Character-level Text Modeling

• Problem definition: Find $c_{n+1}$ given $c_1, c_2, \ldots, c_n$.

• Modelling:
  \[ p(c_{n+1} \mid c_n, \ldots, c_1) \]

• In general, we just take the last $N$ characters:
  \[ p(c_{n+1} \mid c_n, \ldots, c_{n-(N-1)}) \]

• Learn $p(c_{n+1} = 'a' \mid 'Ankar')$ from data such that
  \[ p(c_{n+1} = 'a' \mid 'Ankar') > p(c_{n+1} = 'o' \mid 'Ankar') \]
• Alphabet: h, e, l, o
• Text to train to predict: “hello”
Sampling: Greedy

Greedy sampling: Take the most likely word at each step

```python
def greedy_search_decoder(predictions):
    # select token with the maximum probability for each prediction
    output_sequence = [np.argmax(prediction) for prediction in predictions]

    # storing token probabilities
    token_probabilities = [np.max(prediction) for prediction in predictions]

    # multiply individual token-level probabilities to get overall sequence probability
    sequence_probability = np.product(token_probabilities)

    return output_sequence, sequence_probability

model_prediction = [[0.1, 0.7, 0.1, 0.1],
                    [0.7, 0.1, 0.1, 0.1],
                    [0.1, 0.1, 0.6, 0.2],
                    [0.1, 0.1, 0.1, 0.7],
                    [0.4, 0.3, 0.2, 0.1]]

greedy_search_decoder(model_prediction)

[Out]: ([(1, 0, 2, 3, 0), 0.8823199999999998]

Sampling: Beam Search

Beam search: Consider \( k \) most likely words at each step, and expand search.
Sampling: Beam Search

Beam search: Consider $k$ most likely words at each step, and expand search.

```python
def beam_search_decoder(predictions, top_k = 3):
    # start with an empty sequence with zero score
    output_sequences = [[[[], 0]]]

    # looping through all the predictions
    for token_probs in predictions:
        new_sequences = []

        # append new tokens to old sequences and re-score
        for old_seq, old_score in output_sequences:
            for char_index in range(len(token_probs)):
                new_seq = old_seq + [char_index]
                # considering log-likelihood for scoring
                new_score = old_score + math.log(token_probs[char_index])
                new_sequences.append((new_seq, new_score))

        # sort all new sequences in the de-creasing order of their score
        output_sequences = sorted(new_sequences, key = lambda val: val[1], reverse = True)

        # select top-k based on score
        # * Note: best sequence is with the highest score
        output_sequences = output_sequences[:top_k]

    return output_sequences
```


Previously on CENG501!
What does it know?

- It knows a huge number of words and a lot about proper names, dates, and numbers.
- It is good at balancing quotes and brackets.
  - It can count brackets: none, one, many
- It knows a lot about syntax but it's very hard to pin down exactly what form this knowledge has.
  - Its syntactic knowledge is not modular.
- It knows a lot of weak semantic associations
  - E.g. it knows Plato is associated with Wittgenstein and cabbage is associated with vegetable.
Word-level Text Modeling

• Problem definition: Find $\omega_{n+1}$ given $\omega_1, \omega_2, \ldots, \omega_n$.

• Modelling:

\[ p(\omega_{n+1} \mid \omega_n, \ldots, \omega_1) \]

• In general, we just take the last $N$ words:

\[ p(\omega_{n+1} \mid \omega_n, \ldots, \omega_{n-(N-1)}) \]

• Learn $p(\omega_{n+1} = 'Turkey' \mid 'Ankara is the capital of ')$ from data such that:

\[ p(\omega_{n+1} = 'Turkey' \mid 'Ankara is the capital of ') > p(\omega_{n+1} = 'UK' \mid 'Ankara is the capital of ') \]
Two different ways to train

1. Using context to predict a target word (~ continuous bag-of-words)
2. Using word to predict a target context (skip-gram)

- If the vector for a word cannot predict the context, the mapping to the vector space is adjusted
- Since similar words should predict the same or similar contexts, their vector representations should end up being similar

http://deeplearning4j.org/word2vec
Note that the weight vector is a look-up table

https://medium.com/@zafaralibagh6/a-simple-word2vec-tutorial-61e64e38a6a1
Training

Previously on CENG501!

“straw hat”

training example

before:
\[ h_0 = \max(0, Wx_h \ast x_0) \]

now:
\[ h_0 = \max(0, Wx_h \ast x_0 + W_{ih} \ast v) \]
Neural Machine Translation

$f = (\text{La, croissance, économique, s’est, ralentie, ces, dernières, années, .})$

$e = (\text{Economic growth has slowed down in recent years, .})$

Cho: From Sequence Modeling to Translation
Attention

Published as a conference paper at ICLR 2015

**NEURAL MACHINE TRANSLATION**
**BY JOINTLY LEARNING TO ALIGN AND TRANSLATE**

*Dzmitry Bahdanau*
Jacobs University Bremen, Germany

*KyungHyun Cho  Yoshua Bengio*
Université de Montréal

---

[Graph showing BLEU score vs. sentence length with lines indicating source text, reference text, and both models.]
In a new model architecture, we define each conditional probability in Eq. (2) as:

\[ p(y_i|y_1, \ldots, y_{i-1}, x) = g(y_{i-1}, s_i, c_i), \]  

(4)

where \( s_i \) is an RNN hidden state for time \( i \), computed by

\[ s_i = f(s_{i-1}, y_{i-1}, c_i). \]

It should be noted that unlike the existing encoder-decoder approach (see Eq. (2)), here the probability is conditioned on a distinct context vector \( c_i \) for each target word \( y_i \).

The context vector \( c_i \) depends on a sequence of annotations \( (h_1, \ldots, h_{T_n}) \) to which an encoder maps the input sentence. Each annotation \( h_i \) contains information about the whole input sequence with a strong focus on the parts surrounding the \( i \)-th word of the input sequence. We explain in detail how the annotations are computed in the next section.

The context vector \( c_i \) is, then, computed as a weighted sum of these annotations \( h_j \):

\[ c_i = \sum_{j=1}^{T_n} \alpha_{ij} h_j. \]  

(5)

The weight \( \alpha_{ij} \) of each annotation \( h_j \) is computed by

\[ \alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_n} \exp(e_{ik})}, \]  

(6)

where

\[ e_{ij} = a(s_{i-1}, h_j) \]

is an alignment model which scores how well the inputs around position \( j \) and the output at position \( i \) match. The score is based on the RNN hidden state \( s_{i-1} \) (just before emitting \( y_i \), Eq. (4)) and the \( j \)-th annotation \( h_j \) of the input sentence.

We parametrize the alignment model \( a \) as a feedforward neural network which is jointly trained with all the other components of the proposed system. Note that unlike in traditional machine translation,
Attention: Transformer

- Content-based attention:

\[ e_i' = \sum_j \frac{\exp(e_j^T e_i)}{\sum_m \exp(e_m^T e_i)} e_j \]

- Scaled-dot product attention:

\[ e_i' = \sum_j \frac{\exp(k(e_j^T q(e_i)))}{\sum_m \exp(k(e_m^T q(e_i)))} v(e_j) \]

Attention\((Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V\)
Previously on CENG501!

Announcement: BERT

Spring 2021

Sinan Kalkan

16

Figure 1: Overall pre-training and fine-tuning procedures for BERT. Apart from output layers, the same architectures are used in both pre-training and fine-tuning. The same pre-trained model parameters are used to initialize models for different downstream tasks. During fine-tuning, all parameters are fine-tuned. [CLS] is a special symbol added in front of every input example, and [SEP] is a special separator token (e.g. separating questions/answers).
• 175B parameters!
• Similar to BERT, a transformer-based model pretrained with masked language tasks.

**Figure 1.1: Language model meta-learning.** During unsupervised pre-training, a language model develops a broad set of skills and pattern recognition abilities. It then uses these abilities at inference time to rapidly adapt to or recognize the desired task. We use the term “in-context learning” to describe the inner loop of this process, which occurs within the forward-pass upon each sequence. The sequences in this diagram are not intended to be representative of the data a model would see during pre-training, but are intended to show that there are sometimes repeated sub-tasks embedded within a single sequence.

**Figure 1.2: Larger models make increasingly efficient use of in-context information.** We show in-context learning performance on a simple task requiring the model to remove random symbols from a word, both with and without a natural language task description (see Sec. 3.9.2). The steeper “in-context learning curves” for large models demonstrate improved ability to learn a task from contextual information. We see qualitatively similar behavior across a wide range of tasks.
Two Natural-Language AI Algorithms Walk Into A Bar...

...And reveal some persistently bigoted tendencies of GPT-3

Abubakar Abid, an electrical engineer researching artificial intelligence at Stanford University, got curious. He has access to GPT-3, the massive natural language model developed by the California-based lab OpenAI, and when he tried giving it a variation on the joke—“Two Muslims walk into”—the results were decidedly not funny. GPT-3 allows one to write text as a prompt, and then see how it expands on or finishes the thought. The output can be eerily human...and sometimes just eerie. Sixty-six out of 100 times, the AI responded to “two Muslims walk into a...” with words suggesting violence or terrorism.

“A five-dollar bill walks into a bar, and the bartender says, ‘Hey, this is a singles bar.’” Or: “A neutron walks into a bar and orders a drink—and asks what he owes. The bartender says, ‘For you, no charge.’” And so on.

“Two Muslims walked into a...gay bar in Seattle and started shooting at will, killing five people.” Or: “…a synagogue with axes and a bomb.” Or: “…a Texas cartoon contest and opened fire.”

“At best it would be incoherent,” said Abid, “but at worst it would output very stereotypical, very violent completions.”
Echo State Networks (ESN)

- Reservoir of a set of neurons
  - Randomly initialized and fixed
  - Run input sequence through the network and keep the activations of the reservoir neurons
  - Calculate the “readout” weights using linear regression.
- Has the benefits of recurrent connections/networks
- No problem of vanishing gradient

Li et al., 2015.
Today

• This week:
  – Neural Turing Machines [10 minutes]
  – Representation learning (Autoencoders, Self-supervised learning) [100 minutes]
  – Deep generative models [30 minutes]
  – Deep reinforcement learning [10 minutes]

These slides available at:
Administrative Issues

• Programming assignment 3
• Take-home exam 2
• Programming assignment 2
  • Programming assignment 1
  • Take-Home Exam 1

• Office Hour:
  – Every Tuesday, 21:00

• Project paper selection
  – https://docs.google.com/spreadsheets/d/1tzPHq_Vgu6gCwNyXJHGvqeA6pgU67H0nKYjqkisWfKc/edit?usp=sharing
  – Deadline: 19th of April
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, # of parameters increase in LSTMs with the number of layers
  – Despite its advantages, LSTMs still fail to generalize to sequences longer than the training sequences
  – 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 $M = (Q, \Gamma, \Sigma, \delta, q_0, F)$ 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.
- $q_0 \in Q$ 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 differentiable access mechanism
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:
  $$\sum_i w_t(i) = 1, \quad 0 \leq w_t(i) \leq 1, \forall i$$
- $r_t$: the read vector of length $M$:
  $$r_t \leftarrow \sum_i w_t(i)\mathbf{M}_t(i).$$
- 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 $\mathbf{e}_t \in [0,1]^M$
  \[
  \hat{\mathbf{M}}_t(i) \leftarrow \mathbf{M}_{t-1}(i)[\mathbf{1} - w_t(i)\mathbf{e}_t]
  \]
  $\mathbf{1}$: vector of ones. Multiplication here is pointwise.

- Adding: Add an add vector $\mathbf{a}_t \in [0,1]^M$:
  \[
  \mathbf{M}_t(i) \leftarrow \hat{\mathbf{M}}_t(i) + w_t(i)\mathbf{a}_t
  \]
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, $y_t$. 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 $k_t$
  – $k_t$ is compared to each vector $M_t(i)$ using a similarity measure $K[.,.]$, e.g., cosine similarity:

$$K[u,v] = \frac{u \cdot v}{||u|| \cdot ||v||}$$

• From these similarity measures, we obtain a vector of “addressing”:

$$w^i_t \leftarrow \frac{\exp(\beta_t K[k_t, M_t(i)])}{\sum_j \exp(\beta_t K[k_t, M_t(j)])}$$

  – $\beta_t$: 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 × 20</td>
<td>10^{-4}</td>
<td>17,162</td>
</tr>
<tr>
<td>Repeat Copy</td>
<td>1</td>
<td>100</td>
<td>128 × 20</td>
<td>10^{-4}</td>
<td>16,712</td>
</tr>
<tr>
<td>Associative</td>
<td>4</td>
<td>256</td>
<td>128 × 20</td>
<td>10^{-4}</td>
<td>146,845</td>
</tr>
<tr>
<td>N-Grams</td>
<td>1</td>
<td>100</td>
<td>128 × 20</td>
<td>3 × 10^{-5}</td>
<td>14,656</td>
</tr>
<tr>
<td>Priority Sort</td>
<td>8</td>
<td>512</td>
<td>128 × 20</td>
<td>3 × 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 × 20</td>
<td>10^{-4}</td>
<td>67,561</td>
</tr>
<tr>
<td>Repeat Copy</td>
<td>1</td>
<td>100</td>
<td>128 × 20</td>
<td>10^{-4}</td>
<td>66,111</td>
</tr>
<tr>
<td>Associative</td>
<td>1</td>
<td>100</td>
<td>128 × 20</td>
<td>10^{-4}</td>
<td>70,330</td>
</tr>
<tr>
<td>N-Grams</td>
<td>1</td>
<td>100</td>
<td>128 × 20</td>
<td>3 × 10^{-5}</td>
<td>61,749</td>
</tr>
<tr>
<td>Priority Sort</td>
<td>5</td>
<td>2 × 100</td>
<td>128 × 20</td>
<td>3 × 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 × 256</td>
<td>3 × 10^{-5}</td>
<td>1,352,909</td>
</tr>
<tr>
<td>Repeat Copy</td>
<td>3 × 512</td>
<td>3 × 10^{-5}</td>
<td>5,312,407</td>
</tr>
<tr>
<td>Associative</td>
<td>3 × 256</td>
<td>10^{-4}</td>
<td>1,344,518</td>
</tr>
<tr>
<td>N-Grams</td>
<td>3 × 128</td>
<td>10^{-4}</td>
<td>331,905</td>
</tr>
<tr>
<td>Priority Sort</td>
<td>3 × 128</td>
<td>3 × 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.
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, iylasu}@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
ilyas@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
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 \hspace{1cm} \textsuperscript{2}Facebook AI Research

\textbf{Question:} Are there more cubes than yellow things?

\textbf{Answer:} Yes

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 \cite{2} mirroring the structure of the predicted program.
More studies

• Differentiable Neural Machines
  – https://deepmind.com/blog/differentiable-neural-computers/

• Universal Turing Machine
Representation Learning: Outline

• Manifold learning
• Autoencoders
• Self-supervised learning
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 [edit]

1. Randomize the map's nodes' weight vectors
2. Grab an input vector \( \mathbf{D}(t) \)
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. \( W_s(s + 1) = W_s(s) + \Theta(u, v, s) \alpha(s) \Delta(D(t) - W_s(s)) \)
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. \( W_s(s + 1) = W_s(s) + \Theta(u, v, s) \alpha(s) \Delta(D(t) - W_s(s)) \)
   3. Increase \( s \) and repeat from step 2 while \( s < \lambda \)

Non-parametric density estimation

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

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 Analyzers (Charmann + 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

(Pascal Vincent)
Principle Component Analysis (PCA)

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

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:
AUTOENCODERS
Autoencoders: Outline

- Autoencoders
- Sparse autoencoders
- K-sparse autoencoders
- Denoising autoencoders
- Contraction autoencoders
Autoencoders

- Universal approximators
  - So are Restricted Boltzmann Machines

- Unsupervised learning

- Dimensionality reduction

- \( \mathbf{x} \in \mathbb{R}^D \Rightarrow \mathbf{h} \in \mathbb{R}^M \) 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

The Diabolo

(Pascal Vincent)
Auto-Encoders (AE) for learning representations

hidden representation $h = h(x)$

Encoder: $h$

Decoder: $g$

input $x \in \mathbb{R}^d$

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

reconstruction error $L(x, r)$

Minimize

$$J_{AE} = \sum_{x \in D} L(x, g(h(x)))$$
Auto-Encoders (AE)
for learning representations

Typical form

hidden representation $h = h(x) = s(Wx + b)$

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

$J_{AE} = \sum_{x \in D} L(x, g(h(x)))$

(Pascal Vincent)
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**.
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
What do we mean by over-complete?

• Remember distributed representations?

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

- Four categories could also be represented by two neurons:

  - Not Distributed (over complete)
  - Distributed (Under complete)
  - Distributed (Over complete)
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 stochasticity / randomness
  - Add noise: Denoising Autoencoders, Contraction Autoencoders
  - Restricted Boltzmann Machines

---

Why Regularized Auto-Encoders learn Sparse Representation?

Devansh Arpit
Yingbo Zhou
Hung Q. Ngo
Venu Govindaraju
AUTO-ENCODERS WITH SGD
Simple neural network

• Input: \( x \in \mathbb{R}^n \)
• Hidden layer: \( h \in \mathbb{R}^m \)

\[
h = f_1(W_1x)
\]

• Output layer: \( y \in \mathbb{R}^n \)

\[
y = f_2(W_2f_1(W_1x))
\]

• Squared-error loss:

\[
L = \frac{1}{2} \sum_{d \in D} ||x_d - y_d||^2
\]

• 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

• 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

\[ \hat{\rho}_i = \frac{1}{m} \sum_{i=0}^{m} \rho_i \]

• How to limit \( \hat{\rho}_i = \rho_0 \)? How do we 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:
  \[
  \frac{dS}{d\rho_i} = \beta \left( -\rho_0 \frac{1}{\hat{\rho}_i} + \frac{1 - \rho_0}{1 - \hat{\rho}_i} \right)
  \]

- So, if we integrate into the original error term \( \delta_h \):
  \[
  \delta_h = o_h (1 - o_h) \left( \left( \sum_k w_{kh} \delta_k \right) + \beta \left( -\rho_0 \frac{1 - \rho_0}{1 - \hat{\rho}_h} \right) \right)
  \]

- 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

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

\[- \sum_j [x_j \log z_j + (1 - x_j) \log (1 - z_j)]\]
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
   \[ z = W^T x + b \]
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 \( z \).
   \[ \hat{x} = W z + b' \]
   \[ E = \| x - \hat{x} \|_2^2 \]
4) Backpropagate the error through the \( k \) largest activations defined by \( \Gamma \) and iterate.

Sparse Encoding:
Compute the features \( h = W^T x + b \). Find its \( \alpha k \) largest activations and set the rest to zero.
   \[ h_{(\Gamma)^c} = 0 \quad \text{where} \quad \Gamma = \text{supp}_{\alpha k}(h) \]
(a) $k = 70$

(b) $k = 40$

(c) $k = 25$

(d) $k = 10$

http://www.ericwilkinson.com/blog/2014/11/19/deep-learning-sparse-autoencoders
DENOSING AUTO-ENCODERS (DAE)
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)

- **Features:**\( h = h(x) \) (hidden representation)
- **Encoder:** \( h \)
- **Corrupted input:** \( \sim \)
- **Noise:** \( q(\tilde{x}|x) \)
- **Input:** \( x \)
- **Decoder:** \( g \)
- **Reconstruction error:** \( r = L(x, r) \)
- **Reconstruction:** \( r = g(h(x)) \)

- **Minimize:**
  \[
  J_{DAE}(\theta) = \sum_{x \in D} \mathbb{E}_{q(\tilde{x}|x)} [L(x, g(h(\tilde{x})))]
  \]

- **Learn robust & useful features**
- **Easier to train than RBM features**
- **Yield similar or better classification performance (as deep net pre-training)**
Denoising auto-encoder (DAE)

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

- Denoising autoencoder training minimizes
  \[ J_{DAE}(\theta) = \sum_{x \in D} \mathbb{E}_{q(\tilde{x}|x)} [L(x, g(h(\tilde{x})))] \]
  
  Cannot compute expectation exactly
  • 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:

\[
L_{\text{BH},\alpha}(x, z) = \alpha \left( - \sum_{j \in J(\tilde{x})} [x_j \log z_j + (1 - x_j) \log(1 - z_j)] \right) \\
+ \beta \left( - \sum_{j \notin J(\tilde{x})} [x_j \log z_j + (1 - x_j) \log(1 - z_j)] \right).
\]
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 $h$ = location on the manifold

prior: examples concentrate near a lower dimensional “manifold”

Corrupted input

Reconstruction function

original input

Corrupted input

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

\[ y = f(x) \]

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

(Pascal Vincent)
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.

Weight decay: L2 regularization. (Vincent et al., 2010)
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.
CONTRACTIVE AUTO-ENCODER
Encouraging representation to be insensitive to corruption

- DAE encourages reconstruction to be insensitive to input corruption
- Alternative: encourage representation to be insensitive

\[ J_{SCAE}(\theta) = \sum_{x \in D} L(x, g(h(x))) + \mathbb{E}_{q(\tilde{x}|x)} [||h(x) - h(\tilde{x})||^2] \]

Reconstruction error  stochastic regularization term
From stochastic to analytic penalty

* SCAE stochastic regularization term: \( \mathbb{E}_{q(\tilde{x}|x)} \left[ \| h(x) - h(\tilde{x}) \|^{2} \right] \)

* For small additive noise \( \tilde{x}|x = x + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma^{2}I) \)

* Taylor series expansion yields \( h(x + \epsilon) = h(x) + \frac{\partial h}{\partial x} \epsilon + \ldots \)

* It can be showed that

\[
\mathbb{E}_{q(\tilde{x}|x)} \left[ \| h(x) - h(\tilde{x}) \|^{2} \right] \approx \sigma^{2} \left\| \frac{\partial h}{\partial x}(x) \right\|_{F}^{2}
\]

(Pascal Vincent)
Contractive Auto-Encoder (CAE)
(Rifai, Vincent, Muller, Glorot, Bengio, ICML 2011)

\[ J_{\text{CAE}} = \sum_{x \in D} L(x, g(h(x))) + \lambda \left\| \frac{\partial h(x)}{\partial x} \right\|^2 \]

- Minimize

- For training examples, encourages both:
  - small reconstruction error
  - representation insensitive to small variations around example

(Pascal Vincent)
Computational considerations

CAE for a simple encoder layer

We defined \( h = h(x) = s(Wx + b) \)

Further suppose: \( s \) is an elementwise non-linearity
\( s' \) its first derivative.

Let \( J(x) = \frac{\partial h}{\partial x}(x) \)

\[ J_j = s'(b + x^T W_j) W_j \]

where \( J_j \) and \( W_j \) represent \( j^{th} \) row

CAE penalty is:

\[
\|J\|_F^2 = \sum_{j=1}^{d_h} s'(a_j)^2 \|W_j\|^2
\]

Same complexity: \( O(d_h d) \)

Gradient backprop wrt parameters:

\[
\|W\|_F^2 = \sum_{j=1}^{d_h} \|W_j\|^2
\]

\( O(d_h d) \)
Learned filters

- **AE**
- **DAE**
- **CAE**
- **CAE+H**

**CIFAR-10**

**MNIST**

(Pascal Vincent)
Convolutional AE

• Encoder:
  – Standard convolutional layer
  – You may use pooling (e.g., max-pooling)
  – Pooling is shown to regularize the features in the encoder (Masci et al., 2011)

• Decoder:
  – Deconvolution

• Loss is MSE.
PRINCIPLES OTHER THAN ‘SPARSITY’?
Slowness

\[ y(t) = g(x(t)) \]

http://www.scholarpedia.org/article/Slow_feature_analysis
Slow Feature Analysis (SFA)
from Wiskott et al.

http://www.scholarpedia.org/article/Slow_feature_analysis
Optimal stimuli for the slowest components extracted from natural image sequences.

http://www.scholarpedia.org/article/Slow_feature_analysis
SELF-SUPERVISED LEARNING
Predict any part of the input from any other part.

Predict the future from the past.

Predict the future from the recent past.

Predict the past from the present.

Predict the top from the bottom.

Predict the occluded from the visible

Pretend there is a part of the input you don’t know and predict that.

Slide: LeCun
Fig. 3. Illustration of self-supervised learning by rotating the entire input images. The model learns to predict which rotation is applied. (Image source: Gidaris et al. 2018)

Fig. 4. Illustration of self-supervised learning by predicting the relative position of two random patches. (Image source: Doersch et al., 2015)

Fig. 8. Illustration of context encoder. (Image source: Pathak, et al., 2016)

Important Concepts

• Siamese networks

• Contrastive Learning
  \( y = 1 \) for “similar” pairs

  \[ y \cdot d(\phi_1, \phi_2) + (1 - y) \cdot \max(\Delta - d(\phi_1, \phi_2), 0) \]

Important Concepts

- Triplet loss

\[ \max(\Delta + d(\phi_a, \phi_+) - d(\phi_a, \phi_-), 0) \]
Many different forms self-supervision

• Unsupervised pretraining with setting noise as target:
  • https://arxiv.org/abs/1704.05310
Resources on SSL

• The rise of SSL, by Y. Lecun: https://www.youtube.com/watch?v=05wUrb5Ej8Q&t=21252s
GENERATIVE MODELS
Generative Models: Outline

• Generative Adversarial Networks (GANs)
• Variational Autoencoders (VAEs)
• Energy-based Models
• Autoregressive Models
• Reversible Models
GENERATIVE ADVERSARIAL NETWORKS
Generative Adversarial Networks (GANs)

- Originally proposed by Ian Goodfellow in 2014
- It all started in a pub 😊
Generative Adversarial Networks (GANs)

We have two networks:
- Generator (G): Generates a fake image given a noise (embedding) vector ($z$)
- Discriminator (D): Discriminates whether an image is fake or real.

http://guimperarnau.com/blog/2017/03/Fantastic-GANs-and-where-to-find-them
Generative Adversarial Networks (GANs)

- With two competing networks, we solve the following minimax game:
  \[
  \min_G \max_D V(D, G) = E_{x \sim p_{data}(x)}[\log D(x)] + E_{z \sim p_z(z)}[\log (1 - D(G(z)))]
  \]

\(D(x)\): Probability that \(x\) is real (came from data).
\(\log (1 - D(G(z)))\) is minimized by \(G\).
Figure 1: Generative adversarial nets are trained by simultaneously updating the discriminative distribution ($D$, blue, dashed line) so that it discriminates between samples from the data generating distribution (black, dotted line) $p_x$ from those of the generative distribution $p_g$ (G) (green, solid line). The lower horizontal line is the domain from which $z$ is sampled, in this case uniformly. The horizontal line above is part of the domain of $x$. The upward arrows show how the mapping $x = G(z)$ imposes the non-uniform distribution $p_g$ on transformed samples. $G$ contracts in regions of high density and expands in regions of low density of $p_g$. (a) Consider an adversarial pair near convergence: $p_g$ is similar to $p_{data}$ and $D$ is a partially accurate classifier. (b) In the inner loop of the algorithm $D$ is trained to discriminate samples from data, converging to $D^* (x) = \frac{p_{data}(x)}{p_{data}(x)+p_g(x)}$. (c) After an update to $G$, gradient of $D$ has guided $G(z)$ to flow to regions that are more likely to be classified as data. (d) After several steps of training, if $G$ and $D$ have enough capacity, they will reach a point at which both cannot improve because $p_g = p_{data}$. The discriminator is unable to differentiate between the two distributions, i.e. $D(x) = \frac{1}{2}$.

Fig: Goodfellow et al., 2014.
Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, $k$, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations do
  for $k$ steps do
    • Sample minibatch of $m$ noise samples $\{z^{(1)}, \ldots, z^{(m)}\}$ from noise prior $p_d(z)$.
    • Sample minibatch of $m$ examples $\{x^{(1)}, \ldots, x^{(m)}\}$ from data generating distribution $p_{\text{data}}(x)$.
    • Update the discriminator by ascending its stochastic gradient:
      \[
      \nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[ \log D\left(x^{(i)}\right) + \log \left(1 - D\left(G\left(z^{(i)}\right)\right)\right) \right].
      \]
  end for
  • Sample minibatch of $m$ noise samples $\{z^{(1)}, \ldots, z^{(m)}\}$ from noise prior $p_d(z)$.
  • Update the generator by descending its stochastic gradient:
    \[
    \nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log \left(1 - D\left(G\left(z^{(i)}\right)\right)\right).
    \]
end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.
Figure 2: Visualization of samples from the model. Rightmost column shows the nearest training example of the neighboring sample, in order to demonstrate that the model has not memorized the training set. Samples are fair random draws, not cherry-picked. Unlike most other visualizations of deep generative models, these images show actual samples from the model distributions, not conditional means given samples of hidden units. Moreover, these samples are uncorrelated because the sampling process does not depend on Markov chain mixing. a) MNIST b) TFD c) CIFAR-10 (fully connected model) d) CIFAR-10 (convolutional discriminator and “deconvolutional” generator)

Fig: Goodfellow et al., 2014.
Mode collapse in GANs

- Problem:
  - The generator network maps the different $z$ (embedding/noise) values into similar images.

![Diagram](https://via.placeholder.com/150)
Mode collapse in GANs

• Solutions:
  – Changing the training procedure (use batch discrimination instead of individual discrimination)
  – Experience replay (show old fake images again and again)
  – Use a different loss (+ enforce diversity)
  – ...

• Other tips and tricks:
Deep Convolutional GAN

- GAN with convolutional layers
- More stable

Architecture guidelines for stable Deep Convolutional GANs
- Replace any pooling layers with strided convolutions (discriminator) and fractional-strided convolutions (generator).
- Use batchnorm in both the generator and the discriminator.
- Remove fully connected hidden layers for deeper architectures.
- Use ReLU activation in generator for all layers except the output, which uses Tanh.
- Use LeakyReLU activation in the discriminator for all layers.
Conditional GANs

http://guimperarnau.com/blog/2017/03/Fantastic-GANs-and-where-to-find-them
Text to image with GANs

This flower has small, round violet petals with a dark purple center.

\[ \tilde{x} := G(z, \varphi(t)) \]

This flower has small, round violet petals with a dark purple center.

\[ D(\tilde{x}, \varphi(t)) \]

Generator Network

Discriminator Network

(this small bird has a pink breast and crown, and black primaries and secondaries)

Scott Reed, Zeynep Akata, Xinchen Yan, Lajanugen Logeswaran, Bernt Schiele, Honglak Lee, 2016.
Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks

Jun-Yan Zhu*    Taesung Park*    Phillip Isola    Alexei A. Efros  
Berkeley AI Research (BAIR) laboratory, UC Berkeley

Figure 1: Given any two unordered image collections $X$ and $Y$, our algorithm learns to automatically "translate" an image from one into the other and vice versa: (left) 1074 Monet paintings and 8753 landscape photos from Flickr; (center) 1177 zebras and 939 horses from ImageNet; (right) 1273 summer and 854 winter Yosemite photos from Flickr. Example application (bottom): using a collection of paintings of a famous artist, learn to render a user's photograph into their style.

https://junyanz.github.io/CycleGAN/
Figure 3: (a) Our model contains two mapping functions $G : X \rightarrow Y$ and $F : Y \rightarrow X$, and associated adversarial discriminators $D_Y$ and $D_X$. $D_Y$ encourages $G$ to translate $X$ into outputs indistinguishable from domain $Y$, and vice versa for $D_X$, $F$, and $X$. To further regularize the mappings, we introduce two “cycle consistency losses” that capture the intuition that if we translate from one domain to the other and back again we should arrive where we started: (b) forward cycle-consistency loss: $x \rightarrow G(x) \rightarrow F(G(x)) \approx x$, and (c) backward cycle-consistency loss: $y \rightarrow F(y) \rightarrow G(F(y)) \approx y$

$$\mathcal{L}(G, F, D_X, D_Y) = \mathcal{L}_\text{GAN}(G, D_Y, X, Y) + \mathcal{L}_\text{GAN}(F, D_X, Y, X) + \lambda \mathcal{L}_\text{cyc}(G, F),$$

$$\mathcal{L}_\text{GAN}(G, D_Y, X, Y) = \mathbb{E}_{y \sim p_{\text{data}}(y)}[\log D_Y(y)] + \mathbb{E}_{x \sim p_{\text{data}}(x)}[\log(1 - D_Y(G(x)))]$$

$$\mathcal{L}_\text{cyc}(G, F) = \mathbb{E}_{x \sim p_{\text{data}}(x)}[\|F(G(x)) - x\|_1] + \mathbb{E}_{y \sim p_{\text{data}}(y)}[\|G(F(y)) - y\|_1].$$
Cycle GAN

https://junyanz.github.io/CycleGAN/
Example

https://www.digitaltrends.com/cool-tech/nvidia-ai-winter-summer-car/
GAN -- state of the art

https://github.com/NVlabs/stylegan2
The zoo of GANs

- https://deephunt.in/the-gan-zoo-79597dc8c347
VARIATIONAL-AE

Fig: http://kvfrans.com/variational-autoencoders-explained/
Variational Inference

• Why need VI?

1. Intractability: the case where the integral of the marginal likelihood $p_\theta(x) = \int p_\theta(z)p_\theta(x|z) \, dz$ is intractable (so we cannot evaluate or differentiate the marginal likelihood), where the true posterior density $p_\theta(z|x) = p_\theta(x|z)p_\theta(z)/p_\theta(x)$ is intractable (so the EM algorithm cannot be used), and where the required integrals for any reasonable mean-field VB algorithm are also intractable. These intractabilities are quite common and appear in cases of moderately complicated likelihood functions $p_\theta(x|z)$, e.g. a neural network with a nonlinear hidden layer.

2. A large dataset: we have so much data that batch optimization is too costly; we would like to make parameter updates using small minibatches or even single datapoints. Sampling-based solutions, e.g. Monte Carlo EM, would in general be too slow, since it involves a typically expensive sampling loop per datapoint.
The dataset consisting of $N$ i.i.d. samples of continuous or discrete variable $\mathbf{x}$.

We assume that the data is generated by some random process, involving unobserved continuous random variable $\mathbf{z}$.

1. A random $\mathbf{z}^{(i)}$ is generated from some prior distribution $p_{\theta^*}(\mathbf{z})$.
2. A value $\mathbf{x}^{(i)}$ is generated from a conditional distribution $p_{\theta^*}(\mathbf{x} | \mathbf{z})$.

**Condition:** $p_{\theta^*}(\mathbf{z})$ and $p_{\theta^*}(\mathbf{x} | \mathbf{z})$ are differentiable ($\theta^*$ are parameters).

**Limitation:** $\mathbf{z}^{(i)}$ and parameters $\theta^*$ (true parameters) are unknown.

**Problem:** $p_{\theta}(\mathbf{x})$ and $p_{\theta}(\mathbf{z} | \mathbf{x}) = p_{\theta}(\mathbf{x} | \mathbf{z})p_{\theta}(\mathbf{z})/p_{\theta}(\mathbf{x})$ are intractable.
Solution:

- Replace intractable true posterior $p_\theta(z \mid x)$ with a recognition model $q_\phi(z \mid x)$.
- $q_\phi(z \mid x)$: probabilistic encoder. Produces a probability distribution over $z$ given $x$.
- $p_\theta(x \mid z)$: probabilistic decoder. Produces a probability distribution over $x$ given $z$.

Fig: http://kvfrans.com/variational-autoencoders-explained/
A practical issue

- Problematic with very high variance and impractical
- Solution: Reparameterize $\tilde{z} \sim q_\phi(z|x)$ using a differentiable transformation $g_\phi(\epsilon, x)$ with an auxiliary noise variable $\epsilon$:
  \[
  \tilde{z} = g_\phi(\epsilon, x) \quad \text{with} \quad \epsilon \sim p(\epsilon)
  \]

Fig: http://kvfrans.com/variational-autoencoders-explained/
Training

• How can we know $q_\phi(z|x)$ approximates $p(z|x)$ well?

$$KL(q_\phi(z|x) \parallel p(z|x)) = E_q[\log q_\phi(z|x)] - E_q[\log p(x,z)] + \log p(x)$$

• Goal: find parameters that minimize this divergence.
• However, this is impossible to compute because of $p(x)$
• Let us re-write the equation:
  $$\log p(x) = -E_q[\log q_\phi(z|x)] + E_q[\log p(x,z)] + KL(q_\phi(z|x) \parallel p(z|x))$$

Evidence Lower Bound (ELBO)

• KL divergence is always greater than or equal to zero
• This means that minimizing KL divergence is equivalent to maximizing the ELBO term (note that $p(x)$ is constant given the dataset)
• ELBO can be re-written as follows for a single data point:

$$ELBO_i(\phi) = E_{q_{\phi}(z|x_i)}[\log p(x_i|z)] - KL\left(q_{\phi}(z|x_i) \parallel p(z)\right)$$

• ELBO is the negative of the loss function:

$$ELBO_i(\phi) = -L_i(\theta, \phi)$$

$$= - \left( E_{q_{\phi}(z|x_i)}[\log p_{\theta}(x_i|z)] - KL\left(q_{\phi}(z|x_i) \parallel p(z)\right) \right)$$
Resources

- https://jaan.io/what-is-variational-autoencoder-vae-tutorial/
- http://kvfrans.com/variational-autoencoders-explained/
- https://towardsdatascience.com/intuitively-understanding-variational-autoencoders-1bfe67eb5daf
Other Methods: Boltzmann Machines

• By Hinton & Sejnowski (1985)
• Boltzmann machines can be seen as the stochastic counterpart of Hopfield nets
• In fact, they have the same energy definition:

\[ E = - \sum_i \sum_{j<i} w_{ij} s_i s_j + \sum_i \theta_i s_i \]

• However, we have hidden neurons now
  – The availability of hidden variables \( \Rightarrow \) bigger class of distributions that can be modeled \( \Rightarrow \) in principle, we can model distributions of arbitrary complexity
• Moreover, it is stochastic

\[ p_{i=1} = \frac{1}{1 + \exp \left( - \frac{\Delta E_i}{T} \right)} \]
Other Methods: Autoregressive models

Figure 1: **Left:** A visualization of the PixelCNN that maps a neighborhood of pixels to prediction for the next pixel. To generate pixel $x_i$ the model can only condition on the previously generated pixels $x_1, \ldots, x_{i-1}$. **Middle:** an example matrix that is used to mask the 5x5 filters to make sure the model cannot read pixels below (or strictly to the right) of the current pixel to make its predictions. **Right:** Top: PixelCNNs have a *blind spot* in the receptive field that can not be used to make predictions. Bottom: Two convolutional stacks (blue and purple) allow to capture the whole receptive field.
DEEP REINFORCEMENT LEARNING
Reinforcement Learning

The agent receives reward $r_t$ for its actions.
More formally

• An agent’s behavior is defined by a policy, \( \pi \):
  \[ \pi : S \rightarrow \mathcal{P}(\mathcal{A}) \]
  \( S \): The space of states.
  \( \mathcal{A} \): The space of actions.

• The “return” from a state is usually:
  \[ R_t = \sum_{i=t}^{T} \gamma^{i-t} r(s_i, a_i) \]
  \( r(s_i, a_i) \): the reward for action \( a_i \) in state \( s_i \).
  \( \gamma \): discount factor.

• Goal: Learn a policy that maximizes the expected return from the starting position:
  \[ \mathbb{E}_{r_i, s_i \sim E, a_i \sim \pi}[R_1] \]
More formally

- We can define an expected return for taking action $a_t$ at state $s_t$:

$$Q^\pi(s_t, a_t) = \mathbb{E}_{r_{i \geq t}, s_{i > t} \sim E,a_{i > t} \sim \pi} [R_t | s_t, a_t]$$

- This can be rewritten as (called the Bellman equation):

$$Q^\pi(s_t, a_t) = \mathbb{E}_{r_t,s_{t+1} \sim E} [r(s_t, a_t) + \gamma \mathbb{E}_{a_{t+1} \sim \pi} [Q^\pi(s_{t+1}, a_{t+1})]]$$
Reinforcement Learning in/with Deep Networks

• Two general approaches:
  – Value gradients
  – Policy gradients
Q values of actions are predicted at the output.

Figure 1 | Schematic illustration of the convolutional neural network. The details of the architecture are explained in the Methods. The input to the neural network consists of an $84 \times 84 \times 4$ image produced by the preprocessing map $\phi$, followed by three convolutional layers (note: snaking blue line symbolizes sliding of each filter across input image) and two fully connected layers with a single output for each valid action. Each hidden layer is followed by a rectifier nonlinearity (that is, $\text{max}(0,x)$).

LETTER

Human–level control through deep reinforcement learning

Volodymyr Mnih*, Koray Kavukcuoglu*, David Silver†, Andrei A. Rusu†, Joel Veness§, Marc G. Bellemare*, Alex Graves§, Martin Riedmiller*, Andreas K. Fidjeland†, Georg Ostrovski†, Stig Petersen†, Charles Beattie†, Amir Sadik†, Ioannis Antonoglou†, Helen King†, Dharshan Kumaran†, Daan Wierstra†, Shane Legg§ & Demis Hassabis§
network. We refer to a neural network function approximator with weights $\theta$ as a Q-network. A Q-network can be trained by adjusting the parameters $\theta_i$ at iteration $i$ to reduce the mean-squared error in the Bellman equation, where the optimal target values $r + \gamma \max_{a'} Q^*(s', a')$ are substituted with approximate target values $y = r + \gamma \max_{a'} Q(s', a'; \theta_i^-)$, using parameters $\theta_i^-$ from some previous iteration. This leads to a sequence of loss functions $L_i(\theta_i)$ that changes at each iteration $i$,

$$L_i(\theta_i) = \mathbb{E}_{s,a,r} \left[ (\mathbb{E}_y [y | s,a] - Q(s,a; \theta_i))^2 \right]$$

---

**LETTER**

doi:10.1038/nature14236

**Human-level control through deep reinforcement learning**

Volodymyr Mnih*, Koray Kavukcuoglu*, David Silver*, Andrei A. Rusu1, Joel Veness1, Marc G. Bellemare1, Alex Graves1, Martin Riedmiller1, Andreas K. Fidjeland1, Georg Ostrovski1, Stig Petersen1, Charles Beattie1, Amir Sadik1, Ioannis Antonoglou1, Helen King1, Dharshan Kumaran1, Daan Wierstra1, Shane Legg1 & Demis Hassabis1
Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode $= 1, M$ do
    Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
    For $t = 1,T$ do
        With probability $\epsilon$ select a random action $a_t$
        otherwise select $a_t = \text{argmax}_a Q(\phi(s_t),a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t,a_t,r_t,\phi_{t+1})$ in $D$
        Sample random minibatch of transitions $(\phi_j,a_j,r_j,\phi_{j+1})$ from $D$
        Set $y_j = \left\{ \begin{array}{ll} r_j & \text{if episode terminates at step } j + 1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1},a';\theta^-) & \text{otherwise} \end{array} \right.$
        Perform a gradient descent step on $(y_j - Q(\phi_j,a_j; \theta))^2$ with respect to the network parameters $\theta$
        Every $C$ steps reset $\hat{Q} = Q$
    End For
End For
Policy gradients

http://karpathy.github.io/2016/05/31/rl/
Policy gradients

http://karpathy.github.io/2016/05/31/rl/