Optimized gradient descent algorithms

How to speed up stochastic gradient descent?

- accelerate ("momentum") towards minimum
- Automatically choose learning rate
- ...even different rates for different weights
Summary: a few gradient update methods


<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>adagrad</td>
<td>divide by RMS of all previous gradients</td>
</tr>
<tr>
<td>RMSprop</td>
<td>divide by RMS of last few previous gradients</td>
</tr>
<tr>
<td>adadelta</td>
<td>same, but multiply also by RMS of last few parameter updates</td>
</tr>
<tr>
<td>adam</td>
<td>divide running average of last few gradients by RMS of last few gradients (* with corrections during earliest steps)</td>
</tr>
</tbody>
</table>
\[ g_t = \frac{\partial C}{\partial \theta_t} \]

**Standard:**

\[ \dot{\theta} = -\eta g \]

or in discrete time steps

\[ \theta^{(t+1)} = \theta^{(t)} - \eta g^{(t)} \]

**Idea:** rescale according to estimate of typical size of \( g \)

\[ \Delta \theta_{t+1} = -\eta \frac{g^{(t)}}{\sqrt{\sum_{t'=t}^{t}[g^{(t')}]^2 + 3}} \]

"Root mean square (RMS)" of all previous gradients

\[ g_t = \frac{\partial C}{\partial \theta_t} \]

avoid division by zero
Solution: Only take RMS of "last few gradients"

\[ V^{(t)} = \gamma V^{(t-1)} + (1-\gamma)[g^{(t)}]^2 \]

decline term (e.g. \( \gamma \approx 0.9 \))

\[ \Rightarrow V^{(t)} = (1-\gamma) \sum_{t' \leq t} \gamma^{t-t'} [g^{(t')}]^2 \]

expontial decay of earlier contributions

\[ \Rightarrow \text{Roughly: contributions from } \sim \frac{1}{1-\gamma} \text{ last terms} \]

e.g. for time-independent \( g^{(t)} = g \):

\[ V^{(t)} = (1-\gamma) \left( \sum_{t' \leq t} \gamma^{t-t'} \right) g^2 = g^2 \]

\[ \frac{1}{1-\gamma} \]
\[ \Delta \theta_j^{(t)} = -\eta \frac{g_j^t}{\sqrt{V_j^{(t)} + \varepsilon}} \]
\[ \nabla(t) = \gamma \nabla(t-1) + (1-\gamma)[\Delta \theta(t)]^2 \]

\[ \Delta \theta(t) = -\frac{\nabla(t-1)}{\nabla(t)} g(t) \]

\text{no learning rate } \eta

\text{"has right dimensions" } \frac{\Delta \theta}{g} g = \Delta \theta
\[ m^{(t)} = \beta m^{(t-1)} + (1-\beta) g^{(t)} \]
\[ V^{(t)} \text{ like before} \]

Little problem: \( m^{(t)} \approx 0, V^{(t)} \approx 0 \) in first steps

\[ \Rightarrow \text{correct via} \]
\[ \hat{m}^{(t)} = \frac{m^{(t)}}{1-\beta^t} \]
\[ \hat{V}^{(t)} = \frac{V^{(t)}}{1-\gamma^t} \]

\[ \Rightarrow \text{set} \]
\[ \Delta \theta^{(t)} = -\eta \frac{\hat{m}^{(t)}}{\sqrt{\hat{V}^{(t)} + \varepsilon}} \]

\[ \beta = 0.9 \]
\[ \gamma = 0.999 \]
\[ \varepsilon = 10^{-8} \]
Recurrent neural networks

Networks “with memory”

Useful for analyzing time-evolution (time-series of data), for analyzing and translating sentences, for control/feedback (e.g. robotics or action games), and many other things.
Could use a convolutional network!

input

output

filter size = memory time
Long memories with convolutional nets are challenging:

- would need large filter sizes
- even then, would need to know required memory time beforehand
- can expand memory time efficiently by multi-layer network with subsampling (pooling), but this is still problematic for precise long-term memory

But: may be OK for some physics applications! (problems local in time, with short memory)
Memory

signal  no important signals  recall signal!

time
Solution: Recurrent Neural Networks (RNN)

Advantage: in principle this could give arbitrarily long memory!

Note: each circle may represent multiple neurons (i.e. a layer). Each arrow then represents all possible connections between those neurons.
Solution: Recurrent Neural Networks (RNN)

Note: the weights are not time-dependent, i.e. need to store only one set of weights (similar to convolutional net)
"Backpropagation through time"

"correct answer" known here
Long memories with recurrent networks are challenging, due to a feature of backpropagation: “Exploding gradients” / “Vanishing gradients”

Backpropagation through many layers (in a deep network) or through many time-steps (in a recurrent network):

Something like $\Delta_{t-1} = M_t \Delta_t$ (for the recurrent network case)

Depending on typical eigenvalues of the matrices $M$:
Long short-term memory (LSTM)

Why this name? “Long-term memory” would be the weights that are adapted during training and then stored forever. “Short-term memory” is the input-dependent memory we are talking about here. “Long short-term memory” tries to have long memory times in a robust way, for this short-term memory.

Sepp Hochreiter and Jürgen Schmidhuber, 1997

Main idea: determine read/write/delete operations of a memory cell via the network (through other neurons)

Most of the time, a memory neuron just sits there and is not used/changed!

![Diagram showing signal, no important signals, and recall signal over time]
LSTM: Forget gate (delete)

memory cell content

$C_{t-1}$  $C_t$

**keep:** $C_t = 1 \times C_{t-1}$

**delete:** $C_t = 0 \times C_{t-1}$
LSTM: Forget gate (delete)

Calculate “forget gate”:
\[ f = \sigma(W^{(f)}x_t + b^{(f)}) \]

(usually \( x, b, f \) are vectors, \( W \) the weight matrix)

Obtain new memory content:
\[ c_t = f \star c_{t-1} \]

NEW: for the first time, we are **multiplying** neuron values!
LSTM: Forget gate (delete)

The multiplication $*$ splits the error backpropagation into two branches.

Product rule:

$$
\frac{\partial f_j c_{t-1,j}}{\partial w_*} = \frac{\partial f_j}{\partial w_*} c_{t-1,j} + f_j \frac{\partial c_{t-1,j}}{\partial w_*}
$$

(Note: if time is not specified, we are referring to $t$)
LSTM: Forget gate (delete)
**LSTM:** Write new memory value

\[
\begin{align*}
    i &= \sigma(W^{(i)}x_t + b^{(i)}) \\
    \tilde{c}_t &= \tanh(W^{(c)}x_t + b^{(c)})
\end{align*}
\]

both delete and write together:

\[
c_t = f \ast c_{t-1} + i \ast \tilde{c}_t
\]
LSTM: Read (output) memory value

\[ h_t = o \ast \tanh(c_t) \]

\[ o = \sigma(W^{(o)} x_t + b^{(o)}) \]
LSTM: exploit previous memory output ‘h’

make f,i,o etc. at time t depend on output ‘h’ calculated in previous time step!

(otherwise: ‘h’ could only be used in higher layers, but not to control memory access in present layer)

\[ f = \sigma(W^{(f)}x_t + U^{(f)}h_{t-1} + b^{(f)}) \]

...and likewise for every other quantity!

Thus, result of readout can actually influence subsequent operations (e.g.: readout of some selected other memory cell!)

Sometimes, o is even made to depend on \( c_t \)
LSTM: backpropagation through time is OK

As long as memory content is not read or written, the backpropagation gradient is trivial:

\[ c_t = c_{t-1} = c_{t-2} = \ldots \]

\[ \frac{\partial c_t}{\partial w_*} = \frac{\partial c_{t-1}}{\partial w_*} = \frac{\partial c_{t-2}}{\partial w_*} = \ldots \]

(deviation vector multiplied by 1)

During those ‘silent’ time-intervals: No explosion or vanishing gradient!
Adding an LSTM layer with 10 memory cells:

Each of those cells has the full structure, with $f, i, o$ gates and the memory content $c$, and the output $h$.

```python
rnn.add(LSTM(10, return_sequences=True))
```

whether to return the full time sequence of outputs, or only the output at the final time
Two LSTM layers (input > LSTM > LSTM=output), taking an input of 3 neuron values for each time step and producing a time sequence with 2 neuron values for each time step.

```python
def init_memory_net():
    global rnn, batchsize, timesteps
    rnn = Sequential()
    # note: batch_input_shape is (batchsize,timesteps,data_dim)
    rnn.add(LSTM(5, batch_input_shape=(None, timesteps, 3), return_sequences=True))
    rnn.add(LSTM(2, return_sequences=True))
    rnn.compile(loss='mean_squared_error', optimizer='adam', metrics=['accuracy'])
```

![Diagram of two LSTM layers connecting input to output]
Example: A network for recall
(see code on website)

input time sequence

<table>
<thead>
<tr>
<th>Time</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0.4</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

tell      recall!

desired output time sequence

<table>
<thead>
<tr>
<th>Time</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0.4</td>
</tr>
</tbody>
</table>

tell      recall
Example: A network that counts down
(see code on website)

input time sequence

```
1 0 7 1
```
tell

desired output time sequence

```
0 1
```
7 steps

signal!

Output of the recall network, evolving during training (for a fixed input sequence)

Learning episode (batch of 20 for each episode)
Output of the countdown network, evolving during training (for a fixed input sequence)

Learning episode (batch of 20 for each episode)
input sequence

**THE_THEORY_OF_GE**

(characters in one-hot encoding)

desired output: predict next character

**HE_THEORY_OF_OGEN**

network will output probability for each possible character, at each time step

**ABCDEFGHIJKLMNOPQRSTUVWXYZ_**

(example for second time-step)
Character generation

Example by Andrej Karpathy

training on MBs of text

tyntd-iafhatawiaoihrdemot lytdws e ,tfti, astai f ogoh eoase rrranbyne 'nhthnee e plia tklrsgd t o idoe ns,smtt h ne etie h,hregtrs nigtike,aoaenns lng

"Tmont thithey" fomesscerliund
Keushey. Thom here
sheulke, anmerenith ol sivh I lalterthend Bleipile shuwy fil on
aseterlome
coaniogennnc Phe lism thond hon at. MeiDimorotion in ther thize."

we counter. He stutn co des. His stanted out one ofler that concossions and was to gearang reay Jotrets and with fre colt otf paitt thin wall. Which das stimn

Aftair fall unsuch that the hall for Prince Velzonski's that me of her hearly, and behs to so arwage fiving were to it beloge, pavu say falling misfort how, and Gogition is so overelical and ofter.

"Why do what that day," replied Natasha, and wishing to himself the fact the princess, Princess Mary was easier, fed in had oftened him. Pierre aking his soul came to the packs and drove up his father-in-law women.
Train a network that is eventually able to carry out sums or differences:

<table>
<thead>
<tr>
<th>input</th>
<th>3+5=??</th>
</tr>
</thead>
<tbody>
<tr>
<td>output</td>
<td>. . . . 08</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>input</th>
<th>7−5=??</th>
</tr>
</thead>
<tbody>
<tr>
<td>output</td>
<td>. . . . 02</td>
</tr>
</tbody>
</table>

How do you encode the input/output sequences? What happens when the result has two digits? etc.
simple one-hot encoding of words needs large vectors (and they do not carry any special meaning):

word2vec – reduction to vectors in much lower dimension, where similar words lie closer together:
The remainder of this paper is organized as follows. In Section 2, we discuss related work; Section 3 describes the recurrent neural network language model we used to obtain word vectors; Section 4 discusses the test sets; Section 5 describes our proposed vector offset method; Section 6 summarizes our experiments, and we conclude in Section 7.

2 Related Work

Distributed word representations have a long history, with early proposals including (Hinton, 1986; Pollack, 1990; Elman, 1991; Deerwester et al., 1990). More recently, neural network language models have been proposed for the classical language modeling task of predicting a probability distribution over the “next” word, given some preceding words. These models were first studied in the context of feed-forward networks (Bengio et al., 2003; Bengio et al., 2006), and later in the context of recurrent neural network models (Mikolov et al., 2010; Mikolov et al., 2011b). This early work demonstrated outstanding performance in terms of word-prediction, but also the need for more computationally efficient models. This has been addressed by subsequent work using hierarchical prediction (Morin and Bengio, 2005; Mnih and Hinton, 2009; Le et al., 2011; Mikolov et al., 2011b; Mikolov et al., 2011a). Also of note, the use of distributed topic representations has been studied in (Hinton and Salakhutdinov, 2006; Hinton and Salakhutdinov, 2010), and (Bordes et al., 2012) presents a semantically driven method for obtaining word representations.

3 Recurrent Neural Network Model

The word representations we study are learned by a recurrent neural network language model (Mikolov et al., 2010), as illustrated in Figure 1. This architecture consists of an input layer, a hidden layer with recurrent connections, plus the corresponding weight matrices. The input vector \( w(t) \) represents input word at time \( t \) encoded using 1-of-N coding, and the output layer \( y(t) \) produces a probability distribution over words. The hidden layer \( s(t) \) maintains a representation of the sentence history. The input vector \( w(t) \) and the output vector \( y(t) \) have dimensionality of the vocabulary. The values in the hidden and output layers are computed as follows:

\[
\begin{align*}
s(t) &= f(Uw(t) +Ws(t-1)), \\
y(t) &= g(Vs(t)),
\end{align*}
\]

Where \( f(z) = \frac{1}{1 + e^{-z}} \) is the sigmoid function and \( g(z_m) = \frac{e^{z_m}}{\sum_k e^{z_k}} \) is the softmax function.

In this framework, the word representations are found in the columns of \( U \), with each column representing a word. The RNN is trained with backpropagation to maximize the data log-likelihood under the model. The model itself has no knowledge of syntax or morphology or semantics. Remarkably, training such a purely lexical model to maximize likelihood will induce word representations with striking syntactic and semantic properties.
Predicting the probability of any word in the dictionary, given the context words (most recent word): very expensive!

Alternative:

Noise-contrastive estimation: provide a few noisy (wrong) examples, and train the model to predict that they are fake (but that the true one is correct)!
Two approaches:

“continuous bag of words”  Context words → word

“skip-gram”  word → context words

Example dataset:

the quick brown fox jumped over the lazy dog

<table>
<thead>
<tr>
<th>word</th>
<th>context words (here: just surrounding words)</th>
</tr>
</thead>
<tbody>
<tr>
<td>quick</td>
<td>the, brown</td>
</tr>
<tr>
<td>over</td>
<td>jumped, the</td>
</tr>
<tr>
<td>lazy</td>
<td>the, dog</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Word vectors: how to train them

Model tries to predict:

\[ P_\theta(w, h) \]

prob. that \( w \) is the correct word,
given the context word \( h \)

parameters of the model, i.e. weights, biases,
and entries of embedding vectors

\[ P_\theta(w, h) = \sigma(W_{jk}e_k(h) + b_j) \]

j: index for word \( w \) in dictionary
k: index in embedding vector [Einstein sum]
e(h): embedding vector for word \( h \)
W,b: weights, biases

At each time-step: go down the gradient of

\[ C^{(t)} = \ln P_\theta(w_t, h) + \sum_{\tilde{w}} \ln(1 - P_\theta(\tilde{w}, h)) \]

noisy examples
Word vectors encode meaning

Mikolov, Yih, Zweig 2013

car-cars \sim\ tree-trees

(subtracting the word vectors on each side yields approx. identical vectors)

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{word_vectors}
\caption{Graphical representation of word vectors for "car", "cars", "tree", and "trees".}
\end{figure}
Word vectors encode meaning

Mikolov, Yih, Zweig 2013

Diagram showing the relationship between words such as "man," "woman," "uncle," "aunt," "king," and "queen." The diagram illustrates how these words are represented in a vector space, with arrows indicating the semantic relationships between them.
Country and Capital Vectors Projected by PCA

Figure 2: Two-dimensional PCA projection of the 1000-dimensional Skip-gram vectors of countries and their capital cities. The figure illustrates the ability of the model to automatically organize concepts and learn implicit relationships between them, as during the training we did not provide any supervised information about what a capital city means.

2.3 Subsampling of Frequent Words

In very large corpora, the most frequent words can easily occur hundreds of millions of times (e.g., "in", "the", and "a"). Such words usually provide less information value than the rare words. For example, while the Skip-gram model benefits from observing the co-occurrences of "France" and "Paris", it benefits much less from observing the frequent co-occurrences of "France" and "the", as nearly every word co-occurs frequently within a sentence with "the". This idea can also be applied in the opposite direction; the vector representations of frequent words do not change significantly after training on several million examples.

To counter the imbalance between the rare and frequent words, we use a simple subsampling approach: each word $w_i$ in the training set is discarded with probability computed by the formula:

$$P(w_i) = 1 - \sqrt{t_f(w_i)}$$

Word vectors encode meaning

Mikolov et al. 2013 ”Distributed Representations of Words and Phrases and their Compositionality”
Word vectors in keras

Layer for mapping word indices (integer numbers representing position in a dictionary) to word vectors (of length EMBEDDING_DIM), for input sequences of some given length

```python
embedding_layer = Embedding(len(word_index) + 1,
    EMBEDDING_DIM,
    input_length=MAX_SEQUENCE_LENGTH)
```

Helper routines for converting actual text into a sequence of word indices. See especially:

- function/class
  - Tokenizer
  - pad_sequences
  - (and others)

Keras documentation
- Text Preprocessing
- Sequence Preprocessing

Search for “GloVe word embeddings”: 800 MB database pre-trained on a 2014 dump of the English Wikipedia, encoding 400k words in 100-dimensional vectors
Reinforcement Learning
Reinforcement learning

Self-driving cars, robotics:
Observe immediate environment & move

Games:
Observe board & place stone
Observe video screen & move player

Challenge: the “correct” action is not known!
Therefore: no supervised learning!

Reward will be rare (or decided only at end)
Use reinforcement learning:

Training a network to produce actions based on rare rewards (instead of being told the ‘correct’ action!)

Challenge: We could use the final reward to define a cost function, but we cannot know how the environment reacts to a proposed change of the actions that were taken!

(unless we have a model of the environment)
“State” = full map
“Action” = move
Reward e.g. based on how many “treasures” were collected
Policy Gradient

=REINFORCE (Williams 1992): The simplest model-free general reinforcement learning technique

Basic idea: Use probabilistic action choice. If the reward at the end turns out to be high, make all the actions in this sequence more likely (otherwise do the opposite)

This will also sometimes reinforce ‘bad’ actions, but since they occur more likely in trajectories with low reward, the net effect will still be to suppress them!
Probabilistic policy:

Probability to take action $a$, given the current state $s$:

$$\pi_{\theta}(a|s)$$

Parameters of the network

Environment: makes (possibly stochastic) transition to a new state $s'$, and possibly gives a reward $r$

Transition function $P(s'|s,a)$

Environment transition probabilities:
- Down: 0.1
- Up: 0.6
- Left: 0.2
- Right: 0.1
Probability for having a certain trajectory of actions and states:

\[ P_\theta(\tau) = \prod_t P(s_{t+1} | s_t, a_t) \pi_\theta(a_t | s_t) \]

trajectory: \( \tau = (a, s) \)

\( a = a_0, a_1, a_2, \ldots \)
\( s = s_1, s_2, \ldots \) (state 0 is fixed)

Expected overall reward: sum over all trajectories

\[ \bar{R} = E[R] = \sum_\tau P_\theta(\tau) R(\tau) \]

reward for this sequence (sum over individual rewards \( r \) for all times)

Try to maximize expected reward by changing parameters of policy:

\[ \frac{\partial \bar{R}}{\partial \theta} = ? \]
Policy Gradient

\[ \frac{\partial \bar{R}}{\partial \theta} = \sum_t \sum_{\tau} R(\tau) \frac{\partial \pi_\theta(a_t|s_t)}{\partial \theta} \frac{1}{\pi_\theta(a_t|s_t)} \Pi_{t'} P(s_{t'} + 1|s_{t'}, a_{t'}) \pi_\theta(a_{t'}|s_{t'}) \]

\[ \frac{\partial \ln \pi_\theta(a_t|s_t)}{\partial \theta} \]

Main formula of policy gradient method:

\[ \frac{\partial \bar{R}}{\partial \theta} = \sum_t E[R \frac{\partial \ln \pi_\theta(a_t|s_t)}{\partial \theta}] \]

Stochastic gradient descent:

\[ \Delta \theta = \eta \frac{\partial \bar{R}}{\partial \theta} \]

where \( E[...] \) is approximated via the value for one trajectory (or a batch)
Increase the probability of all action choices in the given sequence, depending on size of reward $R$. Even if $R > 0$ always, due to normalization of probabilities this will tend to suppress the action choices in sequences with lower-than-average rewards.

Abbreviation:

$$G_k = \frac{\partial \ln P_\theta(\tau)}{\partial \theta_k} = \sum_t \frac{\partial \ln \pi_\theta(a_t \mid s_t)}{\partial \theta_k}$$

$$\frac{\partial \bar{R}}{\partial \theta_k} = E[RG_k]$$
Policy Gradient: reward baseline

Challenge: fluctuations of estimate for reward gradient can be huge. Things improve if one subtracts a constant baseline from the reward.

\[
\frac{\partial \tilde{R}}{\partial \theta} = \sum_t E[(R - b) \frac{\partial \ln \pi_\theta(a_t|s_t)}{\partial \theta}]
\]

\[
= E[(R - b)G]
\]

This is the same as before. Proof:

\[
E[G_k] = \sum_\tau P_\theta(\tau) \frac{\partial \ln P_\theta(\tau)}{\partial \theta_k} = \frac{\partial}{\partial \theta_k} \sum_\tau P_\theta(\tau) = 0
\]

However, the variance of the fluctuating random variable \((R-b)G\) is different, and can be smaller (depending on the value of \(b\))!
Optimal baseline

\[ X_k = (R - b_k)G_k \]

\[ \text{Var}[X_k] = E[X_k^2] - E[X_k]^2 = \min \]

\[ \frac{\partial \text{Var}[X_k]}{\partial b_k} = 0 \]

\[ b_k = \frac{E[G_k^2R]}{E[G_k^2]} \]

\[ G_k = \frac{\partial \ln P_{\theta}(\tau)}{\partial \theta_k} \]

\[ \Delta \theta_k = -\eta E[G_k(R - b_k)] \]
Function/Image representation
Image classification
[Handwriting recognition]
Convolutional nets
Autoencoders
Visualization by dimensional reduction
Recurrent networks
Word vectors
Reinforcement learning
For more in-depth treatment, see David Silver’s course on reinforcement learning (University College London):

http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html
The simplest RL example ever

A random walk, where the probability to go “up” is determined by the policy, and where the reward is given by the final position (ideal strategy: always go up!)

(Note: this policy does not even depend on the current state)
The simplest RL example ever

A random walk, where the probability to go “up” is determined by the policy, and where the reward is given by the final position (ideal strategy: always go up!)

(Note: this policy does not even depend on the current state)

Policy

\[ \pi_\theta(\text{up}) = \frac{1}{1 + e^{-\theta}} \]

Reward

\[ R = x(T) \]

RL update

\[ \Delta \theta = \eta \sum_t \left< R \frac{\partial \ln \pi_\theta(a_t)}{\partial \theta} \right> \]

\[ a_t = \text{up or down} \]

\[ \frac{\partial \ln \pi_\theta(a_t)}{\partial \theta} = \pm e^{-\theta} \pi_\theta(a_t) = \pm (1 - \pi_\theta(a_t)) = 1 - \pi_\theta(\text{up}) \text{ for up} \]

\[ -\pi_\theta(\text{up}) \text{ for down} \]

\[ \sum_t \frac{\partial \ln \pi_\theta(a_t)}{\partial \theta} = N_{\text{up}} - N \pi_\theta(\text{up}) \]

\[ N= \text{number of time steps} \]
The simplest RL example ever

**reward** \[ R = x(T) = N_{\text{up}} - N_{\text{down}} = 2N_{\text{up}} - N \]

**RL update** \[ \Delta \theta = \eta \sum_t \left< R \frac{\partial \ln \pi_\theta(a_t)}{\partial \theta} \right> \]

\[ a_t = \text{up or down} \]

\[ \left< R \sum_t \frac{\partial \ln \pi_\theta(a_t)}{\partial \theta} \right> = 2 \left< \left( N_{\text{up}} - \frac{N}{2} \right) \left( N_{\text{up}} - \bar{N}_{\text{up}} \right) \right> \]

(general analytical expression for average update, rare)

Initially, when \[ \pi_\theta(\text{up}) = \frac{1}{2} \):

\[ \Delta \theta = 2\eta \left< \left( N_{\text{up}} - \frac{N}{2} \right)^2 \right> = 2\eta \text{Var}(N_{\text{up}}) = \eta \frac{N}{2} > 0 \]

(binomial distribution!)
The simplest RL example ever

In general:

\[
\left\langle R \sum_t \frac{\partial \ln \pi_\theta(a_t)}{\partial \theta} \right\rangle = 2 \left\langle \left( N_{up} - \frac{N}{2} \right) (N_{up} - \bar{N}_{up}) \right\rangle \\
= 2 \left\langle \left( N_{up} - \bar{N}_{up} \right) + \left( \bar{N}_{up} - \frac{N}{2} \right) \right\rangle (N_{up} - \bar{N}_{up}) \\
= 2\text{Var}N_{up} + 2(\bar{N}_{up} - \frac{N}{2}) \left\langle N_{up} - \bar{N}_{up} \right\rangle \\
= 2\text{Var}N_{up} = 2N\pi_\theta(up)(1 - \pi_\theta(up))
\] (general analytical expression for average update, fully simplified, extremely rare)
The simplest RL example ever

3 learning attempts
strong fluctuations!

(This plot for $N=100$ time steps in a trajectory; $\eta=0.001$)
Spread of the update step

\[ Y = N_{up} - \bar{N}_{up} \quad c = \bar{N}_{up} - N/2 \]

\[ X = (Y + c)Y \]

(Note: to get Var X, we need central moments of binomial distribution up to 4th moment)

(This plot for N=100)
Optimal baseline suppresses spread!

\[ Y = N_{\text{up}} - \bar{N}_{\text{up}} \quad c = \bar{N}_{\text{up}} - N/2 \quad X = (Y + c)Y \]

with optimal baseline:

\[ X' = (Y + c - b)Y \]

\[ b = \frac{\langle Y^2(Y + c) \rangle}{\langle Y^2 \rangle} \]

(This plot for \( N=100 \))
\( M = \text{number of update steps} \)

\[
\Delta X = \sum_{j=1}^{M} X_j
\]

\[
\langle \Delta X \rangle = M \langle X \rangle
\]

\[
\sqrt{\text{Var} \Delta X} = \sqrt{M} \sqrt{\text{Var} X}
\]

Relative spread

\[
\frac{\sqrt{\text{Var} \Delta X}}{\langle \Delta X \rangle} \sim \frac{1}{\sqrt{M}}
\]
Implement the RL update including the optimal baseline and run some stochastic learning attempts. Can you observe the improvement over the no-baseline results shown here?

Note: You do not need to simulate the individual random walk trajectories, just exploit the binomial distribution.
The second-simplest RL example

actions: move or stay  "walker"  "target site"

reward=number of time steps on target

See code on website: “SimpleRL_WalkerTarget”
RL in keras: categorical cross-entropy trick

output = action probabilities (softmax)
\[ \pi_\theta(a|s) \]

\[ a=0 \quad a=1 \quad a=2 \]

input = state

\[ C = - \sum_a P(a) \ln \pi_\theta(a|s) \]

categorical cross-entropy
distr. from net
desired distribution

Set
\[ P(a) = R \]
for \( a \) = action that was taken

\[ P(a) = 0 \]
for all other actions \( a \)

\[ \Delta \theta = -\eta \frac{\partial C}{\partial \theta} \]

implements policy gradient
Among the major board games, "Go" was not yet played on a superhuman level by any program (very large state space on a 19x19 board!)

alpha-Go beat the world’s best player in 2017
"alpha-Go"

First: try to learn from human expert players

Sampled state-action pairs \((s, a)\), using stochastic gradient ascent to maximize the likelihood of the human move \(a\) selected in state \(s\)

\[ \Delta \sigma \propto \frac{\partial \log p_{\sigma}(a | s)}{\partial \sigma} \]

We trained a 13-layer policy network, which we call the SL policy network, from 30 million positions from the KGS Go Server. The net-

Silver et al., “Mastering the game of Go with deep neural networks and tree search” (Google Deepmind team), Nature, January 2016
Second: use policy gradient RL on games played against previous versions of the program to the current policy. We use a reward function $r(s)$ that is zero for all non-terminal time steps $t < T$. The outcome $z_t = \pm r(s_T)$ is the terminal reward at the end of the game from the perspective of the current player at time step $t$: $+1$ for winning and $-1$ for losing. Weights are then updated at each time step $t$ by stochastic gradient ascent in the direction that maximizes expected outcome $^{25}$

$$
\Delta \rho \propto \frac{\partial \log p_\rho (a_t | s_t)}{\partial \rho} z_t
$$

Silver et al., “Mastering the game of Go with deep neural networks and tree search” (Google Deepmind team), Nature, January 2016
Silver et al., “Mastering the game of Go with deep neural networks and tree search” (Google Deepmind team), Nature, January 2016
"alpha-Go"

Silver et al., “Mastering the game of Go with deep neural networks and tree search” (Google Deepmind team), Nature, January 2016
An alternative to the policy gradient approach

Introduce a quality function $Q$ that predicts the future reward for a given state $s$ and a given action $a$. **Deterministic policy**: just select the action with the largest $Q$!
player & possible actions

Q maximal
Q-learning

Introduce a quality function $Q$ that predicts the future reward for a given state $s$ and a given action $a$. **Deterministic policy**: just select the action with the largest $Q$!

$$Q(s_t, a_t) = E[R_t | s_t, a_t]$$  \hspace{1cm} (assuming future steps to follow the policy!)

“Discounted” future reward:

$$R_t = \sum_{t'=t}^{T} r_{t'} \gamma^{t'-t}$$

Reward at time step $t$:

$$r_t$$

Discount factor:

$$0 < \gamma \leq 1$$

Note: The ‘value’ of a state is $V(s) = \max_a Q(s, a)$

How do we obtain $Q$?
Bellmann equation: (from optimal control theory)

\[ Q(s_t, a_t) = E[r_t + \gamma \max_a Q(s_{t+1}, a) | s_t, a_t] \]

In practice, we do not know the Q function yet, so we cannot directly use the Bellmann equation. However, the following update rule has the correct Q function as a fixed point:

\[ Q^{\text{new}}(s_t, a_t) = Q^{\text{old}}(s_t, a_t) + \alpha (r_t + \gamma \max_a Q^{\text{old}}(s_{t+1}, a) - Q^{\text{old}}(s_t, a_t)) \]

small (<1) update factor

will be zero, once we have converged to the correct Q

If we use a neural network to calculate Q, it will be trained to yield the “new” value in each step.
$Q(a=\text{up}, s)$
$Q(a=\text{up}, s)$
$Q(a=\text{up},s)$
Initially, $Q$ is arbitrary. It will be bad to follow this $Q$ all the time. Therefore, introduce probability $\epsilon$ of random action (“exploration”).

Follow $Q$: “exploitation”

Do something random (new): “exploration”

“$\epsilon$-greedy”

Reduce this randomness later!
Example: Learning to play Atari Video Games

“Human-level control through deep reinforcement learning”, Mnih et al., Nature, February 2015

last four 84x84 pixel images as input [=state]
motion as output [=action]
Example: Learning to play Atari Video Games

“Human-level control through deep reinforcement learning”, Mnih et al., Nature, February 2015
“Human-level control through deep reinforcement learning”, Mnih et al., Nature, February 2015

Example: Learning to play Atari Video Games

t-SNE visualization of last hidden layer
Function/Image representation
Image classification
[Handwriting recognition]
Convolutional nets
Autoencoders
Visualization by dimensional reduction
Recurrent networks
Word vectors
Reinforcement learning
Connections to physics
Neural networks with stochastic transitions, and with some energy functional similar to spin models in physics; e.g. as described by Hopfield and others starting from the 80s.
**Goal:** Use a neural network to generate previously unseen examples, according to the probability distribution of training samples

One example already mentioned in these lectures: generating new random (but kind-of reasonable) text after seeing lots of it

Example: Generate new images after looking at many, generate handwritten text

The solution will exploit the connection between neural networks and the statistical physics of spin models!
Boltzmann-Gibbs distribution

Probabilities of states of a physical system, in thermal equilibrium?

\[ P(s) = \frac{1}{Z} e^{-\frac{E(s)}{k_B T}} \]

probability for state \( s \), in thermal equilibrium

\[ Z = \sum_{s'} e^{-\frac{E(s')}{k_B T}} \]

\( Z \) for normalization: “partition function”

Problem: for a many-body system, exponentially many states (for example \( 2^N \) spin states). Cannot go through all of them!

energy high: less likely

energy low: more likely
Monte Carlo approach

Place system in some state, make stochastic transitions to other states (with prescribed transition probabilities)
Monte Carlo approach

Time evolution of ensemble?

\[
\Delta P(s) = \sum_{s'} P(s \leftarrow s') P(s') - P(s' \leftarrow s) P(s)
\]

\(P(s)\) = probability to find the system in state \(s\) (or: fraction of ensemble in this state)

change in one time-step
At long times: stable steady state distribution

If we have “detailed balance”, i.e. if there exists a distribution \( P(s) \), such that for any pair of states:

\[
\frac{P(s \leftarrow s')} {P(s' \leftarrow s)} = \frac{P(s)} {P(s')}
\]

then \( P(s) \) is the long-time distribution!
Monte Carlo approach

Monte Carlo for thermal equilibrium: choose transition probabilities such that \( P(s) \) will be the Boltzmann distribution!

\[
P(s' \leftarrow s) P(s' \leftarrow s') = e^{\frac{E(s') - E(s)}{k_B T}}
\]

example Metropolis algorithm: pick random spin, calculate energy change for spin flip. Do the flip if it lowers the energy. If the energy increases, only flip with probability \( \exp(-\Delta E/k_B T) \)
The sequence of visited states forms a so-called “Markov chain”

Markov = transitions without memory
Restricted Boltzmann Machine

“hidden” units $\mathbf{h}$

“visible” units $\mathbf{v}$

Each “unit” is like a spin (or a bit) that can be 0 or 1
Restricted Boltzmann Machine

“hidden” units $h$

“visible” units $v$

Each “unit” is like a spin (or a bit) that can be 0 or 1

Define “energy” (we will set $k_B T = 1$)

$$E(v, h) = - \sum_{i \in \text{visible}} a_i v_i - \sum_{j \in \text{hidden}} b_j h_j - \sum_{i,j} v_i h_j w_{ij}$$

“restricted”: no coupling $v$-$v$ or $h$-$h$  

$w$: couplings (weights)

Restricted Boltzmann Machine

“hidden” units $h$

“visible” units $v$

Each “unit” is like a spin (or a bit) that can be 0 or 1

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

$$P(v) = \sum_h P(v, h)$$

Goal: adapt weights (and biases), such that the probability distribution of a set of training examples is approximately reproduced by $P(v)$

$$P(v) \approx P_0(v) \quad \text{from training samples}$$
Restricted Boltzmann Machine

“hidden” units $\mathbf{h}$

“visible” units $\mathbf{v}$

Each “unit” is like a spin (or a bit) that can be 0 or 1

Interpretation: the ‘hidden units’ represent categories of data (e.g. “dog+white+big”)
Instead of the full state \( s=(v,h) \): Consider alternating transitions between \( v \) and \( h \) states

Set:

\[
P(h \leftarrow v) = P(h|v) = \frac{P(v, h)}{P(v)}
\]

\[
P(v \leftarrow h) = P(v|h) = \frac{P(v, h)}{P(h)}
\]

These transition probabilities fulfill detailed balance!

\[
\frac{P(h \leftarrow v)}{P(v \leftarrow h)} = \frac{P(h)}{P(v)}
\]

Thus: \( P(v) \) [and \( P(h) \)] are the steady-state distributions!
Building a Markov chain

\[ ZP(v) = \sum_h e^{-E(v,h)} = \sum_h e^{\sum_i a_i v_i + \sum_j b_j h_j + \sum_{i,j} v_i h_j w_{ij}} \]

\[ = e^{\sum_i a_i v_i} \prod_j (1 + e^{z_j}) \]

with: \( z_j = b_j + \sum_i v_i w_{ij} \)

where we used: \( e^{\sum_j X_j} = \prod_j e^{X_j} \)

\[ \sum \ldots = \sum_{h_0=0,1} \sum_{h_1=0,1} \sum_{h_2=0,1} \ldots \]

Therefore:

\[ P(h|v) = \frac{e^{-E(v,h)}}{ZP(v)} = \prod_j \frac{e^{z_j h_j}}{1 + e^{z_j}} \]

Product of probabilities! All the \( h_j \) are independently distributed, with probabilities:

\[ P(h_j = 1|v) = \frac{e^{z_j}}{1 + e^{z_j}} = \sigma(z_j) \]

\[ P(h_j = 0|v) = 1 - \sigma(z_j) \]

sigmoid
Building a Markov chain

Given some visible-units state vector \(v\), calculate the probabilities

\[
P(h_j = 1|v) = \frac{e^{z_j}}{1 + e^{z_j}} = \sigma(z_j)
\]

Then assign 1 or 0, according to these probabilities, to obtain the new hidden state vector \(h\)

Similarly, go from \(h\) to a new \(v'\), using:

\[
P(v'_i = 1|h) = \sigma(z'_i)
\]

\[
z'_i = a_i + \sum_j w_{ij} h_j
\]
Updating the weights

Goal: adapt weights (and biases), such that the probability distribution of a set of training examples is approximately reproduced by \( P(v) \)

\[
P(v) \approx P_0(v) \quad \text{from training samples}
\]

Minimize the categorical cross-entropy

\[
C = - \sum_v P_0(v) \ln P(v)
\]

But now (unlike earlier examples), there are exponentially many values for \( v \), so we cannot simply have a network output \( P(v) \) for all \( v \). Still, let us take the derivative of \( C \) with respect to the weights \( w \)!
Updating the weights

\[ C = - \sum_v P_0(v) \ln P(v) \]

\[
\frac{\partial}{\partial w_{ij}} \ln P(v) = \frac{\frac{\partial}{\partial w_{ij}} \sum_h P(v, h)}{\sum_h P(v, h)}
\]

\[ Z = \sum_{v', h'} e^{-E(v', h')} \]

\[
= \frac{\sum_h v_i h'_j e^{-E(v, h)}}{\sum_h e^{-E(v, h)}} - \frac{\sum_{v', h'} v'_i h'_j e^{-E(v', h')}}{Z}
\]

overall:

\[
\sum_v P_0(v) \frac{\partial}{\partial w_{ij}} \ln P(v) = \sum_{v, h} v_i h_j P(h|v) P_0(v) - \sum_{v', h'} v'_i h'_j P(v', h')
\]
Updating the weights

$$\sum_v P_0(v) \frac{\partial}{\partial w_{ij}} \ln P(v) = \sum_{v, h} v_i h_j P(h|v) P_0(v) - \sum_{v', h'} v'_i h'_j P(h'|v') P(v')$$

easy: draw one training sample $v$, then do one Markov chain step from $v$ to $h$; average over all samples $v$

hard: need to average over the correct distribution $P(v)$ belonging to the Boltzmann machine!
Updating the weights

\[
\sum_v P_0(v) \frac{\partial}{\partial w_{ij}} \ln P(v) = \sum_{v,h} v_i h_j P(h|v) P_0(v) - \sum_{v',h'} v_i' h_j' P(h'|v') P(v')
\]

Could obtain \(P(v)\) by running the Markov chain for really long times! Very expensive!

Rough approximation, used in practice: Just take \(v',h'\) from the second pair of the chain! [For better approx.: can take a pair further down the chain]

\[
\Delta w_{ij} = \eta(\langle v_i h_j \rangle - \langle v_i' h_j' \rangle)
\]

(averaged over a batch of training samples \(v\) starting the chain)
Updating the weights

$$\Delta w_{ij} = \eta (\langle v_i h_j \rangle - \langle v_i' h_j' \rangle)$$
(averaged over a batch of training samples v starting the chain)

“Contrastive Divergence” (CD) algorithm by G. Hinton

Note: At least we can claim that $P_0(v) = P(v)$ would be a fixed point of this update rule, since then the two averages on the right-hand-side yield identical results. Of course, usually the restricted Boltzmann machine will not be able to reach this point, since it cannot represent arbitrary $P(v)$.

$$\Delta a_i = \eta (\langle v_i \rangle - \langle v_i' \rangle)$$
$$\Delta b_j = \eta (\langle h_j \rangle - \langle h_j' \rangle)$$
Restricted Boltzmann Machine for MNIST

example from http://deeplearning.net/tutorial/rbm.html

Each column: a different, independent Markov chain
Restricted Boltzmann Machine for MNIST

element from http://deeplearning.net/tutorial/rbm.html

The learned weights for the 100 hidden units
RBM as a starting point

First train RBM, then connect hidden layer to some output layer for supervised learning of classification.

Idea: RBM provides unsupervised learning of important features in the training set (pre-training).
Deep belief networks

Stack RBMs: First train a simple RBM, then use its hidden units as input to another RBM, and so on.

Afterwards, fine-tune weights, e.g. by supervised learning.
Try to solve a quantum many-body problem (quantum spin model) using the following **variational ansatz** for the wave function amplitudes:

\[
\Psi(S) = \sum e^{\sum_j a_j \sigma_j^z + \sum_i b_i h_i + \sum_{ij} h_{ij} \sigma_j^z W_{ij}}
\]

\[
S = (\sigma_1^z, \sigma_2^z, \ldots, \sigma_N^z)
\]

one basis state in the many-body Hilbert space

\[
\sigma_j^z = \pm 1
\]

\[
h_i = \pm 1
\]

This is exactly (proportional to) the RBM representation for P(v) [with v=S]!
Application to Quantum Physics

“hidden” units \( h \)

“visible” units \( v \)

= spins of quantum model

Minimize the energy

\[
\langle \Psi | \hat{H} | \Psi \rangle
\]

\[
\langle \Psi | \Psi \rangle
\]

by adapting the weights \( W \) and biases \( a \) and \( b \)!

[requires additional Monte Carlo simulation, to obtain a stochastic sampling of the gradient with respect to these parameters]

For example: sample probabilities by using Metropolis algorithm, with transition probabilities

\[
P(S' \leftarrow S) = \min(1, \frac{|\Psi(S')|^2}{|\Psi(S)|^2})
\]
Exploit translational invariance (like in convolutional nets); weights are “filters” (convolutional kernels)
Find updates on

http://machine-learning-for-physicists.org