Introduction to PyTorch

Jianan Zhao
Mila - Quebec AI Institute
Email: jianan.zhao@mila.quebec

Contributors: Jiarui Lu, Meng Qu, Zhaocheng Zhu, Louis-Pascal Xhonneux, Shengchao Liu and Andreea Deac

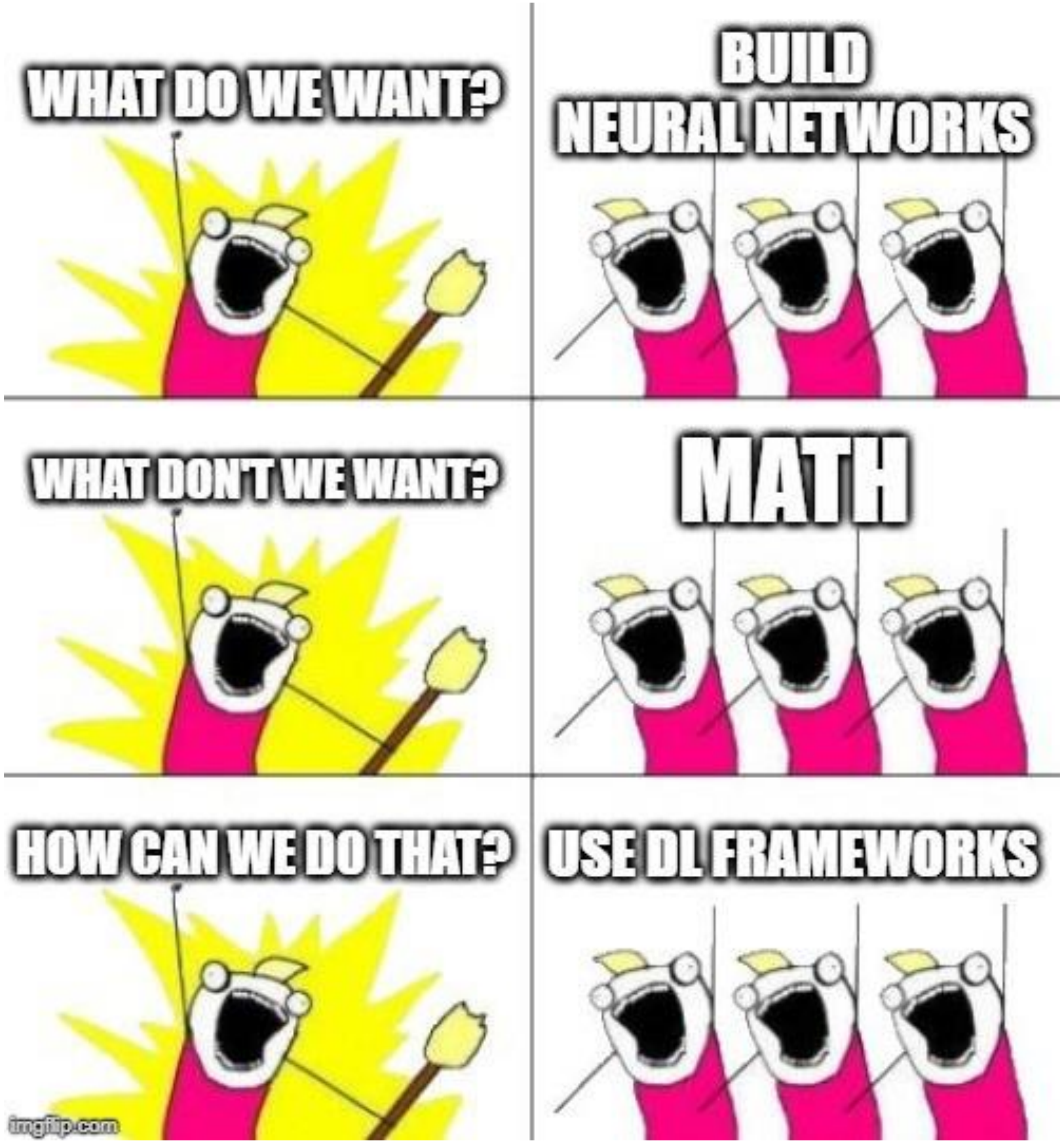
Content

- Part I: Pytorch Basics
  - Idea of deep learning
  - Pytorch Tensor
  - Tensor Operations

- Part II: Deep Learning Pipeline
  - A General DL Pipeline
  - An Image Classification Example
  - Suggestions on Debugging

Part I: PyTorch basics

- Idea of deep learning
- A fundamental data structure: Tensor
- Training through auto-gradient


WHAT DO WE WANT?
BUILD NEURAL NETWORKS
WHAT DON'T WE WANT?
MATH
HOW CAN WE DO THAT?
USE DL FRAMEWORKS
imgflip.com

Why deep learning frameworks?

- DL Frameworks can help us
  - build neural networks without annoying math
  - reduce development efforts on standard modules
  - accelerate training with GPUs or distributed training

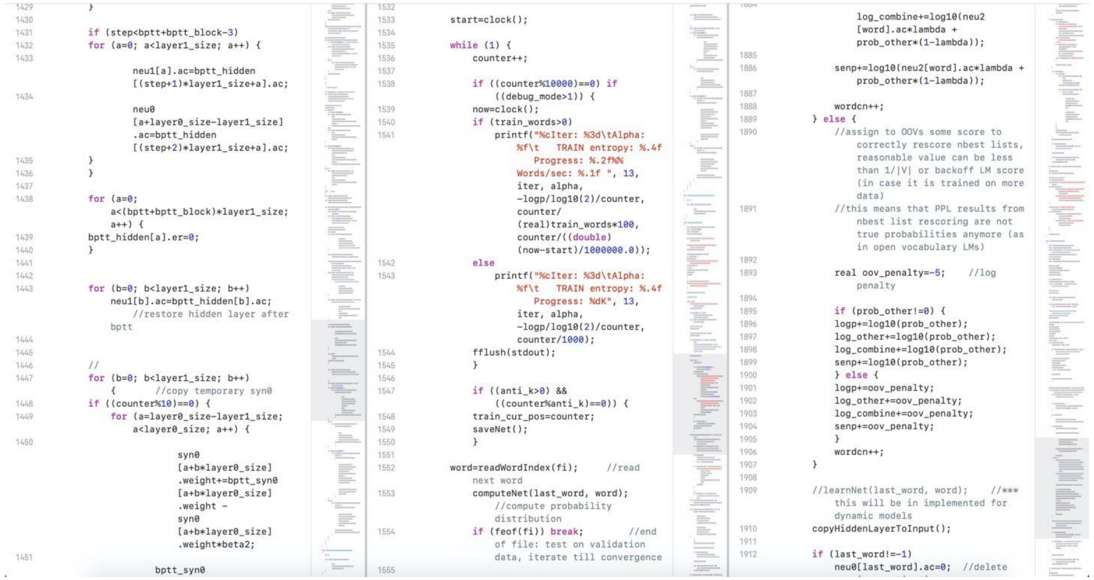
- e.g. You can apply standard models to your own dataset with in ~10 lines of Python code

- Also, most open source projects are based on these frameworks

How powerful are deep learning frameworks?

2012

More than 3000 lines



Today

About 50 lines

```
import torch.nn as nn

class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNN, self).__init__()

        self.hidden_size = hidden_size

        self.i2h = nn.Linear(input_size + hidden_size, hidden_size)
        self.i2o = nn.Linear(input_size + hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden):
        combined = torch.cat((input, hidden), 1)
        hidden = self.i2h(combined)
        output = self.i2o(combined)
        output = self.softmax(output)
        return output, hidden

    def initHidden(self):
        return torch.zeros(1, self.hidden_size)
```

Popular deep learning frameworks

PyTorch

TensorFlow

Chainer

mxnet

JAX

SKORCH

Sonnet

Keras

Which framework to use?

TensorFlow

☺️

- high performance
- good distributed & large-scale training
- great for industrial deployment

☹️

- difficult to get started with
- difficult to debug

PyTorch

☺️

- similar to native Python logic
- easy to get start with
- easy to debug
- rapid prototyping and research

☹️

- bad support of large-scale training
- write bad performance code unintentionally

In this class

PyTorch

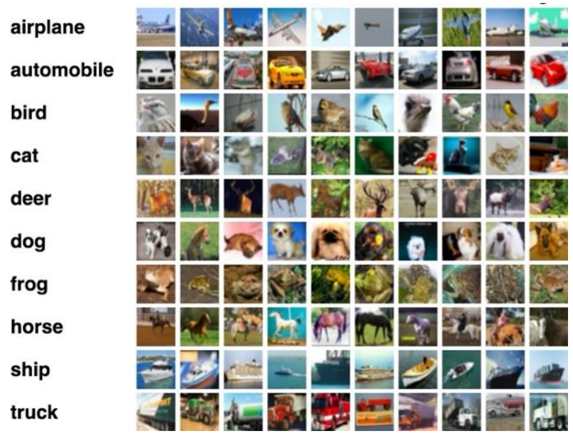
- We will focus on **PyTorch**
  - PyTorch is “pythonic” in its style
  - PyTorch is opensource backed by Facebook

- It provides us with Tensors, Autodifferentiation, and functions commonly used in Deep Learning models.

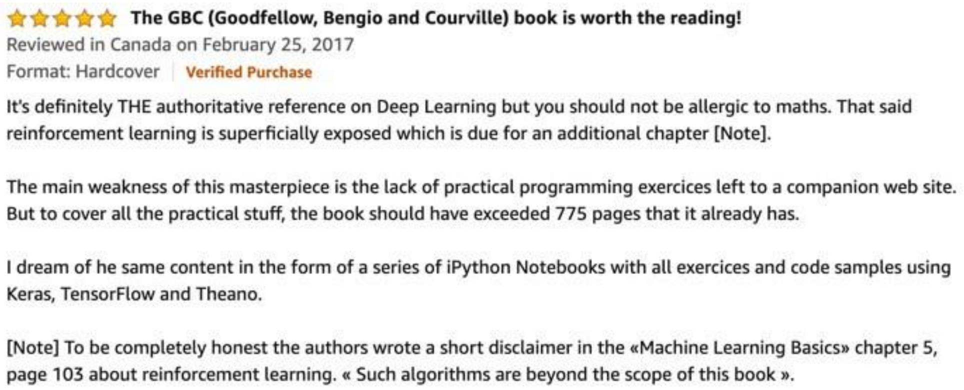
The goal of (supervised) deep learning

- Transform data from one representation to another

- Example 1: Image Classification
  - Input: images
  - Output: image labels



- Example 2: Sentence Regression
  - Input: item reviews
  - Output: corresponding ratings



The GBC (Goodfellow, Bengio and Courville) book is worth the reading!

Reviewed in Canada on February 25, 2017

Format: Hardcover | **Verified Purchase**

It's definitely THE authoritative reference on Deep Learning but you should not be allergic to maths. That said reinforcement learning is superficially exposed which is due for an additional chapter [Note].

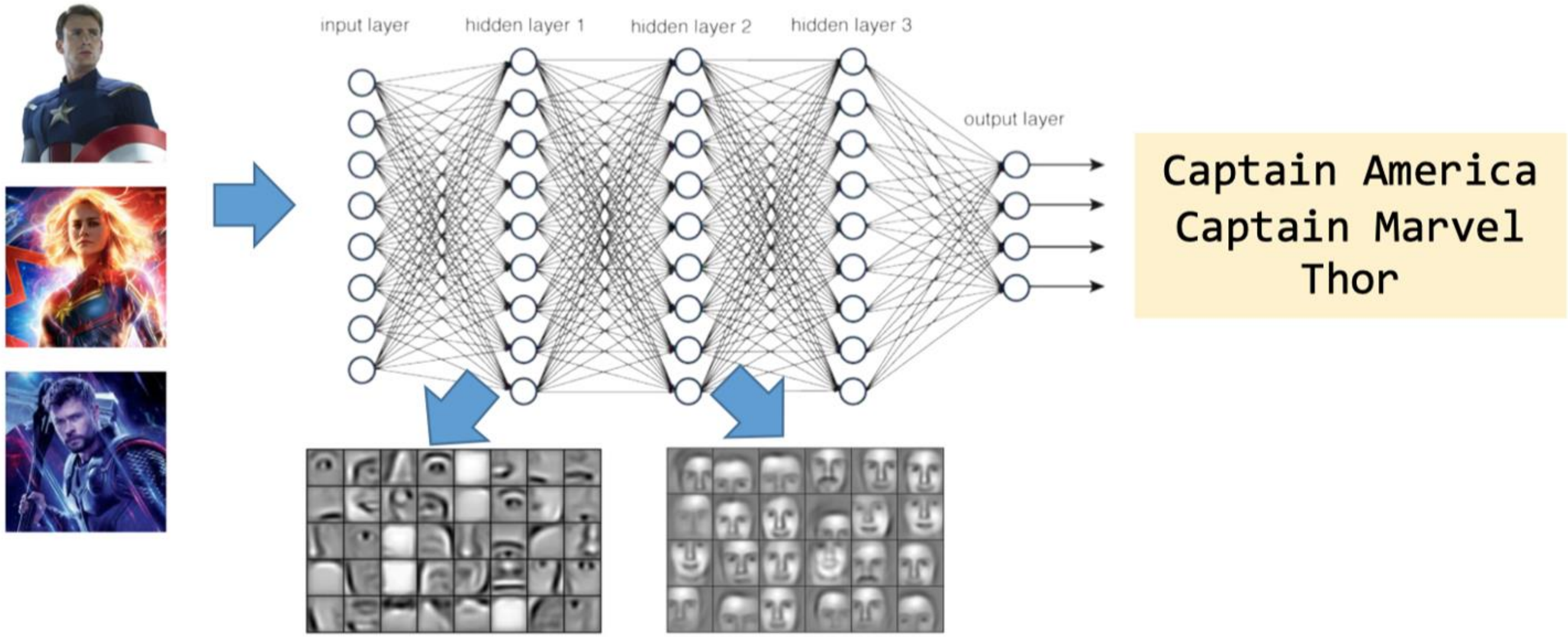
The main weakness of this masterpiece is the lack of practical programming exercices left to a companion web site. But to cover all the practical stuff, the book should have exceeded 775 pages that it already has.

I dream of he same content in the form of a series of iPython Notebooks with all exercices and code samples using Keras, TensorFlow and Theano.

[Note] To be completely honest the authors wrote a short disclaimer in the «Machine Learning Basics» chapter 5, page 103 about reinforcement learning. « Such algorithms are beyond the scope of this book ».

Workflow of deep learning

- We want to train a neural network f_θ for a given task T



- We solve $\min_\theta L(f_\theta(x), y)$ using gradient descent $\theta_{i+1} = \theta_i - \epsilon \nabla_\theta L(f_\theta(x), y)$

Workflow of deep learning

- What we need/want:
  - A way to hold the data (x, y)
  - Functions to code the neural network $f_\theta(x)$
  - Functions to compute the loss $L(f_\theta(x), y)$
  - The ability to compute $\nabla_\theta L(f_\theta(x), y)$ automatically without needing to do maths on paper apriori

- PyTorch gives us these things through
  - Torch.tensor and torch.utils.data.DataLoader (to load data from files)
  - Torch.nn
  - Torch.nn.functional (or Torch.nn)
  - Tensor.backward() (see torch.autograd.backward)

PyTorch

Workflow of deep learning

- Installation
  - Libraries for Python
- Data preparation
  - Know how PyTorch stores data / what the data look like
  - Split the data into train/valid/test
- Model preparation
- Model training
  - Define loss function
  - Choose optimization method
  - Train the model on training data
- Model evaluation
  - Evaluate the model on test data

Ready, steady, go

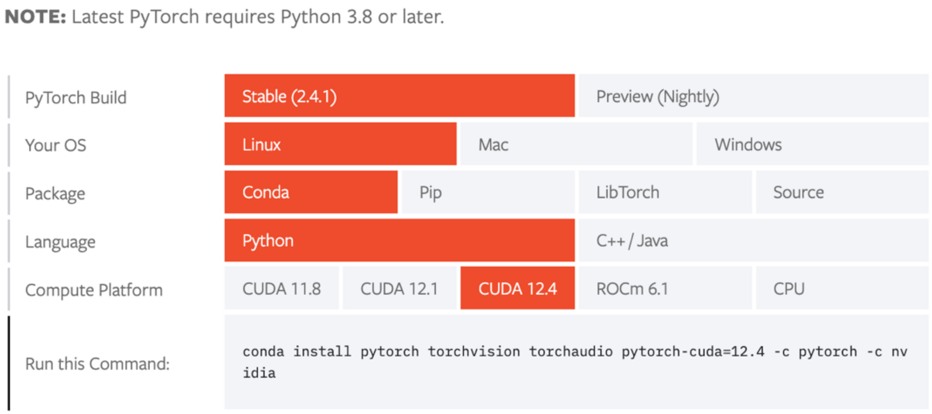
PyTorch

- We will learn how to use PyTorch to build and train Neural Networks

The first part of the notebook: Collab Notebook

Installation

- On your laptop go to https://pytorch.org/get-started/locally/



  - Use the stable build, your OS, either Pip or Conda, and the cuda version you have if you have a Nvidia GPU in your laptop
- **On Google Colab**
  - Faster training with a GPU!
  - Enable it in **Runtime -> Change Runtime Type**

Installation

- Let's check if they are installed properly.

```
>>> import torch
>>> import torchvision
```

- If nothing complains, then you are ready to go.

- To check if GPU acceleration is available,

```
>>> torch.cuda.is_avaiable()
```

- Note this doesn't necessarily mean everything runs on GPU by default!

Part I: PyTorch basics

- Idea of deep learning

- A fundamental data structure: Tensor

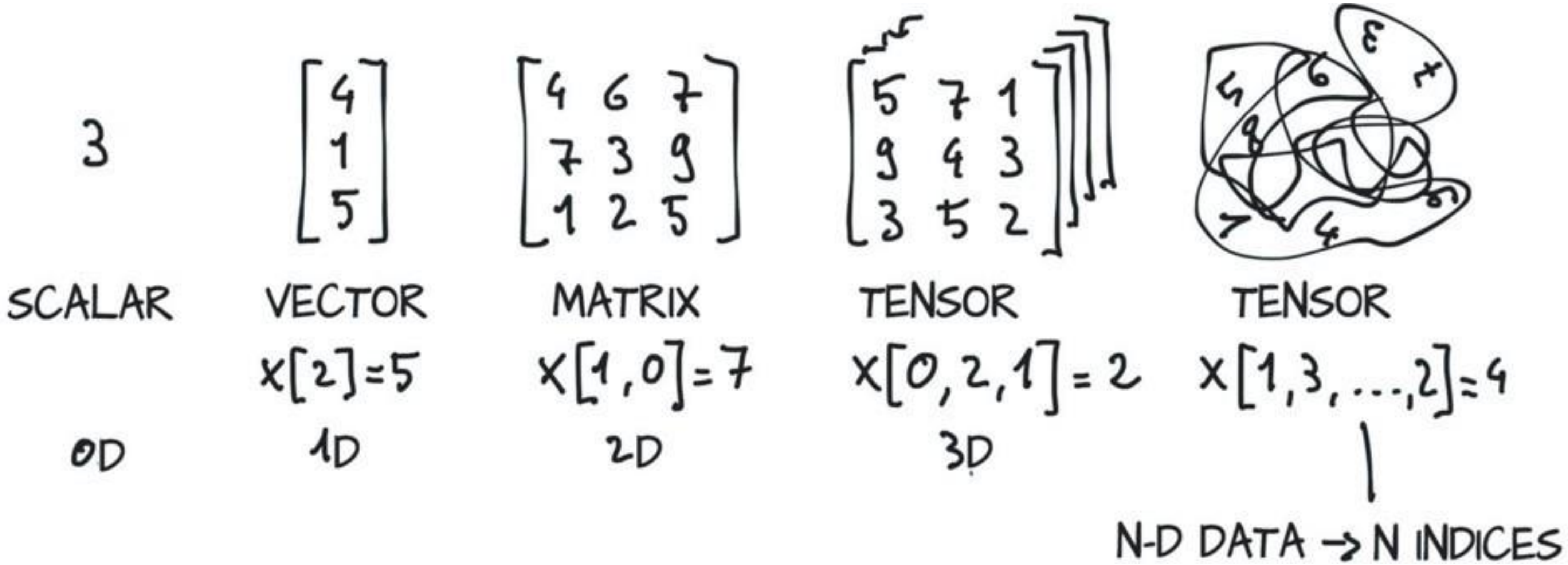
- Training through auto-gradient

Data structure for representations

- Deep learning relies heavily on Linear Algebra
    - Linear algebra uses tensors (e.g. 1-d tensor is a vector, 2-d tensor is a matrix)
    - PyTorch (and most deep learning frameworks), thus uses tensors (also called N-dimensional arrays).

What is a tensor?

- Two different understandings:
  - Generalization of vectors and matrices to an arbitrary number of dimensions
  - Multidimensional arrays



Tensors are powerful for data representations

Examples

- 1. Tabular data:
  - Two-dimensional tensors (matrices)

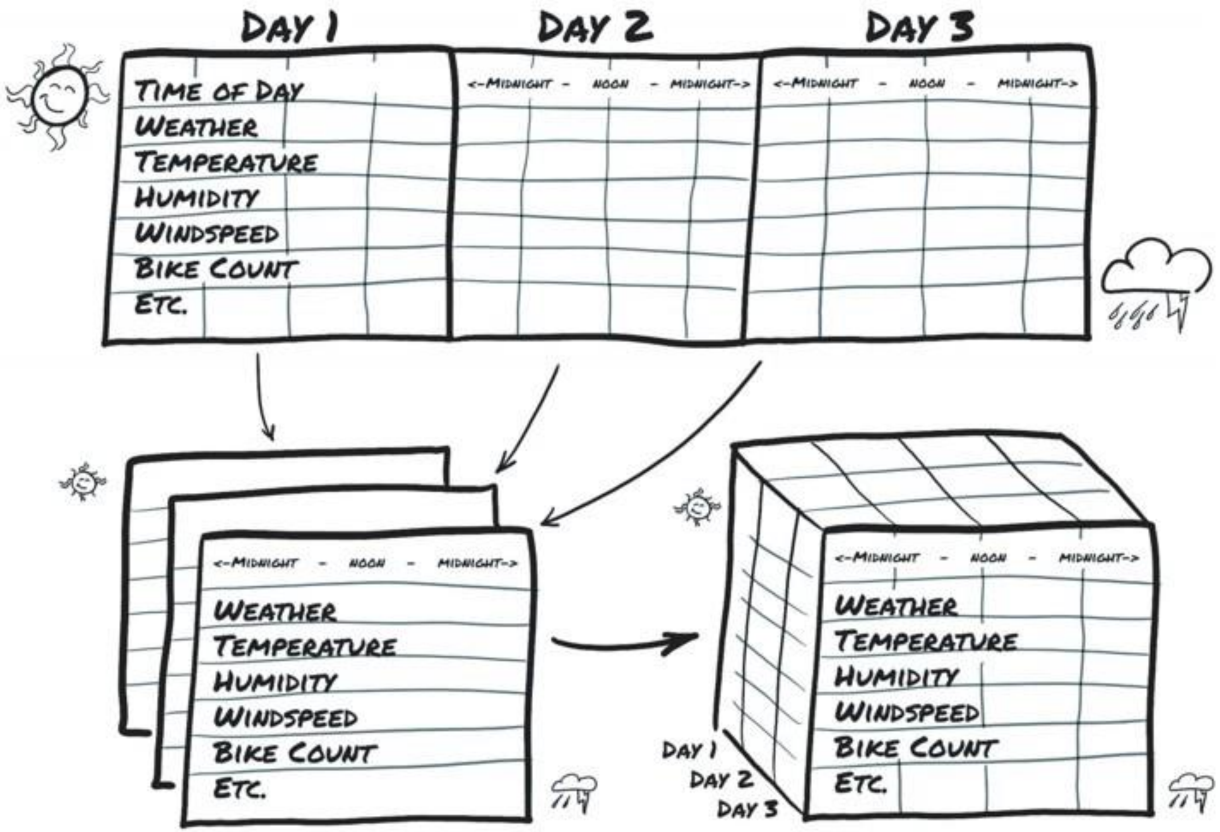
Attributes of 2009EQ.csv Events

Year	Month	Day	Time_UTC	Latitude	Longitude	Magnitude	Depth	Shape *
2009	1	1	12:43:12 PM	-15.43	-173.14	4	35	Point
2009	1	1	9:36:00 PM	17.22	40.52	5	10	Point
2009	1	1	9:50:24 AM	-55.16	-29.09	4.7	35	Point
2009	1	1	1:26:24 PM	80.85	-3.03	4.8	10	Point
2009	1	1	6:28:48 AM	-6.92	155.18	4.6	82	Point
2009	1	1	10:19:12 AM	-6.83	129.99	4.7	50	Point
2009	1	1	12:00:00 PM	-33.8	-72.72	4.6	0	Point
2009	1	1	12:57:36 PM	-58.29	-21.81	4.4	10	Point
2009	1	1	3:21:36 AM	-6.86	155.93	4.7	50	Point
2009	1	1	7:55:12 PM	1.12	120.73	4.7	49	Point
2009	1	1	11:16:48 AM	-11.66	166.75	4.7	254	Point
2009	1	1	2:09:36 AM	40.62	123.02	4.1	10	Point
2009	1	1	5:16:48 AM	-34.84	-107.65	5.8	10	Point
2009	1	1	1:40:48 PM	-9.61	120.72	4.2	20	Point
2009	1	1	2:38:24 AM	-22.04	-179.6	4.5	601	Point
2009	1	1	6:43:12 AM	1.32	121.84	5.1	33	Point
2009	1	1	4:19:12 PM	14.73	-91.39	4.7	169	Point
2009	1	1	5:45:36 PM	9.43	124.15	4.5	525	Point
2009	1	1	4:48:00 PM	-34.88	-107.78	5	10	Point
2009	1	1	8:09:36 PM	44.58	148.22	4.2	59	Point
2009	1	1	2:38:24 AM	-4.33	101.3	5.5	19	Point
2009	1	1	2:52:48 AM	-4.33	101.24	5.3	26	Point

Record: 1 Show: All Selected Records (0 out of 2048 Selected)

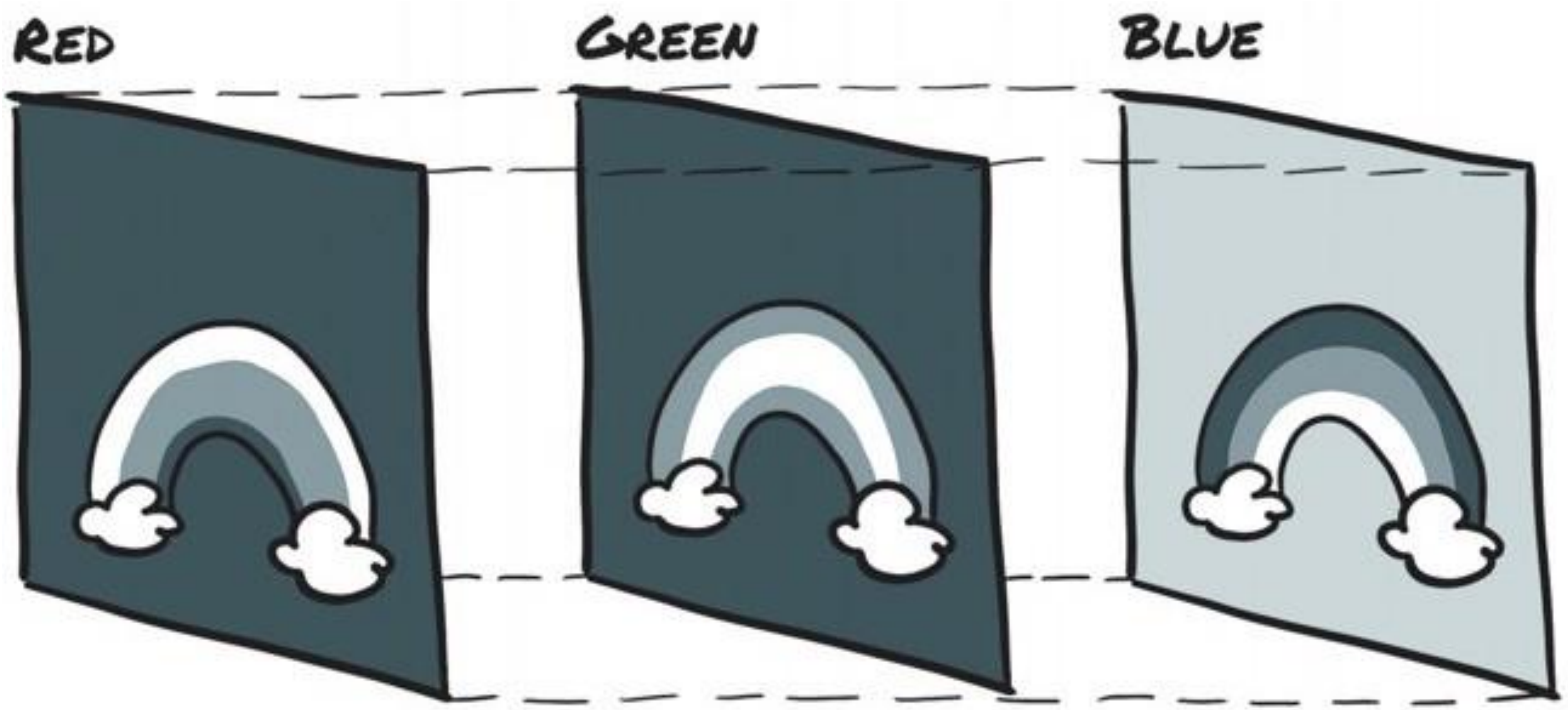
Tensors are powerful for data representations

- 2. Time-series data:
  - Three-dimensional tensors



Tensors are powerful for data representations

- 3. Images:
  - Three-dimensional tensors



Tensors are powerful for data representations

- 4. Texts:
  - As one-dimensional integer tensors
  - As two-dimensional float tensors (embeddings)

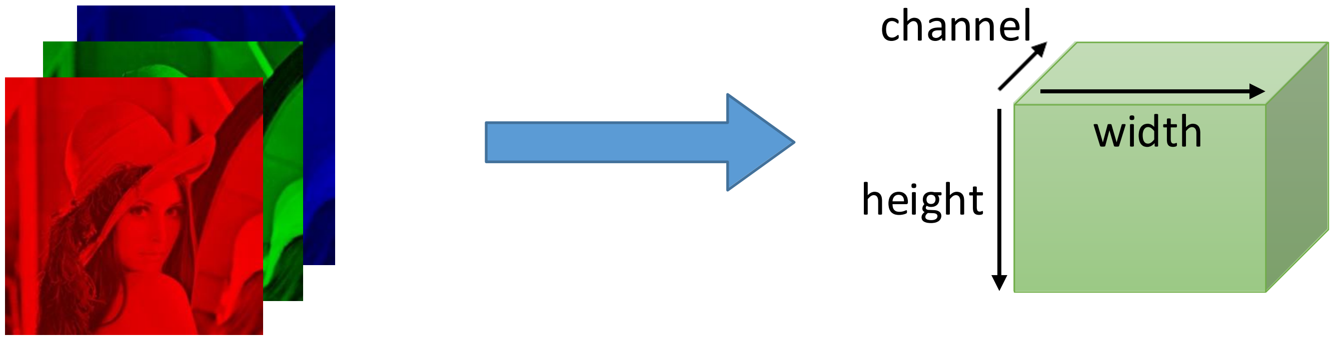

```
The fat cat sat on a mat
0   298  81 641  9 1 109
```

```
The fat cat sat on a mat

0.2  1.2  2.6  ……
0.3  0.6  1.5  ……
1.8  1.7  0.3  ……
```

```
>>> from PIL import Image
```

Basic tensor operations

- A real example of images




```
!wget https://upload.wikimedia.org/wikipedia/en/7/7d/Lenna_%28test_image%29.png
-O lenna.jpg
>>> np_image = np.array(Image.open("lenna.png"))
>>> image = torch.as_tensor(np_image)
>>> plt.imshow(image)
```

Basic tensor operations

- 1. Tensor creation
  - From Python lists or Numpy arrays

```
tensor([[ 0.1000,  1.2000],
        [ 2.2000,  3.1000],
        [ 4.9000,  5.2000]])
```

```
>>> torch.tensor([[0.1, 1.2], [2.2, 3.1], [4.9, 5.2]])
```

```
>>> torch.tensor(np.array([[0.1, 1.2], [2.2, 3.1], [4.9, 5.2]]))
```

  - Tensors of a given size

```
>>> torch.tensor(2,3,4)
```

  - Special tensors

```
>>> torch.zeros(2,3)
```

```
>>> torch.ones(2,3)
```

```
>>> torch.eye(3)
```

```
tensor([[ 0.,  0.,  0.],
        [ 0.,  0.,  0.]])
```

```
tensor([[ 1.,  1.,  1.],
        [ 1.,  1.,  1.]])
```

```
tensor([[ 1.,  0.,  0.],
        [ 0.,  1.,  0.],
        [ 0.,  0.,  1.]])
```

Basic tensor operations

```
x = tensor([[ 0.1000,  1.2000],
            [ 2.2000,  3.1000],
            [ 4.9000,  5.2000]])
```

- 2. Tensor properties
  - Shape

    ```
    >>> x.shape
    ```
    ```
    >>> x.size()
    ```
    ```
    torch.Size([3, 2])
    ```

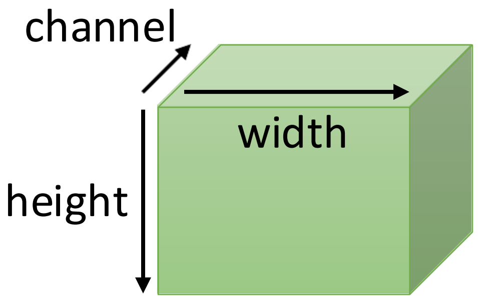
  - Data type

    ```
    >>> x.dtype
    ```
    ```
    >>> x.type()
    ```
    ```
    torch.float32
    ```

  - Number of dimensions

    ```
    >>> x.ndim
    ```
    ```
    2
    ```

Basic tensor operations



- By convention
  - `dimension` refers to an axis of the tensor
  - `size` refers to the length of an axis in the tensor
  - `index` refers to a specific coordinate in the tensor

```
>>> print(image.shape)
>>> print(image.dtype)
>>> print(image.ndim)
```

```
torch.Size([512, 512, 3]) # (height, width, channel)
torch.uint8
3 # 3 dimensions: height, width, and channel
```

Basic tensor operations

- 3. Tensor type transformation
  - Data type transformation

```
>>> x = x.int()
```

```
>>> x = x.float()
```

```
>>> x = x.double()
```

  - Transformation between GPU and CPU (to be revisited)

```
>>> x = x.cuda()
```

```
>>> x = x.cpu()
```

  - Transform a tensor to Numpy arrays

```
>>> x = x.numpy()
```

```
>>> x = x.data.numpy()
```

Tensor Device: on CPU or GPU

- There are generally two types of devices for deep learning
  - CPU: Skilled professor for **serial** complex tasks
  - GPU: Primary school students good at **parallel** simple tasks
- Tensor computation happens on either device

CPU	GPU
Several cores	Many cores
Low latency	High throughput
Ideal for serial processing	Ideal for parallel processing
Handles a handful of operations at once	Handles thousands of operations at once

Tensor Device: on CPU or GPU

- Note that any calculation requires tensors **on the exact SAME device**
- By default, a tensor is created on CPU
- Use .cuda() and .cpu() to move to devices

```
[8] torch.randn((2,3)) @ torch.randn((3,2)).cuda()

---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-8-97064f5a58c1> in <cell line: 1>()
----> 1 torch.randn((2,3)) @ torch.randn((3,2)).cuda()

RuntimeError: Expected all tensors to be on the same device, but found at least two devices, cpu and cuda:0! (when checking
argument for argument mat2 in method wrapper_CUDA_mm)
```

```
torch.randn((2,3)) @ torch.randn((3,2)).cuda().cpu()

tensor([[ 1.5078,  1.9919],
        [ 1.2747, -1.1332]])
```

Tensor Device: on CPU or GPU

- In practice, we use:

```
# Use GPU if available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

Basic tensor operations

```
x = tensor([[ 1.,  2.],
            [ 3.,  4.],
            [ 5.,  6.]])
```

- 4. Tensor indexing
  - Get an element

```
>>> x[0, 1]
```

```
tensor(2.)
```

  - Get a row (the colon ":" stands for all elements)

```
>>> x[2, :]
```

```
tensor([5., 6.])
```

  - Get a column

```
>>> x[:, 0]
```

```
tensor([1., 3., 5.])
```

  - Get rows

```
>>> x[1:3, :]
```

```
tensor([[3., 4.], [5., 6.]])
```

Basic tensor operations

- Operations on tensors are similar to their matrix counterparts

```
>>> plt.imshow(image[:, :256, :])
```
See example in Notebook

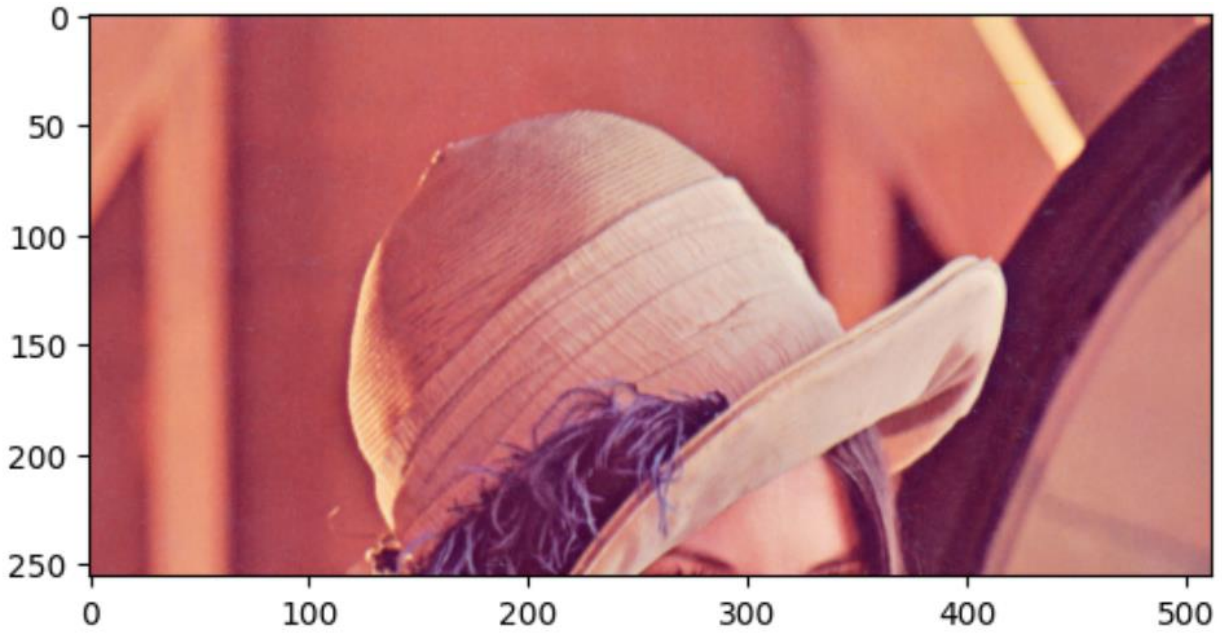
512×512×3

512×256×3

- Slice over the second axis (width axis)

Basic tensor operations

- Practice: How can we obtain **the upper half** of the image?



Expected Output

Basic tensor operations

```
x = tensor([[ 1.,  2.],
            [ 3.,  4.],
            [ 5.,  6.]])
```

See example in Notebook

- 5. Changing tensor dimensions
  - Tensor reshaping

```
>>> x.reshape(6)
```

```
>>> x.view(6)
```

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

  - Tensor squeezing and unsqueezing

```
>>> torch.unsqueeze(x, 0)
```

```
>>> torch.squeeze(x, 0)
```

  - Expansion

See example in Notebook

```
>>> x.expand(3, 2, 4)
```

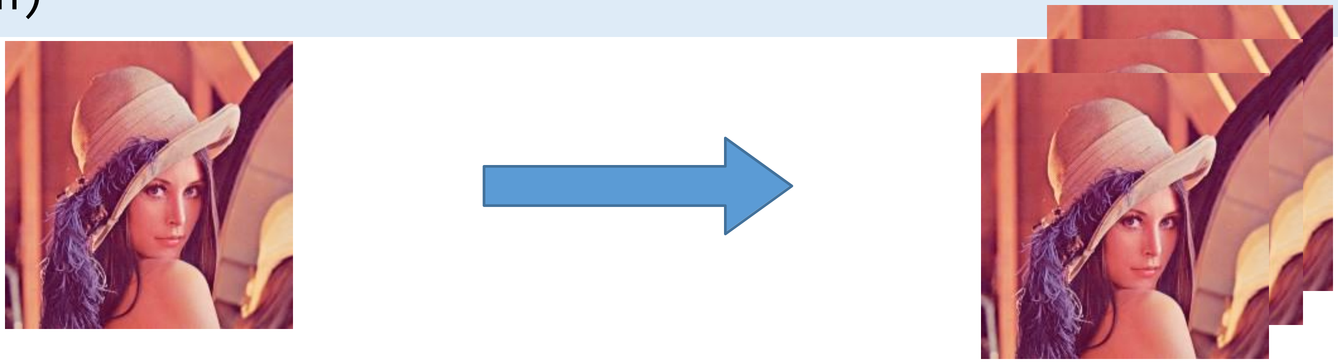
```
>>> x.repeat(3, 2, 4)
```

Basic tensor operations

- Extend a tensor

See example in Notebook

```
>>> batch = image.unsqueeze(0).repeat(3, 1, 1, 1)
>>> plot(batch)
```



512×512×3 3×512×512×3

- `Unsqueeze` creates a new axis with size 1 at the specific dimension

Basic tensor operations

```
x = tensor([1., 2., 3.])
```

- 6. Element-wise operations
  - Addition, subtraction, multiplication and division

```
>>> x + 3
```
```
tensor([4., 5., 6.])
```

  - Exponential, logarithm, power

```
>>> x.exp()
```
```
tensor([ 2.7183,  7.3891, 20.0855])
```
```
>>> x.log()
```
```
tensor([ 0.0000,  0.6931,  1.0986])
```
```
>>> x.pow(2)
```
```
tensor([1., 4., 9.])
```

Basic tensor operations

```
x = tensor([1., 2., 3.])
y = tensor([[1., 2.],[3., 4.]])
z = tensor([[1., 2.]])
```

- 6. A word about broadcasting
  - Addition, subtraction, multiplication and division

```
>>> x + 3          tensor([4., 5., 6.])
>>> x + x          tensor([2., 4., 6.])
```

  - Both commands work, because PyTorch will try it’s best to broadcast shapes for common operators such as addition (+), multiplication (*), etc

```
>>> y + x          Error, because the shapes (3,) and (2,2) cannot be broadcast
>>> y + z          tensor([[2., 4.],[4., 6.]])
>>> y + z.t()      tensor([[2., 3.],[5., 6.]])
```

Basic tensor operations

```
x = tensor([[ 1.,  2.],
            [ 3.,  4.],
            [ 5.,  6.]])
```

- 7. Max/min/sum/mean
  - Overall max/min/sum/mean

```
>>> x.min()        tensor(1.)
>>> x.sum()        tensor(21.)
>>> x.mean()       tensor(3.5)
```

  - Max/min/sum/mean on a specific axis

See example in Notebook

```
>>> x.sum(dim=0)        tensor([ 9., 12.])
>>> x.sum(dim=1)        tensor([ 3.,  7., 11.])
```

Basic tensor operations

- 8. Dot product and matrix multiplication
  - Dot product

```
a = tensor([1., 2., 3.])
```

```
b = tensor([4., 5., 6.])
```

```
>>> torch.dot(a, b)
```

```
32
```

  - Matrix multiplication

```
a = tensor([[ 1.,  2.],
            [ 3.,  4.]])
```

```
b = tensor([[ 5.,  6.],
            [ 7.,  8.]])
```

```
>>> torch.mm(a, b)
```

```
tensor([[19., 22.],
        [43., 50.]])
```

Basic tensor operations

- 9. Commonly-used tensor operations in PyTorch

`torch.t` `torch.transpose`

`torch.cat` `torch.stack` `torch.chunk` `torch.unbind`

`torch.Tensor.view` `torch.Tensor.reshape` `torch.Tensor.expand`

`torch.squeeze` `torch.unsqueeze`

`torch.min` `torch.max` `torch.sum` `torch.mean`

`torch.eq` `torch.ne` `torch.mm` `torch.bmm`

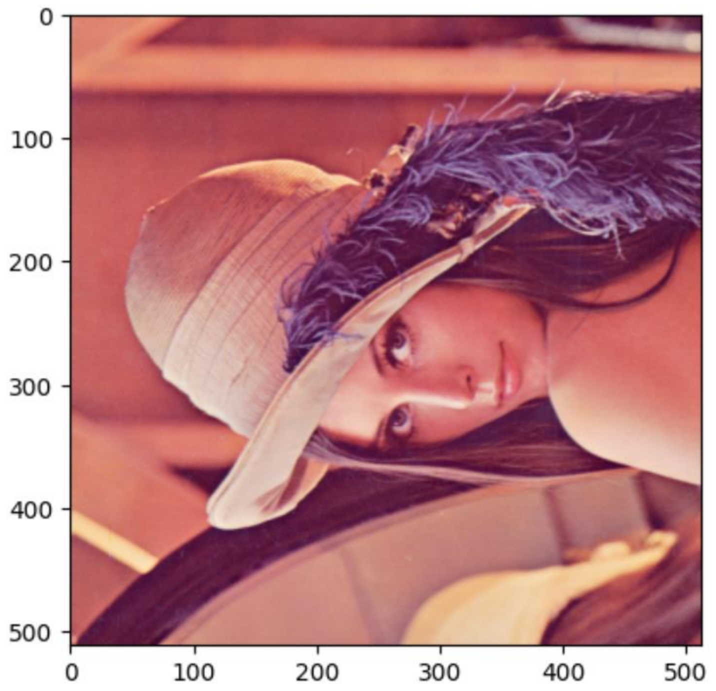
`torch.index_select` `torch.masked_select` `torch.Tensor.masked_fill_`

`torch.gather` `torch.Tensor.scatter_`

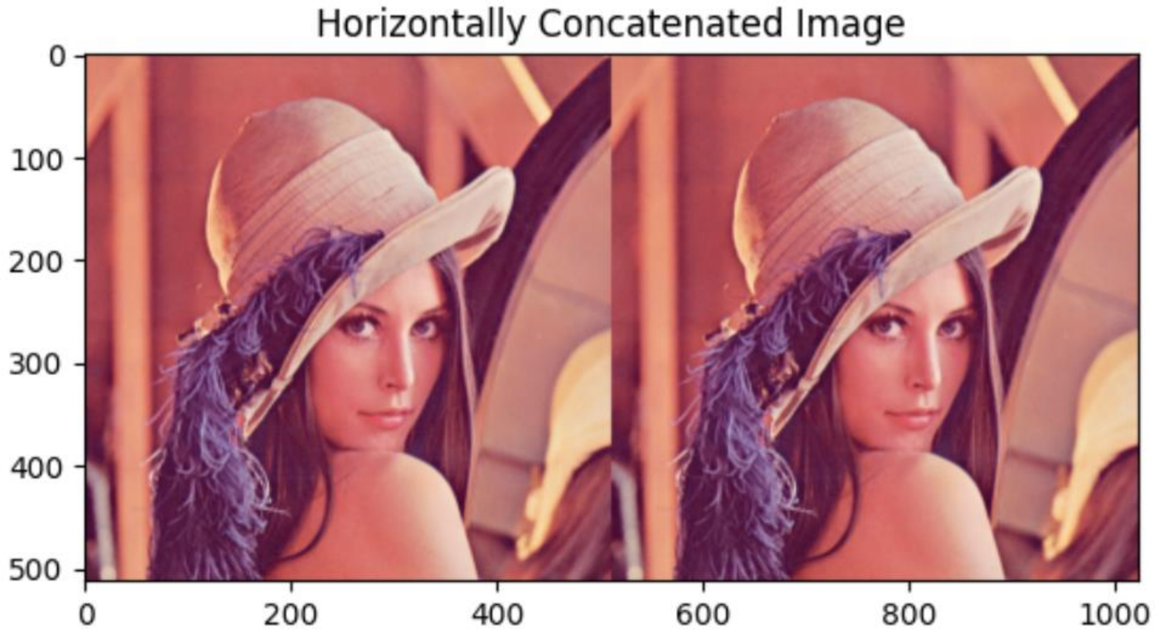
Basic tensor operations

- 9. Commonly-used tensor operations in PyTorch

`torch.t`



`torch.cat`



Basic tensor operations

- Practice:

- 1. **Softmax on a vector**
  - w is a vector of size d
  - $\text{softmax}(w)_i = \frac{e^{w_i}}{\sum_{k=1}^{d} e^{w_k}}$

- **2. KL divergence between two categorical distributions**
  - p and q are two d-dimensional categorical distributions
  - $KL(q,p) = \mathbb{E}_q \left[\log \frac{q}{p}\right] = \sum_x q(x) \log \frac{q(x)}{p(x)}$

Training through auto-gradient

- How to train a deep learning model?

  // Forward pass to make a prediction
  - Convert input into floating-point numbers
  - Use deep learning models to do transformation
    - A sequence of layers and intermediate representations
  - Convert last representations into output

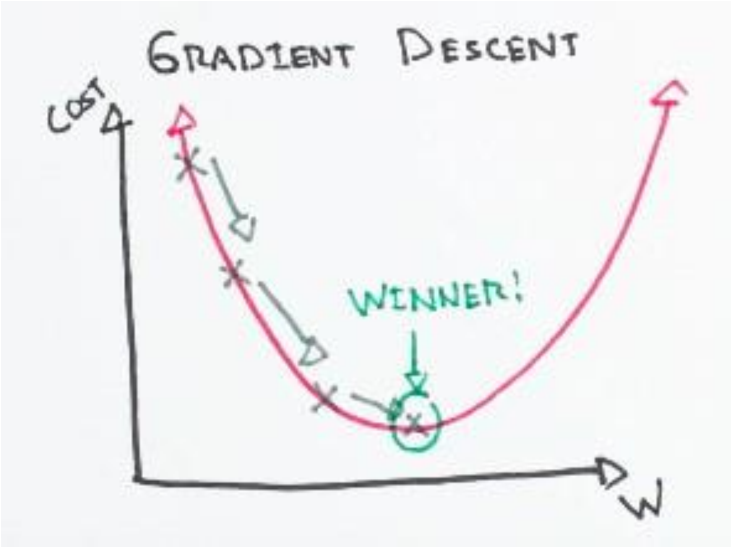
  // Define a loss function
  - Compute a scalar to measure the difference between predictions and targets

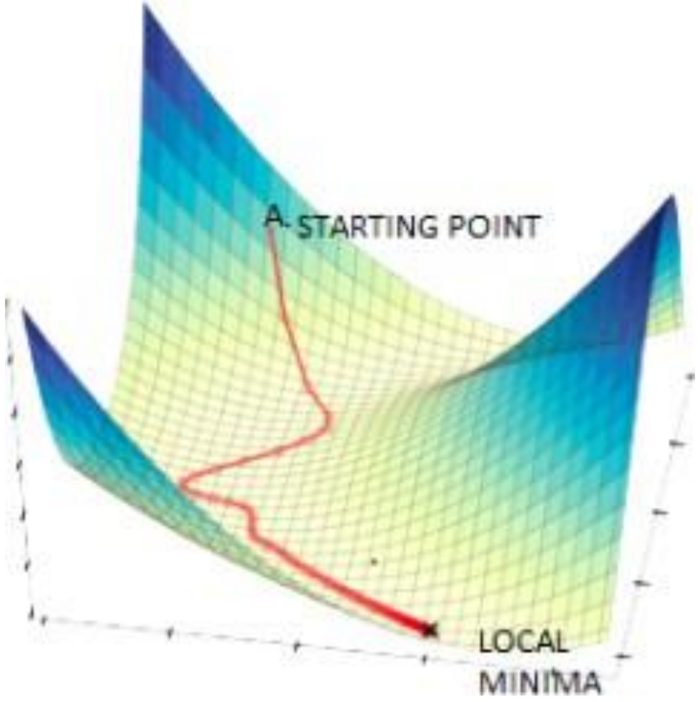
  // Model learning
  - Update model parameters

Training through auto-gradient

- How to update model parameters?
  - Gradient descent

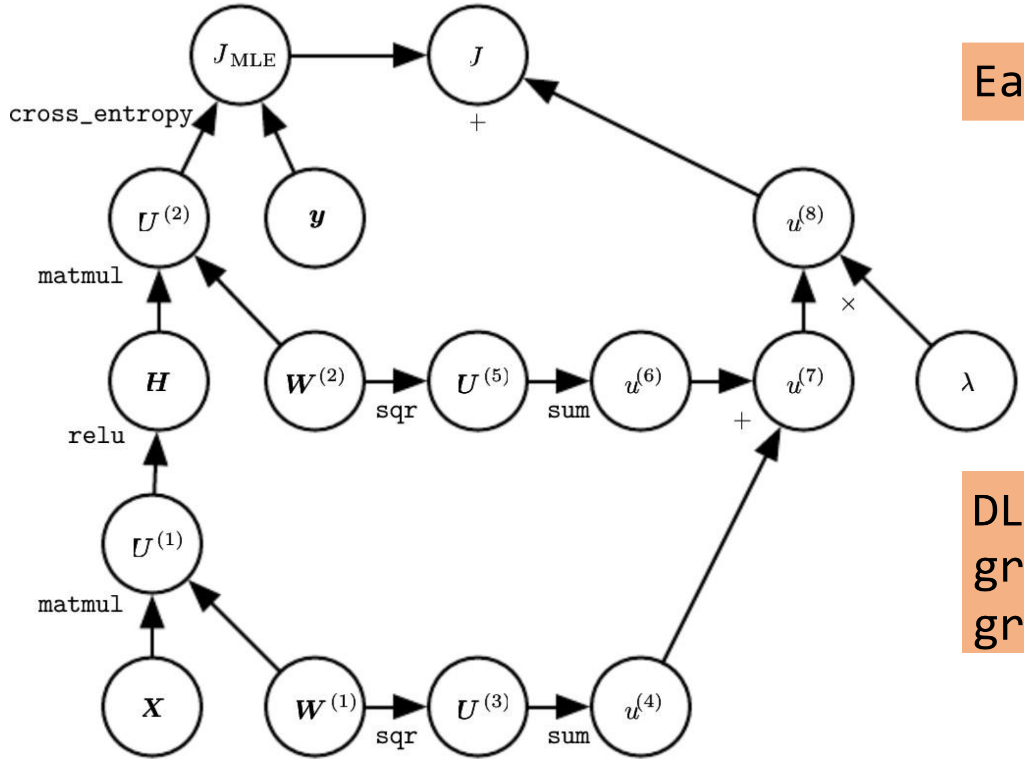
$$w^{(n+1)} = w^{(n)} - \epsilon \frac{\partial \mathcal{L}}{\partial w^{(n)}}$$





Training through auto-gradient

- Deep Learning book Chapter 6, computation graph example



Each arrow is differentiable!!!

DL framework avoids manual gradient computation with auto-gradient!

Training through auto-gradient

- PyTorch's autograd: backpropagate all things
  - Require gradients for a tensor

```
>>> w = torch.tensor([2.0], requires_grad=True)
```

  - Compute a scalar loss $\mathcal{L}$

```
>>> L = f(w)
```

  - Compute the gradient $\frac{\partial \mathcal{L}}{\partial w}$

```
>>> L.backward()
```

  - Print the gradient

```
>>> print(w.grad)
```

- *Or compute using

```
>>> torch.autograd.grad(outputs=L, inputs=w)[0]
```

Training through auto-gradient

- Practice:
  - Compute the derivative of

$$f(x) = \exp(x^3 \sin(\log x)) \text{ at } x = 2$$

Training through auto-gradient

- Practice:
  - Compute the derivative of

$$f(x) = \exp(x^3 \sin(\log x)) \text{ at } x = 2$$

```python
# Define the function f(x) = exp(x^3 * sin(log(x)))
L = torch.exp(w.pow(3) * torch.sin(w.log()))
# Perform backpropagation to compute the gradient by autograd
L.backward()
```

Break time!

Pytorch Part II:
Deep Learning Pipeline

Code: Collab Notebook

Content

- 1 General Pipeline: main components
- 2 Example 1: Iris Classification
- 3 Example 2: Image Classification
    - Practice: Model Capacity
    - Practice: Focal Loss
- 4 Coding suggestions

1 General Pipeline

- Dataset
- Model
- Train model (optimizer, loss function)
- Test/Evaluate model

1 General Pipeline

- Dataset
- Model
- Train model (optimizer, loss function)
- Test/Evaluate model

```
>>> class Dataset(data.Dataset):
>>>     def __init__(self, some_parameter):
>>>         super(Model, self).__init__()
>>>         self.X = …
>>>         self.y = …
>>>
>>>     def __len__(self):
>>>         return len(self.y)
>>>
>>>     def __getitem__(self, index):
>>>         X_sample = self.X[index]
>>>         y_sample = self.y[index]
>>>         return X_sample, y_sample
```

1 General Pipeline

- Dataset
- Model
- Train model (optimizer, loss function)
- Test/Evaluate model

```
>>> class Model(nn.Module):
>>>     def __init__(self, some_parameter):
>>>         super(Model, self).__init__()
>>>         self.function_1 = …
>>>         self.function_2 = …
>>>         self.function_3 = …
>>>
>>>     def forward(self, x):
>>>         intermediate_1 = self.function_1(x)
>>>         intermediate_2 = self.function_2(intermediate_1)
>>>         output = self.function_3(intermediate2)
>>>         return output
```

```
y = model(x)
y = model.forward(x)
```

1 General Pipeline

- Dataset
- Model
- Train model (optimizer, loss function)
- Test/Evaluate model

```
>>> dataset = Dataset()
>>> train_dataloader = data.DataLoader(dataset, batch_size=10, …)
>>> model = Model()
>>> optimizer = optim.SGD(model.parameters(), lr=0.001)
>>> loss_function = nn.CrossEntropyLoss()
>>>
>>> for e in range(100):
>>>     for batch in train_dataloader:
>>>         X, y = batch
>>>         y_pred = model(X) # y_pred = model.forward(X)
>>>         loss = loss_function(y_pred, y_label)
>>>         loss.backward()
```

1 General Pipeline

- Dataset
- Model
- Train model (optimizer, loss function)
- **Test/Evaluate model**

```
>>> dataset = Dataset()
>>> test_dataloader = data.DataLoader(dataset, batch_size=10, …)
>>>
>>> for batch in test_dataloader:
>>>     X, y = batch
>>>     y_pred = model(X)
>>>     acc = calculate_accuracy(y_pred, y)
```

2 Example: Iris Classification

2.1 Data preparation

2.2 Linear model + gradient descent

2.3 Customized model

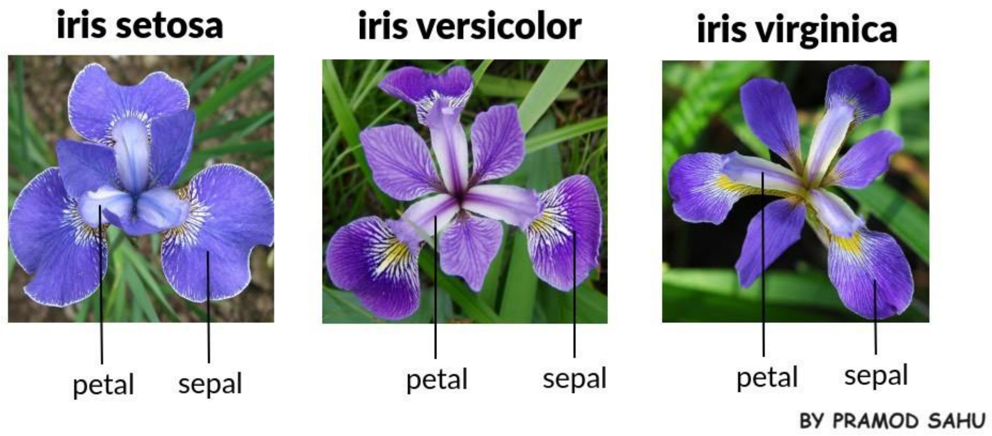
2.4 Mini-batch (stochastic) gradient descent

2.1 Data preparation

Iris: 150 samples

- each has four features (sepal length, sepal width, petal length, petal width)
- each belongs to one of the three classes/types of Iris plant (Setosa, Versicolour, Virginica)

```
          X              y
[5.1  3.5  1.4  0.2]     0
[7.   3.2  4.7  1.4]     1
[6.3  3.3  6.   2.5]     2
```



2.1 Data preparation

```
>>> from sklearn import datasets
>>> from sklearn.model_selection import train_test_split
>>>
>>> def load_data():
>>>     iris = datasets.load_iris()
>>>     X = iris.data      # 150 * 4
>>>     y = iris.target    # 150
>>>     return X, y
>>>
>>> X, y = load_data()
>>> X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
>>>
>>> device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')
>>> X_train = torch.FloatTensor(X_train).to(device)  # 120 * 4
>>> X_test = torch.FloatTensor(X_test).to(device)     # 30 * 4
>>> y_train = torch.LongTensor(y_train).to(device)   # 120
>>> y_test = torch.LongTensor(y_test).to(device)      # 30
```

2.2 Linear model + gradient descent

- Linear model:

$$\hat{y} = W^T X$$

- Loss function:

$$\ell(\hat{y}, y) = \text{Cross-Entropy}(\hat{y}, y)$$

- Gradient descent:

$$\begin{aligned} W^{t+1} &= W^t - \eta \nabla_{W^t} \ell(\hat{y}, y) \\ &= W^t - \frac{\eta}{N} \sum_{i=1}^{N} \nabla_{W^t} \ell(\hat{y}_i, y_i), \text{ where } N \text{ is the dataset size} \end{aligned}$$

2.2 Linear model + gradient descent

```
>>> model = nn.Linear(4, 3).to(device)
>>>
>>> optimizer = optim.SGD(model.parameters(), lr=0.001)
>>> criterion = nn.CrossEntropyLoss()
>>>
>>> ########## Training ##########
>>> model.train()
>>> for e in range(200):
>>>     ########## Get prediction ##########
>>>     y_train_pred = model(X_train)
>>>     loss = criterion(y_train_pred, y_train)
>>>     ########## Clean-up gradients from previous steps ##########
>>>     optimizer.zero_grad()
>>>     ########## Calculate gradients for current step ##########
>>>     loss.backward()
>>>     ########## Update weights using SGD for current step ##########
>>>     optimizer.step()
>>>     print('Epoch: {}\tLoss: {:.5f}'.format(e, loss.item()))
```

2.2 Linear model + gradient descent

```
>>> ########## Evaluation ##########
>>> model.eval()
>>> y_test_pred_output = model(X_test)
>>> _, y_test_pred = torch.max(y_test_pred_output, 1)
>>> acc = torch.true_divide(torch.sum(y_test_pred == y_test), y_test_pred.size()[0])
>>> print('accuracy: {}'.format(acc))
```

2.2 Linear model + gradient descent

- Common arguments:

```
>>> optimizer = optim.SGD(model.parameters(), lr=learning_rate, weight_decay=1e-5)
```

Recall:

$$W^{t+1} = W^t - \frac{\eta}{N} \sum_{i=1}^{N} \nabla_{W^t} \ell(\hat{y}_i, y_i), \text{ where } \ell(\hat{y}, y) = \text{Cross-Entropy}(\hat{y}, y) + \lambda ||W||_2^2$$

2.2 Linear model + gradient descent

- A wide range of optimizers
  - SGD: the classical optimizer
  - RMSprop: a self-adaptive optimizer   adaptive gradient
  - Adam: a self-adaptive optimizer

    adaptive gradient and lr

Loss functions

- Loss functions are also non-parametric layers in PyTorch
  - Classification

    `nn.NLLLoss` $\mathscr{L} = -\log\left(pred_{target}\right)$

    `nn.CrossEntropyLoss` $\mathscr{L} = -\log\left(softmax(pred)_{target}\right)$

  - Regression

    `nn.MSELoss` $\mathscr{L} = (pred - target)^2$

    `nn.SmoothL1Loss` $\mathscr{L} = \min\left((pred - target)^2, |pred - target|\right)$

- **<u>Caveat: the model's output should be consistent with the loss</u>**
- For `nn.NLLLoss`, it takes a log-probability distribution as input
  - The last layer in the forward function should be `F.LogSoftmax`
- For `nn.CrossEntropyLoss`, it takes unbounded logits as input
  - The last layer in the forward function shouldn't be any activation function
- For regression losses, it takes unbounded values as input

2.3 Customize models

- Basic recipe for customizing a model
    1. Define the modules in __init__
    2. Define the forward function
    3. ~~Define the backward function~~ Pytorch takes care of it :)

- Step 1 let the framework know **what to train**.
- Step 2 let the framework know **what the model is**.
- DL frameworks will automatically infer step 3 from step 2 (aka. autograd)

2.3 Customize models

Before in 2.2:

```
model = nn.Linear(4, 3).to(device)
```

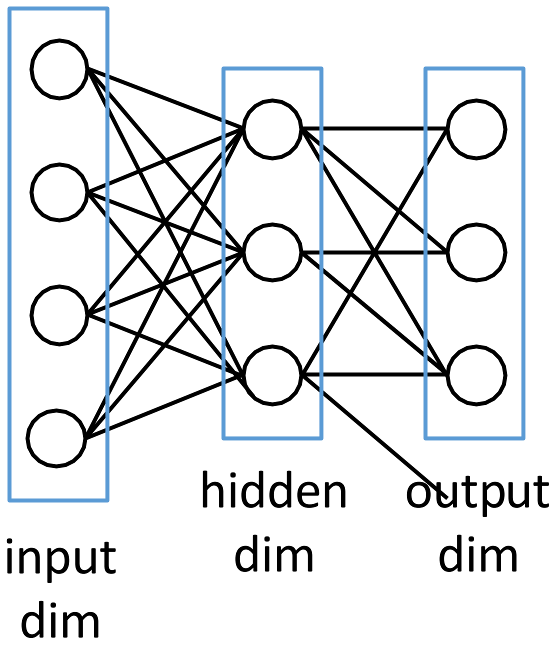
Now customize our own model:

```
>>> class MultiLayerPercptron(nn.Module):
>>>     def __init__(self):
>>>         super(MultiLayerPercptron, self).__init__()
>>>         self.fc1 = nn.Linear(4, 100)
>>>         self.fc2 = nn.Linear(100, 3)
>>>         return
>>>
>>>     def forward(self, x):
>>>         x1 = self.fc1(x)           # (120, 4)   => (120, 100)
>>>         x2 = self.fc2(x1)          # (120, 100) => (120, 3)
>>>         return x2
>>>
>>> model = MultiLayerPercptron().to(device)
```

2.3 Customize models

- In the MLP example, the parameters are two linear layers

```
>>> class MLP(nn.Module):
>>>     def __init__(self, input_dim, hidden_dim, 
output_dim):
>>>         super(MLP, self).__init__()
>>>         self.fc1 = nn.Linear(input_dim, hidden_dim)
>>>         self.fc2 = nn.Linear(hidden_dim, output_dim)
```



- Here `nn.Linear` is a convenient interface to define all the parameters within a linear layer

Customize models

- Let's see how to put these ingredients into implementation
  - Inherit a class from `nn.Module`
  - Parameters are defined in `__init__()`
  - Forward function is defined as `forward()`

```
>>> class MyModel(nn.Module):
>>>     def __init__(self, ...):
>>>         super(MLP, self).__init__()
>>>         # here are parameter definitions
>>>         self.xxx = ...
>>>
>>>     def forward(self, ...):
>>>         # here is the forward function
>>>         return ...
```

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

Customize models

- Forward function of MLP
- Very similar to NumPy

```
>>> class MLP(nn.Module):
>>>     def forward(self, input):
>>>         input = input.flatten(1)
>>>         hidden = F.relu(self.fc1(input))
>>>         output = F.softmax(self.fc2(hidden), dim=-1)
>>>         return output
```

- `self.fc1` and `self.fc2` are called as functions, i.e. linear transformation
- `F.relu` and `F.softmax` are non-parameteric functions
  - They have no trainable parameters
  - We don't need to define them in `__init__(), but it's good practice to generally do so for all functions in F (especially for dropout layer)`

Customize models

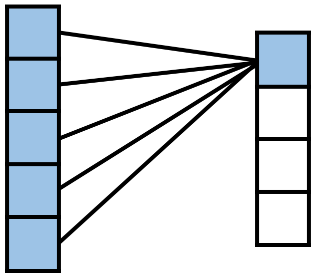
- `nn.Sequential` is a convenient wrapper for multiple layers
- layers are applied in their definition order

```
>>> class MLP(nn.Module):
>>>     def __init__(self, input_dim, hidden_dim, output_dim):
>>>         super(MLP, self).__init__()
>>>         self.model = nn.Sequential(
                         nn.Linear(input_dim, hidden_dim),
                         nn.Linear(hidden_dim, output_dim)
                     )
>>>
>>>     def forward(self, input):
>>>         input = input.flatten(1)
>>>         output = F.softmax(self.model(input), dim=-1)
>>>         return output
```

Common building blocks

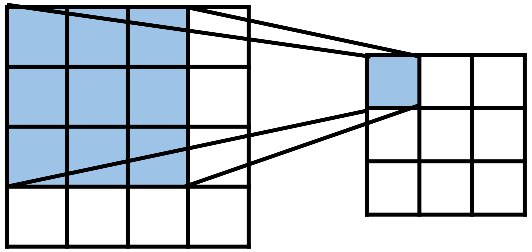
- Parametric layers
  - Linear layer (aka. fully connected / dense layer)

```
nn.Linear(in_features, out_features, bias=True)
```



  - Convolution layer

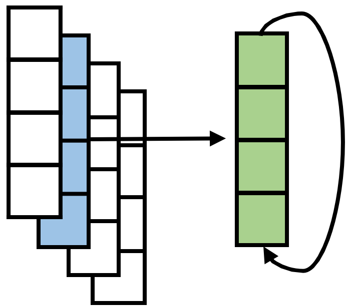
```
nn.Conv2d(in_channels, out_channels, kernel_size, stride=1)
```



Common building blocks

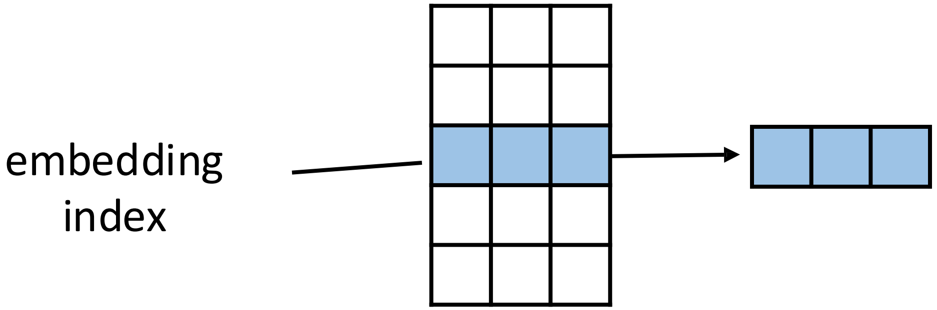
- Parametric layers
  - Recurrent layer (multi-layer)

```
nn.LSTM(input_size, hidden_size, num_layers=1, bias=True)
```



  - Embedding layer

```
nn.Embedding(num_embedding, embedding_dim, max_norm=None, norm_type=2.0)
```



Common building blocks

- Non-parametric layers
  - Activation function

    ```
    F.relu(input)
    F.sigmoid(input)
    F.tanh(input)
    F.softmax(input, dim=None)
    ```

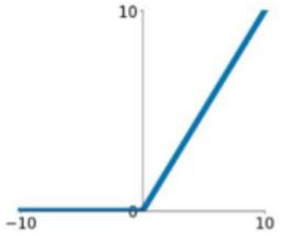
  - Pooling function

    ```
    F.avg_pool2d(kernel_size, stride=None)
    F.max_pool2d(kernel_size, stride=None)
    ```

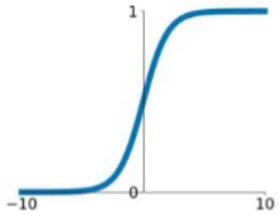
  - Dropout layer

    ```
    nn.Dropout(p=0.5)
    ```

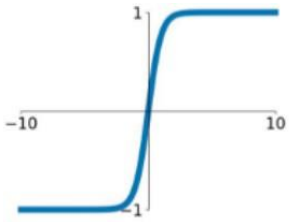
    ~~`F.dropout(input, p=0.5, training=True)`~~ Not recommended!



ReLU



sigmoid



tanh

2.4 Mini-Batch (stochastic) gradient descent

- Customized model:

$$\hat{y} = f_W(X)$$

- Loss function:

$$\ell(\hat{y}, y) = \text{Cross-Entropy}(\hat{y}, y)$$

- As mentioned at step 2, gradient descent has some limitations when it comes to larger dataset.

  Solution: mini-batch (stochastic) gradient descent

$$\begin{aligned} W^{t+1} &= W^t - \eta \nabla_{W^t} \ell(\hat{y}, y) \\ &= W^t - \frac{\eta}{B} \sum_{i=1}^{B} \nabla_{W^t} \ell(\hat{y}_i, y_i), \end{aligned}$$

where B is the batch size

2.4 Mini-Batch (stochastic) gradient descent

(1) Wrap-up customized dataset with torch.utils.data.Dataset

```
>>> class IrisDataset(data.Dataset):
>>>     def __init__(self, X, y):
>>>         self.X = X
>>>         self.y = y
>>>         return
>>>
>>>     def __len__(self):
>>>         return len(self.X)
>>>
>>>     def __getitem__(self, index):
>>>         X_sample = torch.FloatTensor(self.X[index])
>>>         y_sample = torch.LongTensor([self.y[index]])
>>>         return X_sample, y_sample
>>>
>>> train_dataset = IrisDataset(X_train, y_train)
>>> test_dataset = IrisDataset(X_test, y_test)
```

2.4 Mini-Batch (stochastic) gradient descent

(2) Put dataset into torch.utils.data.DataLoader.

```
>>> train_dataloader = data.DataLoader(train_dataset, batch_size=10, shuffle=True)
>>> test_dataloader = data.DataLoader(test_dataset, batch_size=10, shuffle=False)
```

Once we have the dataloader, then we can iterate over the whole dataset batch by batch.

```
>>> for e in range(100):
>>>     for batch in dataloader:
>>>         X_batch, y_batch = batch
>>>         ……
```

Recall in GD, what we did is following:

```
>>> for e in range(100):
>>>     X_train, y_train = …
>>>     ……
```

2.4 Mini-Batch (stochastic) gradient descent

(2) Put dataset into torch.utils.data.DataLoader.

```
>>> train_dataloader = data.DataLoader(train_dataset, batch_size=10, shuffle=True)
>>> test_dataloader = data.DataLoader(test_dataset, batch_size=10, shuffle=False)
>>>
>>> for e in range(100):
>>>     for batch in dataloader:
>>>         X, y = batch
>>>         ……
```

shuffle=False, batch_size=10

e=0: (0, 1, …, 9), (10, 11, …, 19), …

e=1: (0, 1, …, 9), (10, 11, …, 19), …

e=2: (0, 1, …, 9), (10, 11, …, 19), …

2.4 Mini-Batch (stochastic) gradient descent

(2) Put dataset into torch.utils.data.DataLoader.

```
>>> train_dataloader = data.DataLoader(train_dataset, batch_size=10, shuffle=True)
>>> test_dataloader = data.DataLoader(test_dataset, batch_size=10, shuffle=False)
>>>
>>> for e in range(100):
>>>     for batch in dataloader:
>>>         X, y = batch
>>>         ……
```

shuffle=True, batch_size=10

e=0: (29, 11, …, 93), (3, 50, …, 63), …

e=1: (72, 23, …, 18), (1, 31, …, 60), …

e=2: (45, 6, …, 89), (2, 18, …, 102), …

2.4 Mini-Batch (stochastic) gradient descent

(2) Put dataset into torch.utils.data.DataLoader.

```
>>> train_dataloader = data.DataLoader(train_dataset, batch_size=10, shuffle=True)
>>> test_dataloader = data.DataLoader(test_dataset, batch_size=10, shuffle=False)
>>>
>>> for e in range(100):
>>>     for batch in dataloader:
>>>         X, y = batch
>>>         ......
```

Why shuffling?

(1) Theoretical analysis: shuffling has smaller error upper bound, check this [paper](). Any examples?

(2) Empirically, shuffling works quite well.

(3) Intuitionally, shuffling

(a) better matches with the independent and identical distribution (IID) assumption in most of the ML setting.

(b) makes the training harder, thus the learned model is more robust w.r.t. generalization performance.

2.4 Mini-Batch (stochastic) gradient descent

(3) Iterate over the train dataloader // for training

```
>>> model.train()
>>> for e in range(200):
>>>     accum_loss = 0
>>>     for batch in train_dataloader:
>>>         X_train_batch, y_train_batch = batch
>>>         X_train_batch = X_train_batch.to(device)  # size: (batch_size, 4)
>>>         y_train_batch = y_train_batch.to(device)  # size: (batch_size, 1)
>>>         y_train_batch = y_train_batch.squeeze(1)  # size: (batch_size)
>>>         y_train_batch_pred = model(X_train_batch)
>>>         loss = criterion(y_train_batch_pred, y_train_batch)
>>>         optimizer.zero_grad()
>>>         loss.backward()
>>>         optimizer.step()
>>>         accum_loss += loss.item()
>>>     print('Epoch: {}\tLoss: {:.5f}'.format(e, accum_loss/len(train_dataloader)))
```

2.4 Mini-Batch (stochastic) gradient descent

(4) Iterate over the test dataloader // for evaluation

```
>>> model.eval()
>>> y_test, y_test_pred = [], []
>>> for batch in test_dataloader:
>>>     X_test_batch, y_test_batch = batch
>>>     X_test_batch = X_test_batch.to(device)    # size: (batch_size, 4)
>>>     y_test_batch = y_test_batch.to(device)    # size: (batch_size, 1)
>>>     y_test.append(y_test_batch)
>>>     y_test_pred_batch = model(X_test_batch)   # size: (batch_size, 1)
>>>     y_test_pred.append(y_test_pred_batch)
>>>
>>> y_test = torch.cat(y_test, dim=0)             # size: (30, 1)
>>> y_test = y_test.squeeze(1)                    # size: (30)
>>> y_test_pred = torch.cat(y_test_pred, dim=0)   # (30, 3)
>>> _, y_test_pred = torch.max(y_test_pred, 1)
>>> acc = torch.true_divide(torch.sum(y_test_pred == y_test), y_test_pred.size()[0])
>>> print('accuracy: {}'.format(acc))
```

3 Example 2: Image Classification

- Apply the pipeline to different datasets and models
- Practice on model capacity and focal loss

General Pipeline (review)

- Dataset
- Model
- Train model (optimizer, loss function)
- Test/Evaluate model

3 Example 2: Image Classification

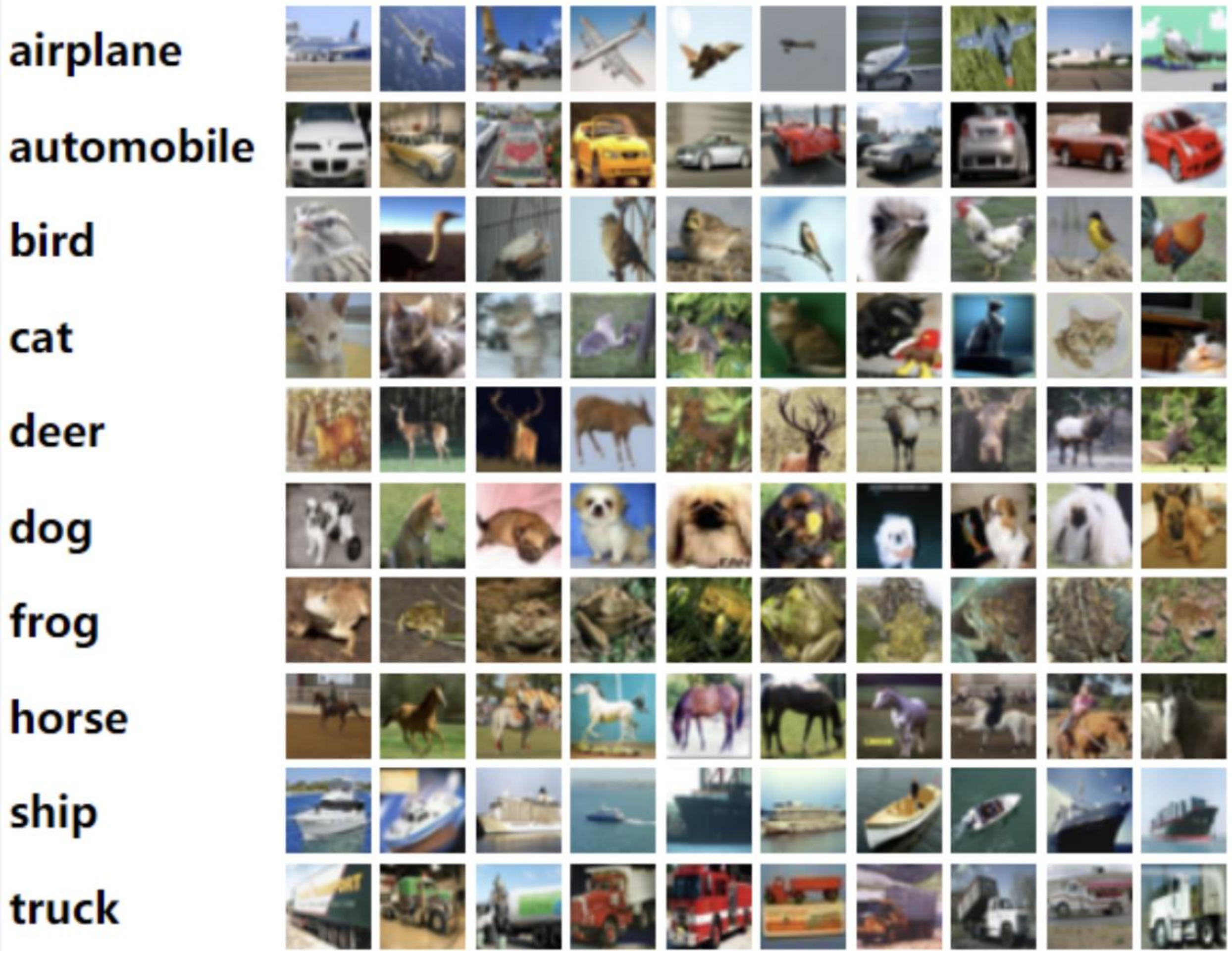
- Same pipeline
- Different Dataset
  - We use datasets provided by torchvision
- Different Model

```
>>> train_dataset = torchvision.datasets.MNIST('../data', train=True, transform=transform)
>>> test_dataset = torchvision.datasets.MNIST('../data', train=False, transform=transform)
>>>
>>> train_dataloader = data.DataLoader(train_dataset, batch_size=128, shuffle=True)
>>> test_dataloader = data.DataLoader(test_dataset, batch_size=128, shuffle=False)
```

```
>>> train_dataset = torchvision.datasets.CIFAR10('../data', train=True, transform=transform)
>>> test_dataset = torchvision.datasets.CIFAR10('../data', train=False, transform=transform)
>>>
>>> train_dataloader = data.DataLoader(train_dataset, batch_size=128, shuffle=True)
>>> test_dataloader = data.DataLoader(test_dataset, batch_size=128, shuffle=False)
```

CIFAR10

- 32x32 color images of 10 classes



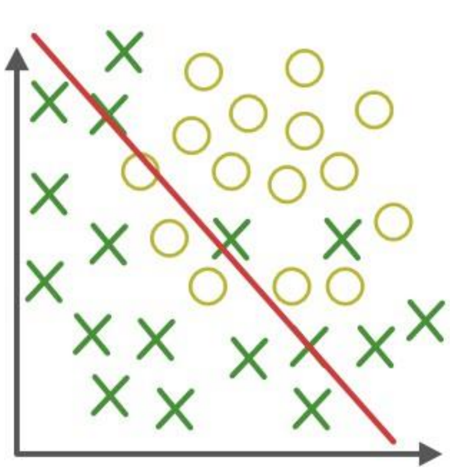
3 Example 2: Image Classification

- Same pipeline
- Different Dataset
- Different Model: 2-layer CNN as feature layers + 2-layer-MLP as classifier

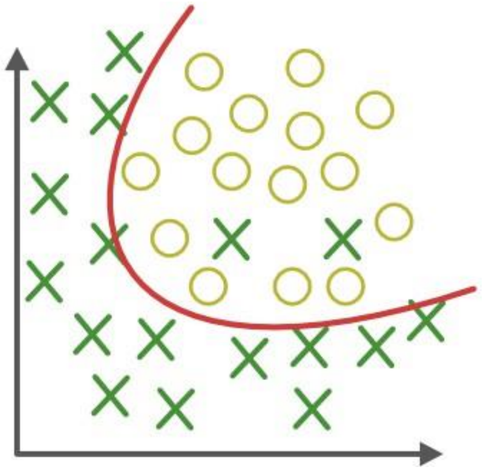
```python
class SimpleCNN(nn.Module):
    def __init__(self, num_classes=10):
        super(SimpleCNN, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=3, padding=1),  # Input channels: 3, Output channels: 32
            nn.ReLU(),
            nn.MaxPool2d(2, 2),  # Output size: 16x16
            nn.Conv2d(32, 64, kernel_size=3, padding=1),  # Output channels: 64
            nn.ReLU(),
            nn.MaxPool2d(2, 2),  # Output size: 8x8
        )
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(64 * 8 * 8, 256),
            nn.ReLU(),
            nn.Linear(256, num_classes),
        )

    def forward(self, x):
        x = self.features(x)
        x = self.classifier(x)
        return x
```

Model capacity

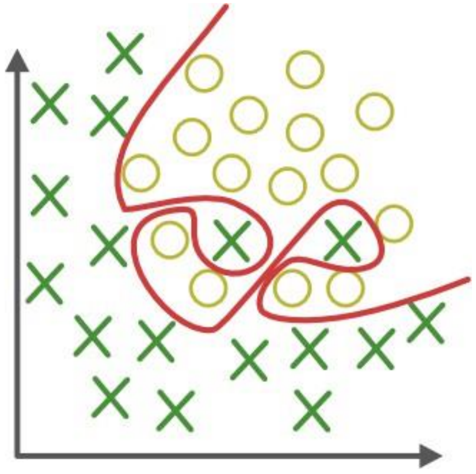
- Capacity: How powerful / complex a model is



Low capacity
Too simple to explain the observation



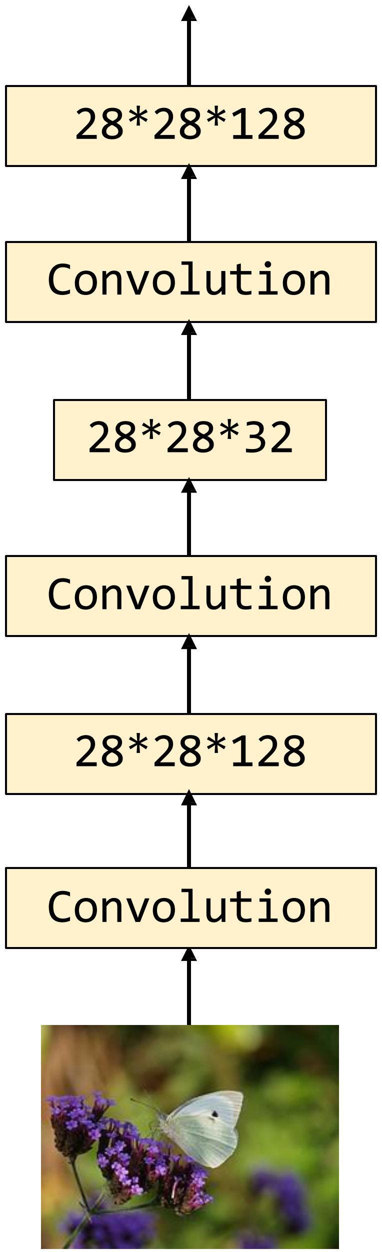
Appropriate capacity



High capacity
Too good to be true

Model capacity

- Capacity is determined by
  - Model architecture
  - Number of learnable parameters
  - Regularization / Dropout / Early stopping
  - ……



Model capacity

- We can obtain #parameter by

```
>>> sum(np.prod(param.shape) for param in net.module_.parameters())
```

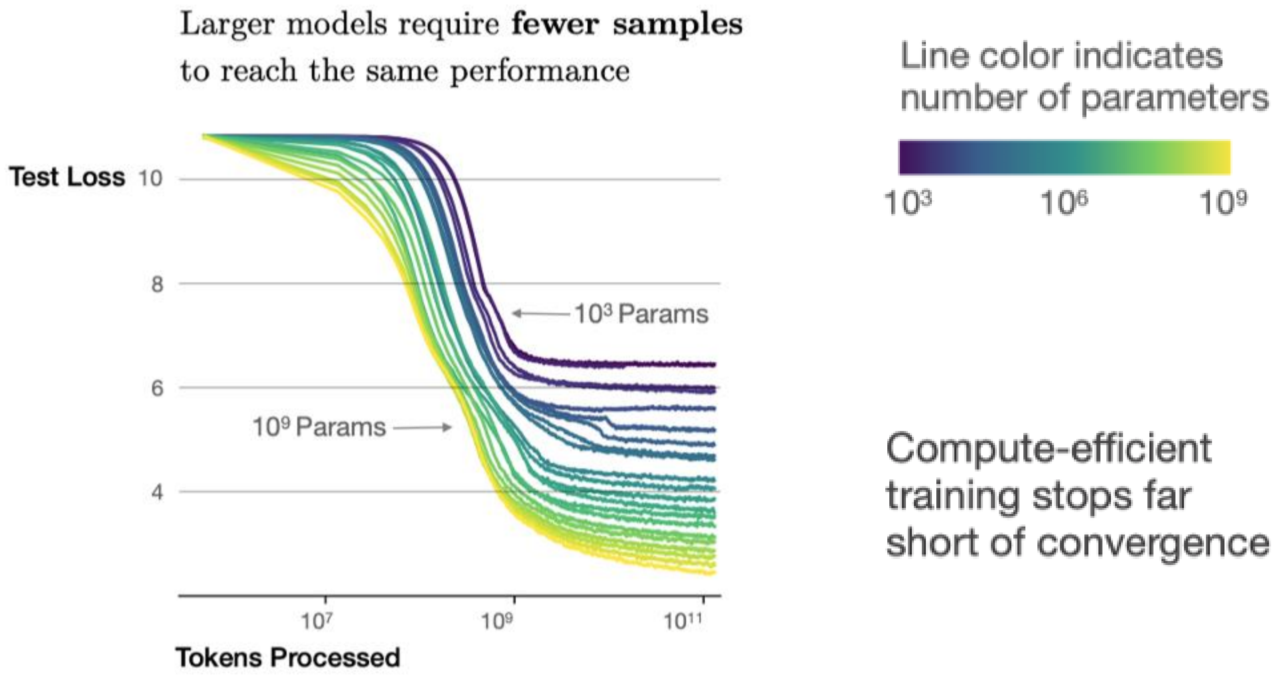
- ResNet18 has ~10M parameters
- GPT-3 has up to 175B parameters.

Model Name	n_{params}	n_{layers}	d_{model}	n_{heads}	d_{head}	Batch Size	Learning Rate
GPT-3 Small	125M	12	768	12	64	0.5M	6.0×10^{-4}
GPT-3 Medium	350M	24	1024	16	64	0.5M	3.0×10^{-4}
GPT-3 Large	760M	24	1536	16	96	0.5M	2.5×10^{-4}
GPT-3 XL	1.3B	24	2048	24	128	1M	2.0×10^{-4}
GPT-3 2.7B	2.7B	32	2560	32	80	1M	1.6×10^{-4}
GPT-3 6.7B	6.7B	32	4096	32	128	2M	1.2×10^{-4}
GPT-3 13B	13.0B	40	5140	40	128	2M	1.0×10^{-4}
GPT-3 175B or "GPT-3"	175.0B	96	12288	96	128	3.2M	0.6×10^{-4}

- Comparing #parameter across different architectures may not be reliable

Model capacity

- Scaling Law



- Comparing #parameter across different architectures may not be reliable

Practice: Model Capacity

- Explore different model capacity
  - Example 1: #hidden units in “classifier”, e.g. 16, 64, 256, 1024, 4096
  - Example 2: Add or remove “convolution layers”
  - Example 3: Modify classifier with more regularization
- Other practices
  - Change dropout ratio
  - Increase and decrease epochs
  - Adding / removing data samples
  - Change optimizer
- What is the best performance you can get?

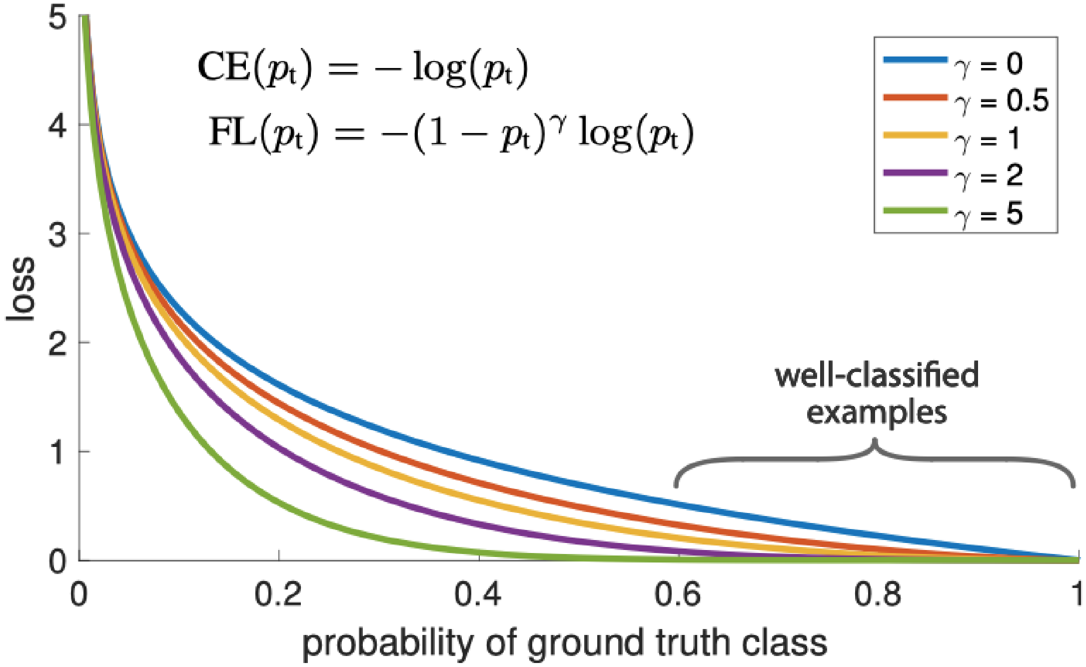
Practice: Focal Loss

- Focal Loss is defined as:

$$\mathrm{CE}(p, y) = \begin{cases} -\log(p) & \text{if } y = 1 \\ -\log(1-p) & \text{otherwise.} \end{cases}$$

$$p_{\mathrm{t}} = \begin{cases} p & \text{if } y = 1 \\ 1-p & \text{otherwise,} \end{cases}$$

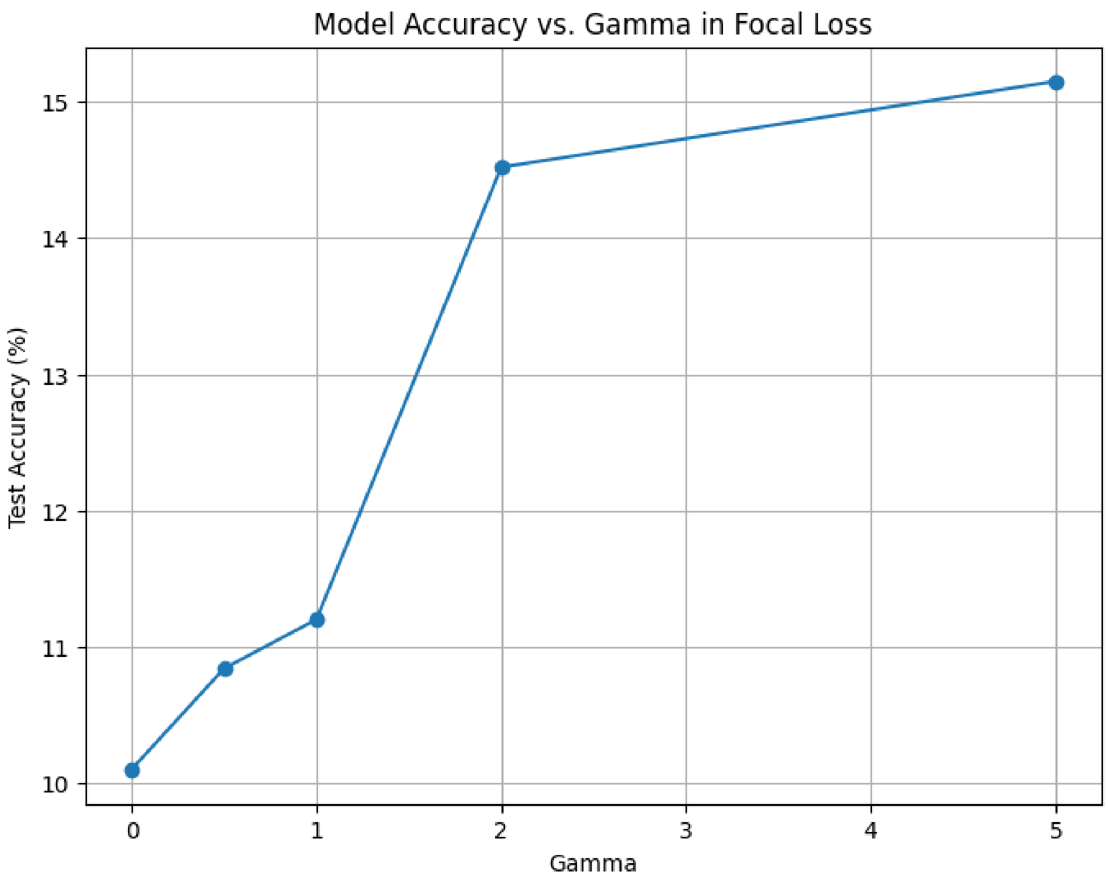
$$\mathrm{CE}(p_{\mathrm{t}}) = -\log(p_{\mathrm{t}})$$
$$\mathrm{FL}(p_{\mathrm{t}}) = -(1-p_{\mathrm{t}})^{\gamma}\log(p_{\mathrm{t}})$$



- Hint 1: Use F.log_softmax to get log-probabilities with numerical stability.
- then calculate p with log(p).exp()
- Hint 2: Use torch.gather to obatin p_t from the predicted distribution

Practice: Focal Loss

- Expected Output

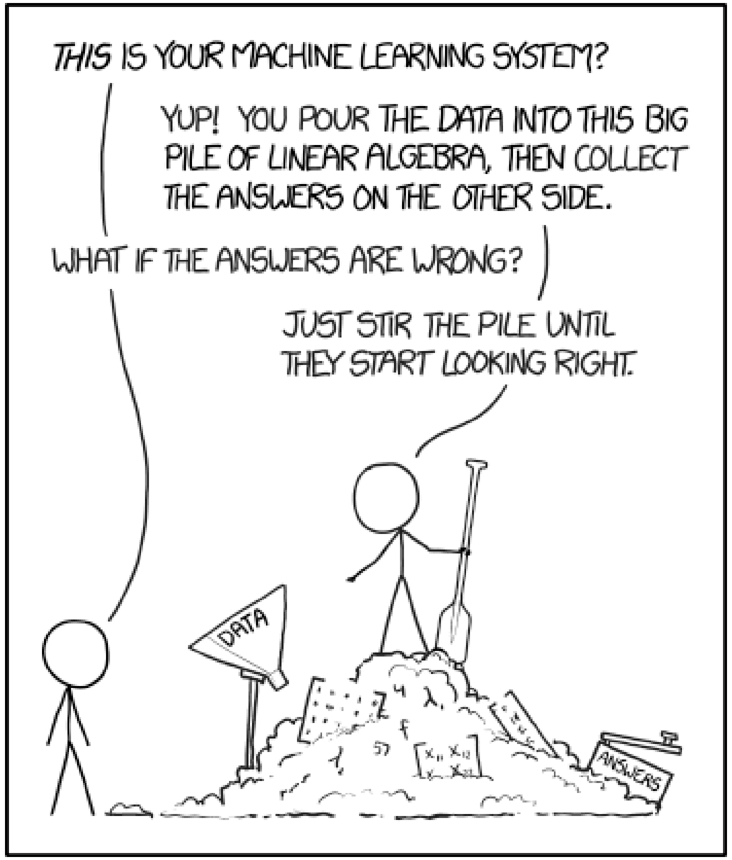


Other Practices

- Change dropout ratio
- Increase and decrease epochs
- Adding / removing data samples
- Change learning rate / learning rate scheduler

Debug models

- *Only 10% of programming is coding. The other 90% is debugging.*
  - Would be better if we are aware of common mistakes!



- General suggestions
- Shape errors
- Model errors
- Model capacity
- Implementation details

General suggestions

- Figure out where the bug is. A recommended order is
    - Check whether the code can run (-> e.g. shape errors)
    - Check the evaluation code
    - Check the ground truth
    - Check optimizer and learning rate
    - Check model errors
    - Check model capacity

Shape errors

- Shape errors are the most common reason if the code can't run

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-24-036a79cd99d7> in <module>()
      6     device="cuda"
      7 )
----> 8 net.fit(train.data.to(torch.float32) / 255.0, train.targets)

/usr/local/lib/python3.6/dist-packages/torch/nn/functional.py in linear(input, weight, bias)
   1368     if input.dim() == 2 and bias is not None:
   1369         # fused op is marginally faster
-> 1370         ret = torch.addmm(bias, input, weight.t())
   1371     else:
   1372         output = input.matmul(weight.t())

RuntimeError: size mismatch, m1: [128 x 128], m2: [1 x 10] at THCTensorMathBlas.cu:290
```

- We can locate the error layer by the hint in the output

Shape errors

- Some typical shape errors

```
RuntimeError: Expected 4-dimensional input for 4-dimensional weight 64 3 7 7,
but got 2-dimensional input of size [128, 2352] instead
```

- It means we have the wrong tensor dimension
- We should reshape the input with `tensor.view` or `tensor.reshape`

```
RuntimeError: size mismatch, m1: [128 x 784], m2: [128 x 10]
```

- It means we have the wrong size
- We should check both definitions of the layer and the input data shape

Model errors

- Model errors are unreasonable model design
  - They may cause phenomenon like gradient vanishing or gradient explosion
  - They can pass all assertions and thus are hard to find
- When shall we think of model errors?
  - Training diverge or converge badly
  - Tuning optimizer and learning rate doesn't help

Model errors

- Common reasons for model errors
  - Consecutive transformation layers — gradient explode

    ```
    x = self.fc2(self.fc1(x))
    ```

  - Consecutive activation layers — gradient vanish

    ```
    x = F.sigmoid(F.sigmoid(x))
    ```

  - Typecast — unintended results

    ```
    x = torch.ones(..., dtype=torch.long)
    x = x / x.sum() # integer division
    ```

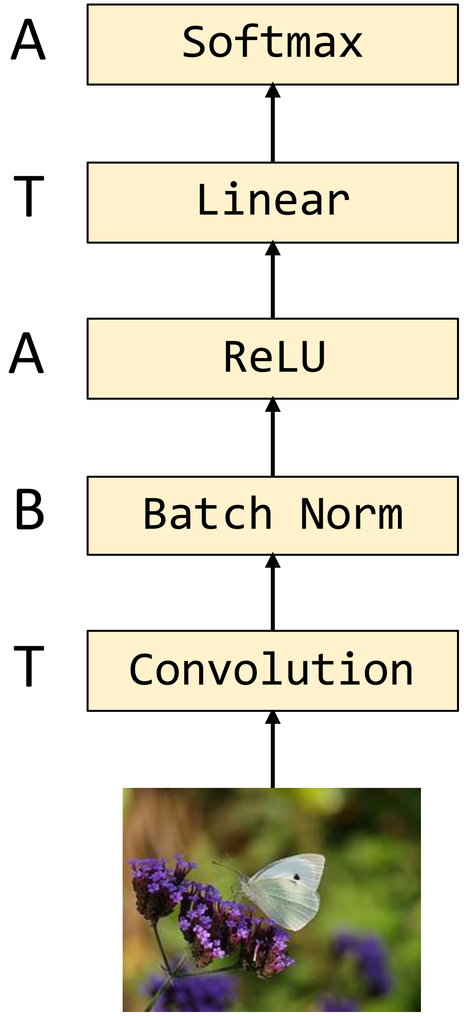
    7 / 3 leads to 2 instead of 2.333

    ```
    x = x / x.sum().float() # float division
    ```

Model errors

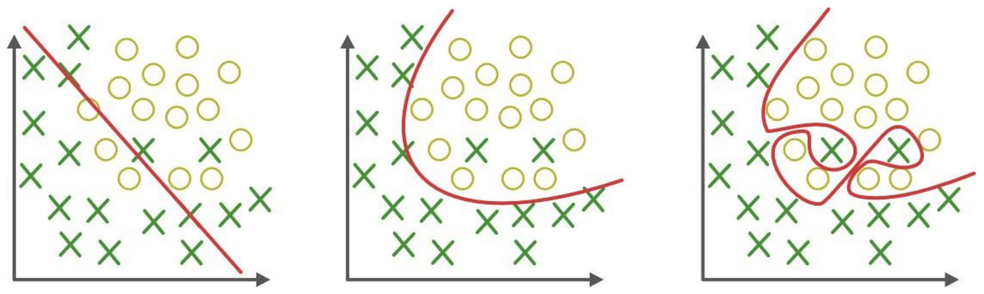
- Common reasons for model errors
  - Incorrect position of normalization layers
  - Incorrect last activation layer
  - ……
- Too difficult to remember? Mnemonic: T(B)A
  - Transformation
  - Batch Normalization (optional)
  - Activation

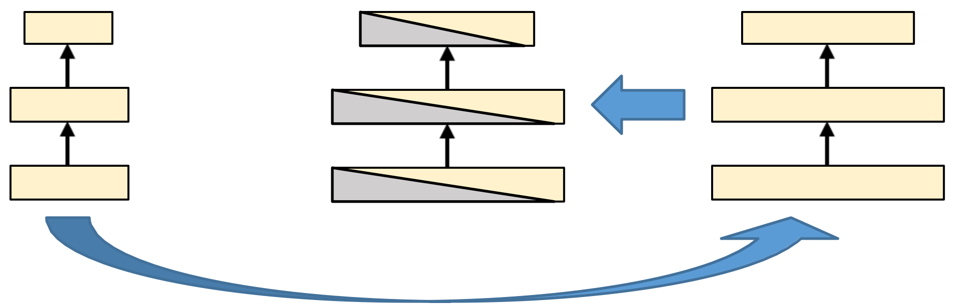
to be announced



Model capacity

- Tips for tuning capacity. A recommend order is
  1. Choose an architecture
  2. Increase the number of hidden units if training is bad
  3. Add regularization if training is good but validation is bad





Other details

- **Normalizing the input** before you go
  - Usually it's better to use an input scale around 1

```
>>> model.fit(train.data.to(torch.float32) / 255.0, train.targets)
```

- Balance different categories (50% pos v.s. 50% neg)
  - Otherwise neural networks tend to guess the most frequent category
    - Like something what we do for multiple choice questions :)
  - A good practice is to reweight each category by the reciprocal of its frequency

Other details

- Model initialization
  - Implicitly carried out in **<u>any</u>** of these lines

```
>>> mlp = MLP()
```

```
>>> resnet18 = torchvision.models.resnet18()
```

- Remember to re-initialize our model every trial

Other details

- Random seed matters (sometimes)
  - Model initialization
  - Data loading order
- Some situations may not be reproduced when using a different random seed
- Fix a random seed

```
>>> seed = 123
>>> torch.random.manual_seed(seed)
>>> torch.cuda.manual_seed_all(seed)
```

General suggestions

- Figure out where the bug is. A recommended order is
  - Check whether the code can run (-> e.g. shape errors)
  - Check the evaluation code
  - Check the ground truth
  - Check optimizer and learning rate
  - Check model errors
  - Check model capacity

Fast development: Rule of thumb

- Start with a small dataset and a short training epoch
  - Try different prototypes
  - Observe and find the best prototype

- Move to the full dataset
  - Try some variants of the best prototype
  - Find the best model
  - Increase to a long training epoch

Summary

- DL Frameworks are excellent helper for building own neural networks
- Tons of standard models / datasets are available in PyTorch
- Use GPU to speedup your training time by >1 magnitude
- Modify the standard ML pipeline we provided for your own need
- Check debug suggestions if powerful models don't work as expected
- Get your hands dirty and gain experiences.

Further readings

- Python / Numpy / Matplotlib tutorial
  - http://cs231n.github.io/python-numpy-tutorial/
- A simple neural network from scratch
  - https://medium.com/dair-ai/a-simple-neural-network-from-scratch-with-pytorch-and-google-colab-c7f3830618e0
- Language classification
  - https://colab.research.google.com/github/pytorch/tutorials/blob/gh-pages/_downloads/char_rnn_classification_tutorial.ipynb
- Dive into Deep Learning (PyTorch version)
  - https://github.com/dsgiitr/d2l-pytorch

Enjoy the PyTorch Journey! 🍺