PyTorch Tutorial

Willie Chang
Pranay Manocha

Installing PyTorch

- 💻 On your own computer
  - Anaconda/Miniconda: `conda install pytorch -c pytorch`
  - Others via pip: `pip3 install torch`

- 🌐 On Princeton CS server (`ssh cycles.cs.princeton.edu`)
  - Non-CS students can request a *class account*.
  - Miniconda is highly recommended, because:
    - It lets you manage your own Python installation
    - It installs locally; no admin privileges required
    - It's lightweight and fits within your disk quota
  - Instructions:
    - `wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh`
    - `chmod u+x ./Miniconda3-latest-Linux-x86_64.sh`
    - `./Miniconda3-latest-Linux-x86_64.sh`
    - After Miniconda is installed: `conda install pytorch -c pytorch`

Writing code

- Up to you; feel free to use emacs, vim, PyCharm, etc. if you want.
- Our recommendations:

Jupyter Notebook

Also try Jupyter Lab!

- Install: `conda/pip3 install jupyter`
- 💻 Run on your computer
  - `jupyter notebook`
- 🌐 Run on Princeton CS server
  - Pick any 4-digit number, say 1234
  - 🌐 `hostname -s`
  - 🌐 `jupyter notebook --no-browser --port=1234`
  - 💻 `ssh -N -L 1234:localhost:1234 __@__.cs.princeton.edu`
  - First blank is username, second is hostname

VS Code

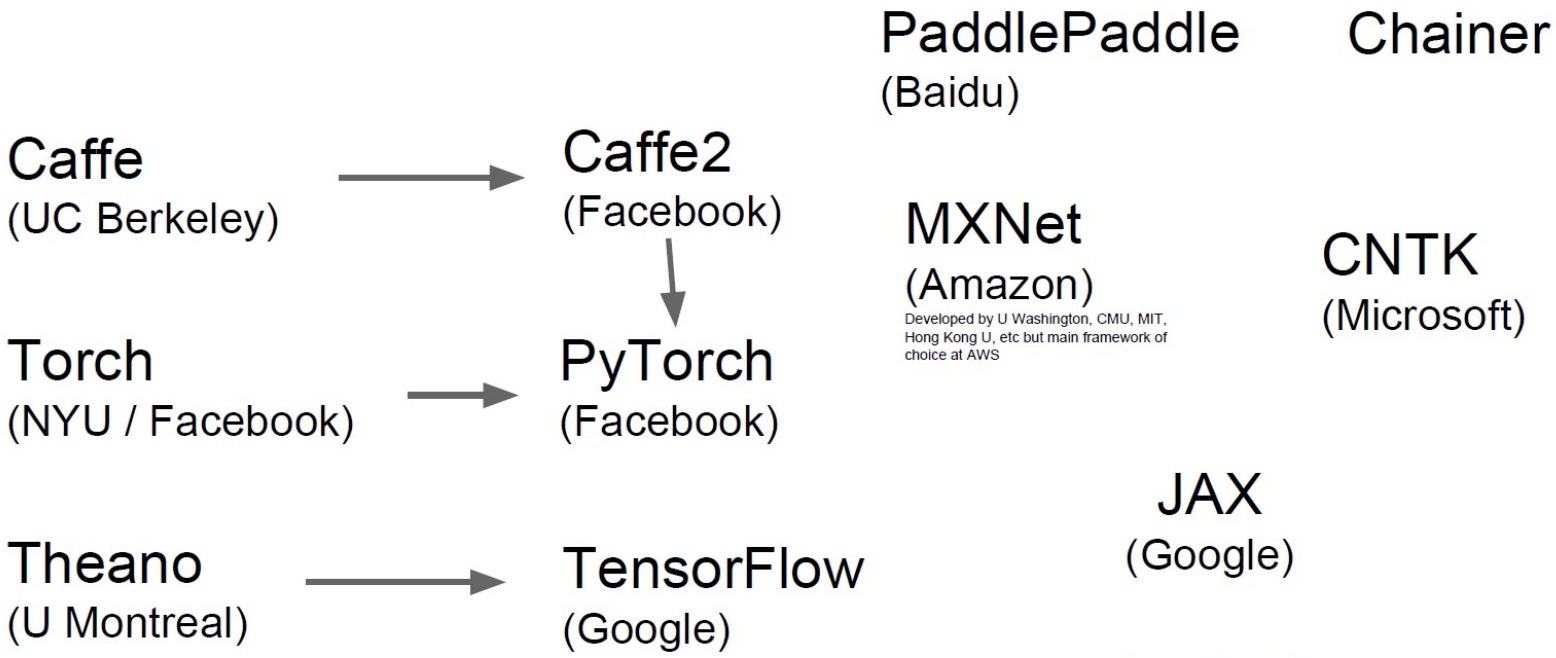
- Install the *Python* extension.
- 🌐 Install the *Remote Development* extension.
- Python files can be run like Jupyter notebooks by delimiting cells/sections with #%%
- Debugging PyTorch code is just like debugging any other Python code: see Piazza @108 for info.

Why talk about libraries?

- Advantage of various deep learning frameworks
  - Quick to develop and test new ideas
  - Automatically compute gradients
  - Run it all efficiently on GPU to speed up computation

Various Frameworks

- Various Deep Learning Frameworks



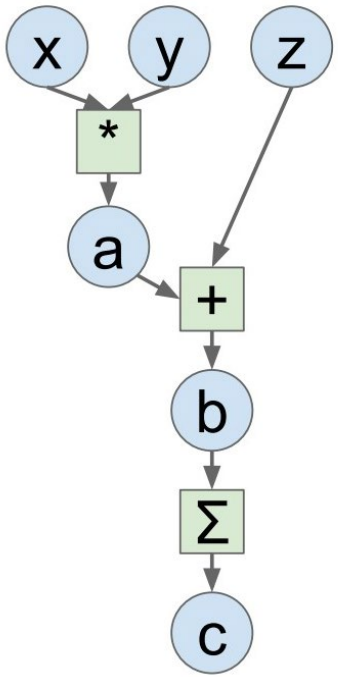
- Focus on PyTorch in this session.

Source: CS231n slides

Preview: (and advantages)

- Preview of Numpy & PyTorch & Tensorflow

Computation Graph



Numpy

```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```

Tensorflow

```
import numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3, 4

with tf.device('/gpu:0'):
    x = tf.placeholder(tf.float32)
    y = tf.placeholder(tf.float32)
    z = tf.placeholder(tf.float32)

    a = x * y
    b = a + z
    c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                   feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

PyTorch

```
import torch

N, D = 3, 4

x = torch.rand((N, D),requires_grad=True)
y = torch.rand((N, D),requires_grad=True)
z = torch.rand((N, D),requires_grad=True)

a =x * y
b =a + z
c=torch.sum(b)

c.backward()
```

Advantages (continued)

- Which one do you think is better?

Advantages (continued)

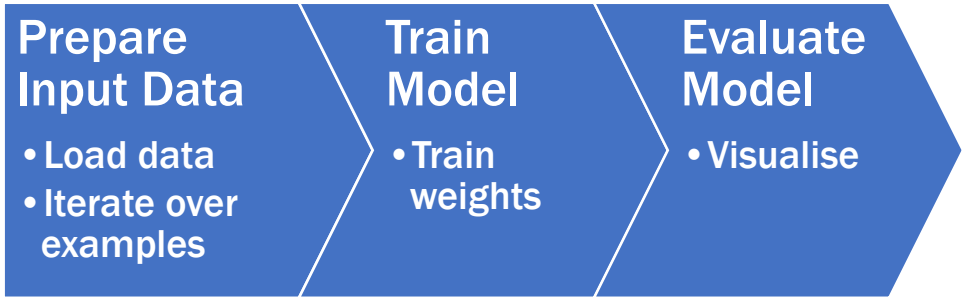
- Which one do you think is better?

PyTorch!

- **Easy Interface** – easy to use API. The code execution in this framework is quite easy. Also need a fewer lines to code in comparison.
  - It is easy to debug and understand the code.
- **Python usage** – This library is considered to be Pythonic which smoothly integrates with the Python data science stack.
  - It can be considered as NumPy extension to GPUs.
- **Computational graphs** – PyTorch provides an excellent platform which offers dynamic computational graphs. Thus a user can change them during runtime.
  - It includes many layers as Torch.
  - It includes lot of loss functions.
  - It allows building networks whose structure is dependent on computation itself.
  - **NLP**: account for variable length sentences. Instead of padding the sentence to a fixed length, we create graphs with different number of LSTM cells based on the sentence's length.

PyTorch

- Fundamental Concepts of PyTorch
  - Tensors
  - Autograd
  - Modular structure
    - Models / Layers
    - Datasets
    - Dataloader
  - Visualization Tools like
    - *TensorboardX* (monitor training)
    - *PyTorchViz* (visualise computation graph)
  - Various other functions
    - loss (MSE,CE etc..)
    - optimizers



Tensor

- **Tensor?**
- PyTorch Tensors are just like numpy arrays, but they can run on GPU.
- Examples:

```python
import numpy
# create a tensor
new_numpy = numpy.array([[1, 2], [3, 4]])
# create a 2 x 3 tensor with random values
empty_numpy = numpy.random.rand(2,3)
# create a 2 x 3 tensor with random values between -1 and 1
uniform_numpy = numpy.reshape(np.random.uniform(-1,1,6),[2,3])
# create a 2 x 3 tensor with random values from a uniform distribution on the interval [0, 1)
rand_numpy = numpy.random.rand(2, 3)
# create a 2 x 3 tensor of zeros
zero_numpy = numpy.zeros([2, 3])
```

```python
import torch
# create a tensor
new_tensor = torch.Tensor([[1, 2], [3, 4]])
# create a 2 x 3 tensor with random values
empty_tensor = torch.Tensor(2, 3)
# create a 2 x 3 tensor with random values between -1and 1
uniform_tensor = torch.Tensor(2, 3).uniform_(-1, 1)
# create a 2 x 3 tensor with random values from a uniform distribution on the interval [0, 1)
rand_tensor = torch.rand(2, 3)
# create a 2 x 3 tensor of zeros
zero_tensor = torch.zeros(2, 3)
```

And more operations like:

Indexing, slicing, reshape, transpose, cross product,
matrix product, element wise multiplication etc...

Tensor (continued)

- Attributes of a tensor *'t'*:
  - `t= torch.randn(1)`
- *`requires_grad` - making a trainable parameter*
  - *By default **False***
  - Turn on:
    - `t.requires_grad_()` or
    - `t= torch.randn(1,requires_grad=True)`
  - Accessing tensor value:
    - t.data
  - Accessing tensor gradient
    - t.grad
- *`grad_fn` – history of operations for autograd*
  - *t.grad_fn*

```python
import torch

N, D = 3, 4

x = torch.rand((N, D),requires_grad=True)
y = torch.rand((N, D),requires_grad=True)
z = torch.rand((N, D),requires_grad=True)

a = x * y
b = a + z
c=torch.sum(b)

c.backward()

print(c.grad_fn)
print(x.data)
print(x.grad)
```

```
<SumBackward0 object at 0x7fd0cb970cc0>
tensor([[0.4118, 0.2576, 0.3470, 0.0240],
        [0.7797, 0.1519, 0.7513, 0.7269],
        [0.8572, 0.1165, 0.8596, 0.2636]])
tensor([[0.6855, 0.9696, 0.4295, 0.4961],
        [0.3849, 0.0825, 0.7400, 0.0036],
        [0.8104, 0.8741, 0.9729, 0.3821]])
```

Loading Data, Devices and CUDA

- *Numpy arrays to PyTorch tensors*
  - **`torch.from_numpy(x_train)`**
  - **`Returns a cpu tensor!`**
- *PyTorch tensor to numpy*
  - **`t.numpy()`**
- *Using GPU acceleration*
  - **`t.to()`**
  - *Sends to whatever device (cuda or cpu)*
- *Fallback to cpu if gpu is unavailable:*
  - **`torch.cuda.is_available()`**
- Check cpu/gpu tensor OR numpy array ?
  - **`type(t)`** or **`t.type()`**
  - returns
    - numpy.ndarray
    - torch.Tensor
      - CPU - torch.cpu.FloatTensor
      - GPU - torch.cuda.FloatTensor

```
1  import torch
2  import torch.optim as optim
3  import torch.nn as nn
4  from torchviz import make_dot
5
6  device = 'cuda' if torch.cuda.is_available() else 'cpu'
7
8  # Our data was in Numpy arrays, but we need to transform them into PyTorch's Tensors
9  # and then we send them to the chosen device
10 x_train_tensor = torch.from_numpy(x_train).float().to(device)
11 y_train_tensor = torch.from_numpy(y_train).float().to(device)
12
13 # Here we can see the difference - notice that .type() is more useful
14 # since it also tells us WHERE the tensor is (device)
15 print(type(x_train), type(x_train_tensor), x_train_tensor.type())
```

*Assume *'t' is a tensor*

Autograd

- Autograd
  - *Automatic Differentiation Package*
  - *Don't need to worry about partial differentiation, chain rule etc..*
- backward() does that
  - `loss.backward()`
- Gradients are accumulated for each step by default:
  - Need to zero out gradients after each update
  - `t.grad.zero_()`

```
import torch

N, D = 3, 4

x = torch.rand((N, D),requires_grad=True)
y = torch.rand((N, D),requires_grad=True)
z = torch.rand((N, D),requires_grad=True)

a =x * y
b =a + z
c=torch.sum(b)

c.backward()
```

*Assume *'t' is a tensor*

Autograd (continued)

- Manual Weight Update - example

```
a = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)
b = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)

for epoch in range(n_epochs):
    yhat = a + b * x_train_tensor
    error = y_train_tensor - yhat
    loss = (error ** 2).mean()

    loss.backward()

    with torch.no_grad():
        a -= lr * a.grad
        b -= lr * b.grad

    a.grad.zero_()
    b.grad.zero_()

print(a, b)
```

Optimizer

- Optimizers (*optim* package)
  - Adam, Adagrad, Adadelta, SGD etc..
  - Manually updating is ok if small number of weights
    - Imagine updating 100k parameters!
  - An optimizer takes the **parameters** we want to update, the **learning rate** we want to use (and possibly many other hyper-parameters as well!)
    and **performs the updates**

```
a = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)
b = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)


# Defines a SGD optimizer to update the parameters
optimizer = optim.SGD([a, b], lr=lr)

for epoch in range(n_epochs):
    yhat = a + b * x_train_tensor
    error = y_train_tensor - yhat
    loss = (error ** 2).mean()

    loss.backward()

    optimizer.step()

    optimizer.zero_grad()

print(a, b)
```

Loss

- Loss
  - Various predefined loss functions to choose from
  - L1, MSE, Cross Entropy ……

```
a = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)
b = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)

# Defines a MSE loss function
loss_fn = nn.MSELoss(reduction='mean')

optimizer = optim.SGD([a, b], lr=lr)

for epoch in range(n_epochs):
    yhat = a + b * x_train_tensor

    loss = loss_fn(y_train_tensor, yhat)

    loss.backward()
    optimizer.step()
    optimizer.zero_grad()

print(a, b)
```

Model

- In PyTorch, a **model** is represented by a regular **Python class** that inherits from the Module class.
  - Two components
    - `__init__(self)`: **it defines the parts that make up the model** —in our case, two *parameters*, ***a*** and ***b***
    - `forward(self, x)`: it performs the **actual computation**, that is, it **outputs a prediction**, given the input **x**

Model (example)

- Example:

```
class ManualLinearRegression(nn.Module):
    def __init__(self):
        super().__init__()
        # To make "a" and "b" real parameters of the model, we need to wrap them with nn.Parameter
        self.a = nn.Parameter(torch.randn(1, requires_grad=True, dtype=torch.float))
        self.b = nn.Parameter(torch.randn(1, requires_grad=True, dtype=torch.float))

    def forward(self, x):
        # Computes the outputs / predictions
        return self.a + self.b * x
```

- Properties:
  - model = ManualLinearRegression()
    - model.state_dic() - returns a dictionary of trainable parameters with their current values
    - model.parameters() - returns a list of all trainable parameters in the model
    - model.train() or model.eval()

Putting things together

- Sample Code in practice

```
# Now we can create a model and send it at once to the device
model = ManualLinearRegression().to(device)
# We can also inspect its parameters using its state_dict
print(model.state_dict())

loss_fn = nn.MSELoss(reduction='mean')
optimizer = optim.SGD(model.parameters(), lr=lr)

for epoch in range(n_epochs):
    model.train()

    yhat = model(x_train_tensor)

    loss = loss_fn(y_train_tensor, yhat)
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()

print(model.state_dict())
```

Complex Models

- Complex Model Class
- Predefined 'layer' modules

```
class LayerLinearRegression(nn.Module):
    def __init__(self):
        super().__init__()
        # Instead of our custom parameters, we use a Linear layer with single input and single output
        self.linear = nn.Linear(1, 1)

    def forward(self, x):
        # Now it only takes a call to the layer to make predictions
        return self.linear(x)
```

- 'Sequential' layer modules

```
class LayerLinearRegression(nn.Module):
    def __init__(self):
        super().__init__()
        # Instead of our custom parameters, we use a Linear layer with single input and single output
        self.seq_linear = nn.Sequential(nn.Linear(1, 2),nn.RELU(),nn.Linear(2,1))

    def forward(self, x):
        # Now it only takes a call to the layer to make predictions
        return self.seq_linear(x)
```

Dataset

- Dataset
  - In PyTorch, a **dataset** is represented by a regular **Python class** that inherits from the Dataset class. You can think of it as a kind of a Python **list of tuples**, each tuple corresponding to **one point (features, label)**
  - **3 components:**
    - __init__(self)
    - __get_item__(self, index)
    - __len__(self)
  - **Unless the dataset is huge (cannot fit in memory), you don't explictly need to define this class. Use `TensorDataset`**

```
from torch.utils.data import Dataset, TensorDataset

class CustomDataset(Dataset):
    def __init__(self, x_tensor, y_tensor):
        self.x = x_tensor
        self.y = y_tensor

    def __getitem__(self, index):
        return (self.x[index], self.y[index])

    def __len__(self):
        return len(self.x)

x_train_tensor = torch.from_numpy(x_train).float()
y_train_tensor = torch.from_numpy(y_train).float()

train_data = CustomDataset(x_train_tensor, y_train_tensor)
print(train_data[0])

train_data = TensorDataset(x_train_tensor, y_train_tensor)
print(train_data[0])
```

Dataloader

- Dataloader
  - What happens if we have a huge dataset? Have to train in 'batches'
  - Use PyTorch's Dataloader class!
    - We tell it which **dataset** to use, the desired **mini-batch size** and if we'd like to **shuffle** it or not. That's it!
    - Our **loader** will behave like an **iterator**, so we can **loop over it** and **fetch a different mini-batch** every time.

```
from torch.utils.data import DataLoader

train_loader = DataLoader(dataset=train_data, batch_size=16, shuffle=True)
```

Dataloader (example)

- Sample Code in Practice:

```
losses = []

model = ManualLinearRegression().to(device)

loss_fn = nn.MSELoss(reduction='mean')
optimizer = optim.SGD(model.parameters(), lr=lr)

for epoch in range(n_epochs):
    for x_batch, y_batch in train_loader:
        model.train()

        x_batch = x_batch.to(device)
        y_batch = y_batch.to(device)
        yhat = model(x_train_tensor)

        loss = loss_fn(y_batch, yhat)
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

        losses.append(loss)

print(model.state_dict())
```

Split Data

- Random Split for Train, Val and Test Set
- [random_split()]

```
from torch.utils.data.dataset import random_split

x_tensor = torch.from_numpy(x).float()
y_tensor = torch.from_numpy(y).float()

dataset = TensorDataset(x_tensor, y_tensor)

train_dataset, val_dataset, test_dataset = random_split(dataset, [60, 20, 20])

train_loader = DataLoader(dataset=train_dataset, batch_size=16)
val_loader = DataLoader(dataset=val_dataset, batch_size=20)
test_loader = DataLoader(dataset=test_dataset, batch_size=20)
```

Saving / Loading Weights

Method 1

- Only inference/evaluation – save only state_dict
- Save:
  - torch.save(model.state_dict(), PATH)
- Load:
  - model = TheModelClass(*args, **kwargs)
  - model.load_state_dict(torch.load(PATH))
  - model.eval()

- CONVENTION IS TO SAVE MODELS USING EITHER A .PT OR A .PTH EXTENSION

https://pytorch.org/tutorials/beginner/saving_loading_models.html

Saving / Loading Weights (continued)

- Method 2
- Checkpoint – resume training / inference
  - Save:
    - torch.save({
      'epoch': epoch,
      'model_state_dict': model.state_dict(),
      'optimizer_state_dict': optimizer.state_dict(),
      'loss': loss,
      ...
      }, PATH)
  - Load:
    - model = TheModelClass(*args, **kwargs)
      optimizer = TheOptimizerClass(*args, **kwargs)

      checkpoint = torch.load(PATH)
      model.load_state_dict(checkpoint['model_state_dict'])
      optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
      epoch = checkpoint['epoch']
      loss = checkpoint['loss']

      model.eval()
      # - or -
      model.train()

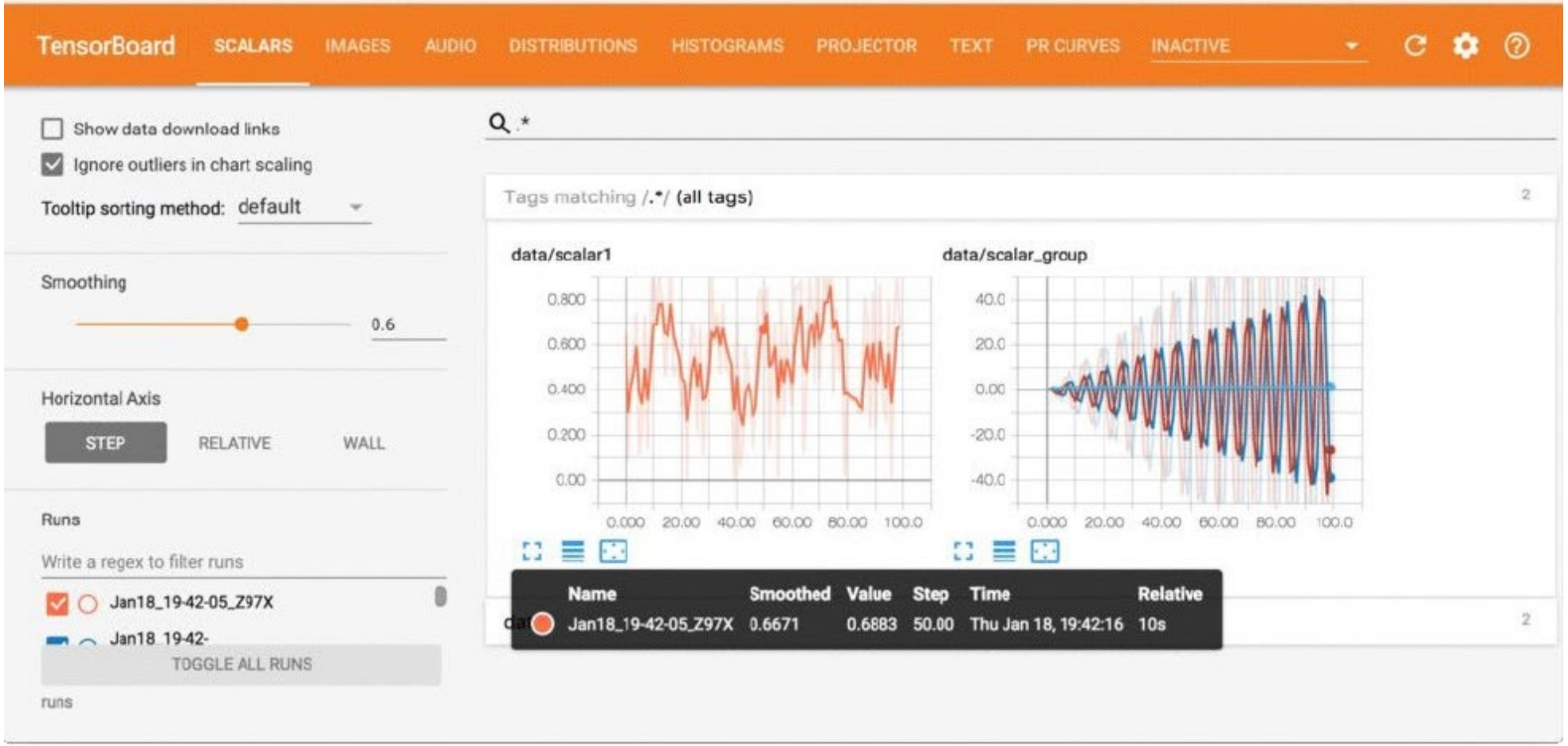
Evaluation

- Two important things:
  - torch.no_grad()
    - Don’t store the history of all computations
  - eval()
    - Tell compiler which mode to run on.

```
losses = []
val_losses = []
model = ManualLinearRegression().to(device)
loss_fn = nn.MSELoss(reduction='mean')
optimizer = optim.SGD(model.parameters(), lr=lr)

for epoch in range(n_epochs):
    for x_batch, y_batch in train_loader:
        model.train()

        x_batch = x_batch.to(device)
        y_batch = y_batch.to(device)
        yhat = model(x_train_tensor)

        loss = loss_fn(y_batch, yhat)
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

        losses.append(loss)

    with torch.no_grad():
        for x_val, y_val in val_loader:
            x_val = x_val.to(device)
            y_val = y_val.to(device)

            model.eval()

            yhat = model(x_val)
            val_loss = loss_fn(y_val, yhat)
            val_losses.append(val_loss.item())
```

Visualization

- TensorboardX (visualise training)
- PyTorchViz (visualise computation graph)

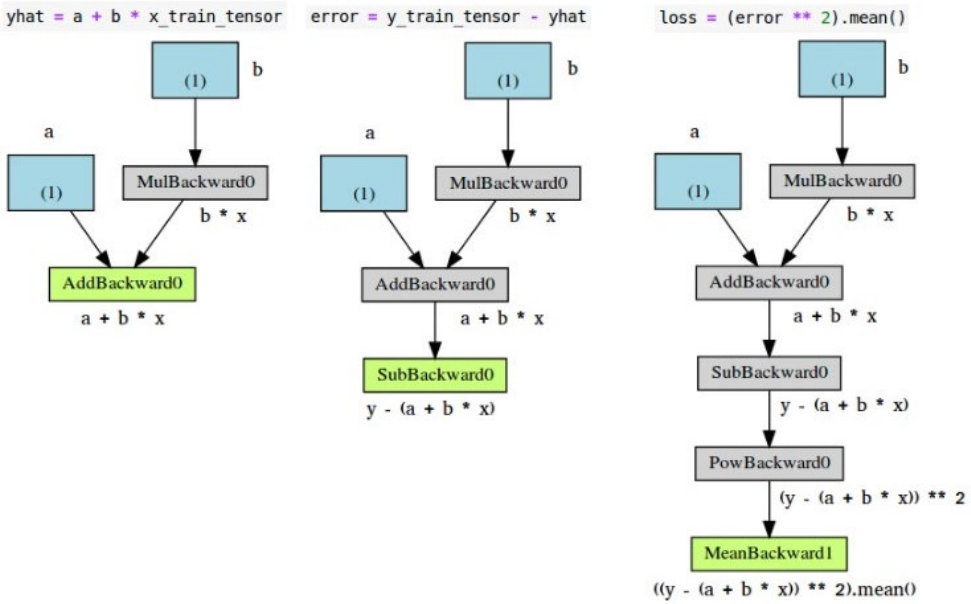


https://github.com/lanpa/tensorboardX/

Visualization (continued)

- PyTorchViz

```
a = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)
b = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)


# Defines a SGD optimizer to update the parameters
optimizer = optim.SGD([a, b], lr=lr)

for epoch in range(n_epochs):
    yhat = a + b * x_train_tensor
    error = y_train_tensor - yhat
    loss = (error ** 2).mean()

    loss.backward()

    optimizer.step()

    optimizer.zero_grad()

print(a, b)
```



https://github.com/szagoruyko/pytorchviz

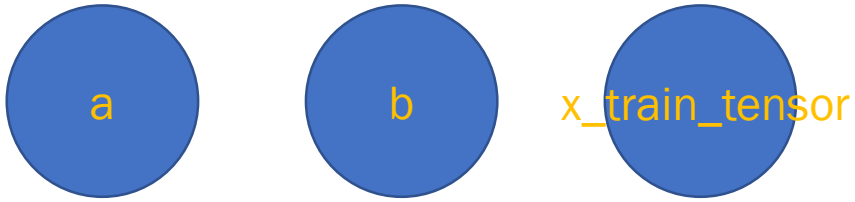
References

- Important References:
  - For setting up jupyter notebook on princeton ionic cluster
    - https://oncomputingwell.princeton.edu/2018/05/jupyter-on-the-cluster/
  - Best reference is PyTorch Documentation
    - https://pytorch.org/ and https://github.com/pytorch/pytorch
  - Good Blogs: (with examples and code)
    - https://lelon.io/blog/2018/02/08/pytorch-with-baby-steps
    - https://www.tutorialspoint.com/pytorch/index.htm
    - https://github.com/hunkim/PyTorchZeroToAll
  - Free GPU access for short time:
    - Google Colab provides free Tesla K80 **GPU** of about 12GB. You can run the session in an interactive **Colab** Notebook for 12 hours.
    - https://colab.research.google.com/

Misc

- Dynamic VS Static Computation Graph

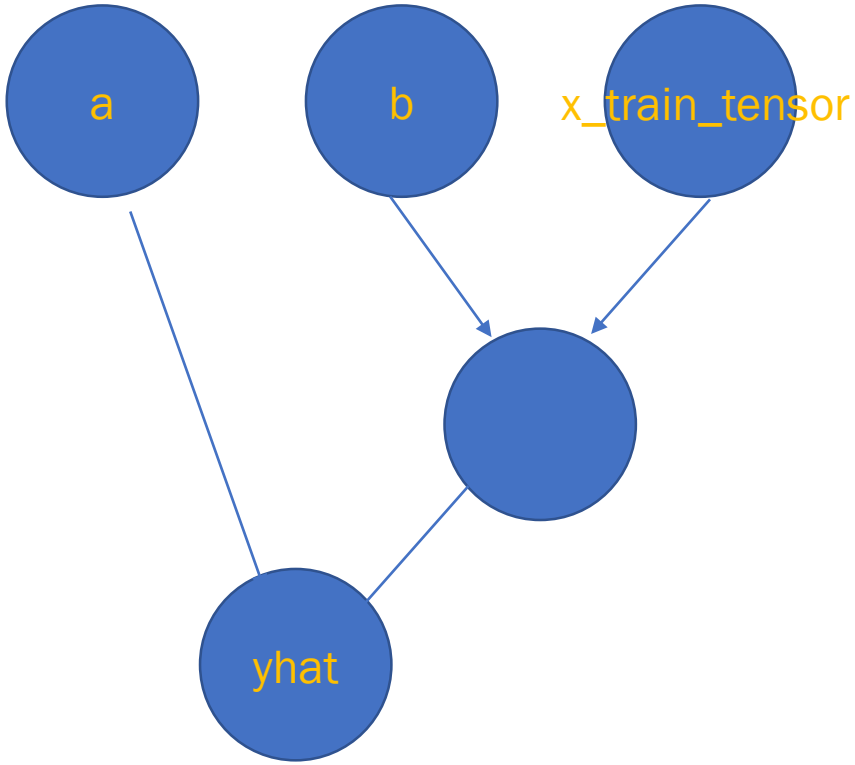
Epoch 1




```
a = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)
b = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)

# Defines a MSE loss function
loss_fn = nn.MSELoss(reduction='mean')

optimizer = optim.SGD([a, b], lr=lr)

for epoch in range(n_epochs):
    yhat = a + b * x_train_tensor

    loss = loss_fn(y_train_tensor, yhat)

    loss.backward()
    optimizer.step()
    optimizer.zero_grad()

print(a, b)
```

Misc

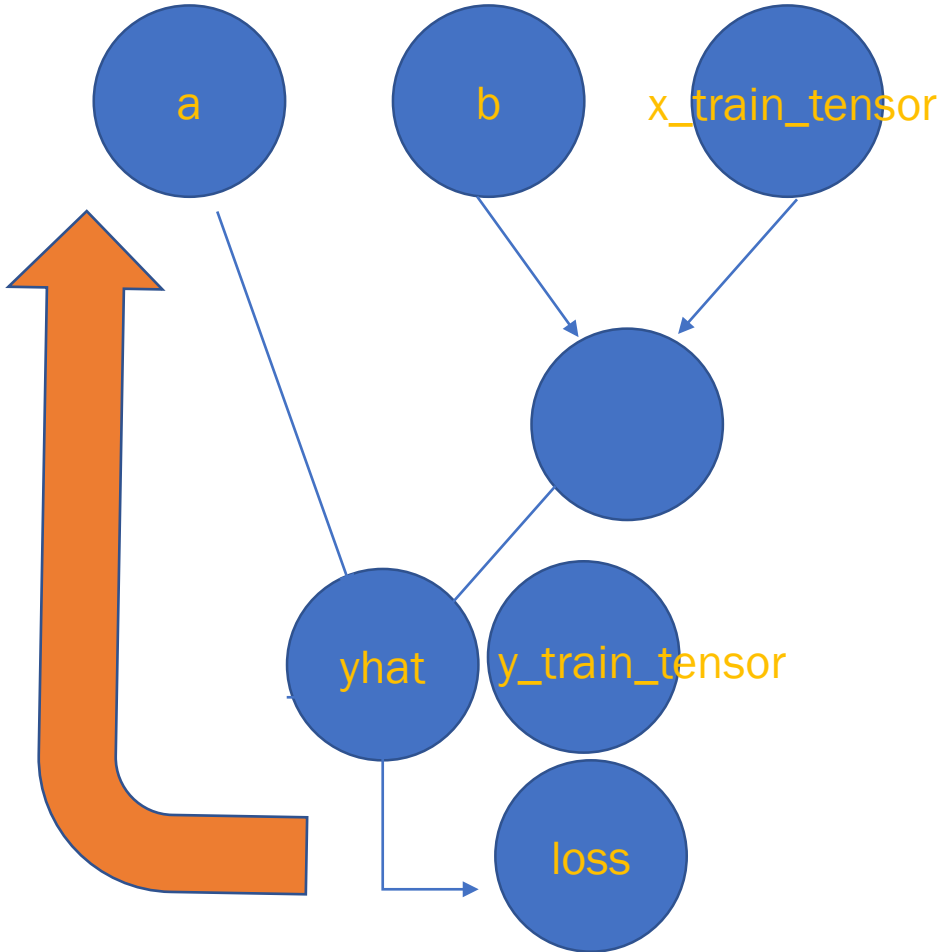
- Dynamic VS Static Computation Graph




```
a = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)
b = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)

# Defines a MSE loss function
loss_fn = nn.MSELoss(reduction='mean')

optimizer = optim.SGD([a, b], lr=lr)

for epoch in range(n_epochs):
    yhat = a + b * x_train_tensor

    loss = loss_fn(y_train_tensor, yhat)

    loss.backward()
    optimizer.step()
    optimizer.zero_grad()

print(a, b)
```

Misc

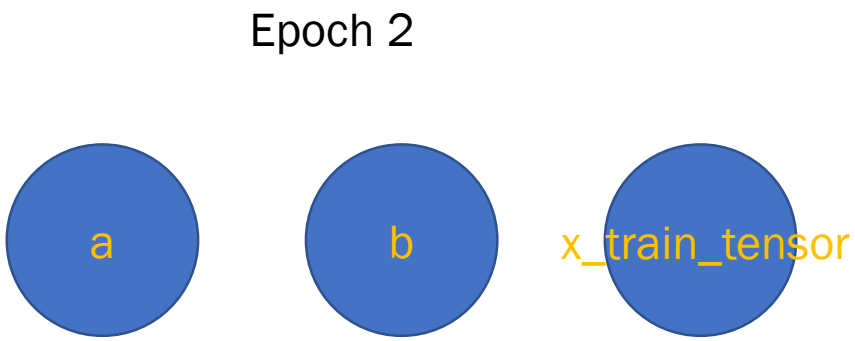
- Dynamic VS Static Computation Graph




```
a = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)
b = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)

# Defines a MSE loss function
loss_fn = nn.MSELoss(reduction='mean')

optimizer = optim.SGD([a, b], lr=lr)

for epoch in range(n_epochs):
    yhat = a + b * x_train_tensor

    loss = loss_fn(y_train_tensor, yhat)

    loss.backward()
    optimizer.step()
    optimizer.zero_grad()

print(a, b)
```

Misc

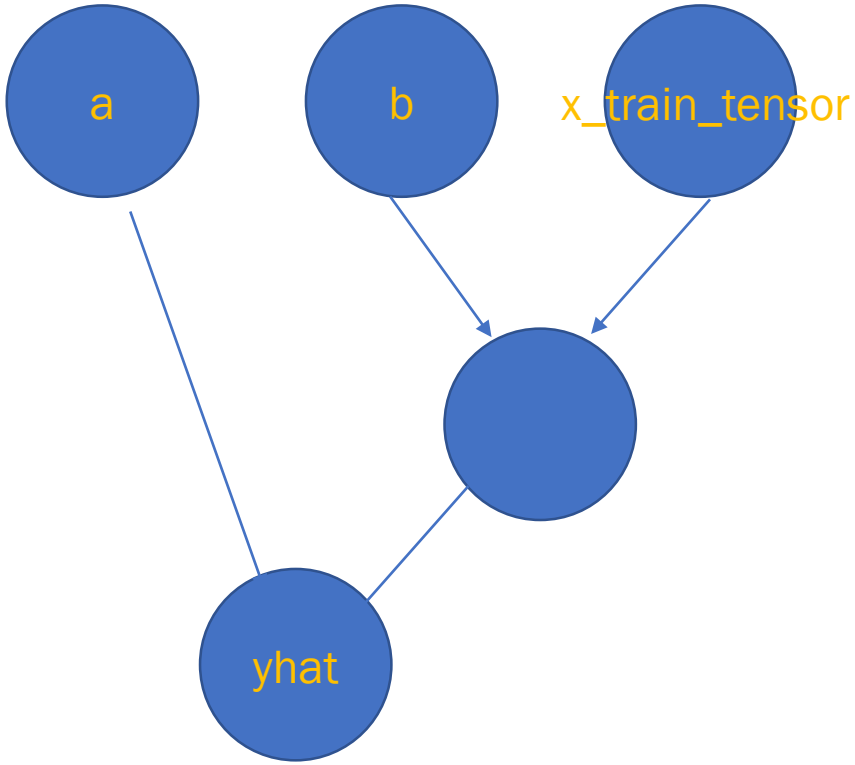
- Dynamic VS Static Computation Graph




```
a = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)
b = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)

# Defines a MSE loss function
loss_fn = nn.MSELoss(reduction='mean')

optimizer = optim.SGD([a, b], lr=lr)

for epoch in range(n_epochs):
    yhat = a + b * x_train_tensor

    loss = loss_fn(y_train_tensor, yhat)

    loss.backward()
    optimizer.step()
    optimizer.zero_grad()

print(a, b)
```

Misc

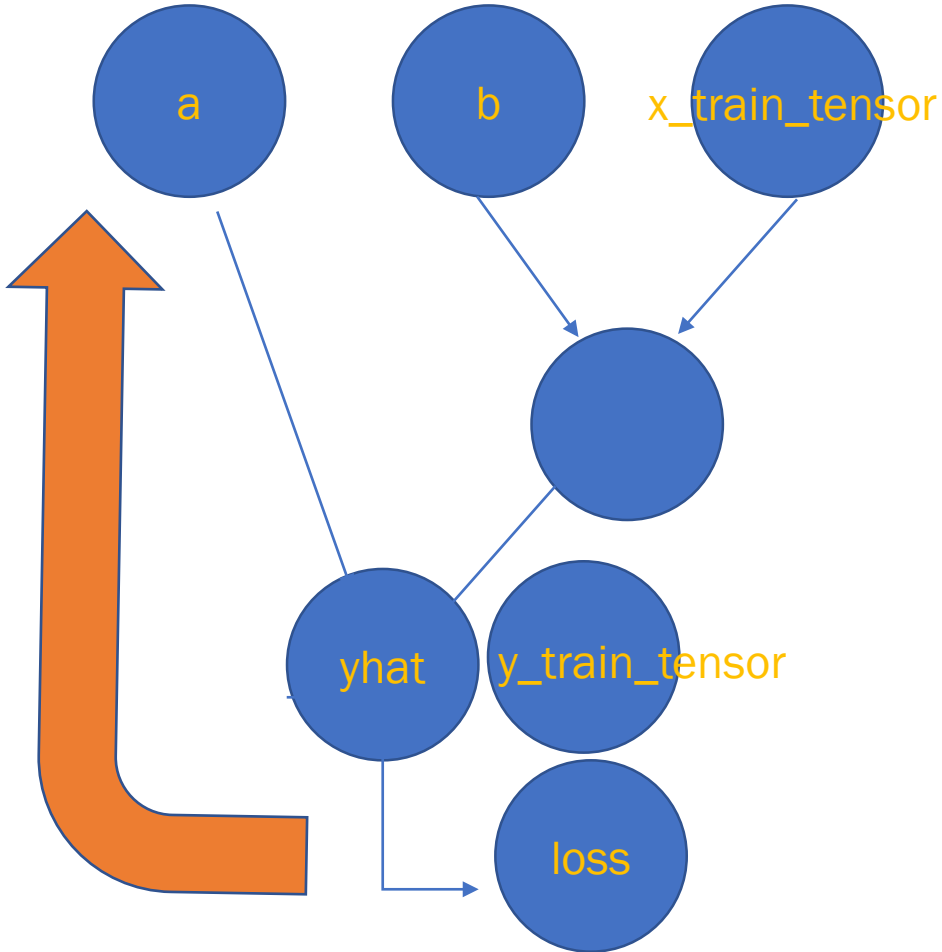
- Dynamic VS Static Computation Graph




```
a = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)
b = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)

# Defines a MSE loss function
loss_fn = nn.MSELoss(reduction='mean')

optimizer = optim.SGD([a, b], lr=lr)

for epoch in range(n_epochs):
    yhat = a + b * x_train_tensor

    loss = loss_fn(y_train_tensor, yhat)

    loss.backward()
    optimizer.step()
    optimizer.zero_grad()

print(a, b)
```

Misc

- Dynamic VS Static Computation Graph




```
a = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)
b = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)

# Defines a MSE loss function
loss_fn = nn.MSELoss(reduction='mean')

optimizer = optim.SGD([a, b], lr=lr)

for epoch in range(n_epochs):
    yhat = a + b * x_train_tensor

    loss = loss_fn(y_train_tensor, yhat)

    loss.backward()
    optimizer.step()
    optimizer.zero_grad()

print(a, b)
```

Misc

- Dynamic VS Static Computation Graph

Building the graph and computing the graph happen at the same time.

Seems inefficient, especially if we are building the same graph over and over again...

```
a = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)
b = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)

# Defines a MSE loss function
loss_fn = nn.MSELoss(reduction='mean')

optimizer = optim.SGD([a, b], lr=lr)

for epoch in range(n_epochs):
    yhat = a + b * x_train_tensor

    loss = loss_fn(y_train_tensor, yhat)

    loss.backward()
    optimizer.step()
    optimizer.zero_grad()

print(a, b)
```

Misc

- Alternative : Static Computation Graphs:

Alternative: Static graphs

Step 1: Build computational graph describing our computation (including finding paths for backprop)

Step 2: Reuse the same graph on every iteration

TensorFlow

```
import numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3, 4

with tf.device('/gpu:0'):
    x = tf.placeholder(tf.float32)
    y = tf.placeholder(tf.float32)
    z = tf.placeholder(tf.float32)

    a = x * y
    b = a + z
    c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                   feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```