10-414/714 – Deep Learning Systems: Algorithms and Implementation

Neural Network Library Abstractions

Fall 2025

Tianqi Chen (this time) and Tim Dettmers
Carnegie Mellon University

Outline

Programming abstractions

High level modular library components

Outline

Programming abstractions

High level modular library components

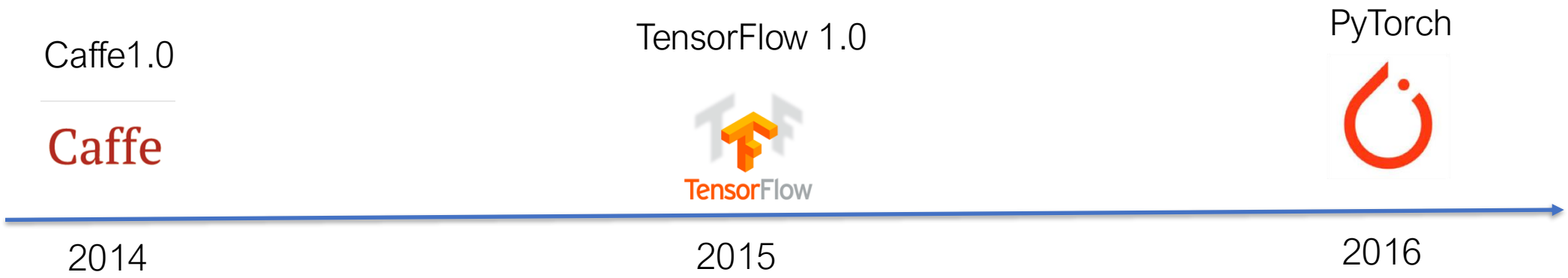
Programming abstractions

The programming abstraction of a framework defines the common ways to implement, extend and execute model computations.

While the design choices may seem obvious after seeing them, it is useful to learn about the thought process, so that:

- We know why the abstractions are designed in this way
- Learn lessons to design new abstractions.

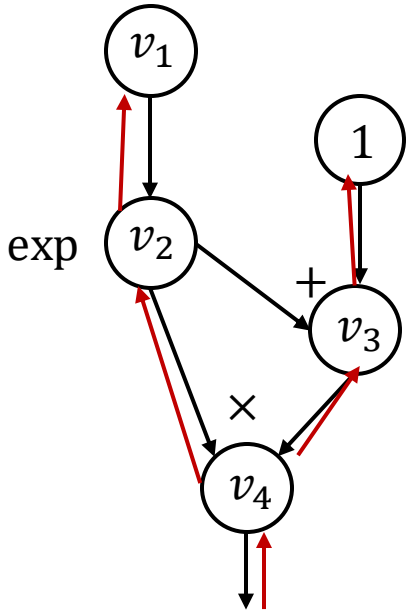
Case studies



There are many frameworks being development along the way that we do not have time to study: theano, torch7, mxnet, caffe2, chainer, jax …

Forward and backward layer interface

Example framework: Caffe 1.0




```
class Layer:
    def forward(bottom, top):
        pass

    def backward(top,
                 propagate_down,
                 bottom):
        pass
```

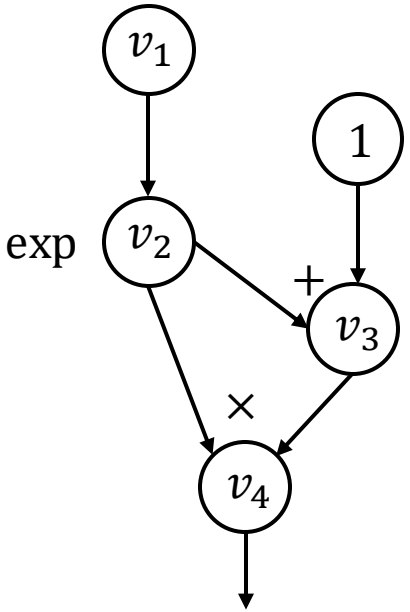
Defines the forward computation and backward(gradient) operations

Used in cuda-convenet (the AlexNet framework)

Early pioneer: cuda-convnet

Computational graph and declarative programming

Example framework: Tensorflow 1.0




```
import tensorflow as tf

v1 = tf.Variable()
v2 = tf.exp(v1)
v3 = v2 + 1
v4 = v2 * v3

sess = tf.Session()
value4 = sess.run(v4, feed_dict={v1: numpy.array([1]})
```

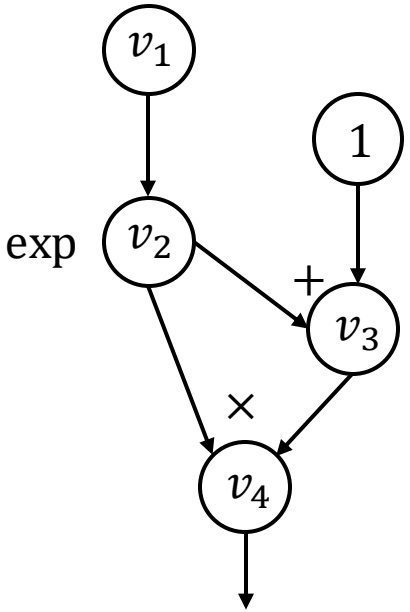
First declare the computational graph

Then execute the graph by feeding input value

Early pioneer: Theano

Imperative automatic differentiation

Example framework: PyTorch (needle:)




```
import needle as ndl

v1 = ndl.Tensor([1])
v2 = ndl.exp(v1)
v3 = v2 + 1
v4 = v2 * v3
```

Executes computation as we construct the computational graph
Allow easy mixing of python control flow and construction

```
if v4.numpy() > 0.5:
    v5 = v4 * 2
else:
    v5 = v4
v5.backward()
```

Early pioneer: Chainer

Discussions

What are the pros and cons of each programming abstraction?

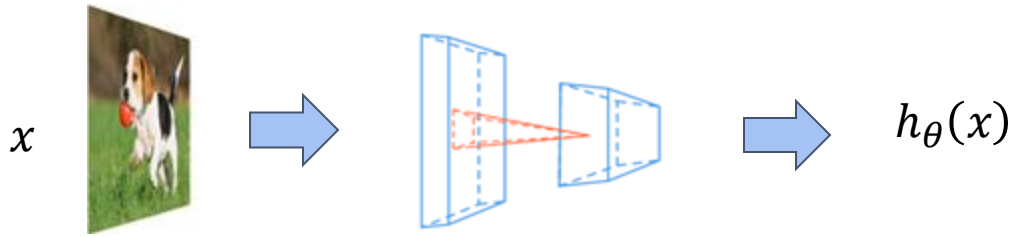
Outline

Programming abstractions

High level modular library components

Elements of Machine Learning

1. The hypothesis class:



2. The loss function:

$$l(h_\theta(x), y) = -h_y(x) + \log \sum_{j=1}^{k} \exp\left(h_j(x)\right)$$

3. An optimization method:

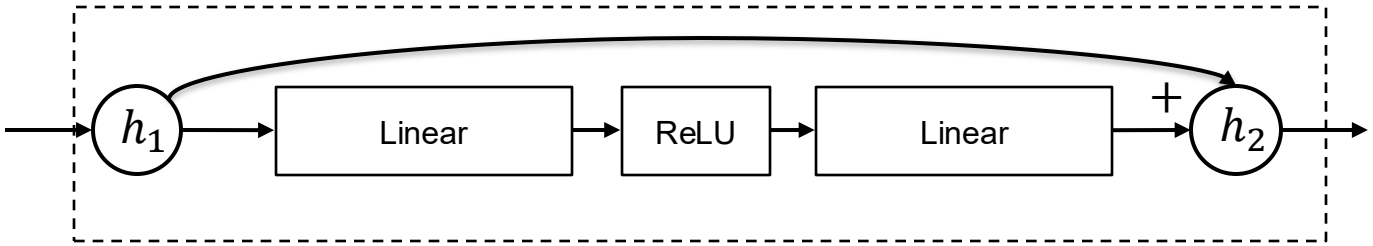
$$\theta := \theta - \frac{\alpha}{B} \sum_{i=1}^{B} \nabla_\theta \ell\left(h_\theta\left(x^{(i)}\right), y^{(i)}\right)$$

Question: how do they translate to modular components in code?

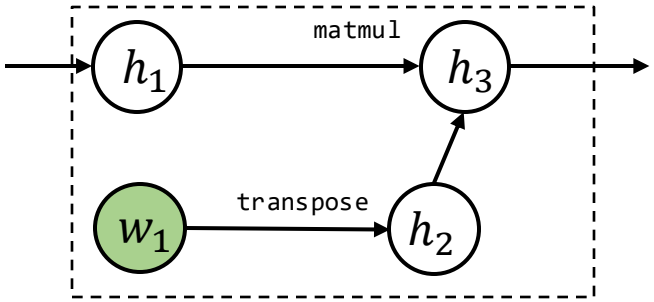
Deep learning is modular in nature

Multi-layer Residual Net

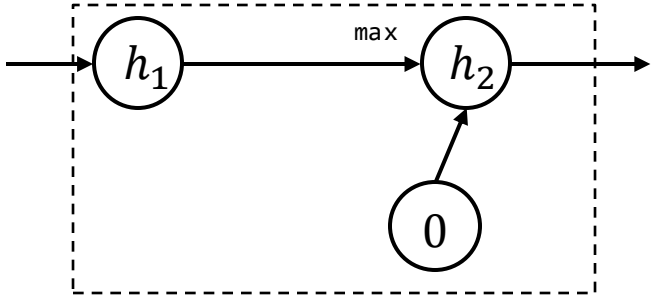
Residual block



Linear



ReLU



Residual Connections

Deep Residual Learning for Image Recognition

Kaiming He Xiangyu Zhang Shaoqing Ren Jian Sun
Microsoft Research
{kahe, v-xiangz, v-shren, jiansun}@microsoft.com

Abstract

Deeper neural networks are more difficult to train. We present a residual learning framework to ease the training of networks that are substantially deeper than those used previously. We explicitly reformulate the layers as learning residual functions with reference to the layer inputs, instead of learning unreferenced functions. We provide comprehensive empirical evidence showing that these residual networks are easier to optimize, and can gain accuracy from considerably increased depth. On the ImageNet dataset we evaluate residual nets with a depth of up to 152 layers—8× deeper than VGG nets [41] but still having lower complexity. An ensemble of these residual nets achieves 3.57% error on the ImageNet test *set. This result won the 1st place on the ILSVRC 2015 classification task. We also present analysis on CIFAR-10 with 100 and 1000 layers.*

The depth of representations is of central importance for many visual recognition tasks. Solely due to our extremely deep representations, we obtain a 28% relative improvement on the COCO object detection dataset. Deep residual nets are foundations of our submissions to ILSVRC & COCO 2015 competitions[1]*, where we also won the 1st places on the tasks of ImageNet detection, ImageNet localization, COCO detection, and COCO segmentation.*

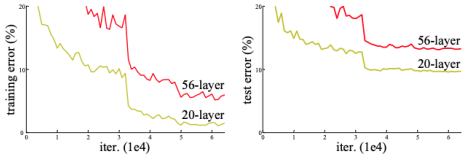


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

greatly benefited from very deep models.

Driven by the significance of depth, a question arises: *Is learning better networks as easy as stacking more layers?* An obstacle to answering this question was the notorious problem of vanishing/exploding gradients [1, 9], which hamper convergence from the beginning. This problem, however, has been largely addressed by normalized initialization [23, 9, 37, 13] and intermediate normalization layers [16], which enable networks with tens of layers to start converging for stochastic gradient descent (SGD) with backpropagation [22].

When deeper networks are able to start converging, a *degradation* problem has been exposed: with the network

Identity Mappings in Deep Residual Networks

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun

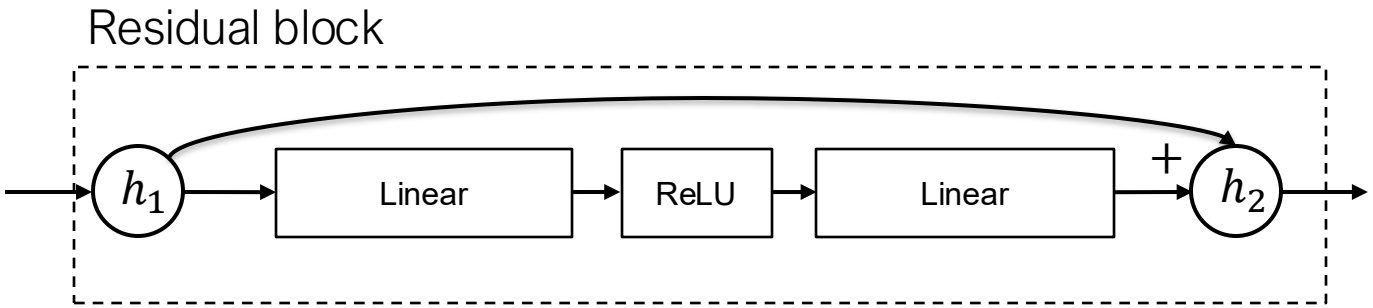
Microsoft Research

Abstract Deep residual networks [1] have emerged as a family of extremely deep architectures showing compelling accuracy and nice convergence behaviors. In this paper, we analyze the propagation formulations behind the residual building blocks, which suggest that the forward and backward signals can be directly propagated from one block to any other block, when using identity mappings as the skip connections and after-addition activation. A series of ablation experiments support the importance of these identity mappings. This motivates us to propose a new residual unit, which makes training easier and improves generalization. We report improved results using a 1001-layer ResNet on CIFAR-10 (4.62% error) and CIFAR-100, and a 200-layer ResNet on ImageNet. Code is available at: https://github.com/KaimingHe/resnet-1k-layers.

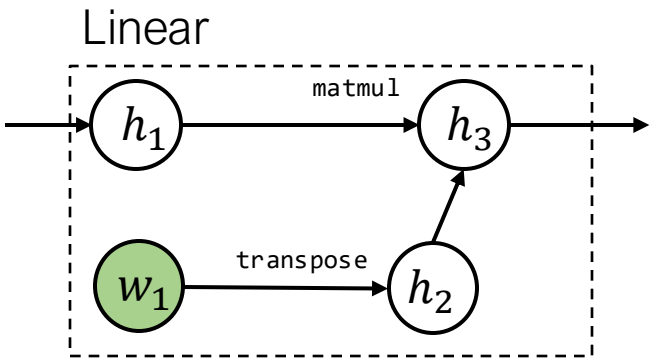
ResNetV2

One of the most well-cited paper

nn.Module: Compose Things Together



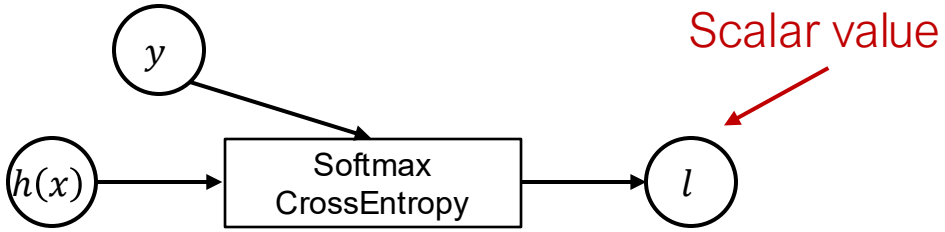
Tensor in, tensor out



Key things to consider:

- For given inputs, how to compute outputs
- Get the list of (trainable) parameters
- Ways to initialize the parameters

Loss functions as a special kind of module



Tensor in, scalar out

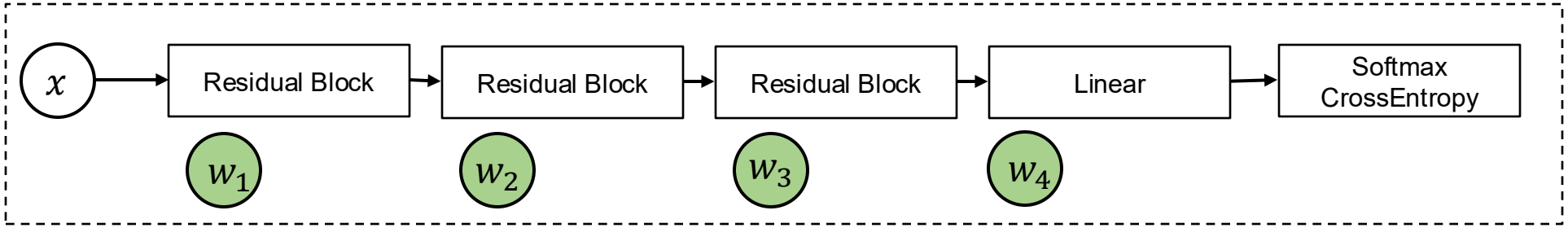
$$l(h_\theta(x), y) = -h_y(x) + \log \sum_{j=1}^{k} \exp\left(h_j(x)\right)$$

Questions

- How to compose multiple objective functions together?
- What happens during inference time after training?

Optimizer

Model



- Takes a list of weights from the model perform steps of optimization
- Keep tracks of auxiliary states (momentum)

SGD

$$w_i \leftarrow w_i - \alpha g_i$$

SGD with momentum

$$u_i \leftarrow \beta u_i + (1-\beta) g_i$$
$$w_i \leftarrow w_i - \alpha u_i$$

Adam

$$u_i \leftarrow \beta_1 u_i + (1-\beta_1) g_i$$
$$v_i \leftarrow \beta_2 v_i + (1-\beta_2) g_i^2$$
$$w_i \leftarrow w_i - \alpha u_i / (v_i^{1/2} + \epsilon)$$

Regularization and optimizer

Two ways to incorporate regularization:

- Implement as part of loss function
- Directly incorporate as part of optimizer update

SGD with weight decay (l_2 regularization) $w_i \leftarrow (1 - \alpha\lambda)\, w_i - \alpha g_i$

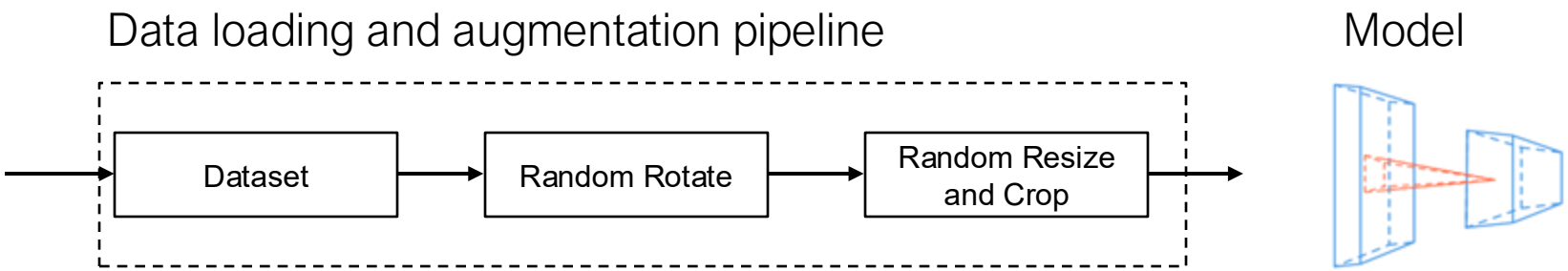
Initialization

Initialization strategy depends on the module being involved and the type of the parameter. Most neural network libraries have a set of common initialization routines

- weights: uniform, order of magnitude depends on input/output
- bias: zero
- Running sum of variance: one

Initialization can be folded into the construction phase of a nn.module.

Data loader and preprocessing

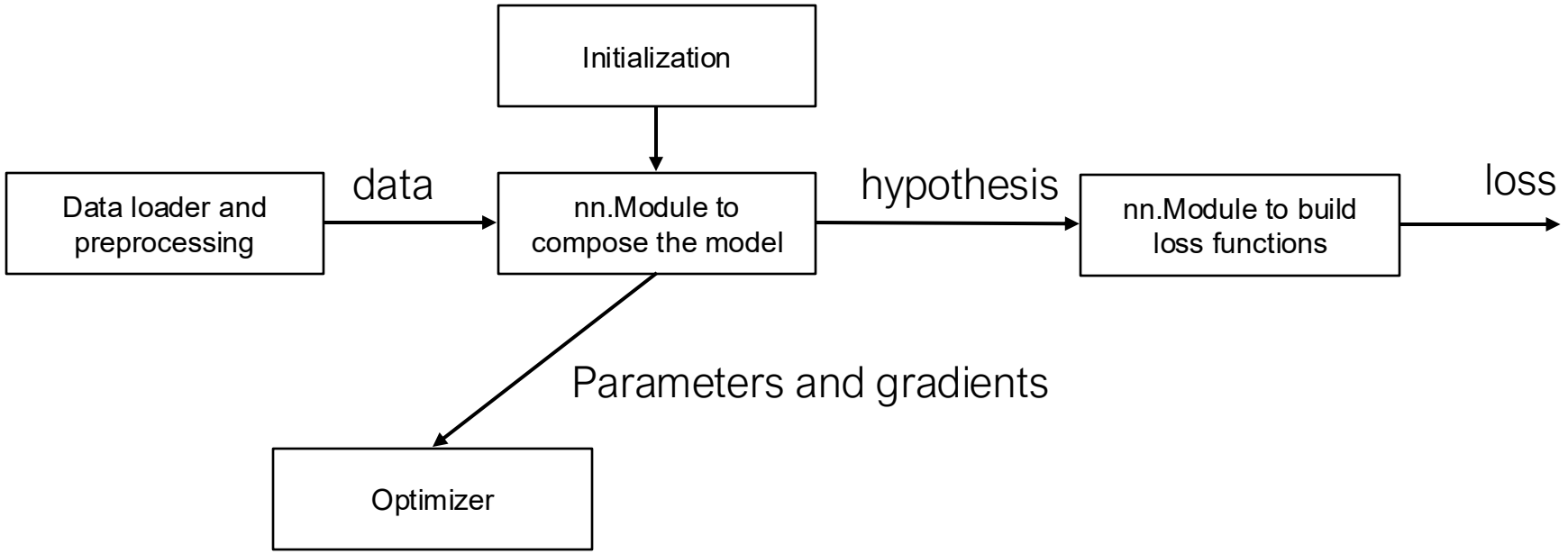


We often preprocess (augment) the dataset by randomly shuffle and transform the input

Data augmentation can account for significant portion of prediction accuracy boost in deep learning models

Data loading and augmentation is also compositional in nature

Deep learning is modular in nature

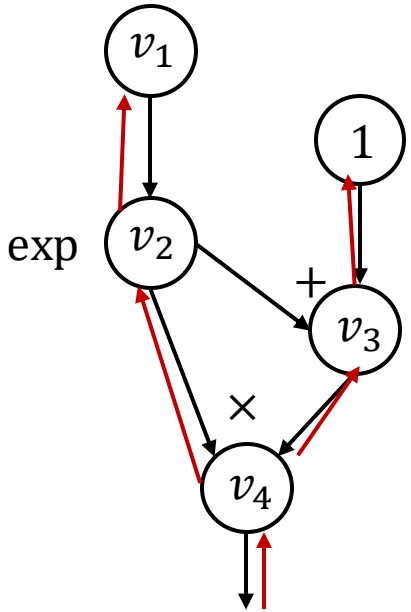


Discussions

What are other possible examples of modular components?

Revisit programming abstraction

Example framework: Caffe 1.0



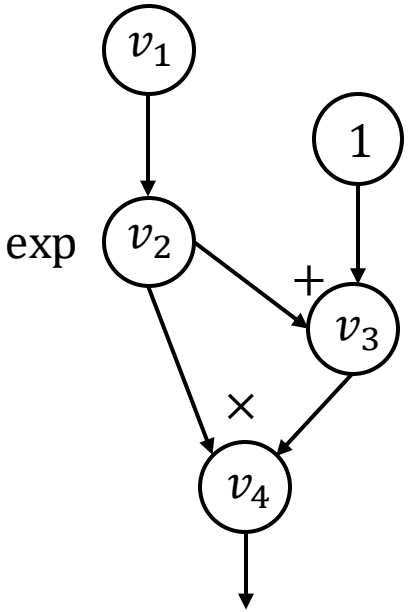

```
class Layer:
    def forward(bottom, top):
        pass

    def backward(top,
                 propagate_down,
                 bottom):
        pass
```

Couples gradient computation with the module composition.

Revisit programming abstraction

Example framework: PyTorch (needle:)




```
import needle as ndl

v1 = ndl.Tensor([0])
v2 = ndl.exp(v1)
v3 = v2 + 1
v4 = v2 * v3
```

Two levels of abstractions

- Computational graph abstraction on Tensors, handles AD
- High level abstraction to handle modular composition

Outline

Programming abstractions

High level modular library components