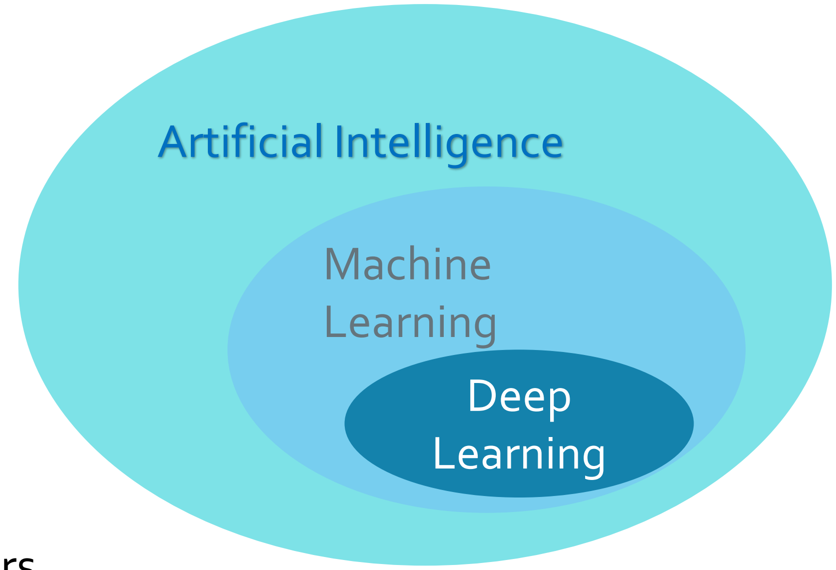
INTRODUCTION TO DEEP LEARNING

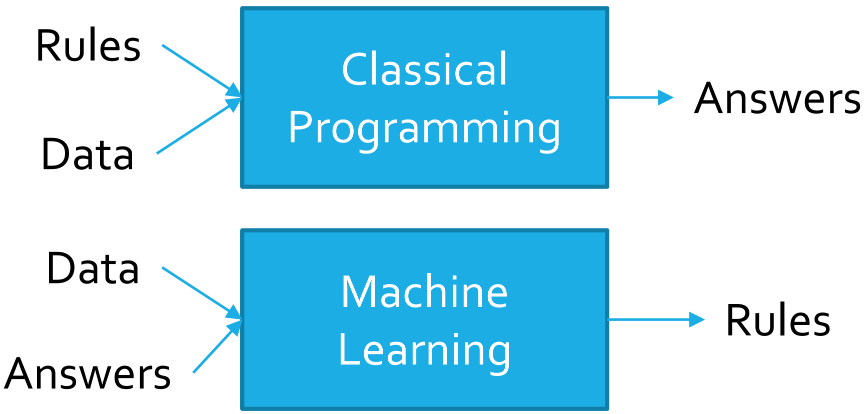
Vishal Gaurav,

PhD Student, Shibata Lab

Deep Learning

- What is AI?
- Symbolic AI
- Machine learning



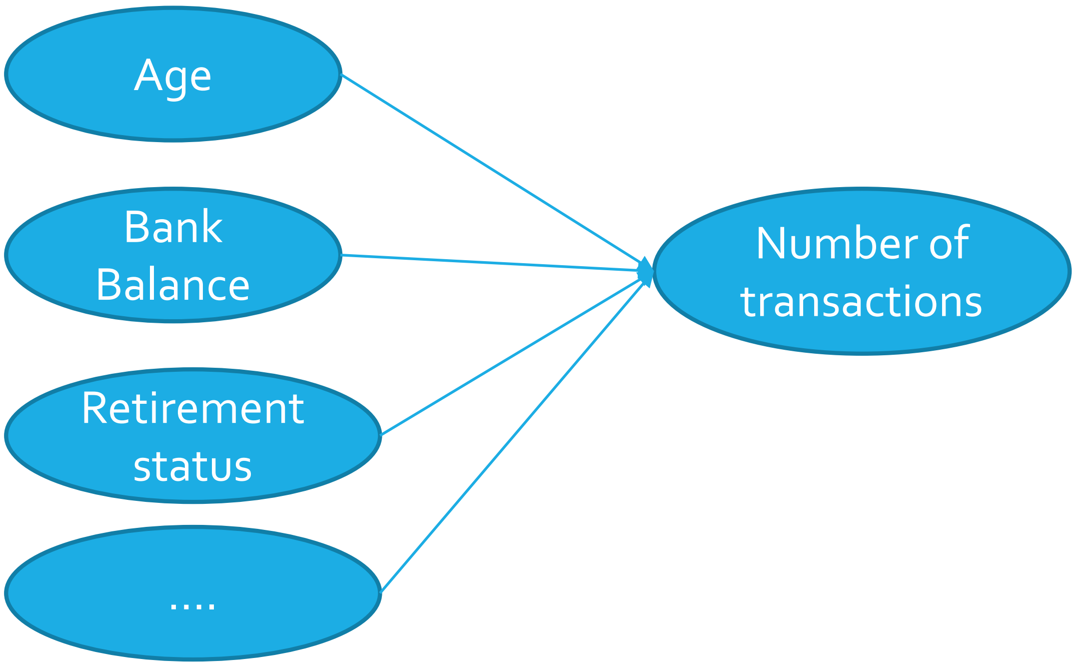


Learning representation from data

- For ML
  - Input data points
  - Examples of the expected output
  - A way to measure weather the algorithm is doing a good job
- DL is a mathematical framework for learning representation from data

Introduction

- Imagine you work for a bank
- Need to predict how many transaction each customer will make next year



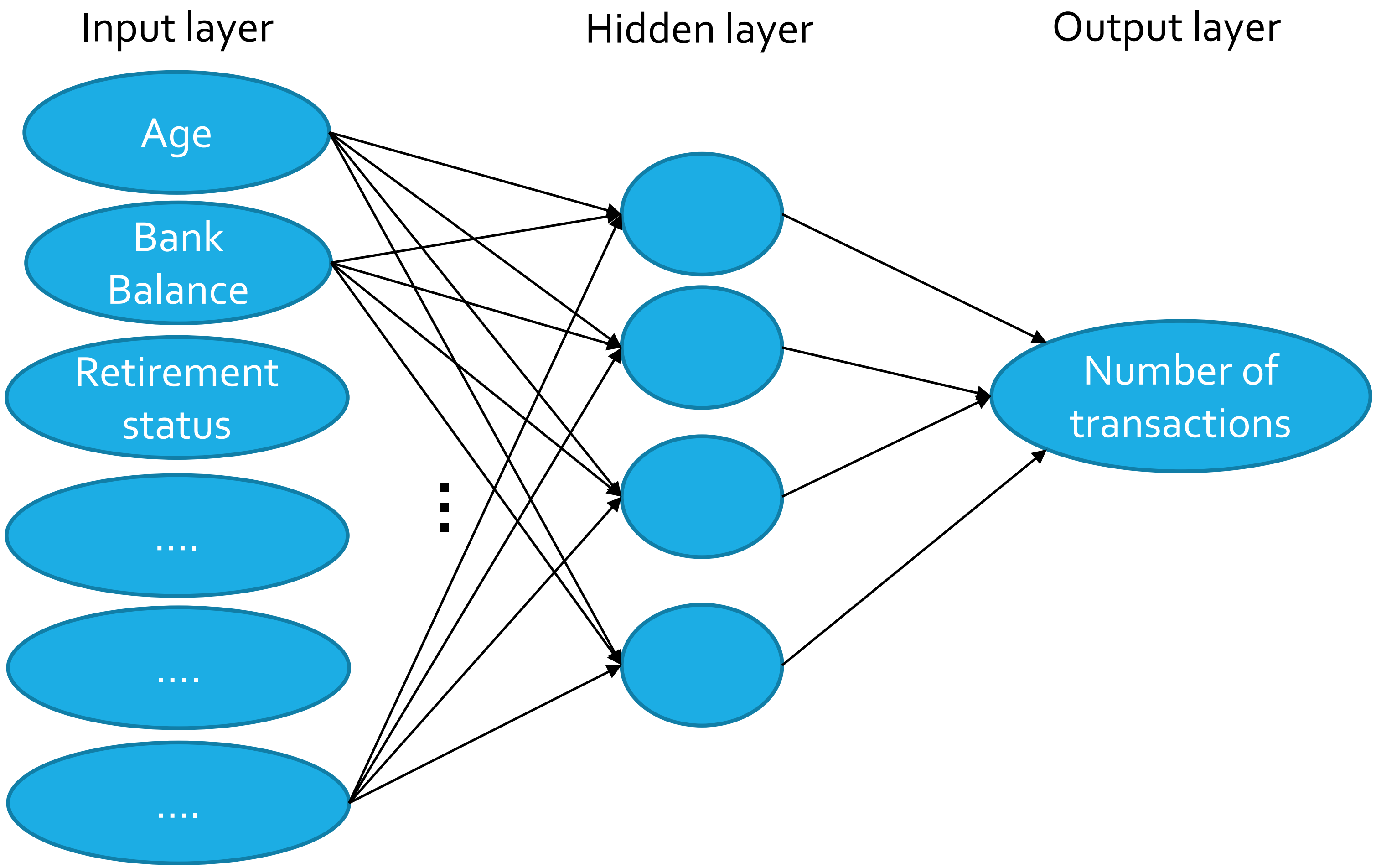
Interaction

- Neural Networks account for interactions really well
- Deep learning uses especially powerful neural networks
- Application
  - Text
  - Images
  - Videos
  - Audio
  - Source code

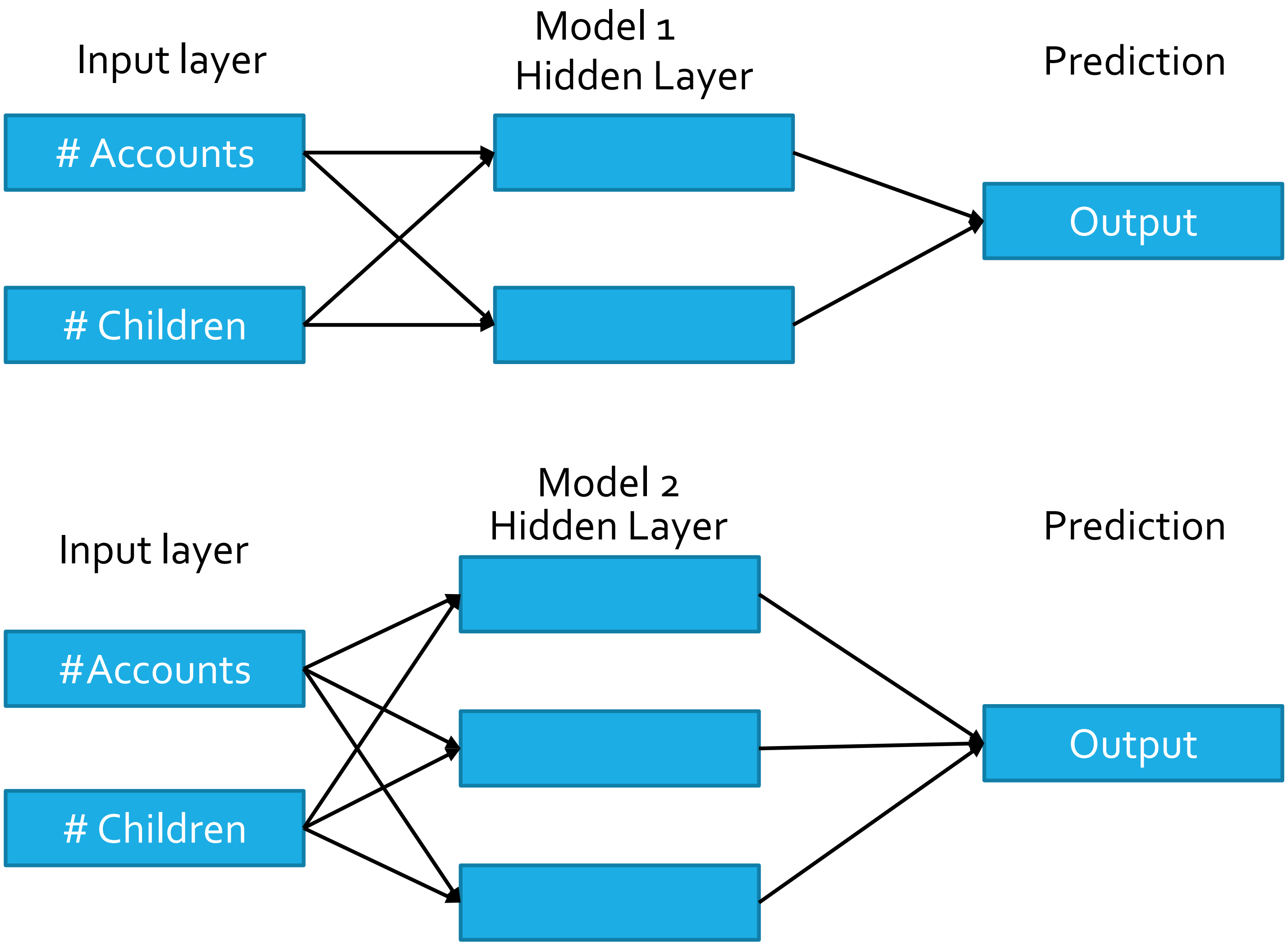
Course structure

- First we focus on conceptual knowledge
  - Debug and tune deep learning models on conventional prediction problems
  - Lay the foundation for progressing towards modern applications

Deep learning models capture interactions



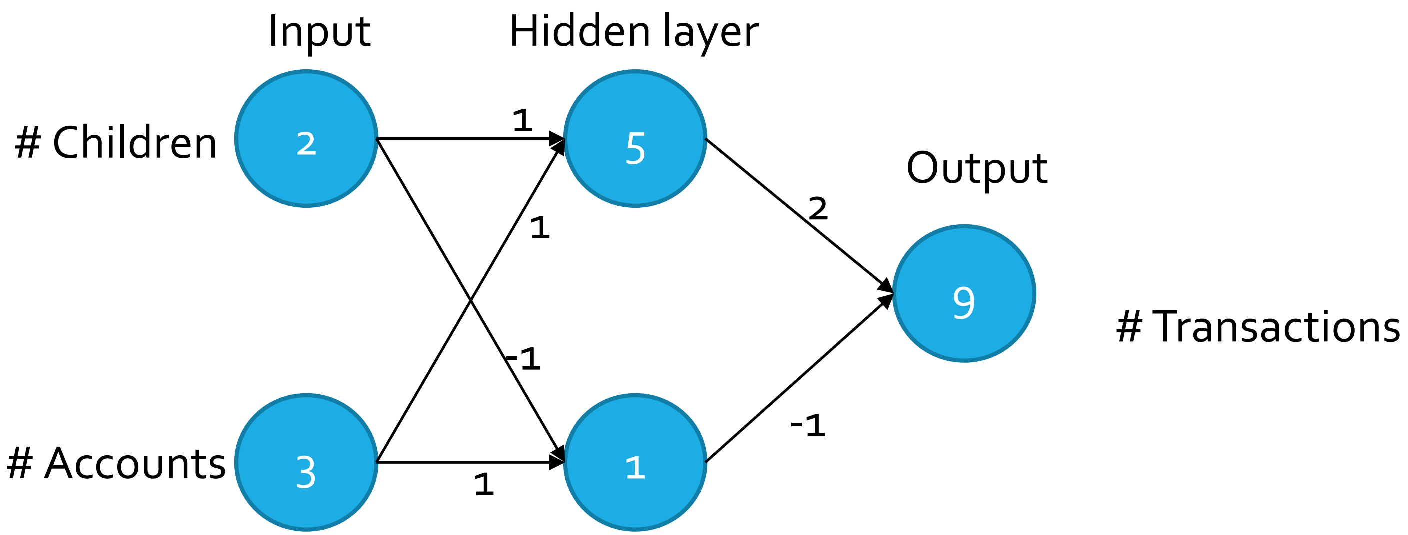
Quiz?



Input layer
Model 1
Hidden Layer
Prediction
Accounts
Children
Output
Input layer
Model 2
Hidden Layer
Prediction
#Accounts
Children
Output

Forward Propagation

- Bank transaction example
- Only using #children and # Accounts

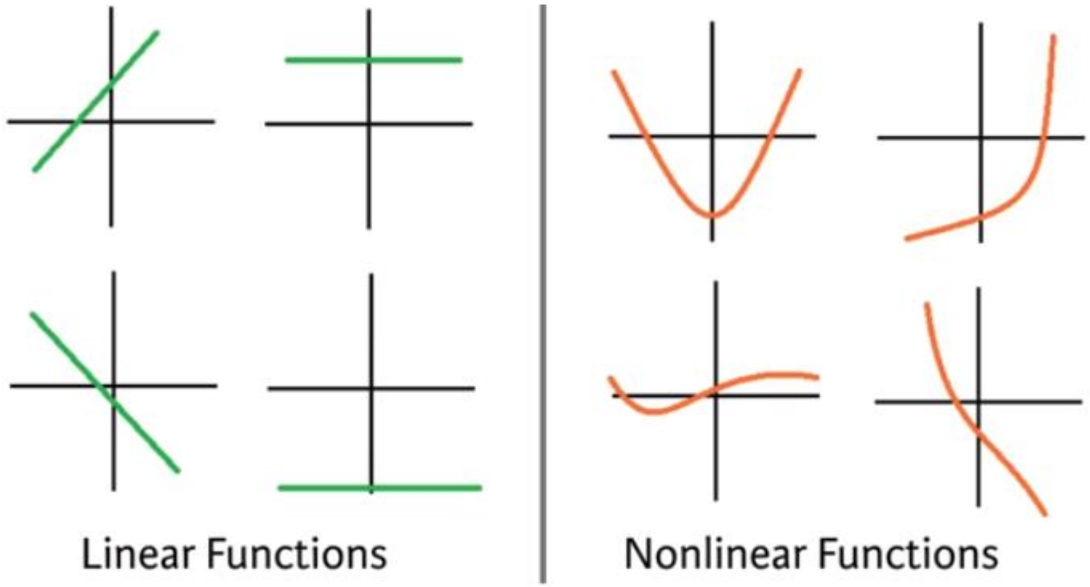


Forward Propagation

- Multiply-add process
- Dot product
- Forward propagation for one data point at a time
- Output is the prediction for that data point

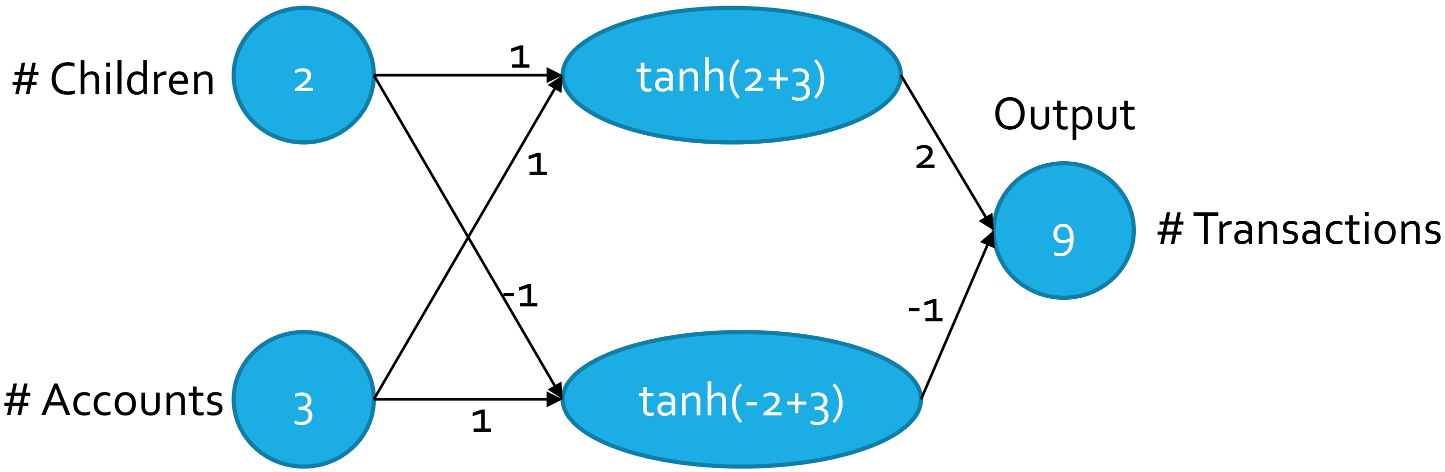
Activation Functions

- Linear vs Non-linear



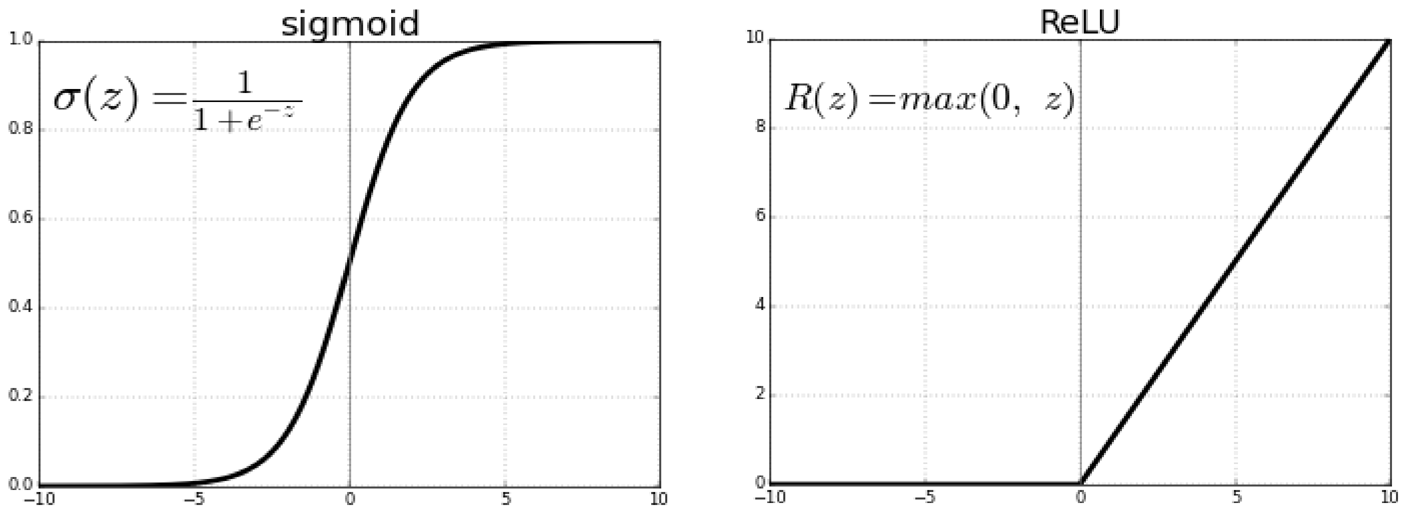
Activation function

- Applied to node inputs to produce node output



- Eg. Sigmoid, tanh, relu, leakyRelu etc..

ReLU (Rectified Linear Units)

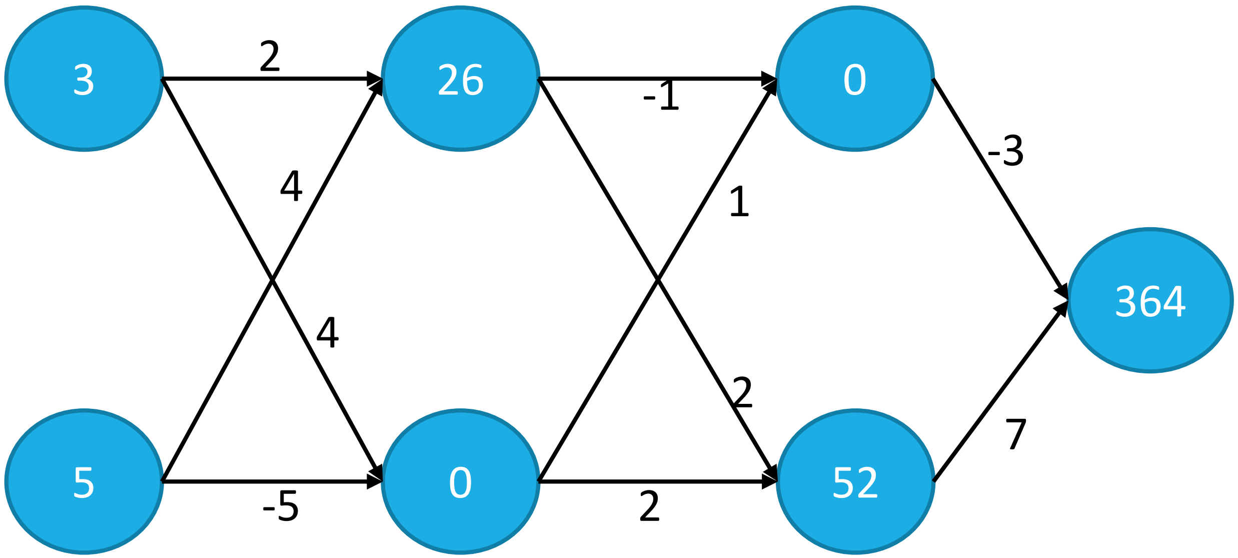


- Defined as the positive part of its argument:

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

- Where *x* is input to neuron
- Introduced by Hahnloser et. Al. in 2000 paper in NATURE.
- The function and its derivative both are monotonic

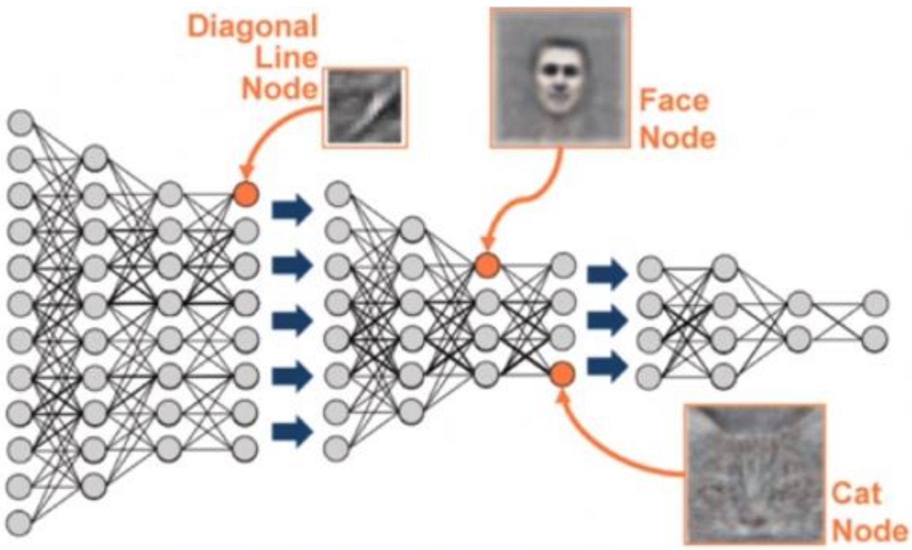
Deeper Networks



Calculated with RELU activation function

Representation learning

- Deep networks internally build representation of patterns in the data
- Partially replace the need for feature engineering
- Subsequent layers build increasingly sophisticated representation of raw data



Deep Learning

- Modeler doesn't need to specify the interactions
- When you train the model, the neural network gets weights that find the relevant patterns to make better predictions

Back Propagation

- Generative equation

$$y = w^T x + b$$

- Where x is input data
- Y is label/target/ output vector
- W and b are weights and bias

Gradient Decent

- Loss function:

$$L(y_i, \hat{y}_i\ ) = -[y_i \log \hat{y}_i + (1 - y_i) \log \hat{y}_i]$$

- We prefer to use convex loss function
- Cost function: its just average of loss

$$J(W, b) = -\frac{1}{m} \sum_{1}^{m} [y_i \log \hat{y}_i + (1 - y_i) \log \hat{y}_i]$$

Gradient Decent

- Parameter Update:

$$W = W - \alpha \frac{\delta J}{\delta W}$$

$$b = b - \alpha \frac{\delta J}{\delta b}$$

- Where α is learning rate.

Assignments

- Implement CNN classification for MNIST dataset. You can either use Keras or tensorflow or Pytorch
- Visualize the activation output of each layer.

Training/Dev/Test

Training Set	Hold/Dev Set	Test

Prev- 70/30 Or 60/20/20

Big Data: 1,000,000
98/1/1 %
Or
99.5/0.4/0.1%

Q: What to do if train/test distribution are different?

Note:
Make sure dev and test come from same distribution

Bias/Variance Tradeoff

- Human Performance $\approx 0\%$
- Example 1
  - Train error: 1% good
  - Dev error: 11% poor
  - High Variance

Overfitting

- Example 2
  - Train error: 15%
  - Dev Error: 16%
  - High Bias

Under fitting

Regularization

- Used when validation/dev error is more i.e. in case of overfitting
- L_2 regularization:

$$J(W,B) = \frac{1}{m}\sum L(\hat{y}^i, y^i) + \frac{\lambda}{2m}||W||_2^2$$

$$||W||_2^2 = \sum_{j=1}^{n_x} W_j^2 = W^T W$$

- L_1 regularization:

$$\frac{\lambda}{2m}\sum_{i=1}^{n_x} |W| = \frac{\lambda}{2m}||W||_1$$

Dropout Regularization

- Example: with layer 3
- KeepProb =0.8
- d3 = np.random.rand(a3.shape[0],a3.shape[1])<KeepProb
- d3 will be a Boolean array but in python the multiply works
- a3 = np.multiply(a3,d3) #a3*=d3
- Element wise multiply
- At test time : no dropout

Cons: cost function is no longer well defined
It is no longer monotonically decreasing

Other regularization techniques

- Data Augmentation
- Early stopping
- Normalization of Input
- Weighted initialization

Optimizers

- Moment
- RMSprop
- ADAM

Weighted/Exponential/moving average

- It is also called weighted average or exponentially weighted average or moving average

$$V_t = \beta V_{t-1} + (1-\beta)\theta_t$$

- For implementation:

$V_\theta = 0$

Keep{

gotNext θ_t

$$V_\theta = \beta V_\theta + (1-\beta)\theta_t$$

}

Note: It is memory efficient. No need to keep track of every input

Bias correction

- $V_t = \beta V_{t-1} + (1-\beta)\theta_t$
- $V_0 = 0$
- $V_1 = 0 + 0.02\theta_1$
- $V_2 = 0.98\ V_1 + 0.02\theta_2$

  $= 0.0196\theta_1 + 0.02\theta_2$
- Bias correction

$$\frac{V_t}{1-\beta^t}$$

Note: Use only when t is small

GD with momentum

- Always works better in terms of speed than GD without momentum.
- Momentum:
- On iteration t:
- Compute dW, db on current minibatch
- $V_{dW} = \beta V_{dW} + (1-\beta)dW,$
- $V_{db} = \beta V_{db} + (1-\beta)db$
- Update W,b
- $W = W - \alpha V_{dW}, \mathrm{b} = b - \alpha V_{db}$