Data 188: Introduction to Deep Learning

Neural Network Library Abstractions

Speaker: Eric Kim
Lecture 07 (Week 04)
2026-02-10, Spring 2026. UC Berkeley.

Acknowledgement: slides are based on CMU's Deep Learning Systems course (10-414/714, Fall 2025). Thanks to: Prof. Zico Kolter, Prof. Tianqi Chen, Prof. Tim Dettmers.

Announcements

- HW1 continues
  - Warning: this homework is substantially more work than HW0. Start early!
- Weekly course surveys
  - "Course Survey (Week 04) (optional, extra credit)"
  - "Course Survey (HW0) (optional)"
- **Reminder: Add/drop deadline is Wed Feb 11th, 11:59 PM PST!**

Outline

Programming abstractions

High level modular library components

Outline

Programming abstractions

High level modular library components

Programming abstractions

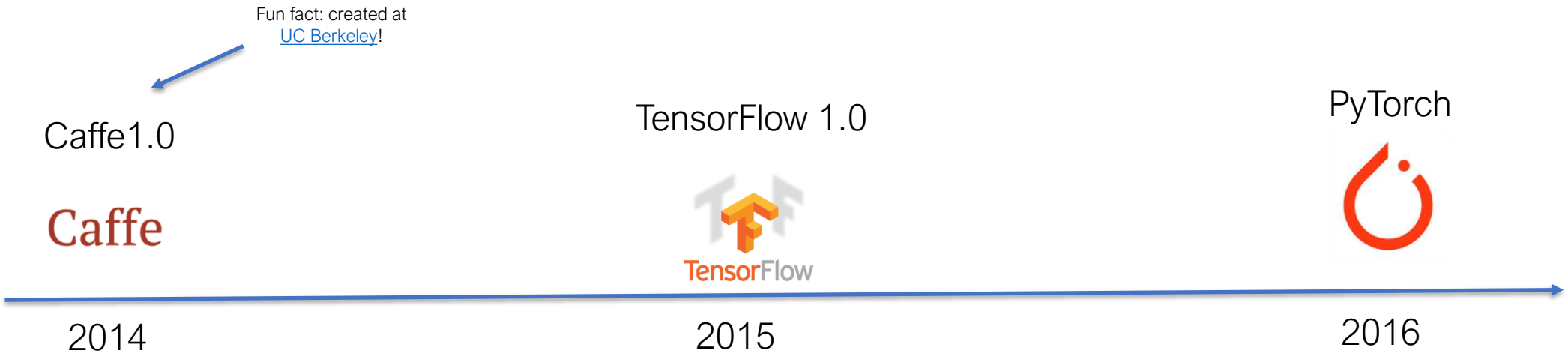
The programming abstraction of a framework defines the common ways to implement, extend and execute model computations.

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.

The importance of abstractions: should remind you of Data C88C!

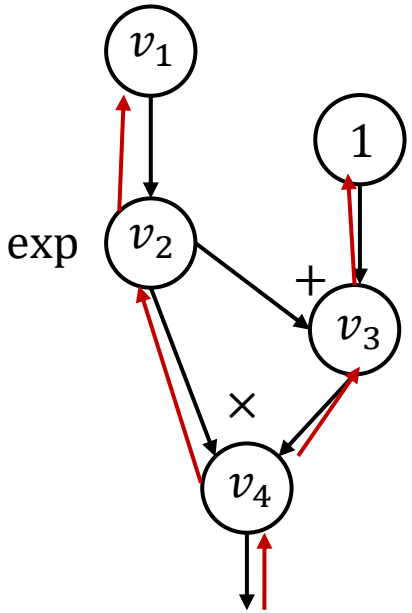
Case studies



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
```

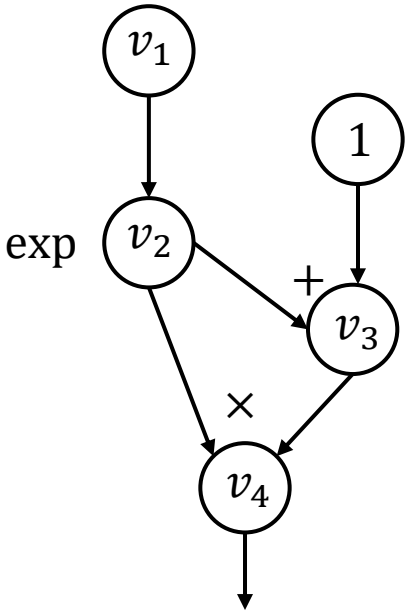
Defines the forward computation and backward(gradient) operations

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]})
```

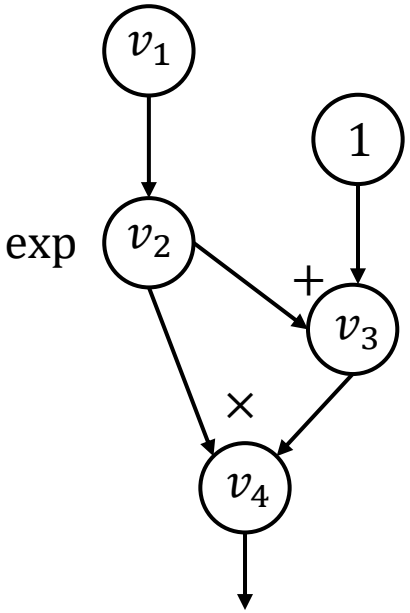
First declare the computational graph

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()
```

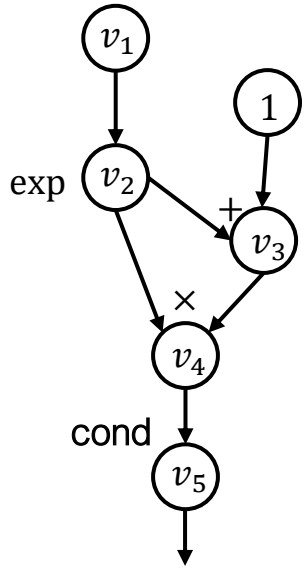
Early pioneer: Chainer

Static computation graphs

In Tensorflow 1.0, you must fully construct the computation graph up-front, aka "**static**" computation graph. Can't modify the graph during training/inference.

Adding control flow (eg "if" statements) is do-able by adding "conditional" nodes (tf.cond)...but the field has largely voted that this technique isn't enjoyable to work with.

```
import tensorflow as tf

v1 = tf.Variable()
v2 = tf.exp(v1)
v3 = v2 + 1
v4 = v2 * v3
v5 = tf.cond(
    v4 > 0.5,
    true_fn=lambda: v4 * 2,
    false_fn=lambda: v4
)

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



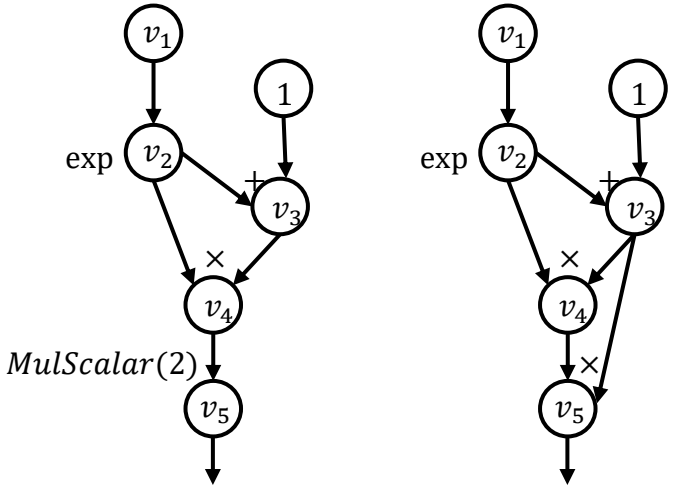
Dynamic computation graph

In PyTorch (and needle!), the computation graph is dynamically constructed on the fly.

During training, this graph is created for training iteration, meaning it's easy for the model architecture to dynamically change for each iteration.

```
import needle as ndl

v1 = ndl.Tensor([1])
v2 = ndl.exp(v1)
v3 = v2 + 1
v4 = v2 * v3
if v4.numpy() > 0.5:
    v5 = v4 * v3
else:
    v5 = 2 * v4
v5.backward()
```

Suppose:
(iter0) v4 = 0.1
(iter1) v4 = 5



Static vs dynamic graphs

Generally speaking, ML researchers/practitioners prefer working with dynamic computation graphs, due to dev velocity and ease of use.

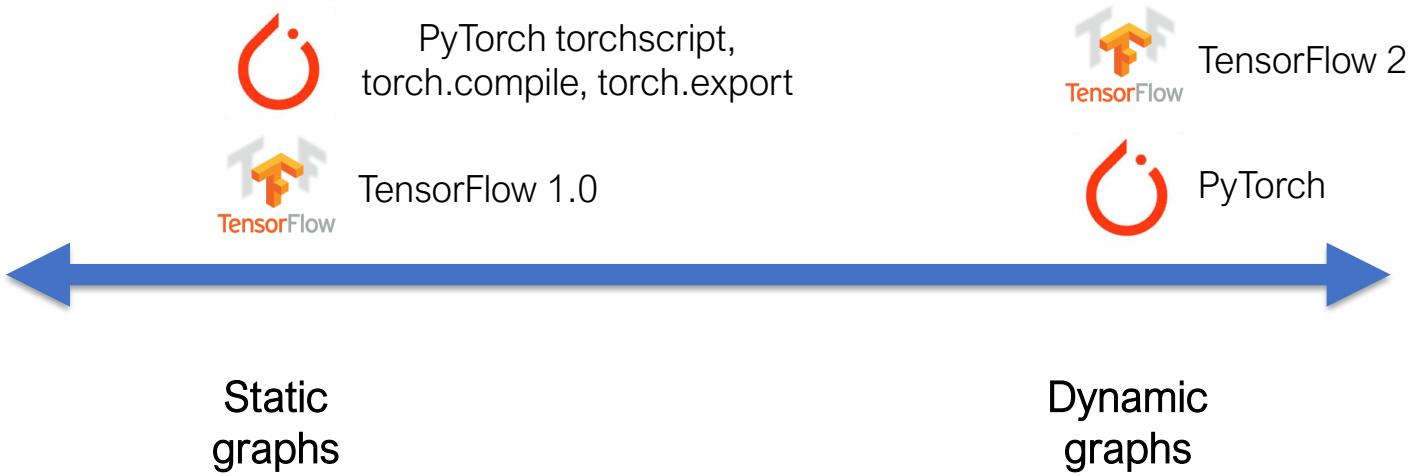
However: static computation graphs can typically be more performant (lower latency, lower memory usage, higher throughput, etc). Also, opportunities for computation graph optimizations (ex: operator fusion, opt techniques from compilers, etc).

(aside) Similar to static vs dynamic debate in programming languages, eg Java/C++ vs Python/Javascript...

Learning from your competition

TensorFlow 2.0: (Sept 2019) added dynamic computation graphs and "eager" mode execution. Improve dev velocity, quality of life.

PyTorch: (v1.0, Dec 2018) added static graph compilation techniques (torchscript, torch.compile and torch.export) to improve training/inference system performance.



(Aside) **Tradeoff**: torchscript-ing models can be challenging, often requires nontrivial changes to your model code to be "torchscript compatible". Replaced by torch.export in pytorch 2.9 (Oct 2025)

torch.compile and torch.export are newer techniques. Easier to use, but there are still some limitations.

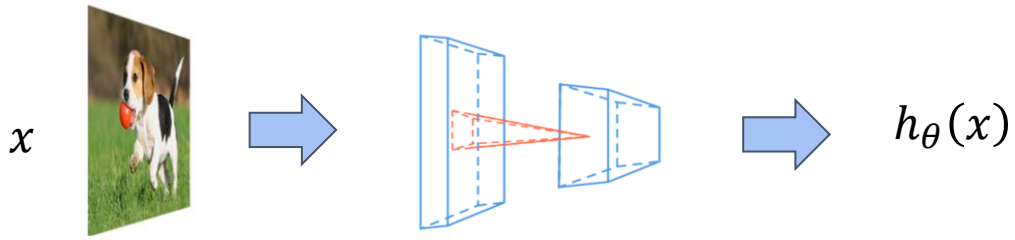
Outline

Programming abstractions

High level modular library components

Elements of Machine Learning

1. The hypothesis class:



2. The loss function:

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

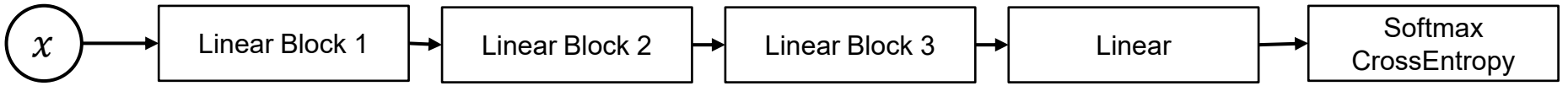
3. An optimization method:

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

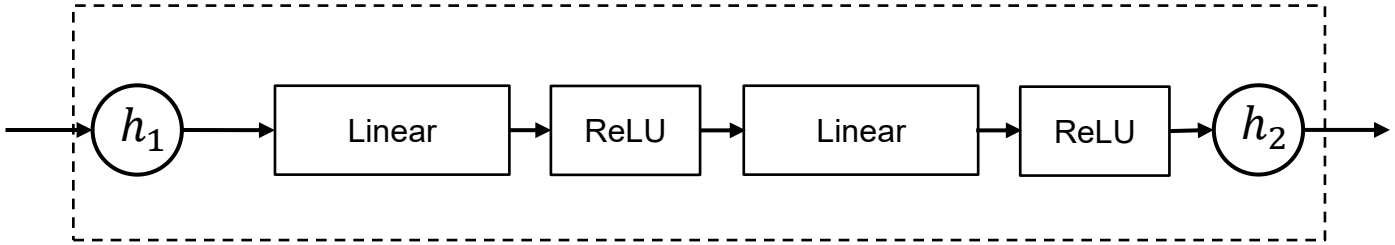
Question: how do they translate to modular components in code?

Deep learning is modular in nature

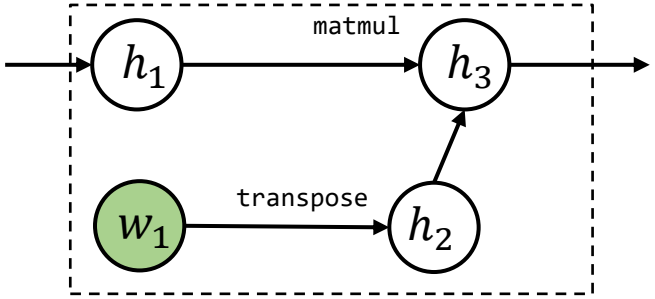
Multi-layer Perceptron (MLP)



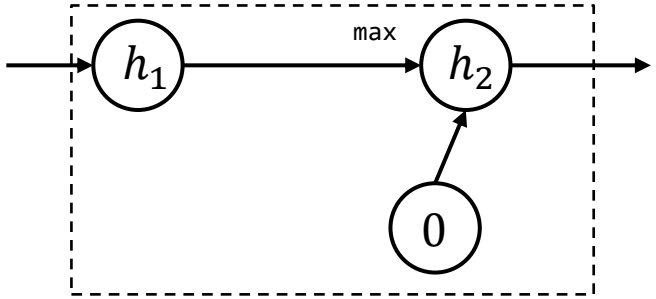
Linear block



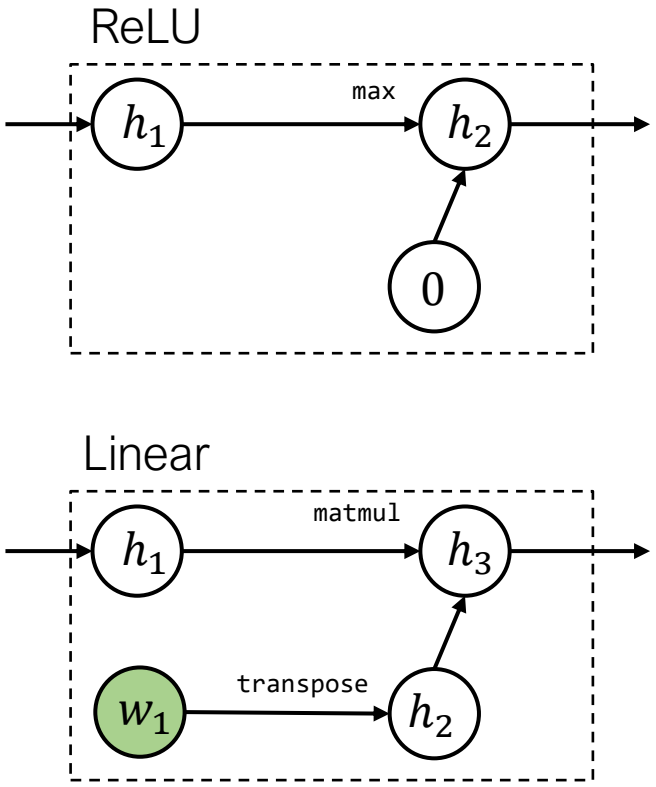
Linear



ReLU



nn.Module: Functional interface (needle)



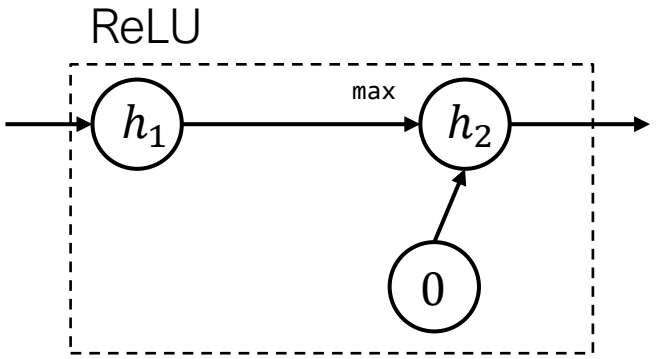

```
class Module:
    def __init__(self):
        # If I have model parameters, initialize
        # then (ex: random, kaiming_norm, etc)
        pass

    def forward(self, *args, **kwargs):
        # Perform forward pass
        raise NotImplementedError()

    def parameters(self) -> list[Tensor]:
        # Return list of parameters in the module
        raise NotImplementedError()
```

Key things to consider:

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

Fun fact: pytorch also has a `torch.nn.Module` which is similar in spirit.

nn.Module: Relu layer as subclass



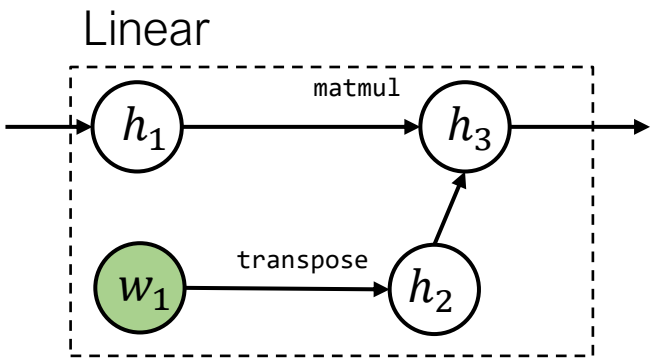

```
class Module:
    def __init__(self):
        # If I have model parameters, initialize
        # then (ex: random, kaiming_norm, etc)
        pass

    def forward(self):
        # Perform forward pass
        raise NotImplementedError()

    def parameters(self) -> list[Tensor]:
        # Return list of parameters in the module
        raise NotImplementedError()
```

```
class Relu(nn.Module):
    def __init__(self):
        # relu has no params
        super().__init__()

    def forward(self, x: ndl.Tensor):
        # Perform forward pass
        return ops.relu(x)

    def parameters(self) -> list[Tensor]:
        # relu has no params
        return []
```

Important: `Relu.forward()` creates a new Tensor node in the computation graph. Means backprop will work!

Fun fact: pytorch also has a `torch.nn.Module` which is similar in spirit.

nn.Module: Linear layer as subclass




```
class Module:
    def __init__(self):
        # If I have model parameters, initialize
        # then (ex: random, kaiming_norm, etc)
        pass

    def forward(self):
        # Perform forward pass
        raise NotImplementedError()

    def parameters(self) -> list[Tensor]:
        # Return list of parameters in the module
        raise NotImplementedError()
```

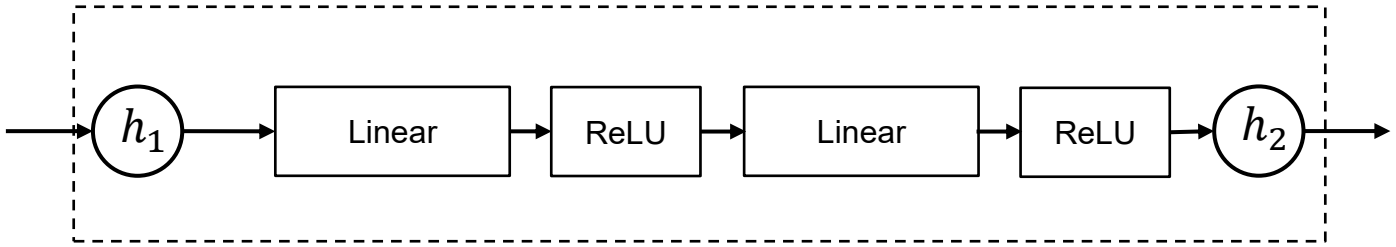
(skip bias vector for now)

```
class Linear(Module):
    def __init__(self, in_features: int, out_features: int):
        super().__init__()
        # create parameters and initialize them
        self.weight = createParameters(
            init_kaiming_normal(
                shape=[in_features, out_features]))

    def forward(self, X: ndl.Tensor) -> ndl.Tensor:
        # X.shape=[batchsize, in_features]
        return ops.matmul(X, self.weight)

    def parameters(self) -> list[ndl.Tensor]:
        # Return list of parameters in the module
        return [self.weight]
```

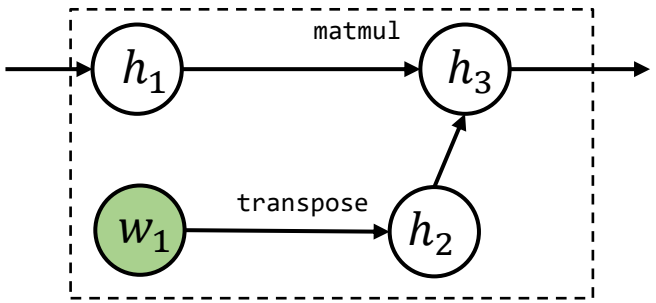
Fun fact: pytorch also has a `torch.nn.Module` which is similar in spirit.

nn.Sequential: composing modules

Linear block



Linear



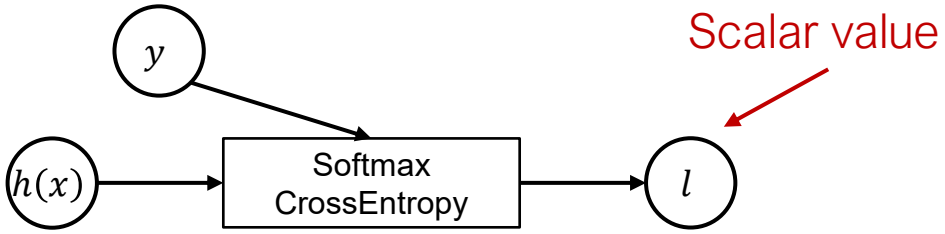

```
class Module:
    def __init__(self):
        # If I have model parameters, initialize
        # then (ex: random, kaiming_norm, etc)
        pass

    def forward(self):
        # Perform forward pass
        raise NotImplementedError()

    def parameters(self) -> list[Tensor]:
        # Return list of parameters in the module
        raise NotImplementedError()
```

```
import needle.nn as nn
linear_block = nn.Sequential(
    nn.Linear(28*28, 128),
    nn.Relu(),
    nn.Linear(128, 64),
    nn.Relu()
)
h2 = linear_block(h1)
```

Convenience Module for "stitching" modules together.

Fun fact: pytorch also has a `torch.nn.Sequential` which is similar in spirit.

Loss functions as a special kind of module



Tensor in, scalar out

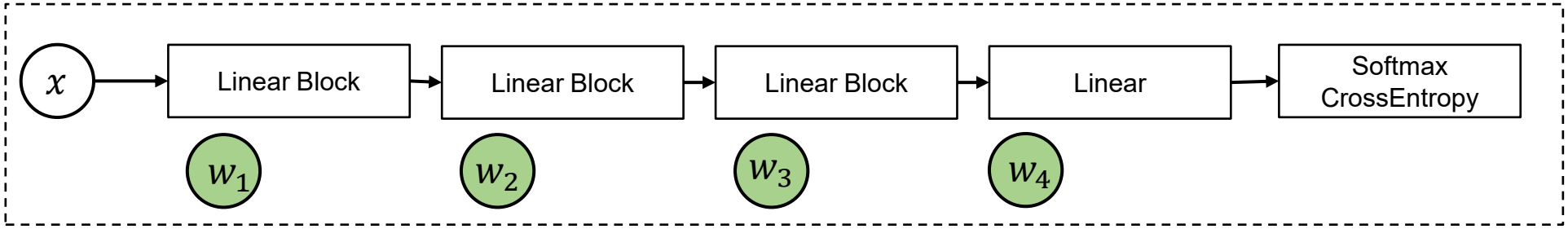
$$l(h_\theta(x), y) = -h_y(x) + \log \sum_{j=1}^{k} \exp\left(h_j(x)\right)$$

Questions

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

Optimizer

Model



- Given list of parameters from the model, optimizes the params
- Keep tracks of auxiliary states (ex: momentum, etc)

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

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

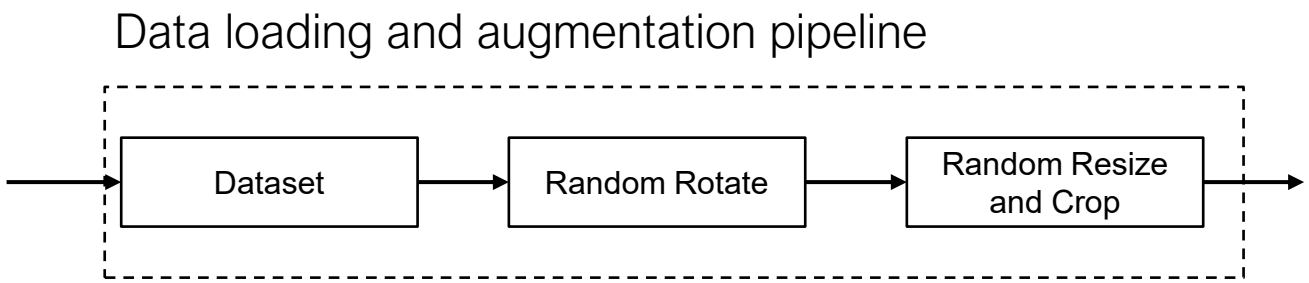
Parameter Initialization

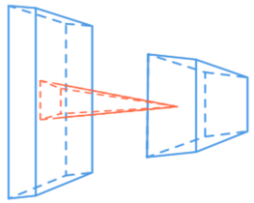
Initialization strategy depends on the module being involved and the type of the parameter. Most neural network libraries have a set of common initialization routines. For example, for Linear layers:

- weights: uniform, order of magnitude depends on input/output
- bias: zero

Initialization is typically placed in the constructor of a nn.Module.

Data loader and preprocessing

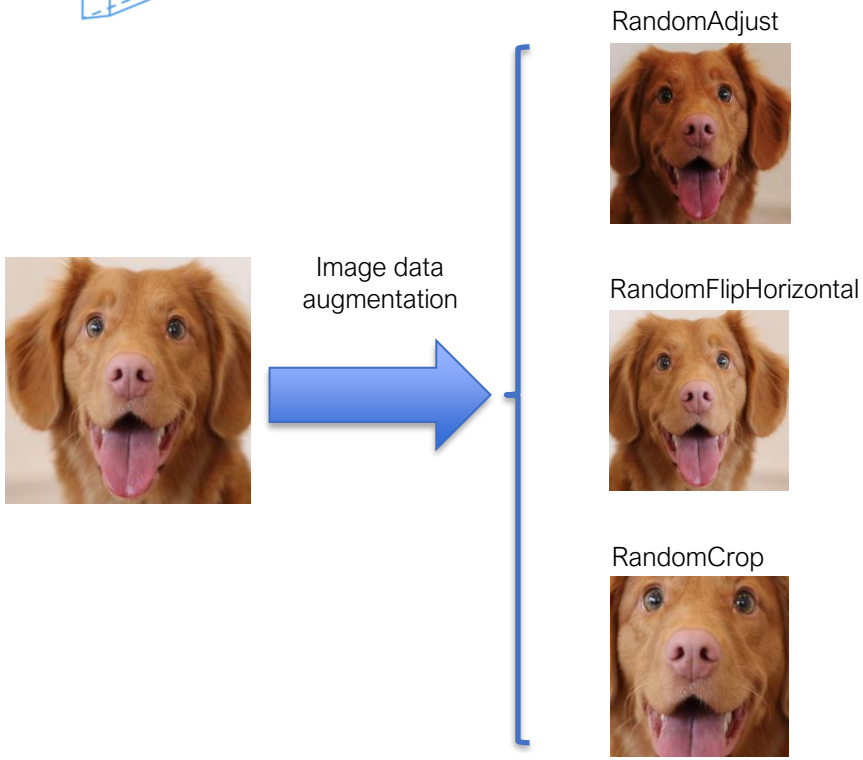




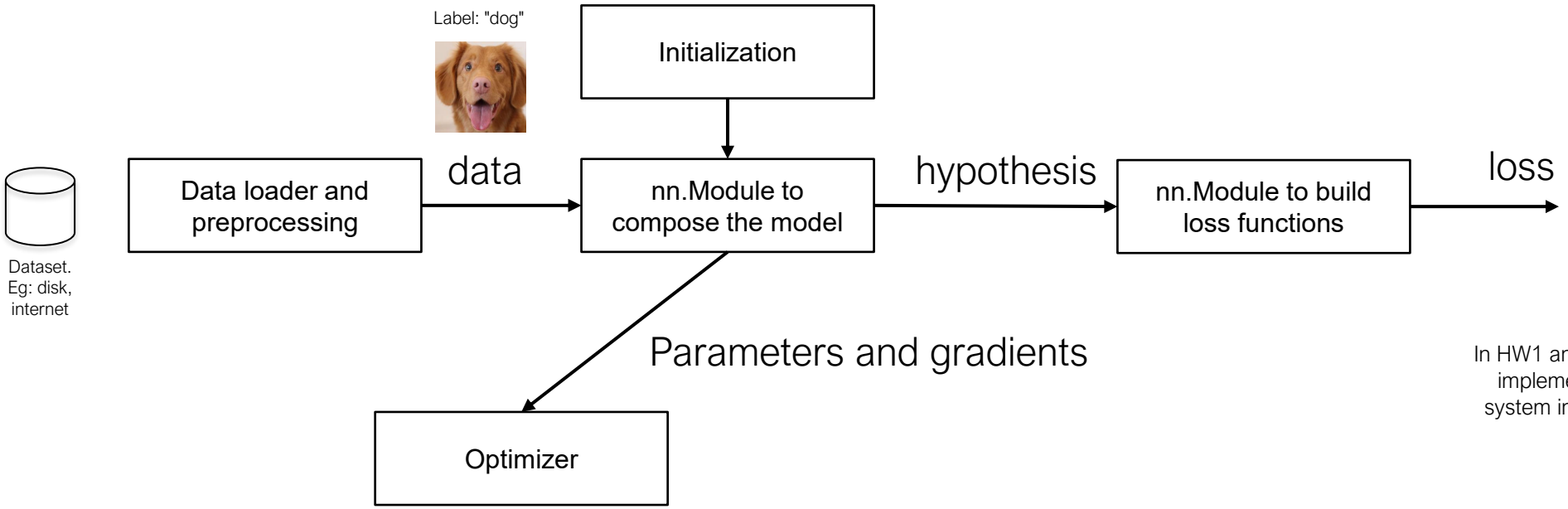
We often preprocess (augment) the dataset by randomly shuffle and transform the input

Data augmentation can significantly boost task performance in machine learning.

Data loading and augmentation is also compositional in nature

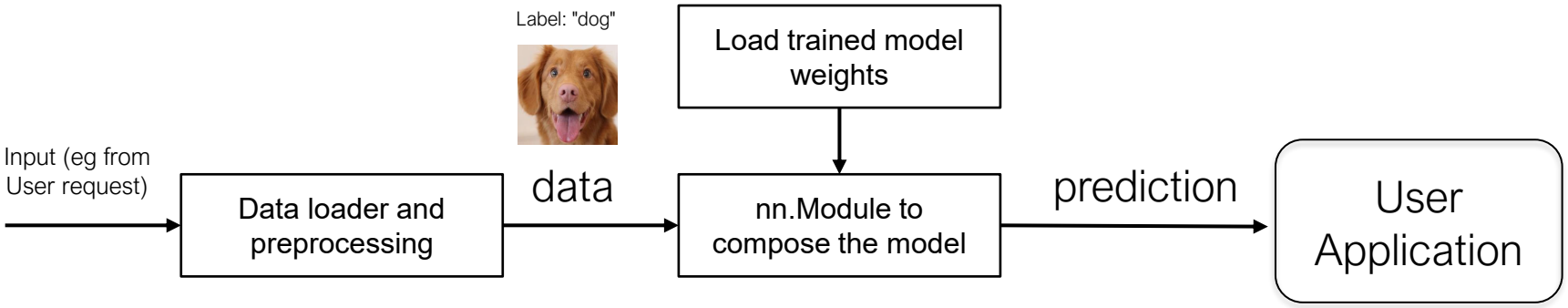


The deep learning pipeline: training



Deep learning libraries like pytorch, tensorflow (and needle!) own this entire system. Neat!

The deep learning pipeline: serving



Serving deep learning models in production is very similar in spirit.

Online serving: requests come in one-by-one, model should return response quickly so that app isn't laggy ("realtime"). **Prioritize latency over throughput**.
Offline serving: batch processing job. Example: daily job that calculates prediction scores on entire corpus. **Prioritize throughput over latency.**

Often, people employ techniques to accelerate models for serving ("inference mode").
Example: for pytorch: torchscript/torch.compile/torch.export.
For Nvidia GPU's: TensorRT (aka "compile pytorch model to something optimized to run on nvidia GPU's")

ML serving is a deep topic ("ML infra"), with lots of interesting stuff going on these days. Neat stuff!