Software Frameworks for Machine Learning

Sargur N. Srihari
srihari@cedar.buffalo.edu

Python and NumPy

- Python
  - widely used high-level, general-purpose, *interpreted*, dynamic *programming* language.
  - Its design philosophy emphasizes code readability, and its syntax allows *programmers* to express concepts in fewer lines of code than would be possible in languages such as C++ or Java
- NumPy
  - an extension to the Python programming language
  - adds support for large, multi-dimensional arrays and matrices,
  - along with a large library of high-level mathematical functions to operate on these arrays.

Python

- FizzBuzz:
  - Print i= 1 to 100, except:
    - if divisible by 3 print fizz,
    - if divisible by 5 print buzz,
    - if divisible by both 3 and 5 print fizzbuzz
- Code in C++ and Python:

```cpp
#include <iostream>
using namespace std;
int main()
{
    for(int i=1;i<=100;i++){
        if((i%3 == 0) && (i%5==0))
            cout<<"FizzBuzz\n";
        else if(i%3 == 0)
            cout<<"Fizz\n";
        else if(i%5 == 0)
            cout<<"Buzz\n";
        else
            cout<<i<<"\n";
     }
    return 0;
}
```

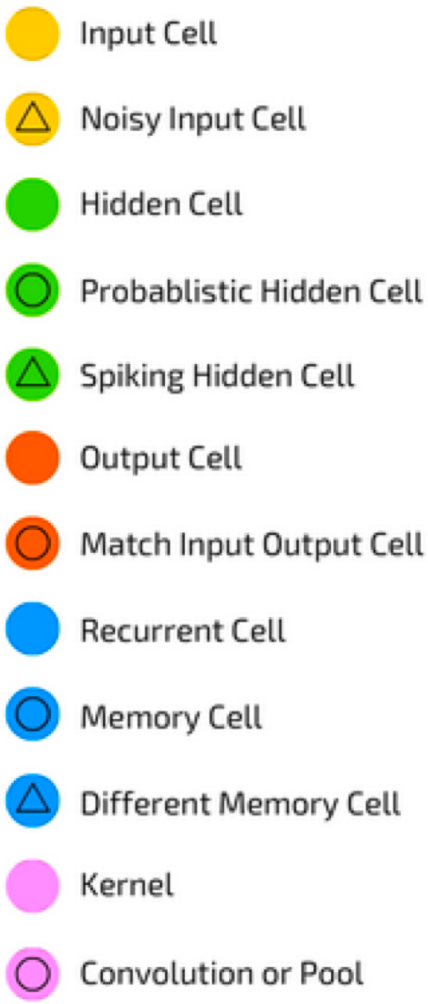
```python
# Enter your code here. Read input from STDIN. Print output to STDOUT
for i in range(101):
    if i==0:
        continue
    elif i%15==0:
        print 'FizzBuzz'
    elif i%5==0:
        print 'Buzz'
    elif i%3==0:
        print 'Fizz'
    else:
        print i
```

1

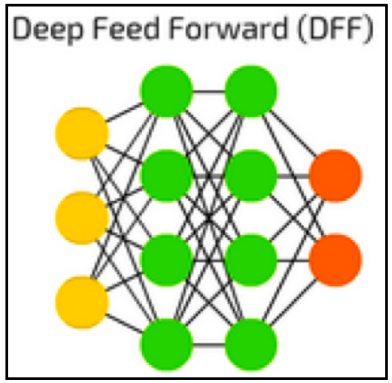
Python
Interpreted language, emphasizes code readability
fewer lines of code than C++ or Java

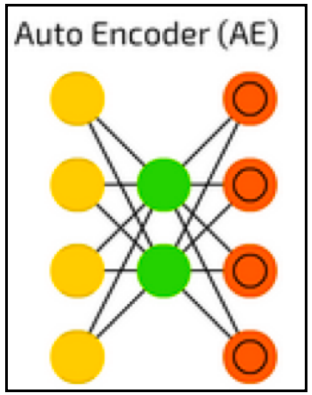
NumPy
an extension to Python: library of functions to operate on arrays.

Feedforward Networks

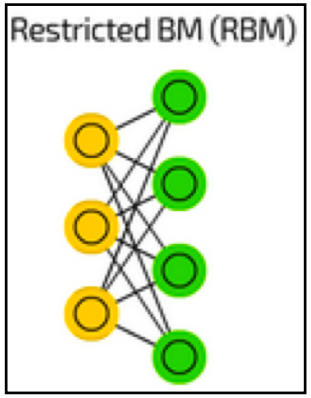


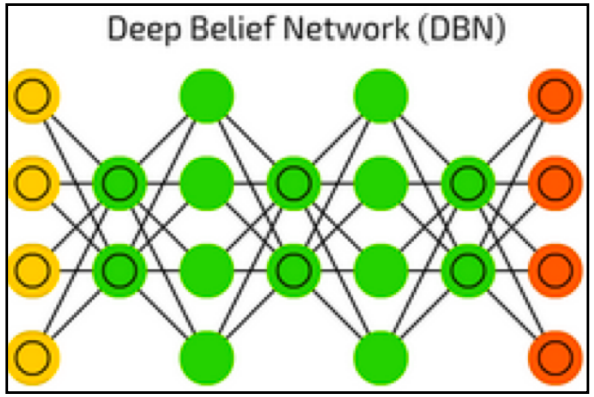
Discriminative Models

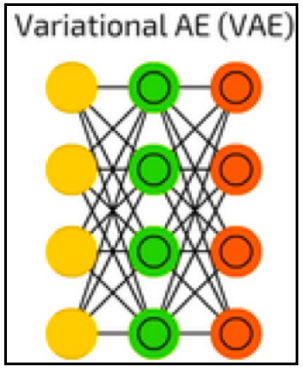


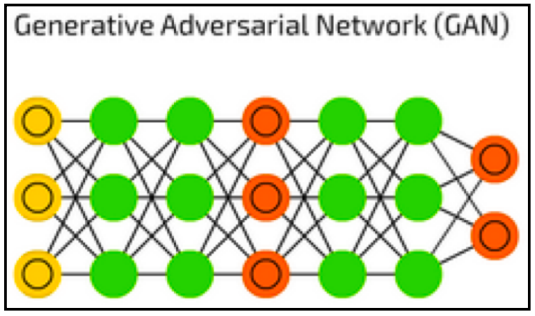


Generative Models

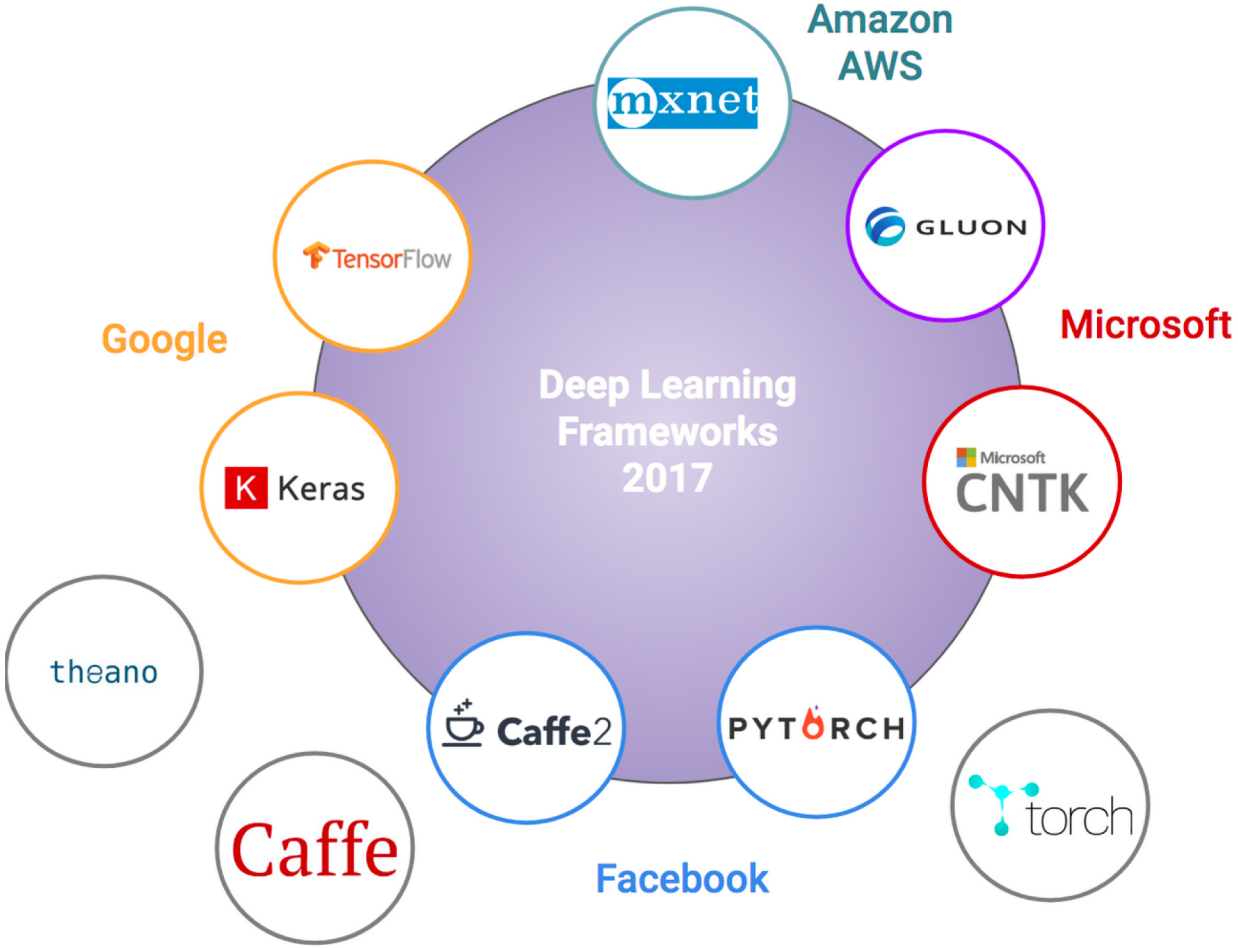








Machine Learning Frameworks



What are ML frameworks?

- Frameworks offer building blocks for designing, training and validating deep neural networks, through a high level programming interface
  - Optimized for performance
  - Provide parallelization for implementation on GPUs
  - Visualization of network modeling and interface

Framework differences

1. Theano (terminated in 2018)
   - NumPy-like numerical computation for CPU/GPUs
2. Tensorflow (Google)
   - For large-scale deployments, especially when cross-platform and embedded deployment
3. PyTorch (Facebook)
   - GPU enabled drop-in replacement for NumPy
   - for rapid prototyping
   - Dynamic computational graphs
4. Gluon (AWS, Microsoft)
   - Model and training algorithms brought closer,
   - High performance training

PyTorch

- PyTorch is essentially a GPU enabled drop-in replacement for NumPy
- Equipped with higher-level functionality for building and training deep neural networks.
  - This makes PyTorch especially easy to learn if you are familiar with NumPy, Python and the usual deep learning abstractions
    - (convolutional layers, recurrent layers, SGD, etc.).

PyTorch vs TensorFlow

- PyTorch is better for rapid prototyping in research, for hobbyists and for small scale projects.
- TensorFlow is better for large-scale deployments, especially when cross-platform and embedded deployment is a consideration
- See https://awni.github.io/pytorch-tensorflow/

Keras

- Keras is a higher-level API with a configurable back-end. IN 2018 TensorFlow, Theano and CNTK are supported not PyTorch.
- Keras is also distributed with TensorFlow as a part of tf.contrib.
- Keras API is especially easy to use. It's one of the fastest ways to get running with many of the more commonly used deep neural network architectures.
  - API is not as flexible as PyTorch or core TensorFlow.

Tensorflow Fold

- DLwith dynamic computation graphs
  - https://research.googleblog.com/2017/02/announcing-tensorflow-fold-deep.html
- In much of machine learning, data used for training and inference undergoes a preprocessing step, where multiple inputs (such as images) are scaled to the same dimensions and stacked into batches. This lets high-performance deep learning libraries like TensorFlow run the same computation graph across all the inputs in the batch in parallel. Batching exploits the SIMD capabilities of modern GPUs and multi-core CPUs to speed up execution. However, there are many problem domains where the size and structure of the input data varies, such as parse trees in natural language understanding, abstract syntax trees in source code, DOM trees for web pages and more. In these cases, the different inputs have different computation graphs that don't naturally batch together, resulting in poor processor, memory, and cache utilization.
- TensorflowFold addresses these challenges

Natural Language Toolkit

- Import nltk
- Moby=nltk.text.Text(nltk.corpus.gutenberg.words('melville-moby_dick.txt'))
- print(moby.common_contexts("ahab")

Fizzbuzz

- Print i= 1 to 100, except:
  - if divisible by 3 print fizz,
  - if divisible by 5 print buzz,
  - if divisible by both 3 and 5 print fizzbuzz
- Conventional Algorithm:

C++:

```
#include <iostream>
using namespace std;
int main()
{
    for(int i=1;i<=100;i++){
        if((i%3 == 0) && (i%5==0))
            cout<<"FizzBuzz\n";
        else if(i%3 == 0)
            cout<<"Fizz\n";
        else if(i%5 == 0)
            cout<<"Buzz\n";
        else
            cout<<i<<"\n";
     }
    return 0;
}
```

Python:

```
# Enter your code here. Read input from STDIN. Print output to STDOUT
for i in range(101):
    if i==0:
        continue
    elif i%15==0:
        print 'FizzBuzz'
    elif i%5==0:
        print 'Buzz'
    elif i%3==0:
        print 'Fizz'
    else:
        print i
```

Python
Interpreted language, emphasizes code readability
fewer lines of code than C++ or Java

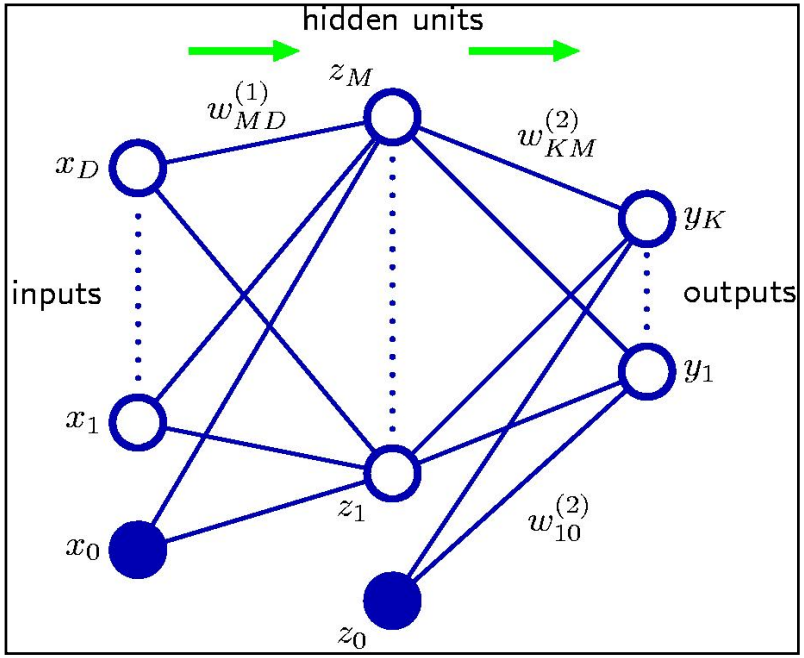
NumPy
an extension to Python: library of functions to operate on arrays.

Fizzbuzz using a simple MLP

– Given supervised output

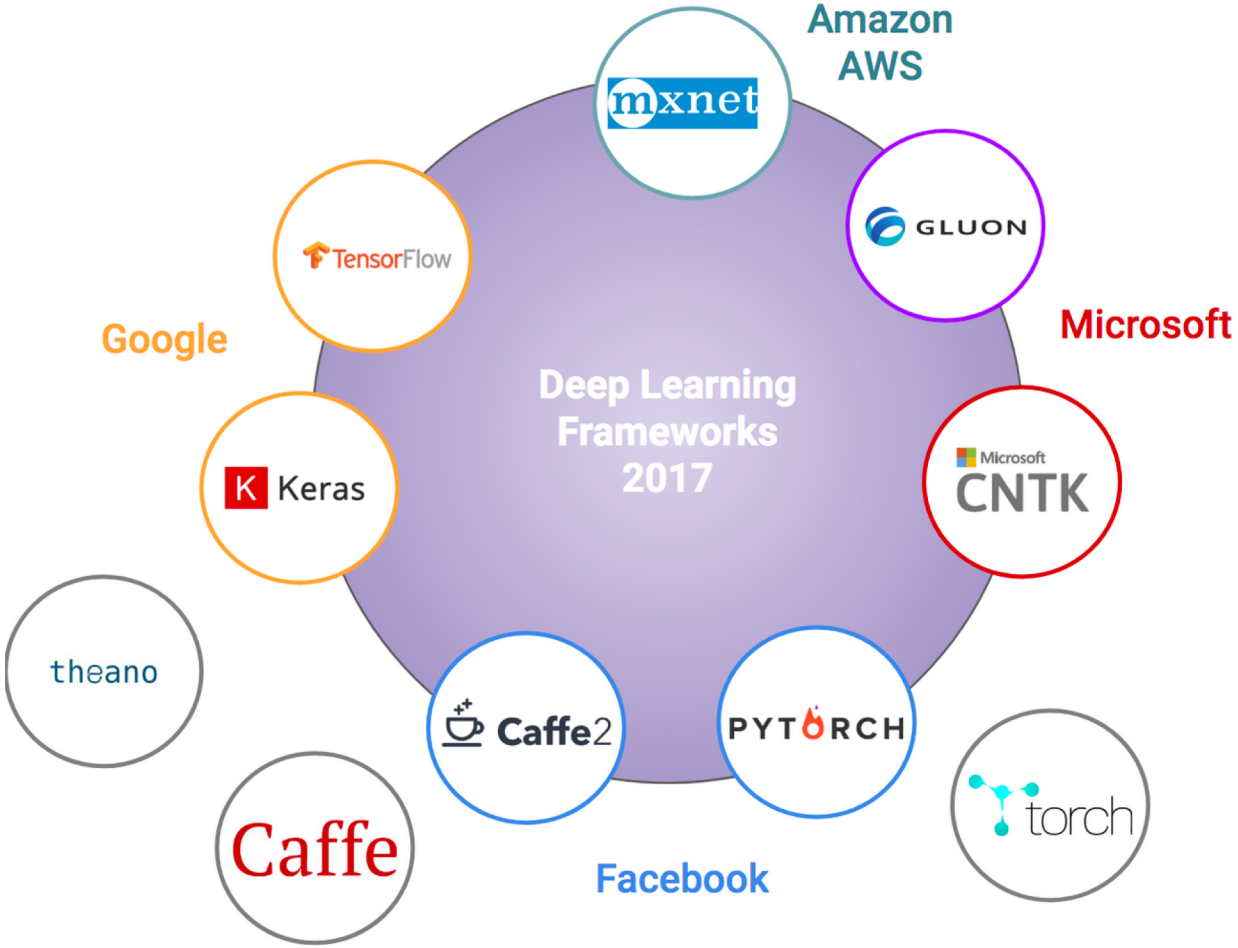
1,2,Fizz,4,Buzz,7,8,Fizz,Buzz,11,Fizz,13,14,FizzBuzz,16,17,…

– As a simple MLP with one hidden layer



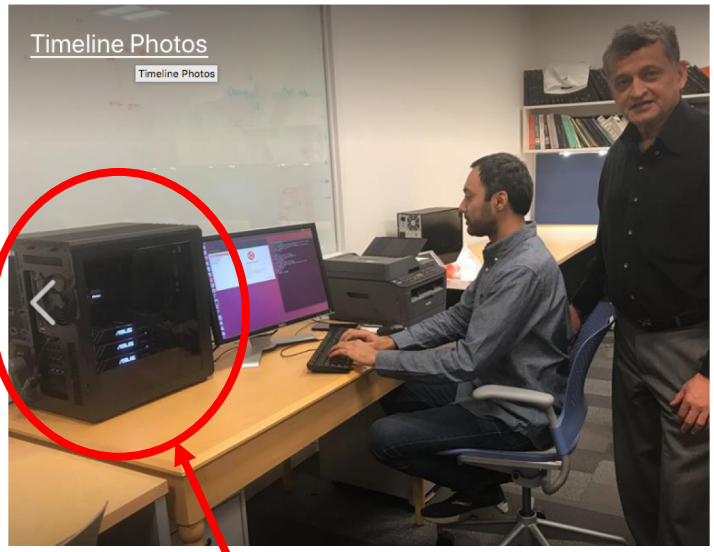
Input X: D=10 (i represented by 10 bits)
No. of hidden units: M=1,000
Output Y: K=4 (One-hot vector with 4 values)

Deep Learning Frameworks



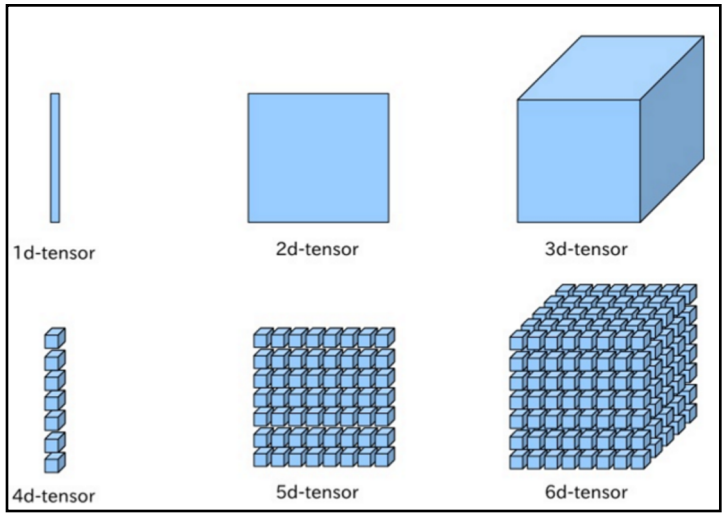
Installing Tensorflow

- TensorFlow with CPU support only
  - Typically easier to install (5 -10 mins)
- TensorFlow with GPU support
  - Significantly faster
    - Hardware: NVIDIA® GPU
    - Software: CUDA API & drivers
- Companies using Tensorflow



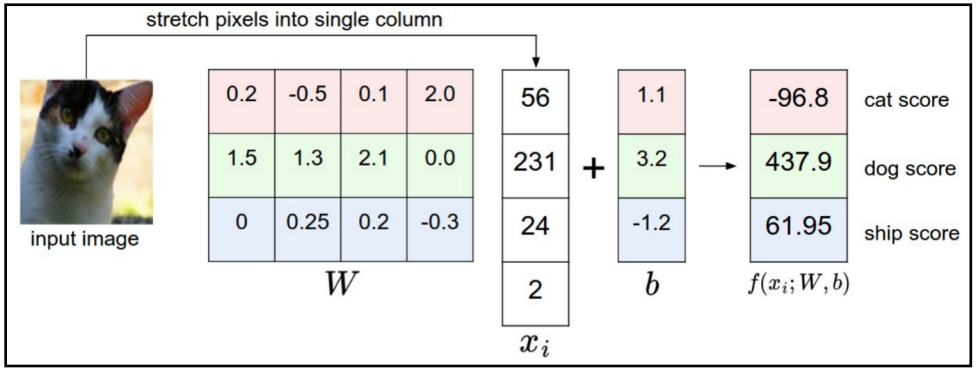
3 x NVIDIA GTX 1080

What is a tensor? What is flow?



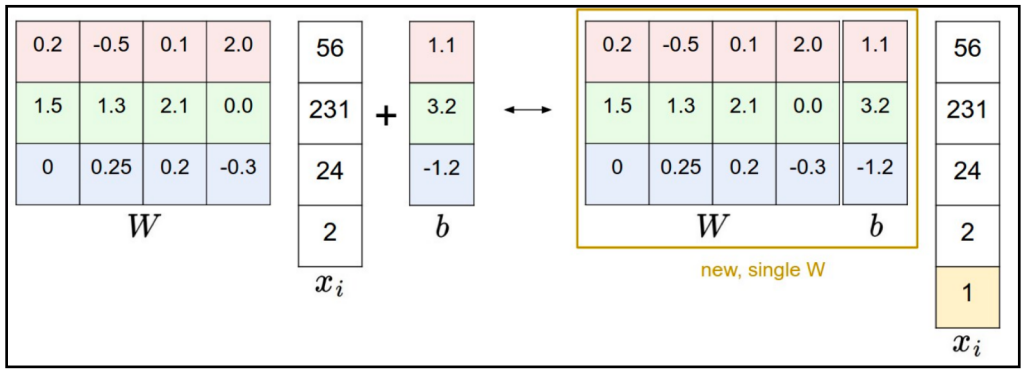
A linear classifier

$\boldsymbol{y} = W\boldsymbol{x}^T + \boldsymbol{b}$



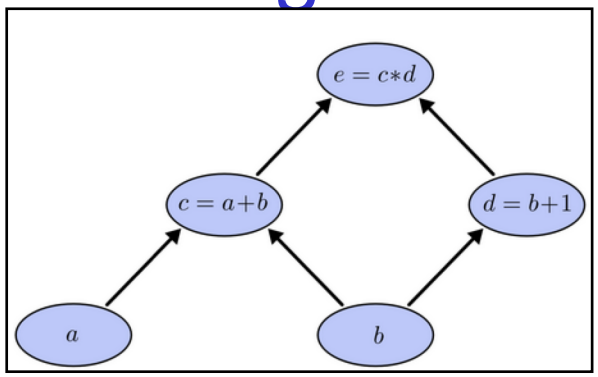
A linear classifier with bias eliminated

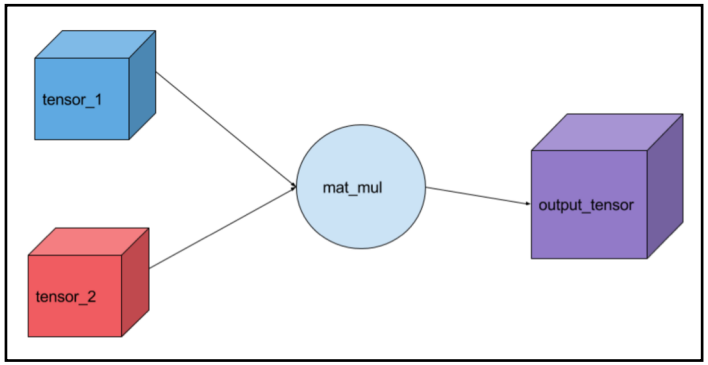
$\boldsymbol{y} = W\boldsymbol{x}^T$

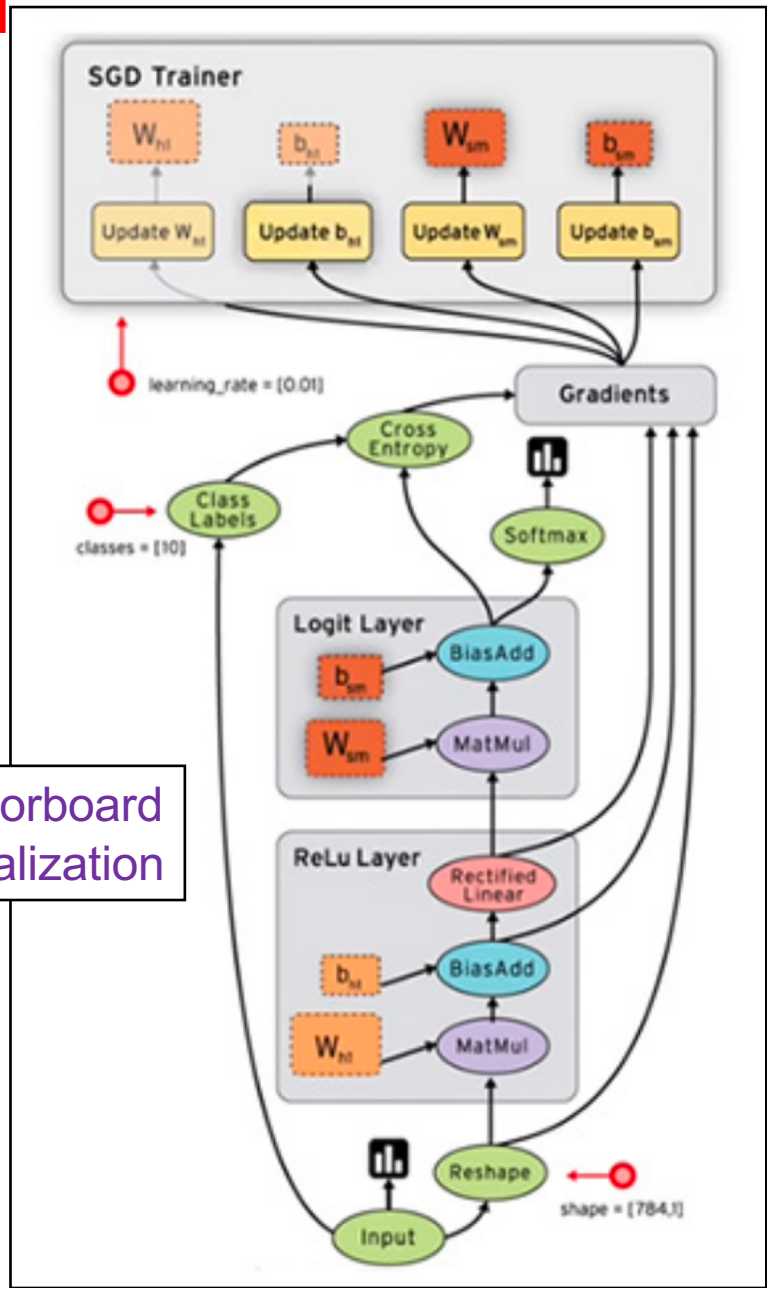


Computational Graph

- Network structure specified by computational graphs
- Makes it easy to visualize and debug







Tensorboard Visualization

Fizzbuzz in Tensorflow

- Import numpy and Tensorflow libraries

```
import numpy as np
import tensorflow as tf
```

Input to Network

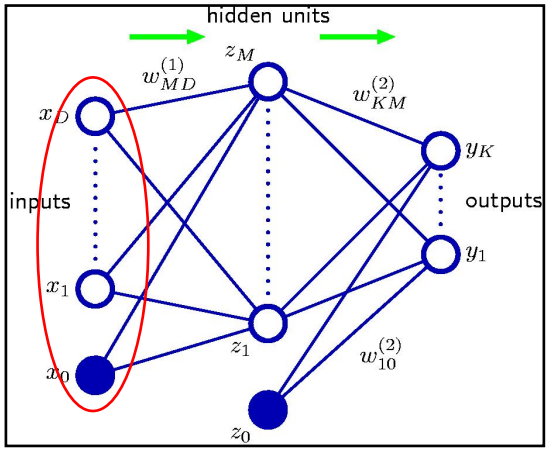
- Input i represented as a bit-vector
  - Distributed representation is key to deep learning
  - Turn each input into a vector of "activations"
    - Define a binary encoder for the input i

```
def binary_encode(i, num_digits):
return np.array([i >> d & 1 for d in range(num_digits)])
```

num_digits=*D*= no. of input bits

- Python Syntax:

1. i>>d returns i shifted to right by d places
2. &, e.g., a&b
   Bitwise logical AND operator



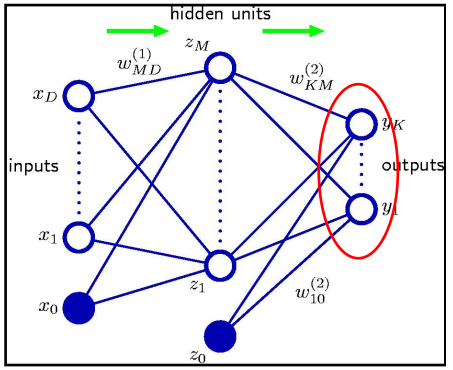
Output for training

Output is a one-hot encoding of fizzbuzz representation of input, where the four positions indicate print as-is, fizz, buzz and fizzbuzz:

```
def fizz_buzz_encode(i):
 if i % 15 == 0:  return np.array([0, 0, 0, 1])
 elif i % 5 == 0: return np.array([0, 0, 1, 0])
 elif i % 3 == 0: return np.array([0, 1, 0, 0]
 else: return np.array([1, 0, 0, 0])
```

- Python Syntax:

1. % operator, e.g., b % a
   divide left operand by right operand and return remainder
2. == condition, e.g., a==b
   Condition becomes true if operands are equal



Generating Training Samples

- No cheating: can't use 1 to 100 in training data
  - We train on all remaining numbers up to 1024:
  - Define trX using binary_encode (101 to 1024)
  - Define trY using fizz_buzz_encode

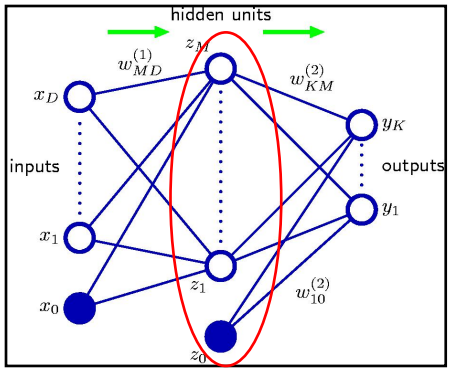
```
num_digits= 10
trX = np.array([binary_encode(i, num_digits)
for i in range(101, 2 ** num_digits)])
trY = np.array([fizz_buzz_encode(i)
for i in range(101, 2 ** num_digits)])
```

Python Syntax: Range function, e.g., range(3)=[0,1,2]

No. of Hidden Units

- Could be 10
- Probably 100 is better

```
num_hidden= 100
```

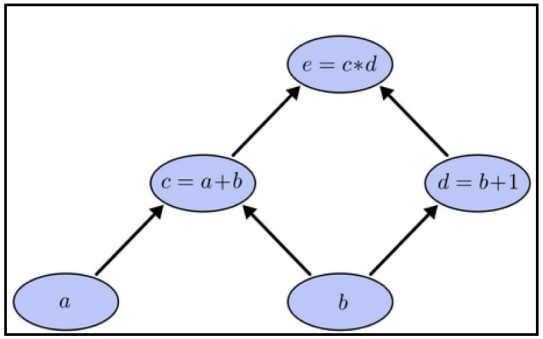


Input and Output Variables

- We'll need an input variable of width num_digits
- Output variable with width 4

```
X = tf.placeholder("float", [none, num_digits])
Y = tf.placeholder("float", [none, 4])
```

  - A **placeholder** is a variable that we will assign data to at a later date. It allows us to create our operations and build our computation graph, without needing the data
  - Computational graph
    - Operations and inputs are nodes
    - Values used in operations are directed edges



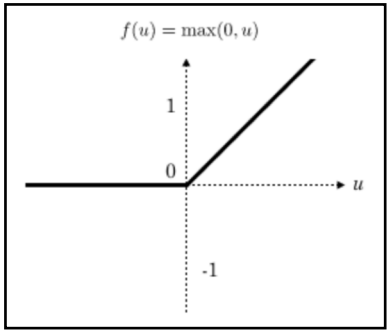
Randomly initialized weights

- One hidden layer and one output layer

```
def init_weights(shape):
        return tf.Variable(tf.random_normal(shape,
        stddev=0.01))
w_h = init_weights([num_digits, num_hidden])
w_o = init_weights([num_hidden, 4])
```

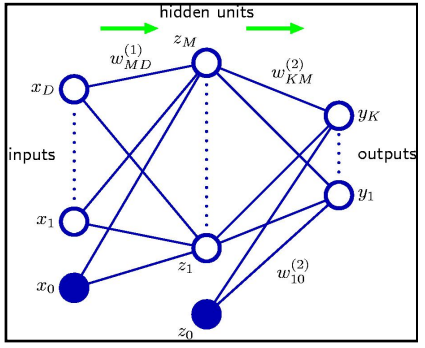
Defining the model

- Define model using one hidden layer and ReLU activation




```
def model(X, w_h, w_o):
    h = tf.nn.relu(tf.matmul(X, w_h))
    return tf.matmul(h, w_o)
```

  - ReLu activation: $g(z)=\max\{0,z\}$
    - Easy to optimize due to similarity with linear units
    - Only difference is that they output 0 across half its domain
    - Derivative is 1 everywhere that the unit is active



Minimizing the Cost function

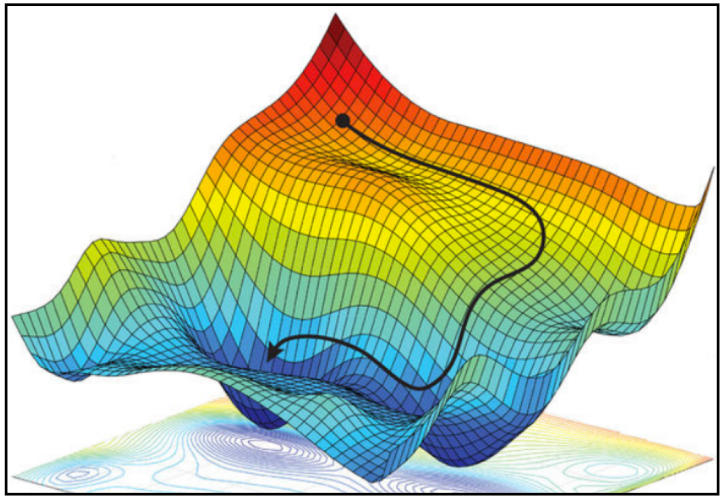
- Softmax cross entropy (with *K* classes, *N* samples)

$$E(\boldsymbol{w}) = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_{kn} \ln y_k(\boldsymbol{x}_n, \boldsymbol{w})$$

$$y_k(\boldsymbol{x}, \boldsymbol{w}) = \frac{\exp(a_k(\boldsymbol{x}, \boldsymbol{w}))}{\sum_j \exp(a_j(\boldsymbol{x}, \boldsymbol{w}))}$$

- Gradient Descent

$$\frac{\partial E}{\partial a_k} = y_k - t_k$$



```
py_x = model(X, w_h, w_o)
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(py_x, Y))
train_op = tf.train.GradientDescentOptimizer(0.05).minimize(cost)
```

Prediction

- Prediction will just be the largest output

```
predict_op = tf.argmax(py_x, 1)
```

- This will output a number from 0 to 3
  - But we want a fizzbuzz output

```
def fizz_buzz(i, prediction):
return [str(i), "fizz", "buzz", "fizzbuzz"][prediction]
```

Training

- We are ready to train the model
  - grab a tensorflow session and initialize variables

```
with tf.Session() as sess:
        tf.initialize_all_variables().run()
```

- Training epochs
  - 10,000 epochs of training
  - Shuffle them for each iteration

```
for epoch in range(10000):
    p = np.random.permutation(range(len(trX))) trX, trY = trX[p], trY[p]
```

  - Each epoch trained in batches of 128

```
BATCH_SIZE=128
```

Each Training Pass

- Each epoch trained in batches of 128

```
BATCH_SIZE=128
```

- Each training pass looks like

```
for start in range(0, len(trX), BATCH_SIZE):
end = start + BATCH_SIZE
sess.run(train_op, feed_dict={X: trX[start:end], Y:
trY[start:end]})
```

Accuracy on the Training Data

- It is helpful see how training accuracy evolves

```
print(epoch, np.mean(np.argmax(trY, axis=1) ==
    sess.run(predict_op, feed_dict={X: trX, Y:    trY})))
```

Fizzbuzz Testing

- Once model is trained, it is Fizzbuzz time
- Input is binary encoding of numbers 1 to 100

```
numbers = np.arange(1, 101)
teX = np.transpose(binary_encode(numbers, num_digits))
```

- Output is fizzbuzz function applied to model output

```
teY = sess.run(predict_op, feed_dict={X: teX})
output = np.vectorize(fizz_buzz)(numbers, teY)
print(output)
```

Performance on Test Set

- array(['1', '2', 'fizz', '4', 'buzz', 'fizz', '7', '8', 'fizz', 'buzz', '11', 'fizz', '13', '14', 'fizzbuzz', '16', '17', 'fizz', '19', 'buzz', '21', '22', '23', 'fizz', 'buzz', '26', 'fizz', '28', '29', 'fizzbuzz', '31', 'fizz', 'fizz', '34', 'buzz', 'fizz', '37', '38', 'fizz', 'buzz', '41', '42', '43', '44', 'fizzbuzz', '46', '47', 'fizz', '49', 'buzz', 'fizz', '52', 'fizz', 'fizz', 'buzz', '56', 'fizz', '58', '59', 'fizzbuzz', '61', '62', 'fizz', '64', 'buzz', 'fizz', '67', '68', '69', 'buzz', '71', 'fizz', '73', '74', 'fizzbuzz', '76', '77', 'fizz', '79', 'buzz', '81', '82', '83', '84', 'buzz', '86', '87', '88', '89', 'fizzbuzz', '91', '92', '93', '94', 'buzz', 'fizz', '97', '98', 'fizz', 'fizz']
- Running code on GitHub got some outputs wrong!
  - 0.90 fizz-accuracy, and 0.99 buzz-accuracy. So it's clearly harder to teach fizzing than buzzing.
- A deeper network may help

Performance of ML Fizzbuzz

- Running this code on GitHub got some of the outputs wrong!
  - 0.90 fizz-accuracy
  - 0.99 buzz-accuracy
- So it's harder to teach fizzing than buzzing
- A deeper network may help

Practical Methodology

1. Performance Metrics
2. Default Baseline Models
3. Determining whether to gather more data
4. Selecting hyperparameters
5. Debugging strategies

Performance Metrics

1. Metrics for Regression: squared error, RMS
2. Metric for Density Estimation: KL divergence
3. Metrics for Classification: Accuracy
4. Metrics for Unbalanced data:
   - Loss, Specificity/Sensitivity
5. Metrics for Retrieval: Precision and Recall
6. Combining Precision and Recall: F-Measure
7. Metrics for Image Segmentation: Dice Coefficient

Choice of Optimization Algorithm

- SGD with momentum with a decaying learning rate
  - Linear decay
  - Exponential decay
  - Decreasing learning rate by factor of 2-10 when validation error rate plateaus
- Adam batch normalization can have a dramatic effect on optimization performance
- Reasonable to omit batch normalization from very first baseline unless optimization is problematic

Regularization

- Unless training set includes tens of millions of examples, should include some form of regularization from start
- Early stopping should be used universally
- Dropout is easy to implement and compatible with many models/training

More data to improve performance

- After end-to-end system established, it is time to measure performance and determine how to improve it
- Instead of other learning algorithms it is often better to get more data than improve learning algorithms

Determining whether more data needed

- Determine performance on training data
- If performance on training data is poor algorithm is not using training data already available, so more data is not needed
  - Instead try increasing model size
    - More layers, more hidden units within layers
  - Try improving learning algorithm
    - E.g., tuning the learning rate hyperparameter
    - If performance unsatisfactory, quality if training data may be responsible; too noisy, not include right inputs
      - Suggesting starting over with new data

Approaches to choosing hyperparams

- Examples of hyperparameters:
  - Learning rate ϵ, regularizing coefficient $\boldsymbol{\lambda}$
- Choosing them manually
  - Requires understanding of what they do
  - Knowledge of how they achieve good generalization
- Choosing them automatically
  - Reduce the need to understand these ideas
  - But computationally expensive

Difficulty of Debugging

- When system performs poorly, difficult to tell:
  - poor performance is intrinsic to algorithm itself or
  - whether there is an implementation bug
- Cannot tell a priori the behavior of algorithm
- Entire point of ML:
  - it will discover useful behavior we were not able to specify ourselves
- If classification test error rate is 5%
  - we cannot tell whether this is expected behavior or suboptimal behavior

Multiple Adaptation Levels

- A difficulty is that ML models have multiple parts that are each adaptive
- If one part is broken, other parts can adapt and get acceptable performance

Debugging strategies

- Need to get around two difficulties:
  1. Whether performance is intrinsically poor or has a bug
  2. Whether parts are compensating for each other
- Design a test case that is so simple that the test behavior can be predicted, or
- Design a test that exercises one part of the neural net implementation in isolation

Important Debugging Tests

1. Visualize the model in action
2. Visualize the worst mistakes
3. Reasoning about software using train and test error
4. Fit a tiny dataset
5. Compare back-propagated derivatives to numerical derivatives
6. Monitor histograms of activations and gradient