Neural Network Libraries: A Deep Learning Framework Designed from Engineers' Perspectives

Akio Hayakawa* Masato Ishii Yoshiyuki Kobayashi

Akira Nakamura Takuya Narihira Yukio Obuchi

Andrew Shin Takuya Yashima Kazuki Yoshiyama

Sony Corporation

Abstract

While there exist a plethora of deep learning tools and frameworks, the fast-growing complexity of the field brings new demands and challenges, such as more flexible network design, speedy computation on distributed setting, and compatibility between different tools. In this paper, we introduce Neural Network Libraries[2], a deep learning framework designed from engineer's perspective, with emphasis on usability and compatibility as its core design principles. We elaborate on each of our design principles and its merits, and validate our attempts via experiments.

1 Introduction

Deep learning has revolutionized the field of artificial intelligence, with state-of-the-art performances in image recognition (He et al. [2016], Xie et al. [2017]), speech recognition (Chiu et al. [2017]), and machine translation (Bahdanau et al. [2015]), just to name a few. Its application is not restricted to research area, and has taken up a substantial part of the real world platforms, such as automated driving and mobile applications. Moreover, with recent surge of generative models (Goodfellow et al. [2014], Karras et al. [2018a,b]), its potential as a tool for contents creation is also getting attention.

Deep learning research and development has seemingly formed a healthy ecosystem in which any researcher in the world can make a significant impact and contribution to the community, with rich amount of resources freely available (Abadi et al. [2015], Jia et al. [2014], Tokui et al. [2015], Seide and Agarwal [2016], Paszke et al. [2019]). On the other hand, as the field advances, more complexities and variations arise, and the demand for a more flexible and efficient tool grows stronger. For example, users need to define complex networks more concisely, and it is also necessary to easily handle static and dynamic computational graphs. Also, with the advent of massively large models (Devlin et al. [2019], Brown et al. [2020]), and the costs for accessing remote GPU servers skyrocketing, the ability to perform computation in a speedy manner, particularly on distributed setting, has become a pivotal factor.

Another issue that emerges from the massive expansion of deep learning tools is compatibility. With countless number of tools developed and released anew on a daily basis, it is possible that we end up with disjoint clusters of research and developments. Such can be concerning for a number of reasons;

[*] Authors are listed in alphabetical order.

[2] https://nnabla.org

first, it can decelerate the speed of progress. Just as lack of appropriate translation between different languages can hinder communication, the absence of a protocol for bridging the gap between tools and frameworks can slow down the spreading of novel ideas. Second, in a similar manner, some of the innovations may fail to receive the deserved attention, due to their failure to merge with the mainstream cluster of tools. A tool to easily make it compatible with other tools and frameworks will alleviate such risks.

We have developed and open-sourced Neural Network Libraries (NNL), focusing on the issues described above, namely usability and compatibility, particularly from the perspective of engineers. Our primary aim is to develop a deep learning framework that 1) enhances usability by flexible network design and speedy computation, and 2) that provides a wide range of compatibility, easily portable to and from other frameworks, thus lessening the unhealthy prospects described earlier. While such aims are equally critical for researchers as well, we attempt to approach the issues under the principle of “Engineers First,” as there already exists a plethora of research-oriented tools, with strikingly less amount of emphasis on engineering side. In this paper, we describe our design principles and their merits, how they are reflected in our framework, and further demonstrate them via experiments. We also briefly introduce Neural Network Console[3], a unique GUI-based development platform, which further promotes our design principles.

2 Usability

Design principle of Neural Network Libraries can best be described as "Engineers First". In other words, Neural Network Libraries is goal-oriented so that users can focus on constructing the network just by stacking the various layers sequentially. Modern-day neural network frameworks utilize 2 types of computation graphs. One is a so-called 'static' graph, and the good example is TensorFlow (Abadi et al. [2015]). Another is a 'dynamic' graph, adopted in PyTorch (Paszke et al. [2019]). Neural Network Libraries supports both computation graphs, offering flexible usability to users. While we put forward the principle "Engineer First" here, researchers should also find it useful for their work since we offer reference implementations of many state-of-the-art models for various tasks such as image recognition or generative models.

2.1 Building Neural Networks with Fewer Lines

When users build a machine learning system, especially using deep learning, they need to 1) build a neural network, 2) prepare the data, and 3) train the network itself. Training takes a large amount of time, and so does data preparation. Also, training neural networks is likely to require several trial-and-errors. It is thus highly desirable to be time-efficient when building neural networks. In Neural Network Libraries, constructing the neural networks is easy and straightforward.

```
import numpy as np
import nnabla as nn
import nnabla.parametric_functions as PF

# Define input variable and computational graph
x = nn.Variable((16, 10), need_grad=True)
y = PF.affine(x, 5)

# Compute output for some random input
x.d = np.random.random(x.shape)
y.forward()

# Compute gradient with respect to input and parameters
y.backward()

# show all the trainable parameters assigned to the existing layers
nn.get_parameters()
```

Listing 1: Forward/Backward of the affine function

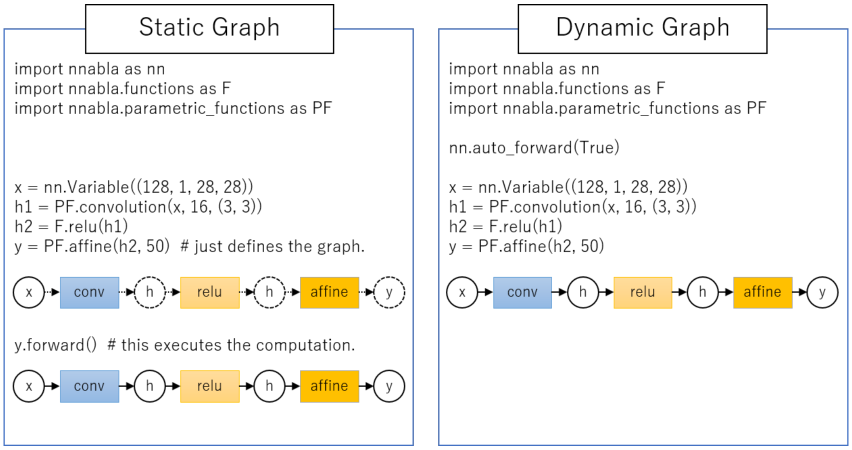
[3] https://dl.sony.com/

As shown in Listing 1, users simply need to prepare a data array and stack the layers to which the data is fed. Once the network is defined, it is represented as a computation graph internally, and calling **forward** method on the output data array executes computation. After forward computation is done, calling **backward** method performs backpropagation, where the gradients are calculated and subsequently stored to the corresponding arrays.

Neural Network Libraries offers three basic building blocks, namely, **Variables**, **Functions**, and **Parametric functions**. **Variables** represent data and their gradients with multi-dimensional arrays, *e.g.*, tensors. **Functions** are mathematical operations that can be applied to variables. **Parametric functions** are functions accompanied with additional trainable parameters. Listing 1 shows example codes of using variables with a parametric function. Using these components, convolutional neural networks can also be easily implemented. A typical example is LeNet (Lecun et al. [1998]), whose code sample is shown in Listing 4, in Sec 3.2.

As can be seen in these examples, writing code with trainable layers (parametric functions) is straightforward since it does not require pre-defined layers and can be executed in a linear manner. In other words, users do not have to spend time on preparing the trainable parameters and assigning them to corresponding layers. All the trainable parameters are registered to a globally accessible dictionary. The last line in the Listing 1 shows the trainable parameters assigned to the existing layers. Thus, users can construct the neural network in fewer lines, which enables them to quickly proceed to the training phase, and easily re-arrange the network design when necessary.

2.2 Flexible Computation Methods



```
import nnabla as nn
import nnabla.functions as F
import nnabla.parametric_functions as PF


x = nn.Variable((128, 1, 28, 28))
h1 = PF.convolution(x, 16, (3, 3))
h2 = F.relu(h1)
y = PF.affine(h2, 50)  # just defines the graph.
```

```
y.forward()  # this executes the computation.
```

```
import nnabla as nn
import nnabla.functions as F
import nnabla.parametric_functions as PF

nn.auto_forward(True)

x = nn.Variable((128, 1, 28, 28))
h1 = PF.convolution(x, 16, (3, 3))
h2 = F.relu(h1)
y = PF.affine(h2, 50)
```

Figure 1: Illustration of static and dynamic graphs.

Early neural network frameworks, such as Theano, Torch and TensorFlow utilize static computation graph. In static graph, users first need to define the entire graph and then use that graph for computation for each input data. The left block of Figure 1 shows a typical usage of static graph and its behavior. Actual computation is not executed at the time of graph definition, and the users need to call forward method explicitly. In static graph, the network architecture is completely fixed once defined. Instead, the computation speed is expected to be fast. Recent frameworks such as Chainer and PyTorch utilize dynamic computation graph in which users can dynamically change the network architecture and execute the computation on-the-fly. The right block of Figure 1 depicts its usage. Not only does it help users to understand what is happening inside the network, but it also provides more flexible usage of the neural network. For example, networks containing randomly dropping layers for each minibatch can be implemented. Switching to dynamic mode is also very simple, since it only requires addition of a single line as shown in Figure 1. Necessity for modification of code is minimized.

2.3 Speed Optimized with CuDNN / Distributed Training

Most of the recent neural network frameworks utilize GPU to accelerate hardware for efficient training, and so does Neural Network Libraries. However, unlike other frameworks in which users need to assign tensors to GPU explicitly, Neural Network Libraries minimize users' efforts necessary to assign tensors to an appropriate device. Users simply need to switch the backend at the beginning of the code, which can be done by a single modification in extension context setting. (See Listing 2.) Once device configuration is set, no other additional setup is needed and all **Variables** are automatically assigned to the chosen device.

```
from nnabla.ext_utils import get_extension_context
nn.set_default_context(get_extension_context('cudnn'))
```

Listing 2: Switching backend in NNL

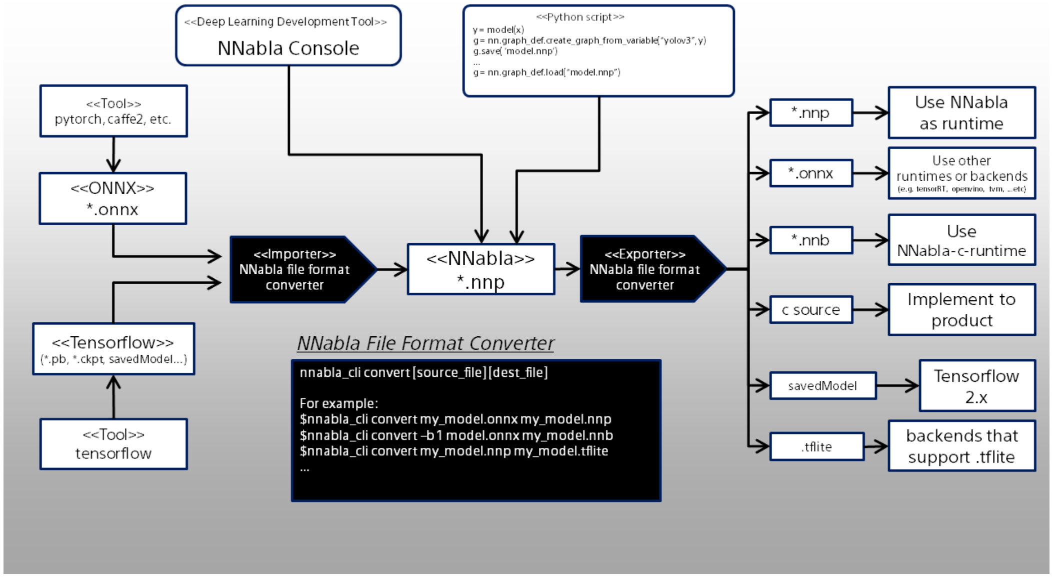
In deep learning, it is generally true that the more complex the model, the longer the training takes. This makes it particularly important to support data-parallel distributed training for deep learning frameworks. Aided by NCCL and MPI, Neural Network Libraries provides enhanced usability on distributed training setting as well, where inter-device communication and parallel computations can be easily implemented with addition of only a few lines as shown in Listing 3.

```
import nnabla.communicators as C
ctx = get_extension_context("cudnn")
comm = C.MultiProcessDataParalellCommunicator(ctx)
comm.init()
...
params = [x.grad for x in nn.get_parameters().values()]
loss.backward(clear_buffer=True)
comm.all_reduce(params)
```

Listing 3: Illustration of required modification for distributed training

More details about the performance in term of computation speed can be found in Sec 4.

3 Compatibility



```
y = model(x)
g = nn.graph_def.create_graph_from_variable("yolov3", y)
g.save( 'model.nnp')
...
g = nn.graph_def.load("model.nnp")
```

```
nnabla_cli convert [source_file] [dest_file]

For example:
$nnabla_cli convert my_model.onnx my_model.nnp
$nnabla_cli convert –b 1 model.onnx my_model.nnb
$nnabla_cli convert my_model.nnp my_model.tflite
...
```

Figure 2: Overview of Neural Network Libraries' compatibility.

We provide tools for a variety of file format conversions, by which networks trained with Neural Network Libraries can be ported to other libraries including TensorFlow and Caffe, and vice versa, allowing for high development flexibility (Figure 2). Training a model on Neural Network Libraries generates an *.nnp* file containing settings and parameters, which is portable to C++. In addition, we have also made it portable for ONNX[4]. Trained networks can also be imported in our GUI module Neural Network Console, which we introduce later in this paper. With high portability, it is easy to deploy the trained models to applications of interest, or various neural network hardware-accelerated backends.

This file format converter uses protobuf defined in Neural Network Libraries as intermediate format. If ONNX file contains a function unsupported by Neural Network Libraries, it may cause error in conversion, so users may use querying commands provided by Neural Network Libraries to check whether it contains unsupported function. This converter also provides some intermediate process functionalities. We currently provide the following file format conversions:

- NNP variations to valid NNP
- ONNX to NNP and vice versa
- NNP to NNB(Binary format for NNabla C Runtime)
- NNP to Tensorflow frozen graph
- Tensorflow checkpoint or frozen graph to NNP
- NNP to TensorFlow Lite models
- Experimental: Convert NNP to C Source code for NNabla C Runtime

Also, OpenCL (Stone et al. [2010]) extension can be easily used by simply building *nnabla* and *nnabla_ext_opencl* from source with OpenCL SDK setup. About 50 functions and solvers are currently implemented in OpenCL (FP-32 only).

3.1 NNP Format

NNP format contains the following information about the model preserved:

- **NNablaProtoBuf**: Root message of NNabla network structure. This message can store GlobalConfig, TrainingConfig, Network(s), Parameter(s), Dataset(s), Optimizer(s), Monitor(s) and Executor(s).
- **Variable**: Internal data structure to store tensor for Neural network I/O and parameters.
- **GlobalConfig**: Configuration of environment for training or inference.
- **TrainingConfig**: Configuration of training.
- **Network**: Network structure.
- **Parameter**: Special variable to store train result. (e.g Weight or Bias of affine layer). From the performance point of view, parameters can be saved in HDF5 (H5) format.
- **Dataset**: Specify dataset for training.
- **Optimizer**: Define network, dataset, and input/output variables for train.
- **Monitor**: Define network, dataset, and input/output variables for monitor training status.
- **Executor**: Define network and input/output variables for train.

3.2 Python-like C++ API

It is easy to overlook the fact that the demand for flexible deployment of deep learning techniques is not limited to software applications, and its unfortunate consequence is the highly limited pool of resources designed from the low-level application or hardware perspective. Python-like C++ API is one of our attempts to alleviate such issue. While the capacity to program in C++ is a critical requirement in many applications integrating hardware, the complexity arising from the difference between C++ and Python programming has been an obstacle for a large number of developers. Our Python-like C++ API reflects the aspirations from such developers to be able to write in C++ as simply as in Python.

Listing 4 and Listing 5 show how to implement LeNet (Lecun et al. [1998]) in Neural Network Libraries with Python and Python-like C++ API respectively. It must be noted that our Python-like C++ API is able to implement the network with the same number of lines as in Python, with the

[4] See `https://nnabla.readthedocs.io/en/latest/support_status.html` for specific function-level and model support status

difference in syntax confined to minimal extent. While many other frameworks including Neural Network Libraries provide low-level C++ API as well, it takes far more number of lines with much higher complexity to implement a network that can be written with a few lines in Python. Such verbosity and complexity frequently become the source of bug, as well as making the maintenance more troublesome. Our Python-like C++ API can perform both training and inference in C++ with the Python-like simplicity and syntax, and as such, it is expected to simplify and accelerate the various dimensions of C++-based applications development, encompassing programming, debugging, and maintenance.

```
h = PF.convolution(x, 16, (5, 5), name="conv1")
h = F.max_pooling(h, (2, 2))
h = F.relu(h, inplace=False)
h = PF.convolution(h, 16, (5, 5), name="conv2")
h = F.max_pooling(h, (2, 2))
h = F.relu(h, inplace=False)
h = PF.affine(h, 50, name="affine3")
h = F.relu(h, inplace=False)
h = PF.affine(h, 10, name="affine4")
```

Listing 4: Implementation of LeNet with Python

```
auto h = pf::convolution(x, 1, 16, {5, 5}, parameters["conv1"]);
h = f::max_pooling(h, {2, 2}, {2, 2}, true, {0, 0});
h = f::relu(h, false);
h = pf::convolution(h, 1, 16, {5, 5}, parameters["conv2"]);
h = f::max_pooling(h, {2, 2}, {2, 2}, true, {0, 0});
h = f::relu(h, false);
h = pf::affine(h, 1, 50, parameters["affine3"]);
h = f::relu(h, false);
h = pf::affine(h, 1, 10, parameters["affine4"]);
```

Listing 5: Implementation of LeNet with Python-like C++ API

3.3 Mixed Precision Training

Mixed precision training (Micikevicius et al. [2018]) is a method to train deep neural networks with half-precision floating points, which nearly halves the memory usage without sacrificing the performance. On top of memory usage, half-precision floating points also lead to the benefits of efficient bandwidth and significant speedup, although there is a potential side effect of unstable gradient computation due to quantization error. Some of the recent processors, such as Tensor Cores in NVIDIA Volta GPUs, support half-precision float computation, and NNL has also been designed to be compatible with mixed precision training.

In NNL, mixed precision training can be used by setting *type_config* as *half* in extension context setting. When using mixed precision training with NVIDIA Volta, storage (weights, activations, gradients) is performed in FP-16. Forward and back-propagation employ TensorCore, where batch normalization is in FP-32. Update is also performed in FP-32, although the weights are managed in both FP-16 and 32. Note that gradients may be too small to be represented in FP-16 and may necessitate loss scaling, which maintains a master copy of weights in FP-32. NNL provides automatic loss scaling updater class for such cases. Listing 6 shows example of mixed precision training in Neural Network Libraries.

```
# Use loss scaling to prevent underflow
loss_scale = 8
loss.backward(loss_scale)
solver.scale_grad(1. / loss_scale)  # some gradient clipping
solver.update()

# Use dynamic loss scaling to prevent overflow/underflow
scaling_factor = 2
counter = 0
interval = 2000
...
```

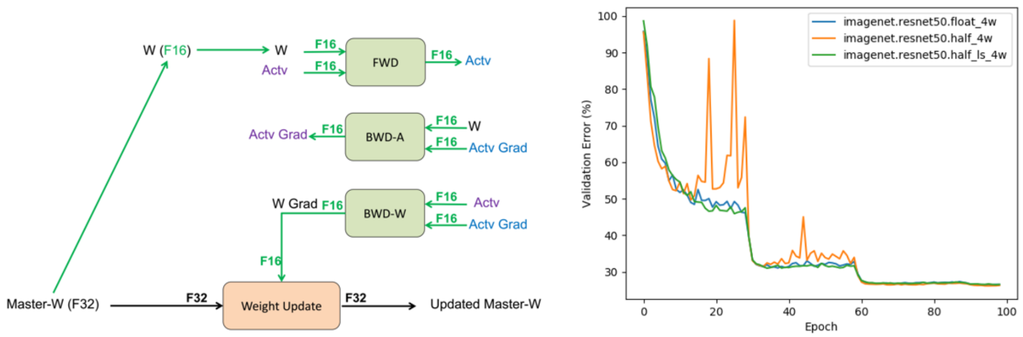


Figure 3: Volta training method (left) and the results of training resnet-50 with 4 volta-distributed training (right).

```
loss.backward(loss_scale, ...)
...
if solver.check_inf_or_nan_grad():
    loss_scale /= scaling_factor
    counter = 0
else:
    solver.scale_grad(1. / loss_scale) # some gradient clipping
    solver.update()
    if counter > interval:
        loss_scale *= scaling_factor
        counter = 0
    counter += 1
```

Listing 6: Mixed precision training in Neural Network Libraries

As we will see in Sec 4, mixed precision training enables faster training than floating points, regardless of whether loss scaling is enabled or disabled.

4 Experiments

To benchmark the performance of Neural Network Libraries, we conducted experiments with ImageNet dataset (ILSVRC 2012 classification dataset) [Deng et al., 2009], which is widely used to evaluate the performance of deep neural networks. We implemented various kinds of popular DNN architectures and evaluated their training speed and test accuracy. For fair comparison with other existing libraries, we follow the setup adopted in NVIDIA Deep Learning Examples[5]. We used DGX-1 with 4 GPUs (NVIDIA V100) for the training and measured training time for 90 and 250 epochs. Since NVIDIA Deep Learning Examples do not report the training time with 4 GPUs, we doubled the reported training time with 8 GPUs and compared our results with them. Our implementation used for this experiment is publicly available in our GitHub repository[6]. Note that this implementation completely relies on NVIDIA's GPUs, and uses NVIDIA's data processing library, called DALI, which works only on Linux.

We first compared the training time of Neural Network Libraries with PyTorch and TensorFlow, the two most popular libraries in deep learning. We trained ResNet-50, one of the most popular network architectures, using Neural Network Libraries and compared its training time with that reported in NVIDIA Deep Learning Examples. Table 1 shows the training time as well as how much it is reduced by adopting the mixed precision. Neural Network Libraries achieves competitive speed when compared to other libraries due to efficient distributed training over multiple GPUs as well as utilization of mixed precision. Figure 3(a) illustrates how we trained the model, along with the results of training.

[5] https://github.com/NVIDIA/DeepLearningExamples

[6] https://github.com/sony/nnabla-examples/tree/master/imagenet-classification

	FP-32	Mixed precision	Speedup by mixed precision
PyTorch	24 h	10 h	x2.3
TensorFlow	20 h	7 h	x3.0
NNabla	23.3 h	7.4 h	x3.1

Table 1: Training time of ResNet-50 for 90 epochs with ImageNet dataset.

We also evaluated the training time and validation error rate with various model architectures. Table 2 shows the results with ResNet variants (ResNeXt [Xie et al., 2017] and SE-ResNet/SE-ResNeXt [Hu et al., 2018]), and Table 3 shows those with several popular lightweight models, such as MobileNet-V3 [Howard et al., 2019] and EfficientNet [Tan and Le, 2019].

Network architecture	Training time (90 epochs)	Training time (250 epochs)	Validation error (250 epochs)
ResNet-18	6.7 h	16.1 h	28.3 %
ResNet-50	7.44 h	20.2 h	21.6 %
ResNeXt-50	12.1 h	33.8 h	21.0 %
SE-ResNet-50	15.0 h	42.2 h	21.2 %
SE-ResNeXt-50	19.7 h	55.7 h	20.1 %

Table 2: Training time and validation error for variations of ResNet architecture

Network architecture	Training time (350 epochs)	Validation error (350 epochs)
MobileNet-V3 small	5.5 h	32.9 %
MobileNet-V3 large	7.6 h	24.9 %
EfficientNet-B0	50.0 h	23.7 %
EfficientNet-B1	79.5 h	21.9 %
EfficientNet-B2	95.5 h	20.9 %
EfficientNet-B3	148.9 h	19.4 %

Table 3: Training time and validation error for lightweight models

5 Extensions

While we designed our framework under the principle of "Engineers First," this does not imply in any way that we neglect to account for the usage by people with lesser or no engineering background. On the contrary, taking full advantage of our engineer-oriented approach, we strove to widen the range of people who can benefit from deep learning technology, by providing novel ways to deploy them. The representative work of such endeavor is Neural Network Console, our GUI-based deep learning development platform, which is also directly compatible with Neural Network Libraries. We have also highly committed to incorporating deep learning into entertainment field, especially contents creation. As such, a wide array of generative models have been constantly made available with pre-trained weight parameters for immediate usage off-the-shelf.

5.1 Neural Network Console

Despite the current buzz around AI and deep learning, people with little or no background in programming have found it challenging to approach it, and frequently ended up on the surface level without understanding its essence by performing design and training neural networks themselves. Even for experienced programmers, developers, and researchers, it requires a very large number of evaluation cycles and can be highly time-consuming to prepare, pre-process, train, and evaluate.

We provide GUI-based deep learning development tool, Neural Network Console (NNC), that enables users to develop their own neural networks more easily. With GUI, users can design neural network structure with visual programming using layers (function blocks) by simple drag and drop. Parameters are automatically calculated and errors can be confirmed immediately. As such, it is particularly useful for learning the concept and learning to design the networks. Also, since all trials are recorded automatically, it is easy to analyze the performance and revert to old records if necessary. Results of experiments are listed and can be compared to past trials. For a classification task, it displays a confusion matrix. Since it checks for validation error, number of parameters, and multiply-adds in real time, users can benefit from significant amount of speed up in evaluation cycle. Highly complex networks and experimental settings such as ResNet-152, generative adversarial networks (GANs) (Goodfellow et al. [2014]), or semi-supervised setting can be implemented as well.

NNC also supports automatic structure search function, which searches for optimal neural network structure automatically by repeating experiments with varying network structures. Multiple network structures are evaluated, simultaneously optimizing for accuracy and computational complexity. Users can select from multiple optimization results. Thus, automatic structure search enables a significant efficiency in optimization of neural networks and is useful for development of embedded applications.

On top of supporting novice developers, NNC provides plugin features that enable users to develop more flexible layers, data processing, and analytical methods. Since user can use Python to design their own plugin, it is easy to develop complex deep learning models such as explainable AI (XAI). For example, we provide a variety of XAI-related plugins, including Grad-CAM (Selvaraju et al. [2017]), LIME (Ribeiro et al. [2016]), and SGD influence (Hara et al. [2019]). Users can examine these cutting-edge techniques simply on GUI.

NNC is mutually portable with NNL. Thus, if users want to visually confirm whether the network designed in NNL is correct, they can simply import the exported file from NNL (*.nntxt* format) into NNC. It can also be useful when they want to footprint the computational workload of the networks designed in NNL. Figure 4 shows example screenshots of Neural Network Console's GUI for training and inference phases respectively.

5.2 Contents Creation

Over the past years, generative models have evolved dramatically from merely generating low-resolution images (Goodfellow et al. [2014]), to generating high-resolution images that are indistinguishable from actual objects (Karras et al. [2018a,b, 2020]). Such dramatic evolution opens up an unbounded possibility for a variety of creative tasks. We have noted that such immense potential of generative models will be clearly beneficial particularly for contents creators, and have striven to provide an environment, in which our users can benefit directly from most important and up-to-date models, readily available with pre-trained weights. Our current lineup[7] encompasses a wide range of generative models, from milestone models such as DCGAN (Radford et al. [2016]), CycleGAN (Zhu et al. [2017]), pix2pix (Isola et al. [2017]), progressive growing of GANs (Karras et al. [2018a]), StyleGAN2 (Karras et al. [2020]), to more recent task-specific models, such as ESR-GAN (Wang et al. [2018]), TecoGAN (Chu et al. [2018]), StarGAN (Choi et al. [2018]), InstaGAN (Mo et al. [2019]), ReenactGAN (Wu et al. [2018]) etc. We plan to aggressively continue to add new models to our lineup.

While generative models can equip contents creators with powerful tools, its benefits and accessibility are still limited to users with moderate level of technical expertise and computational resources. In order to overcome this obstacle and widen the range of users who can benefit from these models, we have prepared a lineup of interactive demos[8], where users can easily experience each model, without having to worry about the technical details, using freely available GPU resources from Google Colab[9]. We also plan to actively add new models to this lineup, and hope to engage more users in our attempts to broaden the user base of deep learning technology.

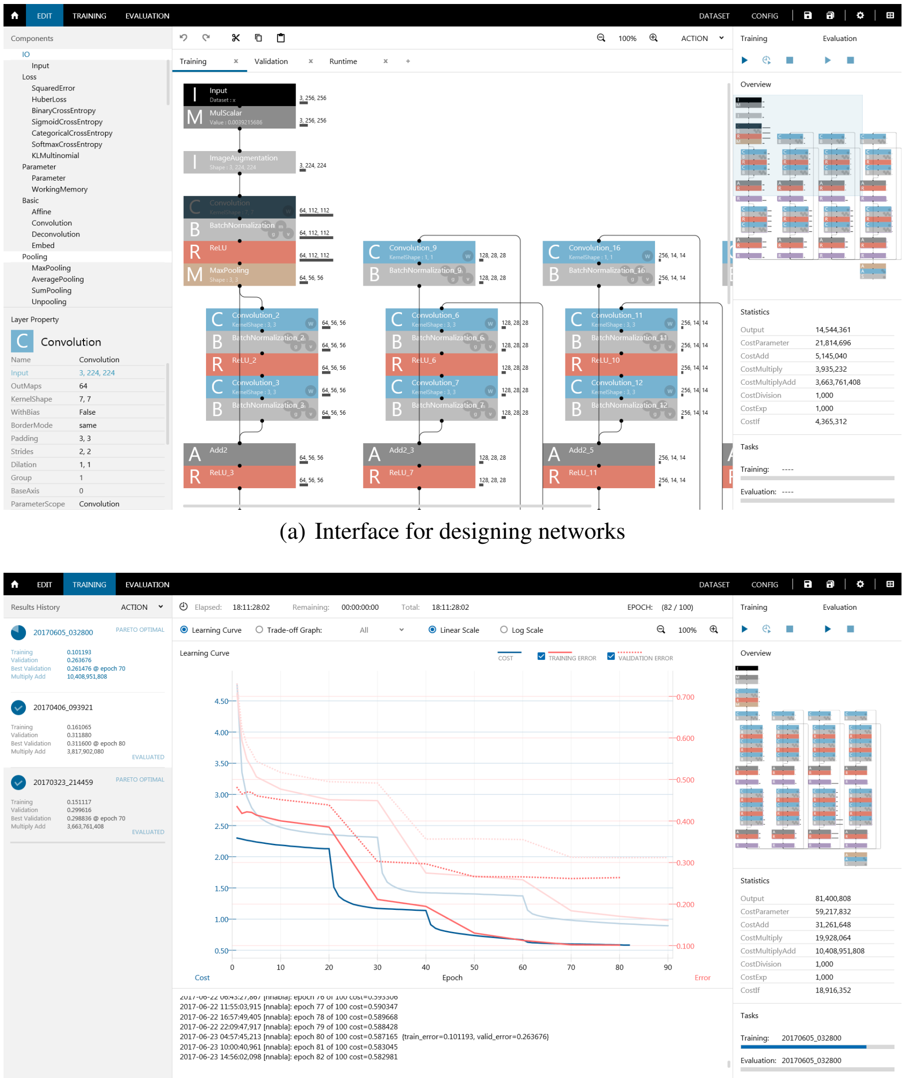
6 Conclusion

We have developed a deep learning framework, Neural Network Libraries, with engineer-oriented design principles, putting strong emphasis on usability and compatibility. The features of Neural Network Libraries reflect such design principles, and we demonstrate via experiments that they are

[7] https://github.com/sony/nnabla-examples/tree/master/GANs

[8] https://github.com/sony/nnabla-examples#interactive-demos

[9] https://colab.research.google.com/



Layer Property

Convolution

Name	Convolution
Input	3, 224, 224
OutMaps	64
KernelShape	7, 7
WithBias	False
BorderMode	same
Padding	3, 3
Strides	2, 2
Dilation	1, 1
Group	1
BaseAxis	0
ParameterScope	Convolution

Statistics

Output	14,544,361
CostParameter	21,814,696
CostAdd	5,145,040
CostMultiply	3,935,232
CostMultiplyAdd	3,663,761,408
CostDivision	1,000
CostExp	1,000
CostIf	4,365,312

Tasks

Training: ----

Evaluation: ----

(a) Interface for designing networks

Results History

20170605_032800 PARETO OPTIMAL

Training	0.101193
Validation	0.263676
Best Validation	0.261476 @ epoch 70
Multiply Add	10,408,951,808

20170406_093921 EVALUATED

Training	0.161065
Validation	0.311880
Best Validation	0.311600 @ epoch 80
Multiply Add	3,817,902,080

20170323_214459 PARETO OPTIMAL EVALUATED

Training	0.151117
Validation	0.299616
Best Validation	0.298836 @ epoch 70
Multiply Add	3,663,761,408

Elapsed: 18:11:28:02 Remaining: 00:00:00:00 Total: 18:11:28:02 EPOCH: (82 / 100)

● Learning Curve ○ Trade-off Graph: All ● Linear Scale ○ Log Scale 100%

```
2017-06-22 06:43:27,867 [nnabla]: epoch 76 of 100 cost=0.593306
2017-06-22 11:55:03,915 [nnabla]: epoch 77 of 100 cost=0.590347
2017-06-22 16:57:49,405 [nnabla]: epoch 78 of 100 cost=0.589668
2017-06-22 22:09:47,917 [nnabla]: epoch 79 of 100 cost=0.588428
2017-06-23 04:57:45,213 [nnabla]: epoch 80 of 100 cost=0.587165  {train_error=0.101193, valid_error=0.263676}
2017-06-23 10:00:40,961 [nnabla]: epoch 81 of 100 cost=0.583045
2017-06-23 14:56:02,098 [nnabla]: epoch 82 of 100 cost=0.582981
```

Statistics

Output	81,400,808
CostParameter	59,217,832
CostAdd	31,261,648
CostMultiply	19,928,064
CostMultiplyAdd	10,408,951,808
CostDivision	1,000
CostExp	1,000
CostIf	18,916,352

Tasks

Training: 20170605_032800

Evaluation: 20170605_032800

(b) Interface for training and evaluation

Figure 4: User interface of Neural Network Console.

efficient. Finally, we have also developed extensions and applications to spread the benefits of Neural Network Libraries to non-engineers as well.

Acknowledgments

We would like to express our genuine gratitude for the contributions made by the members of Sony R&D Center Stuttgart Laboratory, China Software Center of Sony China, Software Architecture Division (SARD) of Sony India Software Center, and many other voluntary contributors. We would also like to thank our users who have provided constructive feedback and insights.

References

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL `http://arxiv.org/abs/1409.0473`.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020. URL `https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf`.

Chung-Cheng Chiu, Tara N. Sainath, Yonghui Wu, Rohit Prabhavalkar, Patrick Nguyen, Zhifeng Chen, Anjuli Kannan, Ron J. Weiss, Kanishka Rao, Katya Gonina, Navdeep Jaitly, Bo Li, Jan Chorowski, and Michiel Bacchiani. State-of-the-art speech recognition with sequence-to-sequence models. *CoRR*, abs/1712.01769, 2017. URL `http://arxiv.org/abs/1712.01769`.

Yunjey Choi, Minje Choi, Munyoung Kim, Jung-Woo Ha, Sunghun Kim, and Jaegul Choo. Stargan: Unified generative adversarial networks for multi-domain image-to-image translation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

Mengyu Chu, You Xie, Laura Leal-Taixé, and Nils Thuerey. Temporally coherent gans for video super-resolution (tecogan). *CoRR*, abs/1811.09393, 2018. URL `http://arxiv.org/abs/1811.09393`.

J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, June 2019.

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in Neural Information Processing Systems 27*, 2014.

Satoshi Hara, Atsushi Nitanda, and Takanori Maehara. Data cleansing for models trained with sgd. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL `https://proceedings.neurips.cc/paper/2019/file/5f14615696649541a025d3d0f8e0447f-Paper.pdf`.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, Quoc V. Le, and Hartwig Adam. Searching for mobilenetv3, 2019.

Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7132–7141, 2018.

Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A Efros. Image-to-image translation with conditional adversarial networks. *CVPR*, 2017.

Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *ACM International Conference on Multimedia*, 2014.

Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. Progressive growing of GANs for improved quality, stability, and variation. In *International Conference on Learning Representations*, 2018a. URL `https://openreview.net/forum?id=Hk99zCeAb`.

Tero Karras, Samuli Laine, and Timo Aila. A style-based generator architecture for generative adversarial networks. *CoRR*, abs/1812.04948, 2018b. URL `http://arxiv.org/abs/1812.04948`.

Tero Karras, Samuli Laine, Miika Aittala, Janne Hellsten, Jaakko Lehtinen, and Timo Aila. Analyzing and improving the image quality of stylegan. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

Yann Lecun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, 1998.

Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training. In *ICLR*, 2018.

Sangwoo Mo, Minsu Cho, and Jinwoo Shin. Instance-aware image-to-image translation. In *International Conference on Learning Representations*, 2019. URL `https://openreview.net/forum?id=ryxwJhC9YX`.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pages 8026–8037, 2019.

Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. In *International Conference on Learning Representation (ICLR)*, 2016.

Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "why should I trust you?": Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, pages 1135–1144, 2016.

Frank Seide and Amit Agarwal. Cntk: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, 2016. ISBN 978-1-4503-4232-2.

Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *Proceedings of the IEEE international conference on computer vision*, pages 618–626, 2017.

J. E. Stone, D. Gohara, and G. Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science Engineering*, 12(3):66–73, 2010.

Mingxing Tan and Quoc Le. EfficientNet: Rethinking model scaling for convolutional neural networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 6105–6114. PMLR, 09–15 Jun 2019. URL `http://proceedings.mlr.press/v97/tan19a.html`.

Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. Chainer: a next-generation open source framework for deep learning. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in Annual Conference on Neural Information Processing Systems (NIPS)*, 2015.

Xintao Wang, Ke Yu, Shixiang Wu, Jinjin Gu, Yihao Liu, Chao Dong, Yu Qiao, and Chen Change Loy. Esrgan: Enhanced super-resolution generative adversarial networks. In *Proceedings of the European Conference on Computer Vision (ECCV) Workshops*, September 2018.

Wayne Wu, Yunxuan Zhang, Cheng Li, Chen Qian, and Chen Change Loy. Reenactgan: Learning to reenact faces via boundary transfer. In *Proceedings of the European Conference on Computer Vision (ECCV)*, September 2018.

Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1492–1500, 2017.

Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. In *Computer Vision (ICCV), 2017 IEEE International Conference on*, 2017.