UiO : Department of Informatics
University of Oslo

IN5400 Machine learning for image classification

Lecture 4 : Introduction to Pytorch

Tollef Jahren

February 6, 2019

About today

- You will get an introduction to Pytorch.
- Pytorch is a widely used deep learning framework, especially in academia.

- Pytorch version 1.0
- Python 3.6

Outline

- Deep learning frameworks
- Pytorch
  - torch.tensor
  - Computational graph
  - Automatic differentiation (torch.autograd)
  - Data loading and preprocessing (torch.utils)
  - Useful functions (torch.nn.functional)
  - Creating the model (torch.nn)
  - Optimizers (torch.optim)
  - Save/load models
- Miscellaneous

Readings

- **Text**:
  - https://pytorch.org/tutorials/

- Note:
  - Don’t be confused. A lot of the available code online is written in an older version of Pytorch. We are using Pytorch version 1.0.

Progress

- **Deep learning frameworks**
- Pytorch
  - torch.tensor
  - Computational graph
  - Automatic differentiation (torch.autograd)
  - Data loading and preprocessing (torch.utils)
  - Useful functions (torch.nn.functional)
  - Creating the model (torch.nn)
  - Optimizers (torch.optim)
  - Save/load models
- Miscellaneous

Why do we need Deep learning frameworks?

- **Speed:**
  - Fast GPU/CPU implementation of matrix multiplication, convolutions and backpropagation

- **Automatic differentiations**:
  - Pre-implementation of the most common functions and it’s gradients.

- **Reuse:**
  - Easy to reuse other people’s models

- **Less error prone:**
  - The more code you write yourself, the more errors

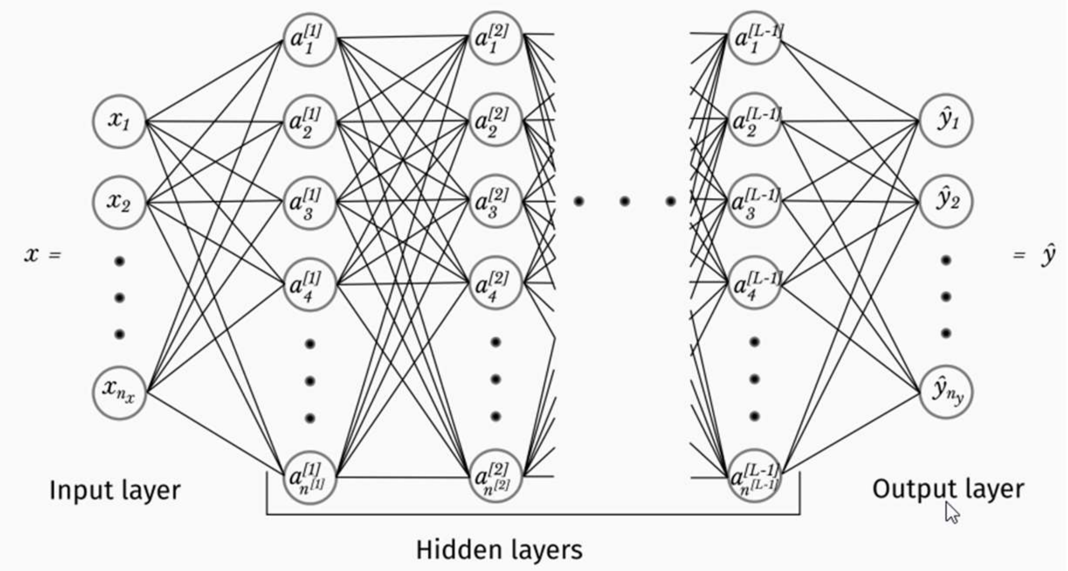
Deep learning frameworks

- Deep learning frameworks does a lot of the complicated computation, remember last week….

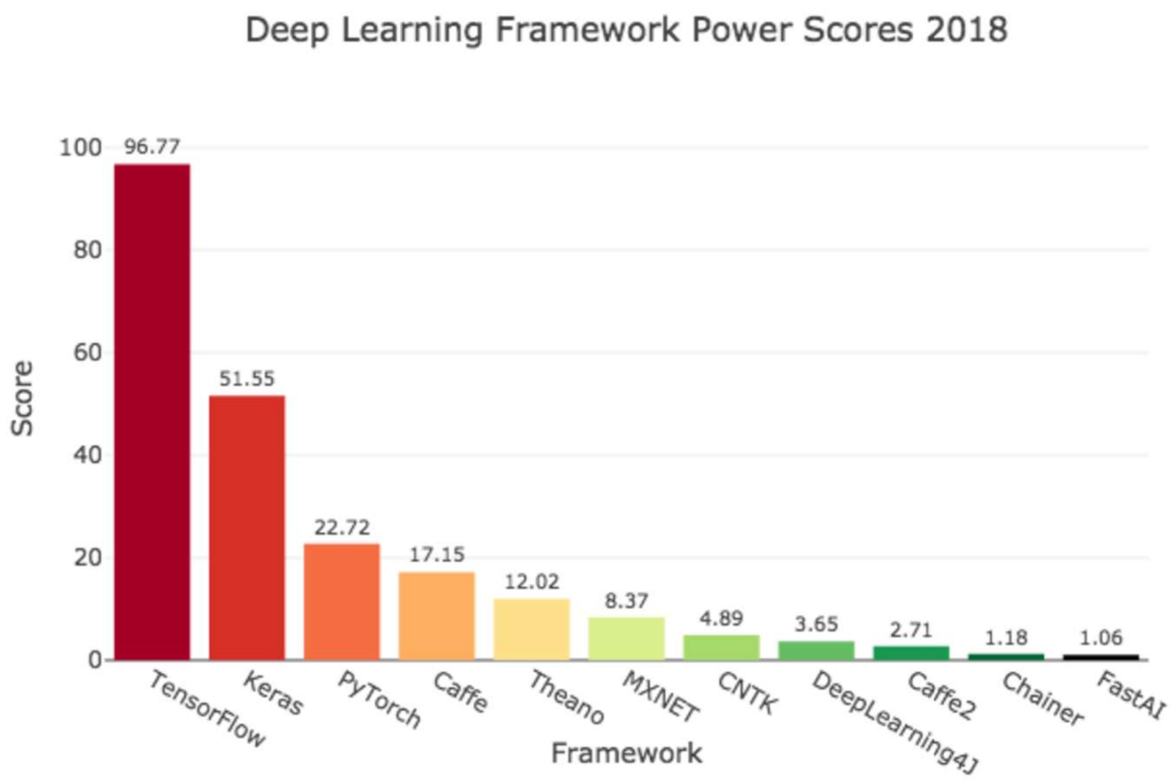
$$
\begin{aligned}
z_k^{[L]} &= \sum_{j=1}^{n^{[L-1]}} w_{jk}^{[L]} a_j^{[L-1]} + b_k^{[L]} \\
a_k^{[L]} &= s(z_k^{[L]}) \\
&= \hat{y}_k
\end{aligned}
$$

for

$$
\begin{aligned}
k &= 1, \ldots, n_y, \\
&= 1, \ldots, n^{[L]}.
\end{aligned}
$$



Popularity



- https://towardsdatascience.com/deep-learning-framework-power-scores-2018-23607ddf297a

Why Pytorch

- Python API
- Can use CPU, GPU
- Supports common platforms:
  - Windows, Mac, Linux

- Pytorch is a thin framework which lets you work closely with programming the neural network
- Focus on the machine learn part not the framework itself
- Pythonic control flow
  - Flexible
  - Cleaner and more intuitive code
  - Easy to debug

- Python debugger
  - With Pytorch we can use the python debugger
  - It does not run all in a c++ environment abstracted way

Pytorch packages

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.

Progress

- Deep learning frameworks
- Pytorch
  - **torch.tensor**
  - Computational graph
  - Automatic differentiation (torch.autograd)
  - Data loading and preprocessing (torch.utils)
  - Useful functions (torch.nn.functional)
  - Creating the model (torch.nn)
  - Optimizers (torch.optim)
  - Save/load models
- Miscellaneous

torch.Tensor class

- Pytorch's tensors are similar to NumPy's ndarrays

```
In [1]: import torch
        V = torch.tensor(1.3 )
        V

Out[1]: tensor(1.3000)
```

```
In [1]: import torch
        V = torch.tensor([1., 2., 3.])
        V

Out[1]: tensor([1., 2., 3.])
```

```
In [8]: import torch
        V = torch.tensor([[1., 2.],[4., 5.]])
        V

Out[8]: tensor([[1., 2.],
                [4., 5.]])
```

```
In [2]: import numpy as np
        V = np.array(1.3)
        V

Out[2]: array(1.3)
```

```
In [2]: import numpy as np
        V = np.array([1., 2., 3.])
        V

Out[2]: array([1., 2., 3.])
```

```
In [9]: import numpy as np
        V = np.array([[1., 2.],[4., 5.]])
        V

Out[9]: array([[1., 2.],
               [4., 5.]])
```

Creating an instance of torch.Tensor

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

```
In [2]: data = np.array([1,2,3], dtype=np.int32)
```

```
In [3]: t1 = torch.Tensor(data)        # Constructor
        t2 = torch.tensor(data)        # Factory function
        t3 = torch.as_tensor(data)     # Factory function
        t4 = torch.from_numpy(data)    # Factory function
```

```
In [4]: print(t1)
        print(t2)
        print(t3)
        print(t4)

tensor([1., 2., 3.])
tensor([1, 2, 3], dtype=torch.int32)
tensor([1, 2, 3], dtype=torch.int32)
tensor([1, 2, 3], dtype=torch.int32)
```

```
In [5]: print(t1.dtype)
        print(t2.dtype)
        print(t3.dtype)
        print(t4.dtype)

torch.float32
torch.int32
torch.int32
torch.int32
```

```
In [6]: torch.get_default_dtype()

Out[6]: torch.float32
```

```
In [9]: t2 = torch.tensor(data, dtype=torch.float64)
        t2

Out[9]: tensor([1., 2., 3.], dtype=torch.float64)
```

Memory: Sharing vs Copying

In [2]:
```python
data = np.array([1,2,3])
data
```
Out[2]:
```
array([1, 2, 3])
```

In [3]:
```python
t1 = torch.Tensor(data)
t2 = torch.tensor(data)
t3 = torch.as_tensor(data)
t4 = torch.from_numpy(data)
```

In [4]:
```python
data[:] = 0
data
```
Out[4]:
```
array([0, 0, 0])
```

In [5]:
```python
print(t1)
print(t2)
```
```
tensor([1., 2., 3.])
tensor([1, 2, 3], dtype=torch.int32)
```

In [6]:
```python
print(t3)
print(t4)
```
```
tensor([0, 0, 0], dtype=torch.int32)
tensor([0, 0, 0], dtype=torch.int32)
```

Sharing memory for performance : Share vs Copy

- **Share Data**
- Torch.as_tensor()
- Torch.from_numpy()

- **Copy Data**
- Torch.Tensor()
- Torch.tensor()

Recommended

Creating instances of torch.Tensor without data

```
In [7]: torch.eye(2)

Out[7]: tensor([[1., 0.],
                [0., 1.]])

In [8]: torch.zeros(2,2)

Out[8]: tensor([[0., 0.],
                [0., 0.]])

In [9]: torch.ones(2,2)

Out[9]: tensor([[1., 1.],
                [1., 1.]])

In [10]: torch.rand(2,2)

Out[10]: tensor([[0.2696, 0.2011],
                 [0.5265, 0.4603]])
```

Tensor indexing

- We can use «normal» indexing as in NumPy

```
In [1]: import torch
        data = torch.tensor([[1,2,3],[4,5,6]])
        data

Out[1]: tensor([[1, 2, 3],
                [4, 5, 6]])
```

```
In [3]: data[1,0]

Out[3]: tensor(4)
```

```
In [4]: data[0,:]

Out[4]: tensor([1, 2, 3])
```

Torch.tensor attributes

Attribute	Data type	Description
data	array_like	list, tuple, NumPy ndarray, scalar
dtype	torch.dtype	The tensor's data type
requires_grad	bool	Should autograd record operation
device	torch.device	Allocated on CPU or CUDA (GPU)

```
In [5]: torch.tensor(data=[1,2,3], dtype=torch.float32, device='cpu', requires_grad=False)
```

Data types

Data type	dtype	CPU tensor	GPU tensor
32-bit floating point	`torch.float32` or `torch.float`	`torch.FloatTensor`	`torch.cuda.FloatTensor`
64-bit floating point	`torch.float64` or `torch.double`	`torch.DoubleTensor`	`torch.cuda.DoubleTensor`
16-bit floating point	`torch.float16` or `torch.half`	`torch.HalfTensor`	`torch.cuda.HalfTensor`
8-bit integer (unsigned)	`torch.uint8`	`torch.ByteTensor`	`torch.cuda.ByteTensor`
8-bit integer (signed)	`torch.int8`	`torch.CharTensor`	`torch.cuda.CharTensor`
16-bit integer (signed)	`torch.int16` or `torch.short`	`torch.ShortTensor`	`torch.cuda.ShortTensor`
32-bit integer (signed)	`torch.int32` or `torch.int`	`torch.IntTensor`	`torch.cuda.IntTensor`
64-bit integer (signed)	`torch.int64` or `torch.long`	`torch.LongTensor`	`torch.cuda.LongTensor`

[

Operations between tensors of different data type is not allowed

```
In [2]: t1 = torch.tensor([1, 2, 3], dtype=torch.int32)
        t1.dtype

Out[2]: torch.int32

In [3]: t2 = torch.tensor([1, 2, 3], dtype=torch.float32)
        t2.dtype

Out[3]: torch.float32

In [4]: t3 = t1 + t2

---------------------------------------------------------------------------
--
RuntimeError                              Traceback (most recent call las
t)
<ipython-input-4-8140cb83dabf> in <module>
----> 1 t3 = t1 + t2

RuntimeError: expected type torch.FloatTensor but got torch.IntTensor
```

Device – CPU / CUDA

- Allocating torch.tensor's on various devices

```
In [2]: t_cpu = torch.tensor(data=[1,2,3], device='cpu')
        t_cpu.device

Out[2]: device(type='cpu')

In [3]: t_cuda = torch.tensor(data=[1,2,3], device='cuda:0')
        t_cuda.device

Out[3]: device(type='cuda', index=0)

In [4]: torch.cuda.current_device()

Out[4]: 0

In [5]: t_cuda.to('cpu')

Out[5]: tensor([1, 2, 3])

In [6]: t_cpu.to('cuda:0')

Out[6]: tensor([1, 2, 3], device='cuda:0')
```

Operations between tensors on different devices is not allowed

```
In [2]: t1 = torch.tensor([1., 2., 3.], device='cpu')
        t1.device

Out[2]: device(type='cpu')

In [3]: t2 = torch.tensor([1., 2., 3.], device='cuda')
        t2.device

Out[3]: device(type='cuda', index=0)

In [4]: t3 = t1 + t2

---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-4-8140cb83dabf> in <module>
----> 1 t3 = t1 + t2

RuntimeError: expected type torch.FloatTensor but got torch.cuda.FloatTensor
```

Torch.tensor functionality

- Common tensor operations:
  - reshape
  - max/min
  - shape/size
  - etc

- Arithmetic operations
  - Abs / round / sqrt / pow / etc

- torch.tensor's support broadcasting

- In-place operations

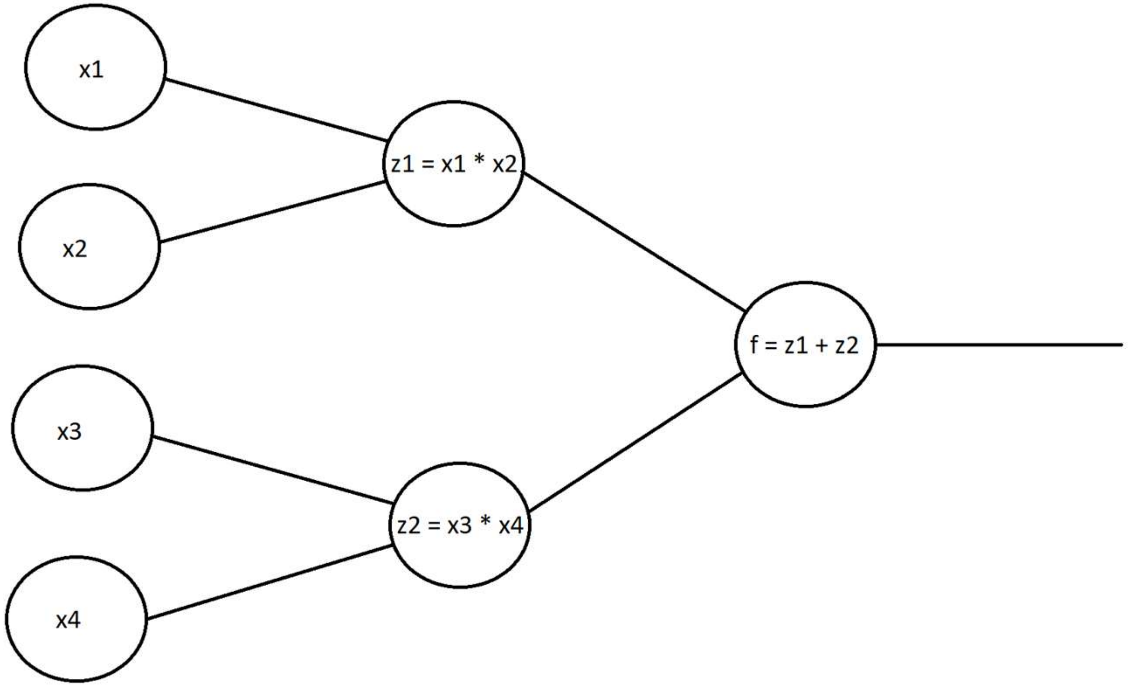
Progress

- Deep learning frameworks
- Pytorch
  - torch.tensor
  - **Computational graph**
  - Automatic differentiation (torch.autograd)
  - Data loading and preprocessing (torch.utils)
  - Useful functions (torch.nn.functional)
  - Creating the model (torch.nn)
  - Optimizers (torch.optim)
  - Save/load models
- Miscellaneous

What is a computational graph?

$f(\vec{x}) = x_1 * x_2 + x_3 * x_4$

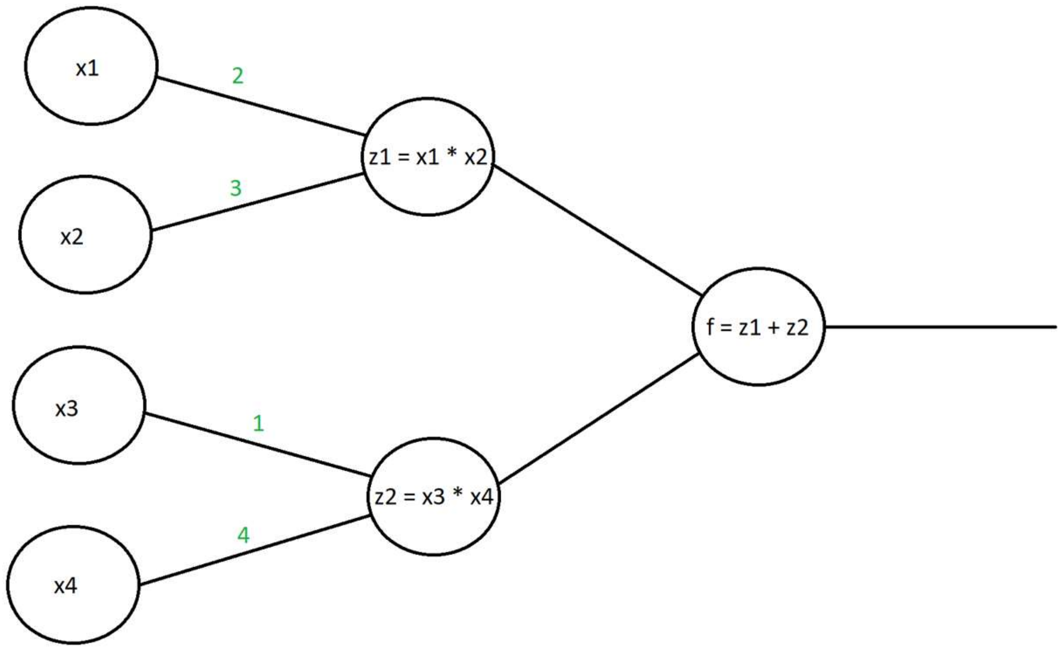
$f(\vec{x}) = z_1 + z_2$



Forward propagation

$f(\vec{x}) = x_1 * x_2 + x_3 * x_4$

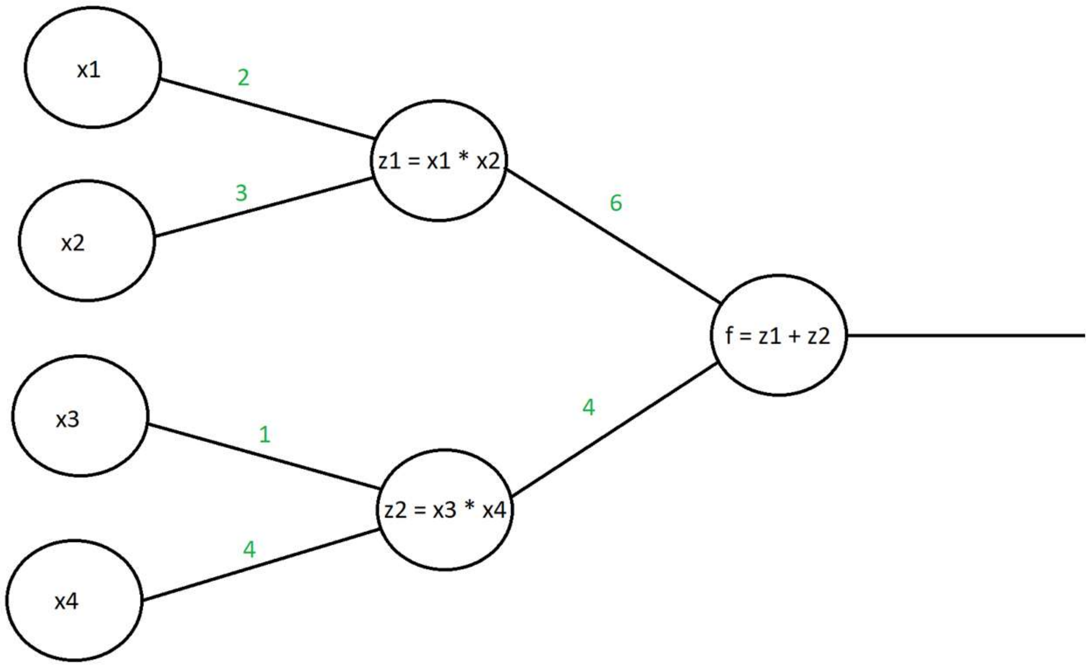
$f(\vec{x}) = z_1 + z_2$



Forward propagation

$f(\vec{x}) = x_1 * x_2 + x_3 * x_4$

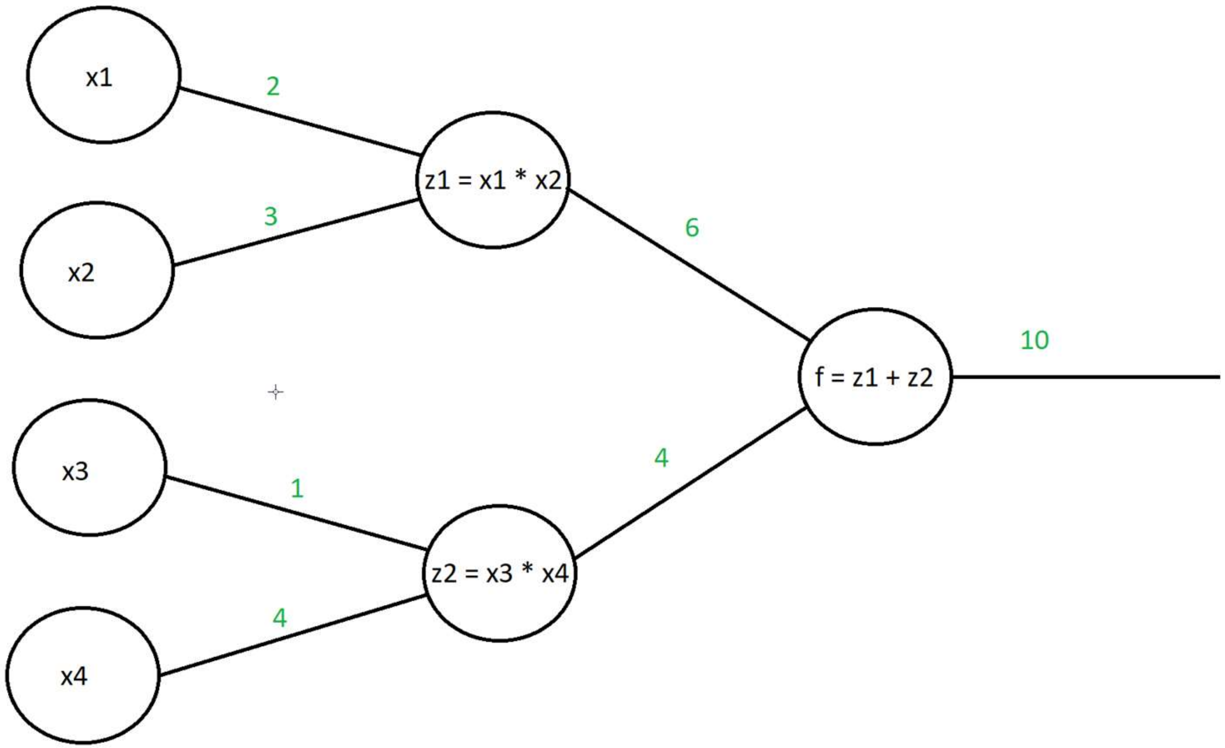
$f(\vec{x}) = z_1 + z_2$



Forward propagation

$f(\vec{x}) = x_1 * x_2 + x_3 * x_4$

$f(\vec{x}) = z_1 + z_2$



Backward propagation

- What if we want to get the derivative of f with respect to the different x values?

$$f(\vec{x}) = x_1 * x_2 + x_3 * x_4$$

$$z_1 = x_1 * x_2$$
$$z_2 = x_3 * x_4$$

$$\frac{\partial f(\vec{x})}{\partial x_1} = \frac{\partial f}{\partial z_1} \frac{\partial z_1}{\partial x_1} = x_2$$

$$\frac{\partial f(\vec{x})}{\partial x_3} = \frac{\partial f}{\partial z_2} \frac{\partial z_2}{\partial x_3} = x_4$$

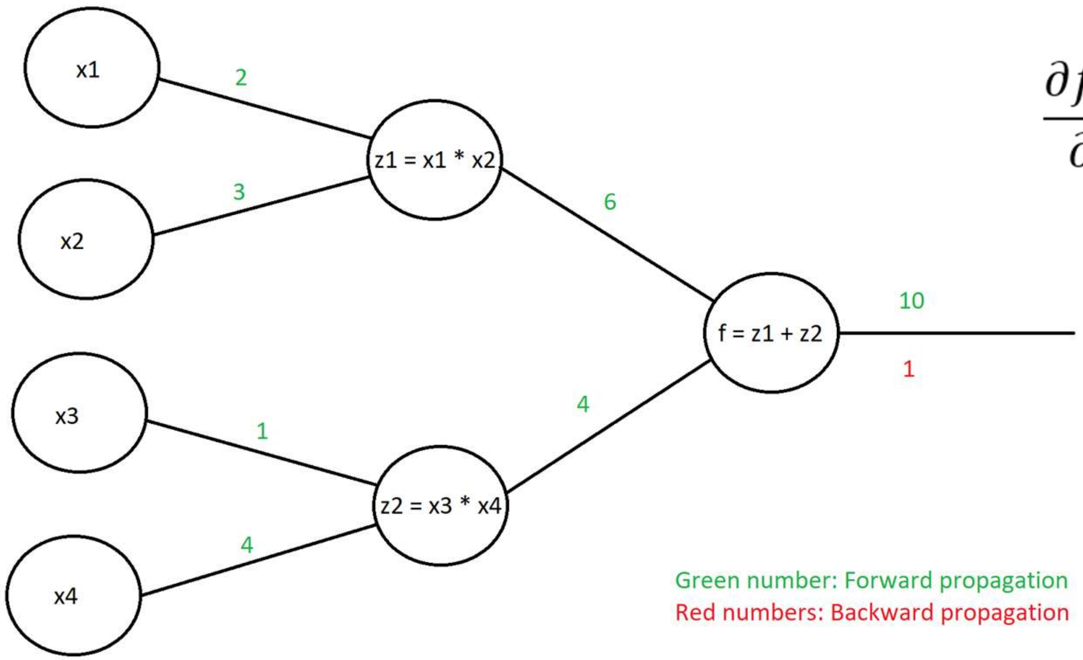
Backward propagation

- Lets take the derivative of f with respect to x1

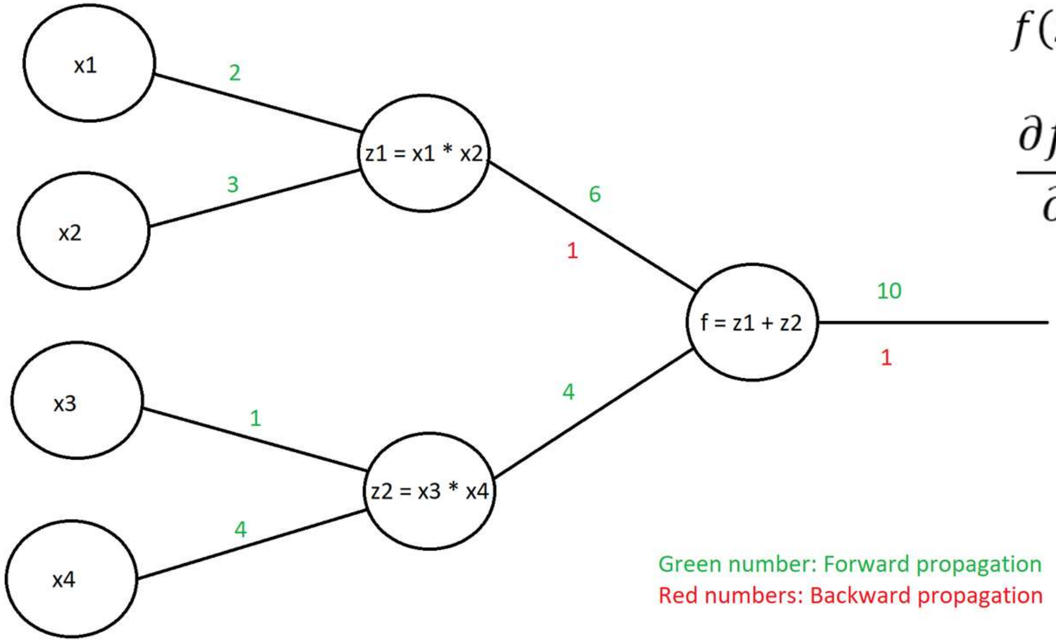
$$f(\vec{x}) = x_1 * x_2 + x_3 * x_4$$

$$f(\vec{x}) = z_1 + z_2$$

$$\frac{\partial f(\vec{x})}{\partial x_1} = \frac{\partial f}{\partial z_1}\frac{\partial z_1}{\partial x_1} = x_2$$



Backward propagation



$$f(\vec{x}) = x_1 * x_2 + x_3 * x_4$$

$$f(\vec{x}) = z_1 + z_2$$

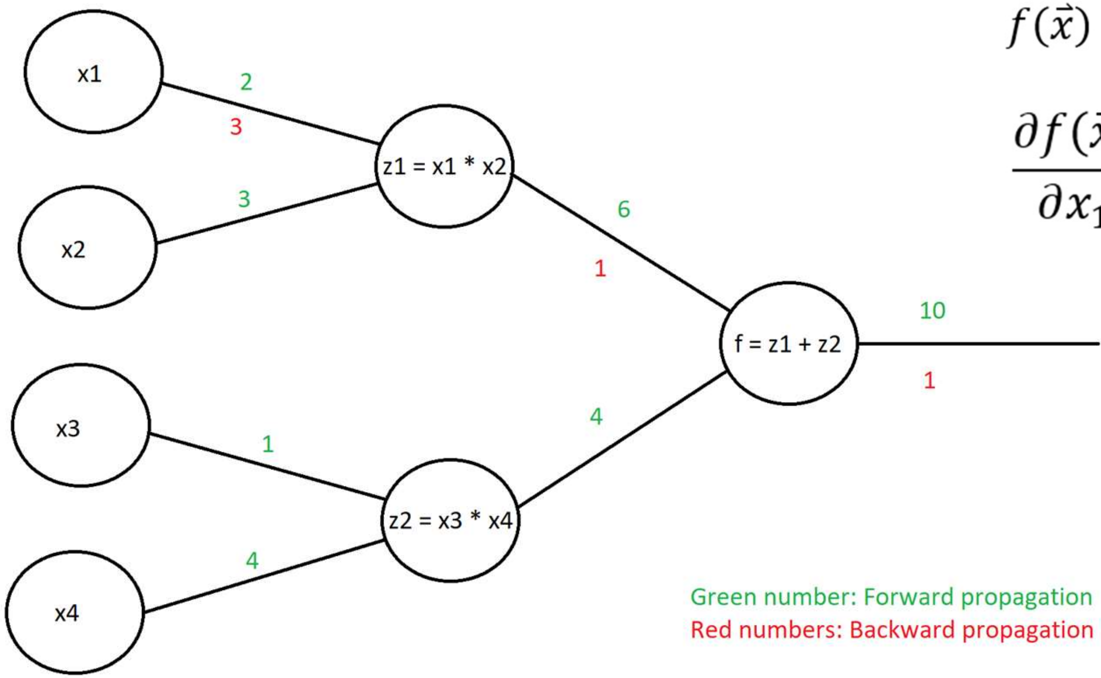
$$\frac{\partial f(\vec{x})}{\partial x_1} = \frac{\partial f}{\partial z_1}\frac{\partial z_1}{\partial x_1} = x_2$$

Backward propagation

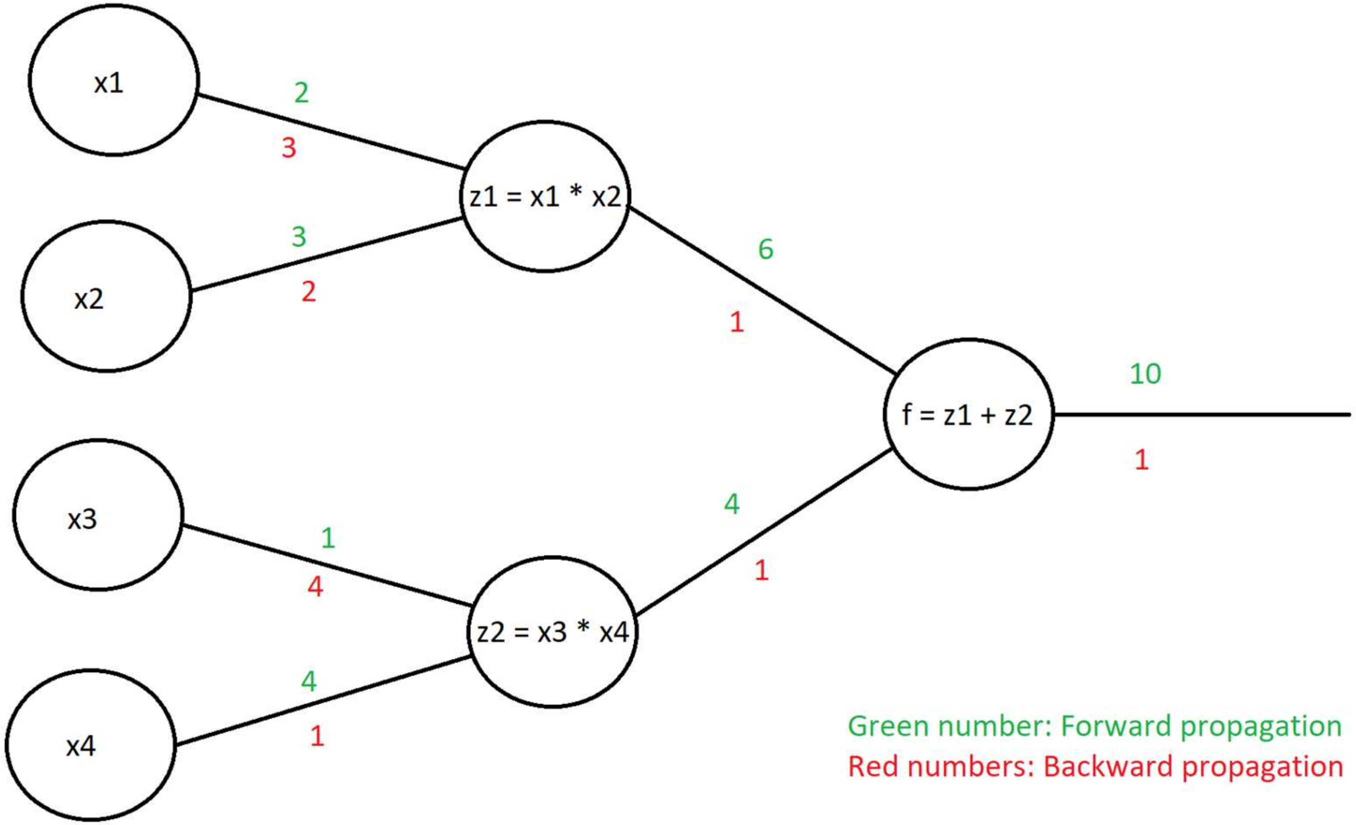
$$f(\vec{x}) = x_1 * x_2 + x_3 * x_4$$

$$f(\vec{x}) = z_1 + z_2$$

$$\frac{\partial f(\vec{x})}{\partial x_1} = \frac{\partial f}{\partial z_1}\frac{\partial z_1}{\partial x_1} = x_2$$



Backward propagation



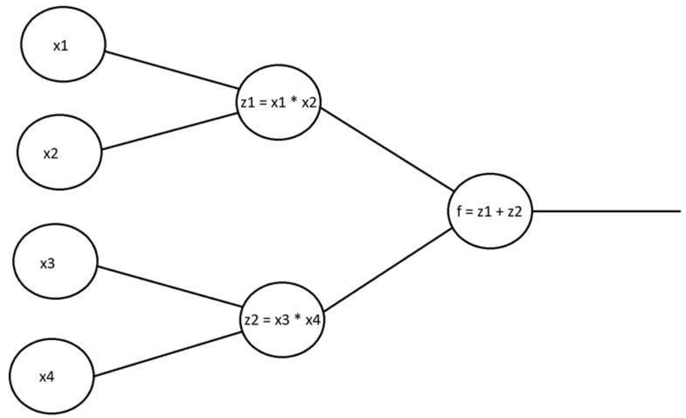
Progress

- Deep learning frameworks
- Pytorch
  - torch.tensor
  - Computational graph
  - **Automatic differentiation (torch.autograd)**
  - Data loading and preprocessing (torch.utils)
  - Useful functions (torch.nn.functional)
  - Creating the model (torch.nn)
  - Optimizers (torch.optim)
  - Save/load models
- Miscellaneous

Autograd

- Autograd – Automatic differentiation for all operations on Tensors
  - Static computational graph (TensorFlow)
  - Dynamic computational graph (Pytorch)

- The backward graph is defined by the forward run!



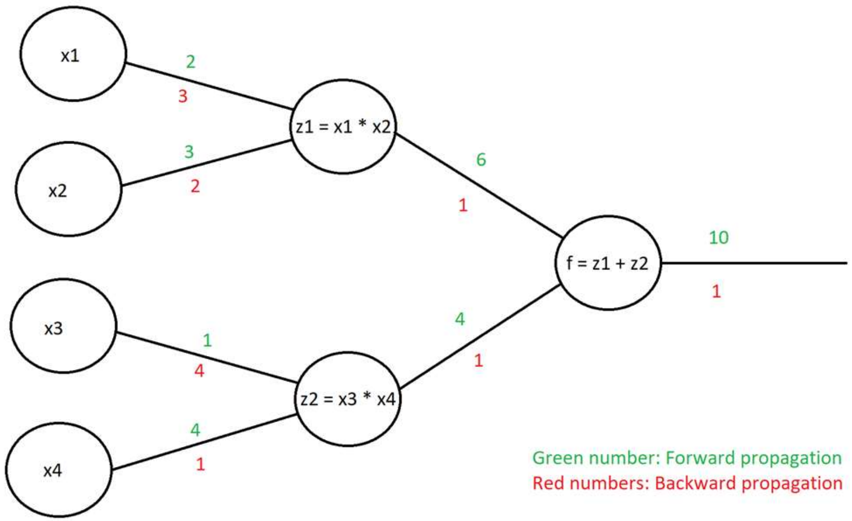
Example 1 (autograd)

In [1]:
```
import torch
from torch.autograd import grad

x1 = torch.tensor(2, requires_grad=True, dtype=torch.float32)
x2 = torch.tensor(3, requires_grad=True, dtype=torch.float32)
x3 = torch.tensor(1, requires_grad=True, dtype=torch.float32)
x4 = torch.tensor(4, requires_grad=True, dtype=torch.float32)
```

In [2]:
```
z1 = x1*x2
z2 = x3*x4

f = z1 + z2

df_dx = grad(outputs=f, inputs=[x1, x2, x3, x4])
```

In [3]:
```
print(f'gradient of x1 = {df_dx[0]}')
print(f'gradient of x2 = {df_dx[1]}')
print(f'gradient of x3 = {df_dx[2]}')
print(f'gradient of x4 = {df_dx[3]}')
```

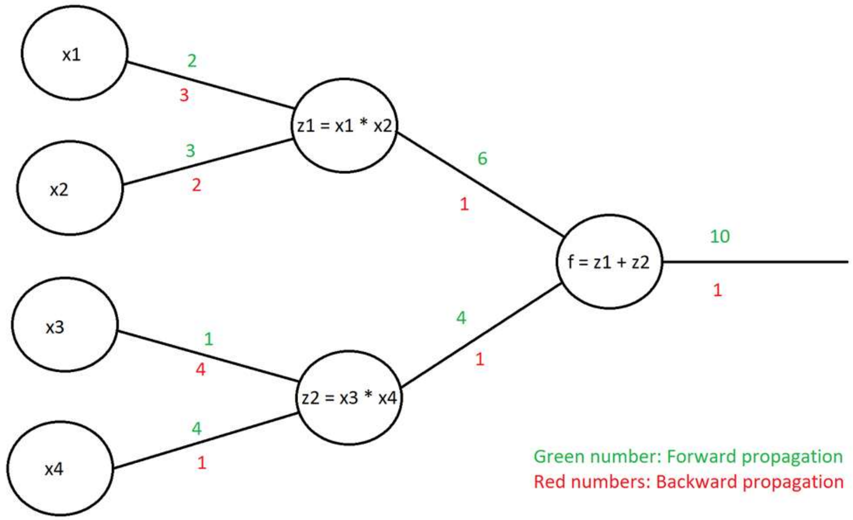
```
gradient of x1 = 3.0
gradient of x2 = 2.0
gradient of x3 = 4.0
gradient of x4 = 1.0
```



Leaf tensor

- A «leaf tensor» is a tensor you created directly, not as the result of an operation.

```
x = torch.tensor(2)    # A leaf tensor
y = x + 1              # Not a leaf tensor
```



Autograd

- The need for specifying all tensors is inconvenient

```
df_dx = grad(outputs=f, inputs=[x1, x2, x3, x4])
```

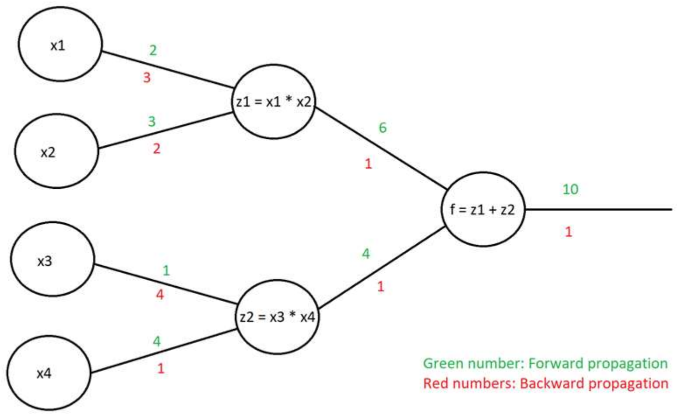
- We use “tensor.backward()” or “torch.autograd.backward()”

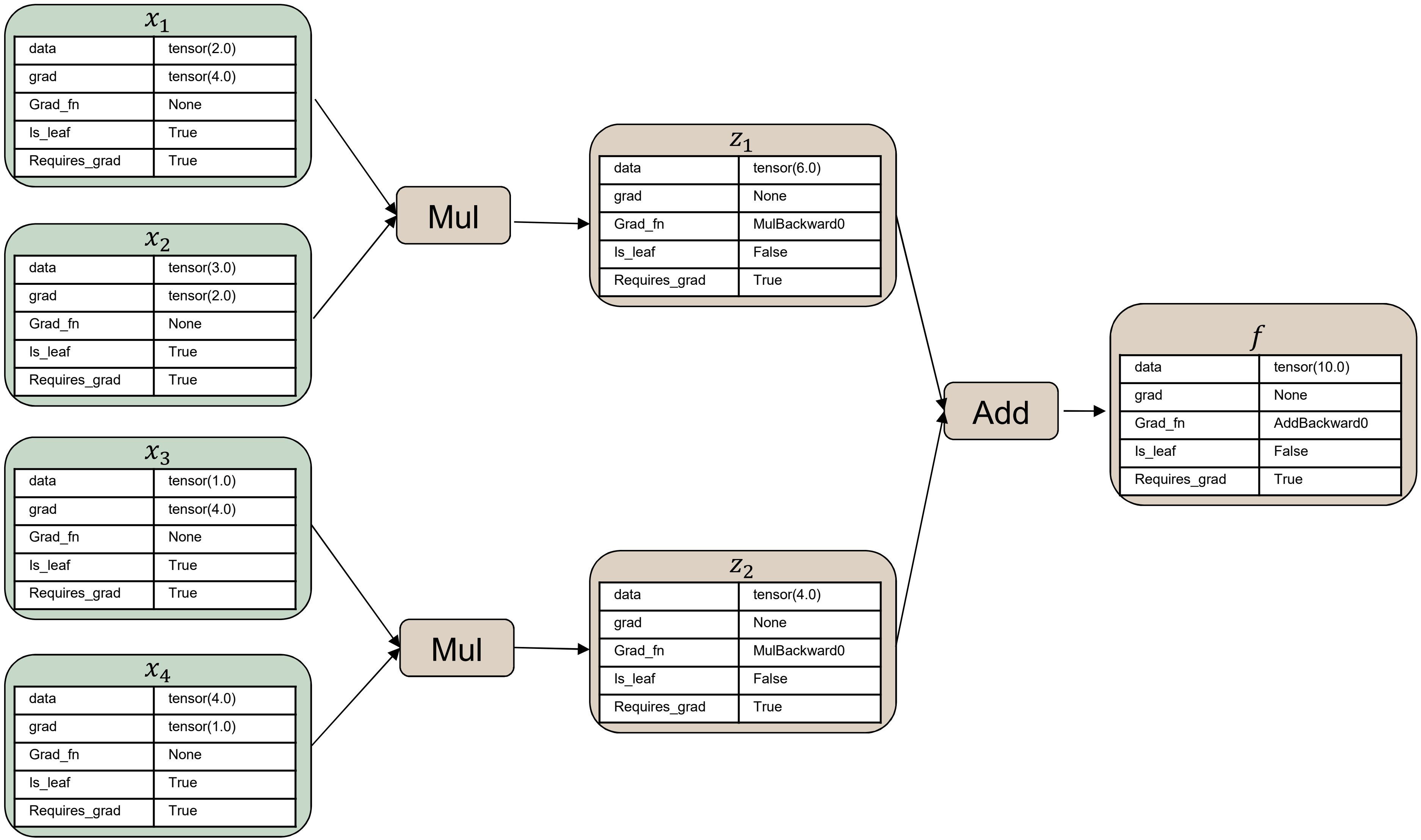
In [1]:
```
import torch
from torch.autograd import grad

x1 = torch.tensor(2, requires_grad=True, dtype=torch.float32)
x2 = torch.tensor(3, requires_grad=True, dtype=torch.float32)
x3 = torch.tensor(1, requires_grad=True, dtype=torch.float32)
x4 = torch.tensor(4, requires_grad=True, dtype=torch.float32)
```

```
z1 = x1*x2
z2 = x3*x4

f = z1 + z2
f.backward()
```

```
print(f'gradient of x1 = {x1.grad}')
print(f'gradient of x2 = {x2.grad}')
print(f'gradient of x3 = {x3.grad}')
print(f'gradient of x4 = {x4.grad}')
```

```
gradient of x1 = 3.0
gradient of x2 = 2.0
gradient of x3 = 4.0
gradient of x4 = 1.0
```




x_1
data | tensor(2.0)
grad | tensor(4.0)
Grad_fn | None
Is_leaf | True
Requires_grad | True
x_2
data | tensor(3.0)
grad | tensor(2.0)
Grad_fn | None
Is_leaf | True
Requires_grad | True
x_3
data | tensor(1.0)
grad | tensor(4.0)
Grad_fn | None
Is_leaf | True
Requires_grad | True
x_4
data | tensor(4.0)
grad | tensor(1.0)
Grad_fn | None
Is_leaf | True
Requires_grad | True
Mul
Mul
z_1
data | tensor(6.0)
grad | None
Grad_fn | MulBackward0
Is_leaf | False
Requires_grad | True
z_2
data | tensor(4.0)
grad | None
Grad_fn | MulBackward0
Is_leaf | False
Requires_grad | True
Add
f
data | tensor(10.0)
grad | None
Grad_fn | AddBackward0
Is_leaf | False
Requires_grad | True

Autograd in depth (optional)

- https://www.youtube.com/watch?v=MswxJw-8PvE

- https://pytorch.org/docs/stable/autograd.html#torch.autograd.grad

Context managers

- We can locally disable/enable gradient calculation
  - torch.no_grad()
  - torch.enable_grad()

```
In [2]: x = torch.tensor([1.0], requires_grad=True)
        y = x * 2
        y.requires_grad

Out[2]: True
```

```
In [3]: x = torch.tensor([1.0], requires_grad=True)
        with torch.no_grad():
            y = x * 2
        y.requires_grad

Out[3]: False
```

```
In [4]: x = torch.tensor([1.0], requires_grad=True)
        with torch.no_grad():
            with torch.enable_grad():
                y = x * 2
        y.requires_grad

Out[4]: True
```

Note: Use «torch.no_grad()» during inference

Example 2 - Solving a linear problem

In [2]:
```
a_ref = -1.5
b_ref = 8
noise = 0.2*np.random.randn(50)
```

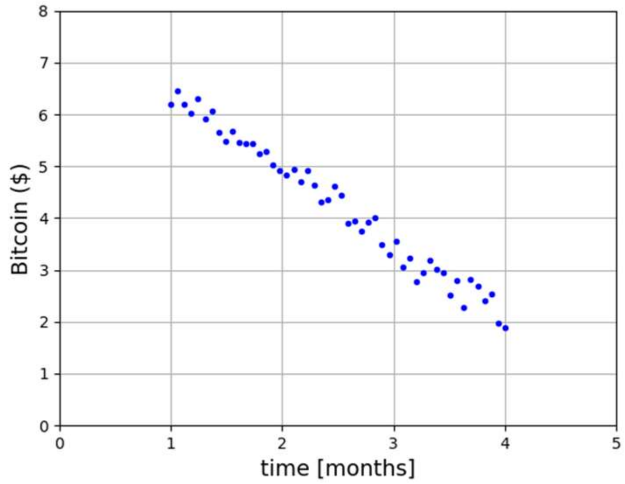
In [3]:
```
#Generat data
x = np.linspace(1,4, 50)
y = a_ref * x + b_ref + noise
```

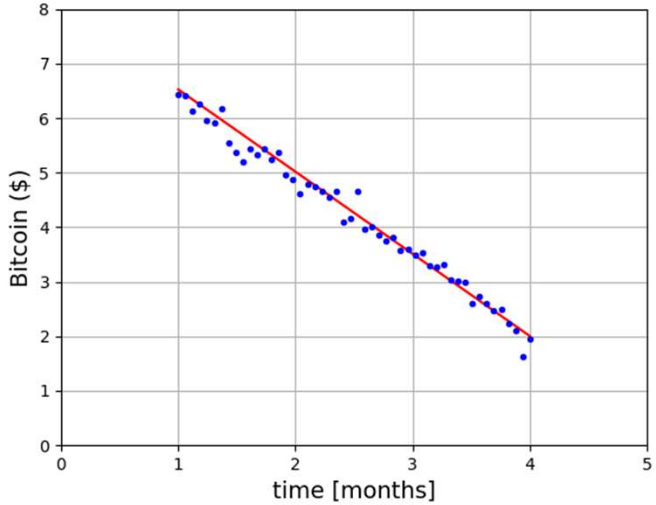
In [4]:
```
def MSE_loss(pred, label):
    return (pred-label).pow(2).mean()
```

In [5]:
```
#Get data as torch.tensors
xx = torch.tensor(x, dtype=torch.float32)
yy = torch.tensor(y, dtype=torch.float32)

#Create our unknown variables
a = torch.tensor(0, requires_grad=True, dtype=torch.float32)
b = torch.tensor(5, requires_grad=True, dtype=torch.float32)
```

In [6]:
```
# training Loop
numbOfEpoch   = 10000
learning_rate = 0.01
for ii in range(numbOfEpoch):
    y_pred = a * xx + b
    loss   = MSE_loss(pred=y_pred, label=yy)
    loss.backward()

    # Gradient descent update
    with torch.no_grad():
        a = a - learning_rate * a.grad
        b = b - learning_rate * b.grad

    a.requires_grad = True
    b.requires_grad = True
    #print(f'ii = {ii} | a = {a:.2f} | b = {b:.2f}')

print(a)
print(b)
```

```
tensor(-1.5061, requires_grad=True)
tensor(8.0354, requires_grad=True)
```





Other useful torch.tensor functions

- If you want to detach a tensor from the graph, you can use «**detach()**»

```
In [2]: x = torch.tensor(2.5, requires_grad=True)
        x

Out[2]: tensor(2.5000, requires_grad=True)

In [3]: y = x.detach()
        y

Out[3]: tensor(2.5000)
```

- If you want to get a python number from a tensor, you can use «**item()**»

```
In [6]: x = torch.tensor(2.5)
        x

Out[6]: tensor(2.5000)

In [7]: x.item()

Out[7]: 2.5
```

Progress

- Deep learning frameworks
- Pytorch
  - torch.tensor
  - Computational graph
  - Automatic differentiation (torch.autograd)
  - **Data loading and preprocessing (torch.utils)**
  - Useful functions (torch.nn.functional)
  - Creating the model (torch.nn)
  - Optimizers (torch.optim)
  - Save/load models
- Miscellaneous

Data loading and preprocessing

- The «torch.utils.data» package have two useful classes for loading and preprocessing data:
  - torch.utils.data.Dataset
  - torch.utils.data.DataLoader

- For more information visit:

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

torch.utils.data.Dataset

- Typical structure of the Dataset class

```
import torch
import CatDataset
```

```
dataPath = 'data/imagesOfCats/'
myCatDataset = CatDataset(dataPath)
```

```
#Iterating through the dataset
for sample in myCatDataset:
    sample
```

```
import glob
from torch.utils.data import Dataset

class CatDataset(Dataset):
    def __init__(self, dataPath):

        self.listOfPaths = glob.glob(dataPath)
        return

    def __len__(self):
        """
        :return: The total number of samples
        """
        return len(self.listOfPaths)

    def __getitem__(self, index):
        """
        Args:
            index (int): Index

        Returns:
            tuple: (image, target) where target is index of the target class.
        """
        data, target = load(self.listOfPaths[indes])

        #Do some augmentation
        data = crop_data(data)

        return img, target
```

torch.utils.data.DataLoader

- «DataLoader» is used to:
  - Batching the dataset
  - Shuffling the dataset
  - Utilizing multiple CPU cores/ threads

```
import torch
import CatDataset
from torch.utils.data import DataLoader

dataPath = 'data/imagesOfCats/'
myCatDataset = CatDataset(dataPath)
train_loader = DataLoader(dataset=myCatDataset, batch_size=32,
                                  shuffle=False, num_workers=2)
```

```
#Iterating through the dataset
for batch_of_samples in train_loader:
    batch_of_samples
```

Progress

- Deep learning frameworks
- Pytorch
  - torch.tensor
  - Computational graph
  - Automatic differentiation (torch.autograd)
  - Data loading and preprocessing (torch.utils)
  - **Useful functions (torch.nn.functional)**
  - Creating the model (torch.nn)
  - Optimizers (torch.optim)
  - Save/load models
- Miscellaneous

torch.nn.functional

- The «torch.nn.functional» package includes many useful functions, some of them are listed in the table below.

Activation functions	Layers	Loss functions
functional.relu	functional.conv2d	functional.binary_cross_entropy
functional.sigmoid	functional.linear	functional.cross_entropy
functional.tanh	functional.batch_norm	functional.kl_div
functional.leaky_relu	functional.embedding	functional.l1_loss
functional.softmax	functional.dropout	functional.mse_loss

Progress

- Deep learning frameworks
- Pytorch
  - torch.tensor
  - Computational graph
  - Automatic differentiation (torch.autograd)
  - Data loading and preprocessing (torch.utils)
  - Useful functions (torch.nn.functional)
  - **Creating the model (torch.nn)**
  - Optimizers (torch.optim)
  - Save/load models
- Miscellaneous

Creating the model

- A model is of a nn.Model class type. A model can contain other models. E.g. we can create the class “Model” based on the stacking nn.Modules of type nn.Linear()
- The nn.Module’s weights as called “Parameters”, and are similar to tensors with “requires_grad=True”
- A nn.Module consists of an initialization of the Parameters and a forward function.

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

```
class Model(nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.fc1 = nn.Linear(in_features=28*28, out_features=128, bias=True)
        self.fc2 = nn.Linear(in_features=128, out_features=64, bias=True)
        self.fc3 = nn.Linear(in_features=64, out_features=10, bias=True)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

nn.Module’s member functions

- Access information of a model:

```
In [3]:  model =  Model()
         list(model.children())

Out[3]: [Linear(in_features=784, out_features=128, bias=True),
         Linear(in_features=128, out_features=64, bias=True),
         Linear(in_features=64, out_features=10, bias=True)]
```

```
In [15]: for key, value in model.state_dict().items():
             print(f'layer = {key:10s} | feature shape = {value.shape}')

layer = fc1.weight | feature shape = torch.Size([128, 784])
layer = fc1.bias   | feature shape = torch.Size([128])
layer = fc2.weight | feature shape = torch.Size([64, 128])
layer = fc2.bias   | feature shape = torch.Size([64])
layer = fc3.weight | feature shape = torch.Size([10, 64])
layer = fc3.bias   | feature shape = torch.Size([10])
```

nn.Module's member functions

- Layers as e.g. "dropout" and "batch_norm" should operate differently during training and evaluation of the model. We can set the model in different state by the following functions.

```
In [7]: model.train()
        model.eval()
```

- # Model parameters can be sent to the CPU/GPU similar as to torch.tensors

```
In [ ]: model.to('CPU')
```

Progress

- Deep learning frameworks
- Pytorch
  - torch.tensor
  - Computational graph
  - Automatic differentiation (torch.autograd)
  - Data loading and preprocessing (torch.utils)
  - Useful functions (torch.nn.functional)
  - Creating the model (torch.nn)
  - **Optimizers (torch.optim)**
  - Save/load models
- Miscellaneous

Define an optimizer and train the model

- Using Pytorch's optimizers is easy!

```
In [ ]: optimizer = optim.SGD(model.parameters(), lr = 0.01)
```

```
In [ ]: for input, target in dataset:
            optimizer.zero_grad()
            output = model(input)
            loss = loss_fn(output, target)
            loss.backward()
            optimizer.step()
```

List of optimization algorithms
SGD
ADAM
RMSprop
Adagrad

Progress

- Deep learning frameworks
- Pytorch
  - torch.tensor
  - Computational graph
  - Automatic differentiation (torch.autograd)
  - Data loading and preprocessing (torch.utils)
  - Useful functions (torch.nn.functional)
  - Creating the model (torch.nn)
  - Optimizers (torch.optim)
  - **Save/load models**
- Miscellaneous

Save/load models

- Saving and loading can easily be don using “torch.save” and “torch.load”
- Pytorch uses “pickling” to serialize the data

```
In [ ]: stored_obj = {
                 'model_state_dict': model.state_dict(),
                 'optimizer_state_dict': optimizer.state_dict(),
        }

        torch.save(stored_obj, 'fileName.pt')
```

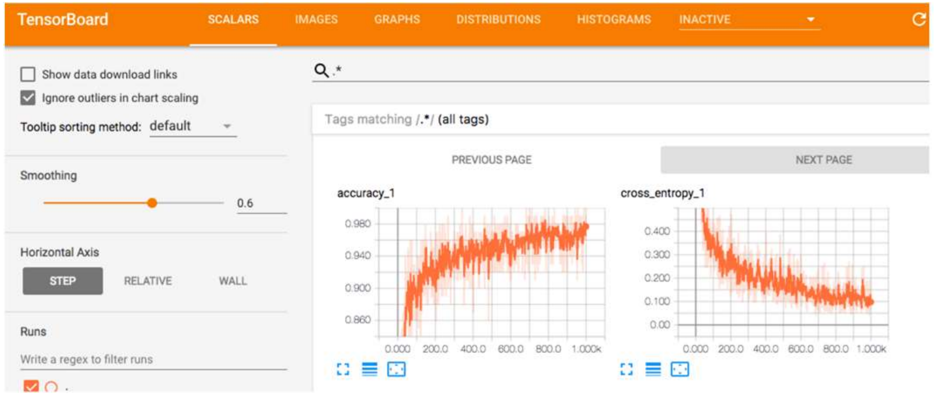
```
In [ ]: model     = Model()
        optimizer = optim.SGD(model.parameters(), lr = 0.01)

        checkpoint = torch.load('fileName.pt')
        model.load_state_dict(checkpoint['model_state_dict'])
        optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
```

Progress

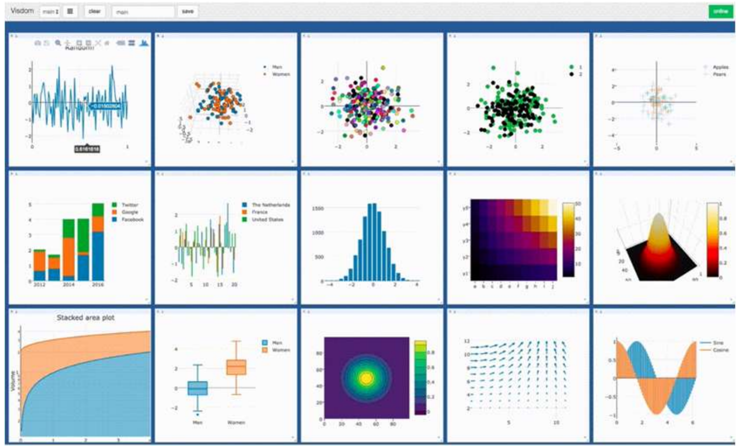
- Deep learning frameworks
- Pytorch
  - torch.tensor
  - Computational graph
  - Automatic differentiation (torch.autograd)
  - Data loading and preprocessing (torch.utils)
  - Useful functions (torch.nn.functional)
  - Creating the model (torch.nn)
  - Optimizers (torch.optim)
  - Save/load models
- **Miscellaneous**

Visualization

- Tensorboard:



- Visdom:



Installing Pytorch

Visit: https://pytorch.org/get-started/locally/

Verifying the install

```
In [1]: import torch

In [2]: print(torch.__version__)

        0.4.1

In [3]: torch.cuda.is_available()
Out[3]: True

In [4]: torch.version.cuda
Out[4]: '9.0'
```

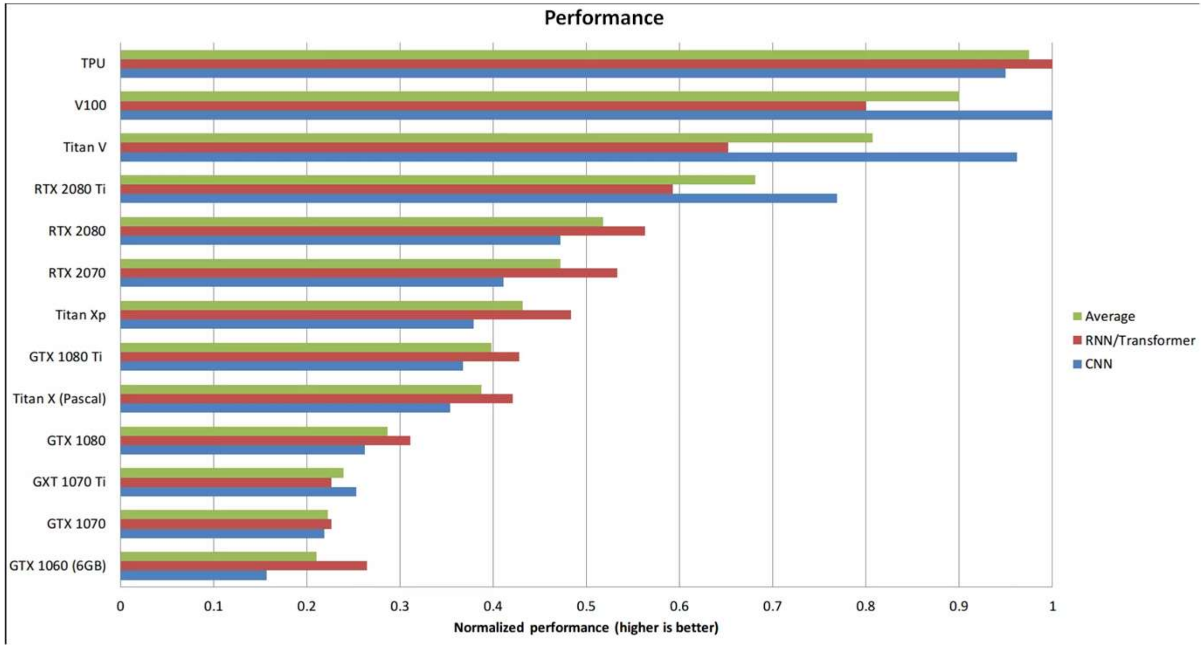
Use of CPU and GPU

- CPU only:
  - Preprosessing
  - Training
  - Evaluation

- CPU and GPU
  - CPU: preprosessing
  - GPU: Training
  - GPU: Evaluation

Building your own deep learning rig

- http://timdettmers.com/2018/12/16/deep-learning-hardware-guide/
- http://timdettmers.com/2018/11/05/which-gpu-for-deep-learning/



Device	Speed of training, examples/sec
2 x AMD Opteron 6168	440
i7-7500U	415
GeForce 940MX	1190
GeForce 1070	6500

https://medium.com/@andriylazorenko/tensorflow-performance-test-cpu-vs-gpu-79fcd39170c