Deep Learning with Torch

Tristan Cazenave

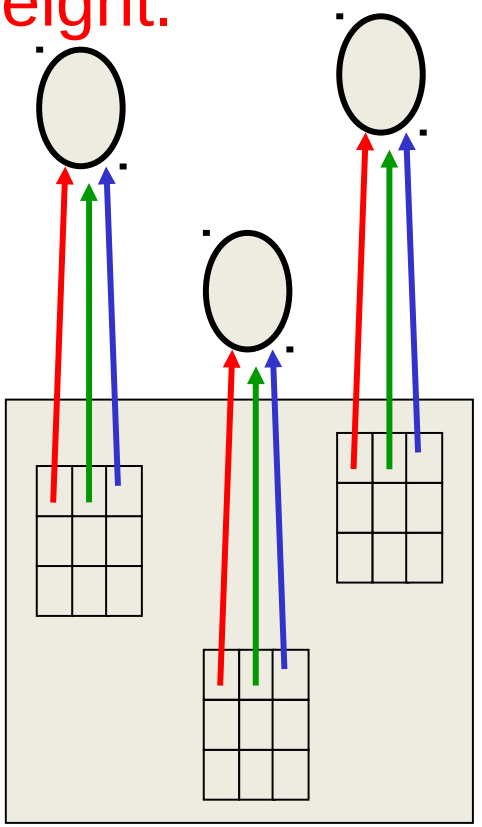
Tristan.Cazenave@dauphine.fr

Deep Learning

The replicated feature approach
(currently the dominant approach for neural networks)

- Use many different copies of the same feature detector with different positions.
  - Could also replicate across scale and orientation (tricky and expensive)
  - Replication greatly reduces the number of free parameters to be learned.
- Use several different feature types, each with its own map of replicated detectors.
  - Allows each patch of image to be represented in several ways.

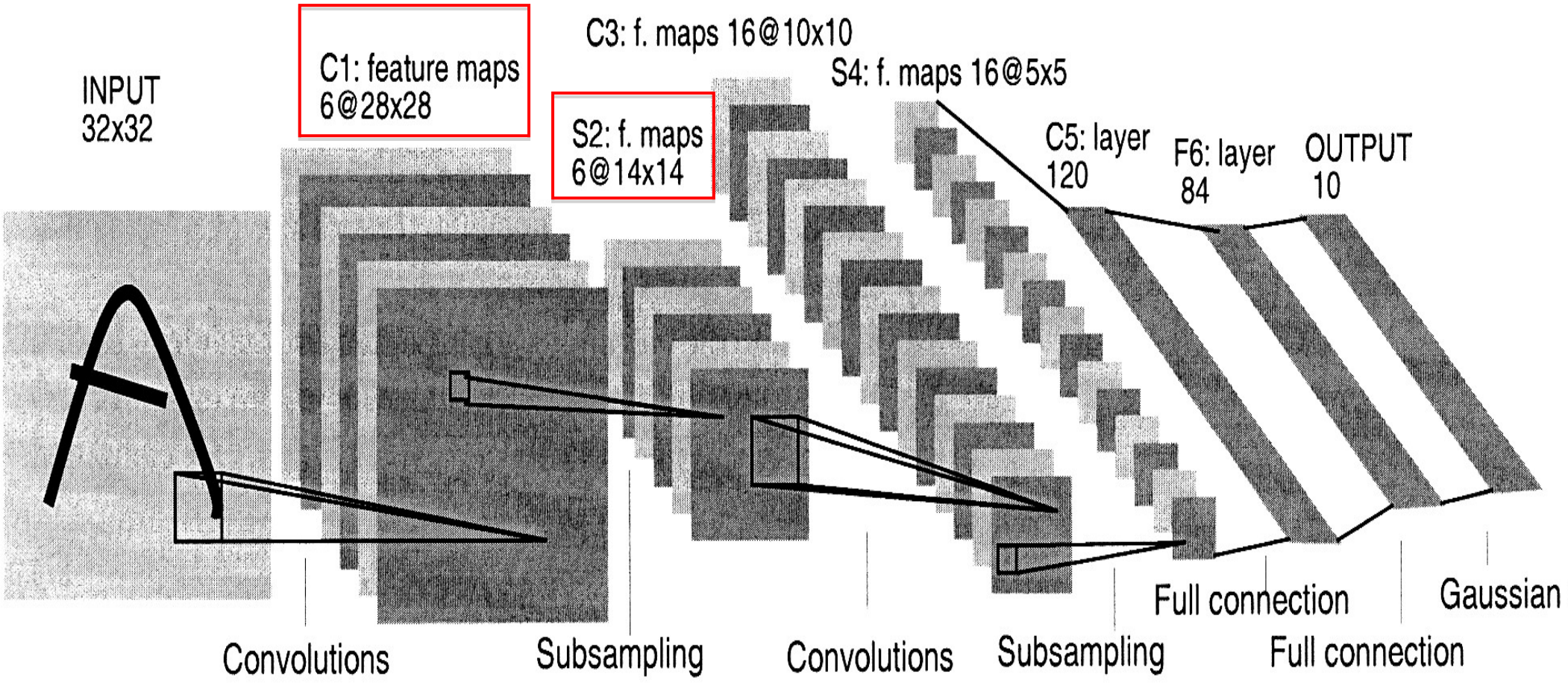
The red connections all have the same weight.

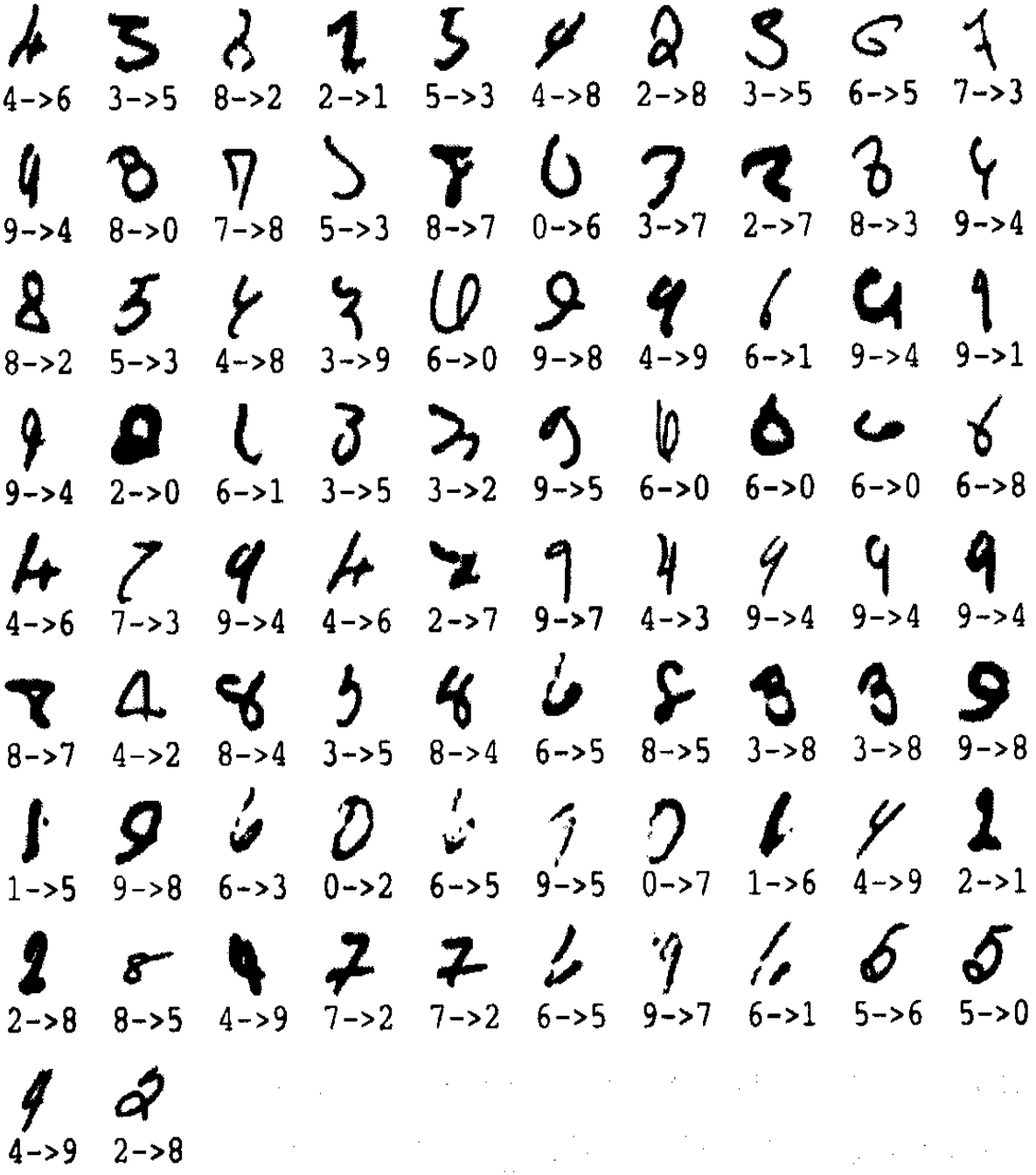


Le Net

- Yann LeCun and his collaborators developed a really good recognizer for handwritten digits by using backpropagation in a feedforward net with:
  - Many hidden layers
  - Many maps of replicated units in each layer.
  - Pooling of the outputs of nearby replicated units.
  - A wide net that can cope with several characters at once even if they overlap.
  - A clever way of training a complete system, not just a recognizer.
- This net was used for reading ~10% of the checks in North America.
- Look the impressive demos of LENET at http://yann.lecun.com

The architecture of LeNet5





The 82 errors made by LeNet5

Notice that most of the errors are cases that people find quite easy.

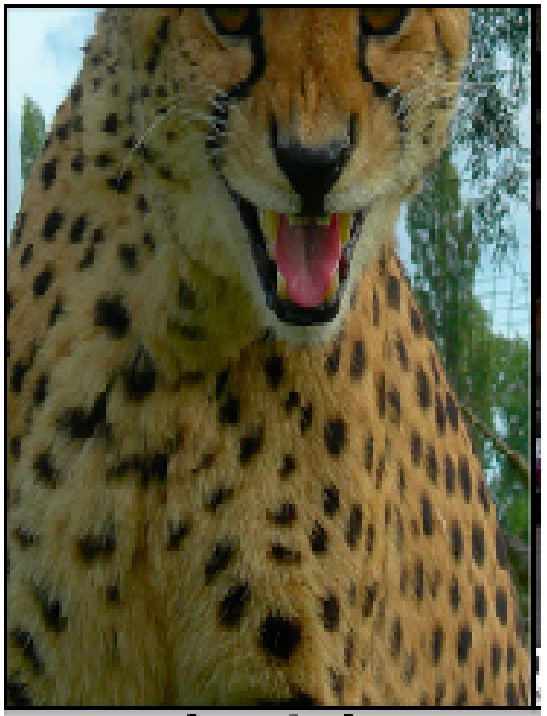
The human error rate is probably 20 to 30 errors but nobody has had the patience to measure it.

The ILSVRC-2012 competition on ImageNet

- The dataset has 1.2 million high-resolution training images.
- The classification task:
  - Get the “correct” class in your top 5 bets. There are 1000 classes.
- The localization task:
  - For each bet, put a box around the object. Your box must have at least 50% overlap with the correct box.

- Some of the best existing computer vision methods were tried on this dataset by leading computer vision groups from Oxford, INRIA, XRCE, …
  - Computer vision systems use complicated multi-stage systems.
  - The early stages are typically hand-tuned by optimizing a few parameters.

Examples from the test set (with the network's guesses)



cheetah

cheetah
leopard
snow leopard
Egyptian cat

bullet train

bullet train
passenger car
subway train
electric locomotive

hand glass

scissors
hand glass
frying pan
stethoscope

- University of Toronto (Alex Krizhevsky)
  - 16.4%
  - 34.1%

Error rates on the ILSVRC-2012 competition

	classification	classification &localization
• University of Tokyo	• 26.1%	53.6%
• Oxford University Computer Vision Group	• 26.9%	50.0%
• INRIA (French national research institute in CS) + XRCE (Xerox Research Center Europe)	• 27.0%	
• University of Amsterdam	• 29.5%	

A neural network for ImageNet

- Alex Krizhevsky (NIPS 2012) developed a very deep convolutional neural net of the type pioneered by Yann Le Cun. Its architecture was:
  - 7 hidden layers not counting some max pooling layers.
  - The early layers were convolutional.
  - The last two layers were globally connected.

- The activation functions were:
  - –Rectified linear units in every hidden layer. These train much faster and are more expressive than logistic units.
  - –Competitive normalization to suppress hidden activities when nearby units have stronger activities. This helps with variations in intensity.

Tricks that significantly improve generalization

- Train on random 224x224 patches from the 256x256 images to get more data. Also use left-right reflections of the images.
  - At test time, combine the opinions from ten different patches: The four 224x224 corner patches plus the central 224x224 patch plus the reflections of those five patches.
- Use “dropout” to regularize the weights in the globally connected layers (which contain most of the parameters).
  - Dropout means that half of the hidden units in a layer are randomly removed for each training example.
  - This stops hidden units from relying too much on other hidden units.

Some more examples of how well the deep net works for object recognition.

The hardware required for Alex's net

- He uses a very efficient implementation of convolutional nets on two Nvidia GTX 580 Graphics Processor Units (over 1000 fast little cores)
  - GPUs are very good for matrix-matrix multiplies.
  - GPUs have very high bandwidth to memory.
  - This allows him to train the network in a week.
  - It also makes it quick to combine results from 10 patches at test time.
- We can spread a network over many cores if we can communicate the states fast enough.
- As cores get cheaper and datasets get bigger, big neural nets will improve faster than old-fashioned (*i.e.* pre Oct 2012) computer vision systems.

Applications of Deep Learning

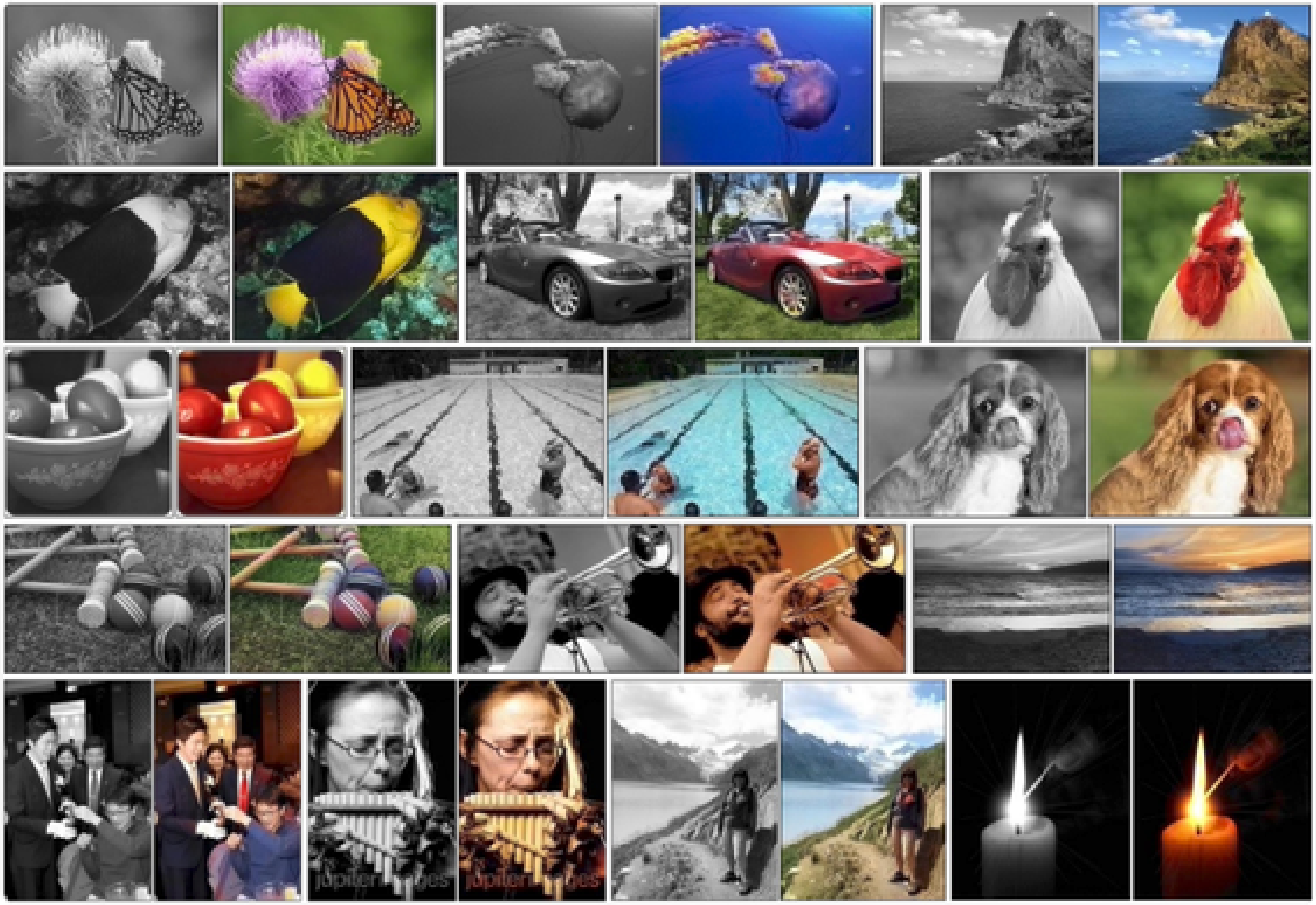
Applications of Deep Learning

- Colorization of Black and White Images.
- Adding Sounds To Silent Movies.
- Machine Translation.
- Object Classification in Photographs.
- Handwriting Generation.
- Character Text Generation.
- Image Caption Generation.
- Game Playing.

Colorization of Black and White Images

- Image colorization is the problem of adding color to black and white photographs.
- Deep learning can be used to use the objects and their context within the photograph to color the image, much like a human operator might approach the problem.
- The approach involves the use of very large convolutional neural networks and supervised layers that recreate the image with the addition of color.

Colorization of Black and White Images



Adding Sounds To Silent Movies

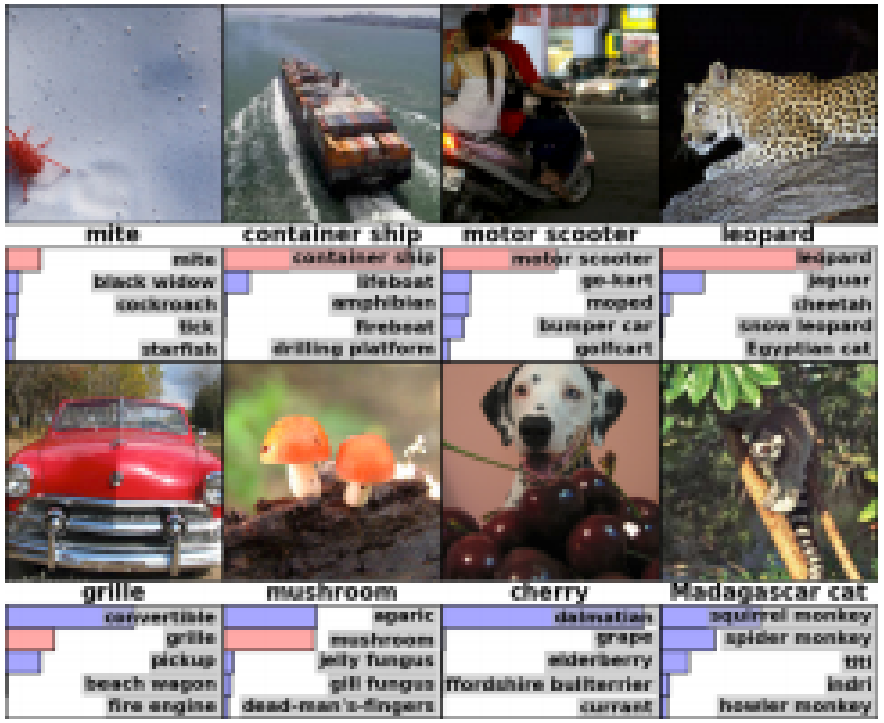
- The system must synthesize sounds to match a silent video.
- A deep learning model associates the video frames with a database of pre-rerecorded sounds in order to select a sound to play that best matches what is happening in the scene.
- The model uses convolutional neural networks and recurrent neural networks.

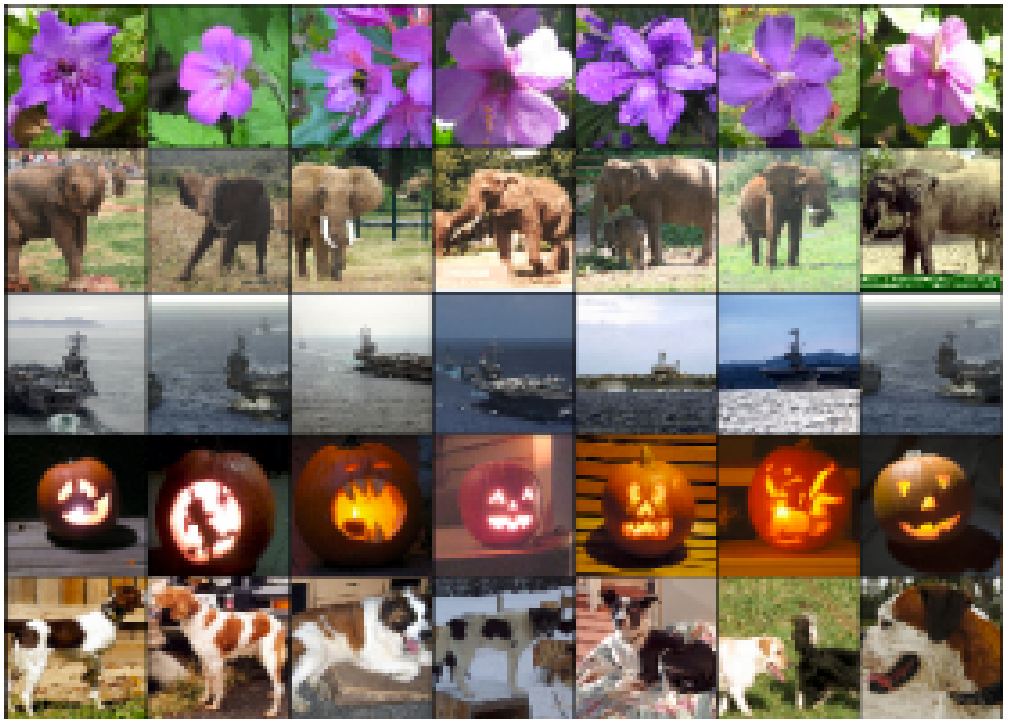
Machine Translation

- Given words, phrase or sentence in one language, automatically translate it into another language.
- Automatic machine translation has been around for a long time, but deep learning is achieving top results in automatic translation of text.
- Text translation can be performed without any preprocessing of the sequence, allowing the algorithm to learn the dependencies between words and their mapping to a new language.
- Stacked networks of large recurrent neural networks are used to perform this translation.

Object Classification in Photographs

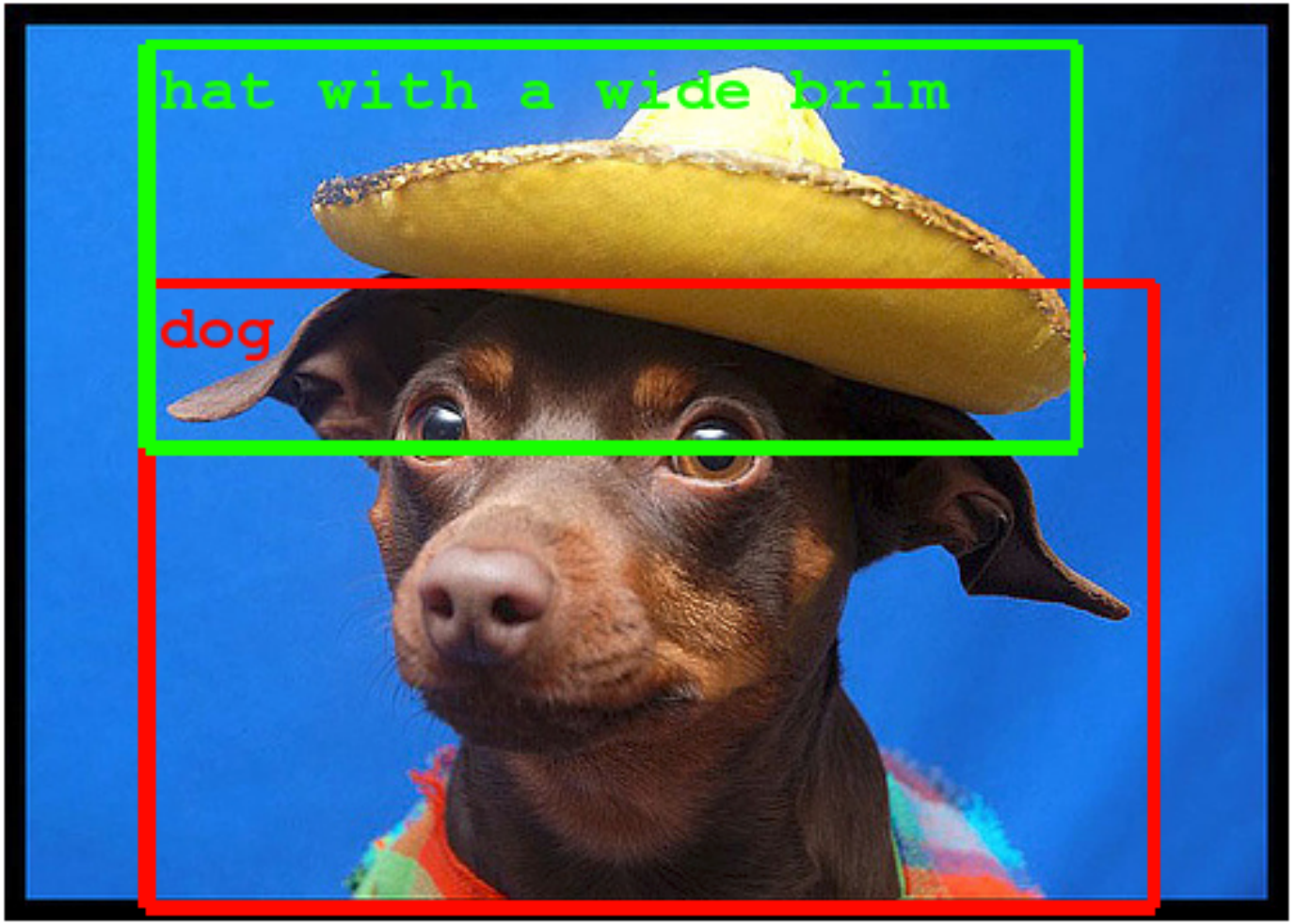
- ImageNet classification problem : AlexNet





Object Classification in Photographs

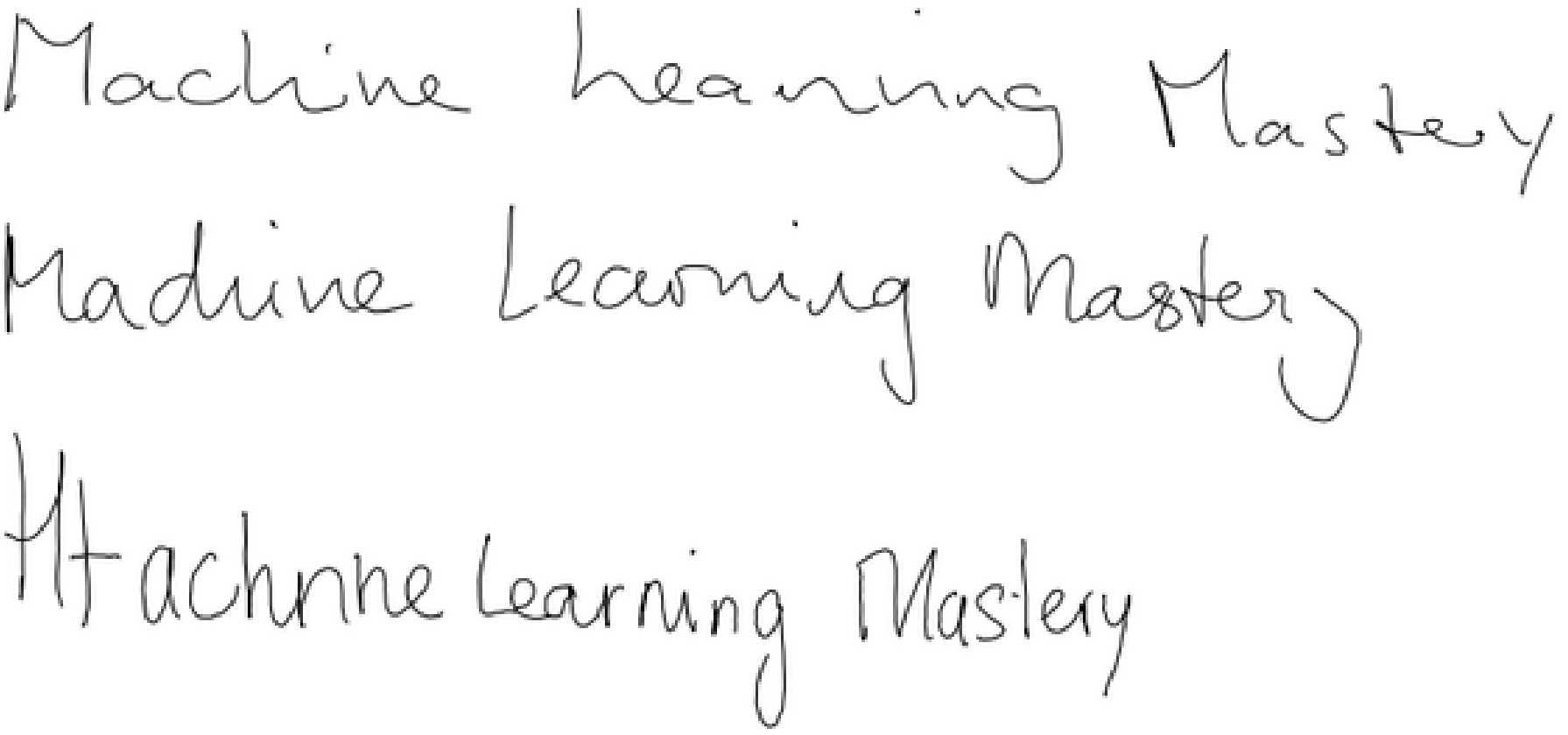
- Object detection :



Handwriting Generation

- Given a corpus of handwriting examples, generate new handwriting for a given word or phrase.
- The handwriting is provided as a sequence of coordinates used by a pen when the handwriting samples were created.
- From this corpus the relationship between the pen movement and the letters is learned and new examples can be generated ad hoc.

Handwriting Generation



Text Generation

- A corpus of text is learned and from this model new text is generated, word-by-word or character-by-character.
- The model is capable of learning how to spell, punctuate, form sentences and even capture the style of the text in the corpus.
- Large recurrent neural networks are used to learn the relationship between items in the sequences of input strings and then generate text.
- Recurrent neural networks are demonstrating great success on this problem using a character-based model, generating one character at time.

Text Generation

- Andrej Karpathy provides many examples in his popular blog post on the topic including:
- Paul Graham essays
- Shakespeare
- Wikipedia articles (including the markup)
- Algebraic Geometry (with LaTeX markup)
- Linux Source Code
- Baby Names

Text Generation

```
PANDARUS:
Alas, I think he shall be come approached and the day
When little srain would be attain'd into being never fed,
And who is but a chain and subjects of his death,
I should not sleep.

Second Senator:
They are away this miseries, produced upon my soul,
Breaking and strongly should be buried, when I perish
The earth and thoughts of many states.

DUKE VINCENTIO:
Well, your wit is in the care of side and that.

Second Lord:
They would be ruled after this chamber, and
my fair nues begun out of the fact, to be conveyed,
Whose noble souls I'll have the heart of the wars.

Clown:
Come, sir, I will make did behold your worship.

VIOLA:
I'll drink it.
```

Image Caption Generation

- Given an image the system must generate a caption that describes the contents of the image.
- Once you can detect objects in photographs and generate labels for those objects, the next step is to turn those labels into a coherent sentence description.
- Use of large convolutional neural networks for the object detection in the photographs and then a recurrent neural network to turn the labels into a coherent sentence.

Image Caption Generation

"man in black shirt is playing guitar."

"construction worker in orange safety vest is working on road."

"two young girls are playing with lego toy."

"girl in pink dress is jumping in air."

"black and white dog jumps over bar."

"young girl in pink shirt is swinging on swing."

Game Playing

- DeepMind : Deep Reinforcement Learning
- Atari Games :

  Better than humans with only the pixels and the score as inputs.
- Computer Go :

  Better than the best humans.

Backpropagation

A spectrum of machine learning tasks

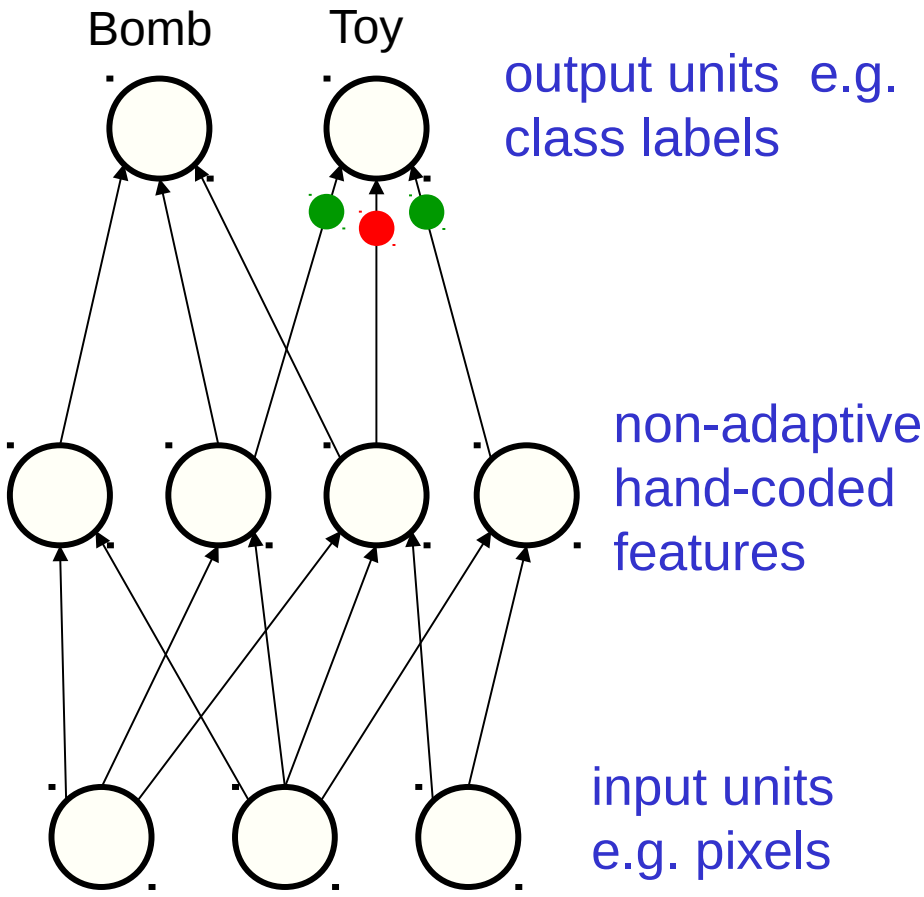
Typical Statistics----------Artificial Intelligence

- Low-dimensional data (e.g. less than 100 dimensions)
- Lots of noise in the data
- There is not much structure in the data, and what structure there is, can be represented by a fairly simple model.
- The main problem is distinguishing true structure from noise.

- High-dimensional data (e.g. more than 100 dimensions)
- The noise is not sufficient to obscure the structure in the data if we process it right.
- There is a huge amount of structure in the data, but the structure is too complicated to be represented by a simple model.
- The main problem is figuring out a way to represent the complicated structure so that it can be learned.

Historical background:
First generation neural networks

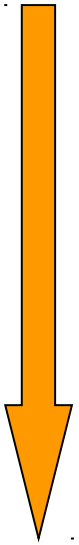
- Perceptrons (~1960) used a layer of hand-coded features and tried to recognize objects by learning how to weight these features.
  - There was a neat learning algorithm for adjusting the weights.
  - But perceptrons are fundamentally limited in what they can learn to do.

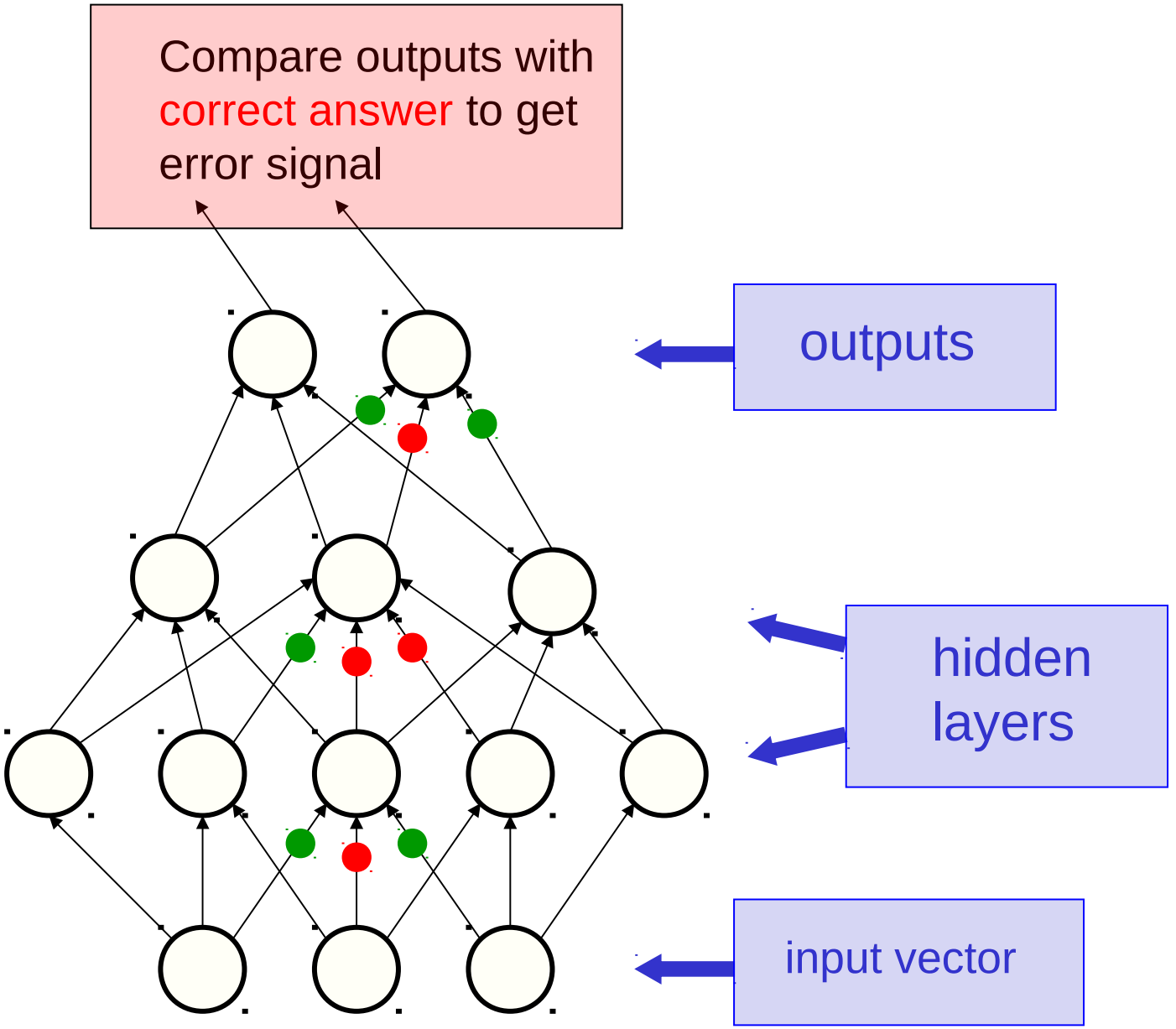


Sketch of a typical perceptron from the 1960's

Second generation neural networks (~1985)

Back-propagate
error signal
to get derivatives
for learning





A temporary digression

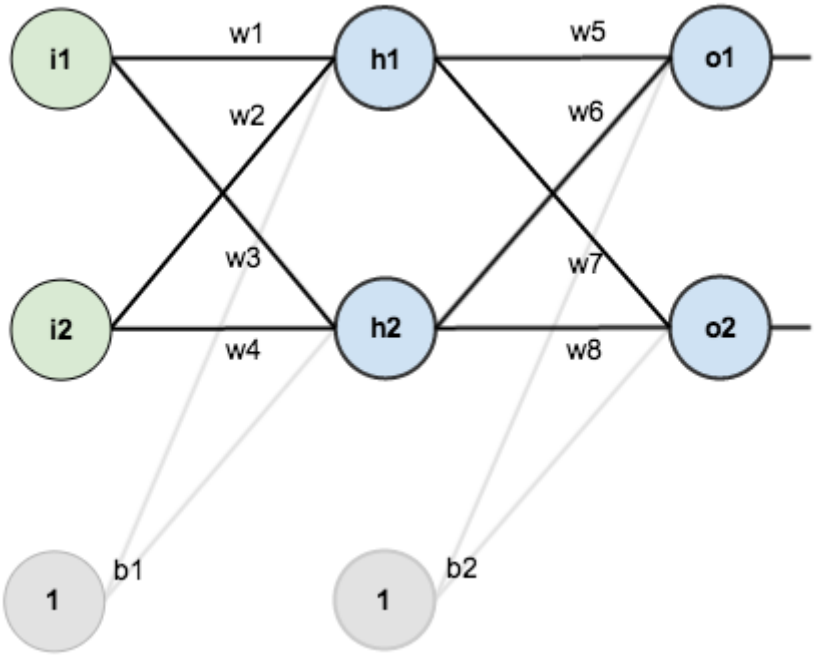
- Vapnik and his co-workers developed a very clever type of perceptron called a Support Vector Machine.
  - Instead of hand-coding the layer of non-adaptive features, each training example is used to create a new feature using a fixed recipe.
    - The feature computes how similar a test example is to that training example.
  - Then a clever optimization technique is used to select the best subset of the features and to decide how to weight each feature when classifying a test case.
    - But its just a perceptron and has all the same limitations.
- In the 1990's, many researchers abandoned neural networks with multiple adaptive hidden layers because Support Vector Machines worked better.

Backpropagation

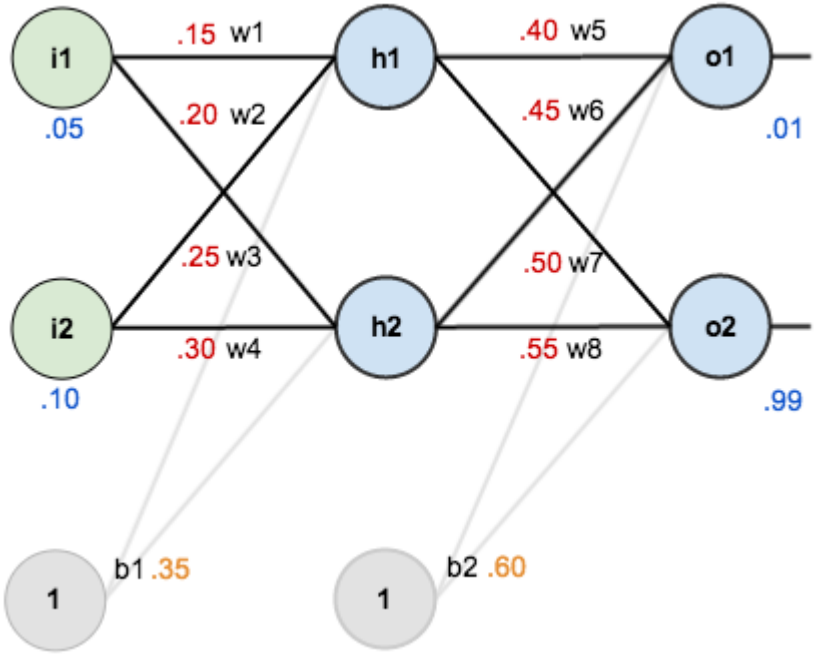
- We are going to explain backpropagation on a simple example.

- We take as example a network with two inputs, two outputs and two hidden neurons .

Backpropagation



Backpropagation



Backpropagation

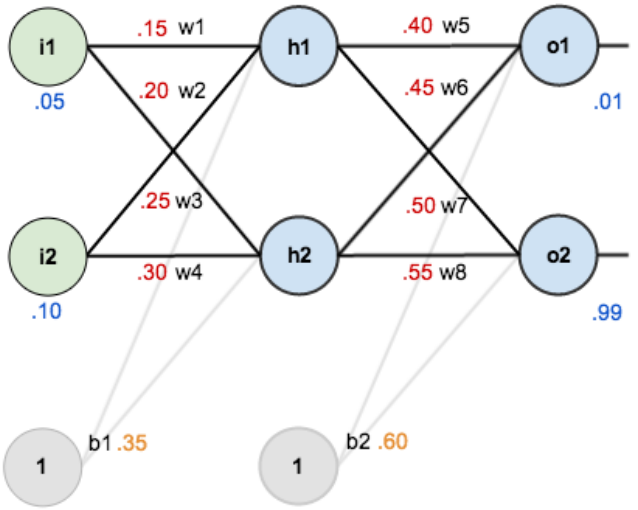
- The goal of backpropagation is to optimize the weights so that the neural network can learn how to correctly map arbitrary inputs to outputs.

- We are going to work with a single training set: given inputs 0.05 and 0.10, we want the neural network to output 0.01 and 0.99.

The Forward Pass

- We figure out the total net input to each hidden layer neuron.
- Squash the total net input using an activation function (here we use the sigmoid function).
- Repeat the process with the output layer neurons.

The Forward Pass



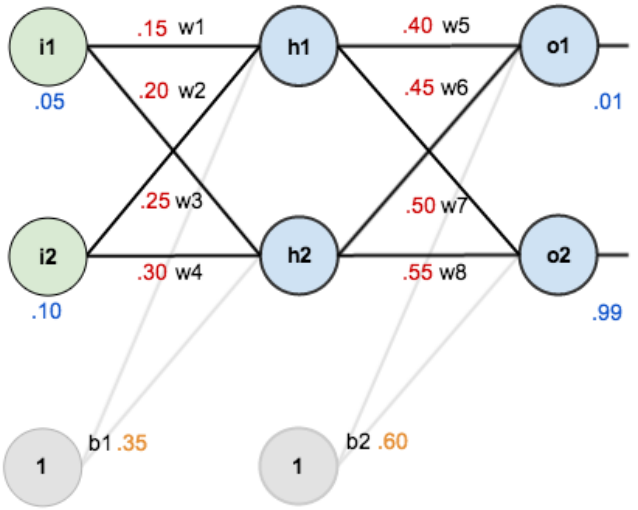
- Here's how we calculate the total net input for h1:

  $net_{h1} = w1 * i1 + w2 * i2 + b1 * 1$

  $net_{h1} = 0.15 * 0.05 + 0.2 * 0.1 + 0.35 * 1 = 0.3775$
- We then squash it using the sigmoid function to get the output of h1 :

  $out_{h1} = 1/(1+e^{-net_{h1}}) = 1/(1+e^{-0.3775}) = 0.593269992$
- Carrying out the same process for h2 we get : $out_{h2} = 0.596884378$

The Forward Pass



- We repeat this process for the output layer neurons, using the output from the hidden layer neurons as inputs.
- Here's the output for o1: $net_{o1} = w5 * out_{h1} + w6 * out_{h2} + b2 * 1$

  $net_{o1} = 0.4 * 0.593269992 + 0.45 * 0.596884378 + 0.6 * 1 = 1.105905967$

  $out_{o1} = 1/(1+e^{-net_{o1}}) = 1/(1+e^{-1.105905967}) = 0.75136507$
- And carrying out the same process for o2 we get: $out_{o2} = 0.772928465$

The Error

- We can now calculate the error for each output neuron using the squared error function and sum them to get the total error:

  $E_{total} = \Sigma\ 1/2\ (target - output)^2$

- For example, the target output for o1 is 0.01 but the neural network output 0.75136507, therefore its error is:

  $E_{o1} = 1/2\ (target_{o1} - out_{o1})^2 = 1/2\ (0.01 - 0.75136507)^2 = 0.274811083$

- Repeating this process for o2 we get: $E_{o2} = 0.023560026$
- The total error for the neural network is the sum of these errors:

  $E_{total} = E_{o1} + E_{o2} = 0.274811083 + 0.023560026 = 0.298371109$

The Backwards Pass

- Our goal with backpropagation is to update each of the weights in the network so that they cause the actual output to be closer the target output, thereby minimizing the error for each output neuron and the network as a whole.

Output Layer

- Consider w5.
- We want to know how much a change in w5 affects the total error:

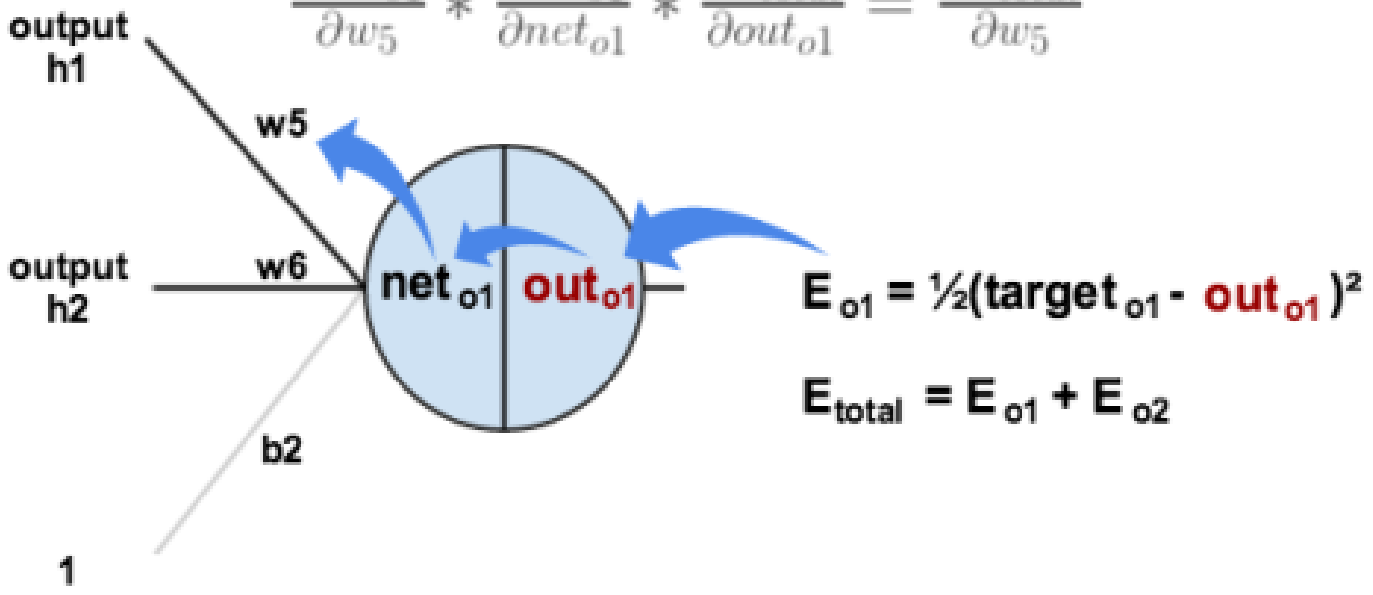
  $\delta E_{total} / \delta w5$

- $\delta E_{total} / \delta w5$ is read as “the partial derivative of E_{total} with respect to w5“.
- You can also say “the gradient with respect to w5“.
- By applying the chain rule we know that:

  $\delta E_{total} / \delta w5 = \delta E_{total} / \delta out_{o1} * \delta out_{o1} / \delta net_{o1} * \delta net_{o1} / \delta w5$

Output Layer

$$\frac{\partial net_{o1}}{\partial w_5} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial E_{total}}{\partial out_{o1}} = \frac{\partial E_{total}}{\partial w_5}$$



Output Layer

- We need to figure out each piece in this equation.
- First, how much does the total error change with respect to the output?

$$E_{total} = 1/2\ (target_{o1} - out_{o1})^2 + 1/2\ (target_{o2} - out_{o2})^2$$

$$\delta E_{total} / \delta out_{o1} = -(target_{o1} - out_{o1})$$

$$\delta E_{total} / \delta out_{o1} = -(0.01 - 0.75136507) = 0.74136507$$

Output Layer

- Next, how much does the output of o1 change with respect to its total net input?

  $out_{o1} = 1/(1+e^{(-net_{o1})})$

  $\delta out_{o1} / \delta net_{o1} = out_{o1}(1 - out_{o1}) = 0.75136507(1 - 0.75136507) = 0.186815602$

- Finally, how much does the total net input of o1 change with respect to w5?

  $net_{o1} = w5 * out_{h1} + w6 * out_{h2} + b2 * 1$

  $\delta net_{o1} / \delta w5 = out_{h1} = 0.593269992$

Output Layer

- Putting it all together:

$$\delta E_{total} / \delta w5 = \delta E_{total} / \delta out_{o1} * \delta out_{o1} / \delta net_{o1} * \delta net_{o1} / \delta w5$$

$$\delta E_{total} / \delta w5 = 0.74136507 * 0.186815602 * 0.593269992 = 0.082167041$$

- To decrease the error, we then subtract this value from the current weight (optionally multiplied by some learning rate, η, which we'll set to 0.5):

$$w5 = w5 - \eta * \delta E_{total} / \delta w5 = 0.4 - 0.5 * 0.082167041 = 0.35891648$$

Output Layer

- We can repeat this process to get the new weights w6, w7, and w8:

  w6 = 0.408666186

  w7 = 0.511301270

  w8 = 0.561370121

- We perform the actual updates in the neural network after we have the new weights leading into the hidden layer neurons (ie, we use the original weights, not the updated weights, when we continue the backpropagation algorithm below).

Hidden Layer

- Next, we'll continue the backwards pass by calculating new values for w1, w2, w3, and w4.

- Big picture, here's what we need to figure out:

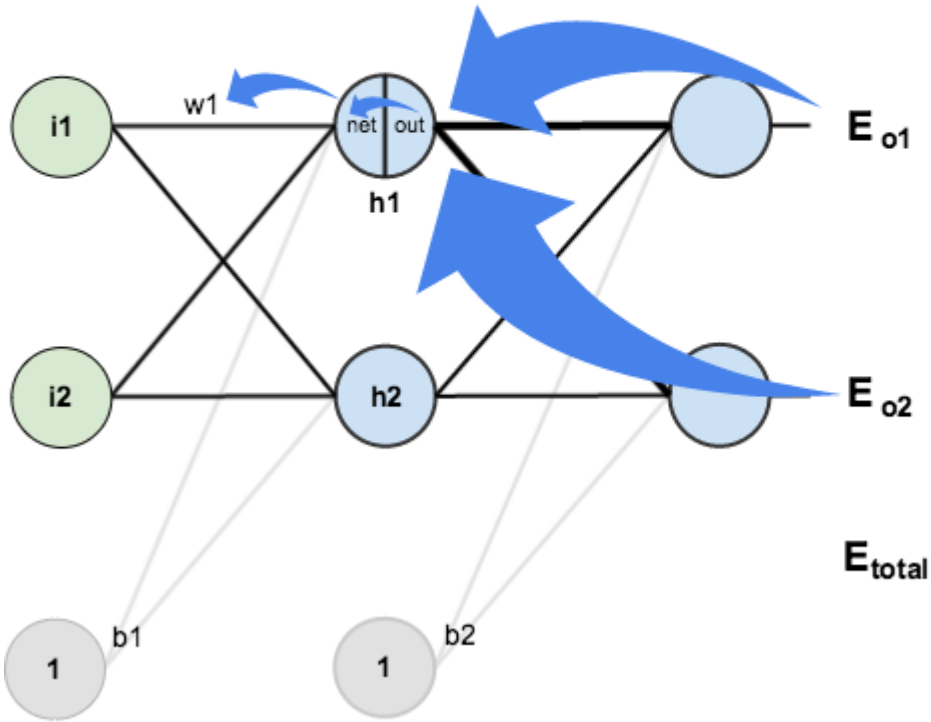
$$\delta E_{total} / \delta w1 = \delta E_{total} / \delta out_{h1} * \delta out_{h1} / \delta net_{h1} * \delta net_{h1} / \delta w1$$

Hidden Layer

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\downarrow$$

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$



$E_{total} = E_{o1} + E_{o2}$

Hidden Layer

- We're going to use a similar process as we did for the output layer, but slightly different to account for the fact that the output of each hidden layer neuron contributes to the output (and therefore error) of multiple output neurons.
- We know that out_{h1} affects both out_{o1} and out_{o2} therefore the $\delta E_{total} / \delta out_{h1}$ needs to take into consideration its effect on the both output neurons:

$$\delta E_{total} / \delta out_{h1} = \delta E_{o1} / \delta out_{h1} + \delta E_{o2} / \delta out_{h1}$$

Hidden Layer

- Starting with $\delta E_{o1} / \delta out_{h1}$:

  $\delta E_{o1} / \delta out_{h1} = \delta E_{o1} / \delta net_{o1} * \delta net_{o1} / \delta out_{h1}$

- We can calculate $\delta E_{o1} / \delta net_{o1}$ using values we calculated earlier:

  $\delta E_{o1} / \delta net_{o1} = \delta E_{o1} / \delta out_{o1} * \delta out_{o1} / \delta net_{o1} = 0.74136507 * 0.186815602 = 0.138498562$

- And $\delta net_{o1} / \delta out_{h1}$ is equal to w5:

  $net_{o1} = w5 * out_{h1} + w6 * out_{h2} + b2 * 1$

  $\delta net_{o1} / \delta out_{h1} = w5 = 0.40$

- Plugging them in:

  $\delta E_{o1} / \delta out_{h1} = \delta E_{o1} / \delta net_{o1} * \delta net_{o1} / \delta out_{h1} = 0.138498562 * 0.40 = 0.055399425$

Hidden Layer

- Following the same process for $\delta E_{o2} / \delta out_{h1}$, we get:

  $\delta E_{o2} / \delta out_{h1} = -0.019049119$

- Therefore:

  $\delta E_{total} / \delta out_{h1} = \delta E_{o1} / \delta out_{h1} + \delta E_{o2} / \delta out_{h1} = 0.055399425 + -0.019049119 = 0.036350306$

- Now that we have $\delta E_{total} / \delta out_{h1}$, we need to figure out $\delta out_{h1} / \delta net_{h1}$ and then $\delta net_{h1} / \delta w$ for each weight:

  $out_{h1} = 1/(1+e^{(-net_{h1})})$

  $\delta out_{h1} / \delta net_{h1} = out_{h1} (1 - out_{h1}) = 0.59326999(1 - 0.59326999) = 0.241300709$

Hidden Layer

- We calculate the partial derivative of the total net input to h1 with respect to w1 the same as we did for the output neuron:

  $net_{h1} = w1 * i1 + w2 * i2 + b1 * 1$

  $\delta net_{h1} / \delta w1 = i1 = 0.05$

- Putting it all together:

  $\delta E_{total} / \delta w1 = \delta E_{total} / \delta out_{h1} * \delta out_{h1} / \delta net_{h1} * \delta net_{h1} / \delta w1$

  $\delta E_{total} / \delta w1 = 0.036350306 * 0.241300709 * 0.05 = 0.000438568$

Hidden Layer

- We can now update w1:

  $w1 = w1 - \eta * \delta E_{total} / \delta w1 = 0.15 - 0.5 * 0.000438568 = 0.149780716$

- Repeating this for w2, w3, and w4 :

  w2 = 0.19956143

  w3 = 0.24975114

  w4 = 0.29950229

Backpropagation

- Finally, we've updated all of our weights!
- When we fed forward the 0.05 and 0.1 inputs originally, the error on the network was 0.298371109.
- After this first round of backpropagation, the total error is now down to 0.291027924.
- It might not seem like much, but after repeating this process 10,000 times, for example, the error plummets to 0.000035085.
- At this point, when we feed forward 0.05 and 0.1, the two outputs neurons generate 0.015912196 (vs 0.01 target) and 0.984065734 (vs 0.99 target).

Torch

Torch

- Torch is a scientific computing framework with wide support for machine learning algorithms. It is easy to use and efficient, thanks to an easy and fast scripting language, LuaJIT, and an underlying C/CUDA implementation.
- A summary of core features:

  a powerful N-dimensional array

  lots of routines for indexing, slicing, transposing, ...

  amazing interface to C, via LuaJIT

  linear algebra routines

  neural network, and energy-based models

  numeric optimization routines

  Fast and efficient GPU support

  Embeddable, with ports to iOS, Android and FPGA backends

Torch

```
print('basic printing')
a = 10
print(a)
print 'something'
print(type(a))
print(type('something'))
require 'image'
i = image.lena()
require 'nn'
n = nn.SpatialConvolution(1,16,12,12)
res = n:forward(image.rgb2y(i))
```

Torch

Loops and tests :

```
for j = 1, 10000 do
    if j % 1000 == 0 then
        print (j)
    end
end
```

Torch

Functions :

```
-- return 1 if x is even
local isEven = function(x)
    if x % 2 == 0 then
       return 1
    end
    return 0
end
```

Torch

Tensors :

Tensors are multi dimensional arrays

Tensor of dimension 1 = array

Tensor of dimension 2 = matrix

Tensor of dimension 3 = cube

Data and input/output of neural networks are represented with tensors in Torch

Torch

Tensors :

matrix = torch.Tensor (4, 2)

tab = torch.DoubleTensor (1000)

Torch

Exercise : Sieve of Eratosthenes

Algorithm for finding all prime numbers up to any given limit.

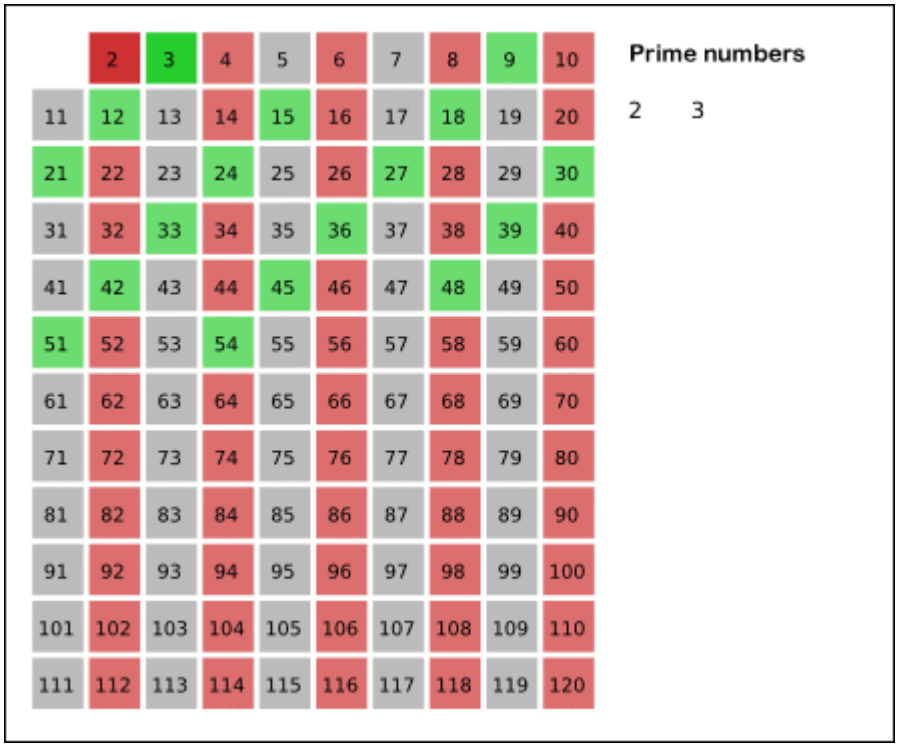
It does so by iteratively marking as composite (i.e., not prime) the multiples of each prime, starting with the multiples of 2.

The multiples of a given prime are generated as a sequence of numbers starting from that prime, with constant difference between them that is equal to that prime.

This is the sieve's key distinction from using trial division to sequentially test each candidate number for divisibility by each prime.

Torch

Sieve of Eratosthenes



Torch

- Include the neural network library :

  require 'nn'
- Declare a perceptron :

  net = nn.Sequential()

  net:add(nn.Linear(10, 1))

  net:add(nn.Tanh())

Torch

- Declare the error Mean Square Error criterion :

  criterion = nn.MSECriterion ()
- Do a forward pass :

  out = net:forward(inputs)
- Calculate the error :

  err = criterion:forward(out, targets)

Torch

- Initialize the gradients to zero :

  net:zeroGradParameters()
- Accumulate gradients :

  net:backward(inputs,

  criterion:backward (net.output, targets))
- Update parameters with a 0.01 learning rate :

  net:updateParameters(0.01)

Practical Work

- Implement a three layers XOR network.
- Make it learn the XOR function with two inputs and one output.

Monte Carlo Tree Search

Monte Carlo Go

- 1993 : first Monte Carlo Go program
  - Gobble, Bernd Bruegmann.
  - How nature would play Go ?
  - Simulated annealing on two lists of moves.
  - Statistics on moves.
  - Only one rule : do not fill eyes.
  - Result = average program for 9x9 Go.
  - Advantage : much more simple than alternative approaches.

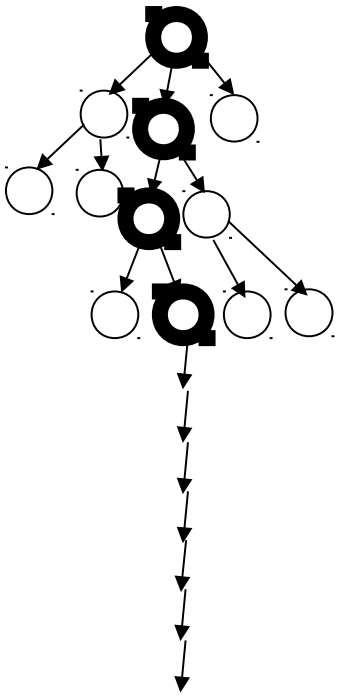
Monte Carlo Phantom Go

- Phantom Go is Go when you cannot see the opponent's moves.
- A referee tells you illegal moves.
- 2005 : Monte Carlo Phantom Go program.
- Many Gold medals at computer Olympiad since then using flat Monte Carlo.
- 2011 : Exhibition against human players at European Go Congress.

UCT

- UCT : Exploration/Exploitation dilemma for trees [Kocsis and Szepesvari 2006].
- Play random random games (playouts).
- Exploitation : choose the move that maximizes the mean of the playouts starting with the move.
- Exploration : add a regret term (UCB).

UCT



End of the game

1) descent of the tree

2) playout

3) update the tree

RAVE

- A big improvement for Go, Hex and other games is Rapid Action Value Estimation (RAVE) [Gelly and Silver 2007].

- RAVE combines the mean of the playouts that start with the move and the mean of the playouts that contain the move.

GRAVE

- Generalized Rapid Action Value Estimation (GRAVE) is a simple modification of RAVE.
- It consists in using the first ancestor node with more than n playouts to compute the RAVE values.
- It is a big improvement over RAVE for Go, Atarigo, Knightthrough and Domineering [Cazenave 2015].

MCTS

- Great success for the game of Go.
- Much better than all previous approaches to computer Go.

AlphaGo

Lee Sedol is among the strongest and most famous 9p Go player :

AlphaGo has won 4-1 against Lee Sedol.

AlphaGo

- AlphaGo combines MCTS and Deep Learning.
- There are four phases to the development of AlphaGo :
- Learn strong players moves => policy network.
- Play against itself and improve the policy network.
- Learn a value network to evaluate states from millions of games played against itself.
- Combine MCTS, policy and value network.

AlphaGo

- The policy network is a 13 layers network.
- It uses either 128 or 256 feature planes.
- It is fully convolutional.
- It learns to predict moves from hundreds of thousands of strong players games.
- Once it has learned, it finds the strong player move 57.0 % of the time.
- It takes 3 ms to run.

AlphaGo

- The value network is also a deep convolutional neural network.
- AlphaGo played a lot of games and kept for each game a state and the corresponding terminal state.
- It learns to evaluate states with the result of the terminal state.
- The value network has learned an evaluation function that gives the probability of winning.

AlphaGo

- The policy network is used as a prior to consider good moves at first.
- Playouts are used to evaluate moves
- The value network is combined with the statistics of the moves coming from the playouts.

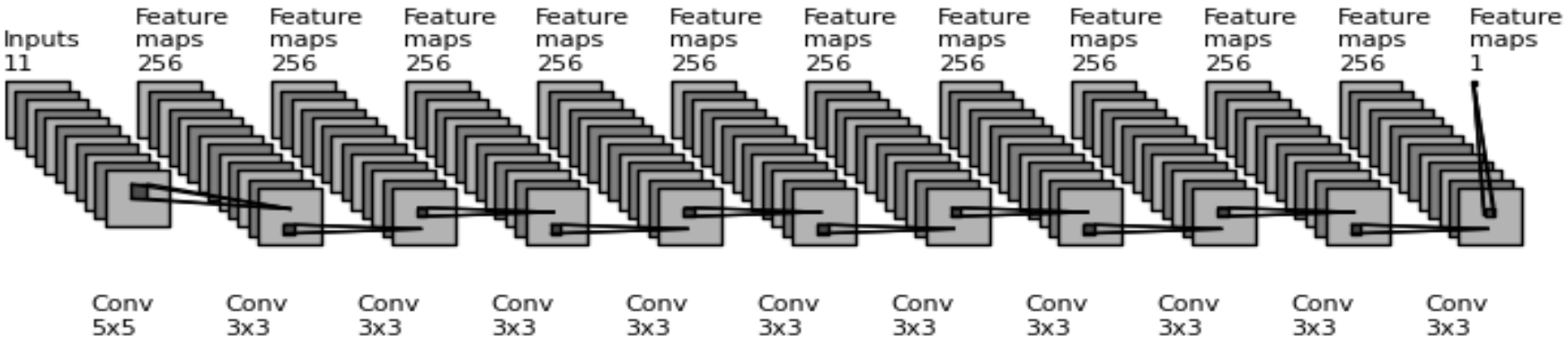
Golois

- I replicated the AlphaGo experiments with the policy network.
- Golois scores 58.4% on the test set (57.0% for AlphaGo).
- Golois plays on the kgs internet Go server.
- It has a strong 1d ranking just with the learned policy network.

Data

- Learning set = games played on the KGS Go server by players being 6d or more between 2000 and 2014.
- No handicap games.
- Each position is rotated to eight possible symmetric positions.
- 160 000 000 positions in the learning set.
- Test set = games played in 2015.
- 100 000 different positions not mirrored.
- Learning and test sets do not include moves from positions containing a ko.

Network


Inputs
11
Feature maps
256
Feature maps
256
Feature maps
256
Feature maps
256
Feature maps
256
Feature maps
256
Feature maps
256
Feature maps
256
Feature maps
256
Feature maps
256
Feature maps
1
Conv
5x5
Conv
3x3
Conv
3x3
Conv
3x3
Conv
3x3
Conv
3x3
Conv
3x3
Conv
3x3
Conv
3x3
Conv
3x3
Conv
3x3

Learning

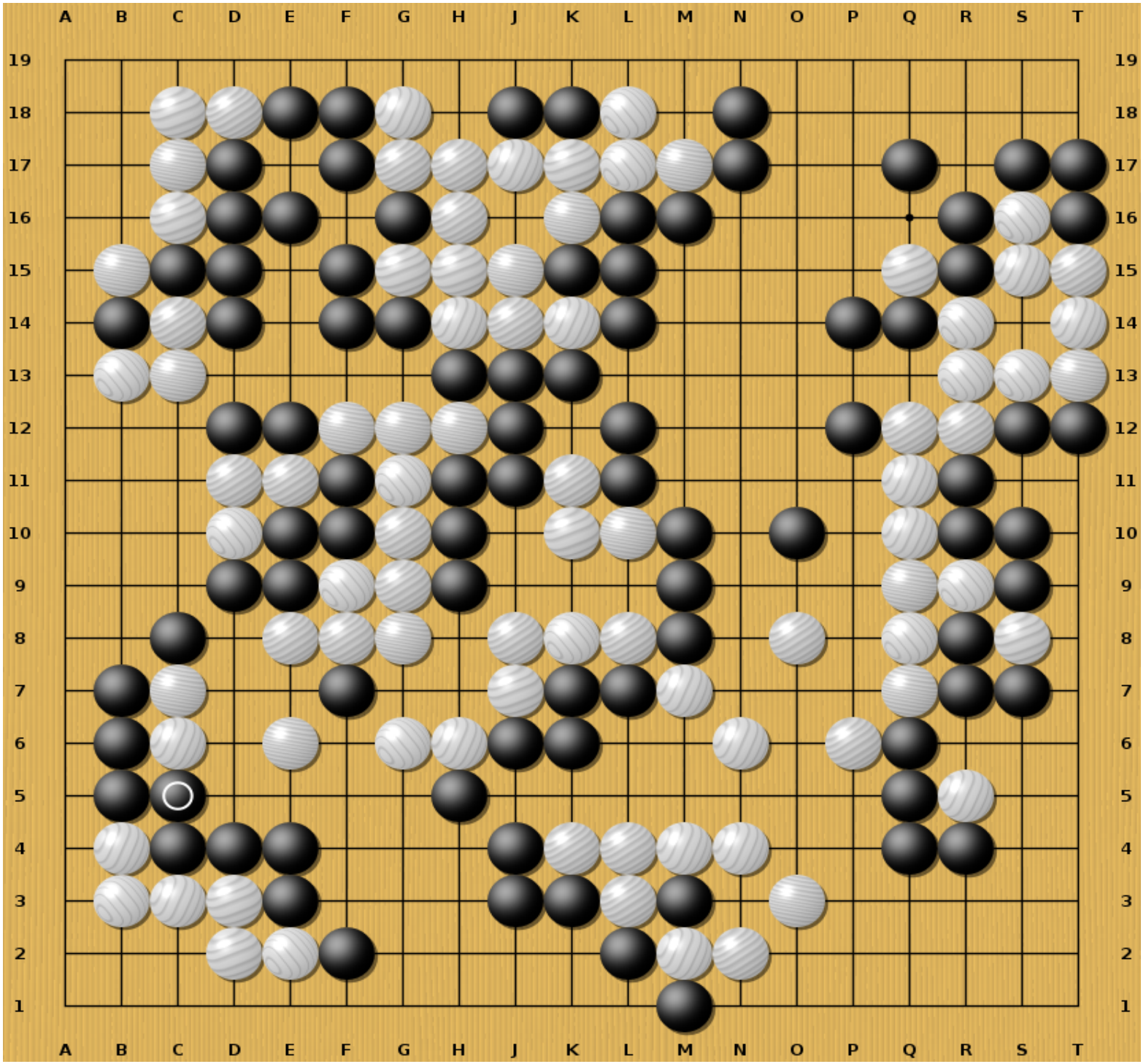
- Policy network = learn to find good moves.
- No softmax layer as a final layer.
- No minibatch.
- SGD on one example at a time.

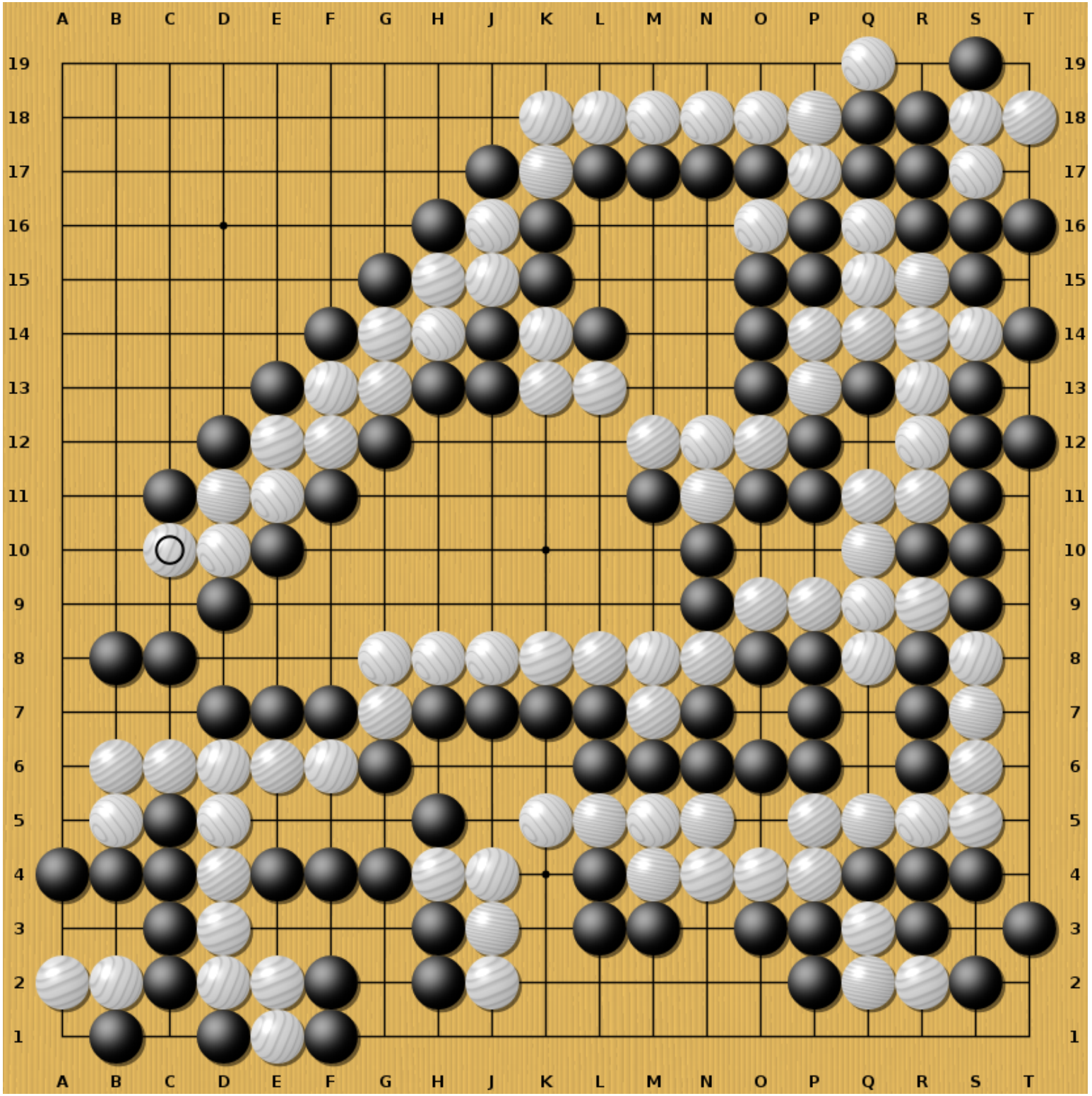
Results

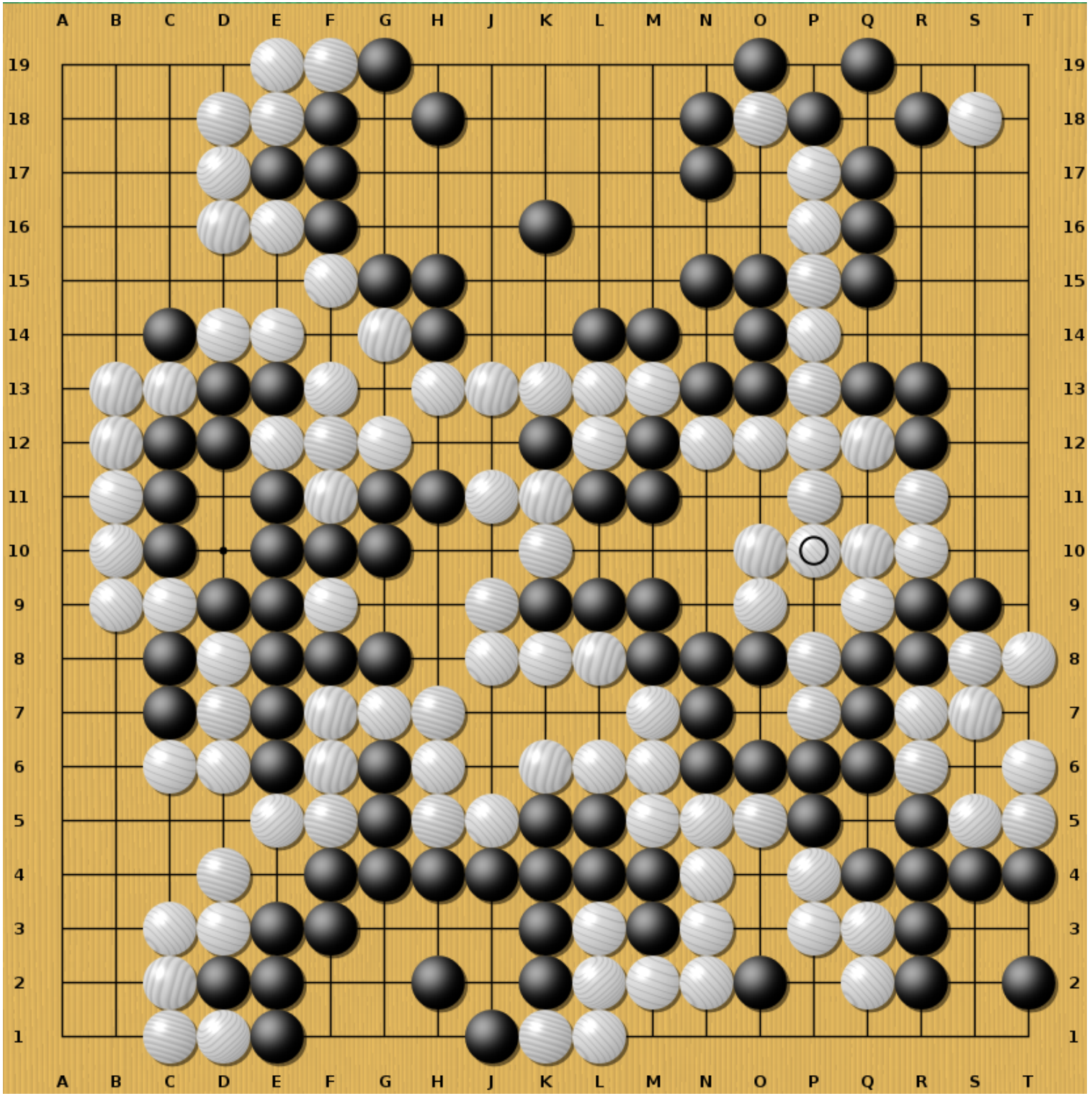
- 55.809% on the test set after 325 000 000 examples learned.
- Bagging = same network applied to the eight possible symmetries of a Go board.
- Value of a move = sum of the values of the symmetric move on the reflected boards.
- 56.513% with bagging.
- 1k on the KGS Go server.

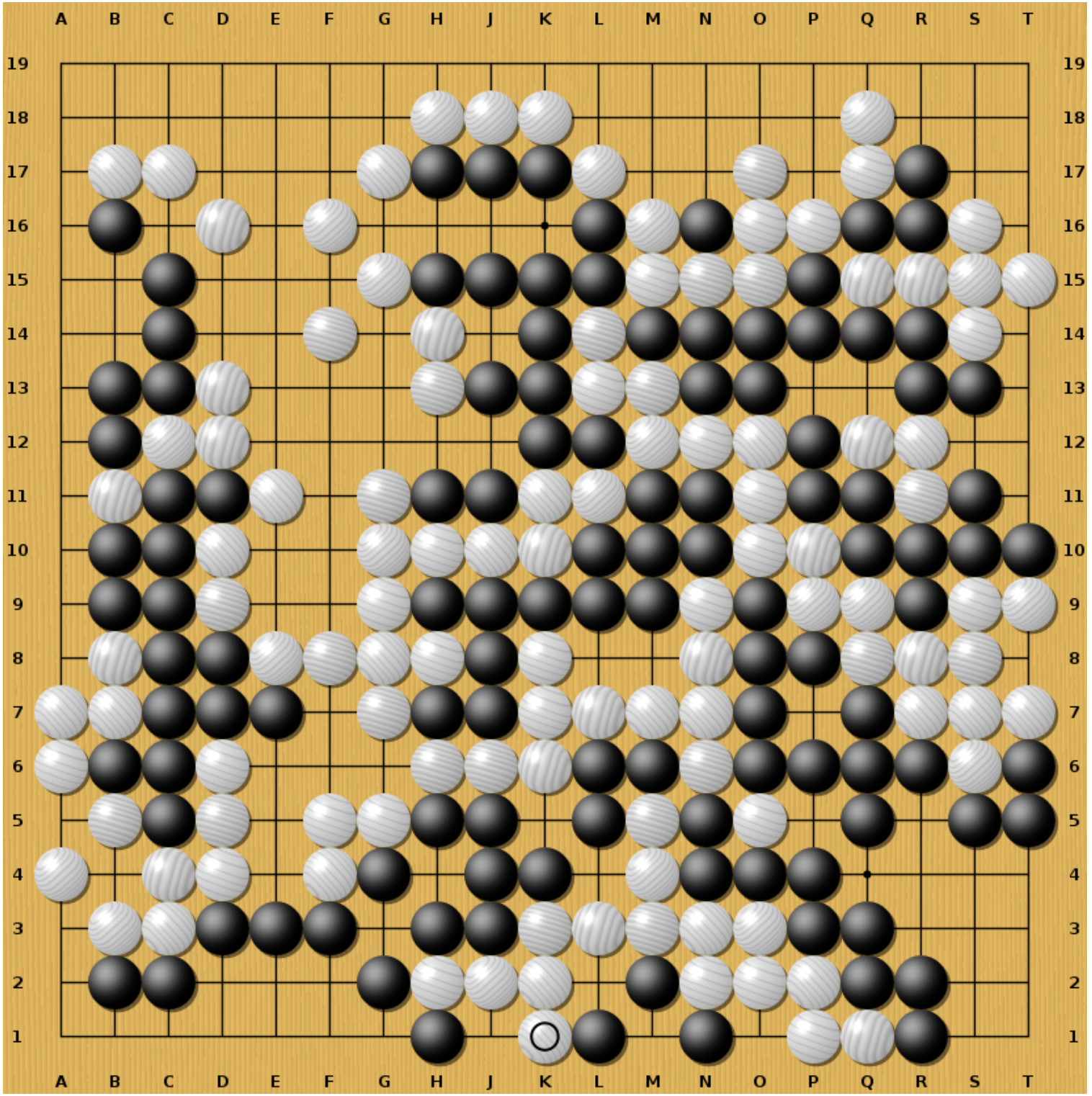
Search

- Simple problems that can be solved by tactical search are missed by the policy network.
- Ladders.
- Life and death.
- Current algorithm = play the tactical move
- Future work = result of tactical search as input to the network.


A B C D E F G H J K L M N O P Q R S T
19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1


A B C D E F G H J K L M N O P Q R S T
19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1


A B C D E F G H J K L M N O P Q R S T
19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1


A B C D E F G H J K L M N O P Q R S T
19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

Combination with tactical search

- Deep convolutional neural networks have difficulties at tactical search.
- Combining them with specialized tactical searches such as capture search or life and death search improves their level of play.
- Future work :
  - use more elaborate tactical search algorithms for capture and life and death.
  - use the results of tactical search as inputs to the neural network.

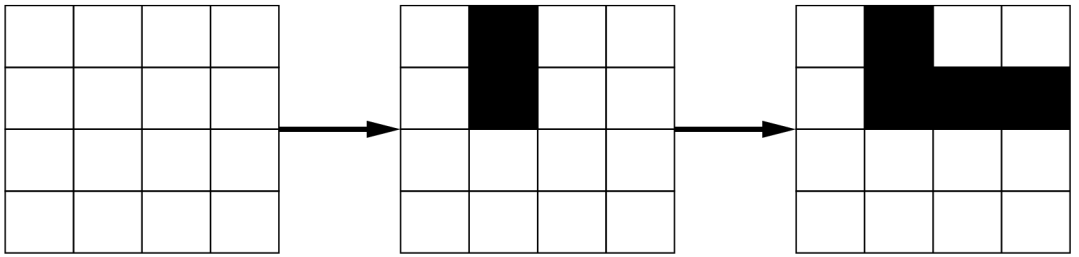
Practical Work

- Generate data with a Monte Carlo evaluation for a game so as to learn a policy network.

- Most simple game to program : Domineering.

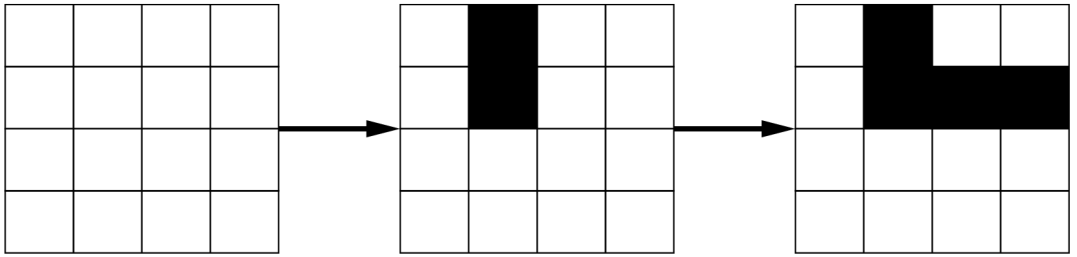
Domineering

- Put dominoes on a board :



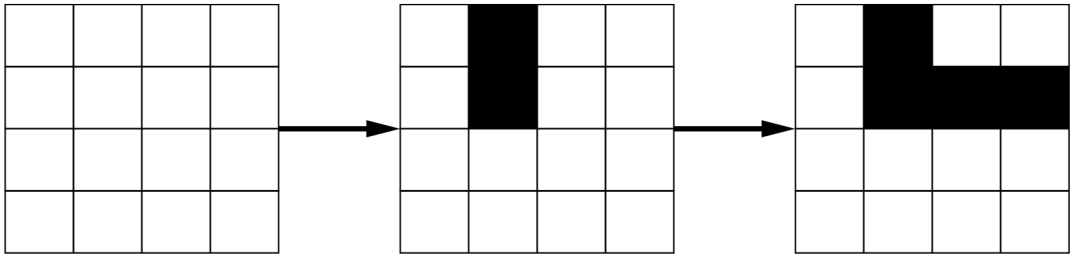
- The first player that cannot move has lost.
- Solve 3x3 and 4x4 Domineering with paper and pencil.

Domineering



- Represent a board and a move for 8x8 Domineering.
- Write a function for listing the possible moves.
- Playout : play a random move until the end of the game.

Domineering



- For each possible move play 100 playouts and calculate the average of the playouts.
- Recall the move that has the best average.

Practical Work

- Generate games with a Monte Carlo evaluation at Domineering.

Practical Work

- Generate csv files for the Domineering positions
- csv line for a 2x2 board :

0,0,0,0,1,1,1,1,1,1,1,1,1,0,0,0

board, flipped board, player plane, move to learn

- The goal is to learn a policy network.

- For each position recall the input (i.e. the board, the flipped board, and the turn) and the output of the same size as the board with only the best move found by the Monte Carlo evaluation marked as 1.

csv files

- Reading csv files with Torch :

```
local csvFile = io.open('domineering.csv', 'r')
for line in csvFile:lines('*l') do
    local l = line:split(',')
    for key, val in ipairs(l) do
        v = tonumber (val)
    end
end
```

csv files

csv line for a 2x2 board :

0,0,0,0,1,1,1,1,1,1,1,1,1,1,0,0

Corresponding input tensor :

0 0 1 1 1 1

0 0 1 1 1 1

Corresponding output tensor :

1 0

1 0

Loading and Writing Tensors

- Loading :

  local net = torch.load ('gofair.42.net')
- Writing

  torch.save ('gofair.42.net', net)
- Ascii mode : ascii()
- Binary mode : binary ()

Practical Work

- Transform .csv into .dat
- Data augmentation : generate the positions for the four symmetries of a board
- Shuffle the data so that similar positions are not next to each other.

Convolutional Networks

- Declare a convolutional layer :

  module = nn.SpatialConvolution(nInputPlane, nOutputPlane, kW, kH, [dW], [dH], [padW], [padH])

- Applies a 2D convolution over an input image composed of several input planes.
- The input tensor in forward(input) is expected to be a 3D tensor (nInputPlane x height x width).
- The parameters are the following:
  - nInputPlane: The number of expected input planes in the image given into forward().
  - nOutputPlane: The number of output planes the convolution layer will produce.
  - kW: The kernel width of the convolution
  - kH: The kernel height of the convolution
  - dW: The step of the convolution in the width dimension. Default is 1.
  - dH: The step of the convolution in the height dimension. Default is 1.
  - padW: The additional zeros added per width to the input planes.
    Default is 0, a good number is (kW-1)/2.
  - padH: The additional zeros added per height to the input planes.
    Default is padW, a good number is (kH-1)/2.

Convolutional Networks

- Note that depending of the size of your kernel, several (of the last) columns or rows of the input image might be lost. It is up to the user to add proper padding in images.

- If the input image is a 3D tensor nInputPlane x height x width, the output image size will be nOutputPlane x oheight x owidth where :

  owidth = floor((width + 2*padW - kW) / dW + 1)

  oheight = floor((height + 2*padH - kH) / dH + 1)

Convolutional Networks

- ReLU layer :
- ReLU is the abbreviation of Rectified Linear Units. This is a layer of neurons that applies the non-saturating activation function f (x) = max (0, x).
- Compared to other functions the usage of ReLU is preferable, because it results in the neural network training several times faster, without making a significant difference to generalisation accuracy.

Convolutional Networks

- Softmax :
- Applies the Softmax function to an n-dimensional input Tensor, rescaling them so that the elements of the n-dimensional output Tensor lie in the range (0,1) and sum to 1.
- Softmax is defined as $\exp(x_i) / \Sigma \exp(x_j)$
- In Torch :

  nn.SoftMax()

Convolutional Networks

- View:
- Enables to transform a tensor from a dimension to another one.
- For example in order to use SoftMax we need a one dimensional tensor instead of the two dimensional output of the network.
- In Torch :

  nn.View (64)

Practical Work

- Implement a convolutional network
- Fully convolutional
- 3 planes for the input layer
- 64 planes for the hidden layers
- Zero padding
- 3x3 filters
- 1 plane for the output layer
- ReLu
- Softmax

A Deep Learning Example

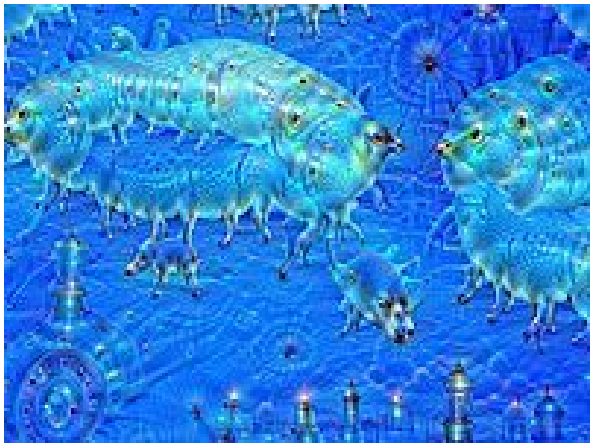
- 2015 Beijing computer Go tournament.
- David Fotland.
- Amazon.
- Recognizing mismatch in boxes.
- A team of people did not make it in two years.
- One person made it in one week using Deep Learning.

Use of Deep Learning

- Convolutional neural networks are extremely good at recognizing shapes.
- There are also unexpected and useful applications :
- Deep neural networks can be used to predict the biomolecular target of a compound.
- In 2015, Atomwise introduced AtomNet, the first deep learning neural networks for structure-based rational drug design.
- AtomNet was used to predict novel candidate biomolecules for several disease targets, most notably treatments for the Ebola virus and multiple sclerosis.

Deep Dream

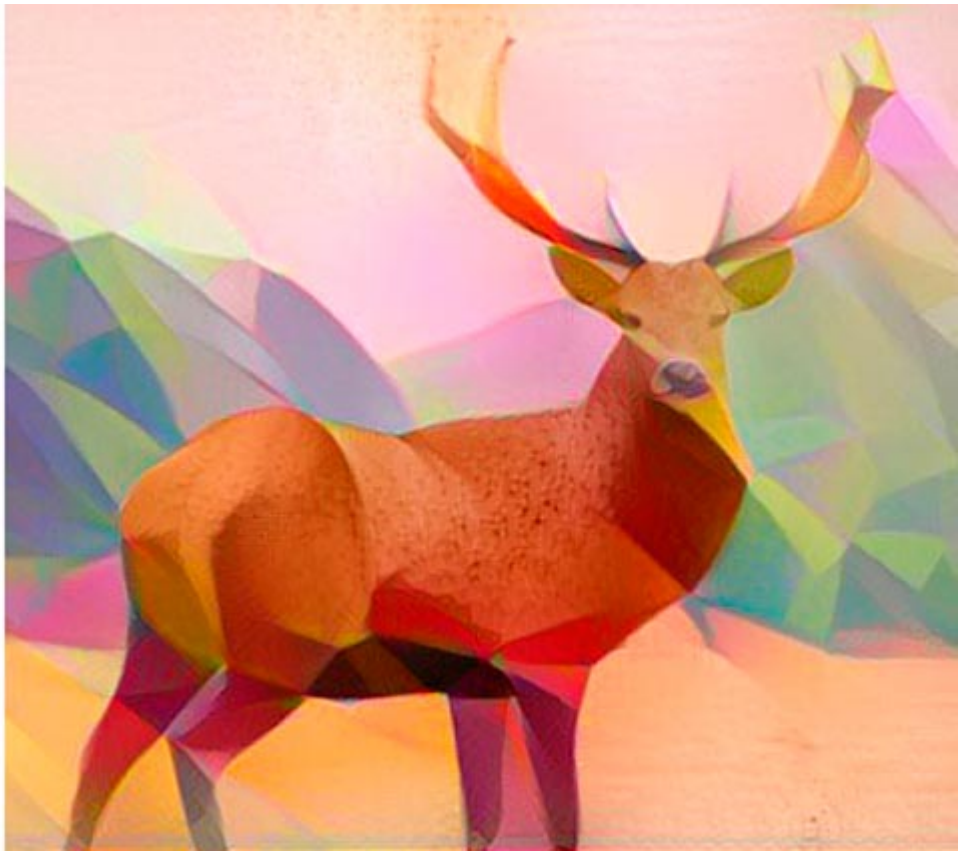
- The same image before and after applying ten iterations of DeepDream :



Deep Dream

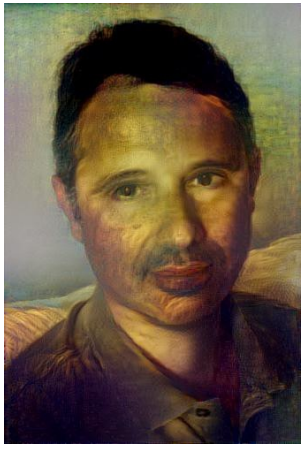
Ostagram

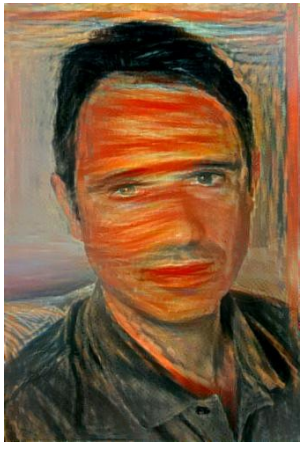
Ostagram

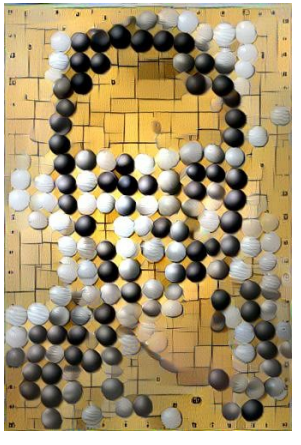


Deep Dream

- http://deepdreamgenerator.com :



Practical Work

- Learn the Domineering Monte Carlo evaluation function.
- Use the last examples as a test set.
- Print the percentage of test examples with the correct answer.
- Print the average test and training errors.

Practical Work

- Compare different network architectures.
- Make the network play games.
- Learn faster with cuda.
- Train with optim.

optim

We define an evaluation function, inside our training loop, and use optim.sgd to run training:

```
require 'optim'
local optimState = {learningRate=0.01}
for epoch=1,50 do
  local function feval(params)
    gradParams:zero()
    local outputs = model:forward(batchInputs)
    local loss = criterion:forward(outputs, batchLabels)
    local dloss_doutput = criterion:backward(outputs, batchLabels)
    model:backward(batchInputs, dloss_doutput)
    return loss,gradParams
  end
  optim.sgd(feval, params, optimState)
end
```

Bagging

- Bagging with the four symmetries of a board.
- Compute the four symmetrical input planes.
- Sum the outputs and choose the greatest one.
- Beware of assigning the correct symmetrical output to the correct original cell.

Bagging

Bagging for a 2x2 board :

0,0,1,0,1,1,0,1,1,1,1,1,1,1,0,0

Corresponding input tensor :

0 1 1 0 1 1

0 0 1 1 1 1

Corresponding symmetrical input tensor :

1 0 0 1 1 1

0 0 1 1 1 1

Bagging

- Calculate the error and the percentage on the test set with bagging.

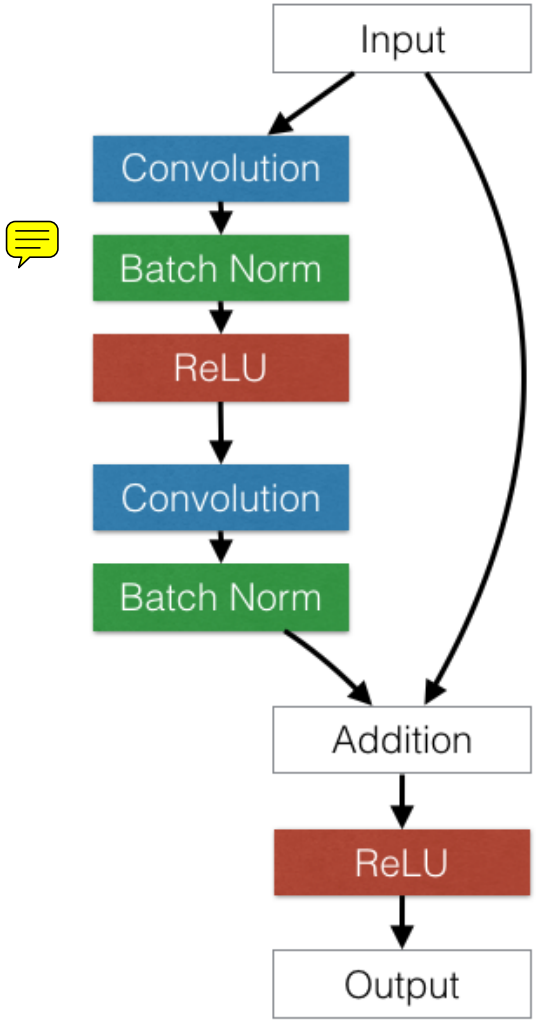
- It should have a better percentage than the network alone.

Minibatch

- Minibatch learning :
- Do forward and backward passes in a row and accumulate gradients before updating the weights.
- How : simply update parameters after a number of passes.

Residual Nets

- Residual Nets :



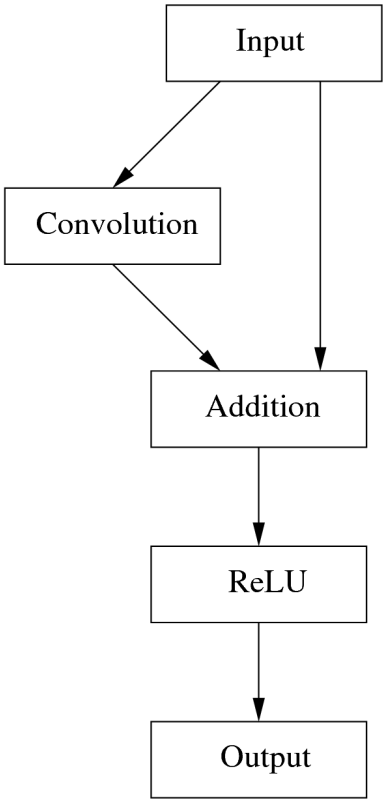
Residual Nets

- Residual Nets :

  nn.ConcatTable ()

  nn.Identity ()

  nn.CAddTable (true)



Practical Work

- Reinforcement learning :
  - Make the network play games
  - Take a state per game and learn the outcome of the game.

Practical Work

- MNIST