VIZML: NEURAL NETWORK VIZ & GENERALIZATION

Shangru Yi [1]

ABSTRACT

We propose VizML, a system especially for neural network construction to tackling the heterogeneity in mainstream machine learning platforms, e.g. TensorFlow and PyTorch. This system is based on the abstraction of mainstream machine learning platforms. By adding abstraction layer for each specified platform API, different levels of network structures abstractions, including blocks and layers, are available for further model construction. By adding abstraction for macro network structures, such as data input and layer type, an extensible network generation system is built. Linking with a specific machine learning platform, we extract the relational graph for the constructed neural network and perform following code generation. Within the construction of relational graph, we can perform simple network structure logical debugging by checking the predefined network invariant. On top of the abstraction layer, the network visualization techniques with user interaction are deployed for easy network structure construction for principle network structures, such as CNN and DNN. We also introduce possible block-level memory profiling along with hardware assignment. With higher level visualization and formatted relational graph generation, further hardware attachment for network training is possible, which is expected to be utilized in cluster deployment or local tests.

1 INTRODUCTION

Identifying the shortage in terms of network construction and possible insufficiency in abstraction, we propose VizML, a system especially for neural network construction to tackling the heterogeneity in mainstream machine learning platforms, e.g. TensorFlow and PyTorch. This system is based on the abstraction of mainstream machine learning platforms. There are several falsifiable hypothesis to address the value of this proposed network construction system.

Achievable visualized programming. Due to the possibility of a structured representation for the neural network, the underlying data flow and component structures can be visualized. With user interfaces with interaction, visualized programming for neural network model is achievable.

Unified representation format. Without considering the actual heterogeneity in programming and network representation in mainstream machine learning platform, an unified representation for the meta structure and internal behavior can be built with high level programming abstraction.

Above-layer abstraction. Above-layer abstraction can speedup the network construction and enable possible profiling behavior.

High-level debugging. High level debugging along with possible hardware assignment can be solved in an easy way with visualization and programming abstraction.

In this paper, we first illustrate the designed primitives in our system and then present system details with architecture designs. In the last, we provide some case study to illustrate the possibility of our system.

2 PRIMITIVE ABSTRACTION

In this section, we first discuss the life cycle of machine learning task and the possible operation on the data carrier in present machine learning platform, e.g. tensor in PyTorch and TensorFlow. Based on the illustrated life cycle and the data operation, we then illustrate the design principles for the primitive abstraction in VizML.

2.1 ML Life Cycle

In general to say, machine learning is guided with dataflow between the defined operation. Once a predefined data forwarding structure (model) is defined, the left parts are the data source and data destination. Giving this high level abstraction illustration, we now present the mainstream workflow and dataflow of a designed machine learning model with focus on the neural network.

Model Definition & Running: Before the possible param-

[1]College of Computing, Georgia Institute of Technology, America. Correspondence to: Alexey Tumanov <atumanov3@gatech.edu>.

Proceedings of the 3^{rd} MLSys Conference, Austin, TX, USA, 2020. Copyright 2020 by the author(s).

eter learning or updates, the first step for machine learning task is to define the overall model structure with some learnable parameters. The overall model structure can be viewed as a dependency graph, with data dependency on previous node(s). Once the previous nodes finishing processing the data and generate output, the following nodes with dependencies fulfilled can start to perform certain operations on the data. The data from multiple precedent nodes will experience a transform and a certain output is produced by this node. The transform can be viewed as a function of the learnable parameters in this model. After several operations on the data, we can expect a score under a metric for guiding the following model update direction is produced. Since we say there are some learnable parameters in this model, the generated score must contain the information of those learnable parameters (with dependencies on those parameters). We can perform a trace back to find which direction that those learnable parameters are expected to be updated in order to generate a better score under the desired metrics. With an iterating manner, we can expect the score improving as the updates of those learnable parameters.

Prepossessing: Considering the preparation stage for the actual model updates, including data prepossessing and possible parameters loading (restoring model training with imported model weights), there is a prepossessing stage for machine learning. As for data prepossessing, we can treat it as a internal transform in the machine learning dataflow. Thus, we refer the prepossessing stage in our following abstraction as the loading of some pre-trained weights.

Post-processing: After the iterations of model and the updates of learnable parameters, the model running comes to the end. Since the trained parameters are expected to be saved or processed to generate the final results, there is a extra stage in the machine learning task. In this stage, some altered data is extracted from the internal components of model (parameters of network layers) and saved in stable storage. Also, before actually saving the data, there are possible operations to processing the data.

2.2 Primitive Design

After illustrating the life cycle of a neural network machine learning task, we propose our designed primitives to fulfilling the tasks on different stages, i.e. model definition, running and possessing. The designed primitives are wrapper for the actual function operations with a uniformed data-driven forwarding behaviors. We now illustrates the details of the primitive designs.

2.2.1 Tensor Abstraction

We follow the same naming of data carrier abstraction, i.e. **tensor**, in mainstream machine learning platform. Instead of directly using the one provided in platforms, considering the possible differences, we wrapper the tensor in a high-level holder with memory size calculated and naming governing. This is the fundamental support for following node and more higher block abstraction.

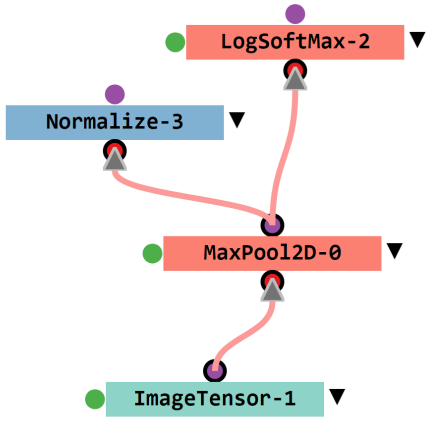

Figure 1. Branching: multiple outputs for **MaxPool2D**

2.2.2 Node Abstraction

Based on the layer level and operation abstraction, e.g. transform function, we implement a upper level abstraction to wrapper the inner function in the similar manner as PyTorch **nn.module**, referring as **node** in VizML. Instead of merely an abstract representation of underlying function, we implement the output data copying, naming government and memory usage for each node. The output data copy is based on the branching behavior of user constructed network structure, as **Figure 1**. Users intend to copy the output of that function and forward it to multiple following operations. Instead of having a extra copy operation to complicating the network structure, we handle the copy inside our abstraction units, i.e. **nodes**. Also, by hiding the detailed internal implementation of underlying machine learning platform but providing a unified interface for data forwarding, we extract the higher level model structure for better visualization. As for naming government, this function is for model saving and model loading, with unique name indicator for node state loading and saving. The state of node refers to the inner parameters of a node, e.g. weights and bias in linear layer. The memory usage part in nodes is for the above-node memory profiling (block-wise) and we will discuss this in later section. We further divide the node into sub diversion based on the underlying operations. Currently, we have four meta types for nodes.

Layer: A layer node is constructed with a layer forward as the internal function. The detailed manner of the layer forwarding is defined in API of machine learning platform.

As for our implementation, the layer node will be feed with data and the data will be taken as inputs to the underlying layer and generate a output. We extract the parameters in specific API and automatically generate our platform API for user inputs in the front visualization. It worthy to mention that the layer node can be stateful due to the possible updates of parameter and gradient in model running.

Transform: A transform node is constructed by a transform function, such as flattening a tensor and adding with other tensor. Comparing with layer nodes, transform nodes are stateless, without learnable parameters for gradient updates.

Constant: A constant node is a holder for tensor and the inner tensor can be loaded to model with some user defined parameters. For example, a constant node can carry a tensor with all zeros and the shape of the tensor is specified according to user inputs. A constant node can also carry some tensor stored in local system. With loading path specified, the local tensor can be loaded to constant node. The constant node is one of the initial node of a model, since it doesn't have data dependencies.

Input: A input node is a special case for constant node with batching support. During model iteration, each iteration can request different batch of input data for model training. Thus, we need to provide this batching iteration function, which is different from constant node. Also, there is possible an extra label output for input nodes. For example, we currently support MNIST dataset for model training. As for each batch, the images along with the labels will be feed to the network. For input node, it is the initial node for the overall model forwarding behavior and we enforce a constraint that each model can only have one input node, while the other possible data can be loaded with constant node.

2.2.3 *Model Governor*

The node is the general abstraction for the model running part, which governs the data forwarding manner and path of data flow. However, this abstraction is not enough for a complete machine learning model. We need two more abstraction for model running and model updates separately.

As for model running, we need an abstraction to specify the epoch number or termination conditions and the post-processing after the model termination, such as data saving. Meanwhile, since there is supports from machine learning platform for the hardware selections, i.e. GPU and CPU, we perform the hardware assignment in the governor node defined in model. The hardware assignment and epoch control is performed in **manager**. The after-training saving of tensor is performed in **saver**.

As for model updating, an abstraction as **optimizer** is provided in our system, which is a wrapper for optimizer in underlying machine learning platform. A loss generated by a node will be the one of the input for optimizer. The nodes have tensors expected to be updated are linked with the other input port of optimizer node. After each iteration of batch, the optimizer will update those tensors based on the loss (auto gradient in PyTorch).

A complete model is shown in **Figure 2**. We use **LBFGS** as optimizer with right port linked with the loss (comparing the output of layer with a saved local tensor) and left port linked with the tensor that we want to perform gradient updates. The green image tensor is the input node with one image as input (batch size also as 1). Since the input image is updated during the model running (we link it with optimizer), we want to save it after the model termination. Thus, we link the image node with the saver node for image saving.

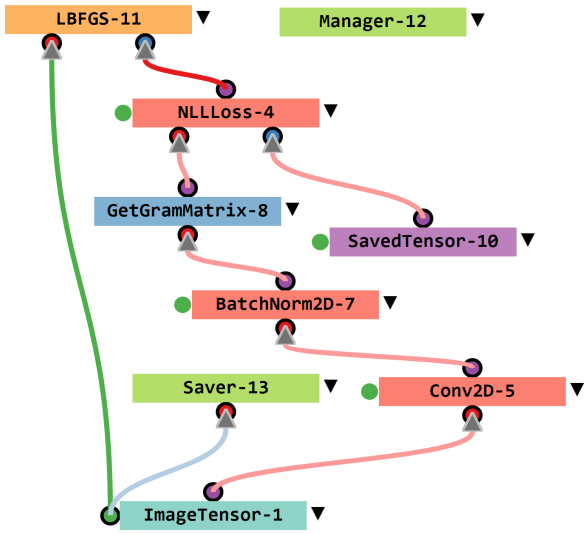

Figure 2. Model Illustration

3 Invariant Check

The invariant refers to the principles in neural network construction and parameter updates. In our system, we implement several basic invariant to demonstrate that the high-level debugging with visualization is achievable. In the model generation, some invariant is added because of the user interaction, such as preventing linking to the same input port twice from different nodes. Also, since we can also expect the gradient update for leaf tensor, i.e. without operation on it, we add this invariant checks in our system. Instead of finding the bug when actually running the model, we can exclude them at the model construction stage. Although these checks are fundamental, we think it is enough to illustrate our points.

4 Block Memory Profiling

One contribution of our system is the above-layer abstraction and corresponding profiling. In our system, we introduce an abstraction level of block, which is a combination of multiple nodes in our model. Since we have provided node the ability to get the memory usage, the memory profiling of block is the grouping of underlying nodes. Moreover, considering the possible inplace forward behavior (multiple nodes sharing memory for generating same outputs), we add a checker to only count unique tensor for more accurate memory profiling.

5 Relational Graph Generation

Since we enforce the constrain that there is only one input in each model, the relational graph generation is started from the input node. There is no data dependency for input node, while the input node provides data for following nodes. We can view the overall model as a directed acyclic graph (DAG). Once the dependency is fulfilled, we can step to next node. The generation is completed until there is no node can be added to the graph. The manager node and saver node will be excluded from the graph before generation. Also, the optimizer is not a part in the relational graph, since it have no influence on the data forwarding stage. The edges link to those nodes will be excluded in advance and we treat them specially.

6 System Overview

Our system is composed of two parts: user interface and backend api along with code generation. The overall frontend is shown as **Figure 3**. We have provided support for several operations and layers in PyTorch. We extract the parameters from the PyTorch API and generate corresponding wrapper for further cross platform extension. It is much easier for users to visualize the underlying network structure and the data forwarding behavior of the constructed models.

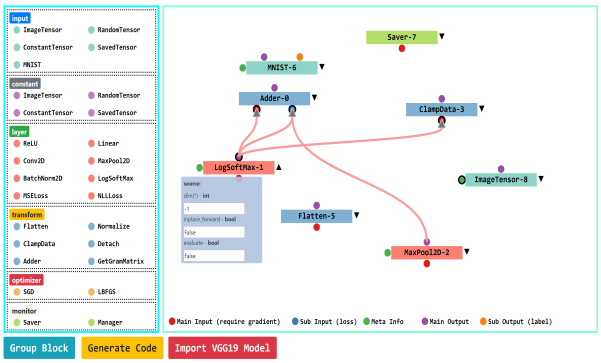

Figure 3. Front End Viz

6.1 System Architecture

The system architecture is similar as general web application. Once the backend starts, it will read the configuration of platform API (PyTorch) and generated API with our system abstraction. The frontend will read the generated API and demonstrate possible operations and nodes in our Viz. After user finishing constructing the network with several generation invariant checked, a **JSON** format network representation is generated and parsed in our backend. After the representation is parsed, the code is generated in unified programming style and naming conventions. User can address the generated code and running with the dependencies to underlying platform.

7 Implementation

Our implementation is in Python 3.7 with dependencies on Flask, a web framework. The frontend user interface is implemented JavaScript and D3. The PyTorch and its dependencies are the API provider of our system.

8 Case Study

In this section, we prepare two case studies to illustrate the functions of our system. In the first case, we illustrate a simple task to use this system to construct the handwritten figure identification task, which is classical example in terms of neural network application. For the identification task, we focus on the block-wise profiling, usability and code extendiability of our system. In the second case, we use this system to implement a more complex task, i.e. style transferring in computation vision. In this task, we import the pre-trained VGG19 network and use the weights to extract the representation of image. The network structure for this task involves more branches and transform abstraction for image processing. For the style transfer task, we focus on the compatibility of our system.

8.1 Handwritten Identification

We follow the basic tutorial and the parameters stated to generate the handwritten task model using VizML. The frontend structure is shown in **Figure 4**. There are images and labels in the MNIST dataset. We first flatten the image from two dimension to 1 one dimension and feed the flatten tensor to the network, starting from the linear layer. As for block, we add linear layer 7, linear layer 5 and linear layer 3 to form a block (show in the gold rectangle in the right side) in our system. For each epoch, the underlying block profiler will check the memory usage for the tensor and gradient within this block. Since we have provided the nodes functionality for extracting **TMU** (tensor memory usage), **TGU** (tensor gradient memory usage), **LMU** (layer weight memory usage) and **LGU** (layer gradient memory

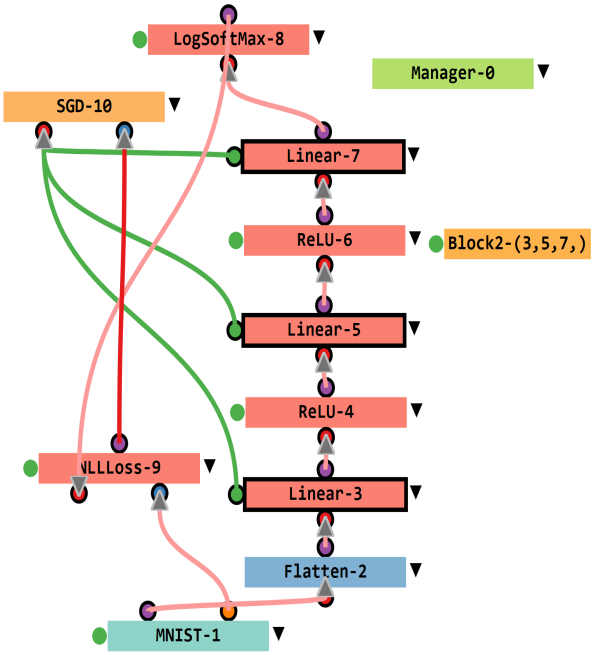

Figure 4. Handwritten Model

```
input_MNIST_0_4_transform_Flatten_0_1 = input_MNIST_0.get_output_tensor_single(0)
input_MNIST_0_5_layer_NLLLoss_0_2 = input_MNIST_0.get_output_label_single(0)
transform_Flatten_0.forward([input_MNIST_0_4_transform_Flatten_0_1])

transform_Flatten_0_4_layer_Linear_0_1 = transform_Flatten_0.get_output_tensor_single(0)
layer_Linear_0.forward([transform_Flatten_0_4_layer_Linear_0_1])

layer_Linear_0_4_layer_ReLU_0_1 = layer_Linear_0.get_output_tensor_single(0)
layer_ReLU_0.forward([layer_Linear_0_4_layer_ReLU_0_1])

layer_ReLU_0_4_layer_Linear_1_1 = layer_ReLU_0.get_output_tensor_single(0)
layer_Linear_1.forward([layer_ReLU_0_4_layer_Linear_1_1])

layer_Linear_1_4_layer_ReLU_1_1 = layer_Linear_1.get_output_tensor_single(0)
layer_ReLU_1.forward([layer_Linear_1_4_layer_ReLU_1_1])
```

Figure 5. Code Generation

usage), the above-nodes profiling for block is the combination abstraction for the underlying nodes. Considering the batch behavior in data inputs, we currently support the max, min profiling for above metrics, i.e. the max, min during the iterations of each epoch will be recorded. We train this model with 10 epochs and the memory profiling result is shown in **Figure 6**. Although there is little variance in the memory profiling, this possible block (node) level profiling can be viewed as preparation for following extension. We also shows partial generated code in **Figure 5**. It is clear to see the unified programming style and formatted naming conventions.

8.2 Style Transfer

We now use an example with more complex network structure to demonstrate the possibility of extension for our system. Given two images, saying one as style image and the other as content image, the goal of style transfer is to make

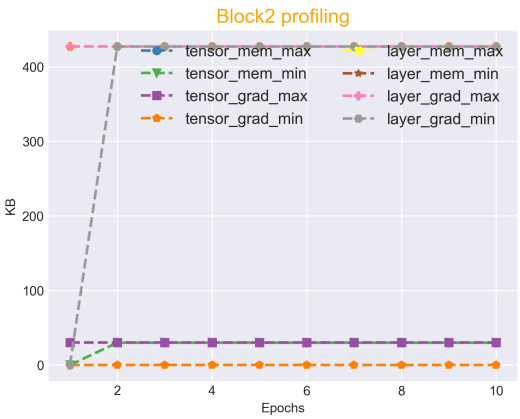

Figure 6. Block Profiling

the content image having similar style, i.e. color and textiles, with the style image. This can be achieved by extracting the representation feature using convolution neural layer. Since there are too much layers in VGG19, we just use several layers with connection links to perform the style transfer task. The partial network structure is shown in **Figure 7**. We generate this structure in advance with same weights and parameters as the pre-trained version of VGG19 provided by PyTorch. We then save the structure along with the parameter under system folder for further usage. We need first extract the feature tensor of the style image for further loss measurement. We can use the imported network structure and saver for this part. After the tensor is saved, we generate another model with the saved tensor loaded as constant node. In this model, we don't need to perform gradient updates on the layer, but merely on the input content image. With the content image as input, we perform optimization on the content image and save the generated image in local using saver node. We show the final network structure in **Figure 8** and the images in **Figure 9**. Although there are redundancy in the network structure, we can apply some functional programming in further optimization and make change in the code generation to adjust to this change. The current redundancy construction is for the ease of code generation.

9 Related Works

There are several visualization works for metrics monitoring, such as TensorBoard. In TensorBoard, they monitors the key metrics such as loss and how they change as training progresses, intending to providing more understanding for the underlying training procedures. For example, these metrics can help to monitor the training, e.g. checking overfitting if the training is too long. TensorBoard is composed of a graph dashboard and a scalar dashboard. The graph

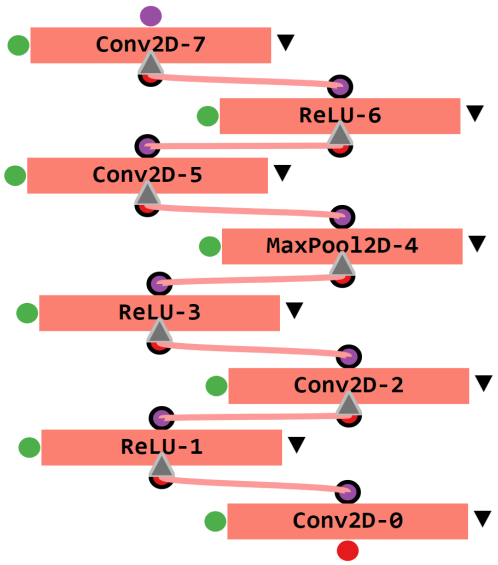

Figure 7. Partial VGG Network

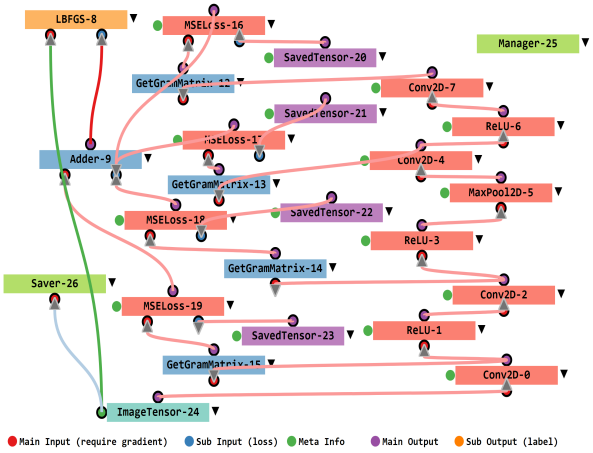

Figure 8. Style Transfer Network

Figure 9. Transfer Image

dashborad is a powerful tool for examining TensorFlow model, i.e. viewing a conceptual graph of model's structure and ensure it matches the intended design, while the scalars dashboard allows one to visualize these metrics using a simple API with little effort. However, there is little focus in TensorBoard for the model construction and less concern about the block abstraction for underlying structure grouping. Azure ML Studio puts focus on the machine learning model construction. With simple user interaction, user can construct a complete model, from input to output. However, there is minimal effort for Azure to support advanced model construction, such as neural network, and it is a platform separated from mainstream machine learning platform, such as TensorFlow and PyTorch, which limits its extendiability.

10 Conclusion

In this paper, we introduce a system, VizML, with supports for cross platform neural network construction and platform-specified code generation. This system is based on the abstraction of mainstream machine learning platforms. By adding abstraction wrapper for each specified platform API, different levels of network structures abstractions, including blocks and layers, are available for further model construction. The current system is still in the initial stage, with fundamental function and supports for PyTorch implemented. As for extension, we plan to add abstraction for more transform in PyTorch and even support user-defined transform (by exposing our API to users). Also, the current hardware assignment is fundamental, supporting only one GPU or CPU training for models. We plan to extend this to support cross-GPU or cluster level model training.

10.1 VizML

The source code of VizML is release in https://github.com/yishangru/VizML. We can provide supports for normal model generation and implement very minimal validation check for the user inputs due to time constrains, which is far from enough. Some strange generation behaviors could make the system out of works. We will keep update this repository with more usability provided and add more invariant for ensuring the model logic correctness. More extensive node operations will be available in following months.

References

Tensorboard. https://www.tensorflow.org/tensorboard/graphs. [Accessed 25-March-2020].

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pp. 265–283, 2016.

Barga, R., Fontama, V., and Tok, W. H. Introducing microsoft azure machine learning. In *Predictive Analytics with Microsoft Azure Machine Learning*, pp. 21–43. Springer, 2015.

Gatys, L. A., Ecker, A. S., and Bethge, M. Image style transfer using convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2414–2423, 2016.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pp. 8024–8035, 2019.