MACHINE INTELLIGENCE COMMUNITY

Neural Networks Notes

September 26th, 2017
Justin Chen, Rex Wang

1. Recap

1.1 Goals of Machine Learning

1. Learn from data (**data-driven**) rather than explicit rules
2. Learn set of parameters so that the function can **generalize to new (unseen) data**
    a. Reduce **generalization error**
3. Deep learning is a subfield of machine learning that models the learning algorithm as **idealized versions of biological neurons**
4. Types of tasks: **Classification**, **Regression**, **Generation**

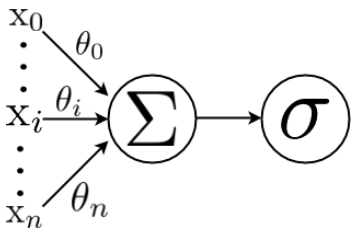
1.2 Gradient-Based Learning

1. Cost/Loss/Objective Functions
    a. Measures how far **hypothesis** function's (another term for learning algorithm) outputs are from the actual output **during training**
    b. Most **cost landscapes** are **non-convex** hence might get stuck in **local extrema**
    c. Unitless value. Larger -> worse
2. Stochastic Gradient Descent (SGD)
    a. Use to search **parameter space** to **minimize cost**
3. Computation Graphs (CG)
    a. A way to represent a **composition of functions**
    b. Visually more appealing for understanding the flow of gradients
    c. Easier to represent for distributed training
4. Backpropagation (BP)
    a. Derivative of cost function w.r.t. Parameters
    b. **Propagate error backwards** through computation graph and update parameter values using SGD

2. Artificial Neuron

2.1 Nonlinear Models

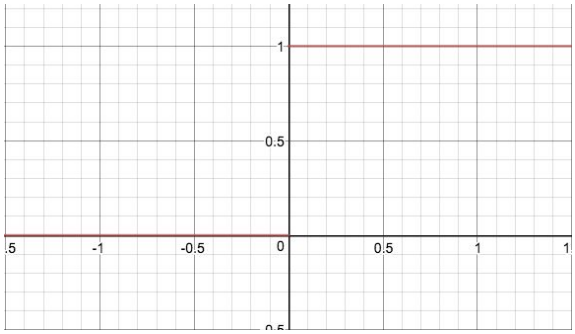
- Models **nonlinear data**
    - Data that cannot be approximated with a simple linear equation e.g. y = mx + b
- A **single neuron** is a linear equation composed into a **nonlinear function σ** like **sigmoid**



=

$$\sigma(\theta_0 x_0 + ... + \theta_n x_n)$$

- The original artificial neuron described by Walter Pitts and Warren McCulloch used the heavy-side nonlinear function (known as **unit step function**)



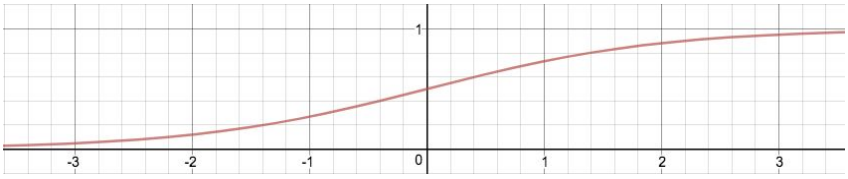
2.1.1 Logistic Regression

- Used for **binary classification** - two-way classification problem e.g. Cat or Dog
- Composes a linear equation into the **logistic sigmoid** function:

$$\frac{1}{1 + e^{-(\theta_0 x_0 + ... + \theta_n x_n)}}$$

Which can be expressed in vector notation as:

$$\frac{1}{1 + e^{-(\theta^\top \bar{x})}}$$



- If the model predicts a value greater than x = 0, then the input belongs to class 1, else the input belongs to class 0.
- The goal of classification is to draw **decision boundaries** across the **input space** (space of input data)
    - Here the decision boundary is at x = 0, when the neuron outputs 0.5.
- PyTorch for a neuron that takes in two inputs

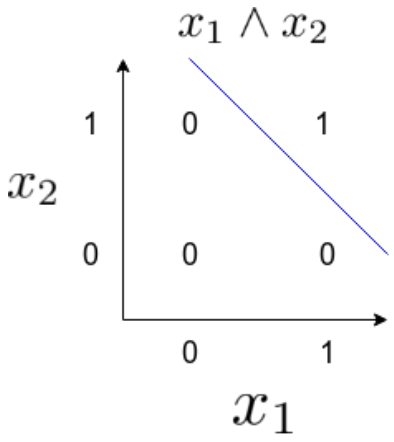
```
>>> from torch import sigmoid, randn
>>> model = sigmoid(randn(1,2))
>>> model

0.3615  0.1404
[torch.FloatTensor of size 1x2]
```

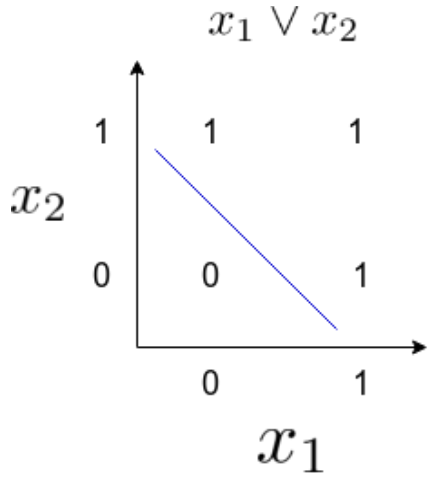
2.1.2 Capacity of Individual Neurons

XOR

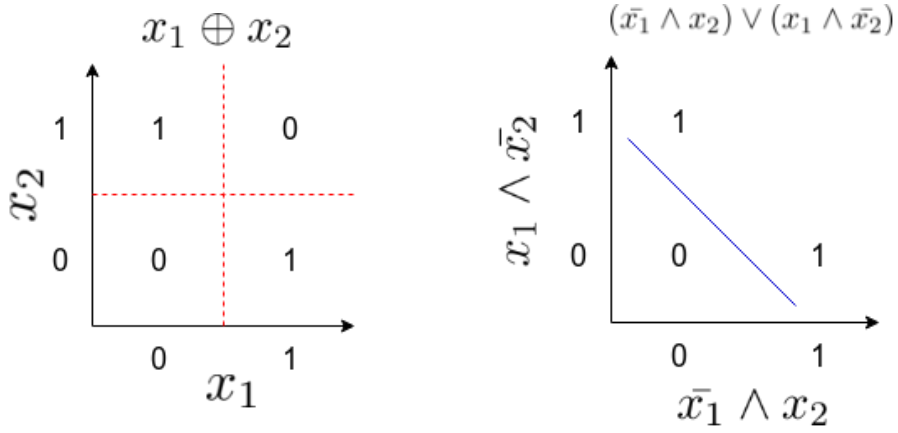
- Motivation: computing simple logical expressions
- It was shown that simple circuits/networks **could not** compute XOR [2]
    - Arguments like these lead to the AI Winter (lack of funding for AI research) and held back decades of progress in artificial intelligence research
- Logical AND and logical OR are both **linearly separable** - can draw a decision boundary between classes
    - In two dimensions this boundary is a straight line
    - In more than two dimensions this boundary is a **hyperplane**
- XOR function is **not linearly separable** in **two dimensions**
    - A single set of linear equations composed into a nonlinear function cannot compute XOR
    - However, it can be separated if transformed through different dimensions



Logical AND



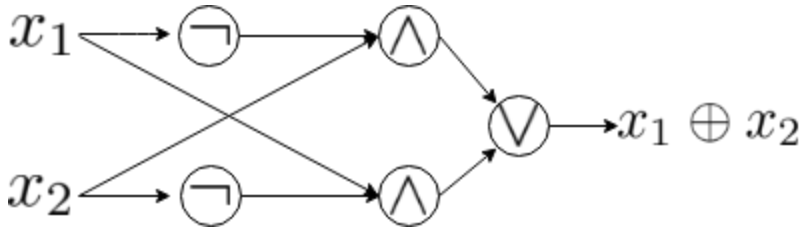
Logical OR



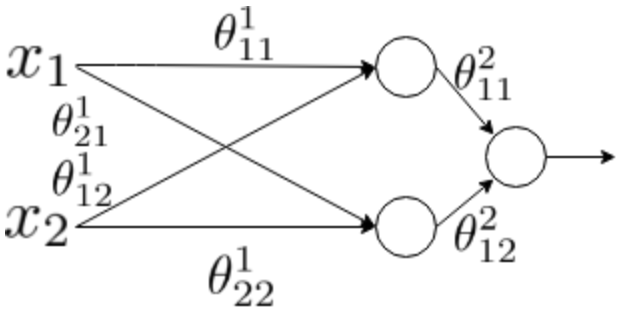
Not linearly separable

Linearly separable

- Need to compose logical expressions to gain an expression that can compute XOR and separate the classes, which will cause the zero class to collapse to the left of the decision boundary.



XOR logical circuit



Computation graph of XOR circuit. Superscript of theta indicates the network layer.

$$\sigma(\theta_{11}^1 x_{11}^1 + \theta_{12}^1 x_{12}^1) = x_{11}^2$$
$$\sigma(\theta_{21}^1 x_{21}^1 + \theta_{22}^1 x_{22}^1) = x_{12}^2$$
$$\sigma(\theta_{11}^2 x_{11}^2 + \theta_{12}^2 x_{12}^2)$$

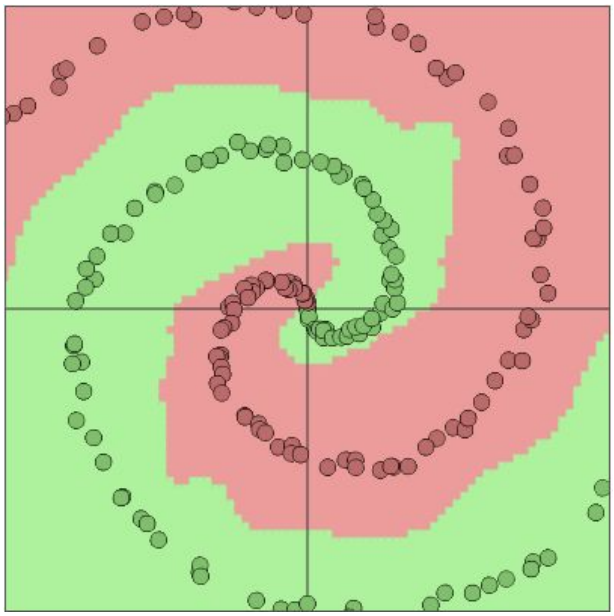
Nonlinear function representation of computation graph of XOR circuit

$$\begin{bmatrix} \theta_{11}^1 & \theta_{12}^1 \\ \theta_{21}^1 & \theta_{22}^1 \end{bmatrix}$$
$$\begin{bmatrix} \theta_{11}^2 & \theta_{12}^2 \end{bmatrix}$$

Matrix notation for weights of nonlinear function for XOR circuit

2.2 Modeling High Dimensional Nonlinear Data

- Simple linear functions fail to compute XOR because XOR is a **higher-order function** - a function that is parameterized by one or more functions and returns a function. Here, the function parameters for XOR are NOT and AND. The returned function is XOR.
- Learning algorithms require more degrees of freedom to handle nonlinear data
- Motivating example: classifying spiral data



Spiral data binary classification

3. Theorems

3.1 Universal Approximation Theorem

- George Cybenko 1989
- Kurt Hornik 1991

4. Multi-Layer Perceptron(MLP) Architecture

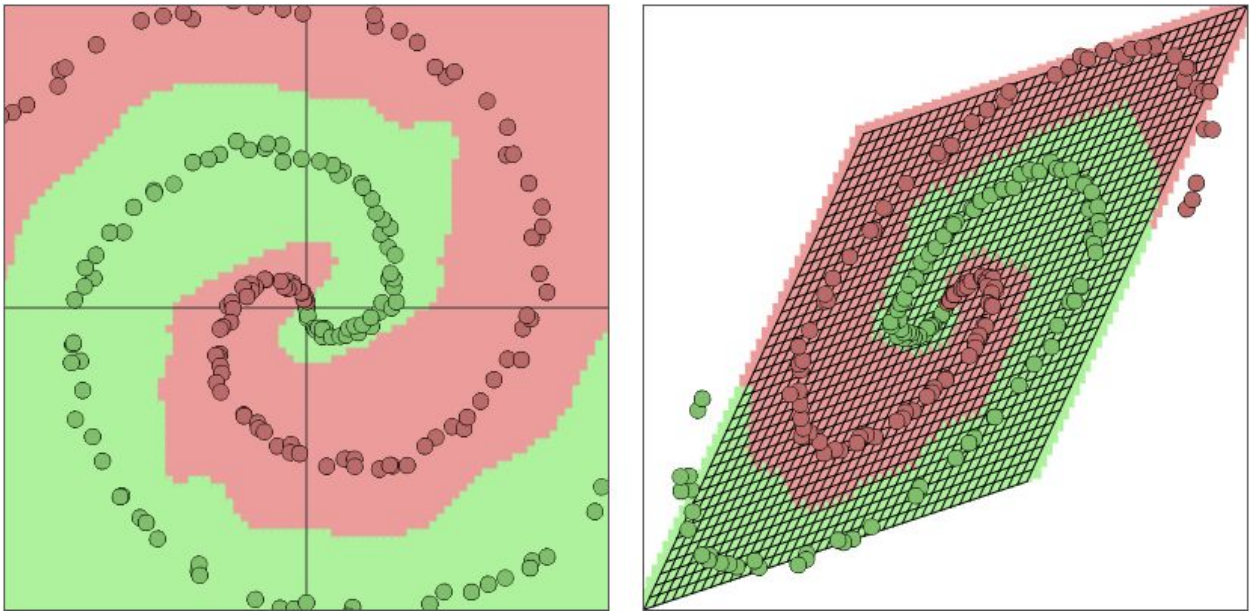
4.1 Perceptron

4.2 Input Layer

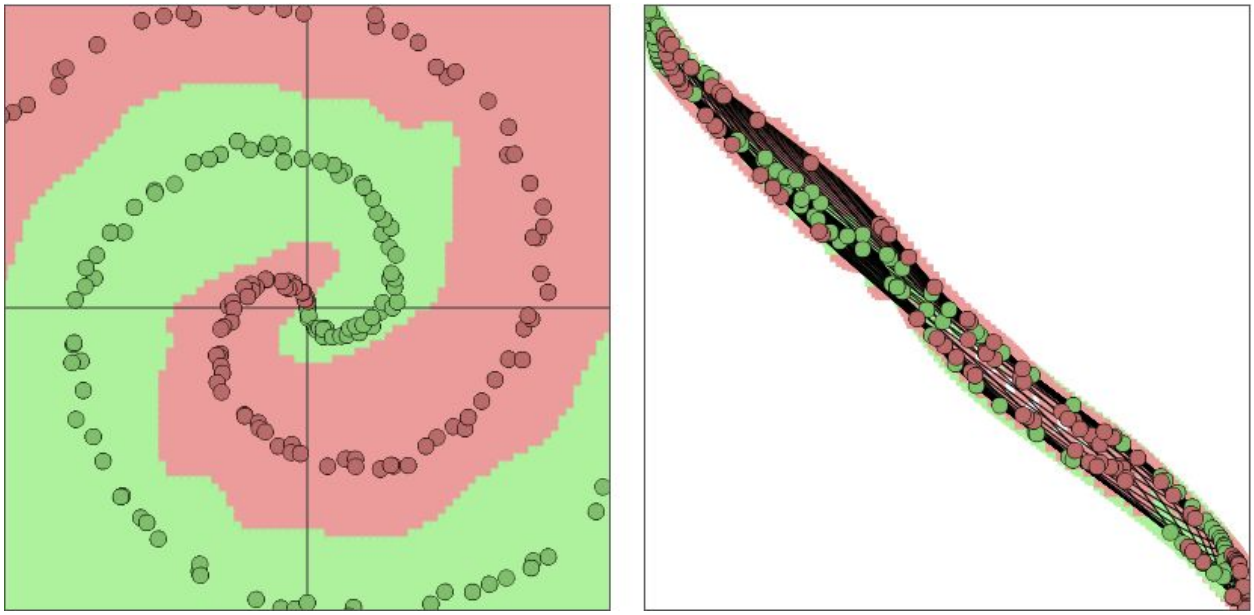
1. Dimensionality

4.3 Hidden Layer

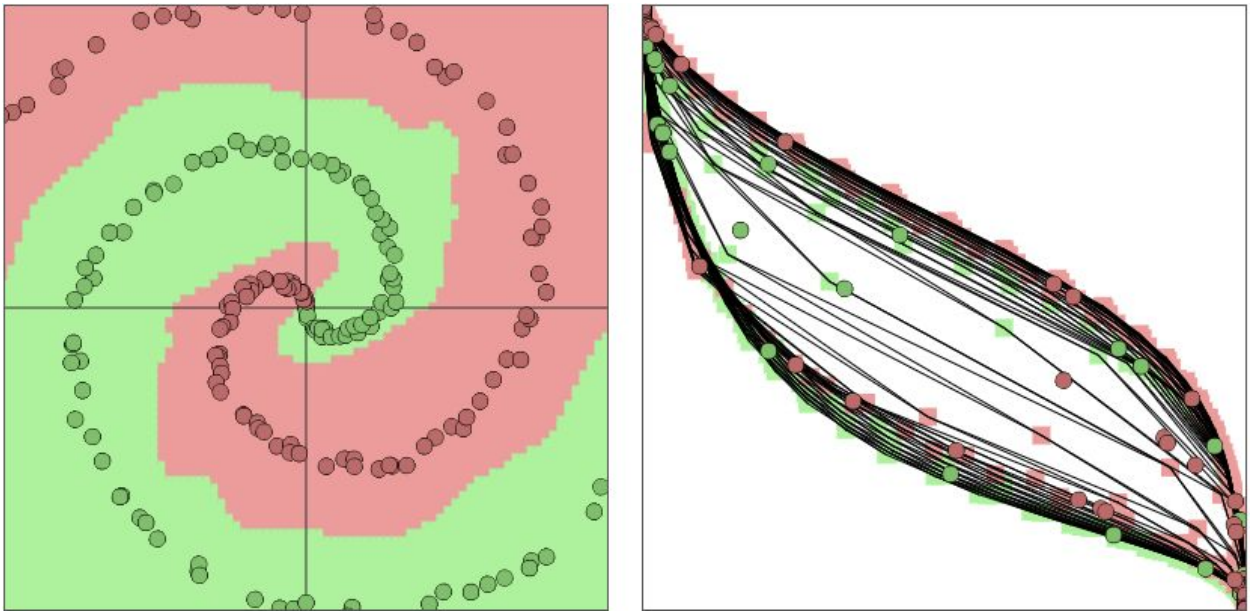
1. Manifold Hypothesis



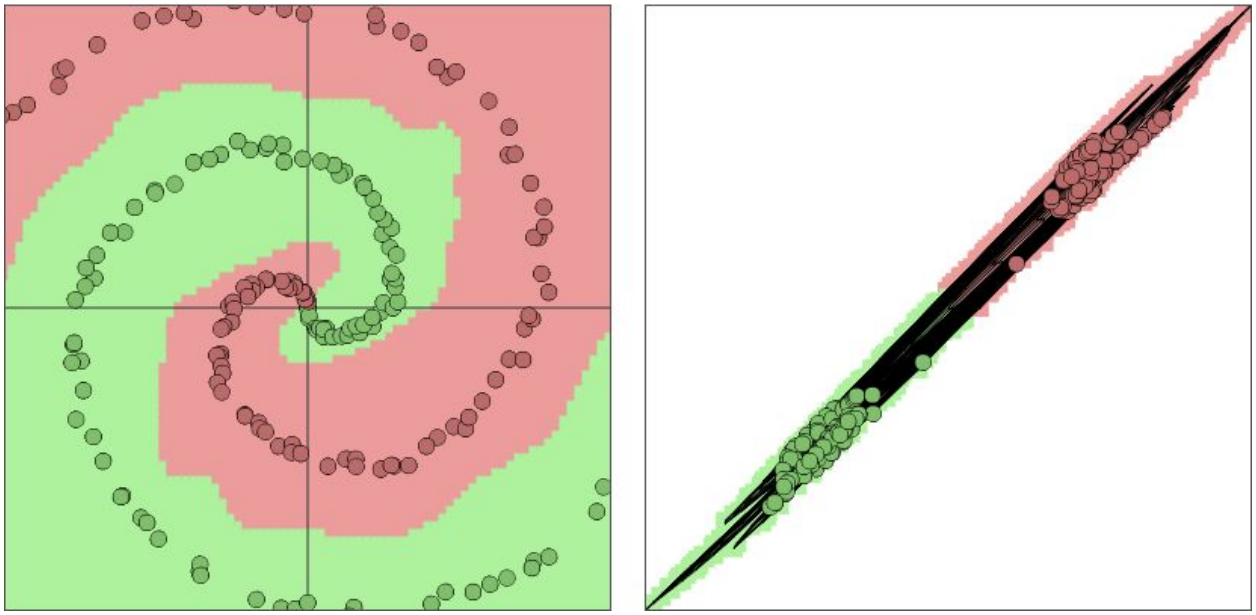
Fully-connected layer 1 with 20 neurons



TanH layer 1



Fully-connected layer 2 with 20 neurons



TanH layer 2
Notice the decision boundary in the right image.

4.4 Output Layer

1. Classification
    a. One-hot encoding
    b. Softmax Layer
    c. 6.4.1.3 PyTorch Code Here
2. Regression
    a. Single Neuron

5. Fully-Connected Network

6. Activation Functions

Since the linear models have limited ability to represent patterns and details of data, we need more non-linear representations to keep, map and represent the features of input data. Based on the **rate code interpretation**, researchers model the firing rate of the actual neurons with an activation function, which represents the frequency of the spikes along the “axon”. In general an activation function has properties like: **i) Nonlinearity ii) Differentiability iii) Monotonicity**. When using activation functions, it is important to work on the **initialization** carefully, besides, the training also partly depends on the **output range** of the activation functions.

7.1 Commonly-Used Activation Functions

1. Sigmoid
   a. Mathematical form:

$$\sigma(x) = 1/(1 + e^{-x})$$
$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

   b. Numpy implementation:

```
import numpy as np
def sigmoid(x):
        return np.exp(-np.logaddexp(0, -x))  # Be careful of numerical overflow
```

or

```
from scipy.special import expit
```

or

```
from scipy.stats import logistic
```

In [32]:

```
import math
from scipy.special import expit
from scipy.stats import logistic

def sigmoid(val):
    return 1/(1+math.exp(-val))

print(sigmoid(0))
print(expit(0))
print(logistic.cdf(0))

%timeit sigmoid(0)
%timeit expit(0)
%timeit logistic.cdf(0)
```

```
0.5
0.5
0.5
The slowest run took 13.84 times longer than the fastest. This could mean that an intermediate result is being cached
.
1000000 loops, best of 3: 503 ns per loop
The slowest run took 16.88 times longer than the fastest. This could mean that an intermediate result is being cached
.
1000000 loops, best of 3: 1.4 µs per loop
10000 loops, best of 3: 99.2 µs per loop
```

   c. PyTorch API:

```
import torch
torch.nn.functional.sigmoid(input)
```

   d. Discussion:
   The sigmoid nonlinearity takes a real-valued number and maps (or “squashes”) it into range [0,1]. **Large negative numbers become 0 and large positive numbers become 1**. The sigmoid function was popular in the past since it has a

nice interpretation as the firing rate of a neuron, but in practice, recently it has fallen out of favor for 2 major drawbacks:
i) Sigmoid saturate and kill gradients.
ii) Sigmoid outputs are not zero-centered.
The first one may result in dying gradient requires extra attention to the initialization of weights. The other drawback affects the efficiency of gradient descent. For further reading, check Andrej Karpathy's Course Notes (http://cs231n.github.io/neural-networks-1/#actfun)

2. TanH
    a. Mathematical form:

$$tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}} = 2\sigma(2x) - 1$$

$$tanh'(x) = 1 - tanh^2(x)$$

    b. PyTorch API:

```
import torch
torch.nn.functional.tanh(input)
```

    c. Discussion:
    Placeholders

3. SoftPlus
    a. Mathematical form:

$$f(x) = ln(1 + e^x)$$

$$f'(x) = \frac{1}{1 + e^{-x}}$$

    b. Numpy implementation:

```
import numpy as np
def softplus(x):
        return np.logaddexp(0, x)
```

    c. PyTorch API:

```
import torch
torch.nn.functional.softplus(input, beta=1, threshold=20)
```

d. Discussion:
Placeholders

4. Rectified Linear Unit (ReLU)
   a. Mathematical form:

$$f(x) = \max(0, x)$$

$$f'(x) = \begin{cases} 0 \text{ for } x<0 \\ 1 \text{ for } x\geq 0 \end{cases}$$

   b. Numpy implementation:

```
import numpy as np
def relu(x):
    return np.maximum(0, x)
```

   c. PyTorch API:

```
import torch
torch.nn.functional.relu(input, inplace=False)
```

   d. Discussion:
   Placeholders

5. Exponential Linear Unit (ELU)
   a. Mathematical form:

$$f(\alpha, x) = \begin{cases} \alpha(e^x - 1) \text{ for } x<0 \\ x \text{ for } x\geq 0 \end{cases}$$

$$f'(\alpha, x) = \begin{cases} f(\alpha, x)+\alpha \text{ for } x<0 \\ 1 \text{ for } x\geq 0 \end{cases}$$

   b. PyTorch API:

```
import torch
torch.nn.functional.elu(input, alpha=1.0, inplace=False)
```

   c. Discussion:
   Placeholders

7.2 Other Activation Functions

1. Leaky ReLU
    a. Mathematical form:

    b. PyTorch API:

```
import torch
torch.nn.functional.leaky_relu(input, negative_slope=0.01, inplace=False)
```

    c. Discussion:
    Placeholders

2. Parametric ReLU
    a. Mathematical form:

    b. PyTorch API:

```
import torch
torch.nn.functional.prelu(input, weight)
```

    c. Discussion:
    Placeholders

3. Randomized ReLU
    a. Mathematical form:

    b. PyTorch API:

```
import torch
torch.nn.functional.rrelu(input, lower=0.125, upper=0.3333333333333333,
training=False, inplace=False)
```

    c. Discussion:
    Placeholders

4. SeLU
    a. Mathematical form:

    b. PyTorch API:

```
import torch
torch.nn.functional.selu(input, inplace=False)
```

c. Discussion:
Placeholders

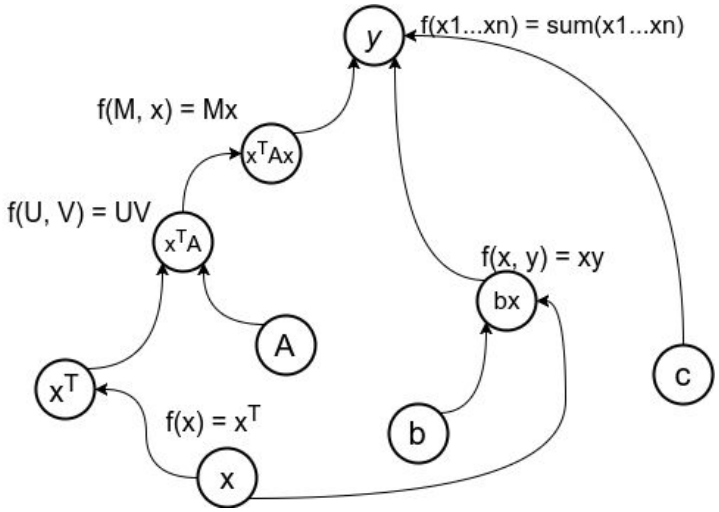
8. Computation Graph

8.1 What is a Computation Graph

In the context of deep learning, a computation graph (or dataflow graphs) is a **representation of a flow of calculations**, a functional description of deep learning models and required computations.

- A general computation graph can do both **forward** and **backward** computations. One can not only think about mathematical expressions behind neural networks easily with computation graphs, but also code a complete neural network by just following the rules of the graphs.

- A node in a computation graph represents a tensor/vector/scalar value, and it knows how to compute its value as well as the value of its derivative, and edge represents a function, a node with an edge is a function of that edge's tail node. For example, a graph of function: $y = Ax + b \cdot x + c$ could be defined as:



Nowadays, most of modern deep learning frameworks are taking advantages of computation graphs, such as Tensorflow, PyTorch, MXNet and so on. It may look strange to have two types of syntax when coding a neural network using Python, pure Python and an API library to define computation graphs at the first place, but most of popular frameworks follow this way.

8.2 Static Computation Graph

Generally, the mechanism of static computation graph, static declaration, is to define the whole graph first (with placeholders to define the size of tensors, once the computation graph is defined, it cannot be changed during the computation(except some special situations).

So the **phase 1 is define an architecture** and **phase 2 is to run a bunch of data through it** to train the model. This kind of strategy can improve the efficiency of computing (the optimization of graphs is offline), but when the network requires the changing of computation graph (such as time step in RNN), static computation graph would not be a good choice.

Also, the language to define the graph will be as complicated as a new programming language (that is why there is a Keras built on Tensorflow to simplify the codes) Currently Tensorflow is the most popular framework using static computation graph, which also provides Tensorflow Fold with dynamic batching technology.

8.3 Dynamic Computation Graph

Dynamic computation graphs use dynamic declaration instead(**graph is defined implicitly**), which means a separate execution and definition of computation graph, and a **less invasive library**. Currently, the most popular representative would be PyTorch.

9. Problems and Challenges of Deep Neural Networks

9.1 Gradient Problems

1. Vanishing Gradient
    a. Residual Connections
2. Exploding gradient

9.2 Data Efficiency

9.3 Training time

9.4 Overfitting

References

1. McCulloch, Warren S., and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity." The bulletin of mathematical biophysics 5.4 (1943): 115-133.

2. Minsky, Marvin, and Seymour Papert. Perceptrons. MIT press, 1988.
3. http://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html
4. http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/
5. http://cs231n.github.io/neural-networks-1/
6. http://www.cs.cornell.edu/courses/cs5740/2017sp/lectures/04-nn-compgraph.pdf
7. Looks, M., Herreshoff, M., Hutchins, D. and Norvig, P., 2017. Deep learning with dynamic computation graphs. arXiv preprint arXiv:1702.02181.