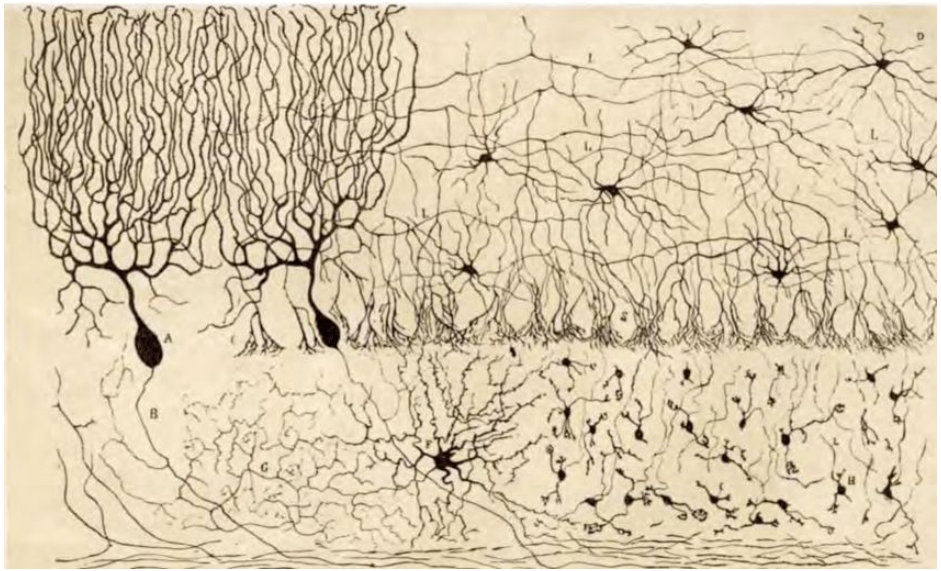
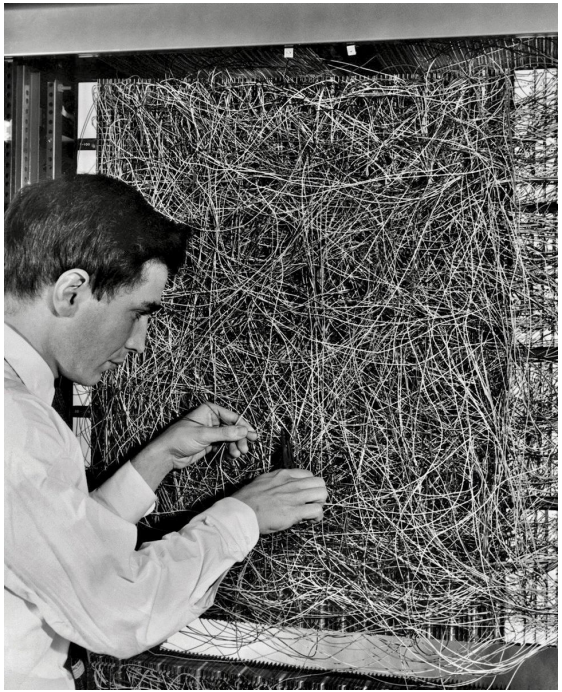
A very short history of Artificial Neural Networks/ Deep Learning



Chick
Cerebellum
Golgi Stain
Ramon y Cajal
Nobel: 1906

Linear learning in Perceptron: Mark I: 1958

$$y = \sigma\left(\mathbf{w}^t \mathbf{x}\right)$$



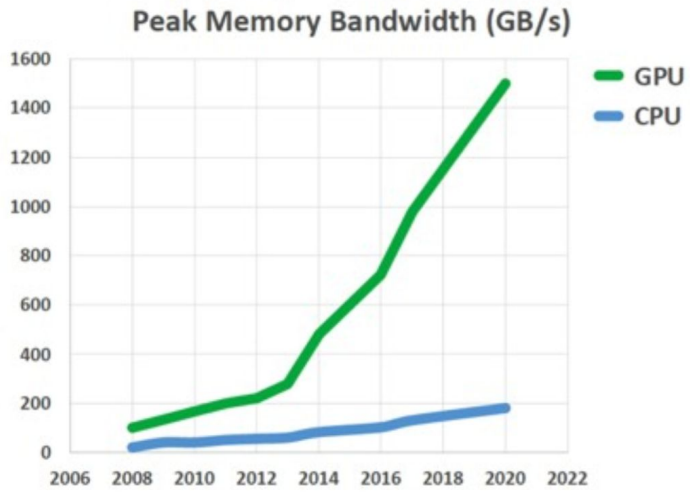
Rosenblatt
(source wikipedia)

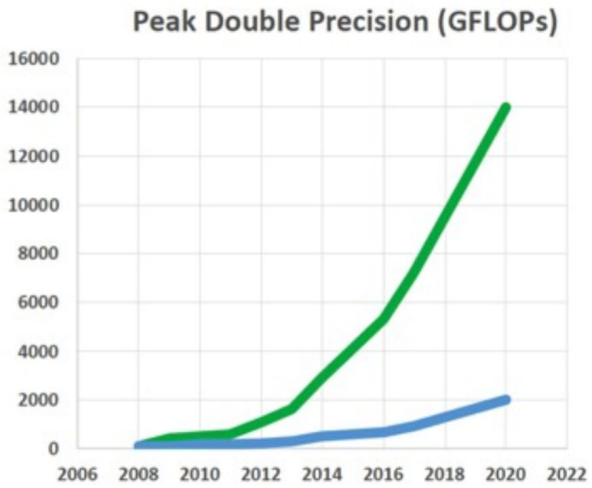
Nonlinear learning in multi-layer perceptrons

$$y = \sigma\left(w_2^t \sigma\left(w_1^t \mathbf{x}\right)\right)$$

...

GPUs





source:
https://www.nextplatform.com/2019/07/10/a-decade-of-accelerated-computing-augurs-well-for-gpus/

Lots of Data!

- 2005-2012: Pascal Visual Object Classes
  - 20 classes, 27.5k annotations (in most recent)
- 2010: ImageNet
  - 27 categories, 21.3k subcategories, ~1M images with annotations!
- 2014-2017 COCO
  - 80 classes, ~250k images with bounding boxes and segmentations
- 2017-2019 OpenImages
  - 9M images, 16M bounding boxes, 600 classes, 2.8M segmentation masks

... And lots of data augmentation! (e.g. rotation, crop, add noise, etc)

Today, Deep Learning matters

- State of the art in most domains of Machine Learning
  - Computer Vision
  - Natural Language Processing
  - Time Series Processing
  - Reinforcement Learning

Deep Learning (DL) is Machine Learning (ML)

- DL Inherits most of its ideas from regular ML
  - Regularization
  - Establishing success
  - Interface to reality
  - Statistics
  - Concepts

Face manipulations

Captioning

"man in black shirt is playing guitar."

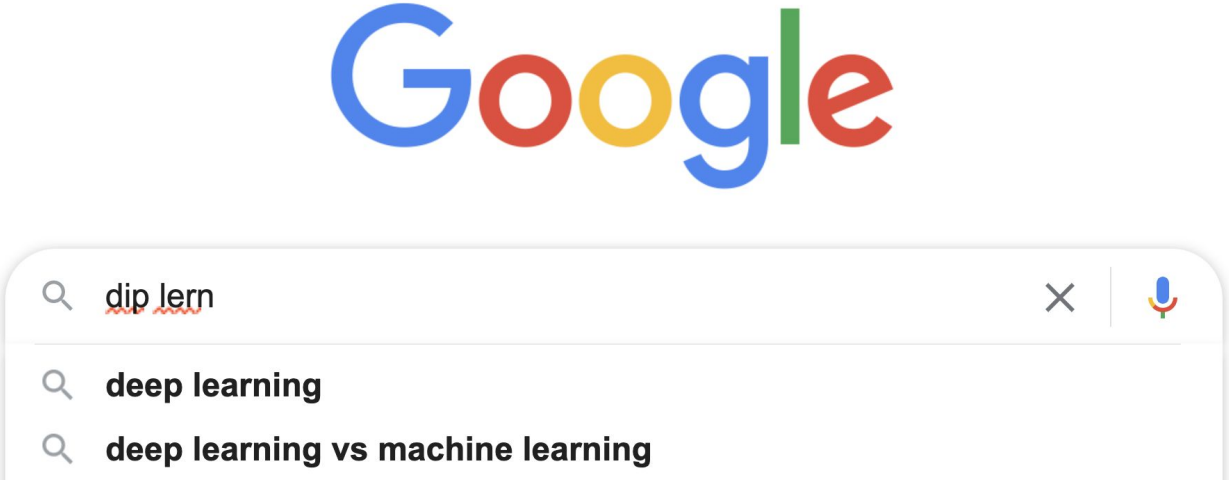
"construction worker in orange safety vest is working on road."

"two young girls are playing with lego toy."

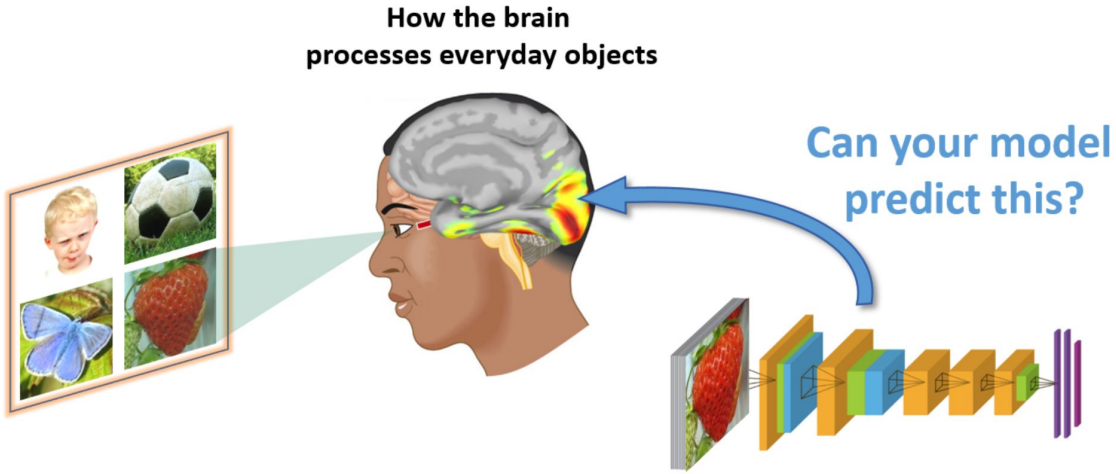
Alpha Zero


Tore verrore mperspedi cuptatur eta audicto p. XX
Quasi volut accae cullit ad queomni quis pp. XX & XX
Perspedi cuptatur audicto volut accae cullit p. XX
Science
$15
7 DECEMBER 2018
sciencemag.org
AAAS
31 63%
18 47%
16%
A DIGITAL
PRODIGY
AlphaZero teaches itself chess, shogi, and Go
pp. XXXX, XXXX, & XXXX

Actual google



Neuroscience



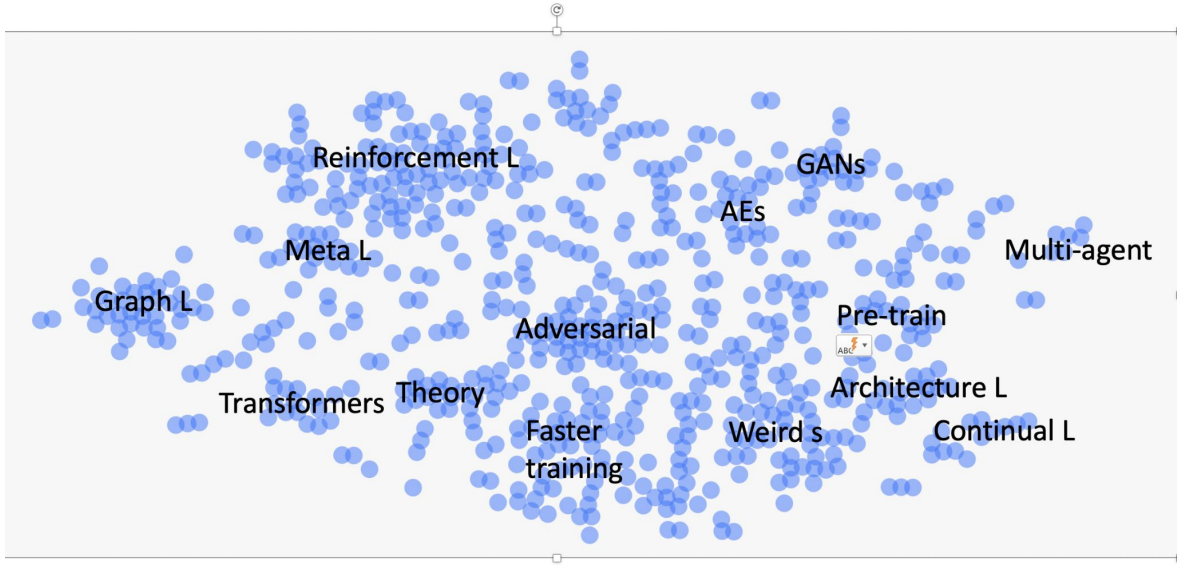
The algonauts project, brain-score.org

And also

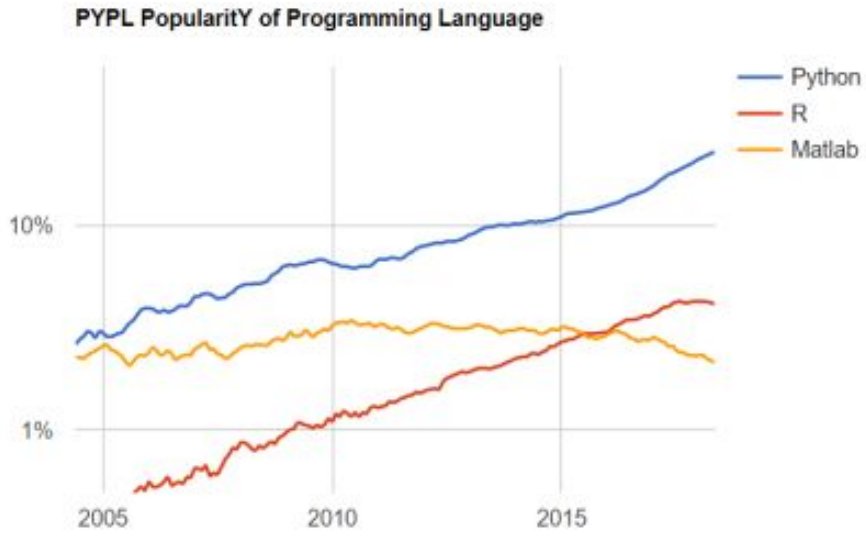
I personally believe that the brain does something like gradient descent

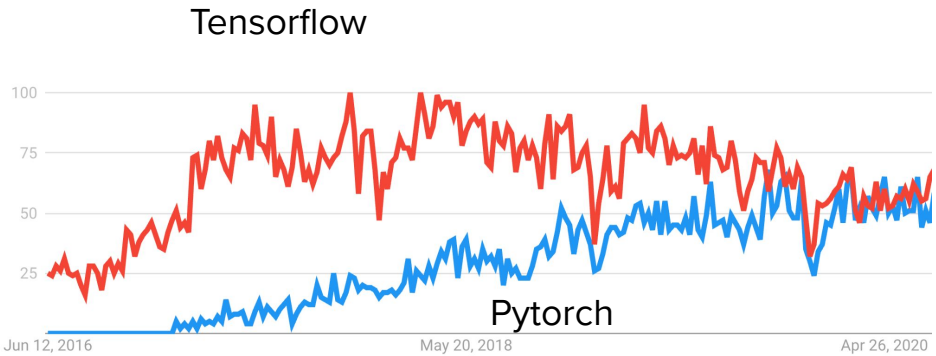
e.g. Kording and Konig 2004, Richards et al 2020

Gestalt of current DL



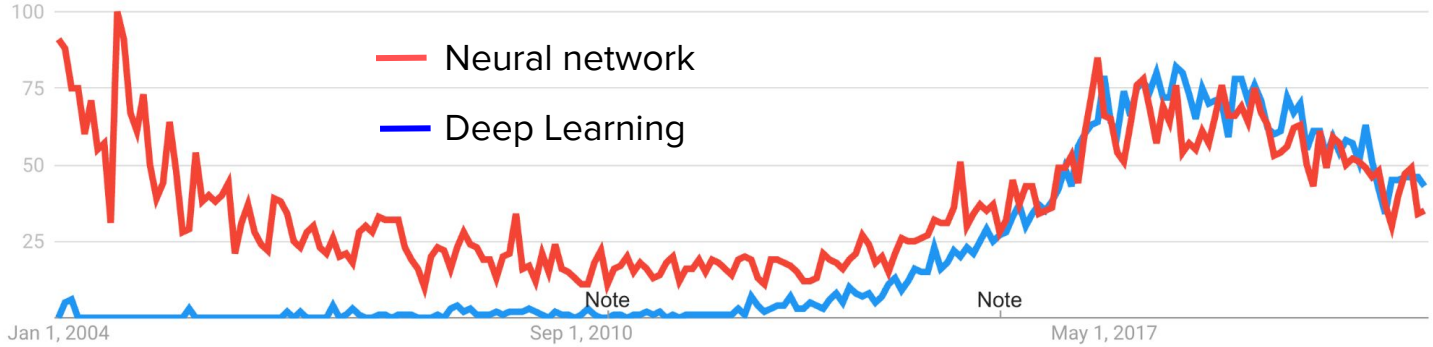
Python and Pytorch





source
https://blog.revolutionanalytics.com/2018/06/pypl-programming-language-trends.html

Are we beyond peak DL?



Pytorch

Numpy
on a GPU
with all kinds of ANN related things
with a focus on tensors
and automatic differentiation

Alternatives to pytorch

Tensorflow - strength in commercial applications

JAX - strength in flexibility

Matlab

DL in Numpy

```
    w2 = w2_init;
    b2 = b2_init;

    for iter = 1: 5000
        z = x_train * w1 + b1;
        a = 1./(1 + exp(-z));
        eta = 1./(1 + exp(-(a*w2+b2)));
        eps = eta - y_train;
        dgz = a .* (1 - a);

        dw1 = (w2 .* ((eps .* dgz)' * x_train))'./m_train;
        db1 = (w2 .* (dgz' * eps))'./m_train;
        dw2 = (eps' * a)'./m_train;
        db2 = sum(eps)/m_train;

        w1 = w1 - step .* dw1;
        b1 = b1 - step .* db1;
        w2 = w2 - step .* dw2;
        b2 = b2 - step .* db2;
    end

    pred_a_train = 1./(1 + exp(-(x_train * w1 + b1)));
    pred_eta_train = 1./(1 + exp(-(pred_a_train * w2 + b2)));
    pred_y_train = sign(pred_eta_train .* 2 - 1);
    pred_y_train = (pred_y_train + 1) ./ 2;
    err_train = classification_error(pred_y_train, y_train);
    err_train_arr(1, i) = err_train;

    pred_a_test = 1./(1 + exp(-(x_test * w1 + b1)));
    pred_eta_test = 1./(1 + exp(-(pred_a_test * w2 + b2)));
    pred_y_test = sign(pred_eta_test .* 2 - 1);
    pred_y_test = (pred_y_test + 1) ./ 2;
    err_test = classification_error(pred_y_test, y_test);
    err_test_arr(1, i) = err_test;

end

%%
plot(d1_arr, err_train_arr, 'color', 'b'); hold on;
plot(d1_arr, err_test_arr, 'color', 'r'); hold on;
plot(d1_arr, err_cv_arr, 'color', 'g');


%%
disp(err_train_arr); hold on; %0.3960    0.1440    0.1120    0.0760    0.0640    0.0360
disp(err_test_arr); hold on; %0.3939    0.2066    0.1726    0.1563    0.1538    0.1544
disp(err_cv_arr); %0.3960    0.1840    0.1640    0.1000    0.1520    0.1440



%%
w1_init = load(strcat(path_init, "w1_", num2str(5), ".mat"));
```

```

b1_init = load(strcat(path_init, "b1_", num2str(5), ".mat"));
b1_init = b1_init.b1;

w2_init = load(strcat(path_init, "w2_", num2str(5), ".mat"));
w2_init = w2_init.w_2;

b2_init = load(strcat(path_init, "b2_", num2str(5), ".mat"));
b2_init = b2_init.b2;

%%
x_train = trainData;
x_train(:, d) = [];
y_train = trainData(:, d);
y_train = (y_train + 1)./2;

%%
x_test = testData;
x_test(:, d) = [];
y_test = testData(:, d);
y_test = (y_test + 1)./2;

%%
step = 0.1;
%%
w1 = w1_init;
b1 = b1_init;
w2 = w2_init;
b2 = b2_init;
for iter = 1: 5000
    z = x_train * w1 + b1;
    a = 1./(1 + exp(-z));
    eta = 1./(1 + exp(-(a*w2+b2)));
    eps = eta - y_train;
    dgz = a .* (1 - a);

    dw1 = (w2 .* ((eps .* dgz)' * x_train))'./m_train;
    db1 = (w2 .* (dgz' * eps))'./m_train;
    dw2 = (eps' * a)'./m_train;
    db2 = sum(eps)/m_train;

    w1 = w1 - step .* dw1;
    b1 = b1 - step .* db1;
    w2 = w2 - step .* dw2;
    b2 = b2 - step .* db2;
end

pred_a_train = 1./(1 + exp(-(x_train * w1 + b1)));
pred_eta_train = 1./(1 + exp(-(pred_a_train * w2 + b2)));
pred_y_train = sign(pred_eta_train .* 2 - 1);
pred_y_train = (pred_y_train + 1) ./ 2;
```

DL in Pytorch

```
# Define the structure of your network
def __init__(self):
  super(NaiveNet, self).__init__()

  # The network is defined as a sequence of operations
  self.layers = nn.Sequential(
      nn.Linear(2, 16), # Transformation from the input to the hidden layer
      nn.ReLU(),        # Activation function (ReLU)
      nn.Linear(16, 2), # Transformation from the hidden to the output layer
  )

# Specify the computations performed on the data
def forward(self, x):
  # Pass the data through the layers
  return self.layers(x)
```

Everything in pytorch are tensors: how to make one

```
# tensor from a list
a = torch.tensor([0,1,2])

#tensor from a tuple of tuples
b = ((1.0, 1.1), (1.2,1.3))
b = torch.tensor(b)

# tensor from a numpy array
c = np.ones([2,3])
c = torch.tensor(c)
```

More tensors: common constructors

```
x = torch.ones(5, 3)
y = torch.zeros(2)
z = torch.empty(1,1,5)
```

Making random tensors

```
# uniform distribution
a = torch.rand(1,3)

# normal distribution
b = torch.randn(3,4)
```

Ranges in pytorch - just like in numpy

```
a = torch.arange(0,10,step=1)
b = np.arange(0,10,step=1)

c = torch.linspace(0,5,steps=11)
d = np.linspace(0,5,num=11)
```

What can we do with tensors?

Everything we do with numpy otherwise.

```
torch.add(a, b, out=c)
torch.multiply(a,b, out=d)
```

By default everything is pointwise

```
x + y, x - y, x * y, x / y, x**y
```

Sums, means etc

Just like in numpy

```
print("Sum of every element of x: ", x.sum())
print("Sum of the columns of x: ", x.sum(axis=0))
print("Sum of the rows of x: ", x.sum(axis=1))
```

Manipulating tensors: indexing

```
x = torch.arange(0, 10)
print(x)
print(x[-1])
print(x[1:3])
print(x[:-2])
```

```
tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
tensor(9)
tensor([1, 2])
tensor([0, 1, 2, 3, 4, 5, 6, 7])
```

Similar logic as numpy for n-dimensional tensors

```
x = torch.rand(1,2,3,4,5)

print(" shape of x[0]:", x[0].shape)
print(" shape of x[0][0]:", x[0][0].shape)
print(" shape of x[0][0][0]:", x[0][0][0].shape)
```

```
 shape of x[0]: torch.Size([2, 3, 4, 5])
 shape of x[0][0]: torch.Size([3, 4, 5])
 shape of x[0][0][0]: torch.Size([4, 5])
```

Flattening/ Reshaping

```
z = torch.arange(12).reshape(6,2)
print("Original z: \n ", z)

# 2D -> 1D
z = z.flatten()
print("Flattened z: \n ", z)
```

```
Original z:
  tensor([[ 0,  1],
          [ 2,  3],
          [ 4,  5],
          [ 6,  7],
          [ 8,  9],
          [10, 11]])
Flattened z:
  tensor([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

Reshaping

```
# and back to 2D
z = z.reshape(3, 4)
print("Reshaped (3x4) z: \n", z)
```

```
Reshaped (3x4) z:
 tensor([[ 0,  1,  2,  3],
         [ 4,  5,  6,  7],
         [ 8,  9, 10, 11]])
```

Irrelevant dimensions

```
x = torch.randn(1,10)
# printing the zeroth element of the tensor will not give us the first number!

print(x.shape)
print("x[0]: ",x[0])
```

```
torch.Size([1, 10])
x[0]:  tensor([-0.7178,  0.2696, -0.7311, -0.7625,  0.6477, -0.5313, -0.0666, -0.7753,
        -2.3820, -0.8742])
```

Squeezing

```
# lets get rid of that singleton dimension and see what happens now
x = x.squeeze(0)
print(x.shape)
print("x[0]: ", x[0])
```

```
torch.Size([10])
x[0]:  tensor(-0.7178)
```

Also: `y = y.unsqueeze(1)`

Dimension permutation

E.g. going from RGB in dimension 1 to in dimension 3

```
# `x` has dimensions [color,image_height,image_width]
x = torch.rand(3,48,64)

# we want to permute our tensor to be [ image_height , image_width , color ]
x = x.permute(1,2,0)
# permute(1,2,0) means:
# the 0th dim of my new tensor = the 1st dim of my old tensor
# the 1st dim of my new tensor = the 2nd
# the 2nd dim of my new tensor = the 0th
print(x.shape)
```

```
torch.Size([48, 64, 3])
```

Concatenation

```
# Create two tensors of the same shape
x = torch.arange(12, dtype=torch.float32).reshape((3, 4))
y = torch.tensor([[2.0, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])


#concatenate them along rows
cat_rows = torch.cat((x, y), dim=0 )

# concatenate along columns
cat_cols = torch.cat((x, y), dim=1 )
```

```
Concatenated by rows: shape[6, 4]
 tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.],
        [ 2.,  1.,  4.,  3.],
        [ 1.,  2.,  3.,  4.],
        [ 4.,  3.,  2.,  1.]])

 Concatenated by colums: shape[3, 8]
 tensor([[ 0.,  1.,  2.,  3.,  2.,  1.,  4.,  3.],
        [ 4.,  5.,  6.,  7.,  1.,  2.,  3.,  4.],
        [ 8.,  9., 10., 11.,  4.,  3.,  2.,  1.]])
```

torch and numpy are friends

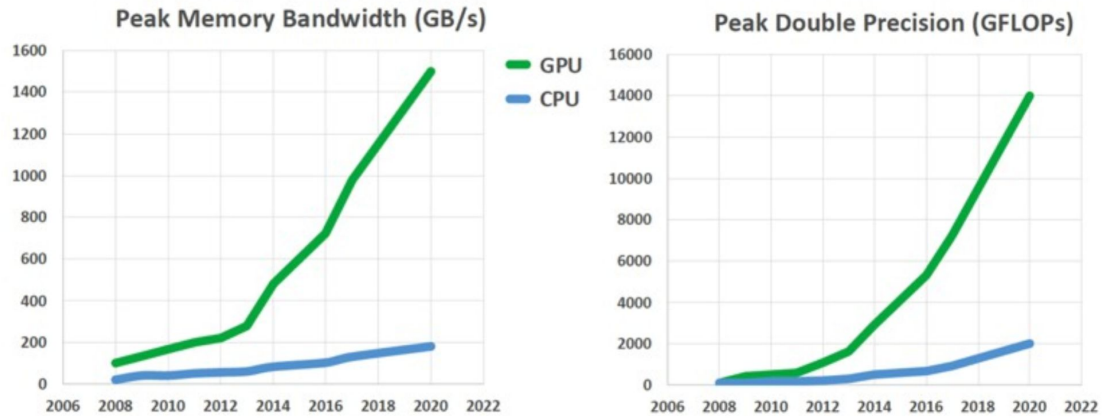
```
x = torch.randn(5)
print(f"x: {x}   |   x type:   {x.type()}")

y = x.numpy()
print(f"y: {y}   |   y type:   {type(y)}")

z = torch.tensor(y)
print(f"z: {z}   |   z type:   {z.type()}")
```

```
x: tensor([ 0.1647, -0.4206, -1.0624,  0.0760, -0.9115])  |  x type:  torch.FloatTensor
y: [ 0.1647326  -0.42064342 -1.062387    0.07596353 -0.91154546]  |  y type:  <class 'numpy.ndarray'>
z: tensor([ 0.1647, -0.4206, -1.0624,  0.0760, -0.9115])  |  z type:  torch.FloatTensor
```

Graphics cards: using GPUs



Ask torch where a variable is

```
x = torch.randn(10)
print(x.device)
```

```
cpu
```

Ask torch if we have a GPU

```
print(torch.cuda.is_available())
```

```
True
```

Specifying devices

```
# common device agnostic way of writing code that can run on cpu OR gpu
# that we provide for you in each of the tutorials
device = "cuda" if torch.cuda.is_available() else "cpu"

# we can specify a device when we first create our tensor
x = torch.randn(2,2, device=device)
print(x.dtype)
print(x.device)

# we can also use the .to() method to change the device a tensor lives on
y = torch.randn(2,2)
print(f"y before calling to() |  device: {y.device} | dtype: {y.type()}")

y = y.to(device)
print(f"y after calling to() |  device: {y.device} | dtype: {y.type()}")
```

```
torch.float32
cuda:0
y before calling to() |  device: cpu | dtype: torch.FloatTensor
y after calling to() |  device: cuda:0 | dtype: torch.cuda.FloatTensor
```

Device matters: no mix and match

We can not just mix and match devices - it would be undefined where the computation happens

```
x = torch.tensor([0,1,2], device="cuda")
y = torch.tensor([3,4,5], device="cpu")

z = x + y
```

Moving CPU<->GPU is easy

```
x = torch.tensor([0,1,2], device="cuda")
y = torch.tensor([3,4,5], device="cpu")
z = torch.tensor([6,7,8], device="cuda")

# moving to cpu
x = x.cpu()
print(x + y)

# moving to gpu
y = y.cuda()
print(y + z)
```

```
tensor([3, 5, 7])
tensor([ 9, 11, 13], device='cuda:0')
```

Datasets

Data

+

Model

+

Training

=

DL system

Doing data - basics

Data science =

50% figure out the question you want to answer

35% sweat the data

10% ML

5% glorious DL

How to get data

A lot of data is easy to load for our DL experiments

```
cifar10_data = datasets.CIFAR10(
    root="data",              # path where the images will be stored
    download=True,            # all images should be downloaded
    transform=ToTensor()      # transform the images to tensors
)
```

Data

Data is not made in heaven

Data is made to answer questions

We need to be agile with data

When we try to answer questions we do not want to lie to ourselves

Let us not lie about data

In DL we generally do prediction

Caution with Causality

The real world differs from our dataset (external validity)

Let us not lie about data: Validation

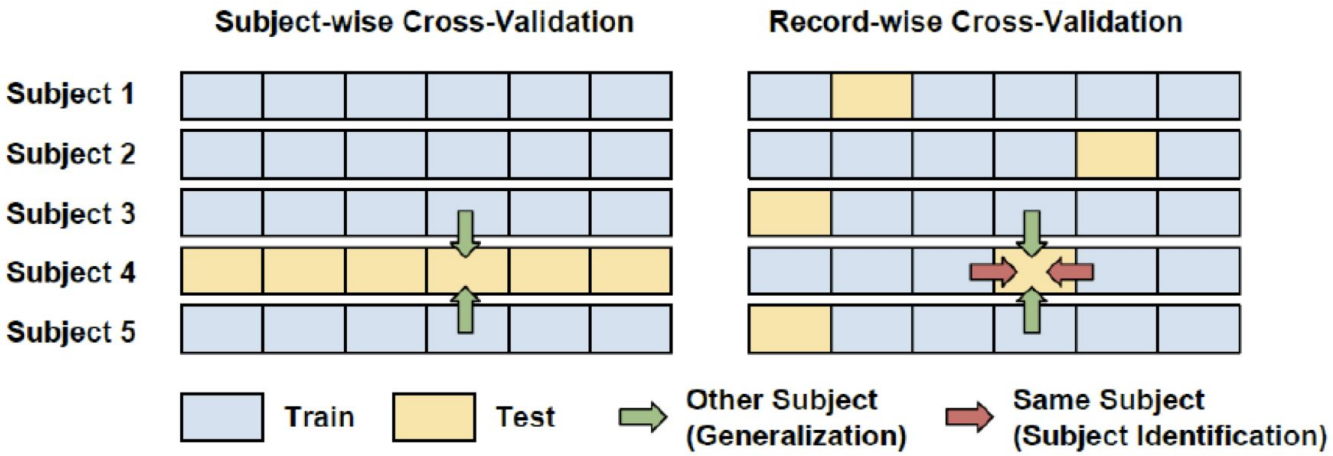
Always have a validation/ Test set not used for training.

Train on training set, test on test set

For hyperparameter optimization you need to further divide the training set

Match the cross-validation strategy to the use case

The cross-validation strategy must match the use case



Saeb ... Kording 2018

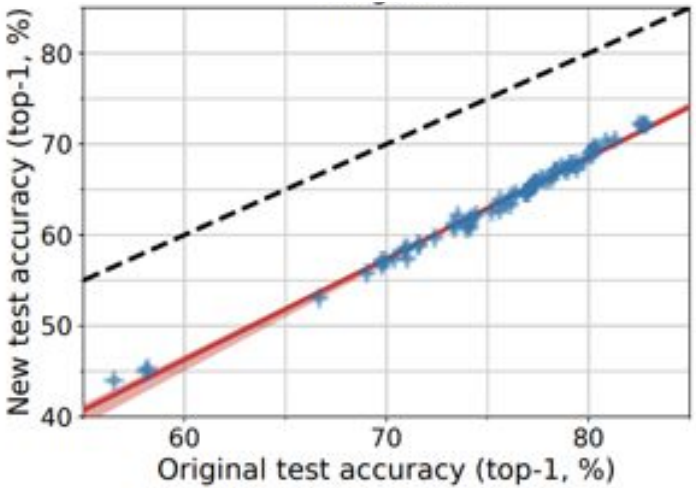
Overfitting! Validation set

Don’t trust yourself

Overfitting is massive for smaller datasets

Ideally have a part of the dataset you don’t have access to

Even some signs for *huge* datasets (imagenet)



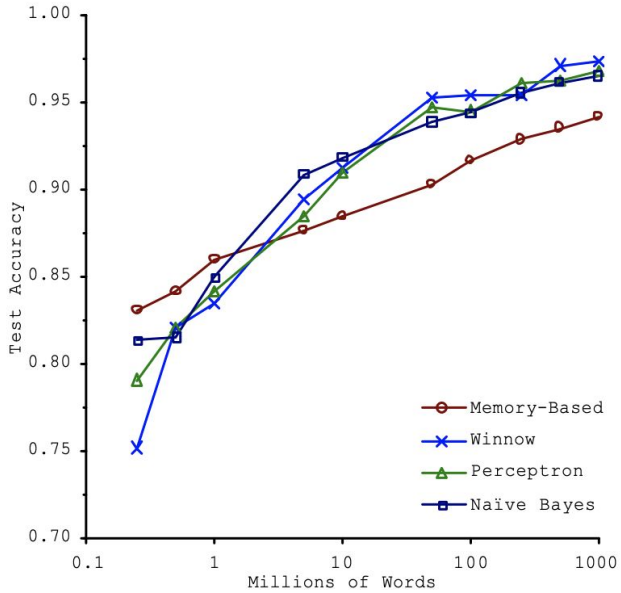
Recht et al 2019

Always have both training and test data

```
# Load the training samples
training_data = datasets.CIFAR10(
    root="data",
    train=True,
    download=True,
    transform=ToTensor()
)

# Load the test samples
test_data = datasets.CIFAR10(
    root="data",
    train=False,
    download=True,
    transform=ToTensor()
)
```

More data is what it is all about



Banko and Brill 2001

Transformations

More data = better learning

How to get more data?

- Get more data
- Transform the data

e.g. add color variation, etc.

Transformations are crucial across DL

Data Loaders

In practice we do not load all data.

But small pieces (minibatches)

For that we have a function that does the loading

```
# Create dataloaders with
train_dataloader = DataLoader(training_data, batch_size=64, shuffle=True)
test_dataloader = DataLoader(test_data, batch_size=64, shuffle=True)
```

Now, let us design a neural network

(step 0) Get Data

(step 1) All the variables and structures we need.
We need to initialize them

(step 2) And then we need to use these variables to define the compute in our network

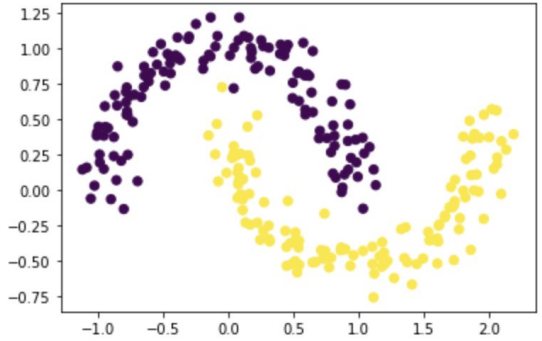
(step 3) And then we need gradients

(step 4) And then we need to optimize

(step 5) And then we need to test

Let us get the data from a csv file

Why? Because many real world datasets are in that format



Let us see the anatomy of the network

First, we need to initialize the relevant variables

```
__init__
```

And then we need to specify how information travels through network

```
forward
```

We will often need to make predictions

While many people just use `forward` we will separate it and use

`predict`

and then we need to

`train`

With __init__ we make network structure

```
# Define the structure of your network
def __init__(self):
  super(NaiveNet, self).__init__()

  # The network is defined as a sequence of operations
  self.layers = nn.Sequential(
      nn.Linear(2, 16), # Transformation from the input to the hidden layer
      nn.ReLU(),        # Activation function (ReLU)
      nn.Linear(16, 2), # Transformation from the hidden to the output layer
  )
```

The other components

```
# Specify the computations performed on the data
def forward(self, x):
  # Pass the data through the layers
  return self.layers(x)

# Convert the output of the network to a probability distribution
def predict(self, x):
  # Pass the data through the networks
  output = self.forward(x)

  # Choose the label with the highest score
  return torch.argmax(output, 1)

# Train the neural network (will be implemented later)
def train(seld, X, y):
  pass
```

Ok hold on. What has just happened

We have a neural network

It is initialized

It produces outputs

But these outputs are not better than chance yet!

Training = lots of small steps into good direction

```
# The Cross Entropy Loss is suitable for classification problems
loss_function = nn.CrossEntropyLoss()

# Create an optimizer (Stochastic Gradient Descent) that will be used to train the network
learning_rate = 1e-2
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

# Number of epochs
epochs = 15000
```

The anatomy of the training loop

```
for i in range(epochs):
  # Pass the data through the network and compute the loss
  y_logits = model(X)
  loss = loss_function(y_logits, y)

  # Clear the previous gradients and compute the new ones
  optimizer.zero_grad()
  loss.backward()

  # Adapt the weights of the network
  optimizer.step()
```

Just like magic

Woodcut illustration from an edition of Pliny the Elder's *Naturalis Historia* (1582)

The rest of this course

We now need intuition for what works and what does not

Architectures

Transfer functions

etc.

One way to start is to take a simple network and explore

The XOR problem

Two inputs 1 output

Output = 1 if the two inputs are different (0 1) or (1 0), 0 otherwise

There is a history to this

Run the XOR widget

Ethics

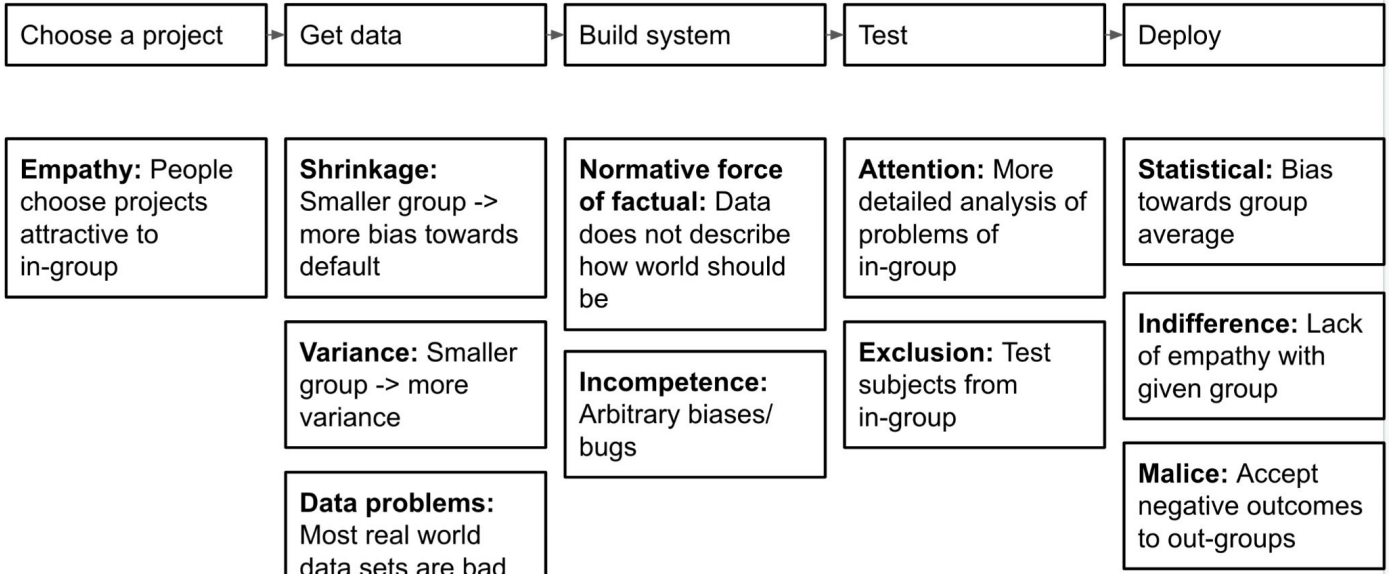
ML is powerful

ML affects lives

Lots of things to say about ethics

But we will instead just listen to victims

Biases, from the unavoidable to the bad



What we learned today

Where data is coming from
How to construct a network (init and forward)
How to optimize a network (training loop)
How to evaluate a network
The need to consider the impact on society

These steps always matter

We also learned about the landscape of DL

Computer Vision
Natural Language Processing
Reinforcement Learning
Generative Adversarial Networks
Recurrent Neural Networks