Deep Learning with Torch7 Framework (December 2016)

Pablo Freddy Ayala Heinrich (Efrei Student)

***Abstract*—Torch is a deep learning framework with wide support for machine learning algorithms. It's open-source, simple to use, and efficient, thanks to an easy and fast scripting language, LuaJIT, and an underlying C / CUDA implementation.**

Torch offers popular neural network and optimization libraries that are easy to use, yet provide maximum flexibility to build complex neural network topologies.

In the following paper we are going to use the torch framework, convolutional neural networks and the LUA language in order to solve the game name "Domineering:

***Index Terms*—AI, Torch, Torch7, deep learning, machine learning, neural network, Artificial Intelligence, Lua.**

I. Introduction

Artificial intelligence (AI) is intelligence exhibited by machines. In computer science, an ideal "intelligent" machine is a flexible rational agent that perceives its environment and takes actions that maximize its chance of success at some goal. Colloquially, the term "artificial intelligence" is applied when a machine mimics "cognitive" functions that humans associate with other human minds, such as "learning" and "problem solving".

As machines become increasingly capable, mental facilities once thought to require intelligence are removed from the definition. For example, optical character recognition is no longer perceived as an exemplar of "artificial intelligence", having become a routine technology. Capabilities currently classified as AI include successfully understanding human speech, competing at a high level in strategic game systems (such as Chess and Go), self-driving cars, and interpreting complex data. Some people also consider AI a danger to humanity if it progresses unabatedly.

AI research is divided into subfields that focus on specific problems or on specific approaches or on the use of a particular tool or towards satisfying particular applications.

Machine learning is the subfield of computer science that "gives computers the ability to learn without being explicitly programmed" (Arthur Samuel, 1959). Evolved from the study of pattern recognition and computational learning theory in artificial intelligence, machine learning explores the study and construction of algorithms that can learn from and make predictions on data – such algorithms overcome following strictly static program instructions by making data driven predictions or decisions, through building a model from sample inputs.

II. Deep learning

Deep learning (also known as deep structured learning, hierarchical learning or deep machine learning) is a branch of machine learning based on a set of algorithms that attempt to model high level abstractions in data. In a simple case, you could have two sets of neurons: ones that receive an input signal and ones that send an output signal. When the input layer receives an input, it passes on a modified version of the input to the next layer. In a deep network, there are many layers between the input and output (and the layers are not made of neurons but it can help to think of it that way), allowing the algorithm to use multiple processing layers, composed of multiple linear and non-linear transformations.

Deep learning is part of a broader family of machine learning methods based on learning representations of data. An observation (e.g., an image) can be represented in many ways such as a vector of intensity values per pixel, or in a more abstract way as a set of edges, regions of particular shape, etc. Some representations are better than others at simplifying the learning task (e.g., face recognition or facial expression recognition). One of the promises of deep learning is replacing handcrafted features with efficient algorithms for unsupervised or semi-supervised feature learning and hierarchical feature extraction.

III. Torch Framework

Torch is an open source machine learning library, a scientific computing framework, and a script language based on the Lua programming language. It provides a wide range of algorithms for deep machine learning, and uses the scripting language LuaJIT, and an underlying C implementation.

The core package of Torch is torch. It provides a flexible N-dimensional array or Tensor, which supports basic routines for indexing, slicing, transposing, type-casting, resizing, sharing storage and cloning. This object is used by most other packages and thus forms the core object of the library. The Tensor also

supports mathematical operations like max, min, sum, statistical distributions like uniform, normal and multinomial, and BLAS operations like dot product, matrix-vector multiplication, matrix-matrix multiplication, matrix-vector product and matrix product.

IV. Convolutional Neural Networks

Neural networks, as its name suggests, is a machine learning technique which is modeled after the brain structure. It comprises of a network of learning units called neurons. These neurons learn how to convert input signals (e.g. picture of a cat) into corresponding output signals (e.g. the label "cat"), forming the basis of automated recognition.

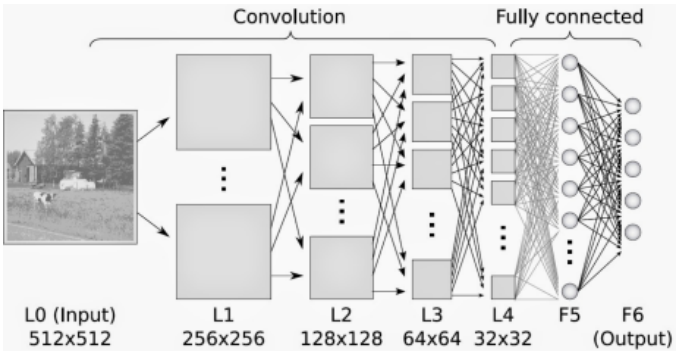
The technique that Google researchers used is called Convolutional Neural Networks (CNN), a type of advanced artificial neural network. It differs from regular neural networks in terms of the flow of signals between neurons. Typical neural networks pass signals along the input-output channel in a single direction, without allowing signals to loop back into the network. This is called a forward feed.

While forward feed networks were successfully employed for image and text recognition, it required all neurons to be connected, resulting in an overly-complex network structure. The cost of complexity grows when the network has to be trained on large datasets which, coupled with the limitations of computer processing speeds, result in grossly long training times. Hence, forward feed networks have fallen into disuse from mainstream machine learning in today's high resolution, high bandwidth, mass media age. A new solution was needed.

In 1986, researchers Hubel and Wiesel were examining a cat's visual cortex when they discovered that its receptive field comprised sub-regions which were layered over each other to cover the entire visual field. These layers act as filters that process input images, which are then passed on to subsequent layers. This proved to be a simpler and more efficient way to carry signals. In 1998, Yann LeCun and Yoshua Bengio tried to capture the organization of neurons in the cat's visual cortex as a form of artificial neural net, establishing the basis of the first CNN.

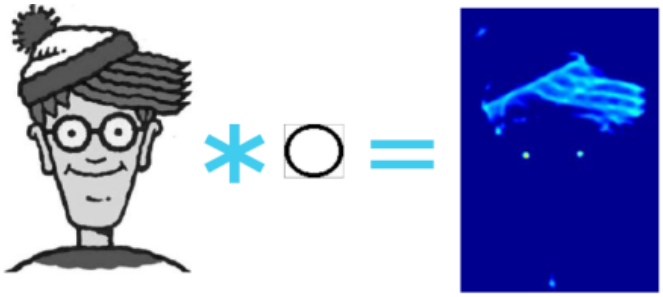
V. CNN Summarized in 4 Steps

There are four main steps in CNN: convolution, subsampling, activation and full connectedness.



A. *Convolution*

The first layers that receive an input signal are called convolution filters. Convolution is a process where the network tries to label the input signal by referring to what it has learned in the past. If the input signal looks like previous cat images it has seen before, the reference signal will be mixed into, or convolved with, the input signal. The resulting output signal is then passed on to the next layer.



B. *Subsampling*

Inputs from the convolution layer can be "smoothened" to reduce the sensitivity of the filters to noise and variations. This smoothing process is called subsampling, and can be achieved by taking averages or taking the maximum over a sample of the signal. Examples of subsampling methods (for image signals) include reducing the size of the image, or reducing the color contrast across red, green, blue (RGB) channels.

C. *Activation*

The activation layer controls how the signal flows from one layer to the next, emulating how neurons are fired in our brain. Output signals which are strongly associated with past references would activate more neurons, enabling signals to be propagated more efficiently for identification.

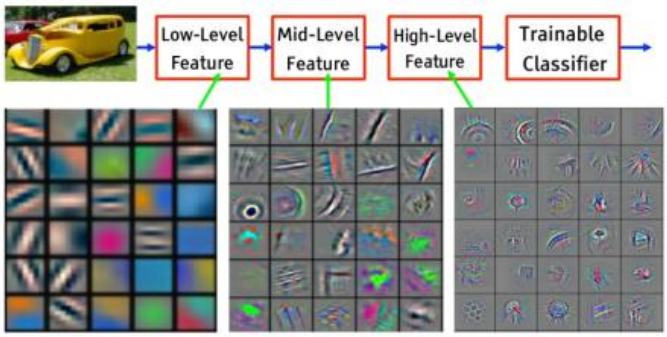
CNN is compatible with a wide variety of complex activation functions to model signal propagation, the most common function being the Rectified Linear Unit (ReLU), which is favored for its faster training speed.

D. *Fully Connected*

The last layers in the network are fully connected, meaning that neurons of preceding layers are connected to every neuron in subsequent layers. This mimics high level reasoning where all possible pathways from the input to output are considered.

When training the neural network, there is additional layer called the loss layer. This layer provides feedback to the neural network on whether it identified inputs correctly, and if not, how far off its guesses were. This helps to guide the neural network to reinforce the right concepts as it trains. This is always the last layer during training.

Learning algorithms require feedback. This is done using a validation set where the CNN would make predictions and compare them with the true labels or ground truth. The predictions which errors are made are then fed backwards to the CNN to refine the weights learned, in a so called backwards pass. Formally, this algorithm is called backpropagation of errors, and it requires functions in the CNN to be differentiable (almost).



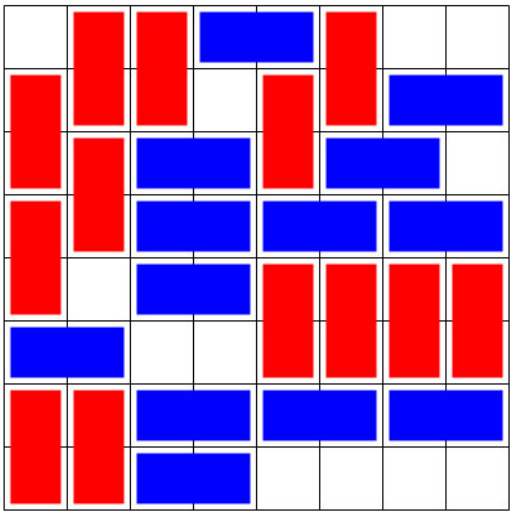
VI. MONTE CARLO SIMULATION

Monte Carlo simulation is a computerized mathematical technique that allows people to account for risk in quantitative analysis and decision making. The technique is used by professionals in such widely disparate fields as finance, project management, energy, manufacturing, engineering, research and development, insurance, oil & gas, transportation, and the environment.

Monte Carlo simulation furnishes the decision-maker with a range of possible outcomes and the probabilities they will occur for any choice of action. It shows the extreme possibilities—the outcomes of going for broke and for the most conservative decision—along with all possible consequences for middle-of-the-road decisions.

The technique was first used by scientists working on the atom bomb; it was named for Monte Carlo, the Monaco resort town renowned for its casinos. Since its introduction in World War II, Monte Carlo simulation has been used to model a variety of physical and conceptual systems.

VII. DOMINEERING GAME

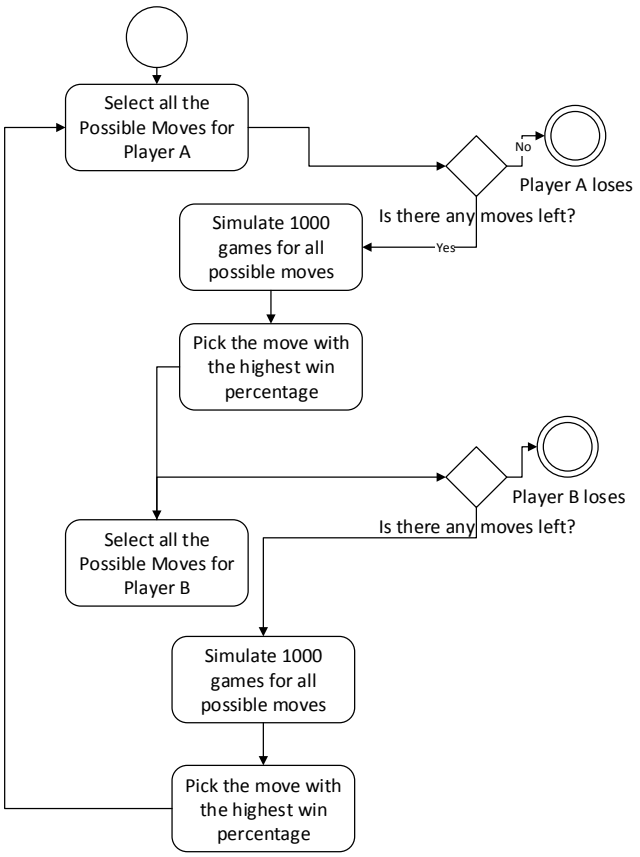


Domineering (also called Stop-Gate or Crosscram) is a mathematical game played on a sheet of graph paper, with any set of designs traced out. For example, it can be played on a 6×6 square, a checkerboard, an entirely irregular polygon, or any combination thereof.

The game is played on checkered boards of different sizes and involves placing dominoes that cover two squares. Player A starts and must place the dominoes in a ‘vertical’ orientation, while Player B places the dominoes in a ‘horizontal’ orientation. The first player who is unable to place a domino on the board has lost – there cannot be a draw.

VIII. GENERATING TRAINING DATA

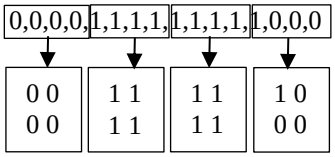
In order to solve the Domineering game using a convolutional neural network implemented in Torch it is necessary first to generate training data that is going to be used to train the neural network. In order to do so we use the MonteCarlo simulation in the following way:



A C# program was written that evaluates all possible random moves for each player and simulates for each move 1000 moves until completion, picking the move that won the most times until the game is completed. In order to load the information into torch the program generates a csv file that has the following structure:

[PlayerBoard], [FlippedBoard],[PlayerPlane],[MovetoLearn]

The real implemented program uses an 8x8 board, a 2x2 representation is shown for demonstration purposes:



- Player Board: 2D representation of the current state of the player board.
- Flipped Board: Reverse values of the Board, changing 1 per 0 and vice versa.
- Player Plane: All positions occupied by 1 for player 2 and 0 for player 1.
- Move to learn: Move to perform.

For each move, there is one line in the csv file generated using the monte carlo simulation.

IX. Loading the Data into Torch

In torch the data is going to be loaded into tensors, that are defined as:

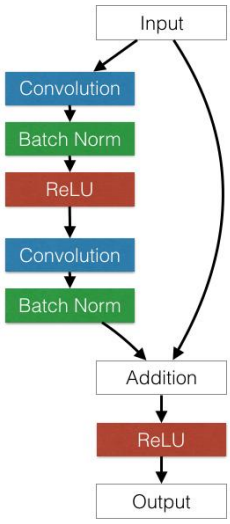
```
local input = torch.Tensor(ctr,3,8,8)
local output = torch.Tensor(ctr,1,8,8)
```

For each line of the csv file, the tensors are loaded in the following manner:

input[linenumber][tensornumber][row][column] = val

output[linenumber][1][row][column] = val

X. Implementation of the Neural Network



In order to solve the problem using torch, we need to define the following neural 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
- Rely
- Softmax

The convolutional neural network is defined in the following way:

```
require 'nn'
local convnet = nn.Sequential()
local nplanes = 64
--Filter, stride, padding
convnet:add(nn.SpatialConvolution(3,nplanes,5,5,1,1,2,2))
convnet:add(nn.ReLU())
convnet:add(nn.SpatialConvolution(nplanes,nplanes,3,3,1,1,
1,1))
```

```
convnet:add(nn.ReLU())
convnet:add(nn.SpatialConvolution(nplanes,nplanes,3,3,1,1,1,1))
convnet:add(nn.ReLU())
convnet:add(nn.SpatialConvolution(nplanes,nplanes,3,3,1,1,1,1))
convnet:add(nn.ReLU())
convnet:add(nn.SpatialConvolution(nplanes,nplanes,3,3,1,1,1,1))
convnet:add(nn.ReLU())
convnet:add(nn.SpatialConvolution(nplanes,nplanes,3,3,1,1,1,1))
convnet:add(nn.ReLU())
convnet:add(nn.SpatialConvolution(nplanes,1,3,3,1,1,1,1))
convnet:add(nn.View(64))
convnet:add(nn.SoftMax())
convnet:add(nn.View(8,8))
```

XI. TRAINING

In order to train our neural networks, we are going to analyses two ways of doing so: criterion and optim SGD.

A. Criterion

Criterions are helpful to train a neural network. Given an input and a target, they compute a gradient according to a given loss function.

Criterions can be grouped into:

- Classification
- Regression
- Embedding criterions
- Miscellaneous criterions

Like the Module class, Criterions is an abstract class with the following methods:

- Forward(predicted, actualTarget). Given a predicted value and a actual target computes the loss function associated to the criterions.
- Backward(input,target). Given an input and a target compute the gradients of the loss function associated to the criterion.

These two function should be used to compute the loss and update the weights of the neural network during the training phase.

In our example, we use the criterion as following to train the neural network 1000 times for every line in the csv:

```
criterion = nn.MSECriterion()
for j =1,1000 do
 for i =1,ctr-1 do

--feed it to the neural network and the criterion
 err=criterion:forward(convnet:forward(input[i]),output[i])
  --train over this example in 3 steps
  --(1) zero the accumulation of the gradients
  convnet:zeroGradParameters()
  --(2) accumulate gradicleents

convnet:backward(input[i],criterion:backward(convnet.output,output[i]))
  --(3) update parameters with a 0.01 learning rate
  convnet:updateParameters(0.2)
   print("Error:"..err)
 end
end
```

B. Optim SGD

This package provides a set of optimization algorithms, which all follow a unified, closure-based API.

This package is fully compatible with the nn package, but can also be used to optimize arbitrary objective functions, from this package we used the Stochastic Gradient Descent.

Gradient descent is an optimization algorithm used to find the values of parameters (coefficients) of a function (f) that minimizes a cost function (cost).

Gradient descent is best used when the parameters cannot be calculated analytically (e.g. using linear algebra) and must be searched for by an optimization algorithm.

Gradient descent can be slow to run on very large datasets. Because one iteration of the gradient descent algorithm requires a prediction for each instance in the training dataset, it can take a long time when you have many millions of instances.

In situations when you have large amounts of data, you can use a variation of gradient descent called stochastic gradient descent. In this variation, the gradient descent procedure described above is run but the update to the coefficients is performed for each training instance, rather than at the end of the batch of instances.

The first step of the procedure requires that the order of the training dataset is randomized. This is to mix up the order that updates are made to the coefficients. Because the coefficients are updated after every training instance, the updates will be noisy jumping all over the place, and so will the corresponding cost function. By mixing up the order for the updates to the coefficients, it harnesses this random walk and avoids it getting distracted or stuck.

The update procedure for the coefficients is the same as that above, except the cost is not summed over all training patterns, but instead calculated for one training pattern.

The learning can be much faster with stochastic gradient descent for very large training datasets and often you only need a small number of passes through the dataset to reach a good or good enough set of coefficients, e.g. 1-to-10 passes through the dataset.

We define our training method as following:

```
criterion = nn.MSECriterion()
for j =1,1000 do
  for i =1,ctr-1 do
  print("Cycle:"..j)
  print("Tensor["..i..."]")
    local optimState = {learningRate=0.2}
    params, gradParams = convnet:getParameters()
      local function feval(params)
        gradParams:zero()
        local outputs = convnet:forward(input[i])
        local loss = criterion:forward(outputs,output[i])
        local dloss_doutput = criterion:backward(outputs,
output[i])
         print('loss:'..loss)
        convnet:backward(input[i],dloss_doutput )
        return loss, gradParams
      end
 optim.sgd(feval, params, optimState)

   end

  end
```

XII. TESTING THE MODEL

In order to test the learned model, we run it against the original output using the method convent:forward.

```
for j=1,ctr-1 do
print(j)
  out = convnet:forward(input[j])
  print("input:")
  print(input[j])
  print("output:(original)")
  print(output[j])
  print("output:(learned)")
  print(out)

end
```

Therefore, if we test the model we get the following results,by using this input:

```
input:
(1,.,.) =
  0  0  0  0  0  1  0  0
  1  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0
  1  0  0  0  0  0  0  0
  1  0  0  0  1  0  0  0
  0  0  0  0  0  0  0  0
  0  0  1  1  0  0  0  0
  0  0  0  1  0  0  0  0

(2,.,.) =
  1  1  1  1  1  0  1  1
  0  1  1  1  1  1  1  1
  1  1  1  1  1  1  1  1
  0  1  1  1  1  1  1  1
  0  1  1  1  0  1  1  1
  1  1  1  1  1  1  1  1
  1  1  0  0  1  1  1  1
  1  1  1  0  1  1  1  1

(3,.,.) =
  1  1  1  1  1  1  1  1
  1  1  1  1  1  1  1  1
  1  1  1  1  1  1  1  1
  1  1  1  1  1  1  1  1
  1  1  1  1  1  1  1  1
  1  1  1  1  1  1  1  1
  1  1  1  1  1  1  1  1
  1  1  1  1  1  1  1  1
[torch.DoubleTensor of size 3x8x8]
```

We can compare the original output

```
output:(original)
(1,.,.) =
  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  1  0
  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0
[torch.DoubleTensor of size 1x8x8]
```

To the learned output:

```
output:(learned)
 0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0
 0  0  0  0  0  0  1  0
 0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0
[torch.DoubleTensor of size 8x8]
```

Therefore, we can verify that our model has correctly learned how to play domineering.

XIII. Conclusions

The goal of Torch is to have maximum flexibility and speed in building your scientific algorithms while making the process extremely simple. Torch comes with a large ecosystem of community-driven packages in machine learning, computer vision, signal processing, parallel processing, image, video, audio and networking among others, and builds on top of the Lua community.

At the heart of Torch are the popular neural network and optimization libraries which are simple to use, while having maximum flexibility in implementing complex neural network topologies. You can build arbitrary graphs of neural networks, and parallelize them over CPUs and GPUs in an efficient manner.

Consequently, we have successfully implemented a model that uses torch in order to play the Domineering game using Deep Learning and convolutional neural networks.

References

[1] http://torch.ch/
[2] https://research.fb.com/fair-open-sources-deep-learning-modules-for-torch/
[3] https://en.wikipedia.org/wiki/Torch_(machine_learning)
[4] https://github.com/torch/torch7/wiki/Cheatsheet
[5] https://github.com/torch/tutorials
[6] http://fastml.com/torch-vs-theano/
[7] http://hunch.net/~nyoml/torch7.pdf