReLU)



1	1	1	0	0
0	1 x1	1 x0	1 x1	0
0	0 x0	1 x1	1 x0	1
0	0 x1	1 x0	1 x1	0
0	1	1	0	0

4	3	4
2	4	3
2	3	4

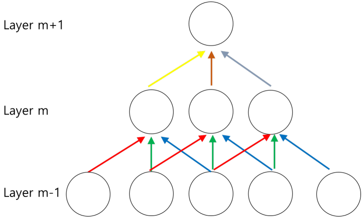
Example of a convolution filter



Example of a convolution layer with 3 feature maps

Convolution layer

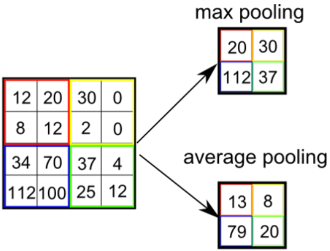
- Sparse connectivity
  - Only connections from units having spatially continuous receptive fields
  - Each unit is unresponsive to variations outside of its receptive field
  - The learnt filters produce the strongest response to a spatially local input pattern, but can be global if we stack many layers
- Weight sharing
  - Weight matrix (tensor) **W** for each layer is sparse (block Toeplitz)
  - Achieves shift (translation) invariance
  - Increases learning efficiencty



Example of sparse connections and parameter sharing

Pooling layer (sub-sampling layer)

- Dimension & computation reduction
  - Pooling is a step that summarizes the output values over a neighborhood
  - Can obtain effciencies in computing and memory
- Translation invariance
  - Makes the model robust to small change in the input
- Usally used immediately after the convolution layer



12	20	30	0
8	12	2	0
34	70	37	4
112	100	25	12

max pooling

20	30
112	37

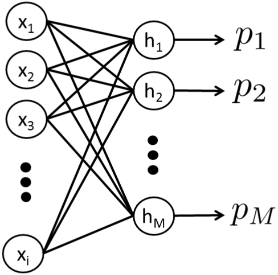
average pooling

13	8
79	20

Examples of pooling: max pooling and average pooling

Fully-connected (classification) layer

- Learns at the more abstract level, integrating the global information from across the entire image, whereas convolution and pooling layers learn about the local spatial structures in the image
- Architecture



Softmax output layer:

$$p_i = \frac{exp(h_i)}{\sum_j exp(h_j)}$$

Cross-entropy loss:

$$L(\mathbf{y}, \mathbf{p}) = -\sum_i^M y_i \log p_i$$

How can we improve the performance of CNNs?

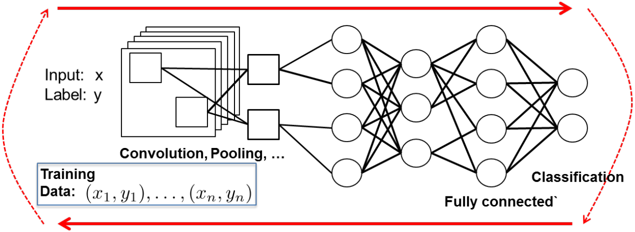
- Apply a second convolution-pooling layer
- Use ReLUs
- Regularization
  - helps networks avoid local minima or getting overfitted
  - e.g., weight decay (L1/L2 regularization), dropout, early stopping
- Artificially expand the training data
  - e.g., rotating, translating, or skewing the training image
- Use an ensemble of networks
  - Create several neural networks and then make them vote to determine the best classification

Outline

- Introduction
- Convolutional neural networks (CNN)
    - A brief history
    - Architecture and properties
    - How to train
    - Applications
- Recurrent neural networks (RNN)
- A statistical view of deep learning
- Open source tools

Training CNN: backprop + *mini-batch* SGD

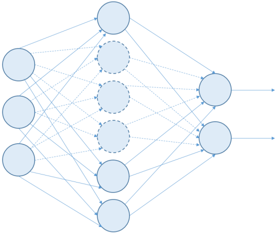
- Forward pass: compute the cumulated loss


Input: x
Label: y
Convolution, Pooling, ...
Training
Data: $(x_1, y_1), \ldots, (x_n, y_n)$
Classification
Fully connected`

- Backward pass: update filter weights using the stochastic gradient descent (SGD)

Training CNN: dropout

- Overfitting
  - Good for training set, but bad for test set
  - Happens when a model is large (= a large number of parmeters) but we have small training data
- Dropout
  - At each forward pass in the final fully-connected layers, randomly drop each node with probability P (typically 0.5)
  - Empirically turns out to be an excellent regularization



Outline

- Introduction
- Convolutional neural networks (CNN)
  - A brief history
  - Architecture and properties
  - How to train
  - Applications
- Recurrent neural networks (RNN)
- A statistical view of deep learning
- Open source tools

Applications of CNN: image recognition

- ImageNet
  - 15 million labeled images(224x224) for 22000 classes
  - Managed by Stanford & UNC Chapel Hill
- ILSVRC (ImageNet Large-Scale Visual Recognition Challenge)
  - 1.2 million (training), 50K (validation), 150K (testing)
  - 1000 classes (roughly 1000 images/class)
  - Annual challenge since 2010

IMAGENET

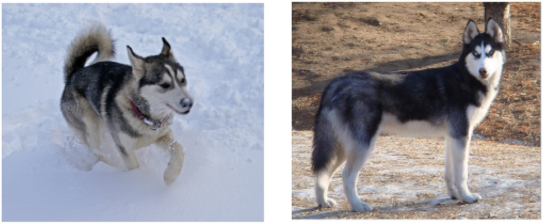
©: http://www.image-net.org

Example images from the classification task

©: http://vision.stanford.edu/Datasets

Two main tasks in ILSVRC: classification

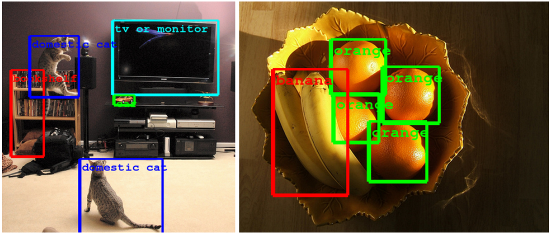
- Classification (image-level annotation)
  - “There is a Siberian husky in this image”, “there are no tigers”
  - Metric: top-5 error (correct if the true class is in top-5 predicted classes)



Siberian husky (left) vs. Eskimo dog (right) from the 1000 classes of ILSVRC 2014

Two main tasks in ILSVRC: detection

- Detection (object-level annotation)
  - “There is an orange centered at position (100, 100) with width of 50 pixels and height of 40 pixels.”
  - Metric: mAP (mean average precision)

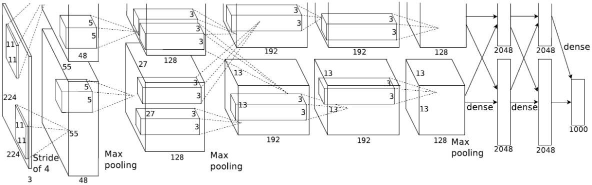


Examples of image detection done by the GoogLeNet team in ILSVRC 2014

©: http://googleresearch.blogspot.com/2014/09/building-deeper-understanding-of-images.html

Classification: AlexNet

- Winner of ILSVRC 2012
- First large-scale implementation of deep CNN using GPUs
- “The current intensity of commercial interests in deep learning begun”
- Architecture
  - 5 convolution layers, 2 fully connected layers (60M parameters, used 2 NVIDIA GPUs and CUDA library [cuda-convnet])
  - 2x2 max-pooling, ReLU
  - Regularization: data augmentation (translation, reflection, intensity), dropout for fully connected layers

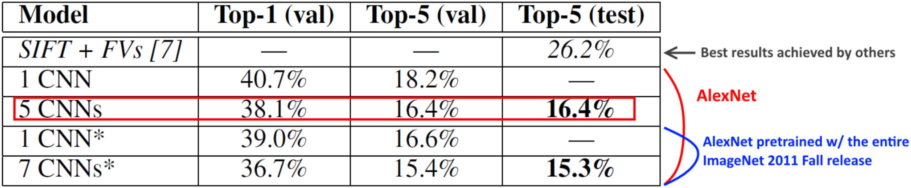


The architecture of AlexNet showing the delineation of responsibilities between the two GPUs

©: [Krizhevsky et. al., ImageNet classification with deep convolutional neural networks, 2012]

Classification: AlexNet (cont'd)

- Result on ILSVRC 2012
  - 38% better than the second best
    - A shocking result for a single improvement!
  - AlexNet was the only model that used CNN in 2012
  - Ensemble of multiple models was also important



Model	Top-1 (val)	Top-5 (val)	Top-5 (test)
SIFT + FVs [7]	—	—	*26.2%*
1 CNN	40.7%	18.2%	—
5 CNNs	38.1%	16.4%	**16.4%**
1 CNN*	39.0%	16.6%	—
7 CNNs*	36.7%	15.4%	**15.3%**

Best results achieved by others

AlexNet

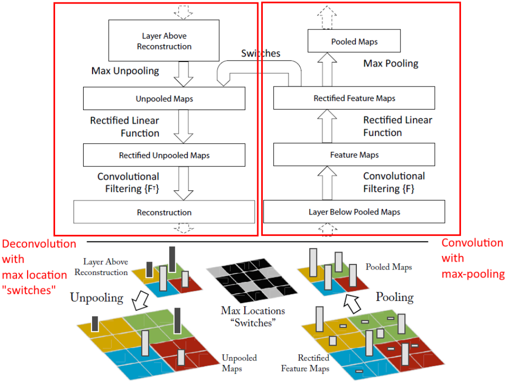
AlexNet pretrained w/ the entire ImageNet 2011 Fall release

Comparison of error rates on ILSVRC 2012 validation and test sets

©: [Krizhevsky et. al., ImageNet classification with deep convolutional neural networks, 2012]

Classification: ZFNet (DeconvNet)

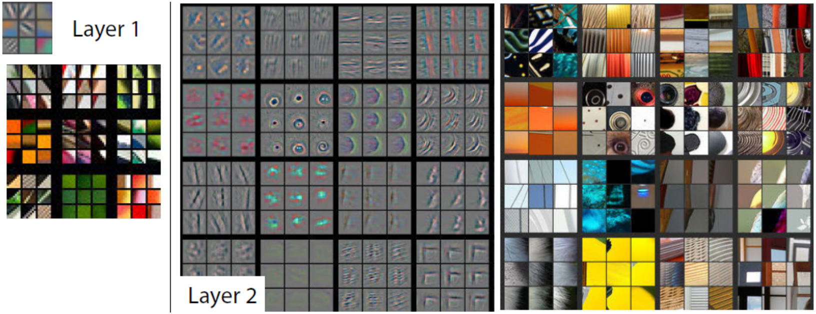
- Winner of ILSVRC 2013
- Devised a visualization technique for CNN ("deconvolution")
  - To see what the convolutional filters are learning, project down each layer's representation to pixel level



©: [Zeiler and Fergus, Visualizing and understanding convolutional networks, 2013]

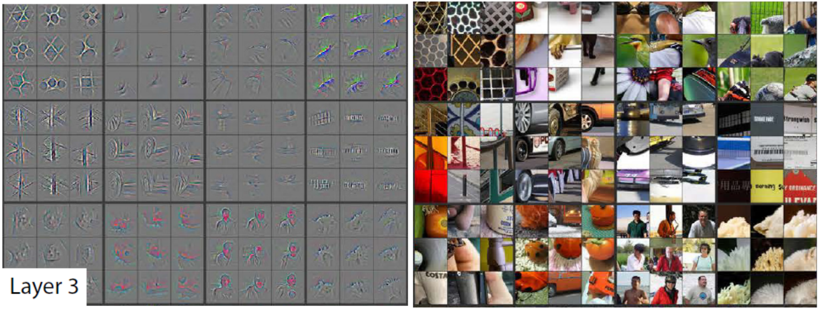
Classification: ZFNet (DeconvNet) (cont’d)

- Visualization of convolution filters (Layer 1, 2)
  - Layer 1: lower level cues (edge, color, etc.)
  - Layer 2: partial objects



©: [Zeiler and Fergus, Visualizing and understanding convolutional networks, 2013]

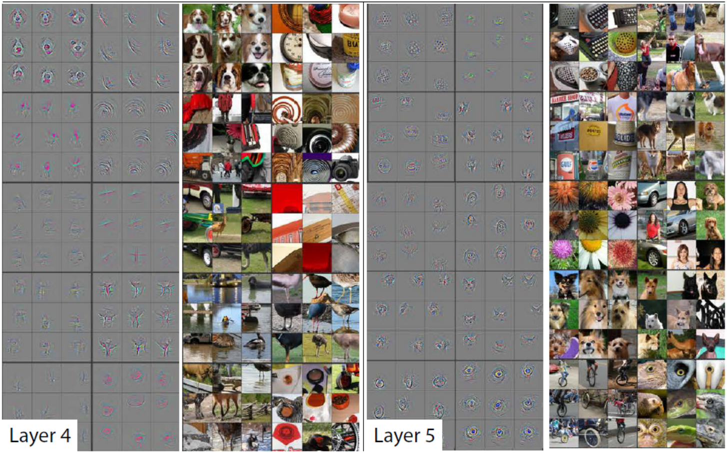
- Layer 3: higher level objects



©: [Zeiler and Fergus, Visualizing and understanding convolutional networks, 2013]

Classification: ZFNet (DeconvNet) (cont'd)

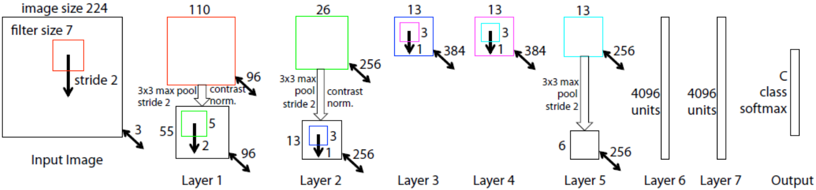
- Layer 4, 5: even higher level objects



©: [Zeiler and Fergus, Visualizing and understanding convolutional networks, 2013]

Classification: ZFNet (DeconvNet) (cont'd)

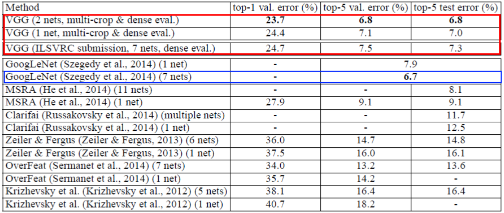
- Utilized visualization for selecting better parameters
  - First layer (11x11, stride 4) ⟶ (7x7, stride 2)
  - New model learns much diverse and stable filters
- Achieved 11.7% top-5 error
  - Another 28% error reduction over AlexNet!



8 layer convnet model with a 224x224 RGB image as an input
©: [Zeiler and Fergus, Visualizing and understanding convolutional networks, 2013]

Classification: VGGNet

- ILSVRC 2014 runner-up
- Motivation: how deep can CNN go?
  - Fix filter/stride size and increase depth up to 19 weight layers
- Demonstrated that the representation depth is beneficial for the classification accuracy, using a conventional CNN architecture with substantially increased depth (16-19 weight layers, 140M parameters)
- Later found that the VGGNet features transfer well to other tasks; pretrained model available publicly



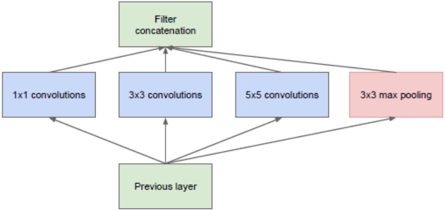
Method	top-1 val. error (%)	top-5 val. error (%)	top-5 test error (%)
VGG (2 nets, multi-crop & dense eval.)	**23.7**	**6.8**	**6.8**
VGG (1 net, multi-crop & dense eval.)	24.4	7.1	7.0
VGG (ILSVRC submission, 7 nets, dense eval.)	24.7	7.5	7.3
GoogLeNet (Szegedy et al., 2014) (1 net)	-	7.9	
GoogLeNet (Szegedy et al., 2014) (7 nets)	-	**6.7**	
MSRA (He et al., 2014) (11 nets)	-	-	8.1
MSRA (He et al., 2014) (1 net)	27.9	9.1	9.1
Clarifai (Russakovsky et al., 2014) (multiple nets)	-	-	11.7
Clarifai (Russakovsky et al., 2014) (1 net)	-	-	12.5
Zeiler & Fergus (Zeiler & Fergus, 2013) (6 nets)	36.0	14.7	14.8
Zeiler & Fergus (Zeiler & Fergus, 2013) (1 net)	37.5	16.0	16.1
OverFeat (Sermanet et al., 2014) (7 nets)	34.0	13.2	13.6
OverFeat (Sermanet et al., 2014) (1 net)	35.7	14.2	-
Krizhevsky et al. (Krizhevsky et al., 2012) (5 nets)	38.1	16.4	16.4
Krizhevsky et al. (Krizhevsky et al., 2012) (1 net)	40.7	18.2	-

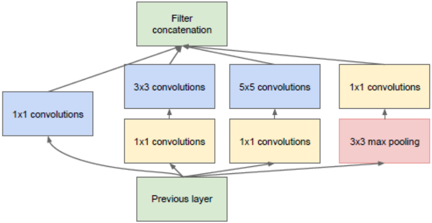
Comparison of error rates on ILSVRC 2014 with the state of the art

©: [Simonyan and Zisserman, Very deep convolutional networks for large-scale image recognition, 2015]

Classification: GoogLeNet

- Winner of ILSVRC 2014
- Motivation: will large model with large data solve everything?
  - No! (overfitting, memory limit)
  - Sparsity may help ⟶ But, GPU cannot handle true sparsity
- Inception module
  - Try to benefit from both sparsity and parallelization
  - Convolution layer decomposes into filters with multiple scales



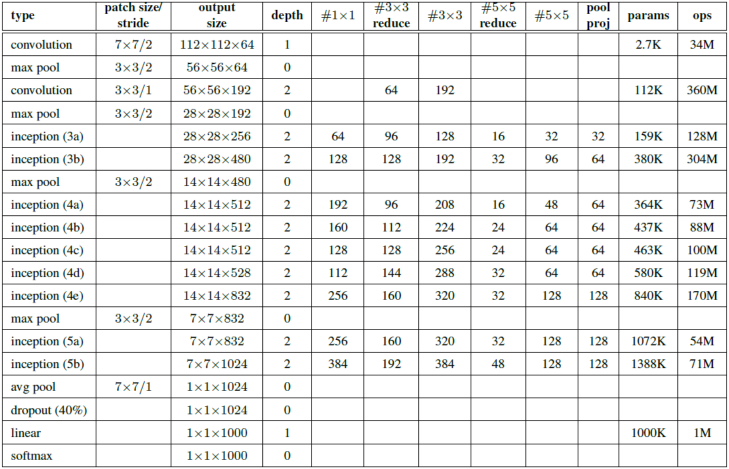


Inception modules. Naive version (left) vs. dim. reduction version (right)

©: [Szegedy et. al., Going deeper with convolutions, 2014]

Classification: GoogLeNet (cont'd)

- Architecture

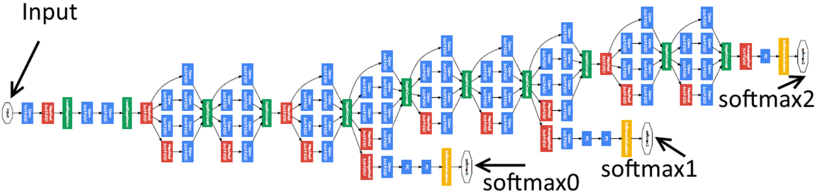


type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout (40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

©: [Szegedy et. al., Going deeper with convolutions, 2014]

Classification: GoogLeNet (cont'd)

- Architecture
  - 22 layers, but 12 times fewer parameters than AlexNet (~6M parameters)

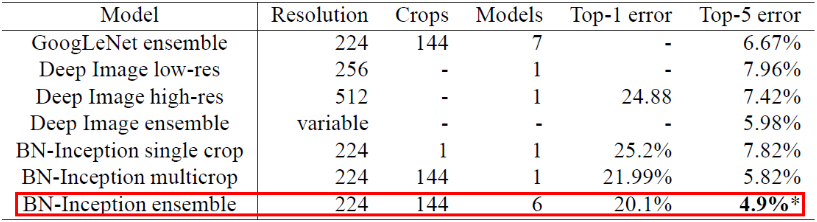


GoogLeNet network with all the bells and whistles
©: [Szegedy et. al., Going deeper with convolutions, 2014]

- Achieved 6.67% top-5 error
  - Another 43% error reduction over DeconvNet!

Classification: batch-normalized CNNs (BN-CNN)

- GoogLeNet + batch normalization
- Achieved 4.82% top-5 error on ILSVRC 2014 set
    - Another 28% error reduction over GoogLeNet!
    - 71% reduction over AlexNet!
    - 82% reduction over pre-CNN in 3 years!



Model	Resolution	Crops	Models	Top-1 error	Top-5 error
GoogLeNet ensemble	224	144	7	-	6.67%
Deep Image low-res	256	-	1	-	7.96%
Deep Image high-res	512	-	1	24.88	7.42%
Deep Image ensemble	variable	-	-	-	5.98%
BN-Inception single crop	224	1	1	25.2%	7.82%
BN-Inception multicrop	224	144	1	21.99%	5.82%
BN-Inception ensemble	224	144	6	20.1%	**4.9%***

Comparison with previous state of the art on ILSVRC 2014 validation set

©: [Ioffe and Szegedy, Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015]

Batch normalization

- Internal covariate shift
  - The distribution of each layer's inputs changes during training, as the parameters of the previous layers change
  - This slows down the training by requiring lower learning rates and careful parameter initialization
- Batch normalization
  - Can reduce internal covariate shift by performing a normalization of each activation x ($=$ each input of a layer) over a mini-batch
  - Allows us to use much higher learning rates and be less careful about initialization by reducing the dependence of gradients on the scale of the parameters or of their initial values
  - Regularizes the model and reduces the need for dropout

Batch normalization (cont'd)

- Batch normalizing transform

Input: values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$

Input: parameters to be learned: γ, β

Output:$\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

- The scaling & shifting step enables BN transform to represent the identity transfrom ($\gamma = \sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}$, $\beta = \mu_{\mathcal{B}}$), which is required as simply normalizing each input of a layer may change what the layer can represent

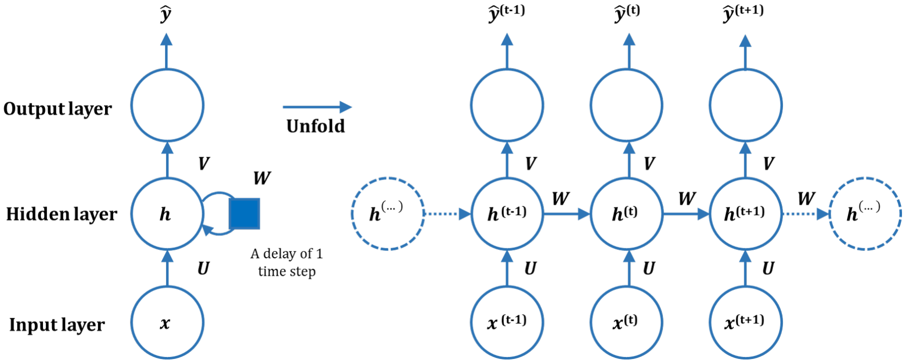
Outline

- Introduction
- Convolutional neural networks (CNN)
- Recurrent neural networks (RNN)
  - Architecure and properties
  - How to train
  - Vanishing/exploding gradients problem
  - Long short-term memory (LSTM)
  - Applications
- A statistical view of deep learning
- Open source tools

Outline

- Introduction
- Convolutional neural networks (CNN)
- Recurrent neural networks (RNN)
    - Architecure and properties
    - How to train
    - Vanishing/exploding gradients problem
    - Long short-term memory (LSTM)
    - Applications
- A statistical view of deep learning
- Open source tools

Architecture



$$\mathbf{h}^{(t)} = f(\mathbf{W}\mathbf{h}^{(t-1)} + \mathbf{U}\mathbf{x}^{(t)} + \mathbf{b})$$
$$\hat{\mathbf{y}}^{(t)} = g(\mathbf{V}\mathbf{h}^{(t)} + \mathbf{c})$$

Properties

- Handling seqeuntial data
  - e.g., language model (predicting next word given past)
- Memory
  - You can think of the hidden state $\mathbf{h}^{(t)}$ as the (lossy) "memory" of the network, which captures information in all the previous time steps
- Parameters sharing
  - The same parameters $\mathbf{W}$, $\mathbf{U}$, $\mathbf{V}$, $\mathbf{b}$, $\mathbf{c}$ across all steps $\longrightarrow$ perform the same task at each step, just with different inputs ("recurrent")
- In theory, the number of time steps can be very deep, but in practice, are limited to look back only a few steps

Outline

- Introduction
- Convolutional neural networks (CNN)
- Recurrent neural networks (RNN)
    - Architecure and properties
    - How to train
    - Vanishing/exploding gradients problem
    - Long short-term memory (LSTM)
    - Applications
- A statistical view of deep learning
- Open source tools

Training RNNs

- Express a RNN as a unfolded computational graph, and then apply the Back-propagation algorithm through time (BPTT)
- Given a training set $\{(\mathbf{x}^{(t)}, \mathbf{y}^{(t)})\}_{t=1}^{\tau}$, the runtime is $O(\tau)$ and cannot be reduced by parallelization because it is inherently sequential; each time step can only be computed after the previous one
- The memory cost is also $O(\tau)$ as states computed in the forward pass must be stored until they are reused during the backward pass

Back-propagation through time (BPTT): example

- (An example from [Goodfellow et al., Deep learning, 2016]) $f = \tanh$, g=softmax, and the loss is the negative log-likelihood of the true target $\mathbf{y}^{(t)}$ given the input so far

$$
\begin{aligned}
\mathbf{h}^{(t)} &= \tanh(\mathbf{W}\mathbf{h}^{(t-1)} + \mathbf{U}\mathbf{x}^{(t)} + \mathbf{b}) \\
\mathbf{o}^{(t)} &= \mathbf{V}\mathbf{h}^{(t)} + \mathbf{c} \\
\hat{\mathbf{y}}^{(t)} &= \mathrm{softmax}(\mathbf{o}^{(t)}) \\
\mathcal{L} &= \sum_t \mathcal{L}^{(t)} = \sum_t \mathcal{L}(\mathbf{y}^{(t)}, \hat{\mathbf{y}}^{(t)}) = \sum_t - \log \hat{y}^{(t)}_{y^{(t)}}
\end{aligned}
$$

$$\longrightarrow \nabla_{\mathbf{c}}\mathcal{L}, \nabla_{\mathbf{b}}\mathcal{L}, \nabla_{\mathbf{V}}\mathcal{L}, \nabla_{\mathbf{W}}\mathcal{L}, \nabla_{\mathbf{U}}\mathcal{L}?$$

Back-propagation through time (BPTT): example (cont'd)

- Start the recursion with the nodes immediately preceding the final loss

$$\frac{\partial \mathcal{L}}{\partial \mathcal{L}^{(t)}} = 1$$

- On the outputs at time step t

$$(\nabla_{\mathbf{o}^{(t)}} \mathcal{L})_i = \frac{\partial \mathcal{L}}{\partial o_i{}^{(t)}} = \frac{\partial \mathcal{L}}{\partial \mathcal{L}^{(t)}} \frac{\partial \mathcal{L}^{(t)}}{\partial o_i{}^{(t)}} = \hat{y}_i^{(t)} - \mathbf{1}_{i,y^{(t)}}$$

- At the final time step τ, $\mathbf{h}^{(\tau)}$ only has $\mathbf{o}^{(\tau)}$ as a descendent

$$\nabla_{\mathbf{h}^{(\tau)}} \mathcal{L} = \left(\frac{\partial \mathbf{o}^{(\tau)}}{\partial \mathbf{h}^{(\tau)}} \right)^{\top} \nabla_{\mathbf{o}^{(\tau)}} \mathcal{L} = \mathbf{V}^{\top} \nabla_{\mathbf{o}^{(\tau)}} \mathcal{L}$$

- Iterate backwards through time, from $t = \tau - 1$ down to $t = 1$, noting that $\mathbf{h}^{(t)}$ (for $t < \tau$) has descendents both and $\mathbf{o}^{(t)}$ and $\mathbf{h}^{(t+1)}$

$$\begin{aligned}\nabla_{\mathbf{h}^{(t)}} \mathcal{L} &= \left(\frac{\partial \mathbf{h}^{(t+1)}}{\partial \mathbf{h}^{(t)}} \right)^{\top} (\nabla_{\mathbf{h}^{(t+1)}} \mathcal{L}) + \left(\frac{\partial \mathbf{o}^{(t)}}{\partial \mathbf{h}^{(t)}} \right)^{\top} (\nabla_{\mathbf{o}^{(t)}} \mathcal{L}) \\ &= \mathbf{W}^{\top} (\nabla_{\mathbf{h}^{(t+1)}} \mathcal{L}) \mathrm{diag} \left(1 - (\mathbf{h}^{(t+1)})^2 \right) + \mathbf{V}^{\top} (\nabla_{\mathbf{o}^{(t)}} \mathcal{L})\end{aligned}$$

Back-propagation through time (BPTT): example (cont'd)

- To clarify our notation, we introduce dummy variables $\mathbf{W}^{(t)}$ ($\mathbf{U}^{(t)}$) that are defined to be copies of $\mathbf{W}$ ($\mathbf{U}$) but with each $\mathbf{W}^{(t)}$ ($\mathbf{U}^{(t)}$) used only time step t. Then let's use $\nabla_{\mathbf{W}^{(t)}}$ ($\nabla_{\mathbf{U}^{(t)}}$) to denote the contribution of the weights at time step t to the gradient. $\Longrightarrow$

$$\nabla_{\mathbf{c}}\mathcal{L} = \sum_t \left(\frac{\partial \mathbf{o}^{(t)}}{\partial \mathbf{c}}\right)^\top \nabla_{\mathbf{o}^{(t)}}\mathcal{L} = \sum_t \nabla_{\mathbf{o}^{(t)}}\mathcal{L}$$

$$\nabla_{\mathbf{b}}\mathcal{L} = \sum_t \left(\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{b}}\right)^\top \nabla_{\mathbf{h}^{(t)}}\mathcal{L} = \sum_t \operatorname{diag}\left(1 - (\mathbf{h}^{(t)})^2\right) \nabla_{\mathbf{h}^{(t)}}\mathcal{L}$$

$$\nabla_{\mathbf{V}}\mathcal{L} = \sum_t \sum_i \left(\frac{\partial \mathcal{L}}{\partial o_i^{(t)}}\right) \nabla_{\mathbf{V}} o_i^{(t)} = \sum_t (\nabla_{\mathbf{o}^{(t)}}\mathcal{L})\mathbf{h}^{(t)\top}$$

$$\nabla_{\mathbf{W}}\mathcal{L} = \sum_t \sum_i \left(\frac{\partial \mathcal{L}}{\partial h_i^{(t)}}\right) \nabla_{\mathbf{W}^{(t)}} h_i^{(t)} = \sum_t \operatorname{diag}\left(1 - (\mathbf{h}^{(t)})^2\right) (\nabla_{\mathbf{h}^{(t)}}\mathcal{L})\mathbf{h}^{(t-1)\top}$$

$$\nabla_{\mathbf{U}}\mathcal{L} = \sum_t \sum_i \left(\frac{\partial \mathcal{L}}{\partial h_i^{(t)}}\right) \nabla_{\mathbf{U}^{(t)}} h_i^{(t)} = \sum_t \operatorname{diag}\left(1 - (\mathbf{h}^{(t)})^2\right) (\nabla_{\mathbf{h}^{(t)}}\mathcal{L})\mathbf{x}^{(t)\top}$$

Outline

- Introduction
- Convolutional neural networks (CNN)
- Recurrent neural networks (RNN)
  - Architecure and properties
  - How to train
  - Vanishing/exploding gradients problem
  - Long short-term memory (LSTM)
  - Applications
- A statistical view of deep learning
- Open source tools

Problems with BPTT: vanishing or exploding gradients

- When unfolded, the depth of RNN can reach 1000s
  - In a language model example below, RNN tends to predict better for the paragraph 1
    - Paragraph 1: "Jane walked into the room. John walked in too. Jane said hi to _____."
    - Paragraph 2: "Jane walked into the room. John walked in too. It was late in the day, and everyone was walking home after a long day at work. Jane said hi to _____."
- Gradients propagated over many stages tend to either vanish (most of the time) or explode (rarely, but with much damage to the optimization)

Vanishing or exploding gradients

- Let's see a simple RNN

$$\mathbf{h}^{(t)} = \mathbf{W} f(\mathbf{h}^{(t-1)}) + \mathbf{U}\mathbf{x}^{(t)} + \mathbf{b}$$
$$\hat{\mathbf{y}}^{(t)} = \mathbf{V} f(\mathbf{h}^{(t)})$$

- For analysis, apply chain rule rather than back-propagation

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \sum_{t=1}^{\tau} \frac{\partial \mathcal{L}^{(t)}}{\partial \mathbf{W}}$$
$$\frac{\partial \mathcal{L}^{(t)}}{\partial \mathbf{W}} = \sum_{k=1}^{t} \frac{\partial \mathcal{L}^{(t)}}{\partial \hat{\mathbf{y}}^{(t)}} \frac{\partial \hat{\mathbf{y}}^{(t)}}{\partial \mathbf{h}^{(t)}} \frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(k)}} \frac{\partial \mathbf{h}^{(k)}}{\partial \mathbf{W}}$$

$$\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(k)}} = \prod_{j=k+1}^{t} \frac{\partial \mathbf{h}^{(j)}}{\partial \mathbf{h}^{(j-1)}} = \prod_{j=k+1}^{t} \mathbf{W}^{\top} \mathrm{diag}[f'(\mathbf{h}^{(j-1)})]$$

Vanishing or exploding gradients (cont'd)

- If we define β's as upper bounds of the norms

$$\begin{aligned}
&\|\frac{\partial \mathbf{h}^{(j)}}{\partial \mathbf{h}^{(j-1)}}\| \le \|\mathbf{W}^\top\|\|\mathrm{diag}[f'(\mathbf{h}^{(j-1)})]\| \le \beta_{\mathbf{w}}\beta_{\mathbf{h}} \\
\Longrightarrow &\|\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(k)}}\| = \|\prod_{j=k+1}^{t} \frac{\partial \mathbf{h}^{(j)}}{\partial \mathbf{h}^{(j-1)}}\| \le (\beta_{\mathbf{w}}\beta_{\mathbf{h}})^{t-k}
\end{aligned}$$

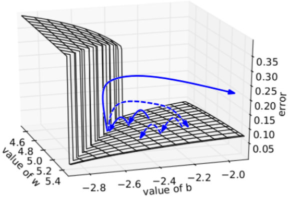
- This can become very small or very large quickly!

Solutions? Make the product of gradients be close to one!

- In the previous slide, we found that the problem mainly occurs due to the value of the norm of Jaconbians $\|\mathbf{J}^{(t)}\| = \|\frac{\partial \mathbf{h}^{(j)}}{\partial \mathbf{h}^{(j-1)}}\|$
- Intuitively, to prevent gradients from vanishing or exploding, we have to make $\|\mathbf{J}^{(t)}\|$ not too small and not too big ≈ 1
- We'll see many solutions in turn, with an emphasis on LSTM

Solution (for exploding): gradient clipping

- The objective function of RNNs often contains sharp nonlinearities in parameter space due to the multiplication of several parameters
- When the parameter gradient is very large, gradient descent could throw the parmeter into a region where the objective function is larger
- Ways to clip the gradient $\mathbf{g}$ (per mini-batch)
    - Clip $\|\mathbf{g}\|$ just before the parameter update if $\|\mathbf{g}\| > v$ by $\mathbf{g} \leftarrow \frac{\mathbf{g}v}{\|\mathbf{g}\|}$
    - Clip g element-wise
    - take a random step if $\|\mathbf{g}\| > v$



Gradient descent without gradient clipping may overshoot the cost function
©: [Pascanu et al., On the diffculty of training recurrent neural networks, 2013a]

Solution: regularization

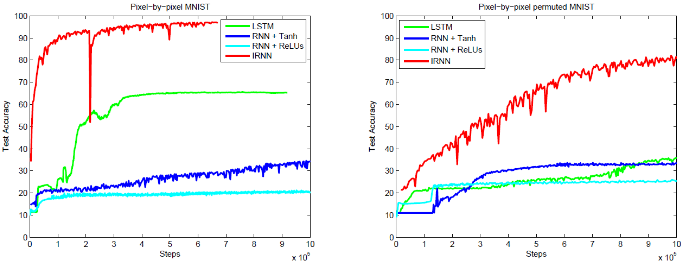
- We would like the gradient vector $\nabla_{\mathbf{h}^{(t)}}\mathcal{L}$ being back-propagated to maintain its magnitude, even if the loss function only penalizes the output at the end of the sequence
- Formally, we want $(\nabla_{\mathbf{h}^{(t)}}\mathcal{L})\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(t-1)}}$ to be as large as $\nabla_{\mathbf{h}^{(t)}}\mathcal{L}$

$$\longrightarrow \Omega = \sum_t \Omega_t = \sum_t \left(\frac{\|(\nabla_{\mathbf{h}^{(t)}}\mathcal{L})\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(t-1)}}\|}{\|\nabla_{\mathbf{h}^{(t)}}\mathcal{L}\|} - 1 \right)^2$$

- Then optimize the cost function regularized by this, i.e., $\tilde{J} = J + \lambda\Omega$

Solution: proper weights & activation functions

- Another way is to initialize the recurrent weight matrix **W** properly, or to select proper activation functions like ReLU
- Combined [Le et al., A simple way to initialize recurrent networks of rectified linear units, 2015]
  - Initialize **W** to be **I** and biases to be zero and use ReLUs
  - Good performance in the MNIST classification experiment, where the sequential inputs are 784 pixels, the output is the cataory, and the networks read one pixel at a time (784 time steps!)



©: [Le et al., A simple way to initialize recurrent networks of rectified linear units, 2015]

Solution: different optimization methods

- Hessian-Free (HF) optimization
    - More sophisticated optimization method than SGD [Martens and Sutskever, Learning recurrent neural networks with Hessian-Free optimization, 2011]
- SGD with momentum and careful initialization
    - SGD with momentum: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \mathbf{v} - \epsilon \nabla_{\boldsymbol{\theta}} \mathcal{L}$
        - Introduces a variable **v** that plays the role of velocity, i.e., the direction and speed at which the parameters move through parameter space
        - Solves two problems: poor conditioning of the Hessian matrix and variance in SGD [Sutskever, Martens, Dahl, and Hinton, On the importance of initialization and momentum in deep learning, 2013]

Solution: echo state networks and leaky units

- Echo state networks
  - Sets the input and reccurent weight $\mathbf{W}$ properly (by fixing the spectral radious of recurrent parameter $\mathbf{W}$) so that the recurrent hidden units capture past information well, and then only learn the ouput weights
  - How to train?
    - Randomly construct a RNN: # of layers, (sparse) $\mathbf{U}$ & $\mathbf{W}$
    - Renormalize the spectral raidous of $\mathbf{W}$: $\mathbf{W} \leftarrow \lambda \frac{\mathbf{W}}{\lambda_M}$
    - Train only the ouput weights
- Leaky (integration) units
  - Hidden recurrent units with linear-self connections and a weight near 1 on these connections
  - $\mathbf{h}^{(t)} = \alpha \mathbf{h}^{(t-1)} + (1-\alpha) f(\mathbf{W}\mathbf{h}^{(t-1)} + \mathbf{U}\mathbf{x}^{(t)} + \mathbf{b})$
  - α is near one, the information is remembered for a long time
  - α is near zero, the information is discarded rapidly
  - α can be chosen manually or learned

Solution: long short-term memory (LSTM)

- As of now, the most effective models based on the idea of making the product of gradients is close to one are long short-term memory (LSTM) and its variants, including the gated recurrent units (GRUs)
- Leaky units vs. LSTM
  - Leaky units $\mathbf{h}^{(t)} = \alpha \mathbf{h}^{(t-1)} + (1-\alpha) f(\mathbf{W}\mathbf{h}^{(t-1)} + \mathbf{U}\mathbf{x}^{(t)} + \mathbf{b})$ have the same α over time, whether it is chosen manually or learned $\longleftrightarrow$ LSTM allows the connection weights to change at each time step
  - Leaky units only accumulate information $\longleftrightarrow$ LSTM can forget the old state
  - The scalar value α can be either near one or near zero, but cannot be both at the same time $\longleftrightarrow$ LSTM achieves this using gate units passed a sigmoid layer

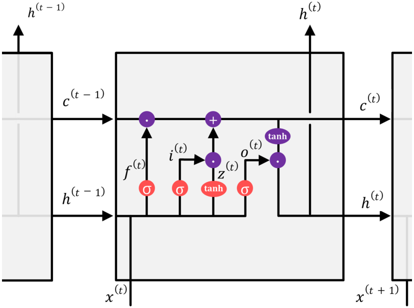
Outline

- Introduction
- Convolutional neural networks (CNN)
- Recurrent neural networks (RNN)
    - Architecure and properties
    - How to train
    - Vanishing/exploding gradients problem
    - Long short-term memory (LSTM)
    - Applications
- A statistical view of deep learning
- Open source tools

Long short-term memory (LSTM)

- The key to LSTMs is the cell state units having an internal recurrence (linear self-loop), which work as a conveyor belt of information
- LSTMs have the ability to remove or add information to the cell state, carefully regulated by structures called gates (= forget/input gates)
- Gates output numbers between 0 and 1, describing how much of each component should be let though
- Invented by Sepp Hochreiter and Jurgen Schmidhuber (1997)

LSTM architecture (unfolded)



$$
\begin{aligned}
\mathbf{h}^{(t)} &= \mathbf{o}^{(t)} \odot g_o(\mathbf{c}^{(t)}) && \text{(block out)}\\
\mathbf{o}^{(t)} &= \sigma(\mathbf{W}_o\mathbf{h}^{(t-1)} + \mathbf{U}_o\mathbf{x}^{(t)} + \mathbf{b}_o) && \text{(output gate)}\\
\mathbf{c}^{(t)} &= \mathbf{i}^{(t)} \odot \mathbf{z}^{(t)} + \mathbf{f}^{(t)} \odot \mathbf{c}^{(t-1)} && \text{(cell state)}\\
\mathbf{f}^{(t)} &= \sigma(\mathbf{W}_f\mathbf{h}^{(t-1)} + \mathbf{U}_f\mathbf{x}^{(t)} + \mathbf{b}_f) && \text{(forget gate)}\\
\mathbf{i}^{(t)} &= \sigma(\mathbf{W}_i\mathbf{h}^{(t-1)} + \mathbf{U}_i\mathbf{x}^{(t)} + \mathbf{b}_i) && \text{(input gate)}\\
\mathbf{z}^{(t)} &= g_i(\mathbf{W}_z\mathbf{h}^{(t-1)} + \mathbf{U}_z\mathbf{x}^{(t)} + \mathbf{b}_z) && \text{(block input)}
\end{aligned}
$$

($\odot$ means element-wise multiplication)

- $\mathbf{o}^{(t)}$: decides what information of $g_o(\mathbf{c}^{(t)})$ we will output (typically $g_o = \tanh$)
- $\mathbf{f}^{(t)}$: decides what information of $\mathbf{c}^{(t-1)}$ we will keep or forget
- $\mathbf{i}^{(t)}$: decides what new information of $\mathbf{z}^{(t)}$ we will store in $\mathbf{c}^{(t)}$
- $\mathbf{z}^{(t)}$: candidate values that can be added to the state (typically $g_i = \tanh$)

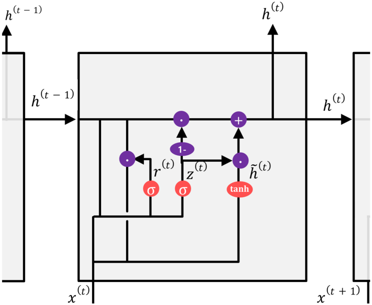
LSTM model - revisited

$$
\begin{aligned}
\mathbf{c}^{(t)} &= \mathbf{i}^{(t)} \odot \mathbf{z}^{(t)} + \mathbf{f}^{(t)} \odot \mathbf{c}^{(t-1)} \\
&= \sigma(\mathbf{W}_i \mathbf{h}^{(t-1)} + \mathbf{U}_i \mathbf{x}^{(t)} + \mathbf{b}_i) \odot g_i(\mathbf{W}_z \mathbf{h}^{(t-1)} + \mathbf{U}_z \mathbf{x}^{(t)} + \mathbf{b}_z) \\
&+ \sigma(\mathbf{W}_f \mathbf{h}^{(t-1)} + \mathbf{U}_f \mathbf{x}^{(t)} + \mathbf{b}_f) \odot \mathbf{c}^{(t-1)} \\
&= F_1(\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}) + F_2(\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}) \odot \mathbf{c}^{(t-1)} \\
&= F(\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}, \mathbf{c}^{(t-1)})
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{h}^{(t)} &= \mathbf{o}^{(t)} \odot g_o(\mathbf{c}^{(t)}) \\
&= \sigma(\mathbf{W}_o \mathbf{h}^{(t-1)} + \mathbf{U}_o \mathbf{x}^{(t)} + \mathbf{b}_o) \odot g_o(\mathbf{c}^{(t)}) \\
&= G(\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}, \mathbf{c}^{(t)})
\end{aligned}
$$

LSTM variants: gated recurrent units (GRUs)

- The main difference with LSTM is the introduction of an “update gate” ($=\mathbf{z}^{(t)}$), which simultaneously controls the forgetting factor and the decision to update the state unit
- The reset gate ($=\mathbf{r}^{(t)}$) controls which parts of the state get used to compute the next target state ($=\tilde{\mathbf{h}}^{(t)}$), introducing an additional nonlinear effect in the relationship between past state and future state
- GRUs also merge the cell state and hidden state



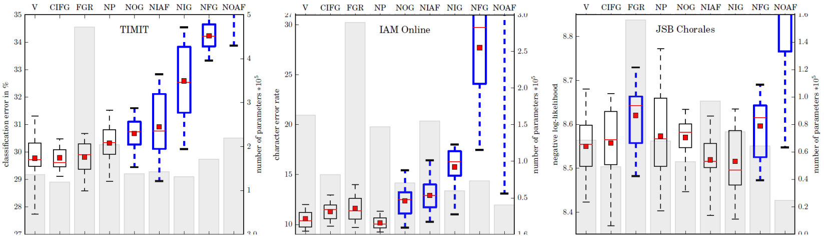
$$
\begin{aligned}
\mathbf{h}^{(t)} &= \mathbf{z}^{(t)} \odot \tilde{\mathbf{h}}^{(t)} + (1 - \mathbf{z}^{(t)}) \odot \mathbf{h}^{(t-1)} && \text{(state gate)}\\
\tilde{\mathbf{h}}^{(t)} &= \tanh(\mathbf{W}_g(\mathbf{r}^{(t)} \odot \mathbf{h}^{(t-1)}) + \mathbf{U}_g\mathbf{x}^{(t)} + \mathbf{b}_g)\\
\mathbf{r}^{(t)} &= \sigma(\mathbf{W}_r\mathbf{h}^{(t-1)} + \mathbf{U}_r\mathbf{x}^{(t)} + \mathbf{b}_r) && \text{(reset gate)}\\
\mathbf{z}^{(t)} &= \sigma(\mathbf{W}_z\mathbf{h}^{(t-1)} + \mathbf{U}_z\mathbf{x}^{(t)} + \mathbf{b}_z) && \text{(update gate)}
\end{aligned}
$$

LSTM variants: performance comparison

- Other variants are:
  - No Input Gate (NIG)
  - No Forget Gate (NFG)
  - No Output Gate (NOG)
  - No Input Activation Function (NIAF)
  - No Output Activation Function (NOAF)
  - No Peepholes (NP)
  - Coupled Input and Forget Gate (CIFG) (= GRUs)
  - Full Gate Recurrence (FGR)
- Which architecture is the best, among the vanilla LSTM and its variants, with different hyperparameters?
  - Empirical evaluations [Greff et al., LSTM: A search space odyssey, 2015]

LSTM variants: performance comparison (cont'd)

- Hyperparamter search



Test set performance for top 10 % hyperparameter settings for each dataset and variant

©: [Greff et al., LSTM: A search space odyssey, 2015]

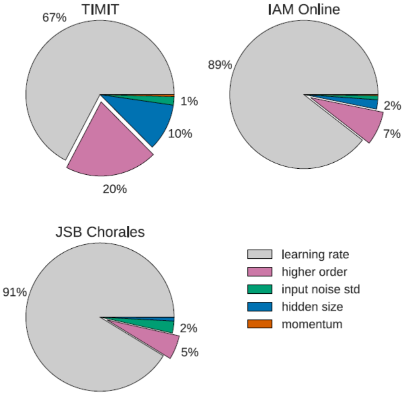
- Vanilla LSTM works well
- CIFG, NP also work reasonably well
- FG, output activation is important

Each hyperparameter search consists of 200 trials (for a total of 5400 trials) of randomly sampling the following hyperparameters:

- # of LSTM blocks per hidden layer: log-uniform samples from [20, 200]
- learning rate: log-uniform samples from [10^{-6}, 10−2]
- momentum: 1 - log-uniform samples from [0.01, 1.0]
- standard deviation of Gaussian input noise: uniform samples from [0, 1]

LSTM variants: performance comparison (cont'd)

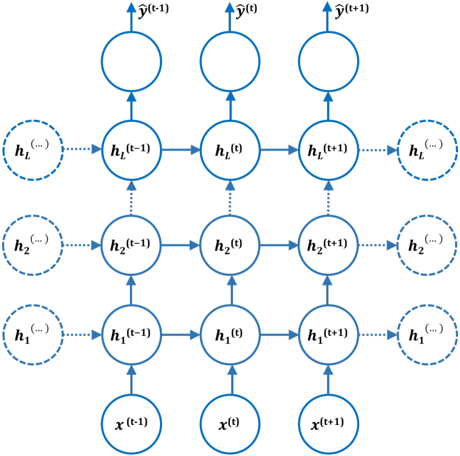
- Test set variance breakdown for each hyperparameter
  - Learning rate is the most sensitive hyperparameter!



©: [Greff et al., LSTM: A search space odyssey, 2015]

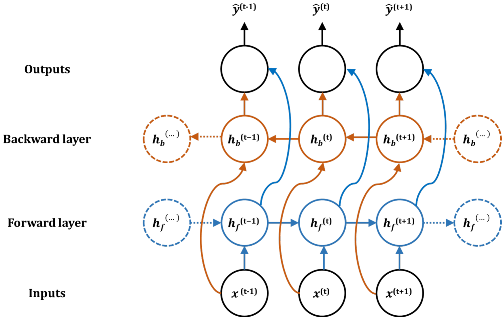
LSTM variants: Deep LSTM (RNN)

- Multiple hidden reccurent states of LSTMs (RNNs)



LSTM variants: bidirectional LSTM (RNN)

- Based on the idea that the output at time t may depend on both the previous and the future elements in the sequence
  - e.g., to predict a missing word in a sequence, we may want to look at both the left and the right context
- Architecture: two LSTMs (RNNs) stacked on top of each other
  - The output is computed based on the hidden state of both LSTMs

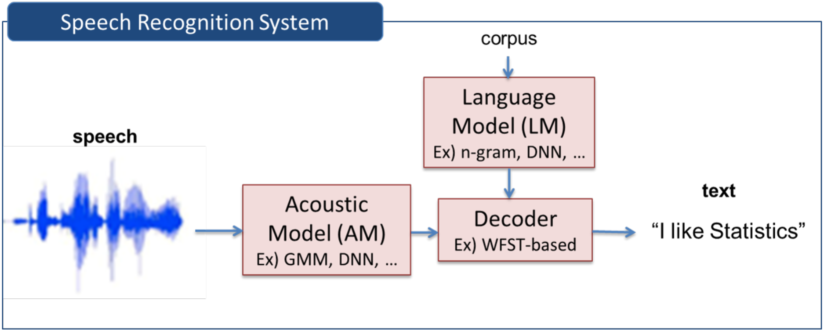


Outline

- Introduction
- Convolutional neural networks (CNN)
- Recurrent neural networks (RNN)
  - Architecure and properties
  - How to train
  - Vanishing/exploding gradients problem
  - Long short-term memory (LSTM)
  - Applications
- A statistical view of deep learning
- Open source tools

Applications of RNN (LSTM): speech recognition

- Three components:
  - Acoustic model (AM): estimate phoneme probability given input waveform
  - Language model (LM): estimate word probability given past word sequence
  - Decoder: combine AM+LM to estimate best sentence

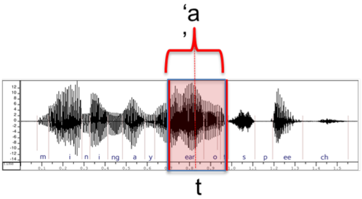


Applications of RNN (LSTM): acoustic model

- BLSTM takes entire speech for recognition at time t
  - Long-term memory can improve the accuracy

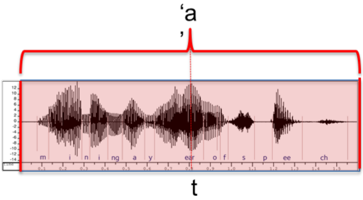
GMM / DNN

- Use finite-size window
  - model complexity:
    exponential in window size



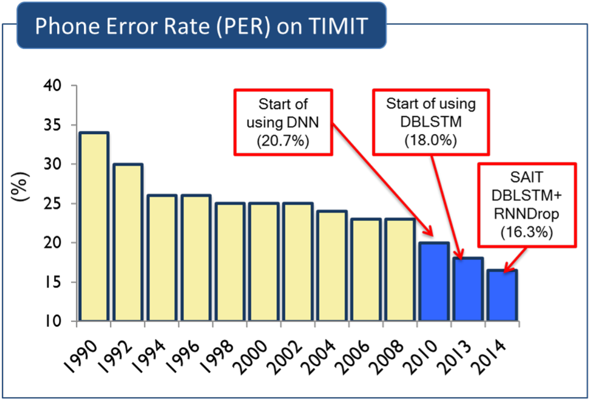
Bidirectional LSTM

- Use entire sequence
  - model complexity:
    fixed with sequence length



Applications of RNN (LSTM): acoustic model

- TIMIT: standard benchmark for phoneme recognition
  - 3.5 hours (small set)

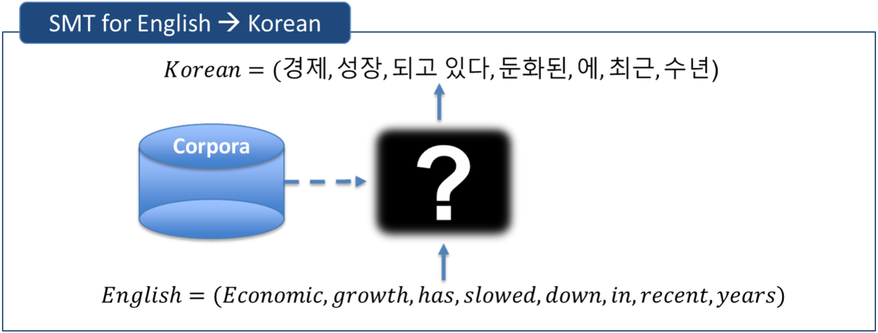


- Similar result in much larger set (>2000 hours) with large vocabulary as well
- LSTM-based LM also gives significant performance boost

[Graves et. al., Speech recognition with deep recurrent neural networks, 2013]
[Hannun et. al., Deep speech: Scaling up end-to-end speech recognition, 2014]

Applications of RNN (LSTM): machine translation

- Statistical machine translation (SMT)
  - Statistically estimates the target sentence from the source sentence
  - Challenge: word order difference, one-to-many
    - How to find (stochastic) mapping between sentences?



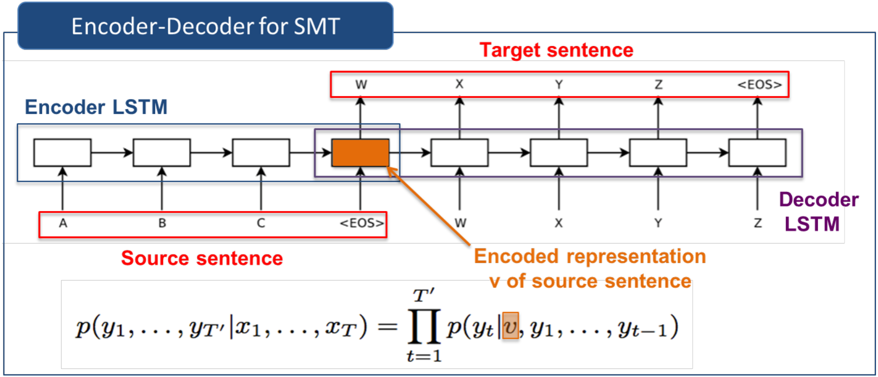
SMT for English → Korean

$Korean =$ (경제, 성장, 되고 있다, 둔화된, 에, 최근, 수 년)

$English = (Economic, growth, has, slowed, down, in, recent, years)$

Applications of RNN (LSTM): machine translation

- Neural machine translation (NMT): LSTM plays a central role
- Main idea: use Encoder-Decoder idea
  - Encoder: find a representation of source
  - Decoder: generate a translation with encoded representation

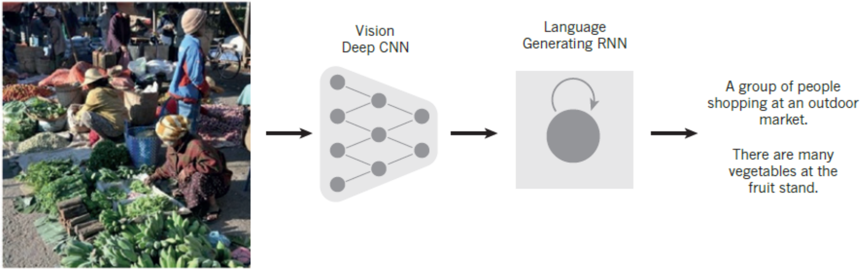


$$p(y_1, \ldots, y_{T'} | x_1, \ldots, x_T) = \prod_{t=1}^{T'} p(y_t | \boxed{v}, y_1, \ldots, y_{t-1})$$

[Cho et. al., Learning phrase representations using RNN encoder-decoder for statistical machine translation, 2014]
[Sutskever et. al., Sequence to sequence learning with neural networks, 2014]

Applications of CNN + RNN (LSTM): image captioning

- “Translate” image to text
  - Same principle as machine translation
  - Combine CNN (encoder) + RNN/LSTM (decoder)



A group of people shopping at an outdoor market.

There are many vegetables at the fruit stand.

©: [LeCun, Bengio, and Hinton, Deep learning, 2015]

Outline

- Introduction
- Convolutional neural networks (CNN)
- Recurrent neural networks (RNN)
- A statistical view of deep learning
    - Recursive GLMs
    - Kernel regression
    - RNNs as state-space models
- Open source tools

Outline

- Introduction
- Convolutional neural networks (CNN)
- Recurrent neural networks (RNN)
- A statistical view of deep learning
    - Recursive GLMs
    - Kernel regression
    - RNNs as state-space models
- Open source tools

Recursive GLMs

- Deep feedforward NNs can be seen as recursive generalized linear models
- Basic linear regression model
  - Assumes that the outputs are corrupted by Gaussian noise of unkown variance σ^2:

    $$\eta = \beta^T \mathbf{x} + \beta_0$$
    $$y = \eta + \epsilon$$

    where $\epsilon \sim N(0, \sigma^2)$
- Generalized linear model (GLM)
  - Extends LM to problems where the distribution of y is not Gaussian but some other distribution (typically a distribution in the exponential family):

    $$\eta = \beta^T \mathbf{x}$$
    $$\mathbb{E}[y] = \mu = g^{-1}(\eta)$$

    where $\beta := [\beta, \beta_0]$, $\mathbf{x} := [\mathbf{x}, 1]$ and $g(\cdot)$ is the link function

Recursive GLMs (cont'd)

- Activation function in NN = inverse link function in GLM
  - Recall what an activation function does: affine transform + nonlinearity

Target type	**Regression**	**Link**	**Inv. link**	**Activation**
Real	Linear	Identity	Identity	Identity
Binary	Logistic	Logit $\log \frac{\mu}{1-\mu}$	Sigmoid $\frac{1}{1+exp(-\eta)}$	Sigmod
Binary	Probit	Inv. Gaussian CDF $\Phi^{-1}(\mu)$	Gaussian CDF $\Phi(\eta)$	Probit
Binary	Logistic		$\tanh(\eta)$	tanh
Categorical	Multinomial		Multilogit $\frac{exp(\eta_i)}{\sum_j exp(\eta_j)}$	Softmax
Sparse	Tobit		$\max(0, \nu)$	ReLU

Recursive GLMs (cont'd)

- For an inverse link or activation function $f^{(\ell)}$ at layer ℓ, consider the following relationship:

$$\mathbf{h}^{(\ell)} \triangleq f^{(\ell)} \circ \eta^{(\ell)}$$

$$\eta^{(\ell)}(\mathbf{h}^{(\ell-1)}) \triangleq \langle \boldsymbol{\beta}^{(\ell)}, \mathbf{h}^{(\ell-1)} \rangle$$

  where $\mathbf{h}^{(0)} = \mathbf{x}$.

- Then we can easily specify a recursive GLM by iteratively applying or composing it:

$$\begin{aligned}\mathbb{E}[y|\mathbf{x}] &= \mathbf{h}^{(L)} \circ \mathbf{h}^{(L-1)} \circ \ldots \circ \mathbf{h}^{(1)}(\mathbf{x}) \\ &= \underbrace{f^{(L)} \circ \eta^{(L)} \circ f^{(L-1)} \circ \eta^{(L-1)} \circ \ldots \circ f^{(1)}}_{=g^{-1}} \circ \eta^{(1)}(\mathbf{x}) \\ &= g^{-1}(\eta^{(1)}(\mathbf{x})) = g^{-1}({\boldsymbol{\beta}^{(1)}}^{\top}\mathbf{x})\end{aligned}$$

  which is exactly the same as the L-layer deep feedforward NN.

Recursive GLMs (cont'd)

- Thus, a DNN can be viewed as a GLM whose inverse link function is recursively defined.
- Learning and estimation
  - Estimation or learning in deep neural networks corresponds directly to maximum likelihood estimation in recursive GLMs: $\mathcal{L} = -\log p(y|\mathbf{x})$

Outline

- Introduction
- Convolutional neural networks (CNN)
- Recurrent neural networks (RNN)
- A statistical view of deep learning
  - Recursive GLMs
  - Kernel regression
  - RNNs as state-space models
- Open source tools

Kernel regression

- To connect NNs to the linear model, let's separate the last linear layer from the layers that appear before it, i.e., denote the first $L-1$ layers by the mapping $\phi(\mathbf{x};\boldsymbol{\theta})$ with parameters $\boldsymbol{\theta}$, and the final layer weight $\mathbf{w}$:

$$\text{Sytematic:} \qquad f = \mathbf{w}^\top \phi(\mathbf{x};\boldsymbol{\theta})$$

$$\text{Random:} \qquad y = f(\mathbf{x}) + \epsilon \qquad \epsilon \sim \mathcal{N}(0, \sigma_y^2)$$

  - The loss function for the output weights is of particular interest, since it will offers us a way to move from neural networks to other types of regression

$$J(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{n}(y_i - \mathbf{w}^\top \phi(\mathbf{x}_i;\boldsymbol{\theta}))^2 + \frac{\lambda}{2}\mathbf{w}^\top\mathbf{w}$$

Kernel regression (cont'd)

- Using the fact $\frac{1}{2}\mathbf{w}^\top \mathbf{w} = \max_{\boldsymbol{\alpha}} \langle \mathbf{w}, \boldsymbol{\alpha} \rangle - \frac{1}{2}\boldsymbol{\alpha}^\top \boldsymbol{\alpha}$,

$$
\begin{aligned}
\min_{\mathbf{w}} J(\mathbf{w}) &= \frac{1}{2}\sum_{i=1}^{n}(y_i - \mathbf{w}^\top \phi(\mathbf{x}_i))^2 + \lambda \left(\max_{\boldsymbol{\alpha}} \langle \mathbf{w}, \boldsymbol{\alpha} \rangle - \frac{1}{2}\boldsymbol{\alpha}^\top \boldsymbol{\alpha} \right) \\
&= \min_{\mathbf{w}} \max_{\boldsymbol{\alpha}} \underbrace{\left(\frac{1}{2}\sum_{i=1}^{n}(y_i - \mathbf{w}^\top \phi(\mathbf{x}_i))^2 + \lambda \mathbf{w}^\top \boldsymbol{\alpha} - \frac{\lambda}{2}\boldsymbol{\alpha}^\top \boldsymbol{\alpha} \right)}_{\mathcal{L}(\mathbf{w},\boldsymbol{\alpha})} \\
&= \min_{\mathbf{w}} \max_{\boldsymbol{\alpha}} \mathcal{L}(\mathbf{w}, \boldsymbol{\alpha}) \\
&= \max_{\boldsymbol{\alpha}} \min_{\mathbf{w}} \mathcal{L}(\mathbf{w}, \boldsymbol{\alpha}) = \max_{\boldsymbol{\alpha}} g(\boldsymbol{\alpha})
\end{aligned}
$$

where $g(\boldsymbol{\alpha}) = \inf_{\mathbf{w}} \mathcal{L}(\mathbf{w}, \boldsymbol{\alpha}) = -\frac{\lambda}{2}\beta^\top (\mathbf{K} + \lambda \mathbf{I})\beta + \lambda \beta^\top \mathbf{y}$ with $\beta_i = \frac{1}{\lambda}(y_i - \mathbf{w}^\top \phi(\mathbf{x}_i))$ and $K_{ij} = \phi(\mathbf{x}_i)\phi(\mathbf{x}_j)$

Kernel regression (cont'd)

- Thus, the dual is

$$\max_{\beta} -\frac{1}{2}\beta^\top(\mathbf{K}+\lambda\mathbf{I})\beta + \beta^\top\mathbf{y}$$

  or $\hat{\beta} = (\mathbf{K}+\lambda\mathbf{I})^{-1}\mathbf{y}$
- DNNs: parametically estimate the function $\phi(\mathbf{x}_i)$
- Kernel machines: only consider inner products and choose a kernel function $k(\mathbf{x}, \mathbf{x}')$

Outline

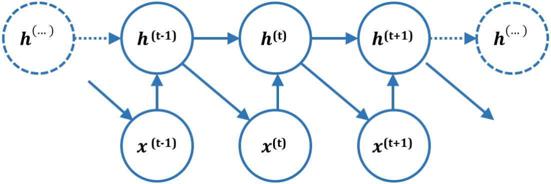
- Introduction
- Convolutional neural networks (CNN)
- Recurrent neural networks (RNN)
- A statistical view of deep learning
  - Recursive GLMs
  - Kernel regression
  - RNNs as state-space models
- Open source tools

RNNs as state-space models

- Let's consider the case of a RNN where inputs are a sequence of random variables $\{\mathbf{x}^{(t)}\}_{t=1}^{\tau}$ and no additional inputs
  - The input at time step t is simply the output at time step $t-1$

$$\mathbf{h}^{(t)} = f(\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}; \boldsymbol{\theta})$$

$$J(\boldsymbol{\theta}) = \sum_{t=1}^{\tau} \mathcal{L}^{(t)} = \sum_{t=1}^{\tau} \mathcal{L}(\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)})$$



RNNs as state-space models (cont'd)

- This RNN can be interpeted as a state-space model with a sequence of latent (or hidden) dynamics length τ
  - Latent states $\mathbf{h}^{(t)}$ and observed data $\mathbf{x}^{(t)}$ are assumed to be probabilistic
  - Transition probability is the same for all time (= parameters sharing in a RNN)
- In probabilistic modeling, the core quantity of interest is the joint distribution of the observed sequence $\mathbf{x}^{(t)}$, i.e.,

$$p(\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(\tau)}) = \prod_{t=1}^{\tau} \int p(\mathbf{x}^{(t)}|\mathbf{h}^{(t-1)}) p(\mathbf{h}^{(t-1)}|\mathbf{h}^{(t-2)}) \mathrm{d}\mathbf{h}^{(t-1)}$$

RNNs as state-space models (cont'd)

- If we use the negatvie log-likelihood loss,

$$J(\boldsymbol{\theta}) = \sum_{t=1}^{\tau} -\log \int p(\mathbf{x}^{(t)}|\mathbf{h}^{(t-1)})p(\mathbf{h}^{(t-1)}|\mathbf{h}^{(t-2)})\mathrm{d}\mathbf{h}^{(t-1)}$$
$$\overset{(*)}{=} \sum_{t=1}^{\tau} -\log p(\mathbf{x}^{(t)}|f(\mathbf{h}^{(t-2)}, \mathbf{x}^{(t-1)}; \boldsymbol{\theta}))$$

$(*)$ holds as the transition dynamics is deterministic, i.e.,

$$p(\mathbf{h}^{(t-1)}|\mathbf{h}^{(t-2)}; \boldsymbol{\theta}) = \delta(\mathbf{h}^{(t-1)}) = f(\mathbf{h}^{(t-2)}, \mathbf{x}^{(t-1)}; \boldsymbol{\theta}))$$

- This loss function is equivalent to that of the RNN, if we set

$$\mathcal{L}(\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}) = -\log p(\mathbf{x}^{(t)}|\mathbf{h}^{(t-1)} = f(\mathbf{h}^{(t-2)}, \mathbf{x}^{(t-1)}; \boldsymbol{\theta}))$$

Outline

- Introduction
- Convolutional neural networks (CNN)
- Recurrent neural networks (RNN)
- A statistical view of deep learning
- Open source tools

Open source tools

- Caffe
  - Maintained by UC Berkeley BLVC
  - http://caffe.berkeleyvision.org/
- Theano
  - Maintained by University of Montreal
  - Strong Python integration
  - http://deeplearning.net/software/theano/
- Torch
  - Maintained by NYU, Facebook
  - Based on Lua
  - http://torch.ch/
- Tensorflow
  - Maintained by Google (most recent)
  - https://www.tensorflow.org/

References

[1] Bengio, Yoshua, Pascal Lamblin, Dan Popovici, and Hugo Larochelle. "Greedy layer-wise training of deep networks." *Advances in neural information processing systems* 19 (2007): 153.
[2] Bengio, Yoshua, Patrice Simard, and Paolo Frasconi. "Learning long-term dependencies with gradient descent is difficult." *Neural Networks, IEEE Transactions on* 5, no. 2 (1994): 157-166.
[3] Chellapilla, Kumar, Sidd Puri, and Patrice Simard. "High performance convolutional neural networks for document processing." In *Tenth International Workshop on Frontiers in Handwriting Recognition*. Suvisoft, 2006.
[4] Cho, Kyunghyun, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. "Learning phrase representations using RNN encoder-decoder for statistical machine translation." *arXiv preprint arXiv:1406.1078* (2014).
[5] Ciresan, Dan Claudiu, Ueli Meier, Luca Maria Gambardella, and Jürgen Schmidhuber. "Deep, big, simple neural nets for handwritten digit recognition." *Neural computation* 22, no. 12 (2010): 3207-3220.
[6] Coates, Adam, Brody Huval, Tao Wang, David Wu, Bryan Catanzaro, and Ng Andrew. "Deep learning with COTS HPC systems." In *Proceedings of the 30th international conference on machine learning*, pp. 1337-1345. 2013.
[7] "Convolutional Neural Networks (LeNet)." DeepLearning 0.1 Documentation.
[8] "CS224d: Deep Learning for Natural Language Processing." Stanford University.
[9] "CS231n: Convolutional Neural Networks for Visual Recognition." Stanford University.
[10] Cybenko, George. "Approximation by superpositions of a sigmoidal function." *Mathematics of control, signals and systems* 2, no. 4 (1989): 303-314.
[11] "Deep learning, DC power grids, & BC-robots." Compute Scotland. April 25, 2013.

References (cont'd)

[12] Erhan, Dumitru, Yoshua Bengio, Aaron Courville, Pierre-Antoine Manzagol, Pascal Vincent, and Samy Bengio. "Why does unsupervised pre-training help deep learning?." *The Journal of Machine Learning Research* 11 (2010): 625-660.

[13] Fukushima, Kunihiko. "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position." *Biological cybernetics* 36, no. 4 (1980): 193-202.

[14] Glorot, Xavier, Antoine Bordes, and Yoshua Bengio. "Deep sparse rectifier neural networks." In *International Conference on Artificial Intelligence and Statistics*, pp. 315-323. 2011.

[15] Graves, Alan, Abdel-rahman Mohamed, and Geoffrey Hinton. "Speech recognition with deep recurrent neural networks." In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pp. 6645-6649. IEEE, 2013.

[16] Greff, Klaus, Rupesh Kumar Srivastava, Jan Koutník, Bas R. Steunebrink, and Jürgen Schmidhuber. "LSTM: A search space odyssey." *arXiv preprint arXiv:1503.04069* (2015).

[17] Hannun, Awni, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger et al. "Deep speech: Scaling up end-to-end speech recognition." *arXiv preprint arXiv:1412.5567* (2014).

[18] Hinton, Geoffrey E. "Learning distributed representations of concepts." In *Proceedings of the eighth annual conference of the cognitive science society*, vol. 1, p. 12. 1986.

[19] Hinton, Geoffrey E., and Ruslan R. Salakhutdinov. "Reducing the dimensionality of data with neural networks." *Science* 313, no. 5786 (2006): 504-507.

[20] Hochreiter, Sepp, and Jürgen Schmidhuber. "Long short-term memory." *Neural computation* 9, no. 8 (1997): 1735-1780.

References (cont'd)

[21] Hornik, Kurt, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators." *Neural networks* 2, no. 5 (1989): 359-366.

[22] Ian Goodfellow, Yoshua Bengio and Courville, A. 2016. Deep learning

[23] Ioffe, Sergey, and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." *arXiv preprint arXiv:1502.03167* (2015).

[24] Jaeger, Herbert, Mantas Lukoševicius, Dan Popovici, and Udo Siewert. "Optimization and applications of echo state networks with leaky-integrator neurons." *Neural Networks* 20, no. 3 (2007): 335-352.

[25] Jarrett, Kevin, Koray Kavukcuoglu, Marc'Aurelio Ranzato, and Yann LeCun. "What is the best multi-stage architecture for object recognition?." In *Computer Vision, 2009 IEEE 12th International Conference on*, pp. 2146-2153. IEEE, 2009.

[26] Kim, Hyung-Joong. "Lessons from the Lee Sedol vs. AlphaGo Match." Korea IT Times. March 14, 2016.

[27] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." In *Advances in neural information processing systems*, pp. 1097-1105. 2012.

[28] Le, Quoc V. "Building high-level features using large scale unsupervised learning." In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pp. 8595-8598. IEEE, 2013.

[29] Le, Quoc V., Navdeep Jaitly, and Geoffrey E. Hinton. "A simple way to initialize recurrent networks of rectified linear units." *arXiv preprint arXiv:1504.00941* (2015).

[30] Le, Quoc V., and Tomas Mikolov. "Distributed representations of sentences and documents." *arXiv preprint arXiv:1405.4053* (2014).

References (cont'd)

[31] LeCun, Yann, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, L. D. Jackel, and H. S. Baird. "Constrained neural network for unconstrained handwritten digit recognition." In *CENPARMI, Concordia University*. 1990.

[32] LeCun, Yann, Bernhard Boser, John S. Denker, Donnie Henderson, Richard E. Howard, Wayne Hubbard, and Lawrence D. Jackel. "Backpropagation applied to handwritten zip code recognition." *Neural computation* 1, no. 4 (1989): 541-551.

[33] LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton. "Deep learning." *Nature* 521, no. 7553 (2015): 436-444.

[34] Leshno, Moshe, Vladimir Ya Lin, Allan Pinkus, and Shimon Schocken. "Multilayer feedforward networks with a nonpolynomial activation function can approximate any function." *Neural networks* 6, no. 6 (1993): 861-867.

[35] Levine, Sergey, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. "End-to-end training of deep visuomotor policies." *arXiv preprint arXiv:1504.00702* (2015).

[36] Lukoševicius, Mantas. "A practical guide to applying echo state networks." In *Neural Networks: Tricks of the Trade*, pp. 659-686. Springer Berlin Heidelberg, 2012.

[37] Markoff, John. "How Many Computers to Identify a Cat? 16,000." The New York Times. June 25, 2012.

[38] Martens, James, and Ilya Sutskever. "Learning recurrent neural networks with hessian-free optimization." In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pp. 1033-1040. 2011.

[39] McCulloch, Warren S., and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity." *The bulletin of mathematical biophysics* 5, no. 4 (1943): 115-133.

[40] Minsky, Marvin, and Seymour Papert. "Perceptrons." (1969).

References (cont'd)

[41] Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves et al. "Human-level control through deep reinforcement learning." *Nature* 518, no. 7540 (2015): 529-533.
[42] Mohamed, Shakir. "A statistical View of Deep Learning: Retrospective." The Spectator, Shakir's Machine Learning Blog. Jul 4, 2015
[43] Olah, Christopher. "Calculus on Computational Graphs: Backpropagation." Colah's Blog. August 31, 2015.
[44] Pascanu, Razvan, Tomas Mikolov, and Yoshua Bengio. "On the difficulty of training recurrent neural networks." *arXiv preprint arXiv:1211.5063* (2012).
[45] Rosenblatt, Frank. "The perceptron: a probabilistic model for information storage and organization in the brain." *Psychological review* 65, no. 6 (1958): 386.
[46] Rumelhart, David E., Paul Smolensky, James L. McClelland, and G. Hinton. "Sequential thought processes in PDP models." *V* 2 (1986): 3-57.
[47] Salakhutdinov, Ruslan, and Geoffrey E. Hinton. "Deep boltzmann machines." In *International conference on artificial intelligence and statistics*, pp. 448-455. 2009.
[48] Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." *arXiv preprint arXiv:1409.1556* (2014).
[49] Sutskever, Ilya, James Martens, George Dahl, and Geoffrey Hinton. "On the importance of initialization and momentum in deep learning." In *Proceedings of the 30th international conference on machine learning (ICML-13)*, pp. 1139-1147. 2013.
[50] Sutskever, Ilya, Oriol Vinyals, and Quoc V. Le. "Sequence to sequence learning with neural networks." In *Advances in neural information processing systems*, pp. 3104-3112. 2014.

References (cont'd)

[51] Szegedy, Christian, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. "Going deeper with convolutions." In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1-9. 2015.

[52] Szegedy, Christian. "Building a deeper understanding of images." Google Research Blog. September 05, 2014.

[53] Widrow, Bernard and Hoff, Marcian E. "Adaptive switching circuits." (1960): 96-104.

[54] Zeiler, Matthew D., and Rob Fergus. "Visualizing and understanding convolutional networks." In *European Conference on Computer Vision*, pp. 818-833. Springer International Publishing, 2014.