CENG 783

Deep Learning

Week – 12
NTM, AEs

Sinan Kalkan
Today

- Neural Turing Machines
- Auto-encoders

NOTES:
- Final Exam date: 31 May, 17:00.
- HW3 announced. Due date: 24 May, 23:55.
- Project demos and papers due: 6 June.
- Extra lecture: 29 May 9:40-12:30
Neural Turing Machines
Why need other mechanisms?

• We mentioned before that RNNs are Turing Complete, right?

• The issues are:
  • The vanishing/exploding gradients (LSTM and other tricks address these issues)
  • However, parameters explode in LSTMs with the number of layers (and stacks)
  • The answer to addressing bigger networks with less parameters is a better abstraction of the computational components, e.g., in a form similar to Turing machines

Weston et al., 2015
Turing Machine

Wikipedia:

Following Hopcroft and Ullman (1979, p. 148), a (one-tape) Turing machine can be formally defined as a 7-tuple $M = (Q, \Gamma, b, \Sigma, \delta, q_0, F)$ where

- $Q$ is a finite, non-empty set of states
- $\Gamma$ is a finite, non-empty set of tape alphabet symbols
- $b \in \Gamma$ is the blank symbol (the only symbol allowed to occur on the tape infinitely often at any step during the computation)
- $\Sigma \subseteq \Gamma \setminus \{b\}$ is the set of input symbols
- $\delta : (Q \setminus F) \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ is a partial function called the transition function, where L is left shift, R is right shift. (A relatively uncommon variant allows "no shift", say N, as a third element of the latter set.) If $\delta$ is not defined on the current state and the current tape symbol, then the machine halts.\[21\]
- $q_0 \in Q$ is the initial state
- $F \subseteq Q$ is the set of final or accepting states. The initial tape contents is said to be accepted by $M$ if it eventually halts in a state from $F$.

Anything that operates according to these specifications is a Turing machine.
Neural Turing Machines

- If we make every component differentiable, we can train such a complex machine

- Accessing only a part of the network is problematic
  - Unlike a computer (TM), we need a ‘blurry’ access mechanism

Figure 1: Neural Turing Machine Architecture. During each update cycle, the controller network receives inputs from an external environment and emits outputs in response. It also reads from and writes to a memory matrix via a set of parallel read and write heads. The dashed line indicates the division between the NTM circuit and the outside world.

Neural Turing Machines

Alex Graves  gravesa@google.com
Greg Wayne  gregwayne@google.com
Ivo Danihelka  danielka@google.com

Neural Turing Machines: Reading

- Let memory $\mathbf{M}$ be an $N \times M$ matrix
  - $N$: the number of “rows”
  - $M$: the size of each row (vector)
- Let $\mathbf{M}_t$ be the memory state at time $t$
- $w_t$: a vector of weightings over $N$ locations emitted by the read head at time $t$. Since the weights are normalized:
  $$\sum_i w_t(i) = 1, \quad 0 \leq w_t(i) \leq 1, \forall i$$
- $\mathbf{r}_t$: the read vector of length $M$:
  $$\mathbf{r}_t \leftarrow \sum_i w_t(i) \mathbf{M}_t(i).$$
- which is differentiable, and therefore, trainable.
Neural Turing Machines: Writing

- Writing = erasing content + adding new content
  - Inspired from LSTM’s forgetting and addition gates.

- Erasing: Multiply with an erase vector $e_t \in [0,1]^M$
  $$ \hat{M}_t(i) \leftarrow M_{t-1}(i)[1 - w_t(i)e_t] $$
  $1$: vector of ones. Multiplication here is pointwise.

- Adding: Add an add vector $a_t \in [0,1]^M$:
  $$ M_t(i) \leftarrow \hat{M}_t(i) + w_t(i)a_t $$

Figure 1: Neural Turing Machine Architecture. During each update cycle, the controller network receives inputs from an external environment and emits outputs in response. It also reads to and writes from a memory matrix via a set of parallel read and write heads. The dashed line indicates the division between the NTM circuit and the outside world.
Neural Turing Machines: **Addressing**

- Content-based addressing
- Location-based addressing
  - In a sense, use variable “names” to access content

---

**Figure 1:** Neural Turing Machine Architecture. During each update cycle, the controller network receives inputs from an external environment and emits outputs in response. It also reads to and writes from a memory matrix via a set of parallel read and write heads. The dashed line indicates the division between the NTM circuit and the outside world.

**Figure 2:** Flow Diagram of the Addressing Mechanism. The key vector, $k_t$, and key strength, $\beta_t$, are used to perform content-based addressing of the memory matrix, $M_t$. The resulting content-based weighting is interpolated with the weighting from the previous time step based on the value of the interpolation gate, $g_t$. The shift weighting, $s_t$, determines whether and by how much the weighting is rotated. Finally, depending on $\gamma_t$, the weighting is sharpened and used for memory access.
Neural Turing Machines: Content-based Addressing

• Each head (reading or writing head) produces an $M$ length key vector $\mathbf{k}_t$
  • $\mathbf{k}_t$ is compared to each vector $\mathbf{M}_t(i)$ using a similarity measure $K[\cdot,\cdot]$, e.g., cosine similarity:
    $$K[\mathbf{u},\mathbf{v}] = \frac{\mathbf{u} \cdot \mathbf{v}}{||\mathbf{u}|| \cdot ||\mathbf{v}||}$$

• From these similarity measures, we obtain a vector of “addressing”:
  $$w_t^c(i) \leftarrow \frac{\exp(\beta_t K[\mathbf{k}_t, \mathbf{M}_t(i)])}{\sum_j \exp(\beta_t K[\mathbf{k}_t, \mathbf{M}_t(j)])}$$
  • $\beta_t$: amplifies or attenuates the precision of the focus
Neural Turing Machines: Location-based Addressing

• Important for e.g. iteration over memory locations, or jumping to an arbitrary memory location

• First: Interpolation between addressing schemes using “interpolation gate” $g_t$:

$$w_t^g \leftarrow g_t w_t^c + (1 - g_t)w_{t-1}$$

  - If $g_t = 1$: weight from content-addressable component is used
  - If $g_t = 0$: weight from previous step is used

• Second: rotationally shift weight to achieve location-based addressing using convolution:

$$\hat{w}_t(i) \leftarrow \sum_{j=0}^{N-1} w_t^g(j)s_t(i - j)$$

  - $s_t$: shift amount. Three elements for how “much” to shift left, right or keep as it is.
  - It needs to be “sharp”. To keep it sharp, each head emits a scalar $\gamma_t \geq 1$:

$$w_t(i) \leftarrow \frac{\hat{w}_t(i)^{\gamma_t}}{\sum_j \hat{w}_t(j)^{\gamma_t}}$$
Neural Turing Machines: Controller Network

- Free parameters
  - The size of the memory
  - Number of read-write heads
  - Range of allowed rotation shifts
  - Type of the neural network for controller

- Alternatives:
  - A recurrent network such as LSTM with its own memory
    - These memory units might be considered like “registers” on the CPU
  - A feed-forward network
    - Can use the memory to achieve recurrence
    - More transparent
Neural Turing Machines: Training

- Binary targets
  - Logistic sigmoid output layers
  - Cross-entropy loss
- Other schemes possible
- Tasks:
  - Copy from input to output
  - Repeat Copy: Make n copies of the input
  - Associative recall: Present a part of a sequence to recall the remaining part
  - N-gram: Learn distribution of 6-grams and make predictions for the next bit based on this distribution
  - Priority sort: Associate a priority as part of each vector and as the target place the sequence according to the priority

<table>
<thead>
<tr>
<th>Task</th>
<th>#Heads</th>
<th>Controller Size</th>
<th>Memory Size</th>
<th>Learning Rate</th>
<th>#Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy</td>
<td>1</td>
<td>100</td>
<td>$128 \times 20$</td>
<td>$10^{-4}$</td>
<td>17,162</td>
</tr>
<tr>
<td>Repeat Copy</td>
<td>1</td>
<td>100</td>
<td>$128 \times 20$</td>
<td>$10^{-4}$</td>
<td>16,712</td>
</tr>
<tr>
<td>Associative</td>
<td>4</td>
<td>256</td>
<td>$128 \times 20$</td>
<td>$10^{-4}$</td>
<td>146,845</td>
</tr>
<tr>
<td>N-Grams</td>
<td>1</td>
<td>100</td>
<td>$128 \times 20$</td>
<td>$3 \times 10^{-5}$</td>
<td>14,656</td>
</tr>
<tr>
<td>Priority Sort</td>
<td>8</td>
<td>512</td>
<td>$128 \times 20$</td>
<td>$3 \times 10^{-5}$</td>
<td>508,305</td>
</tr>
</tbody>
</table>

Table 1: NTM with Feedforward Controller Experimental Settings

<table>
<thead>
<tr>
<th>Task</th>
<th>#Heads</th>
<th>Controller Size</th>
<th>Memory Size</th>
<th>Learning Rate</th>
<th>#Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy</td>
<td>1</td>
<td>100</td>
<td>$128 \times 20$</td>
<td>$10^{-4}$</td>
<td>67,561</td>
</tr>
<tr>
<td>Repeat Copy</td>
<td>1</td>
<td>100</td>
<td>$128 \times 20$</td>
<td>$10^{-4}$</td>
<td>66,111</td>
</tr>
<tr>
<td>Associative</td>
<td>1</td>
<td>100</td>
<td>$128 \times 20$</td>
<td>$10^{-4}$</td>
<td>70,330</td>
</tr>
<tr>
<td>N-Grams</td>
<td>1</td>
<td>100</td>
<td>$128 \times 20$</td>
<td>$3 \times 10^{-5}$</td>
<td>61,749</td>
</tr>
<tr>
<td>Priority Sort</td>
<td>5</td>
<td>$2 \times 100$</td>
<td>$128 \times 20$</td>
<td>$3 \times 10^{-5}$</td>
<td>269,038</td>
</tr>
</tbody>
</table>

Table 2: NTM with LSTM Controller Experimental Settings

<table>
<thead>
<tr>
<th>Task</th>
<th>Network Size</th>
<th>Learning Rate</th>
<th>#Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy</td>
<td>$3 \times 256$</td>
<td>$3 \times 10^{-5}$</td>
<td>1,352,969</td>
</tr>
<tr>
<td>Repeat Copy</td>
<td>$3 \times 512$</td>
<td>$3 \times 10^{-5}$</td>
<td>5,312,007</td>
</tr>
<tr>
<td>Associative</td>
<td>$3 \times 256$</td>
<td>$10^{-4}$</td>
<td>1,344,518</td>
</tr>
<tr>
<td>N-Grams</td>
<td>$3 \times 128$</td>
<td>$10^{-4}$</td>
<td>331,905</td>
</tr>
<tr>
<td>Priority Sort</td>
<td>$3 \times 128$</td>
<td>$3 \times 10^{-5}$</td>
<td>384,424</td>
</tr>
</tbody>
</table>

Table 3: LSTM Network Experimental Settings
Neural Turing Machines: Training

Figure 3: Copy Learning Curves.

Figure 7: Repeat Copy Learning Curves.

Figure 10: Associative Recall Learning Curves for NTM and LSTM.

Figure 18: Priority Sort Learning Curves.
Reinforced version

Under review as a conference paper at ICLR 2016

REINFORCEMENT LEARNING
NEURAL TURING MACHINES - REVISED

Wojciech Zaremba\textsuperscript{1,2}
New York University
Facebook AI Research
woj.zaremba@gmail.com

Ilya Sutskever\textsuperscript{2}
Google Brain
ilyasu@google.com
Other variants/ attempts
Figure 2: One timestep of the NRAM architecture with $R = 4$ registers. The LSTM controller gets the "binarized" values $r_1, r_2, \ldots$ stored in the registers as inputs and outputs the description of the circuit in the grey box and the probability of finishing the execution in the current timestep (See Sec. 3.3 for more detail). The weights of the solid thin connections are outputted by the controller. The weights of the solid thick connections are trainable parameters of the model. Some of the modules (i.e. READ and WRITE) may interact with the memory tape (dashed connections).

Published as a conference paper at ICLR 2016

Neural Random-Access Machines

Karol Kurach* & Marcin Andrychowicz* & Ilya Sutskever
Google
{kkurach, marcina, ilyasu}@google.com
Neural Programmer

Published as a conference paper at ICLR 2016

NEURAL PROGRAMMER: INDUCING LATENT PROGRAMS WITH GRADIENT DESCENT

Arvind Neelakantan*
University of Massachusetts Amherst
arvind@cs.umass.edu

Quoc V. Le
Google Brain
qvl@google.com

Ilya Sutskever
Google Brain
ilyasu@google.com
Figure 3: Examples of our model on Convex hulls (left), Delaunay (center) and TSP (right), trained on $m$ points, and tested on $n$ points. A failure of the LSTM sequence-to-sequence model for Convex hulls is shown in (a). Note that the baselines cannot be applied to a different length from training.
Bilbo travelled to the cave. Gollum dropped the ring there. Bilbo took the ring.
Bilbo went back to the Shire. Bilbo left the ring there. Frodo got the ring.
Frodo journeyed to Mount-Doom. Frodo dropped the ring there. Sauron died.
Frodo went back to the Shire. Bilbo travelled to the Grey-havens. The End.
Where is the ring? **A: Mount-Doom**
Where is Bilbo now? **A: Grey-havens**
Where is Frodo now? **A: Shire**

Published as a conference paper at ICLR 2015

**MEMORY NETWORKS**

Jason Weston, Sumit Chopra & Antoine Bordes
Facebook AI Research
770 Broadway
New York, USA
{jase,spchopra,abordes}@fb.com
Universal Turing Machine
Inferring and Executing Programs for Visual Reasoning

Justin Johnson\textsuperscript{1}  Bharath Hariharan\textsuperscript{2}  Laurens van der Maaten\textsuperscript{2}
Judy Hoffman\textsuperscript{1}  Li Fei-Fei\textsuperscript{1}  C. Lawrence Zitnick\textsuperscript{2}  Ross Girshick\textsuperscript{2}

\textsuperscript{1}Stanford University  \textsuperscript{2}Facebook AI Research

2017

How many chairs are at the table?  How many chairs are at the table?
Is there a pedestrian in my lane?  Is there a pedestrian in my lane?

Is the person with the blue hat touching the bike in the back?  Is there a matte cube that has the same size as the red metal object?

**Question:** Are there more cubes than yellow things?  **Answer:** Yes

![Program Generator Diagram](image)

> **Figure 2.** System overview. The program generator is a sequence-to-sequence model which inputs the question as a sequence of words and outputs a program as a sequence of functions, where the sequence is interpreted as a prefix traversal of the program's abstract syntax tree. The execution engine executes the program on the image by assembling a neural module network [2] mirroring the structure of the predicted program.
Newer studies

- https://deepmind.com/blog/differentiable-neural-computers/
- Differentiable Neural Machines
Unsupervised pre-training with Auto-encoders
Now

- Manifold Learning
  - Principle Component Analysis
  - Independent Component Analysis
- Autoencoders
- Sparse autoencoders
- K-sparse autoencoders
- Denoising autoencoders
- Contraction autoencoders
Manifold Learning
Manifold Learning

• Discovering the “hidden” structure in the high-dimensional space

• Manifold: “hidden” structure.

• Non-linear dimensionality reduction

http://www.convexoptimization.com/dattorro/manifold_learning.html
Manifold Learning

• Many approaches:
  • Self-Organizing Map (Kohonen map/network)
  • Auto-encoders
  • Principles curves & manifolds: Extension of PCA
  • Kernel PCA, Nonlinear PCA
  • Curvilinear Component Analysis
  • Isomap: Floyd-Marshall + Multidimensional scaling
  • Data-driven high-dimensional scaling
  • Locally-linear embedding
  • ...
Manifold learning

- Autoencoders learn lower-dimensional manifolds embedded in higher-dimensional manifolds

- Assumption: “Natural data in high dimensional spaces concentrates close to lower dimensional manifolds”
  - Natural images occupy a very small fraction in a space of possible images

(Pascal Vincent)
Manifold Learning

- Many approaches:
  - Self-Organizing Map (Kohonen map/network)

**Algorithm**

1. Randomize the map's nodes' weight vectors
2. Grab an input vector $D(t)$
3. Traverse each node in the map
   1. Use the Euclidean distance formula to find the similarity between the input vector and the map's node's weight vector
   2. Track the node that produces the smallest distance (this node is the best matching unit, BMU)
4. Update the nodes in the neighborhood of the BMU (including the BMU itself) by pulling them closer to the input vector
   1. $W_v(s + 1) = W_v(s) + \Theta(u, v, s) \alpha(s)(D(t) - W_v(s))$
5. Increase $s$ and repeat from step 2 while $s < \lambda$

A variant algorithm:

1. Randomize the map's nodes' weight vectors
2. Traverse each input vector in the input data set
   1. Traverse each node in the map
      1. Use the Euclidean distance formula to find the similarity between the input vector and the map's node's weight vector
      2. Track the node that produces the smallest distance (this node is the best matching unit, BMU)
   2. Update the nodes in the neighborhood of the BMU (including the BMU itself) by pulling them closer to the input vector
      1. $W_v(s + 1) = W_v(s) + \Theta(u, v, s) \alpha(s)(D(t) - W_v(s))$
3. Increase $s$ and repeat from step 2 while $s < \lambda$

Non-parametric density estimation

\[ \hat{p}(x) = \frac{1}{n} \sum_{i=1}^{n} N(x; x_i, C_i) \]

**Classical Parzen Windows**  
Density estimator

- Archetypal «non-parametric» kernel density estimator
- Isotropic Gaussian centered on each training point
- No sense of manifold direction
- Probability mass allocated away from manifold

**Manifold Parzen Windows**  
Density estimator  
(Vincent and Bengio, NIPS 2003)

- Oriented Gaussian «pancake» centered on each training point
- Uses low-rank parametrization of \( C_i \) learned from nearest neighbors (local PCA)
- «Parametric» cousins:  
Mixtures of Gaussian pancakes (Hinton et al. 95)  
Mixtures of Factor Analysers (Ghahramani + Hinton 96)  
Mixtures of Probabilistic PCA (Tipping + Bishop 99)
Non-local manifold Parzen windows
(Bengio, Larochelle, Vincent, NIPS 2006)

**Isotropic Parzen:**
\[
\hat{p}(x) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{N}(x; x_i, \sigma^2 I)
\]

**Manifold Parzen:**
(Vincent and Bengio, NIPS 2003)
\[
\hat{p}(x) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{N}(x; x_i, C_i)
\]
d\_m high variance directions from PCA on k nearest neighbors

**Non-local manifold Parzen:**
(Bengio, Larochelle, Vincent, NIPS 2006)
\[
\hat{p}(x) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{N}(x; \mu(x_i), C(x_i))
\]
d\_m high variance directions output by neural network
trained to maximize likelihood of k nearest neighbors
Principle Component Analysis (PCA)

• Principle Components:
  • Orthogonal directions with most variance
  • Eigen-vectors of the co-variance matrix

• Mathematical background:
  • Orthogonality:
    • Two vectors $\vec{u}$ and $\vec{v}$ are orthogonal iff $\vec{u} \cdot \vec{v} = 0$
  • Variance:
    $$\sigma(X)^2 = Var(X) = E[(X - \mu)^2] = \sum_i p(x_i)(x_i - \mu)^2$$
    where the (weighted) mean, $\mu = E[X] = \sum_i p(x_i)x_i$.
    
    If $p(x_i) = 1/N$:
    $$Var(X) = \frac{1}{N} \sum_i (x_i - \mu)^2$$
    $$\mu = \frac{1}{N} \sum_i x_i$$
Mathematical background for PCA: Covariance

• Co-variance:

  • Measures how two random variables change wrt each other:

  \[
  Cov(X, Y) = E[(X - E[X])(Y - E[Y])] \\
  = \frac{1}{N} \sum_{i} (x_i - E[X])(y_i - E[Y])
  \]

  • If big values of X & big values of Y “co-occur” and small values of X & small values of Y “co-occur” \(\Rightarrow\) high co-variance.
  • Otherwise, small co-variance.
Mathematical background for PCA: Covariance Matrix

- Co-variance Matrix:
  - Denoted usually by $\Sigma$
  - For an $n$-dimensional space:

  $$\Sigma_{ij} = \text{Cov}(X_i, X_j) = E[(X_i - \mu_i)(X_j - m_j)]$$

$$\Sigma = \begin{bmatrix}
E[(X_1 - \mu_1)(X_1 - \mu_1)] & E[(X_1 - \mu_1)(X_2 - \mu_2)] & \cdots & E[(X_1 - \mu_1)(X_n - \mu_n)] \\
E[(X_2 - \mu_2)(X_1 - \mu_1)] & E[(X_2 - \mu_2)(X_2 - \mu_2)] & \cdots & E[(X_2 - \mu_2)(X_n - \mu_n)] \\
\vdots & \vdots & \ddots & \vdots \\
E[(X_n - \mu_n)(X_1 - \mu_1)] & E[(X_n - \mu_n)(X_2 - \mu_2)] & \cdots & E[(X_n - \mu_n)(X_n - \mu_n)]
\end{bmatrix}.$$
Mathematical background for PCA: Covariance Matrix

- Co-variance Matrix:
  \[ \Sigma_{ij} = \text{Cov}(X_i, X_j) = E[(X_i - \mu_i)(X_j - m_j)] \]

- Properties

1. \( \Sigma = E(XX^T) - \mu \mu^T \)
2. \( \Sigma \) is positive-semidefinite and symmetric.
3. \( \text{cov}(AX + a) = A \text{cov}(X) A^T \)
4. \( \text{cov}(X, Y) = \text{cov}(Y, X)^T \)
5. \( \text{cov}(X_1 + X_2, Y) = \text{cov}(X_1, Y) + \text{cov}(X_2, Y) \)
6. If \( \rho = q \), then \( \text{var}(X + Y) = \text{var}(X) \text{cov}(X, Y) + \text{cov}(Y, X) + \text{var}(Y) \)
7. \( \text{cov}(AX + a, B^TY + b) = A \text{cov}(X, Y) B \)
8. If \( X \) and \( Y \) are independent or uncorrelated, then \( \text{cov}(X, Y) = 0 \)

(Wikipedia)

M is called positive-semidefinite (or sometimes nonnegative-definite) if
\[ x^* M x \geq 0 \]
for all \( x \) in \( \mathbb{C}^n \) (or, all \( x \) in \( \mathbb{R}^n \) for the real matrix).

(Wikipedia)
Mathematical background for PCA: Eigenvectors & Eigenvalues

- **Eigenvectors and eigenvalues:**
  - \( \mathbf{v} \) is an eigenvector of a square matrix \( A \) if
    \[
    A\mathbf{v} = \lambda \mathbf{v}
    \]
    where \( \lambda \) is the eigenvalue (scalar) associated with \( \mathbf{v} \).

- **Interpretation:**
  - “Transformation” \( A \) does not change the direction of the vector.
  - It changes the vector’s scale, i.e., the eigenvalue.

- **Solution:**
  - \( (A - \lambda I)\mathbf{v} = 0 \) has a solution when the determinant \( |A - \lambda I| \) is zero.
  - Find the eigenvalues, then plug in those values to get the eigenvectors.
Mathematical background for PCA: Eigenvectors & Eigenvalues Example

\[ A = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}. \]

- Setting the determinant \(|A - \lambda I|\) to zero:
  \[ p(\lambda) = |A - \lambda I| = 3 - 4\lambda + \lambda^2 = 0, \]

- The roots: \(\lambda = 1\) and \(\lambda = 3\)

- If you plug in those eigenvalues, for \(\lambda = 1\):
  \[ (A - I)v = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}\begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \]
  which gives \(v_1 = \{1, -1\}\).

  For \(\lambda = 3\):
  \[ (A - 3I)w = \begin{bmatrix} -1 & 1 \\ 1 & -1 \end{bmatrix}\begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \]
  which gives \(v_2 = \{1,1\}\).
PCA allows also dimensionality reduction

- Discard components whose eigenvalue is negligible.

See the following tutorial for more on PCA:
Independent Component Analysis (ICA)

- **PCA assumes Gaussianity:**
  - Data along a component should be explainable by a mean and a variance.
  - This may be violated by real signals in the nature.

- **ICA:**
  - Blind-source separation of non-Gaussian and mutually-independent signals.

- **Mutual independence:**

https://cnx.org/contents/-ElGohfq@1.4:gFEtO206@1/Independent-Component-Analysis
Autoencoders
Autoencoders

- Universal approximators
  - So are Restricted Boltzmann Machines
- Unsupervised learning
- Dimensionality reduction
- $\mathbf{x} \in \mathbb{R}^D \Rightarrow \mathbf{h} \in \mathbb{R}^M \text{ s.t. } M < D$
Autoencoders: MLPs used for «unsupervised» representation learning

- Make output layer same size as input layer
- Have target = input
- Loss encourages output (reconstruction) to be close to input.

Autoencoders are also called
- Autoencoders
- Auto-associators
- Diabolo networks
- Sandglass-shaped net

\[ L(x, r) \]

The Diabolo
Auto-Encoders (AE) for learning representations

hidden representation \( h = h(x) \) ∈ \( \mathbb{R}^{d_h} \)

Encoder: \( h \)  
Decoder: \( g \)

input \( x \in \mathbb{R}^{d} \)
reconstruction \( r = g(h(x)) \)

reconstruction error \( L(x, r) \)

Minimize \( \mathcal{J}_{AE} = \sum_{x \in D} L(x, g(h(x))) \)
Auto-Encoders (AE) for learning representations

Typical form

hidden representation \( h = h(x) = s(Wx + b) \) \( \in \mathbb{R}^{d_h} \)

Encoder: \( h \)

Decoder: \( g \)

input \( x \in \mathbb{R}^d \)

reconstruction \( r = g(h(x)) \)

reconstruction error \( L(x, r) \)

squared error: \( \|x - r\|^2 \)
or Bernoulli cross-entropy

Minimize

\[ J_{AE} = \sum_{x \in D} L(x, g(h(x))) \]
connection between
Linear auto-encoders and PCA

\( d_h < d \) (bottleneck, undercomplete representation):

- With linear neurons and squared loss,
  autoencoder learns same subspace as PCA

- Also true with a single sigmoidal hidden layer,
  if using linear output neurons with squared loss
  [Baldi & Hornik 89] and untied weights.

- Won’t learn the exact same basis as PCA,
  but \( W \) will span the same subspace.
Greedy Layer-Wise Pre-training with Auto-Encoders

Stacking basic Auto-Encoders [Bengio et al. 2007]
Stacking autoencoders: learn the first layer

http://ufldl.stanford.edu/wiki/index.php/Stacked_Autoencoders
Stacking autoencoders: learn the second layer

http://ufldl.stanford.edu/wiki/index.php/Stacked_Autoencoders
Stacking autoencoders:
Add, e.g., a softmax layer for mapping to output

http://ufldl.stanford.edu/wiki/index.php/Stacked_Autoencoders
Stacking autoencoders: Overall

http://ufldl.stanford.edu/wiki/index.php/Stacked_Autoencoders
Supervised fine-tuning

- Initial deep mapping was learnt in an *unsupervised* way.
- → initialization for a *supervised* task.
- Output layer gets added.
- Global fine tuning by gradient descent on *supervised criterion*.
Supervised Fine-Tuning is Important

- Greedy layer-wise unsupervised pre-training phase with RBMs or auto-encoders on MNIST
- Supervised phase with or without unsupervised updates, with or without fine-tuning of hidden layers

Classification performance on benchmarks:

- Pre-training basic auto-encoder stack better than no pre-training
- Basic auto-encoder stack almost matched RBM stack...
Basic auto-encoders not as good feature learners as RBMs...

What’s the problem?

✦ Traditional autoencoders were for **dimensionality reduction** ($d_h < d_x$)

✦ Deep learning success seems to depend on ability to learn **overcomplete representations** ($d_h > d_x$)

✦ Overcomplete basic autoencoder yields trivial useless solutions: identity mapping!

✦ Need for alternative **regularization/constraining**
Making auto-encoders learn over-complete representations that are not one-to-one mappings
Wait, what do we mean by over-complete?

- Remember distributed representations?

---

**Not distributed**

<table>
<thead>
<tr>
<th></th>
<th>(a)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>no pattern</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Distributed**

<table>
<thead>
<tr>
<th></th>
<th>(b)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>no pattern</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

Figure Credit: Moontae Lee
Distributed vs. undercomplete vs. overcomplete representations

- Four categories could also be represented by two neurons:
Over-complete = sparse (in distributed representations)

• Why sparsity?

1. Because our brain relies on sparse coding.
   • Why does it do so?
     a. Because it is adapted to an environment which is composed of and can be sensed through the combination of primitive items/entities.
     b. “Sparse coding may be a general strategy of neural systems to augment memory capacity. To adapt to their environments, animals must learn which stimuli are associated with rewards or punishments and distinguish these reinforced stimuli from similar but irrelevant ones. Such task requires implementing stimulus-specific associative memories in which only a few neurons out of a population respond to any given stimulus and each neuron responds to only a few stimuli out of all possible stimuli.”
        – Wikipedia
     c. Theoretically, it has shown that it increases capacity of memory.
Over-complete = sparse (in distributed representations)

• Why sparsity?

2. Because of information theoretical aspects:
   • Sparse codes have lower entropy compared to non-sparse ones.

3. It is easier for the consecutive layers to learn from sparse codes, compared to non-sparse ones.

FIGURE 7. The set of 144 basis functions learned by the sparse coding algorithm. The basis functions are totally overlapping (i.e., the entire set codes for the same image patch). All have been normalized to fill the grey scale, but with zero always represented by the same grey level.
Mechanisms for enforcing over-completeness

- Use stochastic gradient descent
- Add sparsity constraint
  - Into the loss function (sparse autoencoder)
  - Or, in a hard manner (k-sparse autoencoder)
- Add stochasticisity / randomness
  - Add noise: Denoising Autoencoders, Contraction Autoencoders
  - Restricted Boltzmann Machines

Why Regularized Auto-Encoders learn Sparse Representation?
Auto-encoders with SGD
Simple neural network

• Input: $\mathbf{x} \in \mathbb{R}^n$

• Hidden layer: $\mathbf{h} \in \mathbb{R}^m$
  \[ h = f_1(W_1 \mathbf{x}) \]

• Output layer: $\mathbf{y} \in \mathbb{R}^n$
  \[ y = f_2(W_2 f_1(W_1 \mathbf{x})) \]

• Squared-error loss:
  \[ L = \frac{1}{2} \sum_{d \in D} \| \mathbf{x}_d - \mathbf{y}_d \|^2 \]

• For training, use SGD.

• You may try different activation functions for $f_1$ and $f_2$. 
Sparse Autoencoders
Sparse autoencoders

- Input: \( x \in \mathbb{R}^n \)
- Hidden layer: \( h \in \mathbb{R}^m \)
  \[
  h = f_1(W_1x)
  \]
- Output layer: \( y \in \mathbb{R}^n \)
  \[
  y = f_2(W_2f_1(W_1x))
  \]

Over-completeness and sparsity:

- Require
  - \( m > n \), and
  - Hidden neurons to produce only little activation for any input \( \Rightarrow \) i.e., sparsity.

- How to enforce sparsity?
Enforcing sparsity: alternatives

• How?

• Solution 1: $\lambda |w|
  • We have seen before that this enforces sparsity.
  • However, this is not strong enough.

• Solution 2
  • Limit on the amount of average total activation for a neuron throughout training!

• Solution 3
  • Kurtosis: $\frac{\mu_4}{\sigma^4} = \frac{E[(X-\mu)^4]}{(E[(X-\mu)^2])^2}$
    • Calculated over the activations of the whole network.
    • High kurtosis $\Rightarrow$ sparse activations.
    • “Kurtosis has only been studied for response distributions of model neurons where negative responses are allowed. It is unclear whether kurtosis is actually a sensible measure for realistic, non-negative response distributions.” - [http://www.scholarpedia.org/article/Sparse_coding](http://www.scholarpedia.org/article/Sparse_coding)

• And many many other ways...
Enforcing sparsity: a popular choice

- Limit the amount of total activation for a neuron throughout training!
- Use $\rho_i$ to denote the activation of neuron $x$ on input $i$. The average activation of the neuron over the training set:

$$\hat{\rho}_i = \frac{1}{m} \sum_{i}^{m} \rho_i$$

- Now, to enforce sparsity, we limit to $\hat{\rho}_i = \rho_0$.
- $\rho_0$: A small value.
  - Yet another hyperparameter which may be tuned.
  - typical value: 0.05.
- The neuron must be inactive most of the time to keep its activations under the limit.
Enforcing sparsity

\[ \hat{\rho}_i = \frac{1}{m} \sum_{i} \rho_i \]

• How to limit \( \hat{\rho}_i = \rho_0 \)? How do we add integrate this as a penalty term into the loss function?
  • \( \rho_0 \) is called the sparsity parameter.

• Use Kullback-Leibler divergence:
  \[ \sum_{i} KL(\rho_0 || \hat{\rho}_i) \]
  Or, equivalently as (since this is between two Bernoulli variables with mean \( \rho_0 \) and \( \hat{\rho}_i \)):
  \[ \sum_{i} \rho_0 \log \frac{\rho_0}{\hat{\rho}_i} + (1 - \rho_0) \log \frac{1 - \rho_0}{1 - \hat{\rho}_i} \]

\[ D_{KL}(P||Q) = \sum_{i} P(i) \log \frac{P(i)}{Q(i)}. \]
Backpropagation and training

\[ S = \beta \sum_i \rho_0 \log \frac{\rho_0}{\hat{\rho}_i} + (1 - \rho_0) \log \frac{1 - \rho_0}{1 - \hat{\rho}_i} \]

\[ \frac{dS}{d\rho_i} = \beta \left( -\rho_0 \frac{1}{\hat{\rho}_i \ln 10} + (1 - \rho_0) \frac{1}{(1 - \hat{\rho}_i \ln 10)} \right) \]

- If you use \( \ln \) in KL:

\[ \frac{dS}{d\rho_i} = \beta \left( -\frac{\rho_0}{\hat{\rho}_i} + \frac{1 - \rho_0}{1 - \hat{\rho}_i} \right) \]

- So, if we integrate into the original error term:

\[ \delta_h = o_h(1 - o_h) \left( \sum_k w_{kh}\delta_k \right) + \beta \left( -\frac{\rho_0}{\hat{\rho}_h} + \frac{1 - \rho_0}{1 - \hat{\rho}_h} \right) \]

- Need to change \( o_h(1 - o_h) \) if you use a different activation function.

Reminder

- For each hidden unit \( h \), calculate its error term \( \delta_h \):

\[ \delta_h = o_h(1 - o_h) \sum_{k \in \text{outputs}} w_{kh}\delta_k \]

- Update every weight \( w_{ji} \):

\[ w_{ji} = w_{ji} + \eta \delta_j x_{ji} \]
Backpropagation and training

\[ S = \beta \sum_i \rho_0 \log \frac{\rho_0}{\hat{\rho}_i} + (1 - \rho_0) \log \frac{1 - \rho_0}{1 - \hat{\rho}_i} \]

• Do you see a problem here?

• \( \hat{\rho}_i \) should be calculated over the training set.

• In other words, we need to go through the whole dataset (or batch) once to calculate \( \hat{\rho}_i \).
Loss & decoders & encoders

• Be careful about the range of your activations and the range of the output

• Real-valued input:
  • Encoder: use sigmoid
  • Decoder: no need for non-linearity.
  • Loss: Squared-error Loss
  • Vincent et al. (2010):

  For real-valued $x$, that is, $x \in \mathbb{R}^d$: $X|z \sim \mathcal{N}(z, \sigma^2 \mathbf{I})$, that is, $X|z \sim \mathcal{N}(z_j, \sigma^2)$. This yields $L(x, z) = L_2(x, z) = C(\sigma^2) \|x - z\|^2$ where $C(\sigma^2)$ denotes a constant that depends only on $\sigma^2$ and that can be ignored for the optimization.

• Binary-valued input:
  • Encoder: use sigmoid.
  • Decoder: use sigmoid.
  • Loss: use cross-entropy loss:

\[-\Sigma_j [x_j \log z_j + (1 - x_j) \log (1 - z_j)] :\]
Loss & decoders & encoders

• Kullback-Leibler divergence assumes that the variables are in the range $[0,1]$.
  • I.e., you are bound to use sigmoid for the hidden layer if you use KL to limit the activations of hidden units.
k-Sparse Autoencoder
• Note that it doesn’t have an activation function!

• Non-linearity comes from $k$-selection.

---

$k$-Sparse Autoencoders:

**Training:**
1) Perform the feedforward phase and compute
\[ z = W^\top x + b \]
2) Find the $k$ largest activations of $z$ and set the rest to zero.
\[ z_{(\Gamma)^c} = 0 \quad \text{where} \quad \Gamma = \text{supp}_k(z) \]
3) Compute the output and the error using the sparsified $z$.
\[ \hat{x} = Wz + b' \]
\[ E = \|x - \hat{x}\|_2^2 \]
3) Backpropagate the error through the $k$ largest activations defined by $\Gamma$ and iterate.

**Sparse Encoding:**
Compute the features $h = W^\top x + b$. Find its $\alpha k$ largest activations and set the rest to zero.
\[ h_{(\Gamma)^c} = 0 \quad \text{where} \quad \Gamma = \text{supp}_{\alpha k}(h) \]
http://www.ericlwilkinson.com/blog/2014/11/19/deep-learning-sparse-autoencoders
Denoising Auto-encoders (DAE)

Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion

Pascal Vincent
Département d'informatique et de recherche opérationnelle
Université de Montréal
2920, chemin de la Tour
Montréal, Québec, H3T 1J8, Canada

Pascal.Vincent@umontreal.ca

Hugo Larochelle
Department of Computer Science
University of Toronto
10 King's College Road
Toronto, Ontario, M5S 3G4, Canada

Larochelle@cs.toronto.edu

Isabelle Lajoie
Pascal Vincent

Yoshua Bengio
Hugo Larochelle

Pierre-Antoine Manzagol
Isabelle Lajoie

Department d’informatique et de recherche opérationnelle
Université de Montréal
2920, chemin de la Tour
Montréal, Québec, H3T 1J8, Canada

Isabelle.Lajoie.1@umontreal.ca
Yoshua.Bengio@umontreal.ca
Pierre-Antoine.Manzagol@umontreal.ca
Denoising Auto-encoders

• Simple idea:
  • randomly corrupt some of the inputs (as many as half of them) – e.g., set them to zero.
  • Train the autoencoder to reconstruct the input from a corrupted version of it.
  • The auto-encoder is to predict the corrupted (i.e. missing) values from the uncorrupted values.
  • This requires capturing the joint distribution between a set of variables

• A stochastic version of the auto-encoder.
Denoising auto-encoder (DAE)

- learns robust & useful features
- easier to train than RBM features
- yield similar or better classification performance (as deep net pre-training)

Minimize:

\[ J_{\text{DAE}}(\theta) = \sum_{x \in D} \mathbb{E}_{q(\tilde{x}|x)} [L(x, g(h(\tilde{x})))] \]
Denoising auto-encoder (DAE)

- Autoencoder training minimizes:
  \[ J_{\text{AE}}(\theta) = \sum_{x \in D} L(x, g(h(\tilde{x}))) \]

- Denoising autoencoder training minimizes
  \[ J_{\text{DAE}}(\theta) = \sum_{x \in D} \mathbb{E}_{q(\tilde{x}|x)} [L(x, g(h(\tilde{x})))] \]

  Cannot compute expectation exactly
  \(\Rightarrow\) use stochastic gradient descent,
  **sampling corrupted inputs** \(\tilde{x}|x\)

  **Possible corruptions** \(q:\)
  - zeroing pixels at random
    (now called «dropout» noise)
  - additive Gaussian noise
  - salt-and-pepper noise
  - ...

(Pascal Vincent)
Loss in DAE

- You may give extra emphasis on “corrupted” dimensions:

\[ L_{2,\alpha}(x, z) = \alpha \left( \sum_{j \in J(\tilde{x})} (x_j - z_j)^2 \right) + \beta \left( \sum_{j \notin J(\tilde{x})} (x_j - z_j)^2 \right), \]

where \( J(\tilde{x}) \) denotes the indexes of the components of \( x \) that were corrupted.

Or, in cross-entropy-based loss:

\[ L_{\text{H},\alpha}(x, z) = \alpha \left( - \sum_{j \in J(\tilde{x})} [x_j \log z_j + (1 - x_j) \log (1 - z_j)] \right) \]
\[ + \beta \left( - \sum_{j \notin J(\tilde{x})} [x_j \log z_j + (1 - x_j) \log (1 - z_j)] \right). \]
Denoising Auto-encoders

- To undo the effect of a corruption induced by the noise, the network needs to capture the statistical dependencies between the inputs.

- This can be interpreted from many perspectives (see Vincent et al., 2008):
  - the manifold learning perspective,
  - stochastic operator perspective.
Denoising auto-encoders: manifold interpretation

- DAE learns to «project back» corrupted input onto manifold.
- Representation $h \approx$ location on the manifold

prior: examples concentrate near a lower dimensional “manifold”

32

mercredi 5 août 2015

(Pascal Vincent)
Stacked Denoising Auto-Encoders (SDAE)

Advantages over stacking RBMs

- No partition function, can measure training criterion
- Very flexible: encoder & decoder can use any parametrization (more layers...)
- Performs as well or better than stacking RBMs for unsupervised pre-training

Budget of 10 million iterations

Online classification error

Number of examples seen

Infinite MNIST

mercredi 5 août 2015
Types of corruption

• Gaussian Noise (additive, isotropic)

• Masking Noise
  • Set a randomly selected subset of input to zero for each sample (the fraction ratio is constant, a parameter)

• Salt-and-pepper Noise:
  • Set a randomly selected subset of input to maximum or minimum for each sample (the fraction ratio is constant, a parameter)
Figure 6: Weight decay vs. Gaussian noise. We show typical filters learnt from natural image patches in the over-complete case (200 hidden units). *Left*: regular autoencoder with weight decay. We tried a wide range of weight-decay values and learning rates: filters never appeared to capture a more interesting structure than what is shown here. Note that some local blob detectors are recovered compared to using no weight decay at all (Figure 5 right). *Right*: a denoising autoencoder with additive Gaussian noise ($\sigma = 0.5$) learns Gabor-like local oriented edge detectors. Clearly the filters learnt are qualitatively very different in the two cases.
Figure 7: Filters obtained on natural image patches by denoising autoencoders using other noise types. *Left*: with 10% salt-and-pepper noise, we obtain oriented Gabor-like filters. They appear slightly less localized than when using Gaussian noise (contrast with Figure 6 right). *Right*: with 55% zero-masking noise we obtain filters that look like oriented gratings. For the three considered noise types, denoising training appears to learn filters that capture meaningful natural image statistics structure.

(Vincent et al., 2010)
Training DAE

- Training algorithm does not change
  - However, you may give different emphasis on the error of reconstruction of the corrupted input.

- SGD is a popular choice

- Sigmoid is a suitable choice unless you know what you are doing.