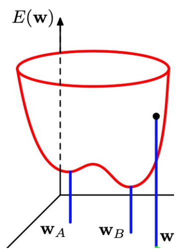
Deep Learning: differentiable programming

Christian Wolf

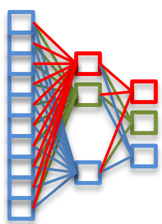
July 8th, 2021

Content

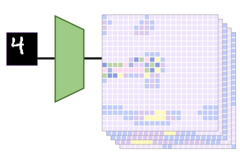
DL Frameworks and Tensors



Automatic differentiation

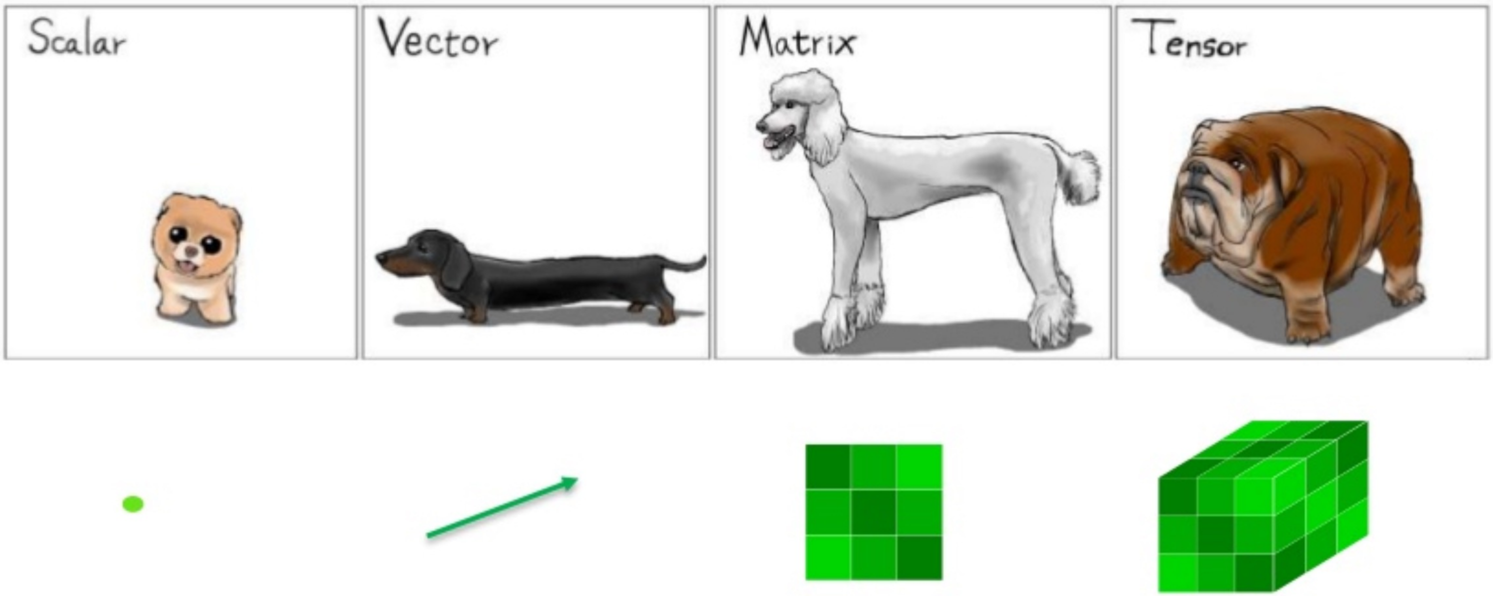


Example : MLP for MNIST



Exercise : CNN for MNIST

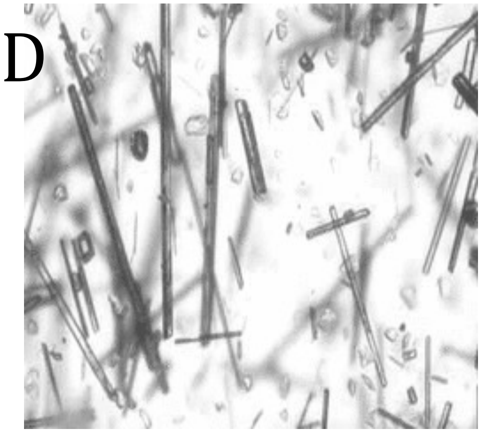
We manipulate tensors

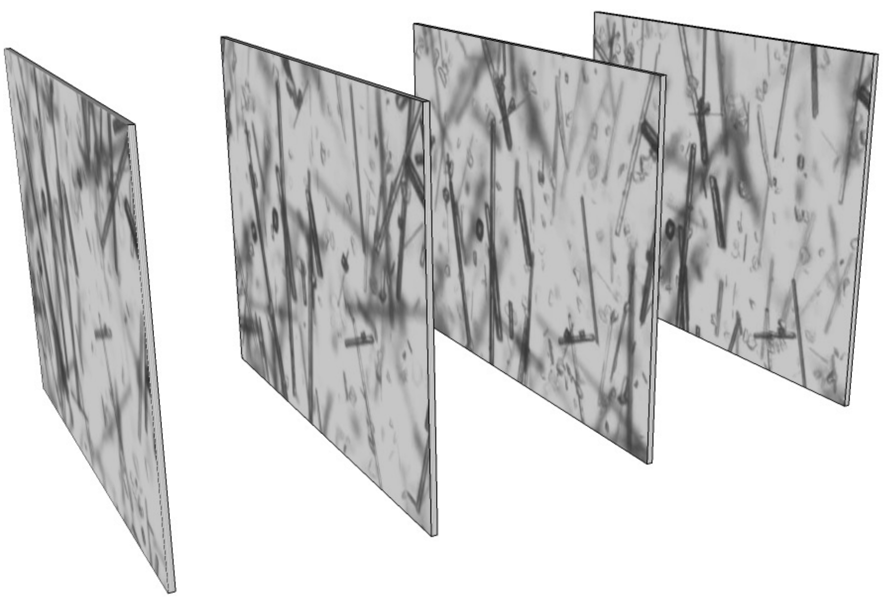


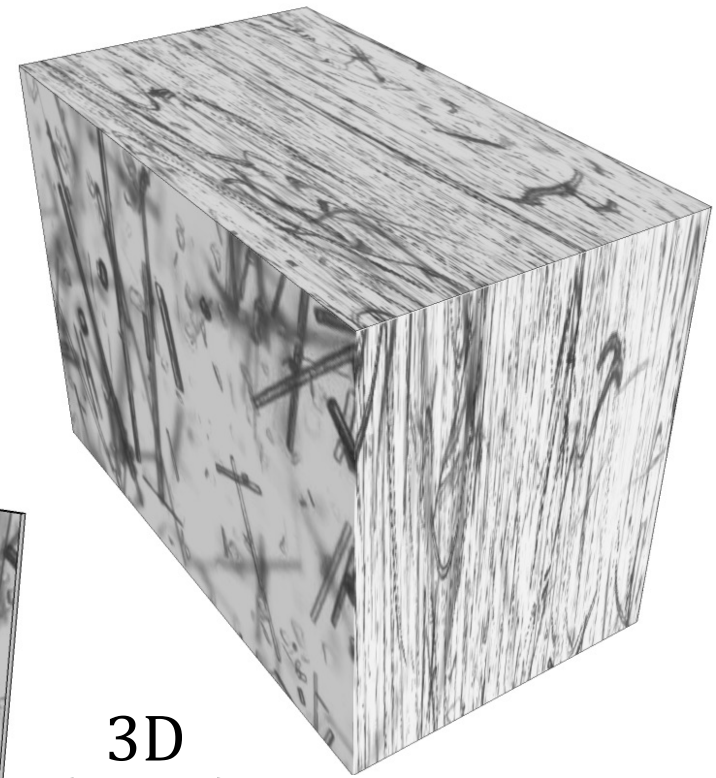
[Figure: Anima Anandkumar]

High dimensional tensors

2D





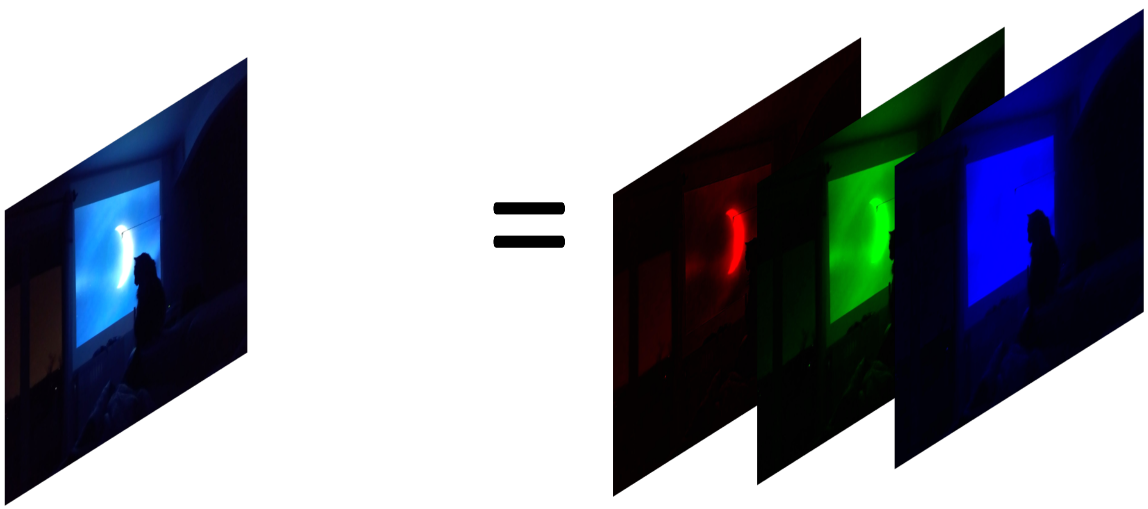


3D
(2D+t)

[Data: LAGEP]

Images as tensors

A color image has 3 color channels (red, green, blue) and is therefore a 3D tensor.



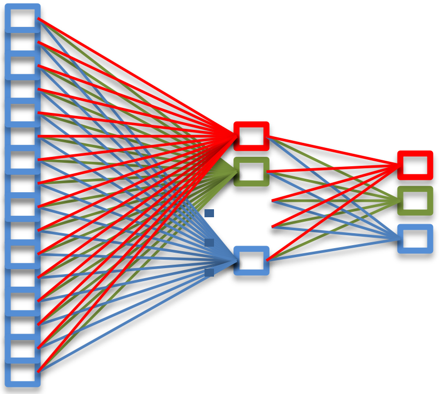
3D 3D

Tensors: examples

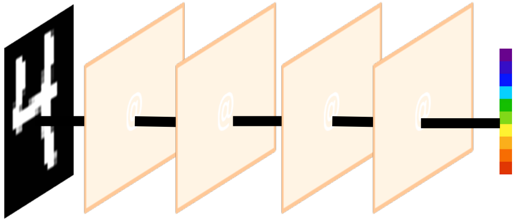
- Example of a tensor of dim 2 (input data, 1D signal)
  - Batch dimension (multipe samples)
  - Signal dimension

- Example of a tensor of dim 2 (output data, classification)
  - Batch dimension (multipe samples)
  - Prediction for different classes

- Example of a tensor of dim 3 (layer activation, 1D signal)
  - Batch dimension (multipe samples)
  - Signal dimension
  - Feature dimension



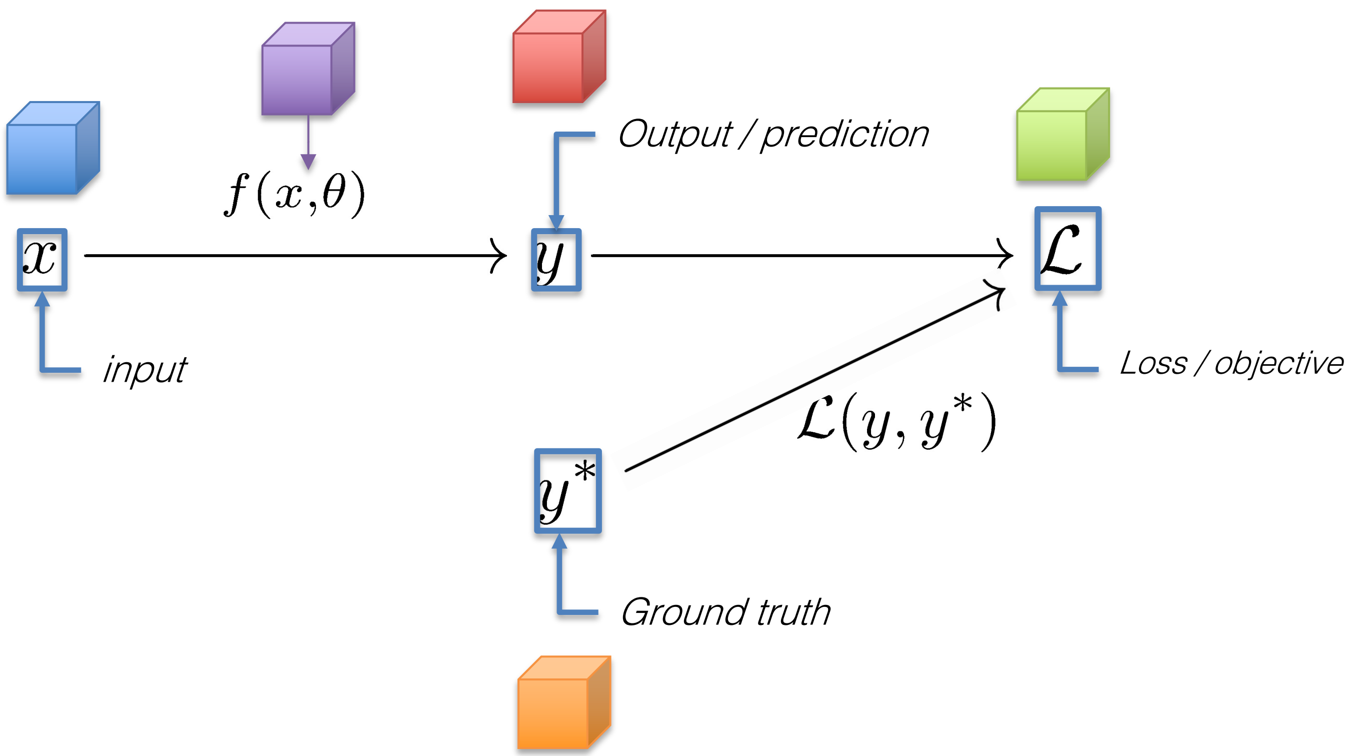
- Example of a tensor of dim 4 (layer activation, 2D image)
  - Batch dimension (multipe samples)
  - Spatial X dimension
  - Spatial Y dimension
  - Feature dimension



- Example of a tensor of dim 5 (input data, 2D+t video)
  - Batch dimension (multipe samples)
  - Spatial X dimension
  - Spatial Y dimension
  - Color channel dimension
  - Time dimension

Implementing a functional mapping

Inputs, outputs, layer activations, weights are tensors of different dimensions.



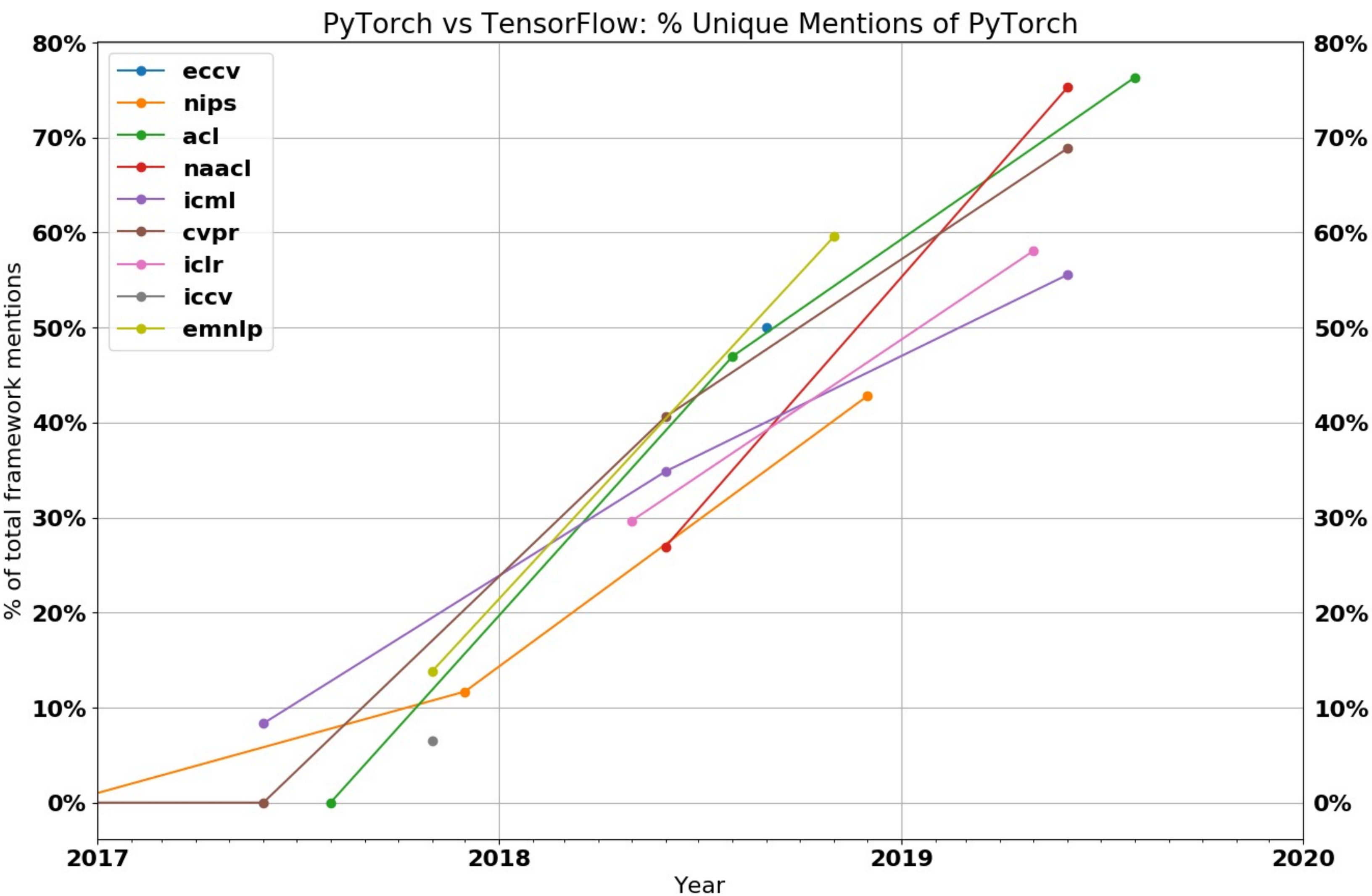
Main frameworks

Tensorflow

PyTorch

- Both support execution and training on CPUs, GPUs, TPUs (google's machine learning hardware)
- Both use python.
- Tensorflow also supports C++, Swift.

PyTorch vs. Tensorflow



https://thegradient.pub/state-of-ml-frameworks-2019-pytorch-dominates-research-tensorflow-dominates-industry/

Creating tensors

```
# loading PyTorch
import torch

# create with given shape
torch.full((shape), value)
torch.full_like(other_tensor,value)

# create with given values
torch.tensor((values))
torch.tensor((values), dtype=torch.int16)

# create from numpy array
torch.from_numpy(numpyArray)

# Create zeros or ones
torch.zeros((shape))
torch.zeros_like(other_tensor)
torch.ones((shape))
torch.ones_like(other_tensor)
```

Creating tensors, tensor I/O

```
# Random tensors
torch.randn(3, 4)

# Tensor I/O
A = torch.load ("A.tensor")
torch.save (A, "A.tensor")
```

Tensor slicing

Slicing is similar to python (NumPy) Slicing or Matlab notation.
Example for a 2D tensor:

```
A[1,5]                  # access an element (row, col)
A[:,5]                  # column access
A[1,:]                  # row access

A[1:6,:]    A[1:,:]     # range access (1:6 = 1,2,3,4,5)
A[:,0:-1]               # Negative index count backwards
                        # -1 = last col/row

A==3                    # provides a tensor of logical
                        # results

A[4:17,3] = B           # replace a slice

A[B==3]=4               # Set values in A to 4 at pos
                        # where there is a 3 in B
```

Manipulating tensors

```
# concatenate tensors
torch.cat((tensors), axis)

# split tensors into chunks of equal size
torch.split(tensor, splitSize, dim=0)

# reshape tensor w/o changing the data
torch.view(tensor, shape)

# Repeat along a given dimension
X.repeat(4,2)

# transpose tensor
torch.t(tensor) # 1D and 2D tensors
torch.transpose(tensor, dim0, dim1)

# Sorting
torch.sort(input, dim=-1)
```

Tensor math

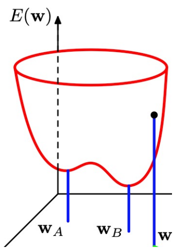
```
# Overloaded operators
x = A+Y*Z-B                   # * is elementwise mul

# Sum, product,  min, max  of all elements
torch.sum(tensor)   torch.min(tensor)
torch.prod(tensor)  torch.max(tensor)

# Linear algebra
torch.mm(A, B)                # Matrix multiplication
torch.inverse(tensor)         # Matrix inversion
torch.det(tensor)             # Determinant
```

Elementwise operations

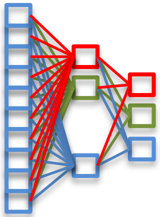
```
torch.exp(tensor)   torch.log(tensor)
torch.cos(tensor)   torch.cosh(tensor)
torch.sin(tensor)   torch.sinh(tensor)
torch.tan(tensor)   torch.tanh(tensor)

torch.add(tensor, tensor2) # or tensor+scalar
torch.div(tensor, tensor2) # or tensor/scalar
torch.mult(tensor,tensor2) # or tensor*scalar
torch.sub(tensor, tensor2) # or tensor-scalar
```

Content

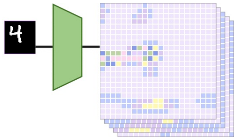
DL Frameworks and Tensors



Automatic differentiation



Example : MLP for MNIST



Exercise : CNN for MNIST

Gradient descent

One optimizer step:

$$\theta^{[t+1]} = \theta^{[t]} + \nu \nabla \mathcal{L}\left(h(x, \theta), y^*\right)$$

The gradient is a vector of partial derivatives:

$$\nabla \mathcal{L} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial \theta_0} \\ \\ \frac{\partial \mathcal{L}}{\partial \theta_1} \\ \\ \vdots \\ \\ \frac{\partial \mathcal{L}}{\partial \theta_N} \end{bmatrix}$$

Autograd

In PyTorch (and some other frameworks), `Autograd` performs automatic differentiation through a sequence of tensor instructions of an imperative language.

Let's consider a simple linear operation:

$$w = [\ 5\ 3\ ], \quad x = [\ 7\ 2\ ], \quad y = wx^T$$

The gradient of y w.r.t to x is given as

$$\nabla = \left[\frac{\partial y}{\partial x_i}\right] = \begin{bmatrix} 5 \\ 3 \end{bmatrix}$$

The gradient of y w.r.t to w is given as

$$\nabla = \left[\frac{\partial y}{\partial w_i}\right] = \begin{bmatrix} 7 \\ 1 \end{bmatrix}$$

Autograd

In PyTorch, we will first create the tensors:

```
w = torch.tensor([5, 3], dtype=float, requires_grad=True)
x = torch.tensor([7, 1], dtype=float, requires_grad=True)
```

The `requires_grad` flag ensures that all calculations are tracked. We perform the linear operation:

```
y = torch.dot(w,x)
```

Since the tensor y has been calculated as result of operations on tracked tensors, it has a gradient function:

```
print (y)
```

```
tensor(38., dtype=torch.float64, grad_fn=<DotBackward>)
```

Autograd

We now run a backward pass on the variable y, which calculates gradients w.r.t. to all involved tensors:

```
y.backward()
```

The gradients are attached to each variable:

```
print (x.grad)
print (w.grad)
```

```
tensor([5., 3.], dtype=torch.float64)
tensor([7., 1.], dtype=torch.float64)
```

Detaching tracking history

The tracking history uses memory in the tensor's space. If tracking is not used anymore for a tensor, it's tracking history can be detached:

```
print (y)
```

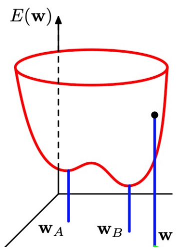
```
tensor(38., dtype=torch.float64, grad_fn=<DotBackward>)
```

```
z = y.detach()
print (z)
```

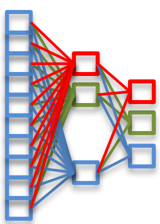
```
tensor(38., dtype=torch.float64)
```

Content

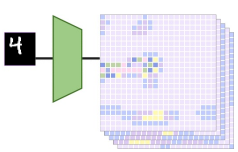
DL Frameworks and Tensors



Automatic differentiation



Example : MLP for MNIST

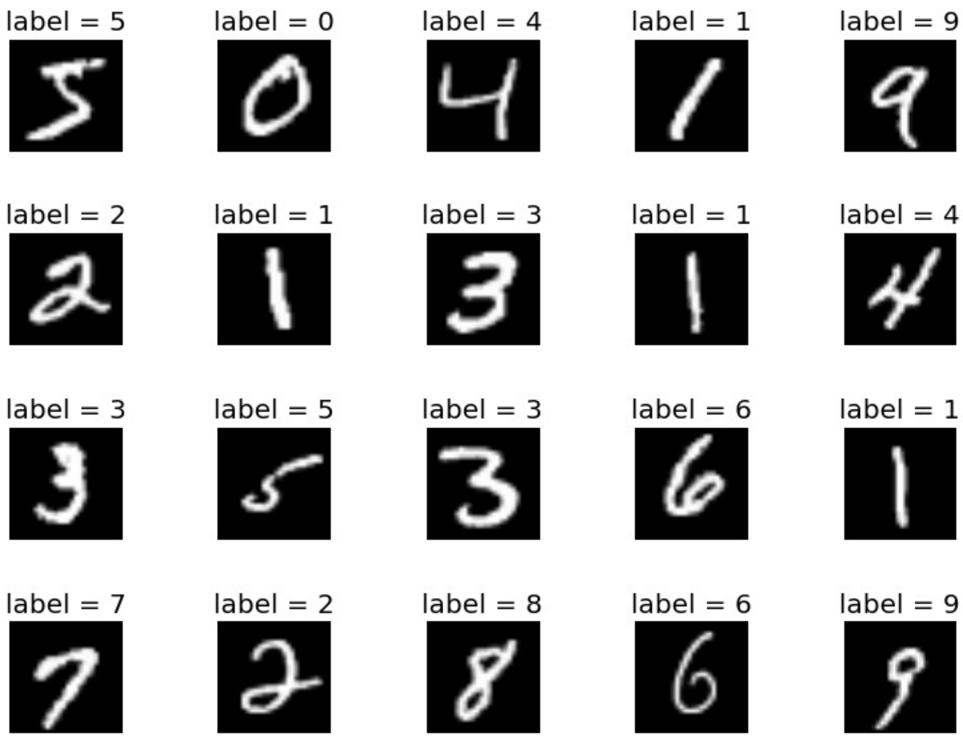


Exercise : CNN for MNIST

Example: the MNIST dataset

A dataset of handwritten digits introduced by Yann LeCun in 1999 with 60 000 training images and 10 000 test images.

One image is of size 28x28 pixels.



http://yann.lecun.com/exdb/mnist/

MNIST : MLP performance

linear classifier (1-layer NN)	none	12.0	LeCun et al. 1998
linear classifier (1-layer NN)	deskewing	8.4	LeCun et al. 1998
pairwise linear classifier	deskewing	7.6	LeCun et al. 1998
2-layer NN, 300 hidden units, mean square error	none	4.7	LeCun et al. 1998
2-layer NN, 300 HU, MSE, [distortions]	none	3.6	LeCun et al. 1998
2-layer NN, 300 HU	deskewing	1.6	LeCun et al. 1998
2-layer NN, 1000 hidden units	none	4.5	LeCun et al. 1998
2-layer NN, 1000 HU, [distortions]	none	3.8	LeCun et al. 1998
3-layer NN, 300+100 hidden units	none	3.05	LeCun et al. 1998
3-layer NN, 300+100 HU [distortions]	none	2.5	LeCun et al. 1998
3-layer NN, 500+150 hidden units	none	2.95	LeCun et al. 1998
3-layer NN, 500+150 HU [distortions]	none	2.45	LeCun et al. 1998

(Validation performance)

http://yann.lecun.com/exdb/mnist/

Writing Data Access

The MNIST dataset is supported by PyTorch through the `Dataset` class, which can even download the data from Yann Lecun's website:

```
train_dataset = datasets.MNIST("MNIST", True, download=True,
    transform=transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))])
    )
```

Writing Data Access

Let's recall training through gradient descent:

$$\theta^{[t+1]} = \theta^{[t]} + \nu \nabla \mathcal{L}(h(x, \theta), y^*)$$

The gradient is rarely (never?) taken over the whole dataset, but over a single sample, or batches (mini-batches) of a certain size. These batches are sample randomly from the dataset.

The actual shuffling and batching is performed by a built-in PyTorch `DataLoader` class, which uses an instance of our `Dataset` subclass:

```
train_dataset = datasets.MNIST("MNIST", True, download=True,
    transform=transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))])
    )

train_loader = torch.utils.data.DataLoader(train_dataset,
    batch_size=BATCHSIZE, shuffle=True)
```

Tensor dimension conventions

PyTorch functions operate on multi-dimensional tensors and follow conventions on the order of dimensions.

- The first dimension is the batch dimension
  - Use 1 if you don't use batches (= batches of size 1).
  - Losses are reduced (sum or mean) over samples in a batch
- the second dimension is the channel dimension
  - Use 1 if you don't use channels (= single channels).
  - Channel arithmetics will be explained in detail in the section on convolutions.
- the following dimensions are application dependant, e.g. rows, columns in images.

The model

```
class LogRegression(torch.nn.Module):
    def __init__(self):
        super(LogRegression, self).__init__()

        # input size to 10 output units
        self.fc1 = torch.nn.Linear(28*28, 10)

    def forward(self, x):
        # Reshape from a 3D tensor (batchsize, 28, 28)
      # to a flattened (batchsize, 28*28)
      # 1 sample = 1 vector
        x = x.view(-1, 28*28)
        return self.fc1(x)
```

Set up the environment

```
# Instantiate the model
model = LogRegression()

# This criterion combines LogSoftMax and NLLLoss in
    one single class.
crossentropy = torch.nn.CrossEntropyLoss()

# Set up the optimizer: stochastic gradient descent
# with a learning rate of 0.01
optimizer = torch.optim.SGD(model.parameters(), lr
    =0.01)

# Init some statistics
running_loss = 0.0
running_correct = 0
running_count = 0
```

Iterative training

```
# Cycle through epochs
for epoch in range(100):

    # Cycle through batches
    for batch_idx, (data, labels) in enumerate(
     train_loader):

        # clear the gradients (they are accumulated)
        optimizer.zero_grad()

        # perform a forward pass and calc the loss
        y = model(data)
        loss = crossentropy(y, labels)

        # backward pass: calculate the gradients
        loss.backward()

        # accumulate the loss, perform one SGD step
        running_loss += loss.item()
        optimizer.step()
```

Track training error

```
# Calculate the winner class
_, predicted = torch.max(y.data, 1)

# How many correct samples?
running_correct += (predicted == labels).sum().item()
running_count += BATCHSIZE

# Every 100 batches, print statistics
if (batch_idx % 100) == 0:

train_err = 100.0*(1.0-running_correct / running_count)
print ('Epoch: %d batch: %5d ' % (epoch + 1, batch_idx + 1), end="")
print ('train-loss: %.3f train-err: %.3f' % (running_loss / 100, train_err))
running_loss = 0.0
running_correct = 0.0
running_count=0.0
```

Logistic Regression: example output

```
Epoch: 1 batch:    1 train-loss: 0.026 train-err: 94.000
Epoch: 1 batch:  101 train-loss: 0.981 train-err: 25.020
Epoch: 1 batch:  201 train-loss: 0.538 train-err: 13.260
Epoch: 1 batch:  301 train-loss: 0.467 train-err: 12.700
Epoch: 1 batch:  401 train-loss: 0.425 train-err: 11.380
Epoch: 1 batch:  501 train-loss: 0.414 train-err: 11.820
Epoch: 1 batch:  601 train-loss: 0.384 train-err: 11.000
Epoch: 1 batch:  701 train-loss: 0.370 train-err: 10.180
Epoch: 1 batch:  801 train-loss: 0.374 train-err: 10.780
Epoch: 1 batch:  901 train-loss: 0.349 train-err: 9.480
Epoch: 1 batch: 1001 train-loss: 0.347 train-err: 9.840
Epoch: 1 batch: 1101 train-loss: 0.350 train-err: 9.920
Epoch: 2 batch:    1 train-loss: 0.347 train-err: 10.100
```

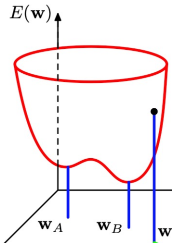
(...)

```
Epoch: 10 batch: 1001 train-loss: 0.277 train-err: 8.140
Epoch: 10 batch: 1101 train-loss: 0.277 train-err: 7.660
Epoch: 11 batch:    1 train-loss: 0.268 train-err: 7.700
```

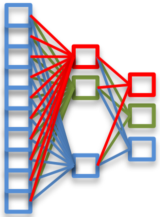
This is training error, not validation error, i.e. **NOT** representative of the performance of the model!

Content

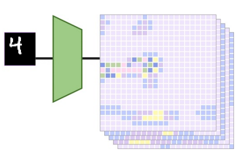
DL Frameworks and Tensors



Automatic differentiation



Example : MLP for MNIST



Exercise : CNN for MNIST

LeNet: example output

Exercise: create a convolution model similar to LeNet:

— One conv layer with $5{\times}5$ filters and 20 output channels.

— One conv layer with $5{\times}5$ filters and 50 output channels.

— One hidden layer with 500 units

— One output layer with 10 units ($=$digit classes)

— ReLU activation functions

— Max pooling $(2{\times}2)$ after each convolutional layer

Questions:

— How many filters does each layer have?

— How many trainable parameters does each layer have?

— What are the dimensions and sizes of the each output tensor?

LeNet: example output

```
Epoch: 1 batch:    1 train-loss: 0.024 train-err: 98.000
Epoch: 1 batch:  101 train-loss: 1.576 train-err: 36.140
Epoch: 1 batch:  201 train-loss: 0.747 train-err: 16.320
Epoch: 1 batch:  301 train-loss: 0.539 train-err: 12.880
Epoch: 1 batch:  401 train-loss: 0.481 train-err: 13.000
Epoch: 1 batch:  501 train-loss: 0.414 train-err: 11.280
Epoch: 1 batch:  601 train-loss: 0.386 train-err: 10.380
Epoch: 1 batch:  701 train-loss: 0.385 train-err: 10.900
Epoch: 1 batch:  801 train-loss: 0.363 train-err: 10.540
Epoch: 1 batch:  901 train-loss: 0.320 train-err: 9.120
Epoch: 1 batch: 1001 train-loss: 0.323 train-err: 8.920
Epoch: 1 batch: 1101 train-loss: 0.325 train-err: 9.400
Epoch: 2 batch:    1 train-loss: 0.304 train-err: 8.880
```

(...)

```
Epoch: 75 batch:  801 train-loss: 0.007 train-err: 0.000
Epoch: 75 batch:  901 train-loss: 0.007 train-err: 0.020
Epoch: 75 batch: 1001 train-loss: 0.008 train-err: 0.000
Epoch: 75 batch: 1101 train-loss: 0.009 train-err: 0.040
```

This is training error, not validation error, i.e. **NOT** representative of the performance of the model!