Lecture 6 – Computational Graphs; PyTorch and Tensorflow

DD2424

April 11, 2019

Outline

- First Part
  - Computation Graphs
  - TensorFlow
  - PyTorch
  - Notes
- Second Part

Frameworks

PYTORCH

TensorFlow

mxnet

Keras

Microsoft CNTK

Caffe

Caffe2

MatConvNet

PYTORCH

TensorFlow

mxnet

K Keras

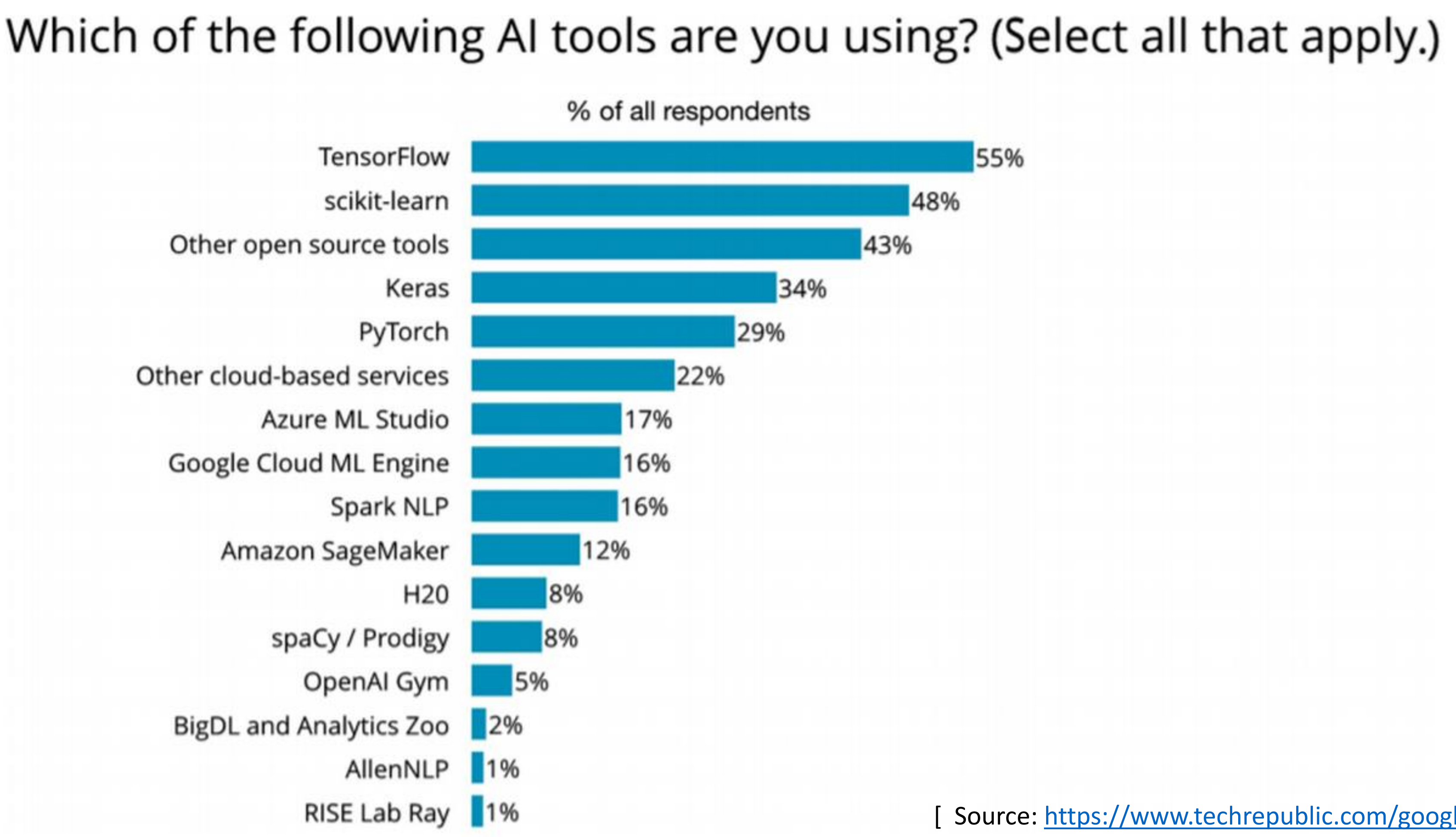
Microsoft CNTK

Caffe

Caffe2

MatConvNet

O'Reilly Poll: Most popular framework for machine learning



[Source: https://www.techrepublic.com/google-amp/article/most-popular-programming-language-frameworks-and-tools-for-machine-learning/]

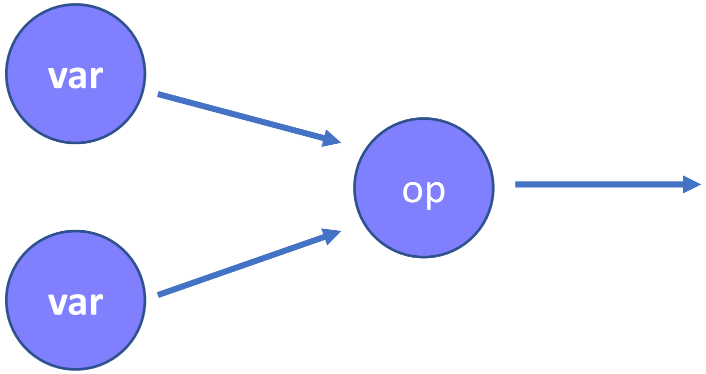
What are computation graphs?

Computation Graph

- DAG (directed acyclic graph)

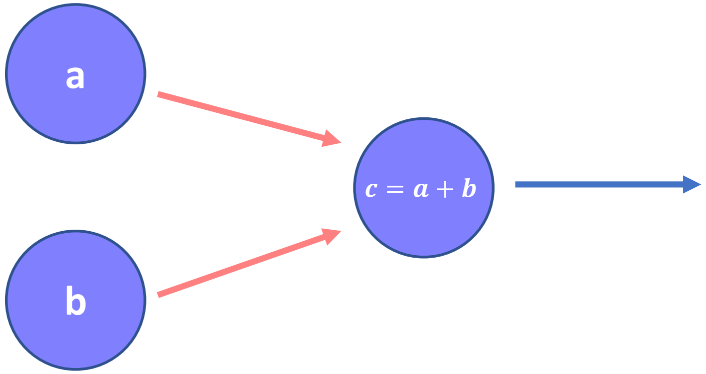
- Nodes
  - Variables
  - Mathematical Operations

- Edges
  - Feeding input



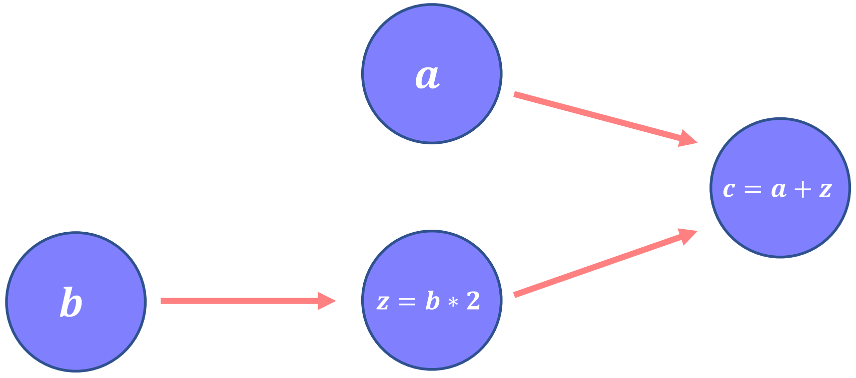
Computation Graph

- $c = a + b$

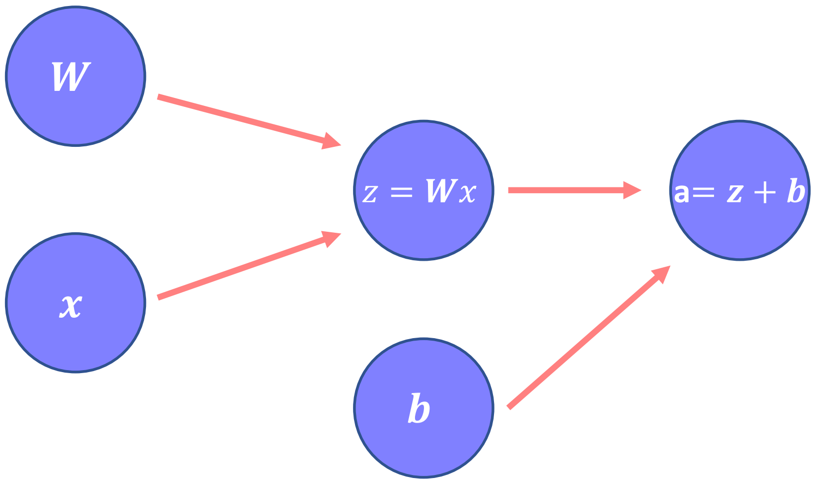


Computation Graph

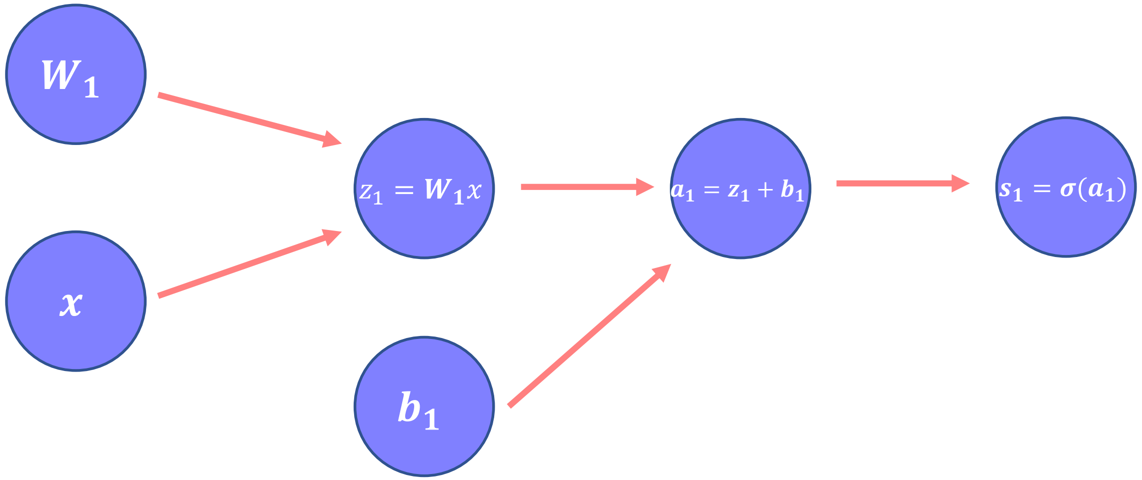
- $c = a + b * 2$



- Tensors: Multi-dimensional arrays
- $\boldsymbol{a} = W\boldsymbol{x} + \boldsymbol{b}$

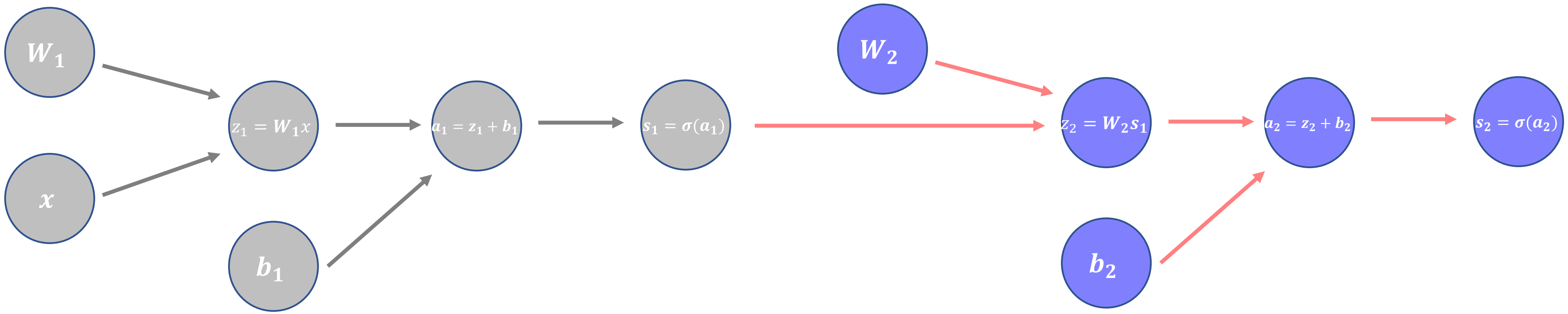


- A feed-forward neural network



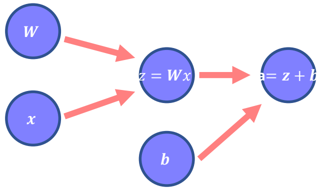
Computation Graph

- A multi-layer feed-forward neural network



Python (NumPy)

```
In [1]: import numpy as np

In [3]: x = np.ndarray(shape=(10,1), dtype=float)          # the input vector
        W = np.random.rand(100, 10)                        # the weight matrix
        b = np.ones(shape=(100, 1), dtype=float)*0.1       # the bias vector

        z = np.matmul(W, x)
        a = z + b

        s = np.maximum(a, 0)                               # ReLU activtion function
```



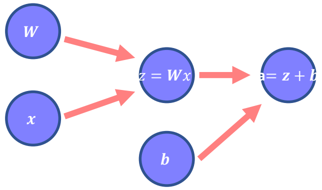
NumPy

```python
In [1]: import numpy as np

In [3]: x = np.ndarray(shape=(10,1), dtype=float)                  # the input vector
        W = np.random.rand(100, 10)                                # the weight matrix
        b = np.ones(shape=(100, 1), dtype=float)  1                # the bias vector

        z = np.matmul(W, x)
        a = z + b

        s = np.maximum(a, 0)                                       # ReLU activtion function
```

Similar Syntax!

PyTorch

```python
In [1]: from __fu      ort print_function
        import torc

In [6]: x = torch.Tensor(10, 1)                                    # the input vector
        W = torch.rand(100, 10)                                    # the weight matrix
        b = torch.ones(100, 1)*0.1                                 # the bias vector

        z = torch.matmul(W, x)
        a = z + b

        s = torch.clamp(a, min=0)                                  # ReLU activation function
```



NumPy

```
In [1]: import numpy as np
```

```
In [3]: x = np.ndarray(shape=(10,1), dtype=float)          # the input vector
        W = np.random.rand(100, 10)                        # the weight matrix
        b = np.ones(shape=(100, 1), dtype=float)*0.1       # the bias vector

        z = np.matmul(W, x)
        a = z + b

        s = np.maximum(a, 0)                               # ReLU activtion function
```

PyTorch

```
In [1]: from __future__ import print_function
        import torch
```

```
In [6]: x = torch.Tensor(10, 1)                            # the input vector
        W = torch.rand(100, 10)                            # the weight matrix
        b = torch.ones(100, 1)*0.1                         # the bias vector

        z = torch.matmul(W, x)
        a = z + b

        s = torch.clamp(a, min=0)                          # ReLU activation function
```

Not always!

PyTorch-NumPy

- Converting a Torch Tensor to a NumPy array and vice versa is a breeze.

```
In [9]: a = torch.ones(5)
        print(a)

 1
 1
 1
 1
 1
[torch.FloatTensor of size 5]

In [10]: b = a.numpy()
         print(b)

[1. 1. 1. 1. 1.]
```

PyTorch-NumPy

- Converting a Torch Tensor to a NumPy array and vice versa is a breeze.

```
In [9]: a = torch.ones(5)
        print(a)

 1
 1
 1
 1
 1
[torch.FloatTensor of size 5]

In [10]: b = a.numpy()
         print(b)

[1. 1. 1. 1. 1.]

In [11]: a.add_(1)
         print(a)
         print(b)

 2
 2
 2
 2
 2
[torch.FloatTensor of size 5]

[2. 2. 2. 2. 2.]
```

Shared Memory

- Converting a Torch Tensor to a NumPy array and vice versa is a breeze.

```
In [9]:  a = torch.ones(5)
         print(a)

          1
          1
          1
          1
          1
         [torch.FloatTensor of size 5]

In [10]: b = a.numpy()
         print(b)

         [1. 1. 1. 1. 1.]

In [11]: a.add_(1)
         print(a)
         print(b)

          2
          2
          2
          2
          2
         [torch.FloatTensor of size 5]

         [2. 2. 2. 2. 2.]
```

```
In [12]: import numpy as np
         a = np.ones(5)
         b = torch.from_numpy(a)
         np.add(a, 1, out=a)
         print(a)
         print(b)

         [2. 2. 2. 2. 2.]

          2
          2
          2
          2
          2
         [torch.DoubleTensor of size 5]
```

“Define by Run” Computation Graphs

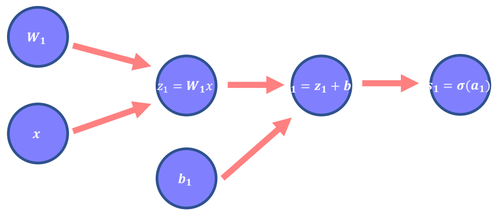
This kind of computation graph is called “define by run“

Also referred to as “dynamic”

“Define and Run” Computation Graphs

- First define the graph structure
- Then run it by feeding in the (input) variables.

Define graph G



Run the graph G

- Run G with x_1, W_1, b_1
- Run G with x_2, W_2, b_2
- …

Also known as “static graphs”

TensorFlow

Define graph

Run graph many times

```
In [13]: import tensorflow as tf

In [16]: # define the graph
         x = tf.placeholder(shape=(10, 1), dtype=tf.float16)          # the input vector
         W = tf.placeholder(shape=(100, 10), dtype=tf.float16)        # the weight matrix
         b = tf.placeholder(shape=(100, 1), dtype=tf.float16)         # the bias vector

         z = tf.matmul(W, x)
         a = z + b

         s = tf.maximum(a, 0)                                          # ReLU activation function

In [18]: # run the graph in a session

         with tf.Session() as sess:
             s_val = sess.run([s], feed_dict={x:np.random.rand(10, 1),
                                              W:np.random.rand(100, 10),
                                              b:np.random.rand(100,1)})    # run the graph

             s_val = sess.run([s], feed_dict={x:np.random.rand(10, 1),
                                              W:np.random.rand(100, 10),
                                              b:np.random.rand(100,1)})    # run the graph with different parameters

             z_val = sess.run([z], feed_dict={x:np.random.rand(10, 1),
                                              W:np.random.rand(100, 10),
                                              b:np.random.rand(100,1)})    # run the initial part of the graph

             a_val = sess.run([a], feed_dict={z:np.random.rand(100, 1),
                                              b:np.random.rand(100,1)})    # run the middle part of the graph
```

- Dynamic Graph

```
for x in X:
    x = torch.Tensor(10, 1)
    W = torch.rand(100, 10)
    b = torch.ones(100, 1)*0.1

    z = torch.matmul(W, x)
    a = z + b

    s = torch.clamp(a, min=0)
```

- Static Graph

```
x = tf.placeholder(shape=(10, 1), dtype=tf.float16)
W = tf.placeholder(shape=(100, 10), dtype=tf.float16)
b = tf.placeholder(shape=(100, 1), dtype=tf.float16)

z = tf.matmul(W, x)
a = z + b

s = tf.maximum(a, 0)

with tf.Session() as sess:
    for x_val in X:
        s_val = sess.run([s], feed_dict={x:x_val})
```

Why computation graphs at all?!

Why computation graphs?

- In lecture 3, you've learnt how to do backprop using the chain rule

The Chain Rule for the composition of n functions

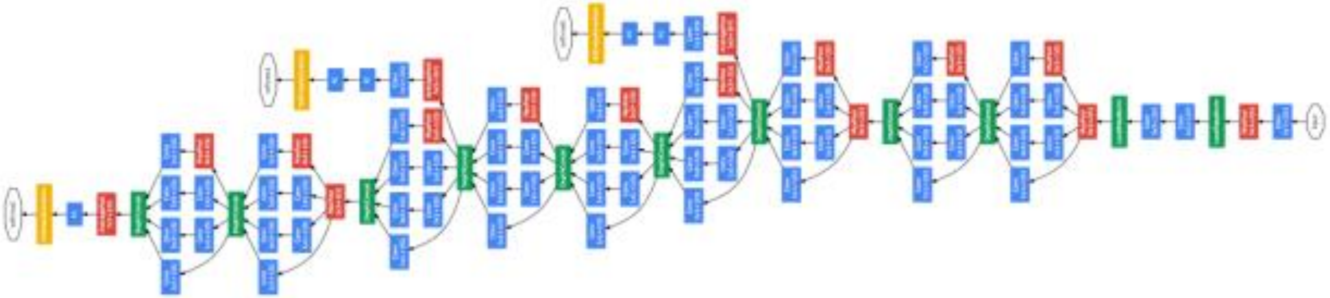
Recursively applying this fact gives:

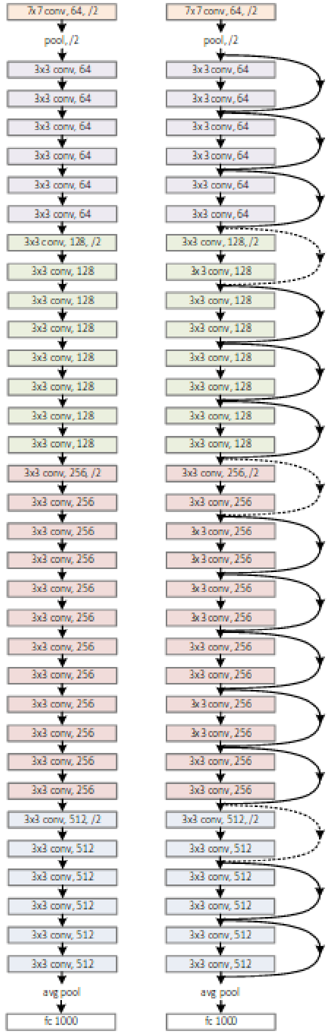
$$
\begin{aligned}
\frac{dh(x)}{dx} &= \frac{dg_1(x)}{dx} && \leftarrow \text{Apply } h = g_1 \\
&= \frac{d(g_2 \circ f_1)(x)}{dx} && \leftarrow \text{Apply } g_1 = g_2 \circ f_1 \\
&= \frac{dg_2(y_1)}{dy_1}\frac{df_1(x)}{dx} && \leftarrow \text{Apply chain rule \& } y_1 = f_1(x) \\
&= \frac{d(g_3 \circ f_2)(y_1)}{dy_1}\frac{df_1(x)}{dx} && \leftarrow \text{Apply } g_2 = g_3 \circ f_2 \\
&= \frac{dg_3(y_2)}{dy_2}\frac{df_2(y_1)}{dy_1}\frac{df_1(x)}{dx} && \leftarrow \text{Apply chain rule \& } y_2 = f_2(y_1) \\
&\;\;\vdots \\
&= \frac{dg_n(y_{n-1})}{dy_{n-1}}\frac{df_{n-1}(y_{n-2})}{dy_{n-2}}\cdots\frac{df_2(y_1)}{dy_1}\frac{df_1(x)}{dx} \\
&= \frac{df_n(y_{n-1})}{dy_{n-1}}\frac{df_{n-1}(y_{n-2})}{dy_{n-2}}\cdots\frac{df_2(y_1)}{dy_1}\frac{df_1(x)}{dx} && \leftarrow \text{Apply } g_n = f_n
\end{aligned}
$$

where $y_j = (f_j \circ f_{j-1} \circ \cdots \circ f_1)(x) = f_j(y_{j-1})$.

Why computation graphs?

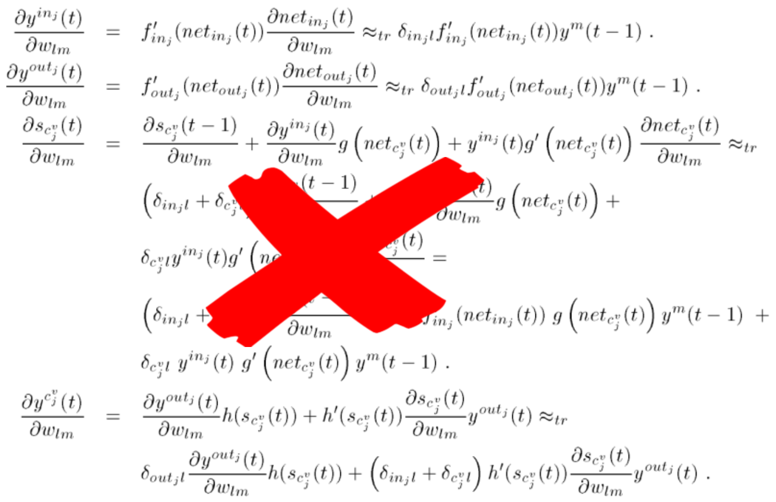
- Is it feasible?





Why computation graphs?

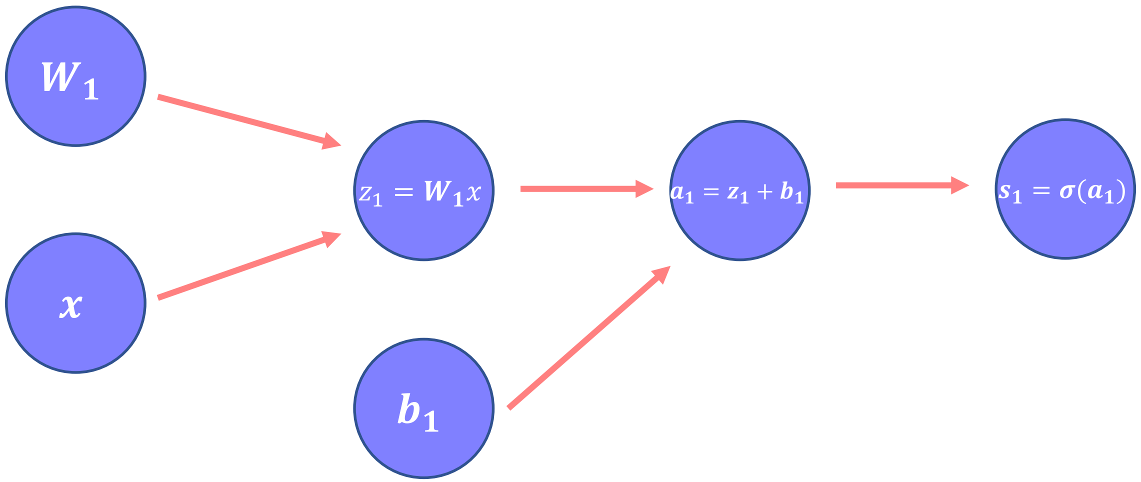
- Automatic chain rule
  - automatic back-prop using implemented operations
    - Each operation has their gradient already implemented
    - If you want to use a novel operation, then you have to provide it’s gradient w.r.t. inputs and its learnable parameters (if any)



Let’s look at examples in PyTorch and TensorFlow

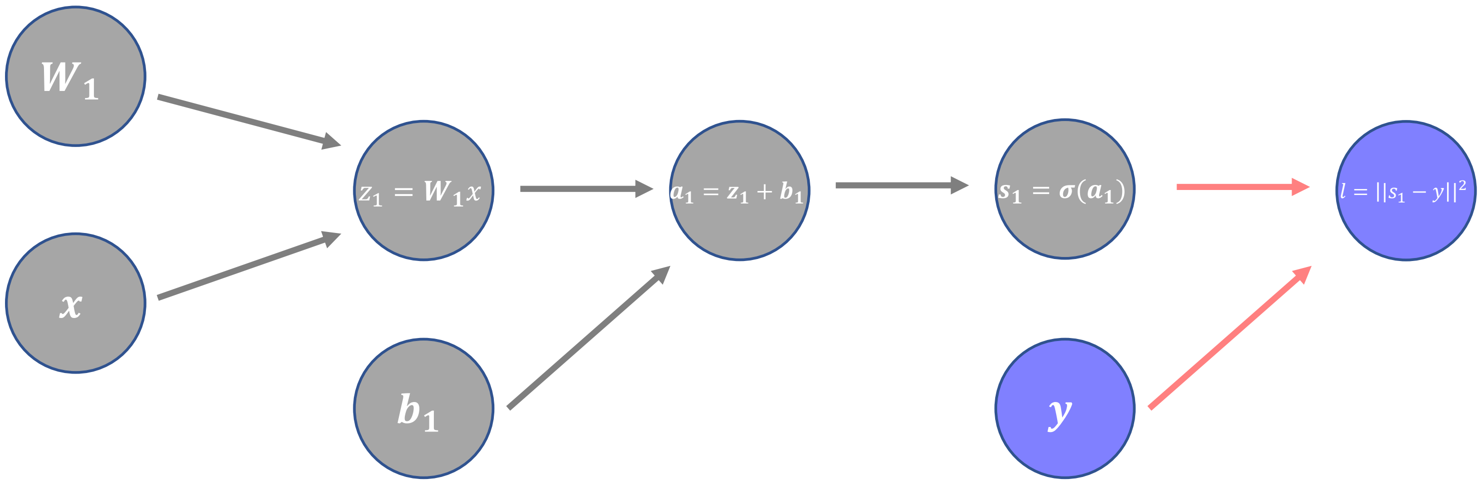
Computation Graph

- A feed-forward neural network

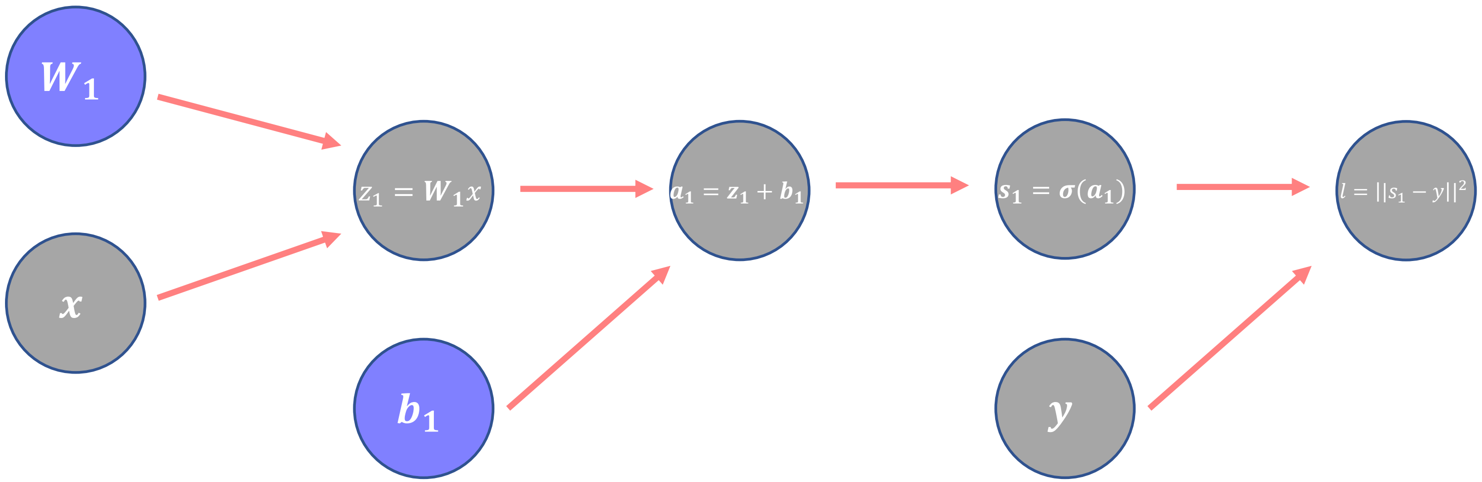


Computation Graph

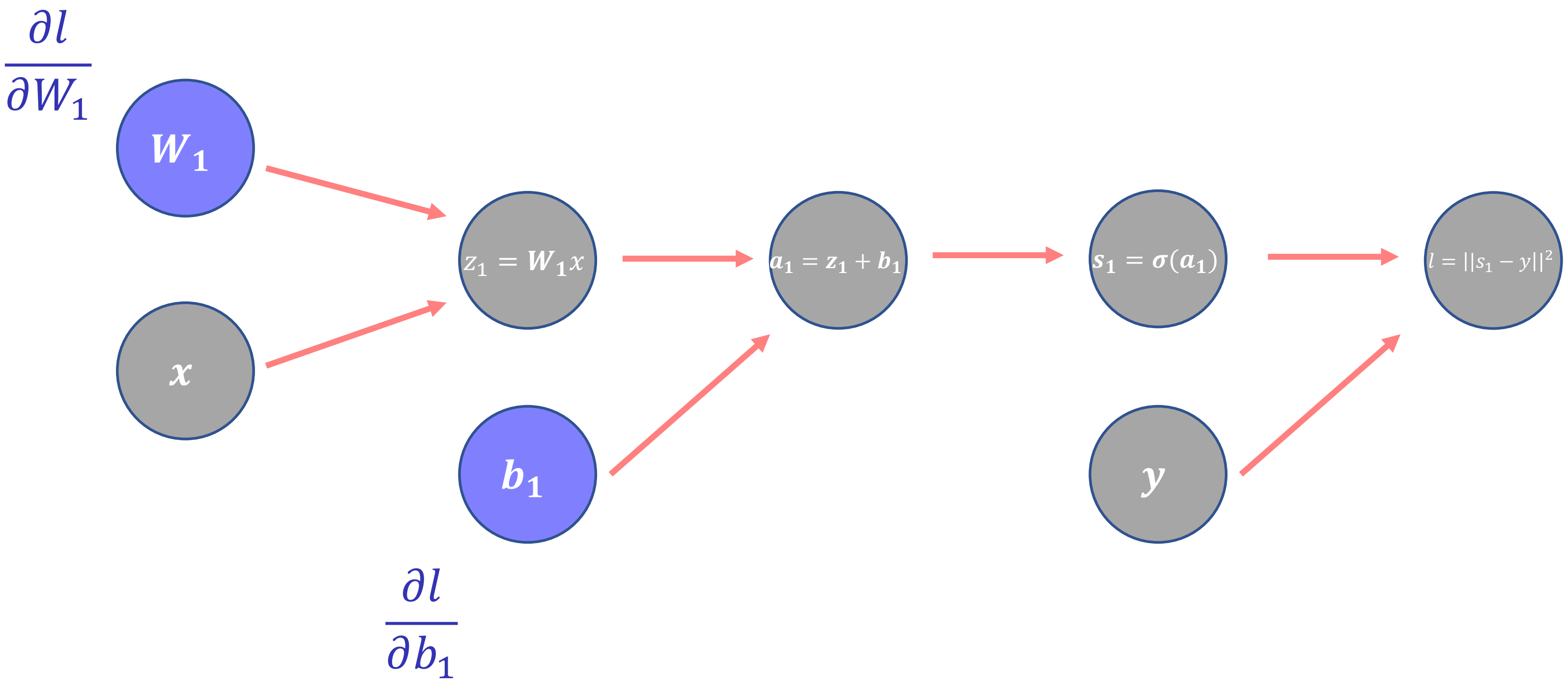
- A feed-forward neural network with squared L_2 loss



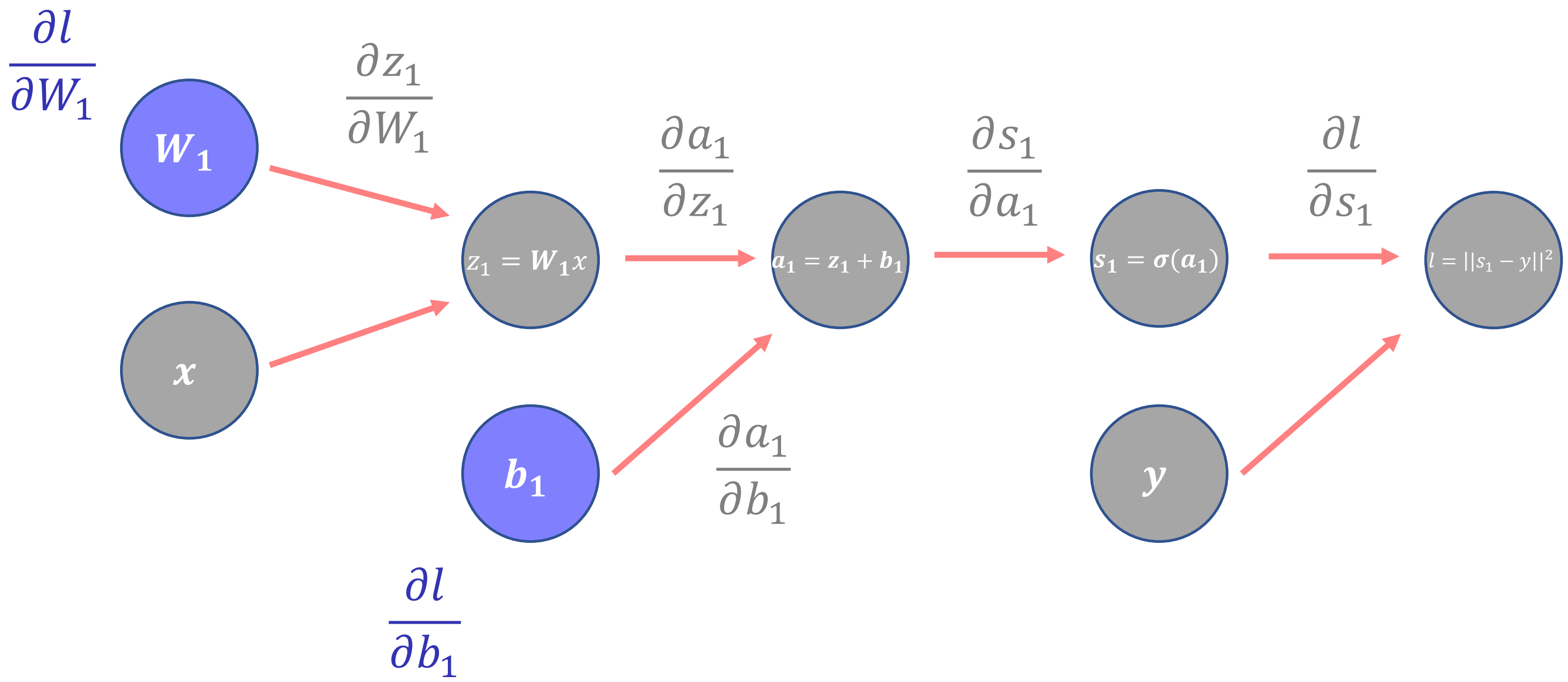
- Learnable parameters



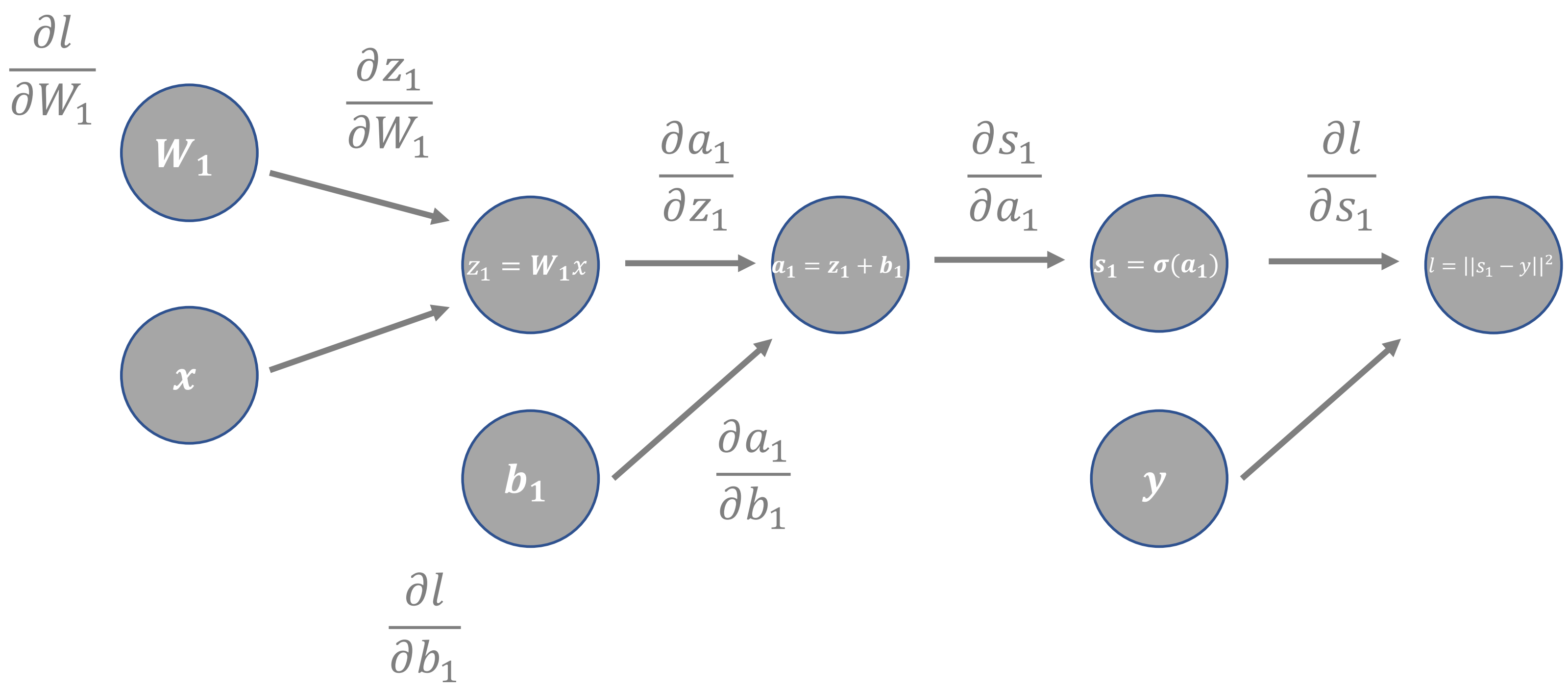
Backprop in Computation Graph



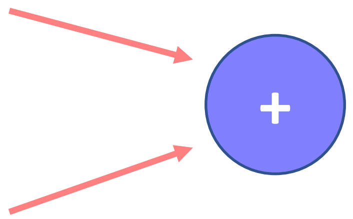
Backprop in Computation Graph



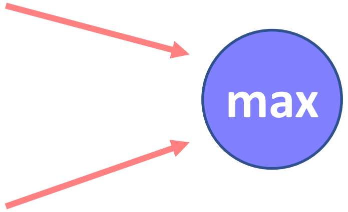
A deep learning framework provides an automatic gradient calculation of its output variables w.r.t. its input variables



- Addition Node
  - Forward pass: $a = b + c$
  - Backward pass: $\frac{\partial a}{\partial b} = 1$ and $\frac{\partial a}{\partial c} = 1$

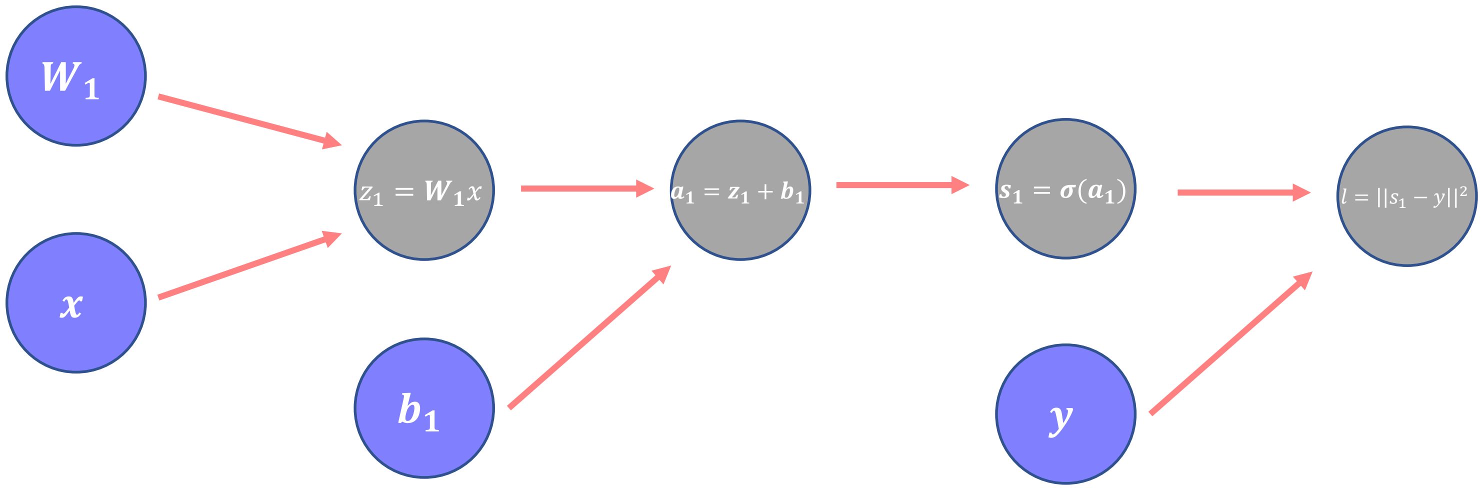


- Max Node
  - Forward pass: $a = \max(b, c)$
  - Backward pass:
    - *If b < c*
      - $\frac{\partial a}{\partial b} = 0$ and $\frac{\partial a}{\partial c} = 1$
    - *If b > c*
      - $\frac{\partial a}{\partial b} = 1$ and $\frac{\partial a}{\partial c} = 0$



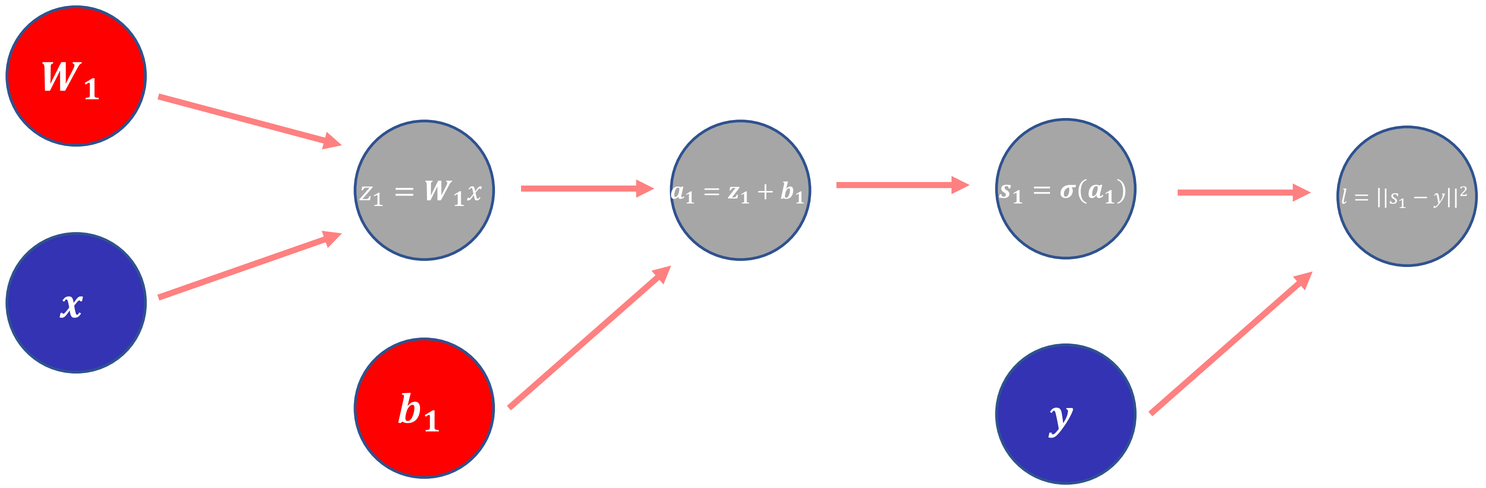
Variables and Ops

- Ops
  - Intermediate or final nodes
- Variables
  - intrinsic parameters of the model
  - input to the model



Variables and Ops

- Ops
  - Intermediate or final nodes
- Variables
  - intrinsic parameters of the model
  - input to the model

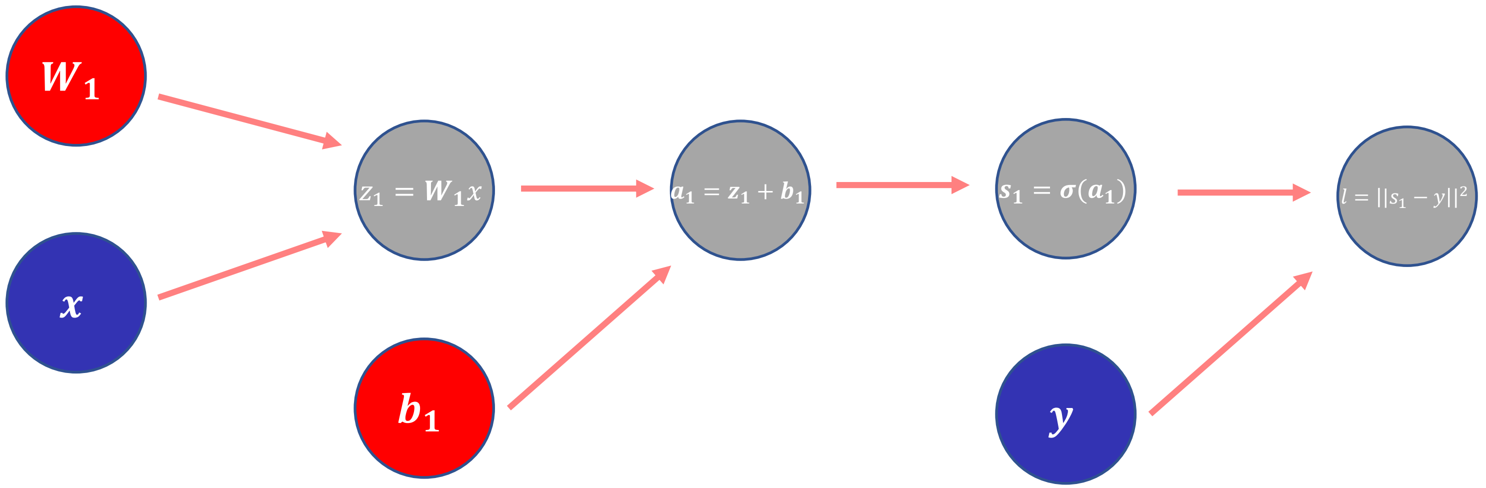


Variables and Ops

- Variables
  - Intrinsic parameters of the model
  - Input to the model

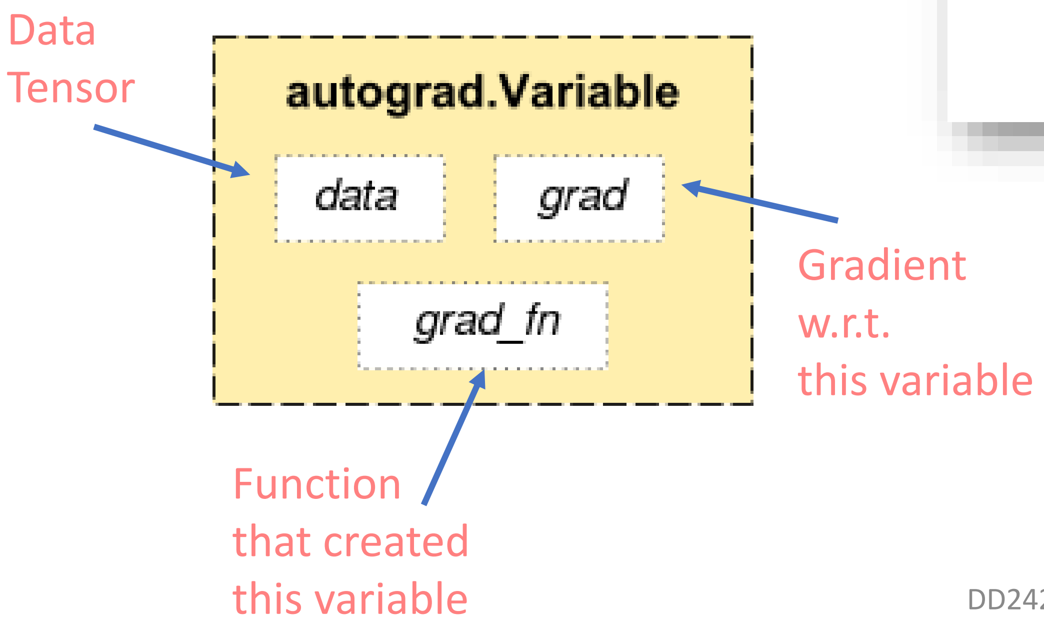
- TensorFlow
  - Variables
  - Place Holders

- PyTorch
  - Variables



PyTorch Autograd

- package: torch.autograd

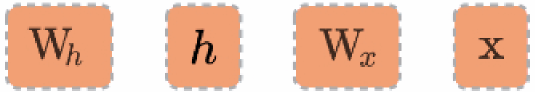



```
In [16]: import torch
         from torch.autograd import Variable

In [35]: x = Variable(torch.rand(10, 1), requires_grad=False)    # the input vector
         W = Variable(torch.rand(100, 10), requires_grad=True)   # the weight matrix
         b = Variable(torch.ones(100, 1)*0.1, requires_grad=True) # the bias vector

         z = torch.matmul(W, x)
         a = z + b

         s = torch.clamp(a, min=0)                                # ReLU activation function
```

Pytorch Autograd

A graph is created on the fly

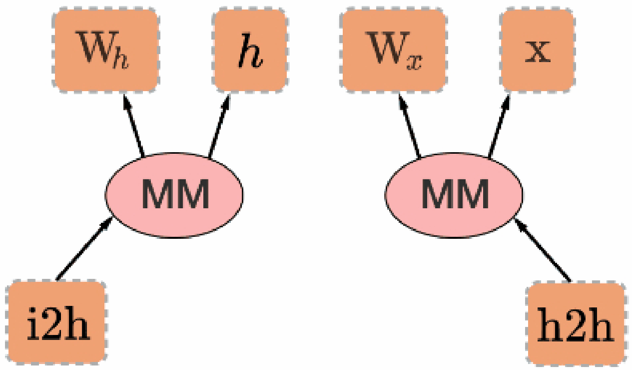
```
from torch.autograd import Variable

x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))
```



Pytorch Autograd

A graph is created on the fly

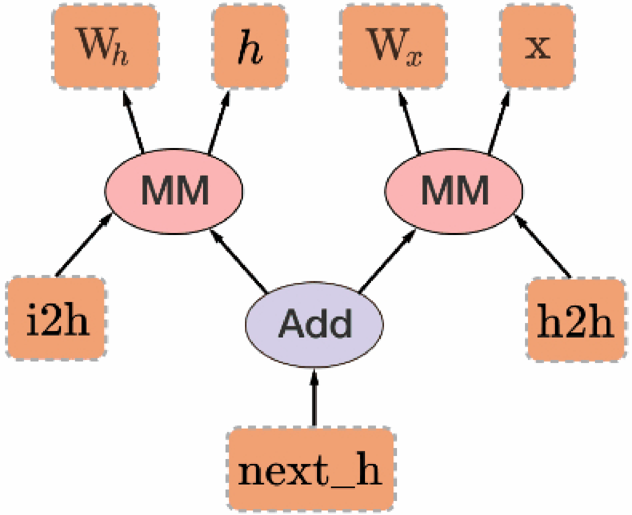
```
from torch.autograd import Variable

x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))

i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
```



Pytorch Autograd

A graph is created on the fly

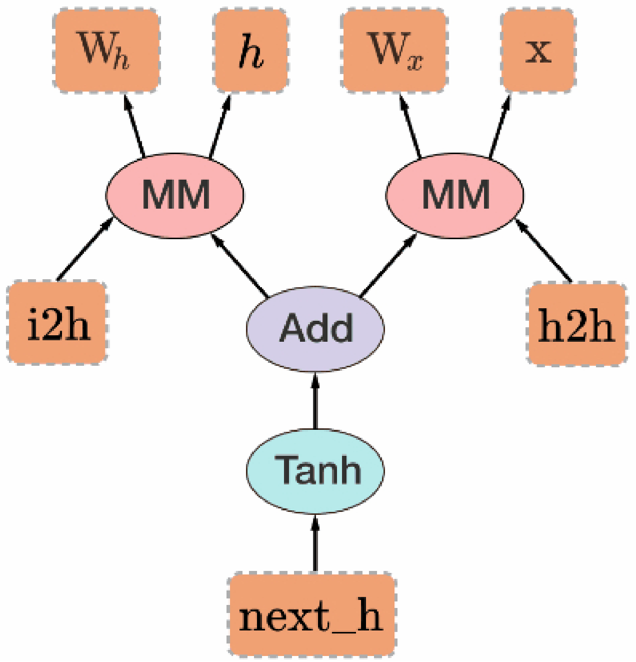
```
from torch.autograd import Variable

x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))

i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
```



Pytorch Autograd

A graph is created on the fly

```
from torch.autograd import Variable

x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))

i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
next_h = next_h.tanh()
```



- Calculate gradient using backward() method of a Variable

- `var.backward()`

```
In [21]: x = Variable(torch.rand(10, 1), requires_grad=True)     # the input vector
         W = Variable(torch.rand(1, 10), requires_grad=True)    # the weight matrix
         b = Variable(torch.ones(1, 1)*0.1, requires_grad=True) # the bias vector
         y = 1

         z = torch.matmul(W, x)
         a = z + b

         s = torch.clamp(a, min=0)                               # ReLU activation function
         l = (s-y)*(s-y)                                         # SE loss
```

```
In [22]: l.backward()
         print(W.grad, b.grad)

         (Variable containing:
          0.9587  1.5793  0.6712  0.6553  2.1702  0.5288  0.5278  1.6710  1.0869  0.4436
         [torch.FloatTensor of size 1x10]
         , Variable containing:
          2.4148
         [torch.FloatTensor of size 1x1]
         )
```

TensorFlow gradients

- Add gradient nodes in the graph where necessary using

`Tf.gradients(ys, xs, gs)`

- And evaluate it

```
In [8]: # define the graph
x = tf.placeholder(shape=(10, 1), dtype=tf.float16)             # the input vector
y = 1                                                            # some label
W = tf.placeholder(shape=(1, 10), dtype=tf.float16)             # the weight matrix
b = tf.placeholder(shape=(1, 1), dtype=tf.float16)              # the bias vector

z = tf.matmul(W, x)
a = z + b

s = tf.maximum(a, 0)                                             # ReLU activation function

l = (y-s)*(y-s)                                                  # SE loss

grad_W, grad_b = tf.gradients(l, [W, b])
```

```
In [9]: W_val = np.random.rand(1, 10)                           # initialize W
b_val = np.random.rand(1, 1)                                     # initialize b

with tf.Session(config=tf.ConfigProto(log_device_placement=True)) as sess:
    l_val, l_grad_W, l_grad_b = sess.run([l, grad_W, grad_b],
                                         feed_dict={x:np.random.rand(10, 1),
                                                    W:W_val,
                                                    b:b_val})    # run the graph
    print(l_val, l_grad_W, l_grad_b)
```

```
(array([[ 3.9375]], dtype=float16), array([[ 1.51171875,  3.62890625,  1.15917969,  3.2265625 ,  1.09570312,
         1.36425781,  3.7109375 ,  2.23632812,  2.44921875,  0.05480957]], dtype=float16), array([[ 3.96875]], dtype=float16))
```

TensorFlow gradients

- Then update the parameters

```
# define the graph
x = tf.placeholder(shape=(10, 1), dtype=tf.float16)          # the input vector
y = 1                                                        # some label
W = tf.placeholder(shape=(1, 10), dtype=tf.float16)          # the weight matrix
b = tf.placeholder(shape=(1, 1), dtype=tf.float16)           # the bias vector

z = tf.matmul(W, x)
a = z + b

s = tf.maximum(a, 0)                                         # ReLU activation function

l = (y-s)*(y-s)                                              # SE loss

grad_W, grad_b = tf.gradients(l, [W, b])
```

```
W_val = np.random.rand(1, 10)                                # initialize W
b_val = np.random.rand(1, 1)                                 # initialize b
learning_rate = 0.01

with tf.Session(config=tf.ConfigProto(log_device_placement=True)) as sess:
    l_val, l_grad_W, l_grad_b = sess.run([l, grad_W, grad_b],
                                         feed_dict={x:np.random.rand(10, 1),
                                                    W:W_val,
                                                    b:b_val})    # run the graph
    print(l_val, l_grad_W, l_grad_b)

    W_val -= learning_rate*l_grad_W
    b_val -= learning_rate*l_grad_b
```

- Use tf.Variable instead

```
In [14]: # define the graph
x = tf.placeholder(shape=(10, 1), dtype=tf.float32)           # the input vector
y = 1                                                         # some label
W = tf.Variable(tf.random_normal((1, 10)))                    # the weight matrix
b = tf.Variable(tf.random_normal((1, 1)))                     # the bias vector

z = tf.matmul(W, x)
a = z + b

s = tf.maximum(a, 0)                                          # ReLU activation function

l = (y-s)*(y-s)                                               # SE loss

grad_W, grad_b = tf.gradients(l, [W, b])

learning_rate = 0.01
updated_W = W.assign(W-learning_rate*grad_W)
updated_b = b.assign(b-learning_rate*grad_b)
```

```
In [17]: with tf.Session(config=tf.ConfigProto(log_device_placement=True)) as sess:

    sess.run(tf.global_variables_initializer())

    l_val, l_updated_W, l_updated_b = sess.run([l, updated_W, updated_b],
                                  feed_dict={x:np.random.rand(10, 1)})    # run the graph
    print(l_val, l_grad_W, l_grad_b)
```

How to use GPU?

Turn variables into "GPU" variables by the following command:

- `var = var.cuda(#)`

```
In [3]: x = Variable(torch.rand(10, 1), requires_grad=False)
        W = Variable(torch.rand(100, 10), requires_grad=True)
        b = Variable(torch.ones(100, 1)*0.1, requires_grad=True)

        x = x.cuda()
        W = W.cuda()
        b = b.cuda()

        z = torch.matmul(W, x)
        a = z + b

        s = torch.clamp(a, min=0)

In [4]: print(z.is_cuda, a.is_cuda, s.is_cuda)

        (True, True, True)
```

PyTorch GPU

Turn back variables into “CPU” variables by the following command:

- `var = var.cpu()`

```
In [10]: x = x.cpu()
         W = W.cpu()
         b = b.cpu()

         z = torch.matmul(W, x)
         a = z + b

         s = torch.clamp(a, min=0)

In [11]: print(z.is_cuda, a.is_cuda, s.is_cuda)

         (False, False, False)
```

TensorFlow GPU

- In TF variables or operations can sit on specific device

- tf.device(/gpu:0)
- tf.device(/gpu:1)
- …
- tf.device(/cpu:0)

In [11]:
```
# define the graph
with tf.device('/gpu:0'):
    x = tf.placeholder(shape=(10, 1), dtype=tf.float16)          # the input vector
    W = tf.placeholder(shape=(100, 10), dtype=tf.float16)        # the weight matrix
    b = tf.placeholder(shape=(100, 1), dtype=tf.float16)         # the bias vector

z = tf.matmul(W, x)
a = z + b

s = tf.maximum(a, 0)                                            # ReLU activation function
```

In [12]:
```
with tf.Session(config=tf.ConfigProto(log_device_placement=True)) as sess:
    s_val = sess.run([s], feed_dict={x:np.random.rand(10, 1),
                                     W:np.random.rand(100, 10),
                                     b:np.random.rand(100,1)})    # run the graph
```

TensorFlow GPU

- In TF variables or operations can sit on specific device

```
tf.Session(config=tf.ConfigProto(log_device_placement=True))
```

In [11]:
```python
# define the graph
with tf.device('/gpu:0'):
    x = tf.placeholder(shape=(10, 1), dtype=tf.float16)         # the input vector
    W = tf.placeholder(shape=(100, 10), dtype=tf.float16)       # the weight matrix
    b = tf.placeholder(shape=(100, 1), dtype=tf.float16)        # the bias vector

z = tf.matmul(W, x)
a = z + b

s = tf.maximum(a, 0)                                            # ReLU activation function
```

In [12]:
```python
with tf.Session(config=tf.ConfigProto(log_device_placement=True)) as sess:
    s_val = sess.run([s], feed_dict={x:np.random.rand(10, 1),
                                     W:np.random.rand(100, 10),
                                     b:np.random.rand(100,1)})    # run the graph
```

```
MatMul: (MatMul): /job:localhost/replica:0/task:0/device:GPU:0
2018-04-10 12:59:09.508497: I tensorflow/core/common_runtime/placer.cc:874] MatMul: (MatMul)/job:localhost/replica:0/task:0/device:GPU:0
add: (Add): /job:localhost/replica:0/task:0/device:GPU:0
2018-04-10 12:59:09.508513: I tensorflow/core/common_runtime/placer.cc:874] add: (Add)/job:localhost/replica:0/task:0/device:GPU:0
Maximum: (Maximum): /job:localhost/replica:0/task:0/device:GPU:0
2018-04-10 12:59:09.508525: I tensorflow/core/common_runtime/placer.cc:874] Maximum: (Maximum)/job:localhost/replica:0/task:0/device:GPU:0
Maximum/y: (Const): /job:localhost/replica:0/task:0/device:GPU:0
2018-04-10 12:59:09.508537: I tensorflow/core/common_runtime/placer.cc:874] Maximum/y: (Const)/job:localhost/replica:0/task:0/device:GPU:0
Placeholder_2: (Placeholder): /job:localhost/replica:0/task:0/device:GPU:0
2018-04-10 12:59:09.508548: I tensorflow/core/common_runtime/placer.cc:874] Placeholder_2: (Placeholder)/job:localhost/replica:0/task:0/device:GPU:0
Placeholder_1: (Placeholder): /job:localhost/replica:0/task:0/device:GPU:0
2018-04-10 12:59:09.508558: I tensorflow/core/common_runtime/placer.cc:874] Placeholder_1: (Placeholder)/job:localhost/replica:0/task:0/device:GPU:0
Placeholder: (Placeholder): /job:localhost/replica:0/task:0/device:GPU:0
2018-04-10 12:59:09.508567: I tensorflow/core/common_runtime/placer.cc:874] Placeholder: (Placeholder)/job:localhost/replica:0/task:0/device:GPU:0
```

TensorFlow GPU

- Some TF operations do not have a CUDA implementation

```
tf.Session(config=tf.ConfigProto(
      allow_soft_placement=True, log_device_placement=True))
```

How to implement complicated models in practice?

- PyTorch package called nn and class called Module

In [9]:
```python
import torch
from torch.autograd import Variable
from collections import OrderedDict

input_dim = 10
hidden_dims = [100, 100, 100]
output_dim = 1

layers = [('input',torch.nn.Linear(input_dim, hidden_dims[0]))]
for l in range (1, len(hidden_dims)):
        layers.append(('relu'+str(l),torch.nn.ReLU()))
        layers.append(('hid'+str(l), torch.nn.Linear(hidden_dims[l-1], hidden_dims[l])))
layers.append(('relu'+str(len(hidden_dims)),torch.nn.ReLU()))
layers.append(('output', torch.nn.Linear(hidden_dims[-1], output_dim)))
model = torch.nn.Sequential(OrderedDict(layers))

loss = torch.nn.MSELoss()

optimizer = torch.optim.Adam(model.parameters(), lr=0.01)

num_samples = 100
X = Variable(torch.randn(num_samples, input_dim))
Y = Variable(torch.randn(num_samples, output_dim))

for i in range(10):
        preds = model(X)
        l = loss(preds, Y)
        print(l)

        optimizer.zero_grad()
        l.backward()

        optimizer.step()
```

TF High-Level Libraries

- Keras: highest abstraction
- SLIM: best pre-trained models
- TFLearn,
- Sonnet,
- Pretty Tensor,
- …

In [8]:
```
from keras.models import Sequential
from keras.layers.core import Dense, Activation
from keras.optimizers import Adam
import numpy as np

input_dim = 10
hidden_dims = [100, 100, 100]
output_dim = 1

model = Sequential()
model.add(Dense(input_dim=input_dim, output_dim=hidden_dims[0]))
for l in range(1, len(hidden_dims)):
        model.add(Dense(input_dim=hidden_dims[l-1], output_dim=hidden_dims[l]))
model.add(Dense(input_dim=hidden_dims[-1], output_dim=output_dim))

optimizer = Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=None, decay=0.0, amsgrad=False)
model.compile(loss='mean_squared_error', optimizer=optimizer)

num_samples = 100
X = np.random.rand(num_samples, input_dim)
Y = np.random.rand(num_samples, output_dim)

hist = model.fit(X, Y, epochs=10, batch_size=32, shuffle=True)
```

Data, storage, and loading

!!!!Important!!!

- Always monitor CPU/GPU usage (linux: nvidia-smi, top)
- Make storage more efficient (TF Records, etc.)
- Make reading pipeline more efficient (parallel readers, prefetching, etc.)

Use Visualization

- Always monitor the loss function on the training and validation sets visually

- Monitor all other important scalars, such as learning rate, regularization loss, layer activations summary, how full your data queues are, and …

- If you have an imbalanced classification problem, visualize the CE loss separately for each class.

- If you work with images, time to time visualize samples from the batch, if you do data augmentation, visualize the original sample as well as the augmented one

- TensorBoard for TF
- TensorBoardX, matplotlib, seaborn, … for PT

Use Visualization

You can have the configuration shown as a text file in tensorboard!

Which one is better? PyTorch or TensorFlow?

pros and cons

- PyTorch: easier for prototyping
- PyTorch: much easier to implement flexible graphs
- PyTorch: different structures in each iteration (dependent on data). This is possible with TF too, but is a pain.
- PyTorch: manipulating weight and gradients
- PyTorch: code-level debugging (breakpoints, imperative, tracing your own code instead of TF kernels)
- PyTorch: probably better abstractions for dataset, variable, parallelism, etc. but TF has many high-level wrappers with better abstractions
- Tie?!: Faster run-time, (NHWC v.s. NCHW)
- TF: TensorBoard
- TF: research-level debugging (TensorBoard)
- TF: windows
- TF: distributed training (PyTorch has it now too, but seems not as developed as the TF version)
- TF: easier with distributing the code over multiple devices (GPUs/CPU) (maybe not anymore)
- TF: online community is noticeably larger
- TF: data readers
- TF: supposedly more optimizations of the graph (done by the engine)
- TF: documentation and tutorials
- TF: more models available
- TF: Serialization, code and portability (saving and loading models for across platforms, or checkpoints)
- TF: Deployment: Server, Mobile, etc. (TensorFlow Serving, TensorFlow Lite)
- TF: Richer API (e.g. FFT)
- TF: Automatic shape inference
- TF has a MOOC: https://eu.udacity.com/course/deep-learning--ud730

TensorFlow Eager execution

- Eager Execution
  - Dynamic!
  - tf.enable_eager_execuation()
  - Considerably Slower (being worked on)
  - https://www.tensorflow.org/guide/eager

Caffe(2)

- Portability is seamless (e.g. mobile apps)
- Simplest framework for fine-tuning or feature extraction
- Used to be fastest (Caffe)

Summary

- Don’t take the following statements too seriously! -- it depends on many factors
  - If you want to use pretrained classic deep networks (AlexNet, VGG, ResNet, …) for feature extraction and/or fine-tuning → Use Caffe and/or Caffe2
  - If you have a mobile application in mind → Use Caffe/Caffe2 or TensorFlow
  - If you want more pythonic → use PyTorch
  - If you are familiar with Matlab and don’t need much flexibility or advanced layers → use MatConvNet
  - If you don’t want so much of flexibility and still use python → use Keras
  - If you are working on NLP applications or complicated RNNs → use PyTorch
  - If you want large community help, sustainable learning of a framework → use TensorFlow
  - If you want to work on bleeding-edge papers → See what framework has the original and/or cleanest implementation (most likely TensorFlow)
  - If you want to prototype many different *novel* setups → Use PyTorch or TF Eager