Attention

Previously on CENG501!

NEURAL MACHINE TRANSLATION
BY JOINTLY LEARNING TO ALIGN AND TRANSLATE

Dzmitry Bahdanau
Jacobs University Bremen, Germany

KyungHyun Cho Yoshua Bengio*
Université de Montréal

https://cloud.google.com/translate/automl/docs/evaluate#bleu
In a new model architecture, we define each conditional probability in Eq. (2) as:
\[ p(y_i | y_1, \ldots, y_{i-1}, x) = g(y_{i-1}, s_i, c_i), \]  
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 \( (h_{i1}, \ldots, h_{iT}) \) 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 \( h_{ij} \):
\[ c_i = \sum_{j=1}^{T} \alpha_{ij} h_{ij}. \]

The weight \( \alpha_{ij} \) of each annotation \( h_{ij} \) is computed by
\[ \alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T} \exp(e_{ik})}, \]
where
\[ e_{ij} = a(s_{i-1}, h_{ij}) \]
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 \( h_{ij} \) 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 Types

<table>
<thead>
<tr>
<th>Name</th>
<th>Alignment score function</th>
<th>Citation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Content-base attention</td>
<td>score(s_i, h_i) = \text{cosine}[s_i, h_i]</td>
<td>Graves2014</td>
</tr>
<tr>
<td>Additive(^*)</td>
<td>score(s_i, h_i) = \text{v}_a^\top \tanh(\text{W}_a[s_i; h_i])</td>
<td>Bahdanau2015</td>
</tr>
<tr>
<td>Location-Base</td>
<td>(\alpha_{i,j} = \text{softmax}(\text{W}_a s_i))</td>
<td>Luong2015</td>
</tr>
<tr>
<td></td>
<td>Note: This simplifies the softmax alignment to only depend on the target position.</td>
<td></td>
</tr>
<tr>
<td>General</td>
<td>score(s_i, h_i) = s_i^\top \text{W}_a h_i</td>
<td>Luong2015</td>
</tr>
<tr>
<td></td>
<td>where (\text{W}_a) is a trainable weight matrix in the attention layer.</td>
<td></td>
</tr>
<tr>
<td>Dot-Product</td>
<td>score(s_i, h_i) = s_i^\top h_i</td>
<td>Luong2015</td>
</tr>
<tr>
<td>Scaled Dot-Product(^\wedge)</td>
<td>score(s_i, h_i) = \frac{s_i^\top h_i}{\sqrt{n}}</td>
<td>Vaswani2017</td>
</tr>
<tr>
<td></td>
<td>Note: very similar to the dot-product attention except for a scaling factor; where (n) is the dimension of the source hidden state.</td>
<td></td>
</tr>
</tbody>
</table>

\(^*\) Referred to as “concat” in Luong, et al., 2015 and as “additive attention” in Vaswani, et al., 2017.

\(^\wedge\) It adds a scaling factor \(1/\sqrt{n}\), motivated by the concern when the input is large, the softmax function may have an extremely small gradient, hard for efficient learning.

# Attention Types

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
<th>Citation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Self-Attention(&amp;)</td>
<td>Relating different positions of the same input sequence. Theoretically the self-attention can adopt any score functions above, but just replace the target sequence with the same input sequence.</td>
<td>Cheng2016</td>
</tr>
<tr>
<td>Global/Soft</td>
<td>Attending to the entire input state space.</td>
<td>Xu2015</td>
</tr>
<tr>
<td>Local/Hard</td>
<td>Attending to the part of input state space; i.e. a patch of the input image.</td>
<td>Xu2015; Luong2015</td>
</tr>
</tbody>
</table>

Soft/hard attention

Fig. 7. “A woman is throwing a frisbee in a park.” (Image source: Fig. 6(b) in Xu et al. 2015)

Global/local attention

Fig. 8. Global vs local attention (Image source: Fig 2 & 3 in Luong, et al., 2015)

Attention: Transformer

- **Vanilla self 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!

https://jalammar.github.io/illustrated-transformer/
Fig. 17. The full model architecture of the transformer. (Image source: Fig 1 & 2 in Vaswani, et al., 2017.)

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).
GPT-3

• 175B parameters!
• Similar to BERT, a transformer-based model pretrained with masked language tasks.

Previously on CENG501!

Language Models are Few-Shot Learners

<table>
<thead>
<tr>
<th>Tom B. Brown</th>
<th>Benjamin Mann</th>
<th>Nick Ryder</th>
<th>Melanie Subhish</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jarred Kaplan</td>
<td>Prafulla Dhariwal</td>
<td>Arvind Neelakantan</td>
<td>Pranav Shyam</td>
</tr>
<tr>
<td>Amanda Askell</td>
<td>Sandhini Agarwal</td>
<td>Ariel Herbert-Voss</td>
<td>Gretchen Krueger</td>
</tr>
<tr>
<td>Rewon Child</td>
<td>Aditya Ramesh</td>
<td>Daniel M. Ziegler</td>
<td>Jeffrey Wu</td>
</tr>
<tr>
<td>Girish Sastry</td>
<td>Tom Henighan</td>
<td></td>
<td>Clemens Winter</td>
</tr>
</tbody>
</table>

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.
• Environmental & financial costs
• Require vast data
  • Not necessarily diverse
  • Includes bias
• Accountability/liability
• Stochastic Parrots

<table>
<thead>
<tr>
<th>Year</th>
<th>Model</th>
<th># of Parameters</th>
<th>Dataset Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>2019</td>
<td>BERT [39]</td>
<td>3.4E+08</td>
<td>16GB</td>
</tr>
<tr>
<td>2019</td>
<td>DistilBERT [113]</td>
<td>6.60E+07</td>
<td>16GB</td>
</tr>
<tr>
<td>2019</td>
<td>ALBERT [70]</td>
<td>2.23E+08</td>
<td>16GB</td>
</tr>
<tr>
<td>2019</td>
<td>XLNet (Large) [150]</td>
<td>3.40E+08</td>
<td>126GB</td>
</tr>
<tr>
<td>2020</td>
<td>ERNIE-GEN (Large) [145]</td>
<td>3.40E+08</td>
<td>16GB</td>
</tr>
<tr>
<td>2019</td>
<td>RoBERTa (Large) [74]</td>
<td>3.55E+08</td>
<td>161GB</td>
</tr>
<tr>
<td>2019</td>
<td>MegatronLM [122]</td>
<td>8.30E+09</td>
<td>174GB</td>
</tr>
<tr>
<td>2020</td>
<td>T5-11B [107]</td>
<td>1.10E+10</td>
<td>745GB</td>
</tr>
<tr>
<td>2020</td>
<td>T-NLG [112]</td>
<td>1.70E+10</td>
<td>174GB</td>
</tr>
<tr>
<td>2020</td>
<td>GPT-3 [25]</td>
<td>1.75E+11</td>
<td>570GB</td>
</tr>
<tr>
<td>2020</td>
<td>GShard [73]</td>
<td>6.00E+11</td>
<td>–</td>
</tr>
<tr>
<td>2021</td>
<td>Switch-C [43]</td>
<td>1.57E+12</td>
<td>745GB</td>
</tr>
</tbody>
</table>

Table 1: Overview of recent large language models
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.
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

---

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

Alex Graves  graveesa@google.com
Greg Wayne  gregwayne@google.com
Ivo Danihelka  danihelka@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 \( \mathbf{e}_t \in [0,1]^M \)
  \[ \tilde{M}_t(i) \leftarrow M_{t-1}(i)[1 - w_t(i)\mathbf{e}_t] \]
  \( \mathbf{1} \): vector of ones. Multiplication here is pointwise.

• Adding: Add an add vector \( \mathbf{a}_t \in [0,1]^M \):
  \[ M_t(i) \leftarrow \tilde{M}_t(i) + w_t(i)\mathbf{a}_t \]
Today

• Vision Transformers
• ConvNext
• Deep Representation Learning
• Deep Generative Modeling
• Deep Reinforcement Learning

<table>
<thead>
<tr>
<th>Week</th>
<th>Topics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Week 14 (10 June)</td>
<td>RNNs + Attention</td>
</tr>
<tr>
<td>Week 15 (17 June)</td>
<td>Recent Studies/Approaches</td>
</tr>
</tbody>
</table>
Administrative Issues

- Programming Assignment 2 (PA2):
  - Deadline: 5 June

- Programming Assignment 3 (PA3):
  - Deadline: 21 June

- Final exam:
  - Dates: 23-26 June.

- Projects:
  - Deadline: 5 July.

May

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>M</th>
<th>T</th>
<th>W</th>
<th>T</th>
<th>F</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>PA2</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
</tr>
<tr>
<td>PA2</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>29</td>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

June

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>M</th>
<th>T</th>
<th>W</th>
<th>T</th>
<th>F</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

July

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>M</th>
<th>T</th>
<th>W</th>
<th>T</th>
<th>F</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

August

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>M</th>
<th>T</th>
<th>W</th>
<th>T</th>
<th>F</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Spring 2022
From: Pashacoffee.com

17 June : Last day of classes
09-12 July: Bayram
12 July : Submitting grades
17 July : Incomplete grades
ViT: Vision Transformers

Figure 1: Model overview. We split an image into fixed-size patches, linearly embed each of them, add position embeddings, and feed the resulting sequence of vectors to a standard Transformer encoder. In order to perform classification, we use the standard approach of adding an extra learnable “classification token” to the sequence. The illustration of the Transformer encoder was inspired by Vaswani et al. (2017).
Swin Transformer: Hierarchical Vision Transformer using Shifted Windows

Ze Liu††  Yutong Lin††  Yue Cao*  Han Hu††  Yixuan Wei†
Zheng Zhang  Stephen Lin  Baining Guo
Microsoft Research Asia
{v-zeliul,v-yutlin,yuecao,hanhu,v-yixwe,zhe,stevelin,baising}@microsoft.com

Figure 2. An illustration of the \textit{shifted window} approach for computing self-attention in the proposed Swin Transformer architecture. In layer \textit{l} (left), a regular window partitioning scheme is adopted, and self-attention is computed within each window. In the next layer \textit{l + 1} (right), the window partitioning is shifted, resulting in new windows. The self-attention computation in the new windows crosses the boundaries of the previous windows in layer \textit{l}, providing connections among them.
# Swin Transformer V2: Scaling Up Capacity and Resolution

Ze Liu†, Han Hu‡, Yutong Lin, Zhuliang Yao, Zhenda Xie, Yixuan Wei, Jia Ning, Yue Cao, Zheng Zhang, Li Dong, Furu Wei, Baining Guo

Microsoft Research Asia

{v-zeliul,hanhu,t-yutonglin,t-zhuyao,t-zhxie,t-yixuanwei,v-jianing}@microsoft.com

{yuecao,zher,lidongl,fiwei,baingguo}@microsoft.com

## Table

<table>
<thead>
<tr>
<th>Method</th>
<th>param</th>
<th>pre-train images</th>
<th>pre-train length (#fm)</th>
<th>pre-train im size</th>
<th>fine-tune im size</th>
<th>ImageNet-1K-V1 top-1 acc</th>
<th>ImageNet-1K-V2 top-1 acc</th>
</tr>
</thead>
<tbody>
<tr>
<td>SwinV1-B</td>
<td>88M</td>
<td>IN-22K-14M</td>
<td>1.3B</td>
<td>224²</td>
<td>&lt;30¹</td>
<td>86.4</td>
<td>76.58</td>
</tr>
<tr>
<td>SwinV1-L</td>
<td>197M</td>
<td>IN-22K-14M</td>
<td>1.3B</td>
<td>224²</td>
<td>&lt;10¹</td>
<td>87.3</td>
<td>77.46</td>
</tr>
<tr>
<td>ViT-G [65]</td>
<td>1.8B</td>
<td>JFT-3B</td>
<td>164B</td>
<td>224²</td>
<td>&gt;30k</td>
<td>518²</td>
<td>90.45</td>
</tr>
<tr>
<td>V-MoE [44]</td>
<td>14.7B*</td>
<td>JFT-3B</td>
<td>-</td>
<td>224²</td>
<td>16.8k</td>
<td>518²</td>
<td>90.35</td>
</tr>
<tr>
<td>CoAtNet-7 [11]</td>
<td>2.44B</td>
<td>JFT-3B</td>
<td>-</td>
<td>224²</td>
<td>20.1k</td>
<td>512²</td>
<td>90.88</td>
</tr>
<tr>
<td>SwinV2-B</td>
<td>88M</td>
<td>IN-22K-14M</td>
<td>1.3B</td>
<td>192²</td>
<td>&lt;30¹</td>
<td>87.1</td>
<td>78.08</td>
</tr>
<tr>
<td>SwinV2-L</td>
<td>197M</td>
<td>IN-22K-14M</td>
<td>1.3B</td>
<td>192²</td>
<td>&lt;20¹</td>
<td>87.7</td>
<td>78.31</td>
</tr>
<tr>
<td>SwinV2-G</td>
<td>3.0B</td>
<td>IN-22K-ext-70M</td>
<td>3.5B</td>
<td>192²</td>
<td>&lt;0.5k</td>
<td>642²</td>
<td>90.17</td>
</tr>
</tbody>
</table>

## Diagram

![Diagram of Swin Transformer V1 and V2](image-url)
**SimMIM: A Simple Framework for Masked Image Modeling**

Zhenda Xie*, Zheng Zhang*, Yue Cao*, Yutong Lin, Jianmin Bao, Zhuliang Yao, Qi Dai, Han Hu*

Microsoft Research Asia

{t-zhxie, zhez, yuecao, t-yutonglin, jianmin.bao, t-zhuyao, qi.dai, hanhu}@microsoft.com

---

**Figure 1.** An illustration of our simple framework for masked language modeling, named *SimMIM*. It predicts raw pixel values of the randomly masked patches by a lightweight one-layer head, and performs learning using a simple $\ell_1$ loss.

**Figure 2.** Illustration of masking area generated by different masking strategies using a same mask ratio of 0.6: square masking [36], block-wise masking [1] apply on 16-sized patches, and our simple random masking strategy on different patch sizes (e.g., 4, 8, 16 and 32).
**A ConvNet for the 2020s**

Zhuang Liu$^{1,2,*}$  Hanzi Mao$^1$  Chao-Yuan Wu$^1$  Christoph Feichtenhofer$^1$  Trevor Darrell$^2$  Saining Xie$^{1,†}$

$^1$Facebook AI Research (FAIR)  $^2$UC Berkeley

Code: [https://github.com/facebookresearch/ConvNeXt](https://github.com/facebookresearch/ConvNeXt)

---

**Figure 1. ImageNet-1K classification results for ConvNets and vision Transformers.** Each bubble’s area is proportional to FLOPs of a variant in a model family. ImageNet-1K/22K models here take $224^2/384^2$ images respectively. ResNet and ViT results were obtained with improved training procedures over the original papers. We demonstrate that a standard ConvNet model can achieve the same level of scalability as hierarchical vision Transformers while being much simpler in design.

---

<table>
<thead>
<tr>
<th>Model</th>
<th>ImageNet-1K Acc.</th>
<th>ImageNet-22K Pre-trained</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResNet-50/200</td>
<td>78.6</td>
<td>82.0</td>
</tr>
<tr>
<td>Macro Design</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stage ratio</td>
<td>78.6</td>
<td>4.1</td>
</tr>
<tr>
<td>&quot;patchily&quot; stem</td>
<td>79.5</td>
<td>4.1</td>
</tr>
<tr>
<td>ResNeXt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>depth conv</td>
<td>79.6</td>
<td>4.5</td>
</tr>
<tr>
<td>width t</td>
<td>80.5</td>
<td>4.4</td>
</tr>
<tr>
<td>Inverted Bottleneck</td>
<td></td>
<td></td>
</tr>
<tr>
<td>inverting dims</td>
<td>80.6</td>
<td>5.3</td>
</tr>
<tr>
<td>Large Kernel</td>
<td></td>
<td></td>
</tr>
<tr>
<td>move 1 d. conv</td>
<td>79.9</td>
<td>4.1</td>
</tr>
<tr>
<td>kernel sz. → 5</td>
<td>80.4</td>
<td>4.1</td>
</tr>
<tr>
<td>kernel sz. → 7</td>
<td>80.6</td>
<td>4.2</td>
</tr>
<tr>
<td>kernel sz. → 9</td>
<td>80.6</td>
<td>4.2</td>
</tr>
<tr>
<td>kernel sz. → 11</td>
<td>80.5</td>
<td>4.3</td>
</tr>
<tr>
<td>Micro Design</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ReLU→QELU</td>
<td>80.5</td>
<td>4.2</td>
</tr>
<tr>
<td>fewer activations</td>
<td>81.3</td>
<td>4.2</td>
</tr>
<tr>
<td>fewer norms</td>
<td>81.4</td>
<td>4.2</td>
</tr>
<tr>
<td>IN→LN</td>
<td>81.5</td>
<td>4.2</td>
</tr>
<tr>
<td>sep. d.s. conv</td>
<td>82.0</td>
<td>4.5</td>
</tr>
<tr>
<td>ConvNeXt-T/B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ConvNeXt-B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Swin-T/B</td>
<td>81.3</td>
<td>4.5</td>
</tr>
</tbody>
</table>
Representation learning
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. \( \mathbf{W}_w(s + 1) = \mathbf{W}_w(s) + \Theta(u, v, s) \alpha(s) \mathbf{D}(t) - \mathbf{W}_w(s) \)
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. \( \mathbf{W}_w(s + 1) = \mathbf{W}_w(s) + \Theta(u, v, s) \alpha(s) \mathbf{D}(t) - \mathbf{W}_w(s) \)
3. Increase \( s \) and repeat from step 2 while \( s < \lambda \)
```

Non-parametric density estimation

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

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 \( \Sigma \) learned from nearest neighbors (local PCA)
- "Parametric" cousins:
  - Mixtures of Gaussian pancakes (Hinton et al. 95)
  - Mixtures of Factor Analyzers (Charmarre & 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:**
(Bengio and Larochelle, 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

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

$L(x,r)$

The Diabolo
Auto-Encoders (AE) for learning representations

hidden representation $\mathbf{h} = h(x) 

Encoder: $h$

Decoder: $g$

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

reconstruction error $L(x, r)$

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

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

Typical form

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

$\in \mathbb{R}^{d_h}$

**Encoder:**

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

**reconstruction**

$r = g(h(x))$

$= s_d(W'h + b_d)$

**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?

![Diagram showing distributed and not distributed representations](image)

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

\[
h = f_1(W_1 x)
\]

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

\[
y = f_2(W_2 f_1(W_1 x))
\]

• 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: \( x \in R^n \)
• Hidden layer: \( h \in R^m \)
  \[ h = f_1(W_1x) \]
• Output layer: \( y \in R^n \)
  \[ y = f_2(W_2f_1(W_1x)) \]

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

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

- 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:
  \[ \sum_{i} KL(\rho_0 \parallel \hat{\rho}_i) \]

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

- 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 \( \hat{\rho}_i \).
• 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:

\[-\sum_j [x_j \log z_j + (1 - x_j) \log (1 - z_j)]\]
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
\[ z = W^T x + b \]
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 $z$.
\[ \hat{x} = W z + b' \]
\[ E = \| x - \hat{x} \|_2^2 \]
3) 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.
\[ h_{(\Gamma)^c} = 0 \quad \text{where} \quad \Gamma = \text{supp}_{\alpha k}(h) \]
http://www.ericwilkinson.com/blog/2014/11/19/deep-learning-sparse-autoencoders
Denoising 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: \( h \)
- Decoder: \( g \)
- corrupted input \( \tilde{x} \)
- noise \( q(\tilde{x} | x) \)
- input \( x \)
- reconstruction \( r = g(h(x)) \)
- reconstruction error \( L(x, r) \)

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

Minimize:

\[
J_{\text{DAE}}(\theta) = \sum_{x \in D} E_{q(\tilde{x} | x)} [L(x, g(h(\tilde{x})))]
\]

(Pascal Vincent)
Denoising auto-encoder (DAE)

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

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

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

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

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

(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 vs. 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.
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.
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

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

- 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 $\tilde{x}|x = x + \epsilon$, $\epsilon \sim \mathcal{N}(0, \sigma^2 I)$

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

* It can be showed that

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

stochastic (SCAE)  analytic (CAE)
**Contractive Auto-Encoder (CAE)**

(Rifai, Vincent, Muller, Glorot, Bengio, ICML 2011)

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

Reconstruction error  
analytic contractive term

+ For training examples, encourages both:
  - small reconstruction error
  - representation insensitive to small variations around example
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: \( O(d_h d) \)

Compare to L2 weight decay:

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

Gradient backprop wrt parameters:

\( O(d_h d) \)

(Pascal Vincent)
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

\[ x(t) \]

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

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

http://www.scholarpedia.org/article/Slow_feature_analysis
Slow Feature Analysis (SFA)

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.
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)

Siamese Networks

Contrastive Loss (Chopra et al., 2005)

\[ y = 1 \text{ for “similar” pairs:} \]

\[
\mathcal{L}_{\text{cont}}(x_i, x_j, \theta) = \mathbb{1}[y_i = y_j] \|f_\theta(x_i) - f_\theta(x_j)\|_2^2 + \mathbb{1}[y_i \neq y_j] \max(0, \epsilon - \|f_\theta(x_i) - f_\theta(x_j)\|_2)^2
\]


https://lilianweng.github.io/lil-log/2021/05/31/contrastive-representation-learning.html#triplet-loss
Triplet Loss (Schroff et al., 2015)

\[
\mathcal{L}_{\text{triplet}}(\mathbf{x}, \mathbf{x}^+, \mathbf{x}^-) = \sum_{\mathbf{x} \in \mathcal{X}} \max \left( 0, \|f(\mathbf{x}) - f(\mathbf{x}^+)\|_2^2 - \|f(\mathbf{x}) - f(\mathbf{x}^-)\|_2^2 + \epsilon \right)
\]

Fig. 1. Illustration of triplet loss given one positive and one negative per anchor. (Image source: Schroff et al. 2015)

https://lilianweng.github.io/lil-log/2021/05/31/contrastive-representation-learning.html#triplet-loss

https://omoindrot.github.io/triplet-loss
Lifted Structure Loss (Song et al., 2015)

Let $D_{ij} = \|f(x_i) - f(x_j)\|_2$, a structured loss function is defined as

$$L_{\text{struct}} = \frac{1}{2|P|} \sum_{(i,j) \in P} \max(0, L_{\text{struct}}^{(ij)})^2$$

where $L_{\text{struct}}^{(ij)} = D_{ij} + \max \left( \max_{(i,k) \in N} e - D_{ik}, \max_{(j,l) \in N} e - D_{jl} \right)$

Fig. 2. Illustration compares contrastive loss, triplet loss and lifted structured loss. Red and blue edges connect similar and dissimilar sample pairs respectively. (Image source: Song et al. 2015)

https://lilianweng.github.io/lil-log/2021/05/31/contrastive-representation-learning.html
N-pair Loss (Sohn 2016)

\[
\mathcal{L}_{N\text{-pair}}(x, x^+, \{x_i^-\}_{i=1}^{N-1}) = \log \left( 1 + \sum_{i=1}^{N-1} \exp(f(x)^T f(x_i^-) - f(x)^T f(x^+)) \right)
\]

\[= -\log \frac{\exp(f(x)^T f(x^+))}{\exp(f(x)^T f(x^+)) + \sum_{i=1}^{N-1} \exp(f(x)^T f(x_i^-))}
\]

https://lilianweng.github.io/lil-log/2021/05/31/contrastive-representation-learning.html
Simple Contrastive Learning (Chen et al., 2020)

3) The contrastive learning loss is defined using cosine similarity $\text{sim}(\cdot, \cdot)$. Note that the loss operates on an extra projection layer of the representation $g(\cdot)$ rather than on the representation space directly. But only the representation $\mathbf{h}$ is used for downstream tasks.

$$
L_{\text{SimCLR}}^{(i,j)} = - \log \frac{\exp(\text{sim}(\mathbf{z}_i, \mathbf{z}_j)/\tau)}{\sum_{k=1}^{2N} \mathbb{1}_{[k \neq i]} \exp(\text{sim}(\mathbf{z}_i, \mathbf{z}_k)/\tau)}
$$

where $\mathbb{1}_{[k \neq i]}$ is an indicator function: 1 if $k \neq i$ 0 otherwise.

SimCLR needs a large batch size to incorporate enough negative samples to achieve good performance.

Fig. 6. A simple framework for contrastive learning of visual representations. (Image source: Chen et al, 2020)

https://lilianweng.github.io/lil-log/2021/05/31/contrastive-representation-learning.html
Momentum Contrast (MoCo – He et al., 2019)

\[ \mathcal{L}_{MoCo} = - \log \frac{\exp(q \cdot k^+ / \tau)}{\sum_{i=1}^{N} \exp(q \cdot k_i / \tau)} \]

\[ \theta_k \leftarrow m\theta_k + (1 - m)\theta_q \]

Fig. 12. Illustration of how Momentum Contrast (MoCo) learns visual representations. (Image source: He et al., 2019)

https://lilianweng.github.io/lil-log/2021/05/31/contrastive-representation-learning.html
Bootstrap Your Own Latent (BYOL – Grill et al., 2020)

- Does not use negative samples

Given an image $x$, the BYOL loss is constructed as follows:

- Create two augmented views: $v = t(x); v' = t'(x)$ with augmentations sampled $t \sim T, t' \sim T'$;
- Then they are encoded into representations, $y_\theta = f_\theta(v), y'_{\theta} = f_\theta(v')$;
- Then they are projected into latent variables, $z_\theta = g_\theta(y_{\theta}), z'_{\theta} = g_\theta(y'_{\theta})$;
- The online network outputs a prediction $q_\theta(z_{\theta})$;
- Both $q_\theta(z_{\theta})$ and $z'$ are L2-normalized, giving us $\tilde{q}_\theta(z_{\theta}) = q_\theta(z_{\theta}) \|q_\theta(z_{\theta})\| \text{ and } \tilde{z}' = z' / \|z'\|$;
- The loss $L^{\text{BYOL}}_\theta$ is MSE between L2-normalized prediction $\tilde{q}_\theta(z)$ and $\tilde{z}'$;
- The other symmetric loss $L^{\text{BYOL}}_\theta$ can be generated by switching $v'$ and $v$; that is, feeding $v'$ to online network and $v$ to target network.
- The final loss is $L^{\text{BYOL}}_\theta + L^{\text{BYOL}}_{\theta}$ and only parameters $\theta$ are optimized.

https://lilianweng.github.io/lil-log/2021/05/31/contrastive-representation-learning.html
Simple Siamese Representation Learning (SimSiam – Chen et al., 2020)

• “BYOL without momentum encoder”

**Algorithm 1 SimSiam Pseudocode, PyTorch-like**

```python
# f: backbone + projection mlp
# h: prediction mlp

for x in loader: # load a minibatch x with n samples
    x1, x2 = aug(x), aug(x) # random augmentation
    z1, z2 = f(x1), f(x2) # projections, n-by-d
    p1, p2 = h(z1), h(z2) # predictions, n-by-d

    L = D(p1, z2)/2 + D(p2, z1)/2 # loss

    L.backward() # back-propagate
    update(f, h) # SGD update

def D(p, z): # negative cosine similarity
    z = z.detach() # stop gradient
    p = normalize(p, dim=1) # l2-normalize
    z = normalize(z, dim=1) # l2-normalize
    return -(p*z).sum(dim=1).mean()
```

https://github.com/facebookresearch/simsiam
Resources on SSL

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

• Self-supervised representation learning:
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 $p_g(z)$.
    • Sample minibatch of $m$ examples \( \{x^{(1)}, \ldots, x^{(m)}\} \) from data generating distribution $p_{data}(x)$.
    • Update the discriminator by ascending its stochastic gradient:
      \[
      \nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[ \log D \left( x^{(i)} \right) + \log \left( 1 - D \left( G \left( z^{(i)} \right) \right) \right) \right].
      \n      \]
  
  end for
  • Sample minibatch of $m$ noise samples \( \{z^{(1)}, \ldots, z^{(m)}\} \) from noise prior $p_g(z)$.
  • 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.
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
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.
Cycle GAN

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 5753 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 $G: X \to Y$ and $F: Y \to X$, 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: $x \to G(x) \to F(G(x)) \approx x$, and (c) backward cycle-consistency loss: $y \to F(y) \to G(F(y)) \approx y$

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

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

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

\[ \text{AdaIN}(x_i, y) = y_{s,i} \frac{x_i - \mu(x_i)}{\sigma(x_i)} + y_{b,i} \]
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?
  • Intractability

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.
• \( X = \{x^{(i)}\}_{i=1}^N \): The dataset consisting of \( N \) i.i.d. samples of continuous or discrete variable \( x \).

• We assume that the data is generated by some random process, involving unobserved continuous random variable \( z \).
  1. A random \( z^{(i)} \) is generated from some prior distribution \( p_{\theta^*}(z) \).
  2. A value \( x^{(i)} \) is generated from a conditional distribution \( p_{\theta^*}(x \mid z) \).

• **Condition:** \( p_{\theta^*}(z) \) and \( p_{\theta^*}(x \mid z) \) are differentiable (\( \theta^* \) are parameters).

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

• **Problem:** \( p_{\theta}(x) \) and \( p_{\theta}(z \mid x) = p_{\theta}(x \mid z)p_{\theta}(z)/p_{\theta}(x) \) 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 $\epsilon$:
  $$\tilde{z} = g_\phi(\epsilon, x) \text{ with } \epsilon \sim p(\epsilon)$$

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:

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

   \[ \textit{Evidence Lower Bound (ELBO)} \]

• 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:
\[
ELBO_i(\phi) = -L_i(\theta, \phi)
\]

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

(reconstruction loss  regularizer)
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, \( \pi: S \rightarrow \mathcal{P}(A) \):
  \( \pi: S \rightarrow \mathcal{P}(A) \)

  - \( S \): The space of states.
  - \( A \): The space of actions.

- The “return” from a state is usually:

  \[
  R_t = \sum_{i=t}^{T} \gamma^{(t-i)} r(s_i, a_i)
  \]

  - \( r(s_i, a_i) \): 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:

  \[
  \mathbb{E}_{r_i, s_i \sim E, a_i \sim \pi}[R_1]
  \]

http://www.cs.ubc.ca/~murphyk/Bayes/pomdp.html
More formally

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

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

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

$$Q^\pi (s_t, a_t) = \mathbb{E}_{r_t, s_{t+1} \sim E} \left[ r(s_t, a_t) + \gamma \mathbb{E}_{a_{t+1} \sim \pi} \left[ Q^\pi (s_{t+1}, a_{t+1}) \right] \right]$$

http://www.cs.ubc.ca/~murphyk/Bayes/pomdp.html
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, $\max(0,x)$).
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 $i$,

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

---

**LETTER**

Human-level control through deep reinforcement learning

Vołodymyr 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†
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 $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
        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/