



INTRODUCTION TO NEURAL NETWORK PROGRAMMING WITH PYTHON AND PYTORCH

Antonio Carta
antonio.carta@di.unipi.it

Key Features

- **Tensor manipulation:** library to manipulate tensors, with MATLAB/Numpy-like API.
- **GPU support:** seamless execution on GPU and CPU devices.
- **Automatic Differentiation:** custom layers only need to define the forward step, because functions is automatically differentiated using the chain rule.
- **High-level API:** ready-to-use high-level API with neural networks layers, losses, and optimizers

Installation

- Python 3 (or C++)
- Cross-platform
- The library can be installed using pip or conda
- The last stable version is PyTorch 1.5
- GPU-enabled

```
conda install pytorch torchvision cudatoolkit=9.0 -c pytorch
```

- CPU-only

```
conda install pytorch-cpu torchvision -c pytorch
```

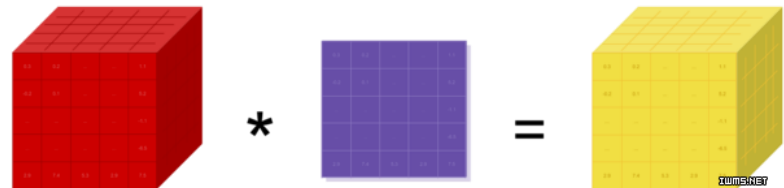
More details on the official website: <http://pytorch.org/>

Tensors

- Tensors are the main data structure. They represent multidimensional arrays
- Equivalent of `numpy.ndarray`
- Support advanced indexing and broadcasting numpy-style

Attributes:

- **dtype**: determine the type of the tensor elements (`float{16, 32, 64}`, `int{8, 16, 32, 64}`, `uint8`). Can be specified during the initialization.
- **device**: memory location (`cpu` or `cuda`)
- **layout**: dense tensors (`strided`) or sparse (`sparse_coo`)



Tensor Initialization

- **torch.tensor**
 - takes any array-like argument and create a new tensor
- **zero initialization**
 - `torch.zeros(*dims)`
- **random**
 - `torch.randn(*dims)`
 - `torch.rand(*dims)`
- **linear range**
 - `torch.linspace(start, end, steps=100)`
- **Numpy bridge**
 - `torch.from_numpy(x)`
 - you can also convert a tensor into a ndarray with the `.numpy` method
 - **note:** the numpy array and the resulting tensor share the memory

```
In [1]: import torch

In [1]: cuda = torch.device("cuda")

In [2]: a = torch.tensor([[1], [2], [3]],
                          dtype=torch.half, device=cuda)

In [3]: print(a)

Out[3]:

tensor([[ 1],
        [ 2],
        [ 3]], device='cuda:0')
```

Tensor Operations

- some operators are overloaded
 - `+`, `-` for addition and subtraction
 - `*` is the elementwise multiplication (not the matrix product)
 - `@` for matrix multiplication (`torch.matmul`)
- in-place operations are defined with a suffix underscore
 - `add_`, `sub_`, `matmul_` are the in-place equivalent for the previous operators
- check the documentation:
<http://pytorch.org/docs/stable/torch.html#tensors>

Indexing

- **basic tensor indexing** is similar to list indexing, but with multiple dimensions
- **boolean condition:** boolean arrays can be used to filter elements that satisfy some condition
- if the indices are less than the number of dimensions the missing indices are considered complete slices

```
# first k elements
x = a[:k]

# all but the first k
x = a[k:]

# negative indexing
x = a[-k:]

# mixed indexing
a[:t_max, b:b+k, :]

# indexing with Boolean condition

def relu(x):
    x[x < 0] = 0
    return x
```

Broadcasting

- Broadcasting allows to perform an operation when tensors have different shapes (e.g. elementwise multiplication between matrix and vector)
- Useful to avoid lots of reshape operations
- Can I use broadcasting? Yes, if you satisfy two conditions:
 - each Tensor has at least one dimension
 - When iterating over the dimension sizes, starting at the trailing dimension, the dimension sizes must either be equal, one of them is 1, or one of them does not exist.

Broadcasting examples

```
In [3]: a = torch.rand(3, 3)
```

```
In [4]: b = torch.rand(3, 1)
```

```
In [5]: s1 = a + b
```

ok, b is expanded
this is equivalent to `a + b.expand(-1, 3)`

```
In [6]: c = torch.rand(3, 1, 1)
```

```
In [7]: s2 = a + c
```

ok, a and c are expanded
`a.unsqueeze(2).expand(3,3,3) + c.expand(3,3,3)`

```
In [8]: d = torch.rand(3, 2)
```

```
In [9]: a + d
```

error, a and d are not broadcastable

```
RuntimeError: inconsistent tensor size, expected r_ [3 x 3], t  
[3 x 3] and src [3 x 2] to have the same number of elements,  
but got 9, 9 and 6 elements respectively at  
d:\projects\pytorch\torch\lib\th\generic/THTensorMath.c:887
```

GPU usage



- **torch.cuda** API for GPU management
- during the creation of a tensor you can choose the device (CPU or GPU)
- all the tensor arguments of an operator must reside on the same device
- the result of the operation will be allocated on the same device
- Tensors can be moved to the GPU with **cuda** and **to** methods
- can take the GPU id as an optional argument if you have multiple GPUs
- You can move tensors to the CPU with the **cpu** method
- Check if CUDA is supported on the machine with **torch.cuda.is_available**

CUDA – How to select a single GPU

On a server you typically have access to multiple shared GPU and you must select one to run your code.

- Manual selection using the device argument ('cuda:0', 'cuda:1'...)
- Using the context manager `torch.cuda.device`
- Changing the shell environment variable `CUDA_VISIBLE_DEVICES` to limit the visible GPUs
 - `export CUDA_VISIBLE_DEVICES=0`
- The library `setGPU` automatically finds the best GPU for you and uses `CUDA_VISIBLE_DEVICES` to mask the others (<https://github.com/bamos/setGPU>)
 - `import setGPU` at the top of your main script

```

cuda = torch.device(device('cuda')) # Default CUDA device
cuda0 = torch.device(device('cuda:0'))
cuda2 = torch.device(device('cuda:2')) # GPU 2

x = torch.tensor([1., 2.], device=cuda0)
# x.device is device(type='cuda', index=0)
y = torch.tensor([1., 2.]).cuda()
# y.device is device(type='cuda', index=0)

With torch.cuda.device(1):
    # allocates a tensor on GPU 1
    a = torch.tensor([1., 2.], device=cuda)

    # transfers a tensor from CPU to GPU 1
    b = torch.tensor([1., 2.]).cuda()
    # a.device and b.device are device(type='cuda', index=1)

    # You can also use ``Tensor.to`` to transfer a tensor:
    b2 = torch.tensor([1., 2.]).to(device=cuda)
    # b.device and b2.device are device(type='cuda', index=1)

    c = a + b # c.device is device(type='cuda', index=1)
    z = x + y # z.device is device(type='cuda', index=0)

    # even within a context, you can specify the device
    # (or give a GPU index to the .cuda call)
    d = torch.randn(2, device=cuda2)
    e = torch.randn(2).to(cuda2)
    f = torch.randn(2).cuda(cuda2)
    # d.device, e.device, and f.device are all device(type='cuda', index=2)

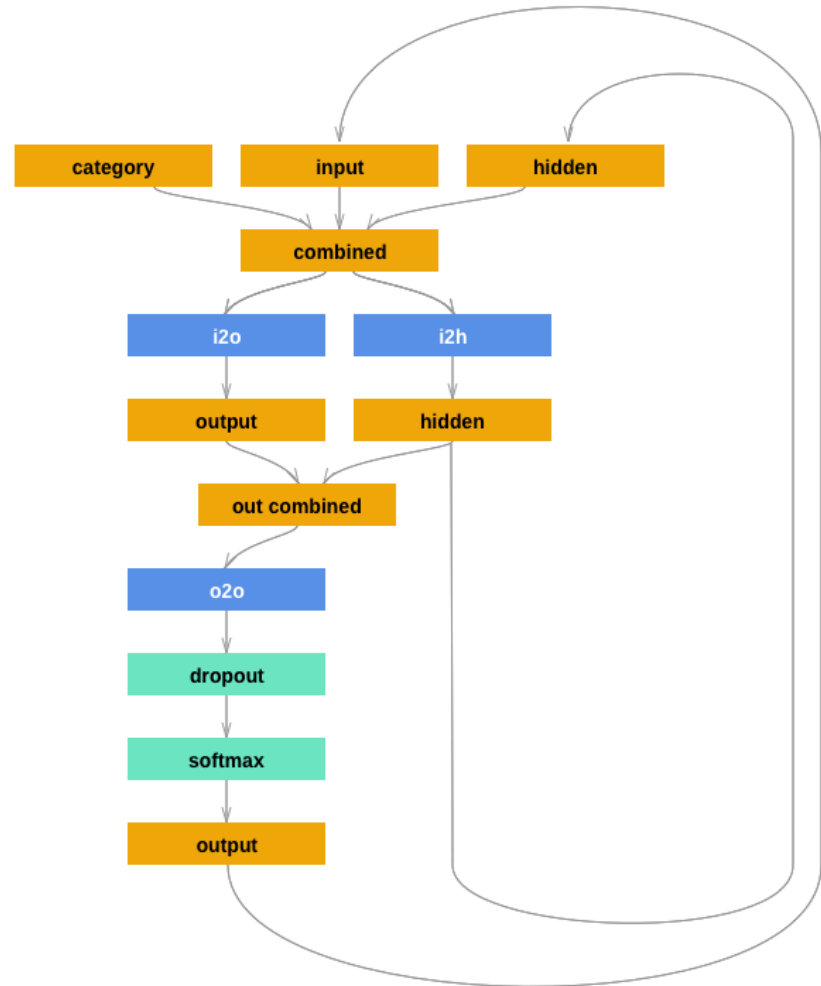
```

Automatic Differentiation

torch.autograd is the package responsible for the automatic differentiation

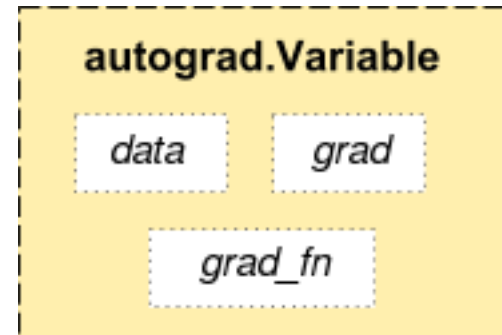
Each computation creates a dynamic computational graph. Each operation adds a **Function** node, connected to its **Tensor** arguments

The graph is used to compute the gradient by calling the method **backward**.



Autograd (1)

- Tensor objects are the data nodes of the computational graph
- The main attributes related to the graph structure are:
 - **data**: Tensor containing the Variable value
 - **grad**: Tensor containing the gradient (initially set to None)
 - **grad_fn**: the function used to compute the gradient
- Each Function implements two methods:
 - **forward**: function application
 - **backward**: gradient computation



Autograd (2)

- The **requires_grad** attribute is used to specify if the gradient computation should propagate into the Tensor or stop
 - for model's parameters `requires_grad=True`
 - for input data or constant values `requires_grad=False`
- You can truncate the gradient using **detach**. The method removes the Tensor from the graph, making it a leaf.
- **in-place** modification is not allowed because it breaks the automatic differentiation.
- at inference time you can speed up the computation by using the context manager **torch.no_grad**, which disables the graph construction required for the backward computation, saving space and time.
- autograd documentation
<http://pytorch.org/docs/stable/autograd.html>

graph leaves. Data and Parameters

Back-propagation
uses the dynamically built graph

```
from torch.autograd import Variable
```

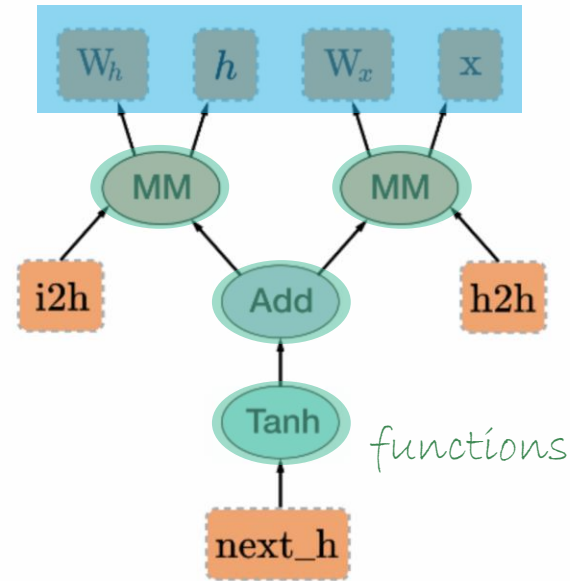
```
x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))

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

```
next_h.backward(torch.ones(1, 20))
```

gradient computation

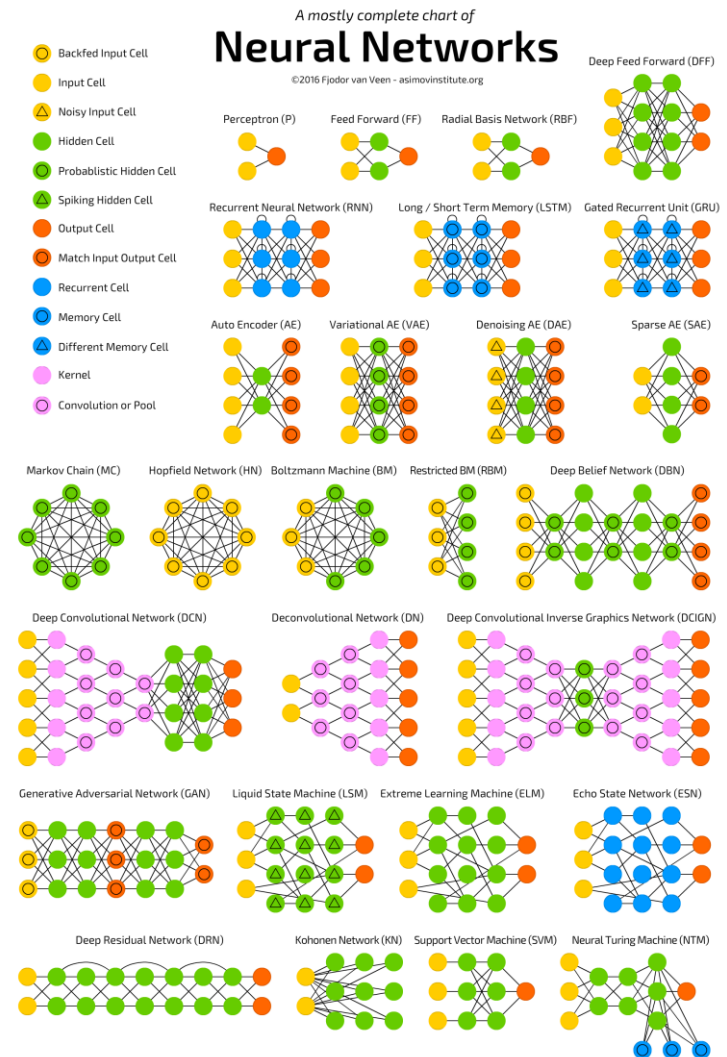
forward
step and
dynamic
graph
creation



Building the dynamic graph

torch.nn

- torch.nn contains the basic components to define your neural networks, loss functions, regularization techniques and optimizers.
- We will see in the next few slides
 - What is a Module
 - how to define a custom Module
 - how to set up a basic training loop



nn.Module

- **Module** is the base class for all the neural network submodules
 - Linear, convolutional, recurrent layers are all Module subclasses
- A Module contain **Parameters**:
 - these are typically the trainable parameters of your model
 - Parameter is a wrapper of a tensor with **requires_grad=True**
 - you can iterate over all the parameters using the **parameters()** method
- you can compute the output of a network by using it like a function (e.g. **y_pred = net(X)**)
 - that is possible because **__apply__** is overridden
 - the computation is performed by the **forward** method, but if you forward directly the module's hooks are not activated
- It is possible to define forward and backward hooks
 - e.g. you can check for NaN gradients after the backward pass
 - you can register the hook with methods like **register_forward_hook()**

How to Subclass Module

- Override the **forward** method to define how the computation is performed. Backward is automatically implemented with autograd
- Override the **__init__** method, defining your parameters
 - remember to call the constructor of the super class!
- When you add a **Parameter** as an attribute it is automatically registered for you. It also works for submodules.
- If you want to add a list of parameters or modules use the **ParameterList** and **ModuleList** containers. If you use a list the parameter will not be registered and cannot be iterated with the **parameters** method
- you can print the network to see the registered parameters and submodules

Example – Custom Module

```
class Net(nn.Module):
    def __init__(self):
        super().__init__() <- remember to call the superclass
        # 1 input image channel, 6 output channels,
        # 5x5 square convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        # an affine operation:  $y = Wx + b$ 
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        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
```

ParameterList usage

```
model = MyModel(**kwargs)
```

Correct!

```
model.x = nn.ParameterList([  
    torch.randn((10, 10), requires_grad=True),  
    torch.randn((10, 10), requires_grad=True)  
])
```

Wrong!!!

```
model.x = [  
    torch.randn((10, 10), requires_grad=True),  
    torch.randn((10, 10), requires_grad=True)  
]
```

Example – print model

if we create a Net object and print it we obtain the following output:

```
Net(  
  (conv1): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1))  
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))  
  (fc1): Linear(in_features=400, 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)  
)
```

We can see the two convolutional layers and the three fully connected layers.

Loss

- To define a training loop we need a loss and an optimizer
- torch.nn defines many different loss functions
 - nn.MSELoss, nn.CrossEntropyLoss, nn.NLLLoss, nn.BCELoss, ...
 - you can also use the functional version, defined in nn.functional. The only difference is that you don't need to create an object.
 - **always** check to documentation for the correct shape and input arguments (does the loss needs logits or probabilities? Which dimension should be the last? Is the average for each element or for each sample?)

```
import nn.functional as F
net = Net()
out = net(X)
loss = F.MSELoss(out, target)
```

Optimizer

- Simple gradient descent:

```
learning_rate = 0.01
net.zero_grad() ← do not forget it
loss.backward()
for f in net.parameters():
    f.data.sub_(f.grad.data * learning_rate)
```

- note the call to the **zero_grad** method. It is needed to reset the gradient buffers
- you can also use an optimizer defined in torch.optim (next slide)
 - SGD, Adam, RMSProp
 - they take as arguments the learning rate, momentum, l2 weight decay
 - the **step** method performs the update (no need to call sub_ explicitly)

Training Loop

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

for epoch in range(100): # loop over the dataset multiple
    times
    running_loss = 0.0
    for i, data in enumerate(dataset):
        inputs, labels = data # get the inputs
        optimizer.zero_grad() # zero the parameter gradients

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.data[0]
        if i % 2000 == 1999: # print every 2000 mini-batches
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0

print('Finished Training')
```

torch.nn Modules

Available Modules:

- Convolutional layers: Conv2D, MaxPool2D
- Recurrent layers: RNN, LSTM, GRU, {RNN, LSTM, GRU}Cell
- FeedForward: Linear
- activation functions defined in torch.nn.functional

train/eval mode

- Modules have a **train/eval** mode. This is useful for layers (e.g. Dropout, BatchNormalization) that define a different behaviour during train and test. Always set it during training with **net.train()**, and disable it during the test phase (**net.eval()**).

Feedforward Network

Simple architectures can be defined as a **sequential** application of modules.

A feedforward network is a sequence of linear transformations and nonlinear activations.

```
import torch.nn as nn

model = nn.Sequential(
    nn.Linear(100, 50),
    nn.ReLU(),
    nn.Linear(50, 50),
    nn.ReLU(),
    nn.Linear(50, 10),
    nn.Softmax()
)

y_out = model(X)
```

Recurrent Neural Networks

{LSTM, RNN, GRU}Cell
implement a recurrent layer.
Combining them we can
build a recurrent network.

The default input shape is
(time, batch, features)

Different from Keras.

You also need to keep track
of the hidden and cell
states.

```
model = torch.nn.LSTMCell(input_size, hidden_size)

out = []

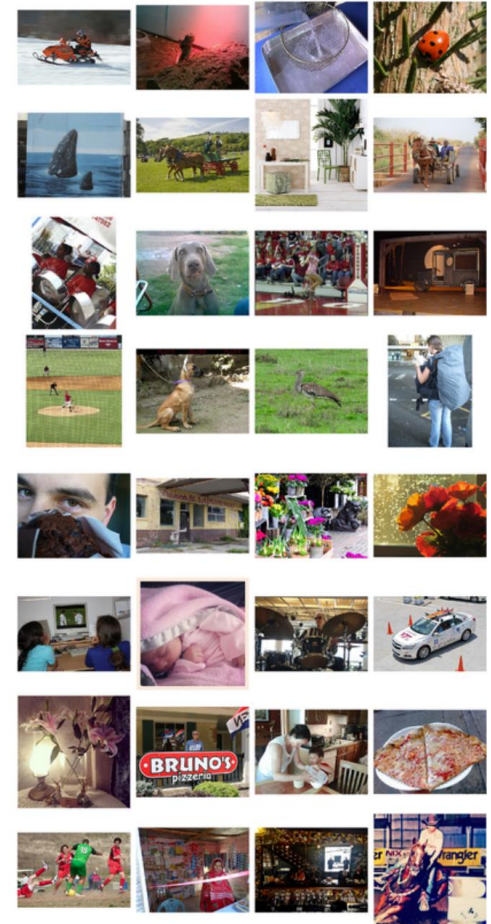
h_prev = torch.zeros((batch_size, hidden_size))
c_prev = torch.zeros((batch_size, hidden_size))

for t in range(n_steps):
    X_t = X[t]
    h_prev, c_prev = model(X_t, (h_prev, c_prev))
    out.append(o_t)

out = torch.stack(out)
```

Dataset

- For small datasets you can load the data as a numpy array and convert it to a pytorch tensor (remember to check the dimensions in case you need to transpose some dimension)
- the alternative is to use the utilities provided in **torch.data.utils**
- **DataLoader** can be used to load the dataset in parallel. It is useful only when you are using heavy preprocessing (e.g. image data with lots of data augmentation)
- **Sampler** classes for sequential or random sampling from a dataset.
- check the documentation:
<http://pytorch.org/docs/stable/data.html>



Model serialization and logging

- PyTorch provides some guidelines regarding serialization
<http://pytorch.org/docs/stable/notes/serialization.html>
- save a network

```
torch.save(the_model.state_dict(), PATH)
```

- load back the model

```
the_model = TheModelClass(*args, **kwargs)
```

```
the_model.load_state_dict(torch.load(PATH))
```

- You can also use Tensorboard to log training metrics
 - <https://pytorch.org/docs/stable/tensorboard.html>

Static vs Dynamic Graphs

Computation Graph: representation of your model's computation, needed to perform the backpropagation.

Static graph: Defined statically.

- More possibility for optimization and other graph transformations
- Harder to code and debug
- Painful to code if you have input with a dynamic structure

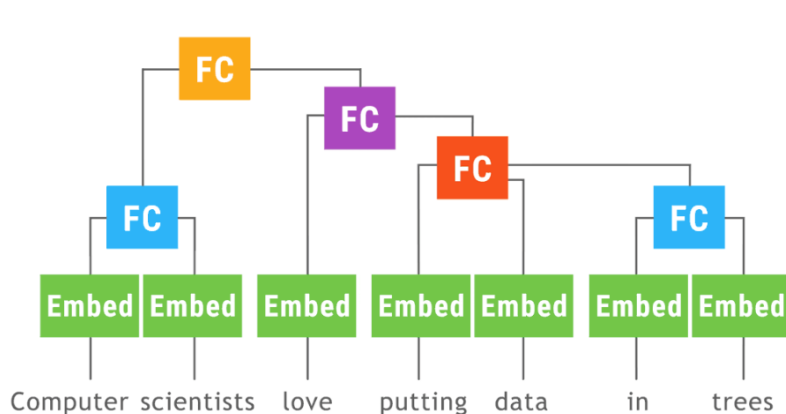
Dynamic graph: Defined and created at runtime.

- Easier to code
- Slower execution (not always)

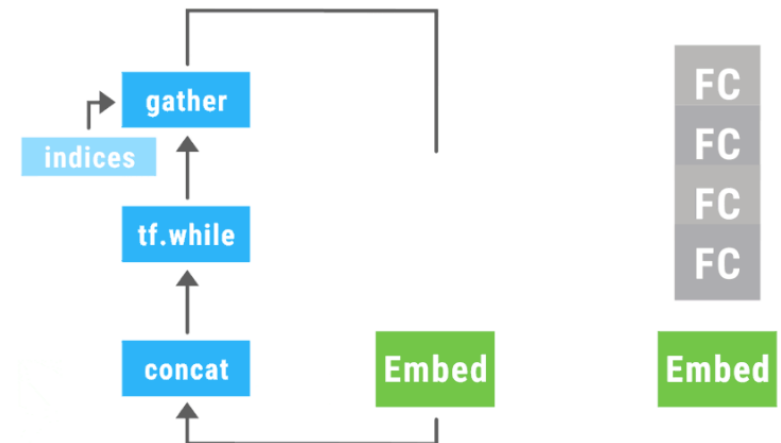
Pytorch uses dynamic graphs by default. For static graphs you need to use **torch.jit**

Static vs Dynamic Graphs (2)

- **Debugging:** dynamic graphs are (a lot) easier to debug due to the ability to track the variables at runtime. The execution of a static graph is harder to inspect.
- **Structured Data:** variable structures are easy to process with dynamic graphs. Static graphs require explicit control flow and dynamic batching to process structured data efficiently.
- **Deployment:** static graphs are easier to deploy and can be easily serialized and loaded into different environments.
- **Optimization:** static graphs are easier to optimize. You can gain about 30% with basic CNN in memory and time consumption with a fully optimized graph at inference time



With a **dynamic** approach the computational graph and the sample have the same structure.



With a **static** approach the computational graph contains explicit control flow because it remains the same for different samples.

Pytorch lightning

- High-level keras-like library that implements most of the boilerplate code for you
- Uses a **callback system** that allows the user to customize the training loop and easily add functionality

PyTorch

```
# model
class Net(nn.Module):
    def __init__(self):
        self.layer_1 = torch.nn.Linear(28 * 28, 128)
        self.layer_2 = torch.nn.Linear(128, 10)

    def forward(self, x):
        x = x.view(x.size(0), -1)
        x = self.layer_1(x)
        x = F.relu(x)
        x = self.layer_2(x)
        return x

# train loader
mnist_train = MNIST(os.getcwd(), train=True, download=True,
                    transform=transforms.ToTensor())
mnist_train = DataLoader(mnist_train, batch_size=64)

net = Net()

# optimizer + scheduler
optimizer = torch.optim.Adam(net.parameters(), lr=1e-3)
scheduler = StepLR(optimizer, step_size=1)

# train
for epoch in range(1, 100):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)

        loss.backward()
        optimizer.step()
    if batch_idx % args.log_interval == 0:
        print('Train Epoch: {} {} {} ({:.0f}%) \tLoss: {:.6f}'.format(
            epoch, batch_idx * len(data), len(train_loader.dataset),
            100. * batch_idx / len(train_loader), loss.item()))
```

PyTorch Lightning

```
# model
class Net(LightningModule):
    def __init__(self):
        self.layer_1 = torch.nn.Linear(28 * 28, 128)
        self.layer_2 = torch.nn.Linear(128, 10)

    def forward(self, x):
        x = x.view(x.size(0), -1)
        x = self.layer_1(x)
        x = F.relu(x)
        x = self.layer_2(x)
        return x

    def train_loader(self):
        mnist_train = MNIST(os.getcwd(), train=True, download=True,
                            transform=transforms.ToTensor())
        return DataLoader(mnist_train, batch_size=64)

    def configure_optimizers(self):
        optimizer = torch.optim.Adam(self.parameters(), lr=1e-3)
        scheduler = StepLR(optimizer, step_size=1)
        return optimizer, scheduler

    def training_step(self, batch, batch_idx):
        data, target = batch
        output = self.forward(data)
        loss = F.nll_loss(output, target)
        return {'loss': loss}
```

```

# this is just a plain nn.Module with some structure
class LitClassifier(pl.LightningModule):
    def __init__(self):
        super().__init__()
        self.l1 = torch.nn.Linear(28 * 28, 10)

    def forward(self, x):
        return torch.relu(self.l1(x.view(x.size(0), -1)))

    def training_step(self, batch, batch_nb):
        x, y = batch
        loss = F.cross_entropy(self(x), y)
        tensorboard_logs = {'train_loss': loss}
        return {'loss': loss, 'log': tensorboard_logs}

    def configure_optimizers(self):
        return torch.optim.Adam(self.parameters(), lr=0.02)

# train!
train_loader = DataLoader(MNIST(os.getcwd(), train=True, download=True,
                                transform=transforms.ToTensor()), batch_size=32)
model = LitClassifier()
trainer = pl.Trainer(gpus=8, precision=16)
trainer.fit(model, train_loader)

```

Example Code

- Regression

<https://github.com/pytorch/examples/blob/master/regression/main.py>

- CNN

<https://github.com/pytorch/examples/blob/master/mnist/main.py>

References

official documentation: <http://pytorch.org/docs>

official tutorials: <http://pytorch.org/tutorials/>

official examples: <https://github.com/pytorch/examples>

PyTorch in 60 minutes:

http://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html

tensor manipulation: <https://github.com/rougier/numpy-100>