Knet: beginning deep learning with 100 lines of Julia

Deniz Yuret
Department of Computer Engineering
Koç University, İstanbul
dyuret@ku.edu.tr

Abstract

Knet (pronounced "kay-net") is the Koç University machine learning framework implemented in Julia, a high-level, high-performance, dynamic programming language. Unlike gradient generating compilers like Theano and TensorFlow which restrict users into a modeling mini-language, Knet allows models to be defined by just describing their forward computation in plain Julia, allowing the use of loops, conditionals, recursion, closures, tuples, dictionaries, array indexing, concatenation and other high level language features. High performance is achieved by combining automatic differentiation of most of Julia with efficient GPU kernels and memory management. Several examples and benchmarks are provided to demonstrate that GPU support and automatic differentiation of a high level language are sufficient for concise definition and efficient training of sophisticated models.

1 Introduction

Knet [1] is a machine learning framework that allows users to define and train their models in plain Julia. Specifically, only the forward computation, i.e. calculation of a loss value for a given set of parameters and data, needs to be implemented by the user. Computation of gradients required for training is handled automatically. GPU operations are automatically triggered when the inputs are constructed as device arrays. Most of Julia can be used in model implementations, including all control statements, array operations, high level data types, helper functions and closures. This gives users great flexibility in defining compound models where multiple parts can pass information to each other, and the exact sequence of operations may depend on the data.

Julia [2] is an open source, high-level programming language that often matches the performance of C with its high performance LLVM-based just-in-time (JIT) compiler. It comes with an extensive array and mathematical function library with support for parallel and GPU computation. Julia is easy to extend using its dynamic type system, multiple dispatch support, C function calls and Lisp-like macro and metaprogramming facilities. Knet uses each of these features to implement a practical and high-performance machine learning framework.

The next section gives the Julia/Knet implementations of five example models and timing comparisons with other frameworks. Section 3 describes the implementation of Knet.

2 Examples and Benchmarks

In Knet, a machine learning model is defined using plain Julia code. A typical model consists of a *prediction* and a *loss* function. The prediction function takes model parameters and some input, returns the prediction of the model for that input. The loss function measures how bad the prediction is with respect to some desired output. We train a model by adjusting its parameters to reduce the loss. In this section we will see the prediction, loss, and training functions for five models: linear regression, softmax classification, fully-connected, convolutional and recurrent neural networks. Note

30th Conference on Neural Information Processing Systems (NIPS 2016), Barcelona, Spain.

that each example below comes from the actual code from a working demo in the Knet repository used in the benchmarking experiments in this paper.

2.1 Linear Regression

Here is the prediction and the quadratic loss function for a simple linear regression model:

```
function predict(w,x)
  return w[1]*x .+ w[2]
end
```

```
function loss(w,x,y)
  return sumabs2(y-predict(w,x))/size(y,2)
end
```

Both `predict` and `loss` are regular Julia functions: we could have used loops, conditionals, recursion etc. in their definition. The variable `w` is a list of parameters, `x` is the input and `y` is the desired output. By providing different types of inputs `x`, `y`, and `w[i]`, the user can control where the code is run. If the inputs are regular Julia arrays, the code will run on the CPU using operators from the Julia standard library. If they are device arrays (see `KnetArray` in Section 3.2), the same code will run on the GPU using equivalent operators provided by Knet.

To train this model, we want to adjust its parameters to reduce the loss on given training examples. The direction in the parameter space in which the loss reduction is maximum is given by the negative gradient of the loss. Knet provides the `grad` function (see Section 3.1) to compute the gradient:

```
using Knet
lossgradient = grad(loss)
```

`grad` is a higher-order function that takes the `loss` function as input and returns a `lossgradient` function as output. `lossgradient` takes the same arguments as `loss`, e.g. `dw = lossgradient(w,x,y)`. Instead of returning a loss value however, `lossgradient` returns `dw`, the gradient of `loss` with respect to its first argument `w`. The type and size of `dw` is identical to `w`, which could be a `Tuple`, `Array`, or `Dict`. Each entry in `dw` gives the derivative of the loss with respect to the corresponding entry in `w`.

Given some training data `[(x1,y1),(x2,y2),...]`, we can train this model using Stochastic Gradient Descent (SGD) [3]. We simply iterate over the input-output pairs in `data`, calculate the `lossgradient` for each pair, and move the parameters in the negative gradient direction with a step size determined by the learning rate `lr`:

```
function train(w, data; lr=0.1)
    for (x,y) in data
        dw = lossgradient(w, x, y)
        for i in 1:length(w)
            w[i] -= lr * dw[i]
        end
    end
    return w
end
```

This is the full implementation. All the user has to do at this point is to load the data into an `data` array, initialize the weights `w`, and use the `train` function to get an optimized model.

2.2 Softmax classification

Classification models handle discrete outputs as opposed to regression models for continuous outputs. We typically use the cross entropy loss function in classification models:

```
function loss(w,x,ygold)
    ypred = predict(w,x)
    ynorm = ypred .- log(sum(exp(ypred),1))
    -sum(ygold .* ynorm) / size(ygold,2)
end
```

Other than the change in the loss function the softmax model is identical to the linear regression model. We use the same `predict`, same `train` and set `lossgradient=grad(loss)` as before.

2.3 Multi-layer perceptrons

A multi-layer perceptron (MLP), i.e. a fully connected feed-forward neural network, is basically several linear regression models stuck together with non-linearities in between. We can define an MLP by slightly modifying the predict function from the softmax example:

```
function predict(w,x)
    for i=1:2:length(w)-2
        x = max(0, w[i]*x .+ w[i+1])
    end
    return w[end-1]*x .+ w[end]
end
```

Here `w[2k-1]` is the weight matrix and `w[2k]` is the bias vector for the k'th layer. `max(0,a)` implements the popular rectifier function as a non-linearity. Note that if `w` only has two entries, this is equivalent to the `predict` in linear and softmax models. By adding more entries to `w`, we can define multi-layer perceptrons of arbitrary depth. The rest of the code for training an MLP is identical to the softmax model. We can use the same cross-entropy loss function and the same SGD training loop.

2.4 Convolutional neural networks

Convolutional neural networks (CNN) are similar to MLPs except they use two extra operations: (i) convolutions, which implement sparsely connected layers with tied weights, and (ii) pooling, which subsample and reduce the input size. Knet provides the `conv4(w,x)` and `pool(x)` functions that implement these operations. A typical CNN consists of a number of convolutional layers followed by one or more fully connected layers. Here is an implementation of the LeNet [4] CNN model in Knet:

```
function predict(w,x,n)
    for i=1:2:n
        x = pool(max(0, conv4(w[i],x) .+ w[i+1]))
    end
    x = mat(x)
    for i=n+1:2:length(w)-2
        x = max(0, w[i]*x .+ w[i+1])
    end
    return w[end-1]*x .+ w[end]
end
```

Here `n` determines the index of the last convolution weight and `mat` reshapes a multidimensional array into a matrix. Other than the first few lines in the `predict` function the rest of the code, including `loss`, `train` etc., is identical to the MLP example. A similar implementation for a deep network like the 19 layer VGG model [5] takes about a dozen lines of code. For comparison, the original Caffe model specification for LeNet is 130 lines long and VGG is 396 lines long.

2.5 Recurrent neural networks

A recurrent neural network (RNN) is a model where connections between units form a directed cycle, which allows it to keep a persistent state over time. Here is one possible Knet implementation of an LSTM unit [6] commonly used as a building block in RNN models:

```
function lstm(weight,bias,hidden,cell,input)
    gates   = hcat(hidden,input) * weight .+ bias
    hsize   = size(hidden,2)
    forget  = sigm(gates[:,1:hsize])
    ingate  = sigm(gates[:,1+hsize:2hsize])
    outgate = sigm(gates[:,1+2hsize:3hsize])
    change  = tanh(gates[:,1+3hsize:end])
    cell    = cell .* forget + ingate .* change
    hidden  = outgate .* tanh(cell)
    return (hidden,cell)
end
```

The first two arguments are the parameters: the weight matrix and the bias vector. The next two arguments hold the internal state of the LSTM: the hidden and cell arrays. The last argument is the input. Note that for performance reasons we lump all the parameters of the LSTM into one matrix-vector pair instead of using separate parameters for each gate. This way we can perform a single matrix multiplication, and recover the gates using array indexing. We represent `input`, `hidden` and `cell` as row vectors rather than column vectors for more efficient concatenation and indexing (Julia is column-major). `sigm` and `tanh` are the sigmoid and the hyperbolic tangent activation functions. The LSTM returns the updated state variables `hidden` and `cell`.

As an example application, let's build a character based language model inspired by [7] using the LSTM as the basic building block. Here is the `predict` function for a multi-layer LSTM model:

```
function predict(w, s, x)
    x = x * w[end-2]
    for i = 1:2:length(s)
        (s[i],s[i+1]) = lstm(w[i],w[i+1],s[i],s[i+1],x)
        x = s[i]
    end
    return x * w[end-1] .+ w[end]
end
```

A language model predicts the next token in a sequence given the current token and the recent history as encoded in its internal state. `predict` takes the current token encoded in `x` as a one-hot row vector. `s[2k-1:2k]` hold the hidden and cell arrays and `w[2k-1:2k]` hold the weight and bias parameters for the k'th LSTM layer. The last three elements of `w` are the embedding matrix and the weight/bias for the final prediction. `predict` multiplies the input `x` with the embedding matrix, passes it through a number of LSTM layers, and converts the output of the final layer to the same number of dimensions as the input using a linear transformation. The state variable `s` is modified in-place.

To train the RNN model we use Backpropagation Through Time (BPTT) [8] which basically means running the network on a given sequence and updating the parameters based on the total loss. Here is a function that calculates the total cross-entropy loss for a given (sub)sequence:

```
function loss(param,state,sequence,range)
    total = 0.0; count = 0
    atype = typeof(getval(param[1]))
    input = convert(atype,sequence[first(range)])
    for t in range
        ypred = predict(param,state,input)
        ynorm = ypred .- log(sum(exp(ypred),2))
        ygold = convert(atype,sequence[t+1])
        total += sum(ygold .* ynorm)
        count += size(ygold,1)
        input = ygold
    end
    return -total / count
end
```

Here `param` and `state` hold the parameters and the state of the model, `sequence` and `range` give us the input sequence and a range over it to process. We convert the entries in the sequence to inputs that have the same type as the parameters one at a time (to conserve GPU memory if one is using device arrays). We use each token in the given range as an input to predict the next token. The average cross-entropy loss per token is returned. At this point we can train our RNN model on any given piece of text (or other discrete sequence) by using `lossgradient=grad(loss)` and writing a training script that initializes parameters and state, and processes the data. For efficiency it is best to minibatch the training data and run BPTT on small subsequences. The complete character-level language model example with weight and state initialization, data loading and minibatching, model training and text generation takes about 200 lines of Julia code and is available in the Knet repository along with the rest of the examples in this section.

The clarity and economy of defining and training machine learning models using a high level language like Julia become more evident with more complex models like RNNs, bidirectional LSTMs and

attention models. I hope the examples in this section provide some evidence that GPU support and differentiation for a high level language are sufficient to construct sophisticated models and once this support is fully realized, specialized modules and layers are largely unnecessary. Nevertheless, Knet provides a few additional optimization methods and utility functions documented in [1].

2.6 Benchmarks

Each of the examples in this section was used as a benchmark to compare the performance of Knet with other frameworks. The table below shows the number of seconds it takes to train a given model for a particular dataset using a GPU with number of epochs and minibatch size for Knet [1], Theano [9], Torch [10], Caffe [11] and TensorFlow [12]. Knet has comparable performance to other commonly used frameworks.[1]

model	dataset	epochs	batch	Knet	Theano	Torch	Caffe	TFlow
LinReg	Housing	10K	506	2.84	1.88	2.66	2.35	5.92
Softmax	MNIST	10	100	2.35	1.40	2.88	2.45	5.57
MLP	MNIST	10	100	3.68	2.31	4.03	3.69	6.94
LeNet	MNIST	1	100	3.59	3.03	1.69	3.54	8.77
CharLM	Hiawatha	1	128	2.25	2.42	2.23	1.43	2.86

3 Implementation

Knet implements an AutoGrad package and a KnetArray data type for its functionality and performance. AutoGrad computes the gradient of Julia functions and KnetArray implements high performance GPU arrays with custom memory management. This section briefly describes them.

3.1 AutoGrad

As we have seen, many common machine learning models can be expressed as differentiable programs that input parameters and data and output a scalar loss value. The loss value measures how close the model predictions are to desired values with the given parameters. Training a model can then be seen as an optimization problem: find the parameters that minimize the loss. Typically, a gradient based optimization algorithm is used for computational efficiency: the direction in the parameter space in which the loss reduction is maximum is given by the negative gradient of the loss with respect to the parameters. Thus gradient computations take a central stage in software frameworks for machine learning. In this section I will briefly outline existing gradient computation techniques and motivate the particular approach taken by Knet.

Computation of gradients in computer models is performed by four main methods [13]:

1. manual differentiation (programming the derivatives)
2. numerical differentiation (using finite difference approximations)
3. symbolic differentiation (using expression manipulation)
4. automatic differentiation (detailed below)

Manually taking derivatives and coding the result is labor intensive, error-prone, and all but impossible with complex deep learning models. Numerical differentiation is simple: $f'(x) = (f(x+\epsilon) - f(x-\epsilon))/(2\epsilon)$ but impractical: the finite difference equation needs to be evaluated for each individual parameter, of which there are typically many. Pure symbolic differentiation using expression manipulation, as implemented in software such as Maxima, Maple, and Mathematica

[1] The benchmarking was done on g2.2xlarge GPU instances on Amazon AWS: Intel Xeon E5-2670 CPU at 2.6GHz with 15GB RAM, NVIDIA GRID K520 GPU with 1536 CUDA cores and 4GB RAM. The code to run all benchmarks is available as machine image `deep_AMI` at AWS N.California or at `https://github.com/ozanarkancan/Knet8-Benchmarks`. The datasets are available online; Housing: `https://archive.ics.uci.edu/ml/datasets/Housing`, MNIST: `http://yann.lecun.com/exdb/mnist`, Hiawatha: `http://www.gutenberg.org/files/19/19.txt`. The MLP uses a single hidden layer of 64 units. CharLM uses a single layer LSTM language model with embedding and hidden layer sizes set to 256 and trained using BPTT with a sequence length of 100. Each dataset was minibatched and transferred to GPU prior to benchmarking when possible.

is impractical for different reasons: (i) it may not be feasible to express a machine learning model as a closed form mathematical expression, and (ii) the symbolic derivative can be exponentially larger than the model itself leading to inefficient run-time calculation. This leaves us with automatic differentiation.

Automatic differentiation is the idea of using symbolic derivatives only at the level of elementary operations, and computing the gradient of a compound function by applying the chain rule to intermediate numerical results. For example, pure symbolic differentiation of $\sin^2(x)$ could give us $2\sin(x)\cos(x)$ directly. Automatic differentiation would use the intermediate numerical values $x_1 = \sin(x)$, $x_2 = x_1^2$ and the elementary derivatives $dx_2/dx_1 = 2x_1$, $dx_1/dx = \cos(x)$ to compute the same answer without ever building a full gradient expression.

To implement automatic differentiation the target function needs to be decomposed into its elementary operations, a process similar to compilation. Most machine learning frameworks [2] compile models expressed in a restricted mini-language into a computational graph of elementary operations that have pre-defined derivatives. There are two drawbacks with this approach: (i) the restricted mini-languages tend to have limited support for high-level language features such as conditionals, loops, helper functions, array indexing, etc. (e.g. the infamous `scan` operation in Theano) (ii) the sequence of elementary operations that unfold at run-time needs to be known in advance, and cases where the sequence of operations is data dependent are non-trivial to handle.

There is an alternative: high-level languages, like Julia and Python, already know how to decompose functions into their elementary operations. If we let the users define their models directly in a high-level language, then record the elementary operations during loss calculation at run-time, the computational graph can be constructed from the recorded operations dynamically. The cost of recording is not prohibitive: The table on the right gives cumulative times for elementary operations of an MLP with quadratic loss. Recording only adds 15% to the raw cost of the forward computation. Backpropagation roughly doubles the total time as expected.

op	secs
`a1=w1*x`	0.67
`a2=w2.+a1`	0.71
`a3=max(0,a2)`	0.75
`a4=w3*a3`	0.81
`a5=w4.+a4`	0.85
`a6=a5-y`	0.89
`a7=sumabs2(a6)`	1.18
+recording	1.33
+backprop	2.79

This is the approach taken by the popular `autograd` Python package [14] and the `AutoGrad` Julia package [15] implemented for Knet. In these implementations `g=grad(f)` generates a gradient function `g`, which takes the same inputs as the function `f` but returns the gradient. The gradient function `g` triggers recording by boxing the parameters in a special data type and calls `f`. The elementary operations in `f` are overloaded to record their actions and output boxed answers when their inputs are boxed. The sequence of recorded operations is then used to compute gradients. In the Julia AutoGrad package, derivatives can be defined independently for each method of a function (determined by argument types) making full use of Julia's multiple dispatch. New elementary operations and derivatives can be defined concisely using Julia's macro and meta-programming facilities. See [15] for details.

3.2 KnetArray

GPUs have become indispensable for training large deep learning models. Even the small examples implemented in this paper run up to 17x faster on the GPU compared to the CPU on the architecture described in Footnote 1. However GPU implementations have a few potential pitfalls: (i) GPU memory allocation is slow, (ii) GPU-RAM memory transfer is slow, (iii) reduction operations (like `sum`) can be very slow unless implemented properly [16].

Knet implements KnetArray as a Julia data type that wraps GPU array pointers. KnetArray is based on the more standard CudaArray [17] with a few important differences: (i) KnetArrays have a custom memory manager, similar to ArrayFire [18], which reuse pointers garbage collected by Julia to reduce the number of GPU memory allocations, (ii) array ranges (e.g. `a[:,3:5]`) are handled as views with shared pointers instead of copies when possible, and (iii) a number of custom CUDA kernels written for KnetArrays implement element-wise, broadcasting, and scalar and vector reduction operations efficiently. As a result Knet allows users to implement their models using high-level code, yet be competitive in performance with other frameworks as demonstrated in the benchmarks section.

[2] Theano, Torch, Caffe, Tensorflow and older versions of Knet prior to v0.8 are some examples with such gradient generating compilers.

Acknowledgments

This work was supported by the Scientific and Technological Research Council of Turkey (TÜBİTAK) grants 114E628 and 215E201. I am grateful to USC/ISI for their hospitality during my sabbatical where most of the early Knet development took place. I thank my students Ozan Arkan Can, Erenay Dayanık, İlker Kesen, and Ömer Kırnap for their help with running the benchmark tests for this paper and their feedback and contributions to Knet development.

References

[1] Deniz Yuret. Knet: Koç University deep learning framework. `https://github.com/denizyuret/Knet.jl`, 2016.

[2] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *arXiv preprint arXiv:1411.1607*, 2014.

[3] Léon Bottou. Online learning and stochastic approximations. *On-line learning in neural networks*, 17(9):142, 1998.

[4] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[5] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[6] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[7] Andrej Karpathy. The unreasonable effectiveness of recurrent neural networks. `http://karpathy.github.io/2015/05/21/rnn-effectiveness`, 2015.

[8] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[9] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.

[10] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A MATLAB-like environment for machine learning. In *BigLearn, NIPS Workshop*, number EPFL-CONF-192376, 2011.

[11] Yangqing Jia et al. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

[12] Martín Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[13] Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *arXiv preprint arXiv:1502.05767*, 2015.

[14] Dougal Maclaurin, David Duvenaud, and Ryan P Adams. Autograd: Effortless gradients in numpy. In *ICML 2015 AutoML Workshop*, 2015.

[15] Deniz Yuret. Autograd: an automatic differentiation package for Julia. `https://github.com/denizyuret/AutoGrad.jl`, 2016.

[16] Mark Harris et al. Optimizing parallel reduction in CUDA. *NVIDIA Developer Technology*, 2(4), 2007.

[17] Tim Holy. CUDArt Julia package for the CUDA runtime API, 2014.

[18] Pavan Yalamanchili, Umar Arshad, Zakiuddin Mohammed, Pradeep Garigipati, Peter Entschev, Brian Kloppenborg, James Malcolm, and John Melonakos. ArrayFire - A high performance software library for parallel computing with an easy-to-use API, 2015.