Chainer: a Next-Generation Open Source Framework for Deep Learning

Seiya Tokui
Preferred Networks
Tokyo, Japan.
`tokui@preferred.jp`

Kenta Oono
Preferred Networks
Tokyo, Japan.
`oono@preferred.jp`

Shohei Hido
Preferred Networks America
San Mateo, CA.
`hido@preferred.jp`

Justin Clayton
Preferred Networks America
San Mateo, CA.
`jclayton@preferred-america.com`

Abstract

Software frameworks for neural networks play key roles in the development and application of deep learning methods. However, as new types of deep learning models are developed, existing frameworks designed for convolutional neural networks are becoming less useful. In this paper, we introduce Chainer, a Python-based, standalone open source framework for deep learning models. Chainer provides a flexible, intuitive, and high performance means of implementing a full range of deep learning models, including state-of-the-art models such as recurrent neural networks and variational autoencoders.

1 Introduction

Deep learning is driving the third wave of artificial intelligence research [13]. Recent papers indicate that deep learning is moving beyond its early successes in pattern recognition and towards new applications in diverse domains and industries. In order to put these research ideas into practice, a software framework for deep learning is needed.

Implementing neural networks (NNs) requires a set of specialized building blocks, including multidimensional arrays, activation functions, and autonomous gradient computation. To avoid duplicating these tools, many developers use open source deep learning frameworks such as Caffe [9] or Torch [6]. Because deep learning was first used successfully in the areas of computer vision and speech recognition, existing deep learning frameworks were designed mainly for feed-forward networks such as convolutional neural networks (CNNs), which are effective for analyzing data samples of fixed length, such as images.

More recently, new types of deep learning models other than CNNs have become major topics of research. Following the epic results in game playing presented by Google DeepMind at the NIPS Deep Learning workshop 2013 [16], deep reinforcement learning has become an important and promising area of research. In addition, after recurrent neural networks (RNNs) showed promising results on variable-length data such as natural language text, the use of these models has increased. RNNs with Long Short-Term Memory (LSTM) are currently being used successfully for machine translation [20] and conversation models [24].

However, the new deep learning architectures have moved beyond the original and mostly unchanged deep learning frameworks. This makes implementation of new models difficult. As most of the existing deep learning frameworks were designed for image processing using CNNs, their methods for abstracting data structures and training models are not optimal for implementing the newer generation of deep learning models. In addition, most existing frameworks use a domain specific language for representing deep learning models, along with an interpreter to translate them into a data structure stored in memory. Therefore, developers using these frameworks cannot use standard programming language debuggers–a significant problem as debugging is a major part of developing and tuning deep learning models.

In this paper, we introduce Chainer [22], a versatile open source software framework for deep learning that provides simple and efficient methods for implementing complex algorithms, training models, and tuning model parameters. The remainder of the paper is organized as follows. Section 2 describes the standard architecture on which most existing deep learning frameworks are built and its drawbacks. Section 3 introduces a novel architecture, used in Chainer, that avoids these limitations. Section 4 shows benchmark results obtained using Chainer, and Section 5 gives a summary and directions for future work.

2 Deep Learning Framework

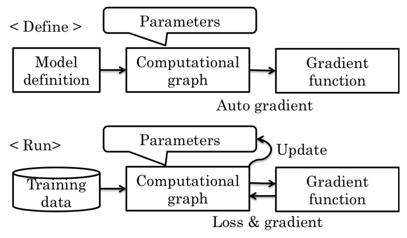
In typical NN frameworks, models are built in two phases, a paradigm we name *Define-and-Run* (Figure 1a). In the Define phase, a computational graph is constructed; in the Run phase, the model is trained on a training data set. The Define phase is the instantiation of a neural network object based on a model definition that specifies the inter-layer connections, initial weights, and activation functions. After the graph is built on memory and the forward computation is set, the corresponding backward computation for back propagation can be defined by automatic gradient functionalities. In the Run phase, given a set of training examples, the model is trained by minimizing the loss function using optimization algorithms such as stochastic gradient descent.

Under the *Define-and-Run* paradigm, static NN models, such as CNNs, can be easily implemented. The model definition may be written in a specific markup language, such as Protobuf for Caffe or YAML for PyLearn2 [8]. The deep learning framework then acts as an interpreter and processes the model definition as an object of a single class, which can be regarded as an independent NN program. The NN program receives inputs (data examples), processes these inputs (forward/backward computation), changes the models internal state (updating), and outputs the results (predictions).

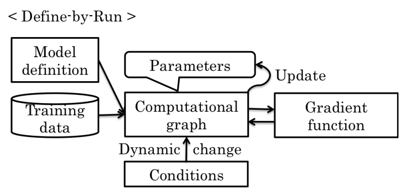
Although the *Define-and-Run* paradigm works well for implementing CNN models, when it is used for implementing other types of NN models three major problems become evident.

The first is inefficient memory usage. Because the computational graph is built before the model is trained, all layers of the NN must remain in memory even if some layers are needed only at the beginning or end of the training process. For example, RNN models are usually trained with backpropagation through time (BPTT) [25], which uses a heuristic to threshold the propagation for computational efficiency. However, in *Define-and-Run* frameworks, the entire computational graph must remain in memory regardless of whether certain layers are no longer being used in BPTT.

The second problem is that frameworks designed around the *Define-and-Run* scheme have limited extensibility. In order to maintain backward compatibility, the developers of the initial deep learning frameworks do not have the freedom to extend the *Define-and-Run* scheme to accomodate more complex models. Therefore, users who want to implement original NN architectures have



(a) *Define-and-Run*: existing approach



(b) *Define-by-Run*: new approach

Figure 1: Relationships between computational graph construction and training

two choices: extend the framework by forking the repository, or impose implementation by hacking the existing code base. Unfortunately, these options are sub-optimal for both the community and the developers. Forking divides contributions and efforts while hacking decreases efficiency of development and may create difficulties in source code maintenance. For example, many forked versions of Caffe exist for implementing specific algorithms, but these versions are mutually incompatible and cannot be merged. In addition, although Theano [2], on which PyLearn and other Python frameworks depend, supports variable-length inputs and loops in NNs with a special operation called scan(), many users find this operation insufficient to implement complex NNs more complex than standard RNNs.

The third problem is that under the *Define-and-Run* scheme, the inner workings of the neural network are not accessible to the user. This presents several difficulties in the creation of an effective model. For example, in order to debug and tune a model effectively, a user needs to be able to see what is happening inside the model. However, as a large object of a single class, the computational graph contains the entirety of the model's information–its structure, weights, gradients, and inter-node operations–meaning that it is essentially a black box. Consequently, development tools such as Profiler and Debugger cannot determine what is wrong with the model or how it could be improved.

A new example of the use of the *Define-and-Run* scheme is TensorFlow [1], an open source deep learning library recently released by Google. Although TensorFlow is designed to enhance performance by optimizing the allocation of computations between multiple nodes and GPUs, the initial version still has some of the limitations found in other *Define-and-Run* frameworks. For example, it still constructs the computational graph before training, which can lead to inefficient memory usage when using naive implementation.

3 Chainer

In this section, we introduce Chainer, a second-generation deep learning framework based on a novel paradigm. Compared to existing frameworks, Chainer provides an easier and more straightforward way to implement the more complex deep learning architectures currently being researched. Python was chosen as the programming language for Chainer due to the simplicity, popularity, and benefits

```
# (1) Function Set definition
model = FunctionSet(
  l1=F.Linear(784, 100),
  l2=F.Linear(100, 100),
  l3=F.Linear(100, 10)).to_gpu()
# (2) Optimizer Setup
opt = optimizers.SGD()
opt.setup(model)
# (3) Forward computation
def forward(x, t):
  h1 = F.relu(model.l1(x))
  h2 = F.relu(model.l2(h1))
  y = model.l3(h2)
  return F.softmax_cross_entropy(y, t)
```

```
# (4) Training loop
for epoch in xrange(n_epoch):
  for i in xrange(0, N, b_size):
    x = Variable(to_gpu(...))
    t = Variable(to_gpu(...))
    opt.zero_grads()
    loss = forward(x, t)
    loss.backward()
    opt.update()
```

(a) Multi-layer perceptron

```
# (1) Function Set definition
model = FunctionSet(
  emb=F.EmbedID(1000, 100),
  x2h=F.Linear(100, 50),
  h2h=F.Linear(50, 50),
  h2y=F.Linear(50, 1000)).to_gpu()
# (2) Optimizer Setup
opt = optimizers.SGD()
opt.setup(model)
# (3) One step forward
def fwd1step(h, w, t):
  x = F.tanh(model.emb(w))
  h = F.tanh(model.x2h(x) + model.h2h(h))
  y = model.h2y(h2)
  return h, F.softmax_cross_entropy(y, t)
```

```
# (4) Full RNN forward computation
def forward(seq)
  h = Variable() # init state
  loss = 0
  for curw, nextw in zip(seq,seq[1:]):
    w = Variable(curw)
    t = Variable(nextw)
    h, new_loss = fwd1step(h, w, t)
    loss += new_loss
  return loss
```

(b) Simple RNN

Figure 2: Code examples of Chainer

of using the existing multi-dimensional array library (NumPy [17]). To allow GPU use for faster computation, Chainer implements CuPy, which is partially compatible with NumPy. Chainer also supports popular optimization methods, such as stochastic gradient descent (SGD) [3], AdaGrad [7], RMSprop [21], and Adam [10] as well as other frameworks do. Automatic gradients can also be computed for back propagation. Many numerical operations for building neural networks, such as convolutions, losses, and activation functions are implemented as *Function*.

The most unique aspect of Chainer is the way in which a model's definition is closely related to its training. In contrast to other frameworks, Chainer does not fix a model's computational graph before the model is trained. Instead, the computational graph is implicitly memorized when the forward computation for the training data set takes place as shown in Figure 1b. This new approach is called *Define-by-Run*.

Figure 2a shows sample code for implementing a simple multi-layer perceptron (MLP). (1) shows the initial operations for defining three linear (fully-connected) layers. Note that this consists only of layer definitions and does not include information about inter-layer connections. The function to_gpu() indicates that the model will be stored on GPU. In (2), the optimizer is initialized with SGD and the reference to the MLP model is set. (3) shows the forward() method, the key to defining the relationships between layers. x are the input variables and t are the corresponding target variables. Using ReLu as the activation function, layers L1, L2, and L3 are connected, and the softmax entropy between t and the output y is returned.

(4) represents the training loop, in which a training data set of size N is used for n_epoch epochs in training, and b_size specifies the number of samples used for each gradient computation in mini-batch training. Given training samples x and corresponding target variable t, opt.zero_grads() initializes the gradient with zeros. In the next line, the forward computation along the neural network's structure as defined inside the forward() method is called. At the same time, the computational graph is memorized and returned with a Variable class instance of softmax_cross_entropy. The loss.backward() function executes the gradient based on the loss from the output layer back to the input layer, for back propagation. Finally, opt.update() adjusts the parameters of the model and another mini-batch follows.

The following example implementation of a RNN highlights the simplicity of defining a model in Chainer. Figure 2b illustrates a simple 4-layer RNN for the prediction of successive elements of a sequence.

In (1) and (2), the initialization of the model and optimizer is nearly the same as in the earlier MLP example, with the exception that here the model uses EmbedID, a word2vec type of layer to transform words into vector representations. The method *fwd1step* in (3) is key to implementing a RNN: the method includes h, the recurrent layer in both the argument and the returned values. In the second line, h is updated using the current input x and the previous state h. This is a straightforward method of representing a RNN, a method that is not available in earlier deep learning frameworks that use text-based model definitions. (4) shows the actual forward computation for input sequence, *seq*. By taking the current element as x and next element t, *fwd1step* is repeated and losses at each step are collected. The returned loss contains all of the information necessary for updating the RNN model using backpropagation through time (BPTT).

These sample codes show that the expression of forward computation, the most important part of running any NN model, can be written intuitively as simple Python code. The remaining elements, such as functions, losses, and training loops, are well-abstracted, very simple, and consistent across most types of models. In addition, because Chainer is written in Python, standard Python debugging and profiling tools can be used when creating models. The flexibility and simplicity of Chainer allow users to implement complex algorithms intuitively and efficiently.

4 Benchmark

In this section, Chainer is compared with Caffe using various CNNs. The experimental setting follows a public benchmark repository [4] [1]. For elementary comparison, five standard CNNs are used. For a more practical comparison, popular CNNs for ImageNet datasets (AlexNet [12], Overfeat [19],

[1]Chainer is submitted to this repository and currently waiting for being added to the result.

and VGG [18]) are used. Table 1 summarizes execution times for these models. In Chainer, only the first batch requires memory pool allocation and kernel transfer on GPU; therefore, the first batch is considered a special case and its computation time is not included. Accordingly, for Chainer, the mean computation time for batches 2-11 is shown. For Caffe, since the first batch does not require longer training time, the mean of computation time for batches 1-10 is shown. Note that these results are for Chainer v1.5.0.2 with CuPy; measurements for comparison with other frameworks such as Keras [5] and TensorFlow [1] will be published in the near future.

Table 1: Mean time for computations on various networks (msec).

			Basic convolutional nets					ImageNet		
			conv1	conv2	conv3	conv4	conv5	AlexNet	Overfeat	VGG
		batchsize	64	32	64	128	128	128	128	16
forward	Chainer	1st	245.62	253.08	266.21	199.95	199.80	500.77	667.10	394.20
	Chainer	2-11th	72.31	69.00	85.17	29.14	30.68	222.14	406.91	145.77
	Caffe	1-10th	47.14	34.66	59.54	25.19	28.48	122.89	368.11	84.54
backward	Chainer	1st	103.01	101.14	90.44	39.00	48.39	293.87	553.65	238.88
	Chainer	2-11th	85.48	92.33	82.58	27.89	34.74	277.22	545.51	227.08
	Caffe	1-10th	101.44	105.76	86.46	29.16	31.43	205.05	471.61	191.18

Table 1 shows that the first batch in Chainer is up to 7.0 times slower than subsequent batches. However, the overhead induced by the first batch becomes insignificant as thousands of mini-batches are executed. Compared to Caffe, the forward computation time in Chainer is 1.1 to 1.8 times slower, although the backward computation time is comparable or faster in some cases. The difference in forward computation time seems reasonable as Chainer is written in Python while Caffe is written in C++. Because Chainer does not need to recompile after changes are made to the code, the total time required for trial-and-error testing in implementing and tuning new algorithms is likely to be similar to that required in Caffe.

5 Conclusion

This paper introduced Chainer, a next-generation deep learning framework that enables the easy implementation of new kinds of algorithms as complex neural networks. Chainer is already being used successfully in a variety of leading-edge applications, including deep reinforcement learning [16], word2vec distributed representations [15], recurrent neural network language models [14], human pose estimation [23], and variational auto-encoders [11]. Because dedicated developers (including the authors) and users from around the world are actively collaborating on Github to improve Chainer, we anticipate that Chainer will become more versatile and useful in the future. In particular, the performance improvement gained by improving CuPy allows users to apply many kinds of deep learning models with CPU/GPU-agnostic code. The authors invite all members of the deep learning community to try Chainer and to contribute to its development.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] F. Bastien, P. Lamblin, R. Pascanu, J. Bergstra, I. J. Goodfellow, A. Bergeron, N. Bouchard, and Y. Bengio. Theano: new features and speed improvements. NIPS Workshop on Deep Learning and Unsupervised Feature Learning, 2012.

[3] L. Bottou. Stochastic gradient tricks. In *Neural Networks, Tricks of the Trade, Reloaded*, pages 430–445. Springer, 2012.

[4] S. Chintala. convnet-benchmarks. https://github.com/soumith/convnet-benchmarks.

[5] F. Chollet. Keras. http://keras.io/.

[6] R. Collobert. Torch. NIPS Workshop on Machine Learning Open Source Software, 2008.

[7] J. C. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12:2121–2159, 2011.

[8] I. J. Goodfellow, D. Warde-Farley, P. Lamblin, V. Dumoulin, M. Mirza, R. Pascanu, J. Bergstra, F. Bastien, and Y. Bengio. Pylearn2: a machine learning research library. *CoRR*, abs/1308.4214, 2013.

[9] Y. Jia. Caffe: An open source convolutional architecture for fast feature embedding, 2013.

[10] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[11] D. P. Kingma and M. Welling. Auto-encoding variational bayes. *ICLR*, 2014.

[12] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, pages 1106–1114, 2012.

[13] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521:436–444, 2015.

[14] T. Mikolov, M. Karafiát, L. Burget, J. Černocký, and S. Khudanpur. Recurrent neural network based language model. In *INTERSPEECH*, pages 1045–1048, 2010.

[15] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. *NIPS*, pages 3111–3119, 2013.

[16] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning, 2013. NIPS Deep Learning Workshop.

[17] T. Oliphant. *Guide to NumPy*. Trelgol Publishing, 2006.

[18] T. Sercu, C. Puhrsch, B. Kingsbury, and Y. LeCun. Very deep multilingual convolutional neural networks for LVCSR. *CoRR*, abs/1509.08967, 2015.

[19] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun. Overfeat: Integrated recognition, localization and detection using convolutional networks. *CoRR*, abs/1312.6229, 2013.

[20] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. *NIPS*, pages 3104–3112, 2014.

[21] T. Tieleman and G. Hinton. Lecture 6.5-RMSprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*, 4, 2012.

[22] S. Tokui. Chainer. http://chainer.org/.

[23] A. Toshev and C. Szegedy. Deeppose: Human pose estimation via deep neural networks. In *CVPR*, pages 1653–1660, 2014.

[24] O. Vinyals and Q. V. Le. A neural conversational model. *CoRR*, abs/1506.05869, 2015.

[25] P. J. Werbos. Generalization of backpropagation with application to a recurrent gas market model. *Neural Networks*, 1(4):339–356, 1988.