Tsinghua University

Advanced Computer Vision

THU×SENSETIME – 80231202

sensetime

Chapter 1 - Section 5

Training Framwork and Model Optimization

Dr. Liu Yu

Friday, March 19, 2021

Acknowledge : Song Guanglu , Liu Boxiao , Zhang Manyuan

Outline

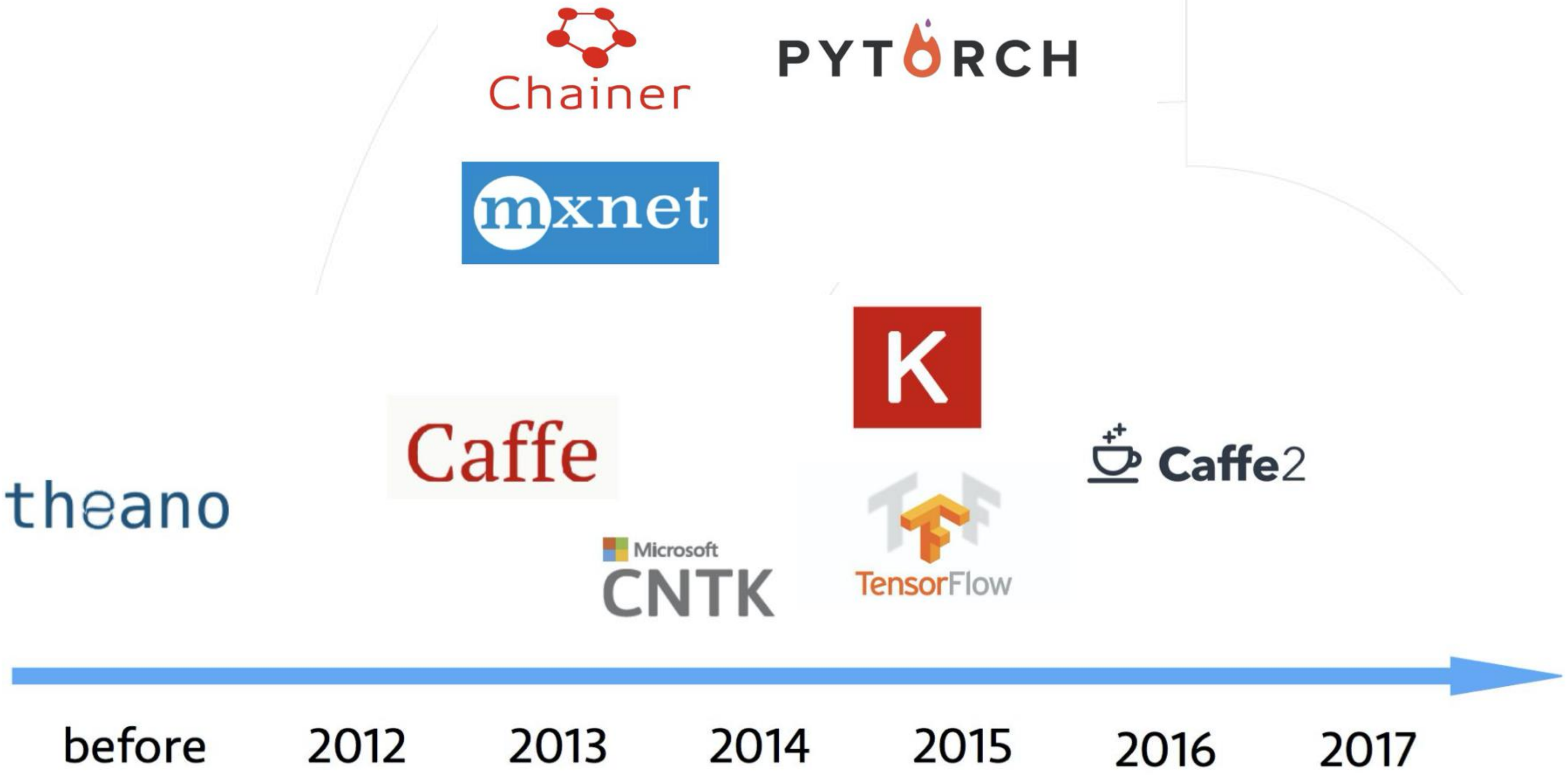
商汤 sensetime

清華大學 Tsinghua University

Highlights

- Learn to build neural network by PyTorch
- Learn to train a whole network
- Learn the features of basic optimizers
- Understand the FP16 training and visualization of Tensorboard
- Understand the features of different training frameworks

- **The history of deep learning framework**



gluon-cv.mxnet.io

Deep learning framework

- **The history of deep learning framework**

Caffe (UC Berkeley) → Caffe2 (Facebook)

Torch (Facebook) → Pytorch (Facebook)

Theano (U Montreal) → Tensorflow (Google)

PaddlePaddle (Baidu)

CNTK (Microsoft)

MXNet (Amazon/Apache)

Parrots (SenseTime)

Keras (Google)

… …

Deep learning framework

- **Caffe**

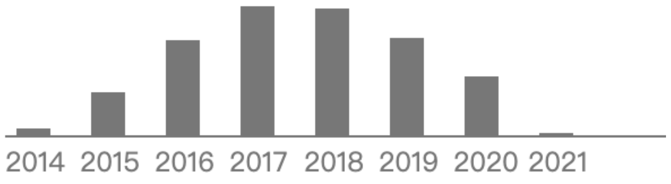
BVLC / caffe

Watch 2.2k | Star 31.3k | Fork 18.8k

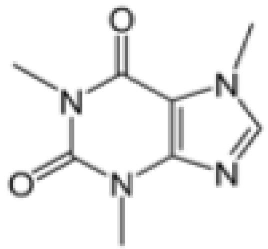
Code | Issues 873 | Pull requests 285 | Actions | Projects | Wiki | Security | Insights

Yangqing Jia created the project during his PhD at UC Berkeley **in 2013.**

Total citations [Cited by 15018](#)



BERKELEY ARTIFICIAL INTELLIGENCE RESEARCH

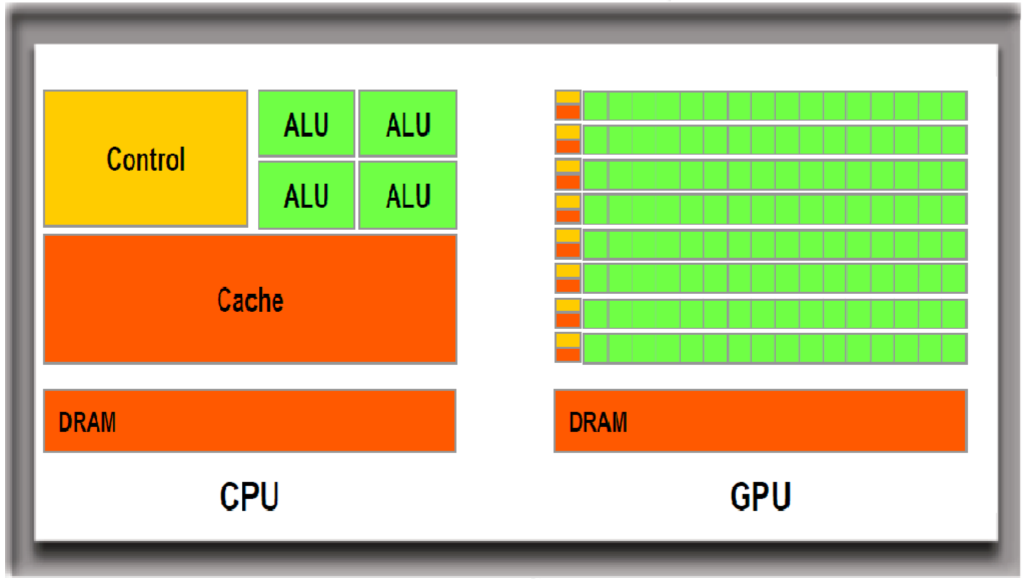


caffe.berkeleyvision.org

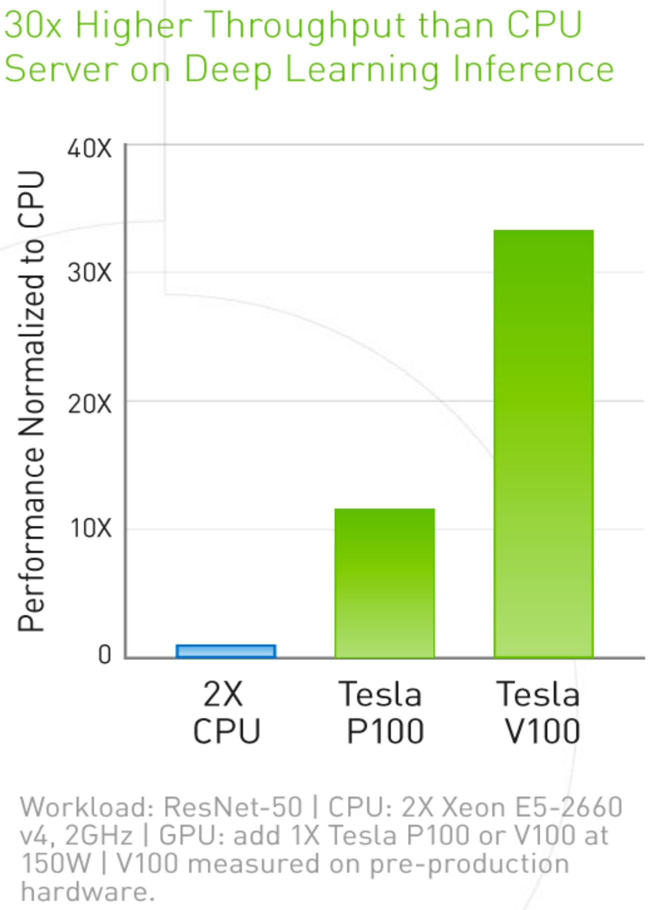
github.com/BVLC/caffe

Deep learning framework

- **CPU vs GPU**



https://www.omnisci.com/technical-glossary/cpu-vs-gpu



https://ph.news.yahoo.com/nvidia-achieves-massive-increase-ai-013811898.html

- **Static vs Dynamic Graphs**

TensorFlow: Build graph once, then run many times **(static)**

```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.Variable(tf.random_normal((D, H)))
w2 = tf.Variable(tf.random_normal((H, D)))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

learning_rate = 1e-5
new_w1 = w1.assign(w1 - learning_rate * grad_w1)
new_w2 = w2.assign(w2 - learning_rate * grad_w2)
updates = tf.group(new_w1, new_w2)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D),}
    losses = []
    for t in range(50):
        loss_val, _ = sess.run([loss, updates],
                               feed_dict=values)
```

Build graph

Run each iteration

PyTorch: Each forward pass defines a new graph **(dynamic)**

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()
```

New graph each iteration

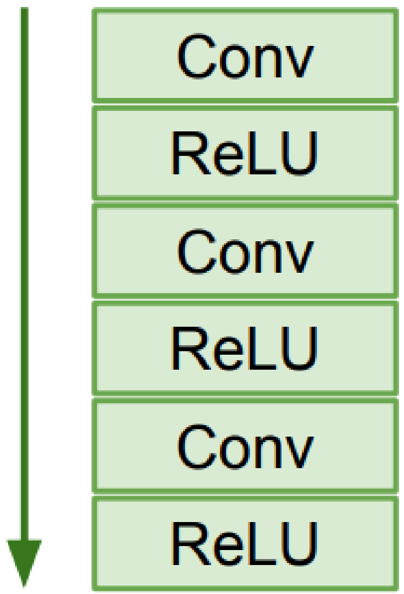
http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture08.pdf

Deep learning framework

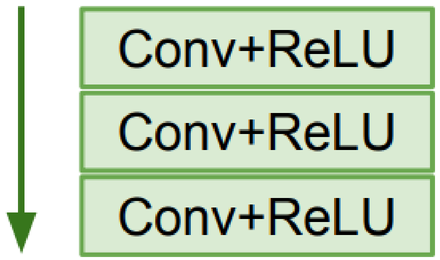
- **<u>Static</u> vs Dynamic: Optimization**

With static graphs, framework can optimize the graph for you before it runs!

The graph you wrote



Equivalent graph with fused operations



http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture08.pdf

- **Static vs Dynamic: Conditional**

$$y = \begin{cases} w1 * x & \text{if } z > 0 \\ w2 * x & \text{otherwise} \end{cases}$$

PyTorch: Normal Python

```
N, D, H = 3, 4, 5

x = torch.randn(N, D, requires_grad=True)
w1 = torch.randn(D, H)
w2 = torch.randn(D, H)

z = 10
if z > 0:
    y = x.mm(w1)
else:
    y = x.mm(w2)
```

TensorFlow: Special TF control flow operator!

```
N, D, H = 3, 4, 5
x = tf.placeholder(tf.float32, shape=(N, D))
z = tf.placeholder(tf.float32, shape=None)
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(D, H))

def f1(): return tf.matmul(x, w1)
def f2(): return tf.matmul(x, w2)
y = tf.cond(tf.less(z, 0), f1, f2)

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        z: 10,
        w1: np.random.randn(D, H),
        w2: np.random.randn(D, H),
    }
    y_val = sess.run(y, feed_dict=values)
```

http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture08.pdf

商汤 sensetime

清華大學 Tsinghua University

Outline

Introduction to Pytorch

- **Advantages of Using Pytorch**

PyTorch

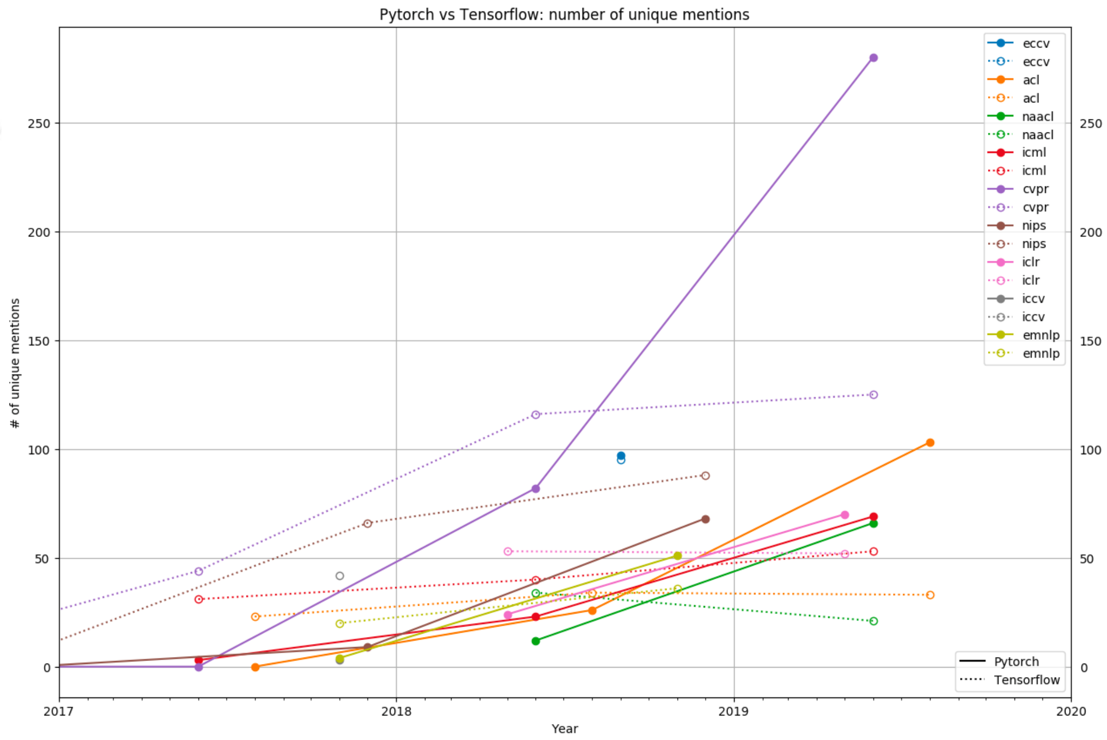
Library functions

Computational efficiency + GPU support

Auto-differentiation

Online community

The most popular framework for academia



http://bangqu.com/AT8859.html

- **Installing PyTorch**

PyTorch Build	Stable (1.7.1)	Preview (Nightly)			
Your OS	Linux	Mac	Windows		
Package	Conda	Pip	LibTorch	Source	
Language	Python	C++ / Java			
CUDA	9.2	10.1	10.2	11.0	None

Run this Command:

```
NOTE: Python 3.9 users will need to add '-c=conda-forge' for installation
conda install pytorch torchvision torchaudio -c pytorch
# MacOS Binaries dont support CUDA, install from source if CUDA is needed
```

https://pytorch.org/

- **Pytorch package**

Package	Description
torch	The top-level PyTorch package and tensor library.
torch.nn	A subpackage that contains modules and extensible classes for building neural networks.
torch.autograd	A subpackage that supports all the differentiable Tensor operations in PyTorch.
torch.nn.functional	A functional interface that contains typical operations used for building neural networks like loss functions, activation functions, and convolution operations.
torch.optim	A subpackage that contains standard optimization operations like SGD and Adam.
torch.utils	A subpackage that contains utility classes like data sets and data loaders that make data preprocessing easier.
torchvision	A package that provides access to popular datasets, model architectures, and image transformations for computer vision.

https://pytorch.org/

- **Pytorch Tensors**

```
import numpy as np
import cudamat as cm
import torch
```

```
n, p = int(2e3), int(40e3)
A = np.random.randn(n, p)
B = np.random.randn(p, n)
%timeit A @ B
```

916 ms ± 45.8 ms per loop
(numpy + cpu)

```
cm.cublas_init()
cm.CUDAMatrix.init_random()
A_cm = cm.empty((n, p)).fill_with_randn()
B_cm = cm.empty((p, n)).fill_with_randn()
%timeit A_cm.dot(B_cm)
cm.cublas_shutdown()
```

26.6 ms ± 721 µs per loop
(numpy+gpu)

```
A=torch.rand(n,p).cuda()
B=torch.rand(p,n).cuda()
%timeit torch.mm(A,B)
```

26.8 ms ± 545 µs per loop
(torch+gpu)

https://transfer.d2.mpi-inf.mpg.de/rshetty/hlcv/Pytorch_tutorial.pdf

- **Pytorch Tensors**

Tensor Initialization

Directly from data

```
data = [[1, 2],[3, 4]]
x_data = torch.tensor(data)
```

From a NumPy array

```
np_array = np.array(data)
x_np = torch.from_numpy(np_array)
```

From another tensor:

```
x_ones = torch.ones_like(x_data) # retains the properties of x_data
print(f"Ones Tensor: \n {x_ones} \n")

x_rand = torch.rand_like(x_data, dtype=torch.float) # overrides the datatype of x_data
print(f"Random Tensor: \n {x_rand} \n")
```

```
Ones Tensor:
 tensor([[1, 1],
        [1, 1]])

Random Tensor:
 tensor([[0.2143, 0.8153],
        [0.5212, 0.8607]])
```

With random or constant values

```
shape = (2,3,)
rand_tensor = torch.rand(shape)
ones_tensor = torch.ones(shape)
zeros_tensor = torch.zeros(shape)
```

https://pytorch.org/tutorials/

- **Tensor Operations**

Moving to GPU

```
if torch.cuda.is_available():
  tensor = tensor.to('cuda')
```

Standard numpy-like indexing and slicing:

```
tensor = torch.ones(4, 4)
tensor[:,1] = 0
print(tensor)
```

```
tensor([[1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.]])
```

Multiplying tensors

```
print(f"tensor.mul(tensor) \n {tensor.mul(tensor)} \n")
# Alternative syntax:
print(f"tensor * tensor \n {tensor * tensor}")
```

```
tensor.mul(tensor)
 tensor([[1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.]])

tensor * tensor
 tensor([[1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.]])
```

In-place operations

```
print(tensor, "\n")
tensor.add_(5)
print(tensor)
```

```
tensor([[1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.]])

tensor([[6., 5., 6., 6.],
        [6., 5., 6., 6.],
        [6., 5., 6., 6.],
        [6., 5., 6., 6.]])
```

https://pytorch.org/tutorials/

- **The NumPy Bridge – Arrays And Tensors**

Tensor to NumPy array

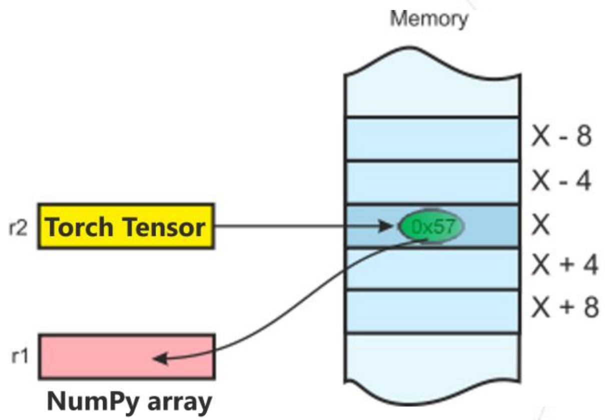
```
t = torch.ones(5)
print(f"t: {t}")
n = t.numpy()
print(f"n: {n}")
```

```
t: tensor([1., 1., 1., 1., 1.])
n: [1. 1. 1. 1. 1.]
```

```
t.add_(1)
print(f"t: {t}")
print(f"n: {n}")
```

```
t: tensor([2., 2., 2., 2., 2.])
n: [2. 2. 2. 2. 2.]
```

share their underlying memory



Tensor to NumPy array

```
n = np.ones(5)
t = torch.from_numpy(n)
```

https://pytorch.org/tutorials/

- **Pytorch Autograd**

```
import torch

a = torch.tensor([2., 3.], requires_grad=True)
b = torch.tensor([6., 4.], requires_grad=True)
```

$$Q = 3a^3 - b^2$$

```
Q = 3*a**3 - b**2
```

$$\frac{\partial Q}{\partial a} = 9a^2$$

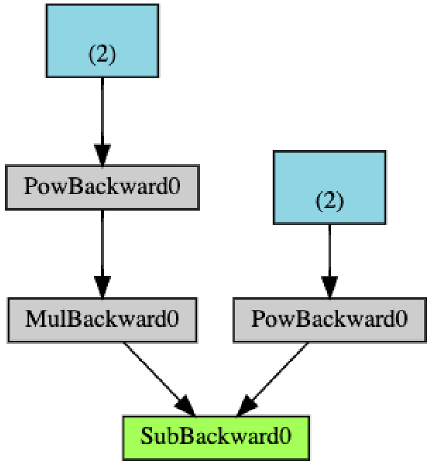
$$\frac{\partial Q}{\partial b} = -2b$$

```
external_grad = torch.tensor([1., 1.])
Q.backward(gradient=external_grad)
```

```
print(9*a**2 == a.grad)
print(-2*b == b.grad)
```

```
tensor([True, True])
tensor([True, True])
```

Computational Graph



Autograd forward pass

Run the requested operation to compute a resulting tensor, and

Maintain the operation’ s gradient function in the directed acyclic graph

Autograd backward pass

Computes the gradients from each .grad_fn

Accumulates them in the respective tensors .grad attribute, and

Using the chain rule, propagates all the way to the leaf tensors.

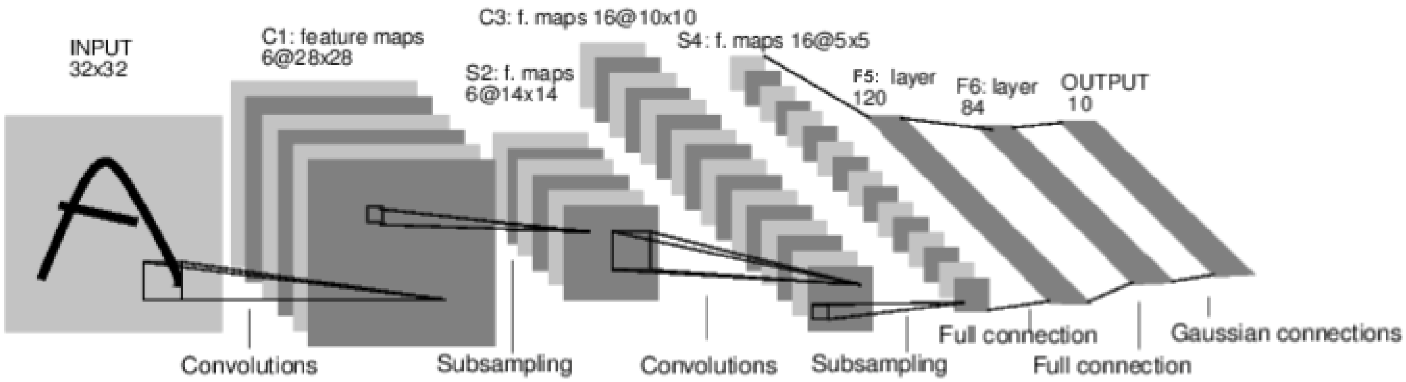
https://pytorch.org/tutorials/

商汤 sensetime

清華大學 Tsinghua University

Outline

- **Overview**



A typical training procedure for a neural network is as follows:

- Define the neural network that has some learnable parameters (or weights)
- Iterate over a dataset of inputs
- Process input through the network
- Compute the loss (how far is the output from being correct)
- Propagate gradients back into the network’ s parameters
- Update the weights of the network, typically using a simple update rule: *weight = weight - learning_rate * gradient*

https://pytorch.org/tutorials/

Use Pytorch to build neural network

- **Define the network**

```python
import torch
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels, 3x3 square convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 3)
        self.conv2 = nn.Conv2d(6, 16, 3)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 6 * 6, 120)  # 6*6 from image dimension
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # If the size is a square you can only specify a single number
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features


net = Net()
print(net)
```

```
Net(
  (conv1): Conv2d(1, 6, kernel_size=(3, 3), stride=(1, 1))
  (conv2): Conv2d(6, 16, kernel_size=(3, 3), stride=(1, 1))
  (fc1): Linear(in_features=576, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
```

```python
input = torch.randn(1, 1, 32, 32)
out = net(input)
print(out)
```

```
tensor([[ 0.1002, -0.0694, -0.0436,  0.0103,  0.0488, -0.0429, -0.0941, -0.0146,
         -0.0031, -0.0923]], grad_fn=<AddmmBackward>)
```

https://pytorch.org/tutorials/

Use Pytorch to build neural network

- ## Define the loss function

```
criterion = nn.MSELoss()

loss = criterion(output, target)
print(loss)
```

```
tensor(0.4969, grad_fn=<MseLossBackward>)
```

- ## Backprop

```
net.zero_grad()     # zeroes the gradient buffers of all parameters

print('conv1.bias.grad before backward')
print(net.conv1.bias.grad)

loss.backward()

print('conv1.bias.grad after backward')
print(net.conv1.bias.grad)
```

```
conv1.bias.grad before backward
tensor([0., 0., 0., 0., 0., 0.])
conv1.bias.grad after backward
tensor([ 0.0111, -0.0064,  0.0053, -0.0047,  0.0026, -0.0153])
```

```
weight = weight - learning_rate * gradient
```

- ## Update the weights-naive

```
learning_rate = 0.01
for f in net.parameters():
    f.data.sub_(f.grad.data * learning_rate)
```

- ## Update the weights-torch.optim

```
import torch.optim as optim

# create your optimizer
optimizer = optim.SGD(net.parameters(), lr=0.01)

# in your training loop:
optimizer.zero_grad()   # zero the gradient buffers
output = net(input)
loss = criterion(output, target)
loss.backward()
optimizer.step()    # Does the update
```

https://pytorch.org/tutorials/

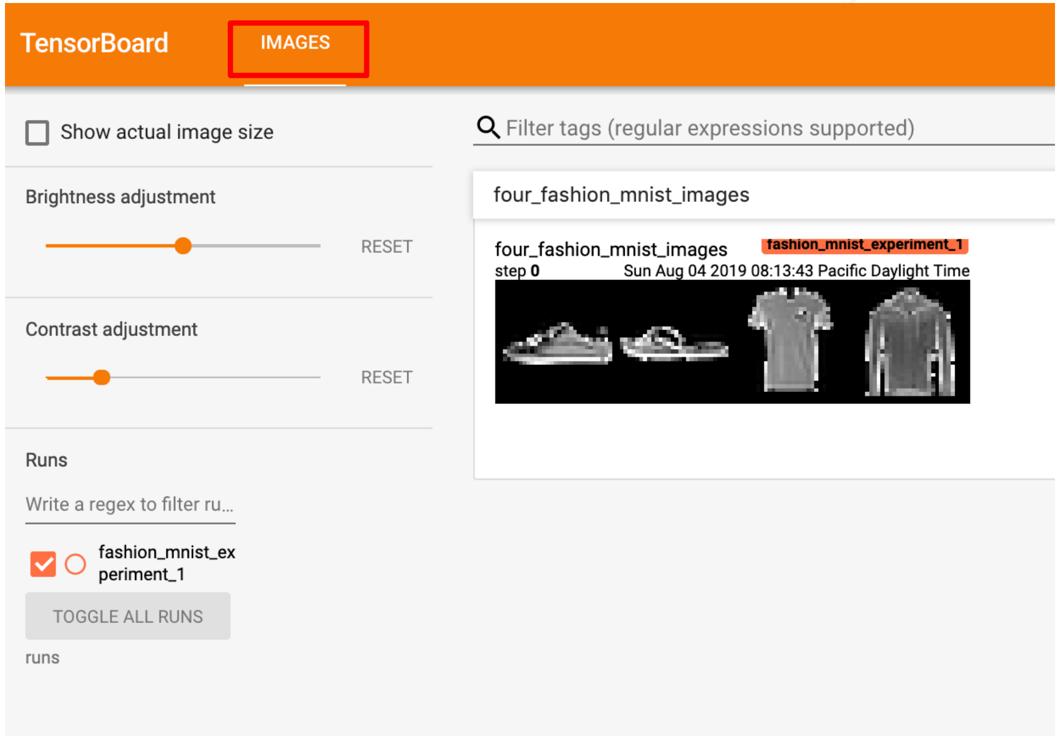
sensetime

Tsinghua University

Outline

Tensorboard and FP16 training

```python
from torch.utils.tensorboard import SummaryWriter
writer = SummaryWriter('runs/fashion_mnist_experiment_1')
```



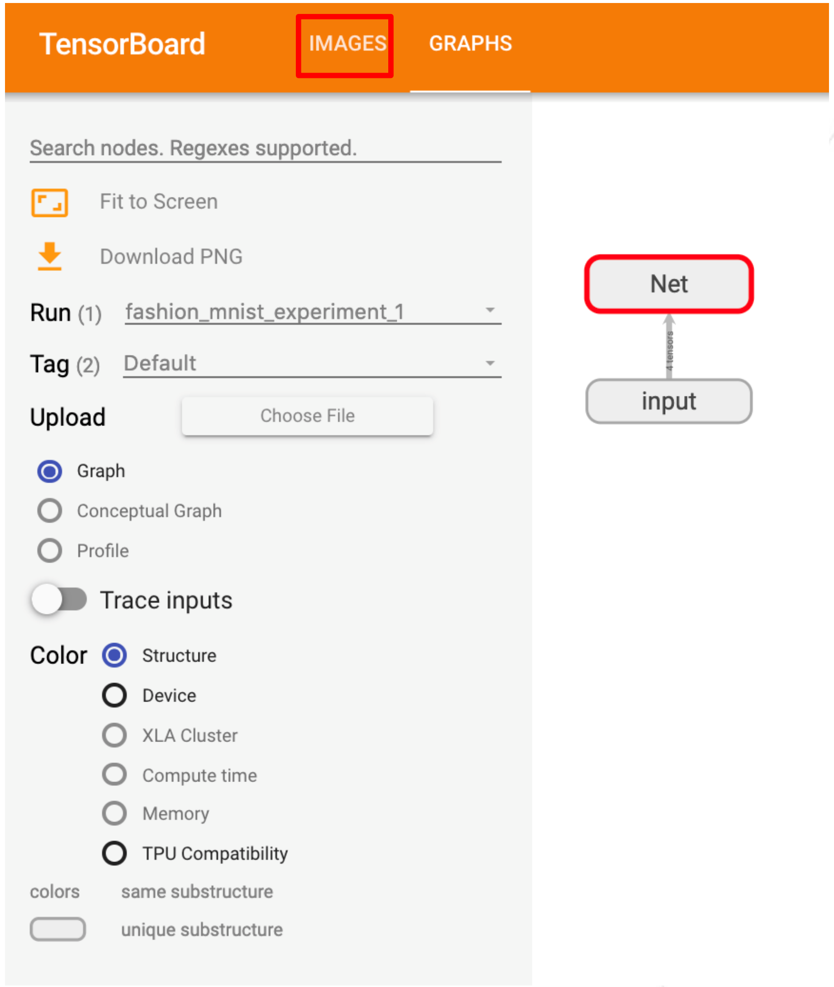

```python
writer.add_image('four_fashion_mnist_images', img_grid)
```

```
tensorboard --logdir=runs
```

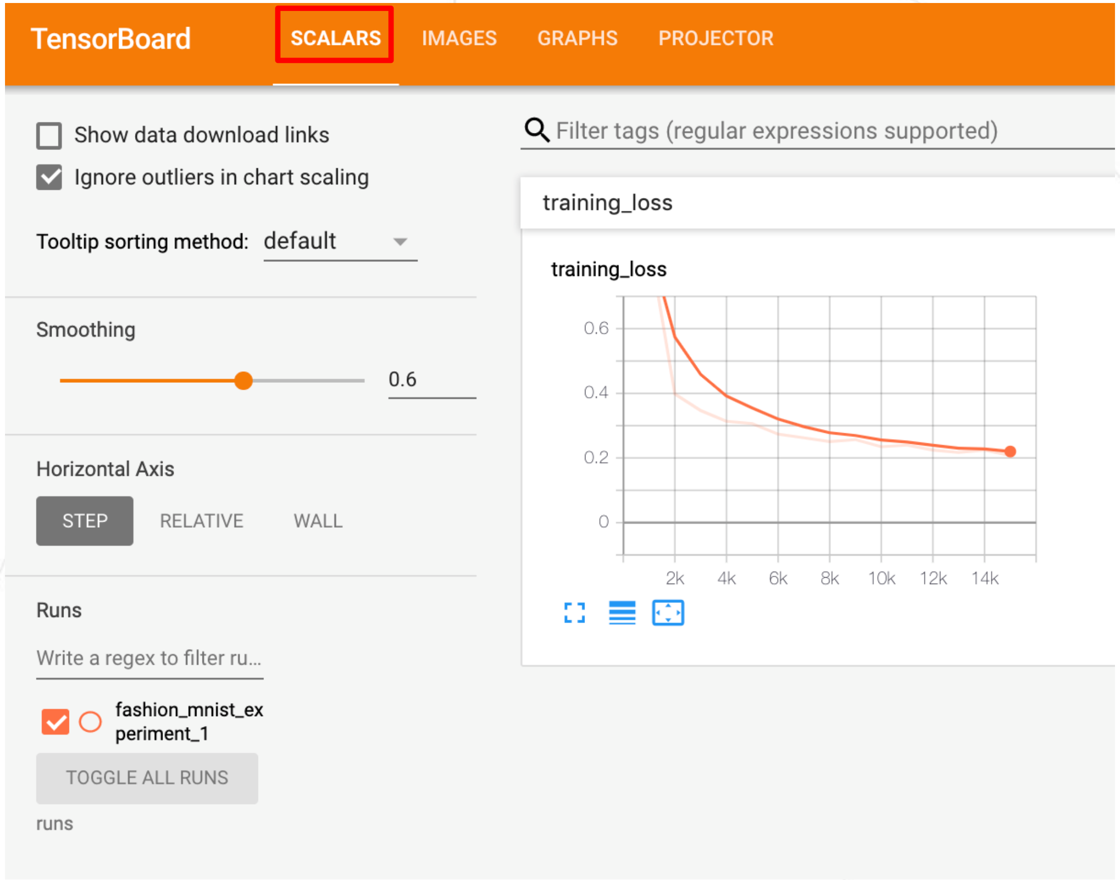

```python
writer.add_embedding(features,
                     metadata=class_labels,
                     label_img=images.unsqueeze(1))
```


https://pytorch.org/tutorials/

Tensorboard and FP16 training




```
writer.add_graph(net, images)
```




```
writer.add_scalar('training loss',
                  running_loss / 1000,
                  epoch * len(trainloader) + i)
```


https://pytorch.org/tutorials/

Tensorboard and FP16 training

- **Mixed-precision training**

combined single-**precision (FP32)** with half-precision **(e.g. FP16)**

- Shorter training time
- Lower memory requirements, enabling larger batch sizes, larger models, or larger inputs.

```python
import torch
# Creates once at the beginning of training
scaler = torch.cuda.amp.GradScaler()

for data, label in data_iter:
   optimizer.zero_grad()
   # Casts operations to mixed precision
   with torch.cuda.amp.autocast():
      loss = model(data)

   # Scales the loss, and calls backward()
   # to create scaled gradients
   scaler.scale(loss).backward()

   # Unscales gradients and calls
   # or skips optimizer.step()
   scaler.step(optimizer)

   # Updates the scale for next iteration
   scaler.update()
```

Training accuracy: NVIDIA DGX A100 (8x A100 40GB)

epochs	Mixed Precision Top 1(%)	TF32 Top1(%)
90	76.93	76.85

FP16 on NVIDIA V100 vs. FP32 on V100

https://pytorch.org/blog/accelerating-training-on-nvidia-gpus-with-pytorch-automatic-mixed-precision/

商汤 sensetime

清華大學 Tsinghua University

Outline

Non-convex optimizer

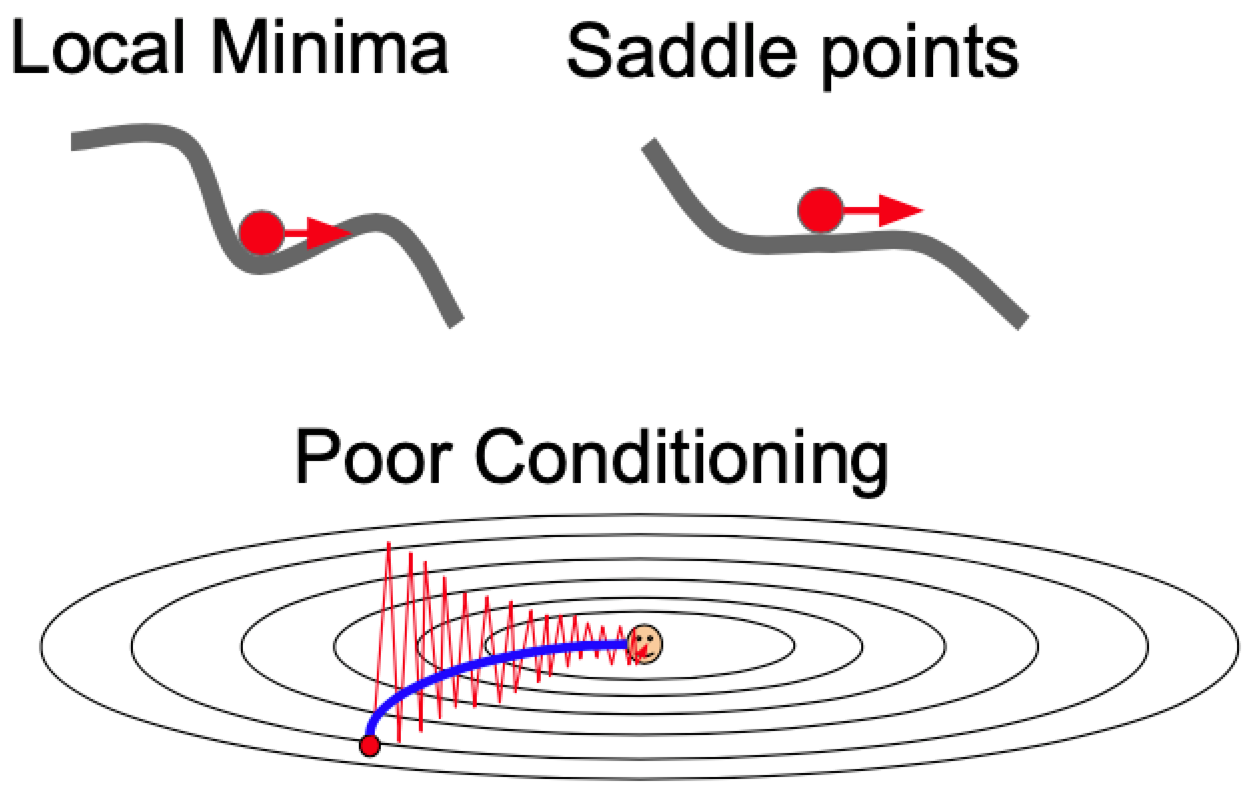
- **Gradient Descent**
  - Batch gradient descent

$$x = x - \eta \nabla_x J(x)$$

  - Stochastic gradient descent

$$x = x - \eta \nabla_x J\left(x; I^{(i:i+n)}; y^{(i:i+n)}\right)$$

- **SGD**
- Considering two cases:
  - the local loss landscape is a smooth hill
    - may take a long time to reach the bottom
  - the local loss landscape is a steep ravine
    - may oscillate back and forth near the bottom



Non-convex optimizer

- **SGD + momentum**

SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
while True:
  dx = compute_gradient(x)
  x -= learning_rate * dx
```

SGD + Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$

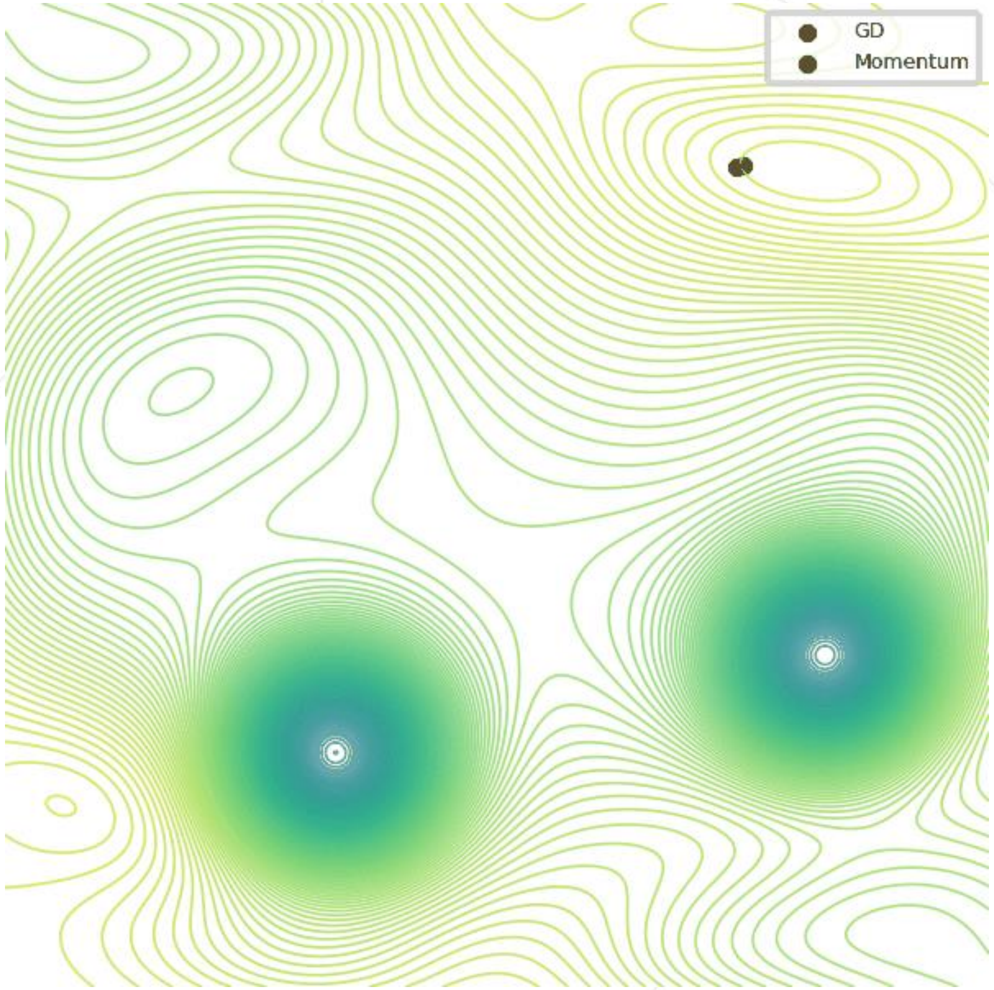
$$x_{t+1} = x_+ - \alpha v_{t+1}$$

```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x -= learning_rate * vx
```

Sutskever I, Martens J, Dahl G, et al. On the importance of initialization and momentum in deep learning. ICML, 2013.

Non-convex optimizer

- **SGD + momentum**
- momentum term
    - increases for dimensions whose gradients point in the same directions
    - reduces updates for dimensions whose gradients change directions



- **SGD + momentum**
  - We'd like momentum term has a notion of where it is going
  - It may slow down when a local minimum is near

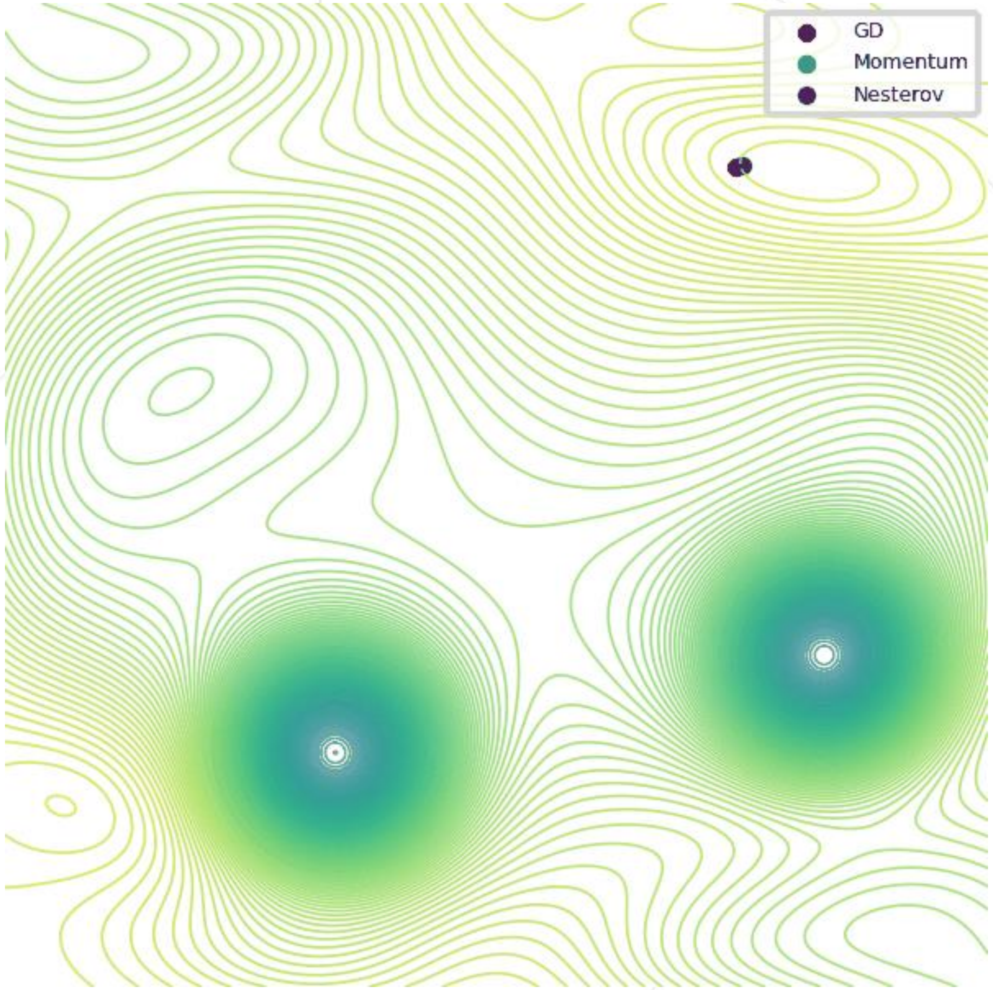
- **Nesterov momentum**

Non-convex optimizer

- **Nesterov momentum**
  - Look ahead by calculating the gradient at the approximate future position of the parameters instead of current ones

$$v_{t+1} = \rho v_t - \alpha \nabla f(\tilde{x}_t)$$

$$\begin{aligned}\tilde{x}_{t+1} &= \tilde{x}_t - \rho v_t + (1+\rho) v_{t+1} \\ &= \tilde{x}_t + v_t + \rho(v_t - v_{t-1})\end{aligned}$$



- **SGD + momentum**
  - Choosing a proper learning rate can be difficult.
  - The same learning rate applies to all parameter updates
- **AdaGrad**

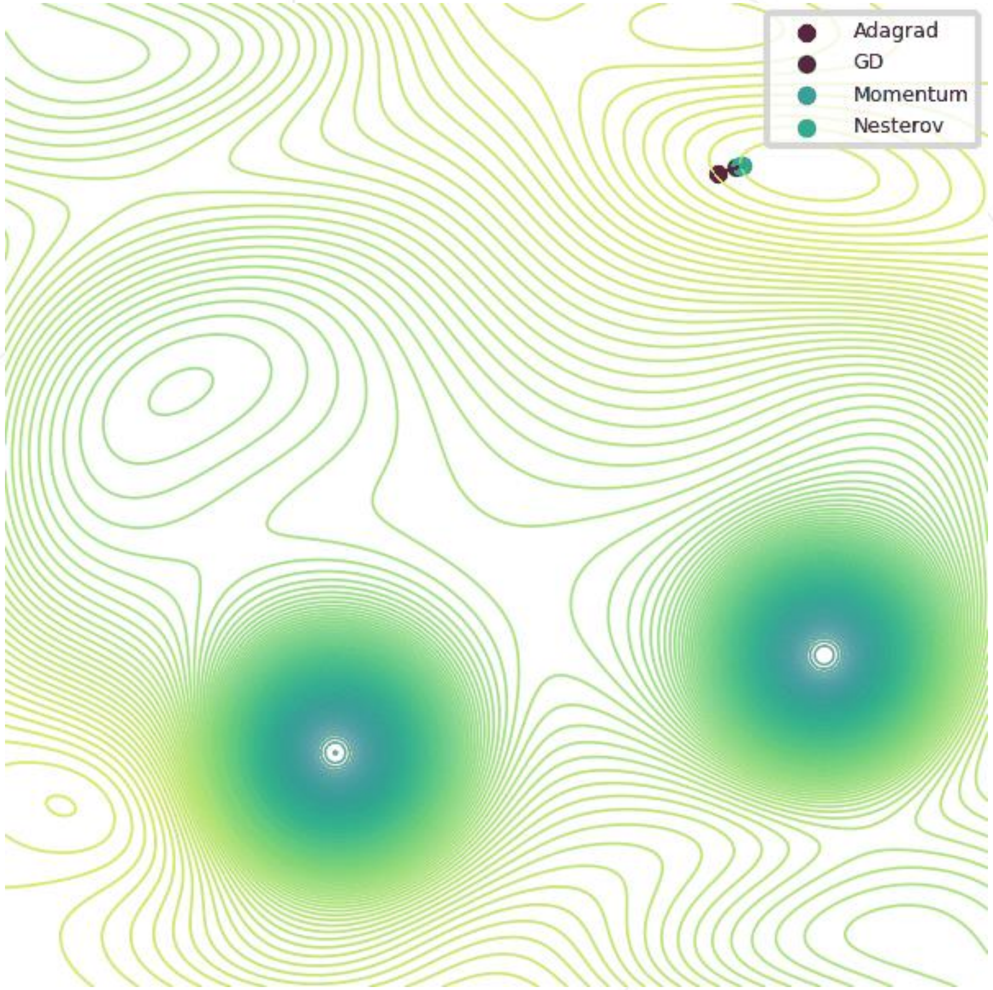
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t.$$

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Duchi J, Hazan E, Singer Y. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. JMLR, 2011.

Non-convex optimizer

- **AdaGrad**
- Gradient-based optimization
  - Smaller learning rates for frequently occurring features
  - Higher learning rates for parameters associated with infrequent features



Duchi J, Hazan E, Singer Y. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. JMLR, 2011.

- **RMSProp**
  - AdaGrad + momentum

AdaGrad

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

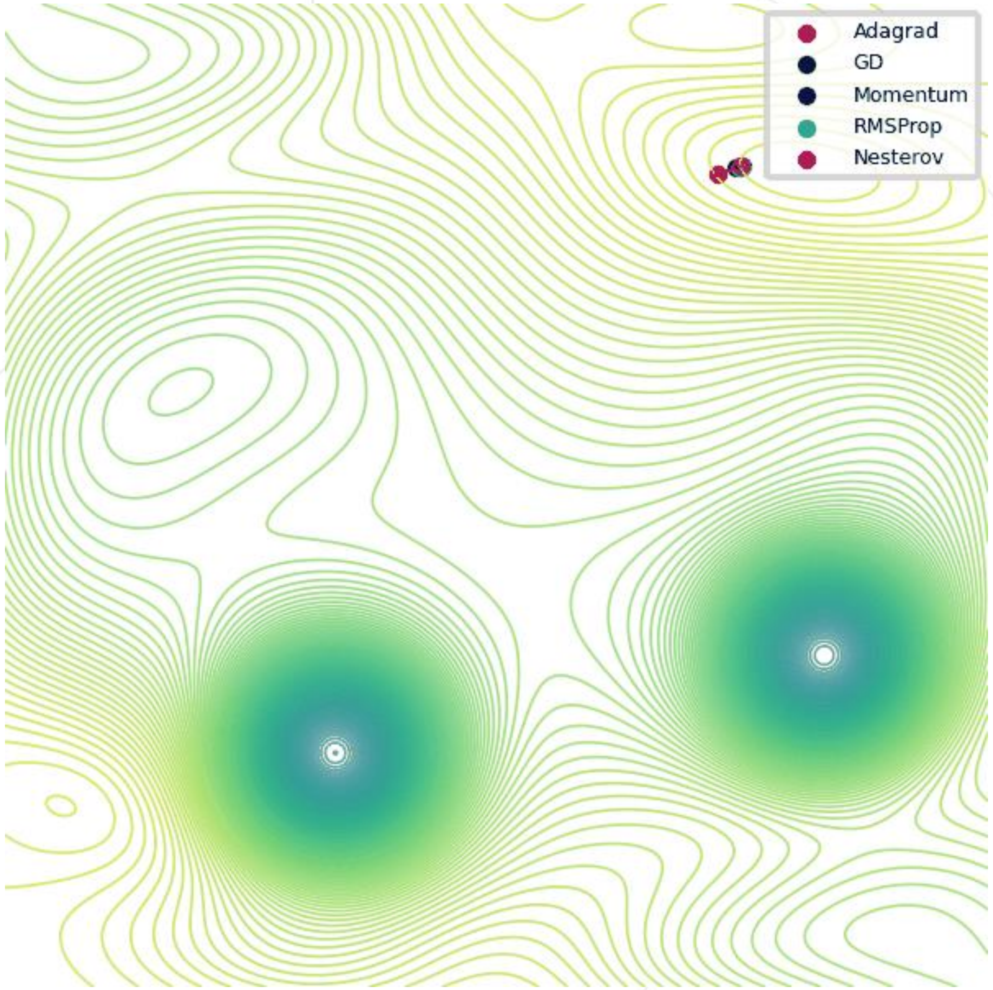
RMSProp

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Tieleman and Hinton, 2012

- **RMSProp**
  - AdaGrad + momentum

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$



Tieleman and Hinton, 2012

Non-convex optimizer

- **Adam**
  - momentum: a ball running down a slope
  - Adam: a heavy ball with friction
    - prefers flat minima in the error surface

```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  first_unbias = first_moment / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)
  x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Momentum

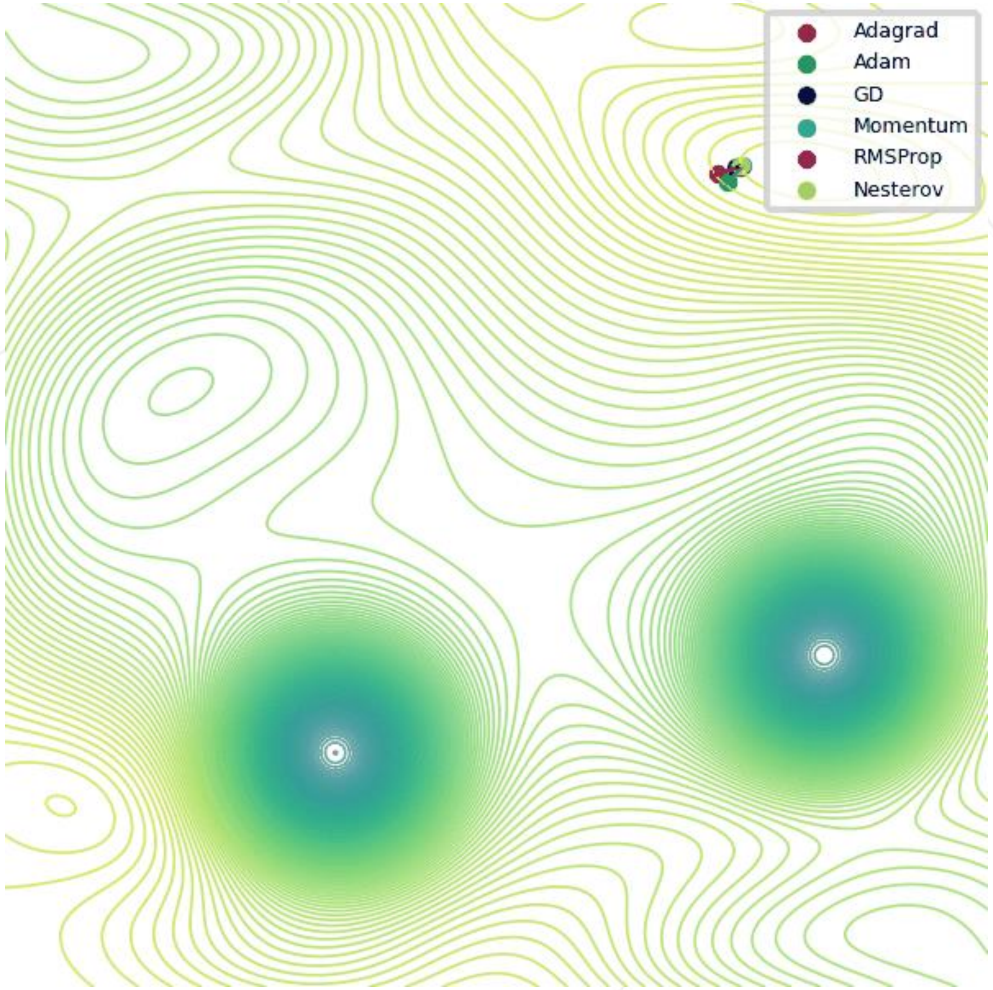
Bias correction

AdaGrad / RMSProp

Kingma D, Ba J. Adam: A Method for Stochastic Optimization. ICLR, 2015.

- **Adam**
  - momentum: a ball running down a slope
  - Adam: a heavy ball with friction
    - prefers flat minima in the error surface

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t.$$



Kingma D, Ba J. Adam: A Method for Stochastic Optimization. ICLR, 2015.

Non-convex optimizer

- **Adam + Weight decay**
  - Weight decay is a technique in training neural networks

$$g_t = \nabla f(\theta_t) + w_t \theta_t,$$

- **AdamW**
  - adjusts the weight decay term to appear in the gradient update
  - small details can make a noticeable difference

$$\theta_{t+1,i} = \theta_{t,i} - \eta \left(\frac{1}{\sqrt{\hat{\mathcal{E}}^{g \circ g}_{t+1,i} + \varepsilon}} \cdot \mathcal{E}^{g}_{t+1,i} + w_{t,i} \theta_{t,i} \right),$$

Kingma D, Ba J. Adam: A Method for Stochastic Optimization. ICLR, 2015.

Non-convex optimizer

- **Which one to choose?**
- Case 1: Little budge to hyperparameter tuning?

  Adam, AdamW, Demon Adam ...
- Case 2: Best performance?

  SGD + Momentum