Deep Learning: An Introduction

By Mohit Kumar

Visa University

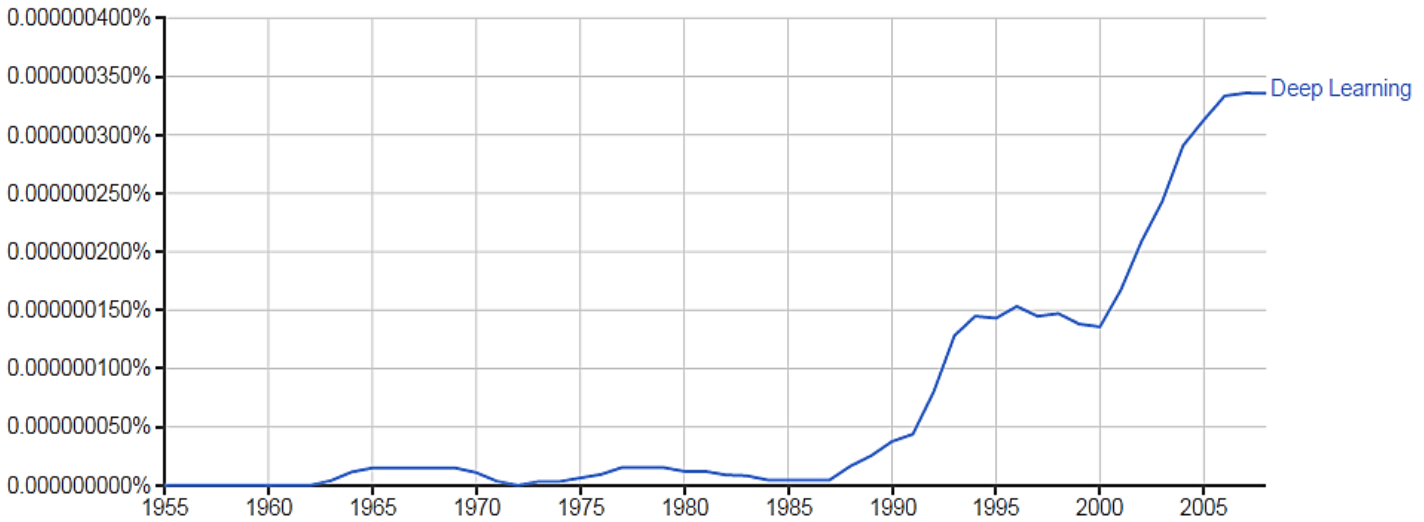
Agenda Slide

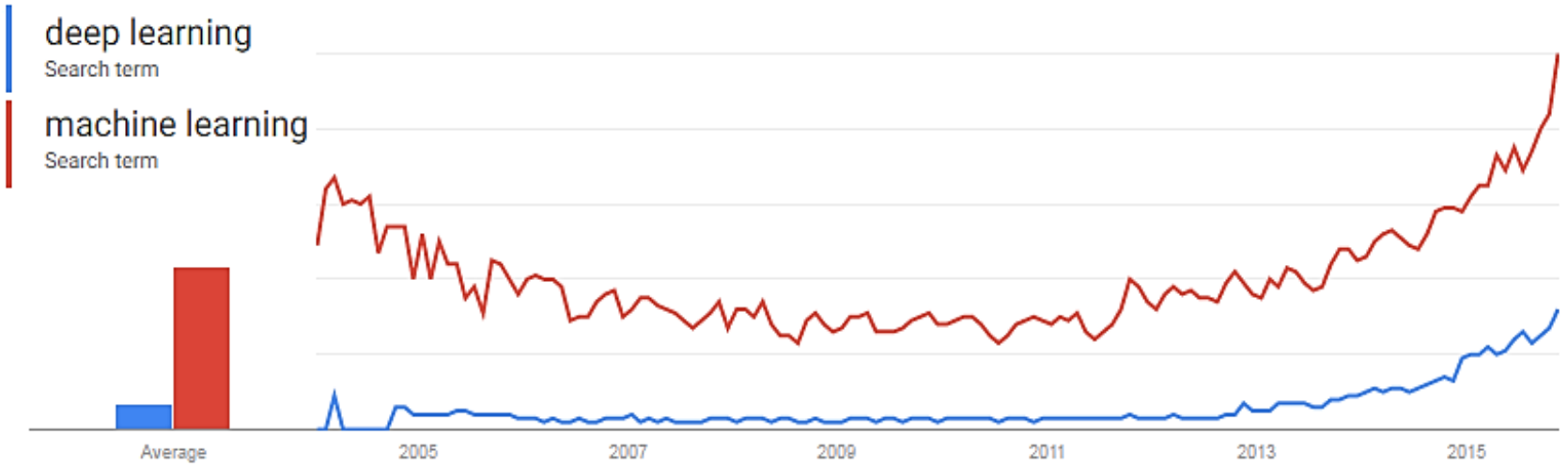
Deep Learning: An Introduction

- Foundation: Linear Regression
- Deep Learning: Going Deep:Forward pass
- Deep Learning: Going Deep:Back Prop
- Deep Learning: Architectures
- Deep Learning: Fintechs
- Deep Learning: GPU and TPU

Interest

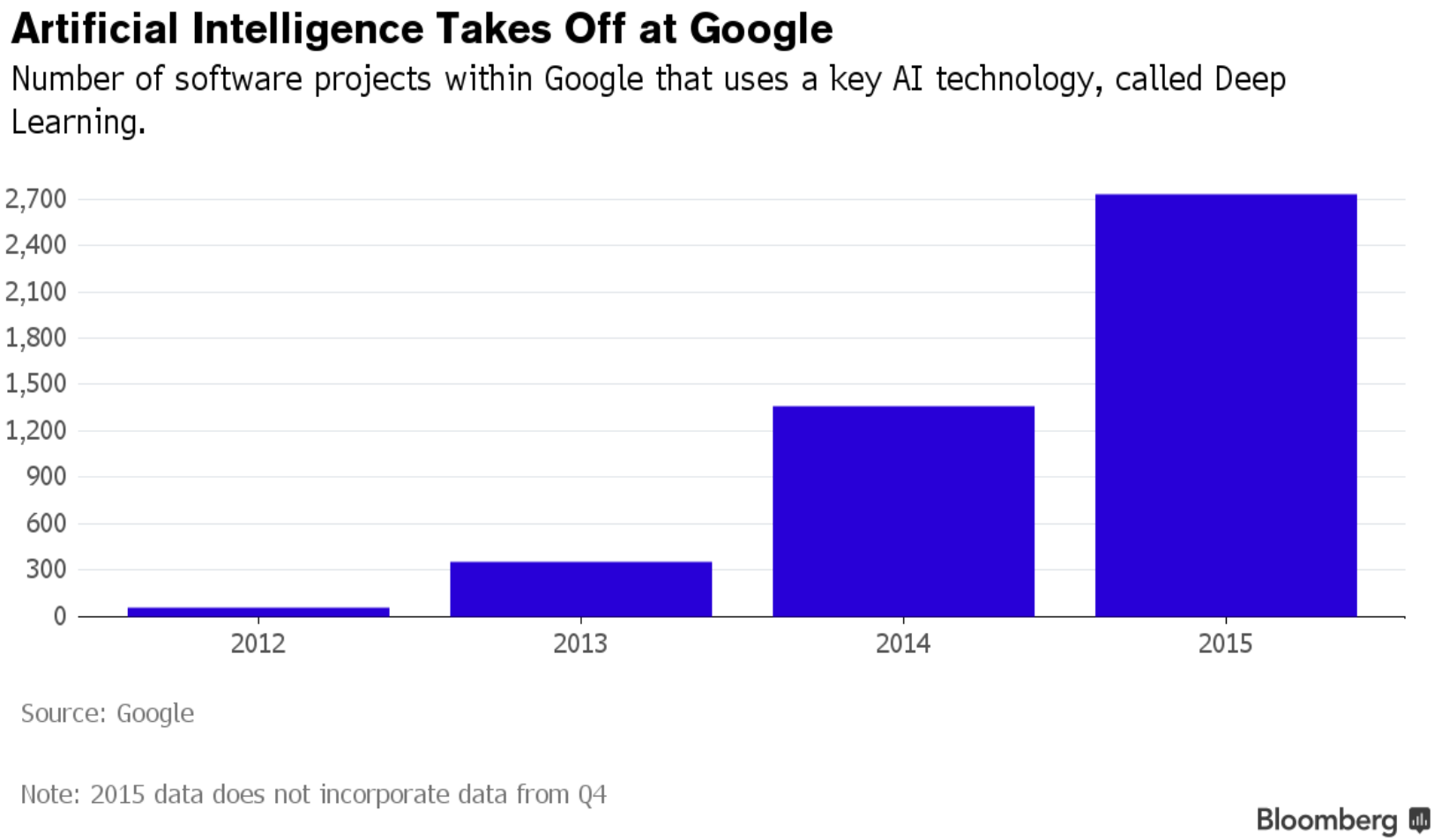
Google NGRAM & Google Trends





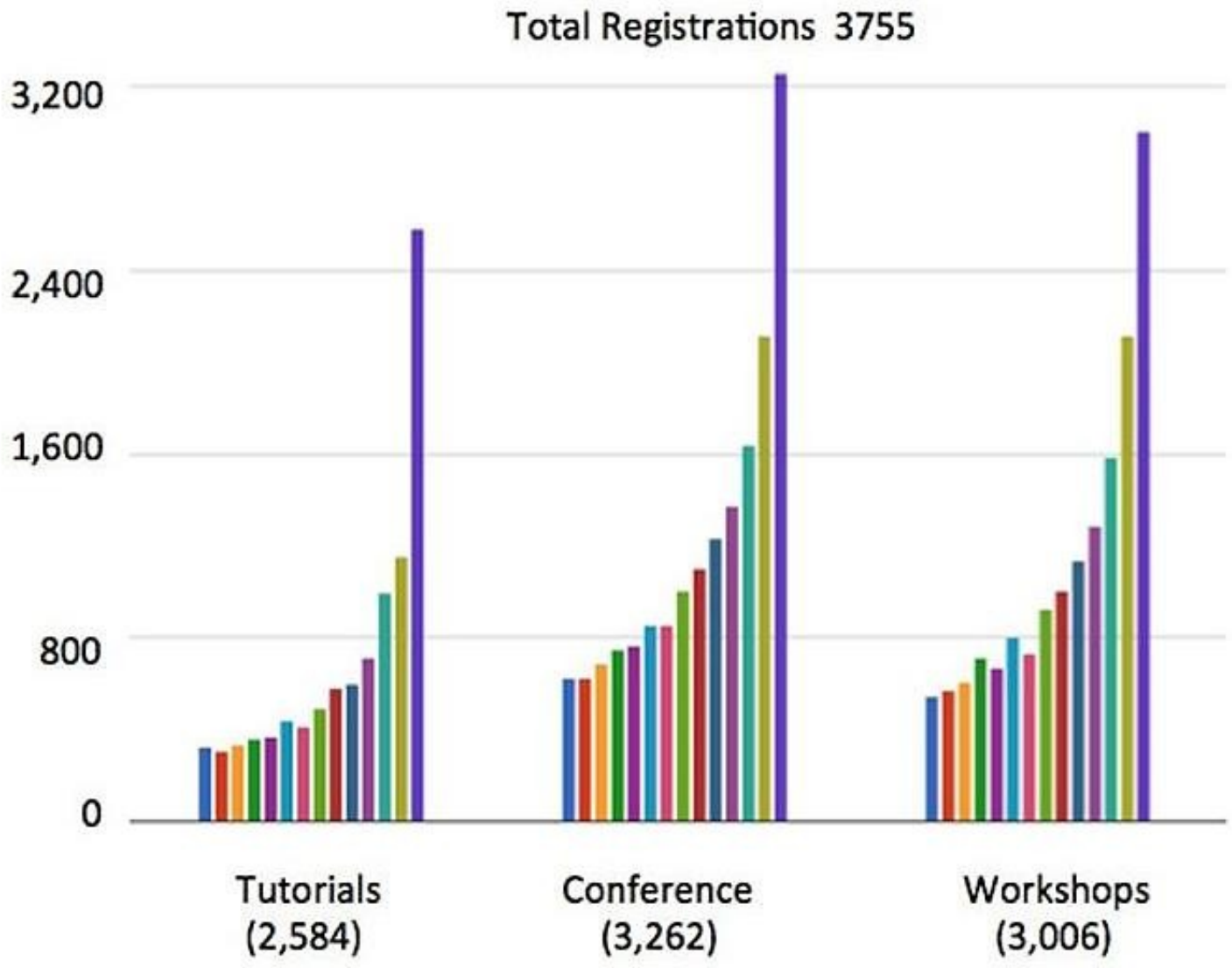
Hype or Reality?

Deep Learning at Google

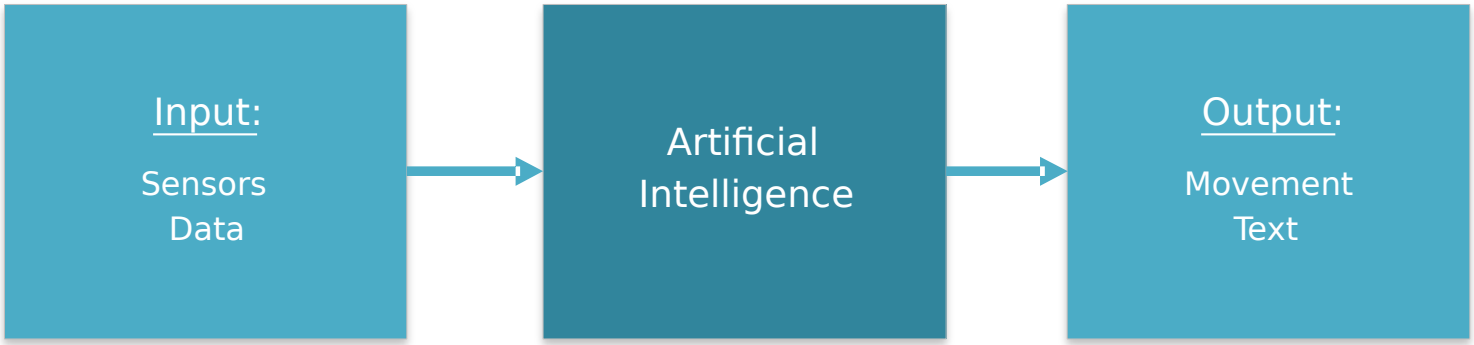


Hype or Reality?

NIPS (Computational Neuroscience Conference) Growth



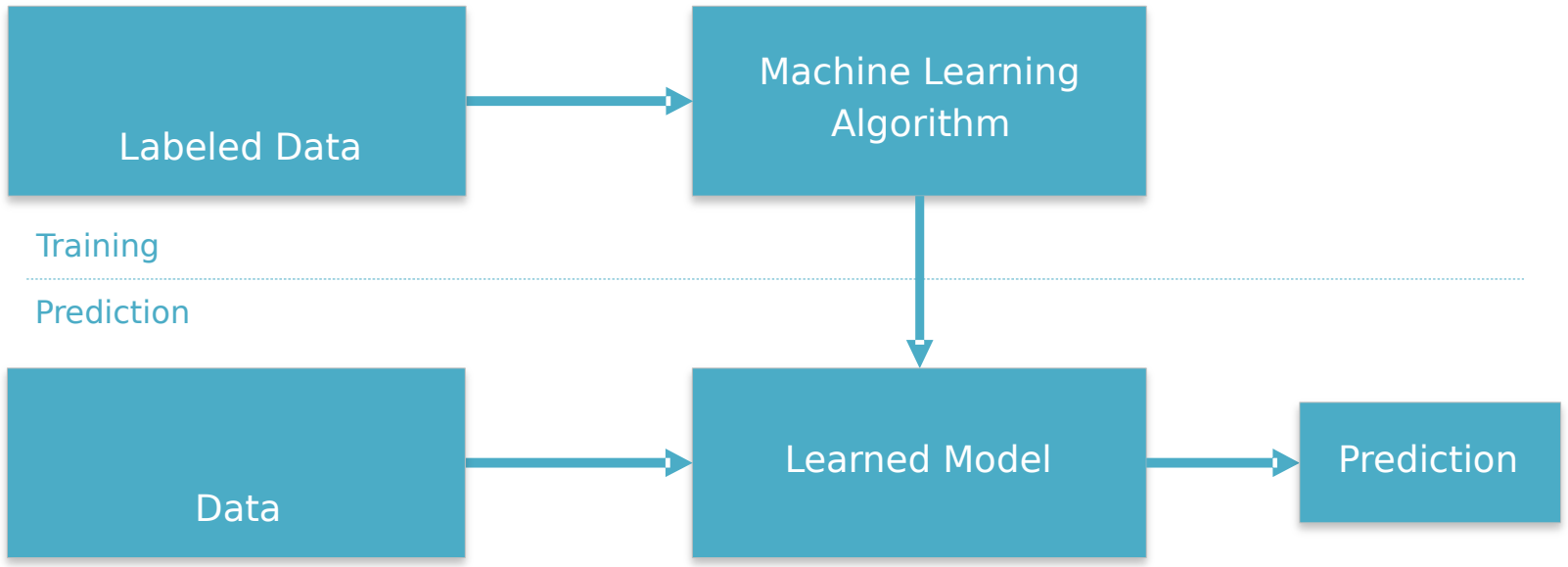
What is Artificial Intelligence?



Machine Learning - Basics

Introduction

Machine Learning is a type of Artificial Intelligence that provides computers with the ability to learn without being explicitly programmed.



Provides various techniques that can learn from and make predictions on data

Machine Learning - Basics

Learning Approaches

Supervised Learning: Learning with a labeled training set

Example: email spam detector with training set of already labeled emails

Unsupervised Learning: Discovering patterns in unlabeled data

Example: cluster similar documents based on the text content

Reinforcement Learning: learning based on feedback or reward

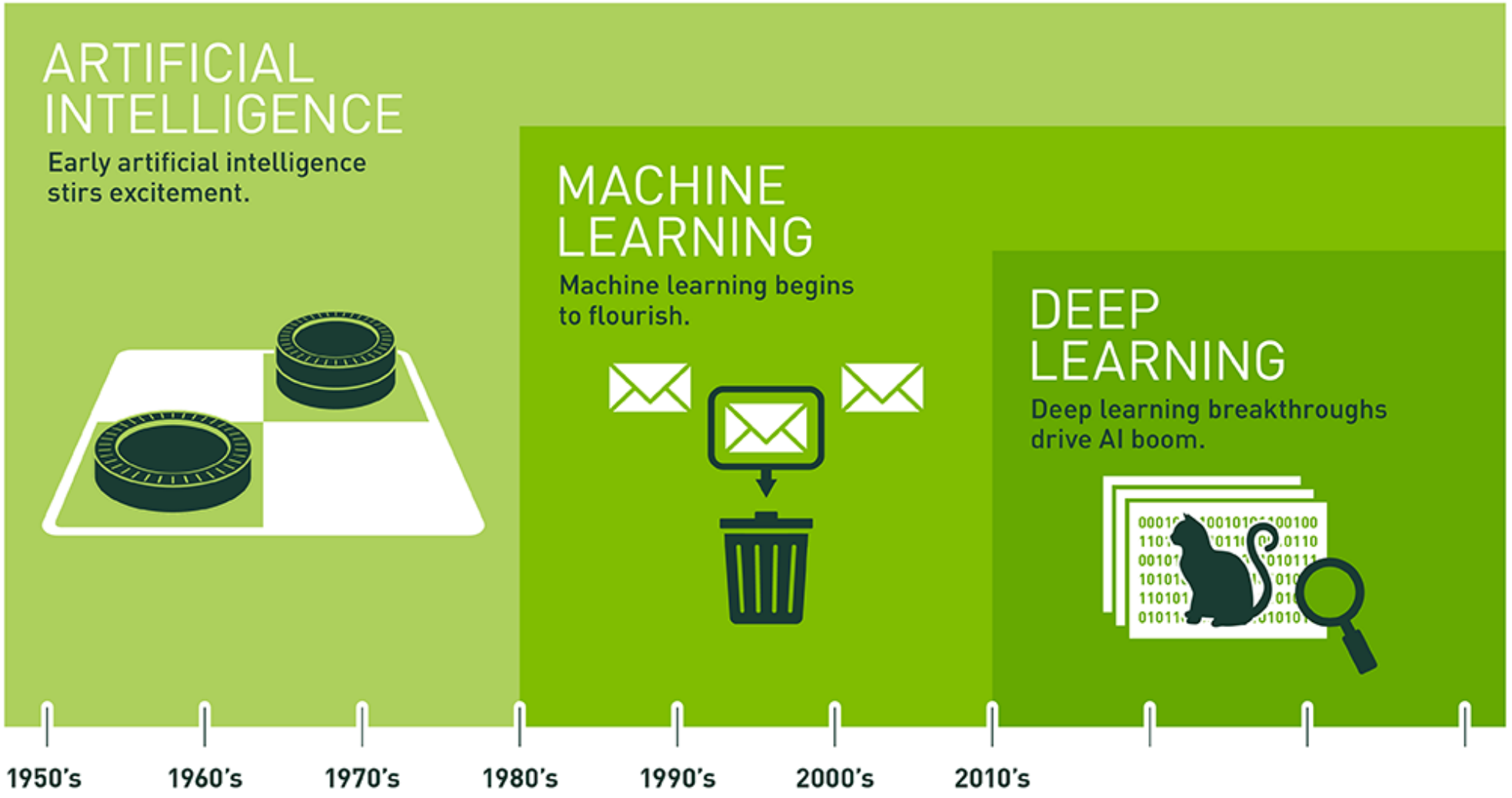
Example: learn to play chess by winning or losing

What is Deep Learning?

Part of the machine learning field of learning representations of data. Exceptional effective at learning patterns.

Utilizes learning algorithms that derive meaning out of data by using a hierarchy of multiple layers that mimic the neural networks of our brain.

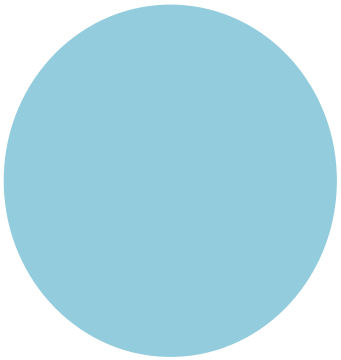
If you provide the system tons of information, it begins to understand it and respond in useful ways.



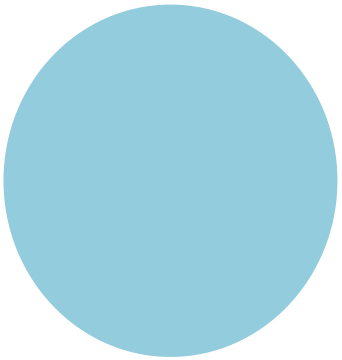
Since an early flush of optimism in the 1950s, smaller subsets of artificial intelligence – first machine learning, then deep learning, a subset of machine learning – have created ever larger disruptions.

Why Deep Learning?

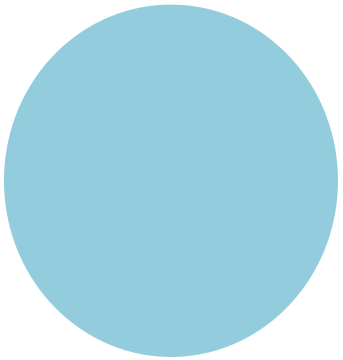
Applications



Speech Recognition



Computer Vision



Natural Language Processing

A brief History

A long time ago...

1958 Perceptron

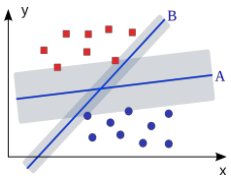
1969
Perceptron criticized

1974 Backpropagation

awkward silence (AI Winter)

Convolution Neural Networks for Handwritten Recognition
1998

1995
SVM reigns



2006
Restricted Boltzmann Machine

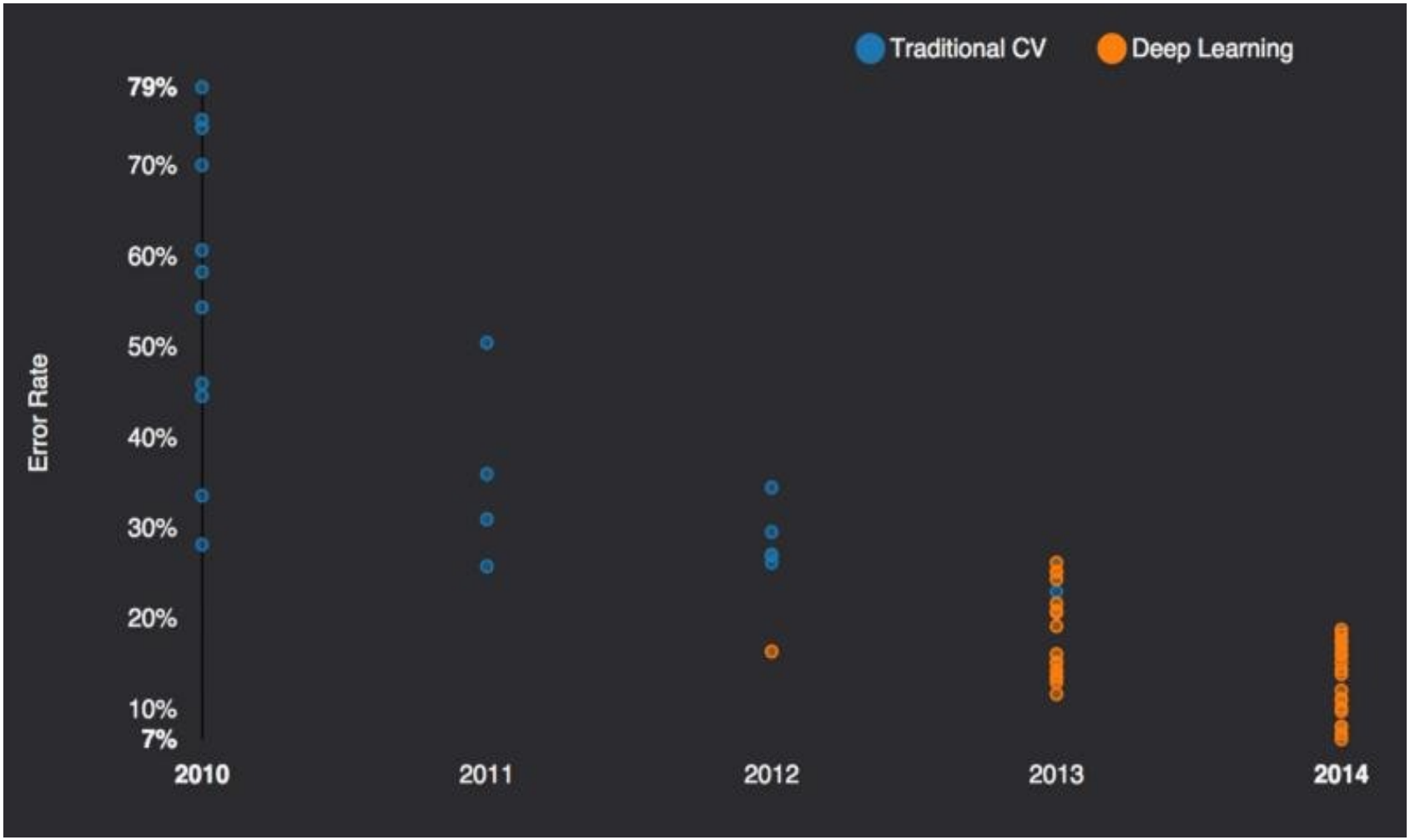
Google Brain Project on 16k Cores
2012

2012
AlexNet wins ImageNet

IMAGENET

A brief History

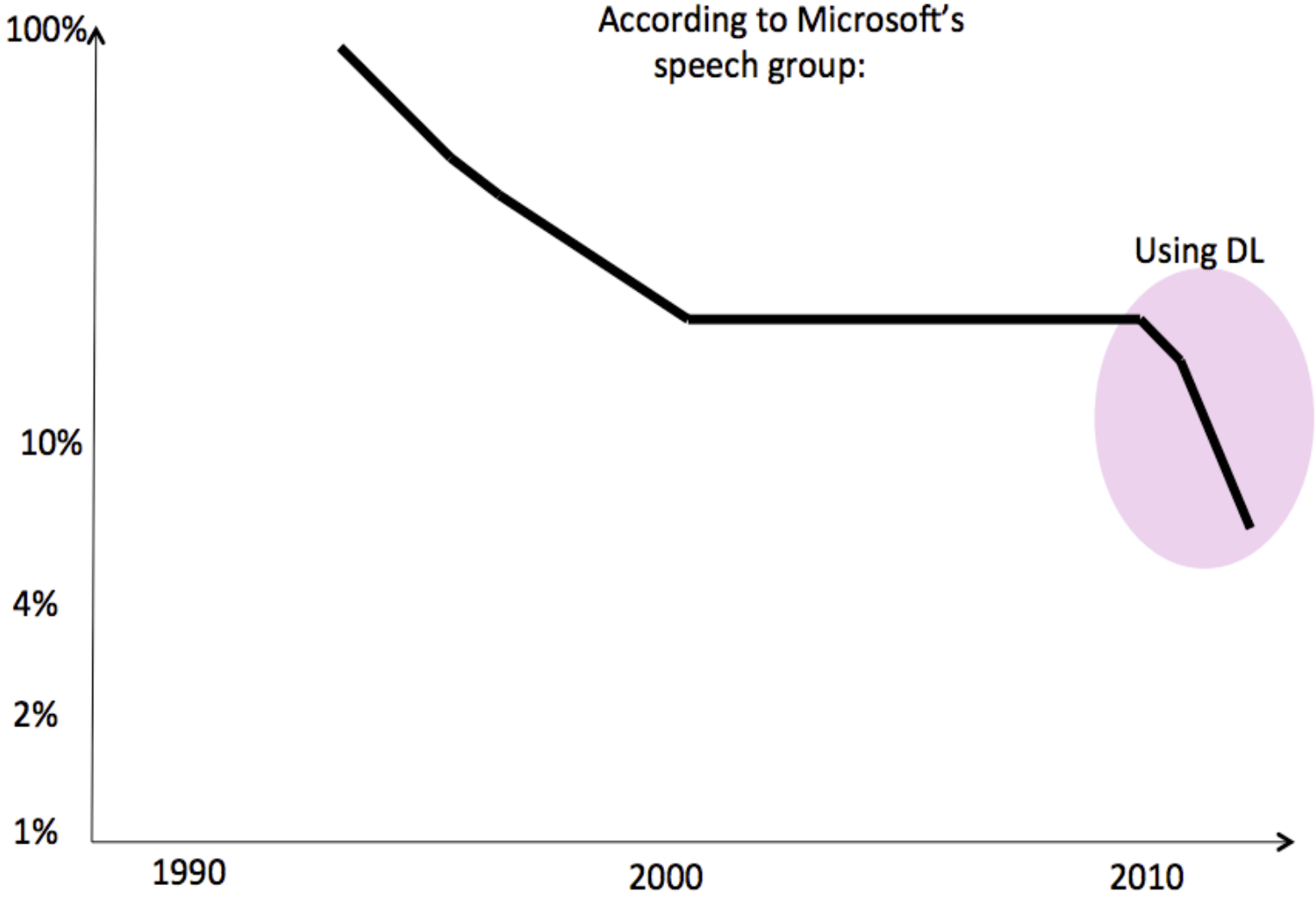
The Big Bang aka “One net to rule them all”



ImageNet: The “computer vision World Cup”

A brief History

The Big Bang aka “One net to rule them all”



Deep Learning in Speech Recognition

The Big Players

Superstar Researchers

Geoffrey Hinton: University of Toronto & Google

Yann LeCun: New York University & Facebook

Andrew Ng: Stanford & Baidu

Yoshua Bengio: University of Montreal

Jürgen Schmidhuber: Swiss AI Lab & NNAISENSE

Big Players(Companies)

- Google DeepMind
  - Deepmind always have a preference of using reinforcement learning in their approach. They certainly use deep learning as a component in most of their research, but always seem to emphasize it's combination with reinforcement learning.
  - They also have the focus on attention mechanism and memory augmented networks. In terms of the breadth of research I don't think there is any organization remotely close to DeepMind.
  - The research is driven by the need to discover the nature of Intelligence.

Big Players(Companies)

- Google Brain
  - Google has a distinct pragmatic and Engineering airport in how they undertake their research.
  - They made endless tweaks on the Inception architect. Google also combines traditional algorithms like beam search,graph traversal and generalized linear models with deep learning.
  - In addition, Google also construct their own hardware. They have deep learning specific ASIC which is on it 2nd generation, capable of a massive 180 tera operations /second.

Big Players(Companies)

- Facebook

  They have done some good work applying DL in certain problems. it is difficult to see if there is any particular research preference.

- Microsoft
  - Similar to Google in their approach. They have a top notch computer science talent that led to the discovery of Residual networks. Their cognitive Toolkit, despite coming late in the game, is a piece of high quality engineer. Probably second to Google in the research contributions

Big Players(Companies)

- Open AI
  - Tends to favour the approach of using generative models ,more specifically generative adversarial networks. they also made serious effort towards reinforcement learning space
- Nvidia
  - They have the best engineering team out there tweaking the GPU to get maximum performance.
  - They are completely absent in terms call academic papers, but you can trust them to have spend serious effort in building competition resources required for deep learning.
  - Nvidia latest Volta best has Incorporated deep learning specific component call Tensor Core capable of 120 trillion operations per second.
  - Their end to end deep learning solution for self driving Cars is one of those DL research papers that is notable.

Agenda:

Deep Learning: Introduction

Foundation:Linear Regression

Deep Learning: Going Deep:Forward pass

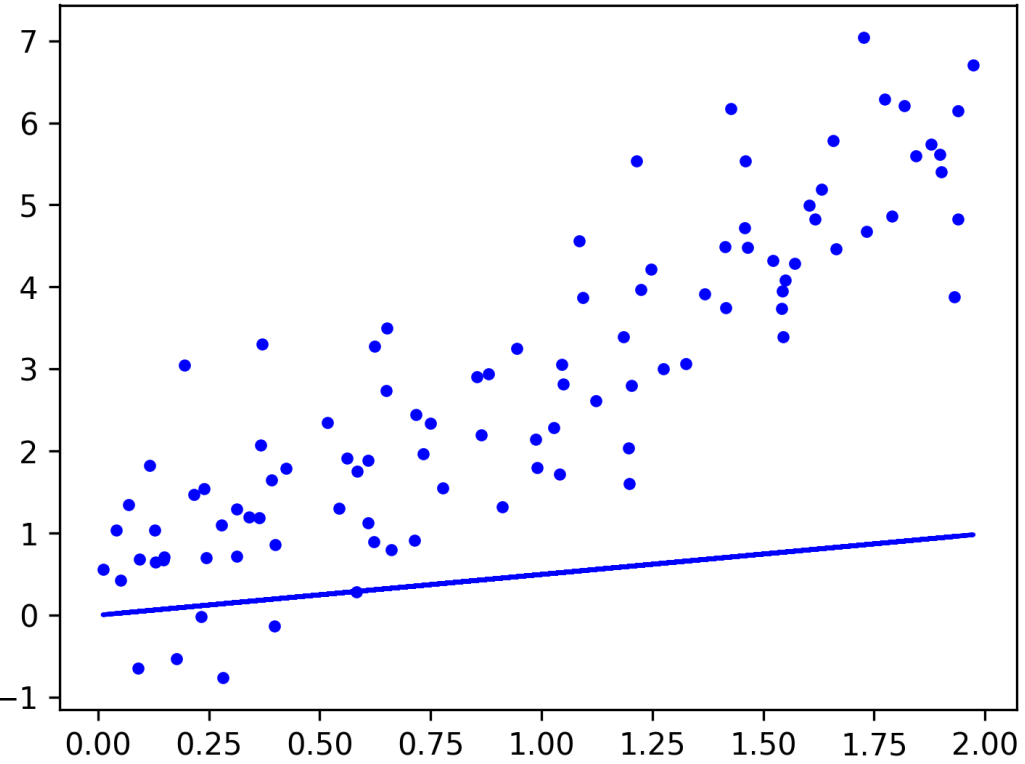
Deep Learning: Going Deep:Back Prop

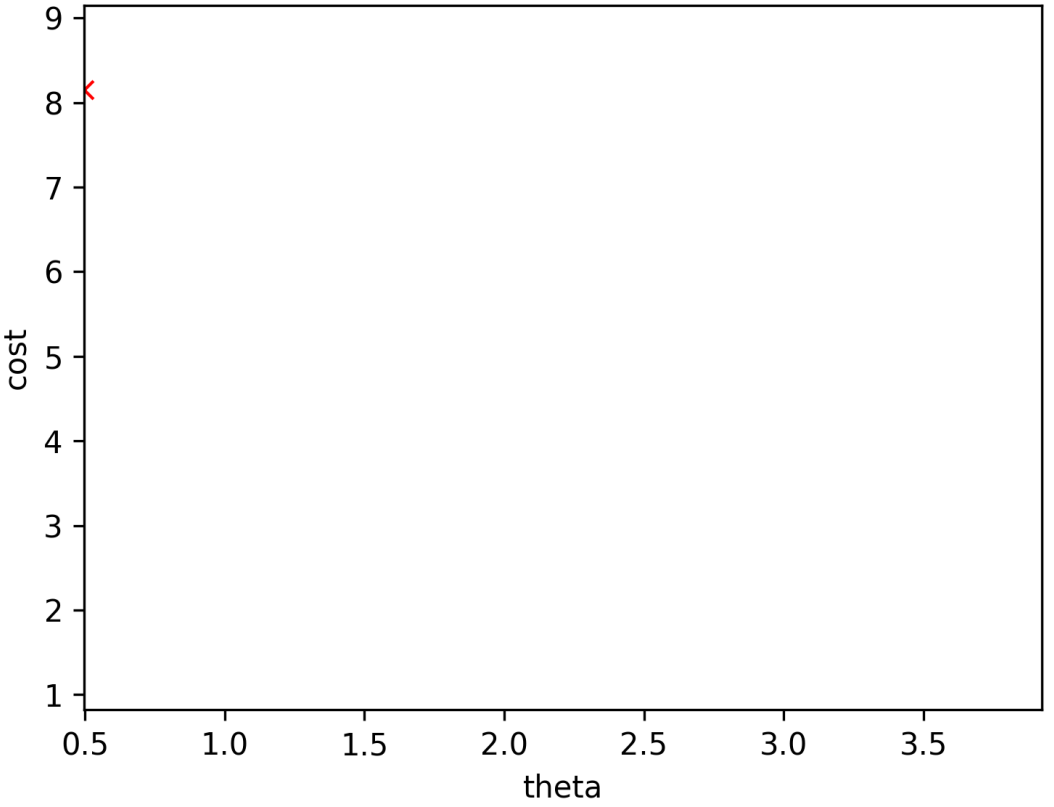
Deep Learning: Architectures

Deep Learning: Fintechs

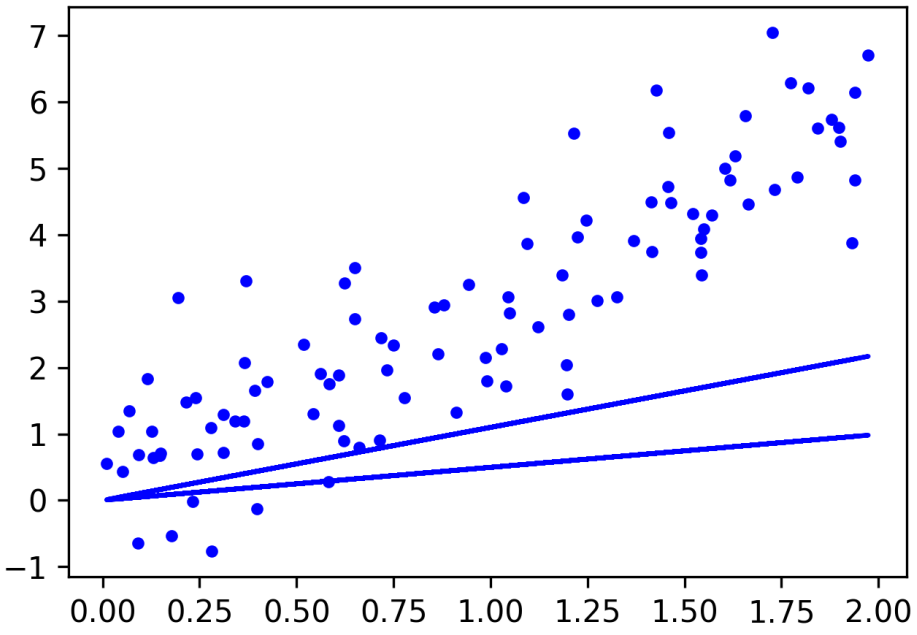
Deep Learning: GPU and TPU

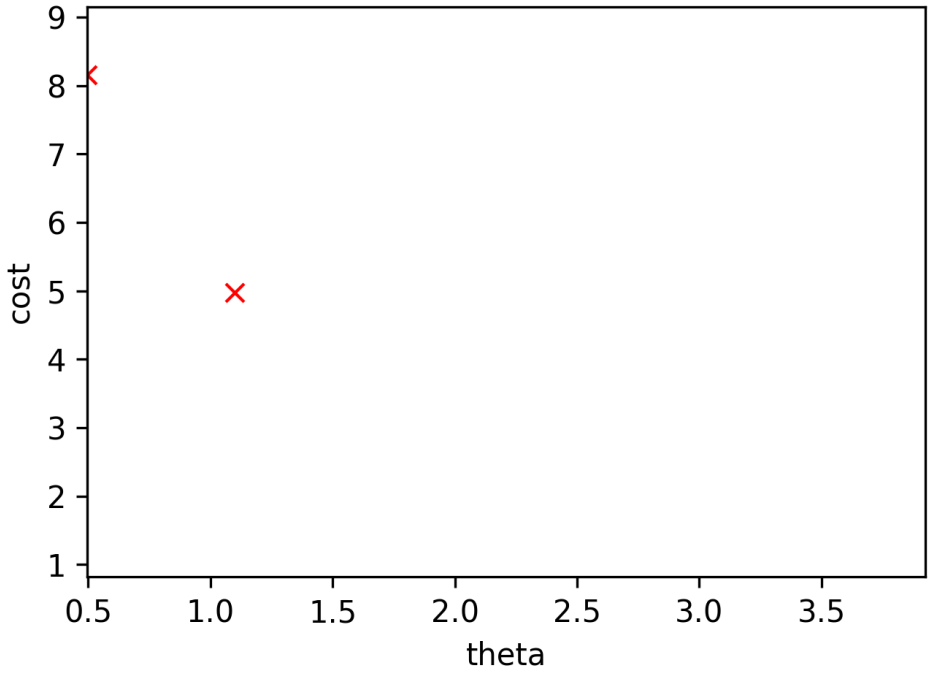
Linear Regression:BatchGradientDescent



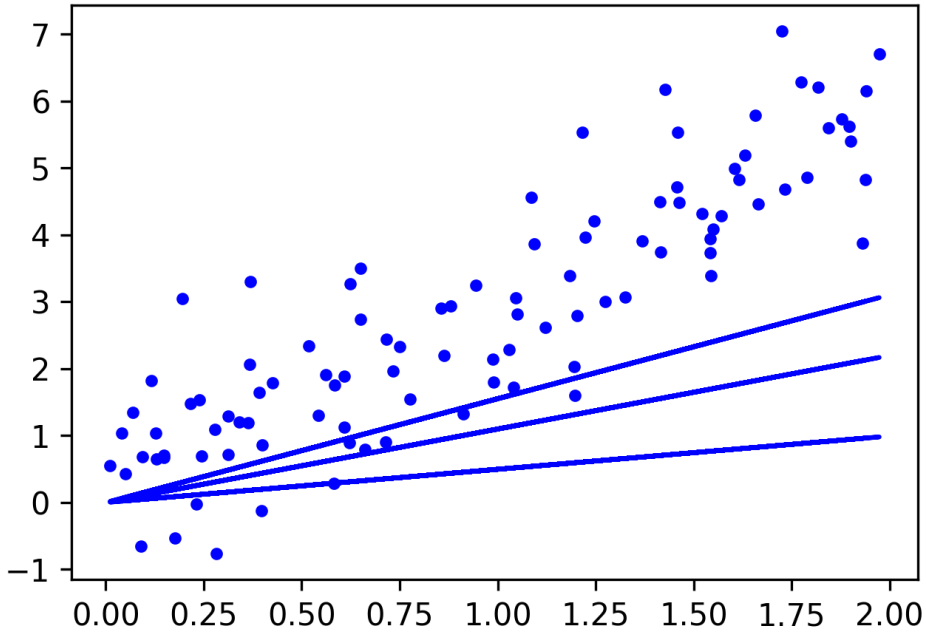


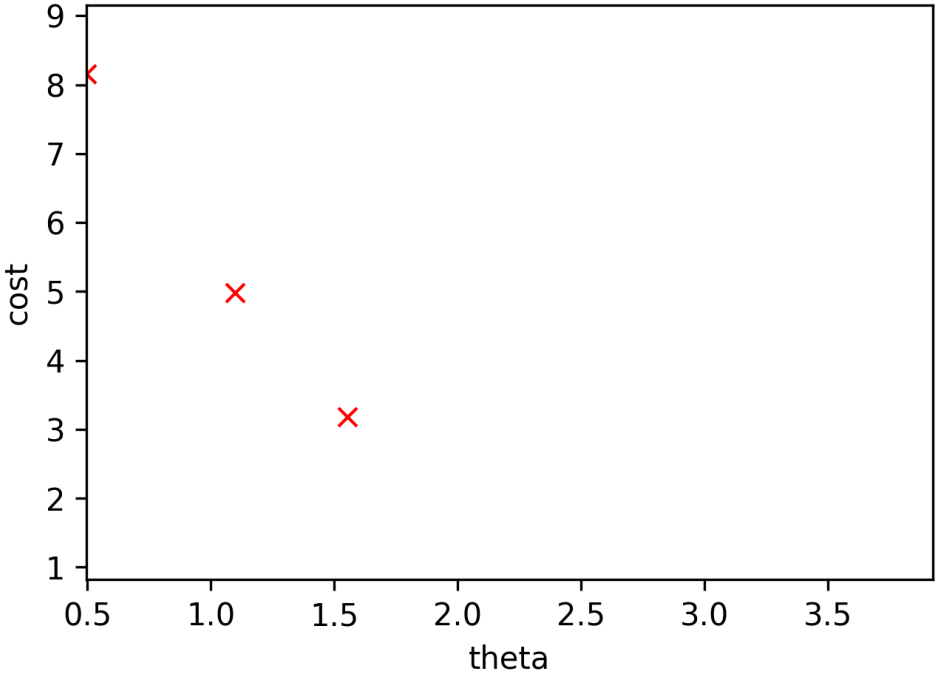
Linear Regression:BatchGradientDescent



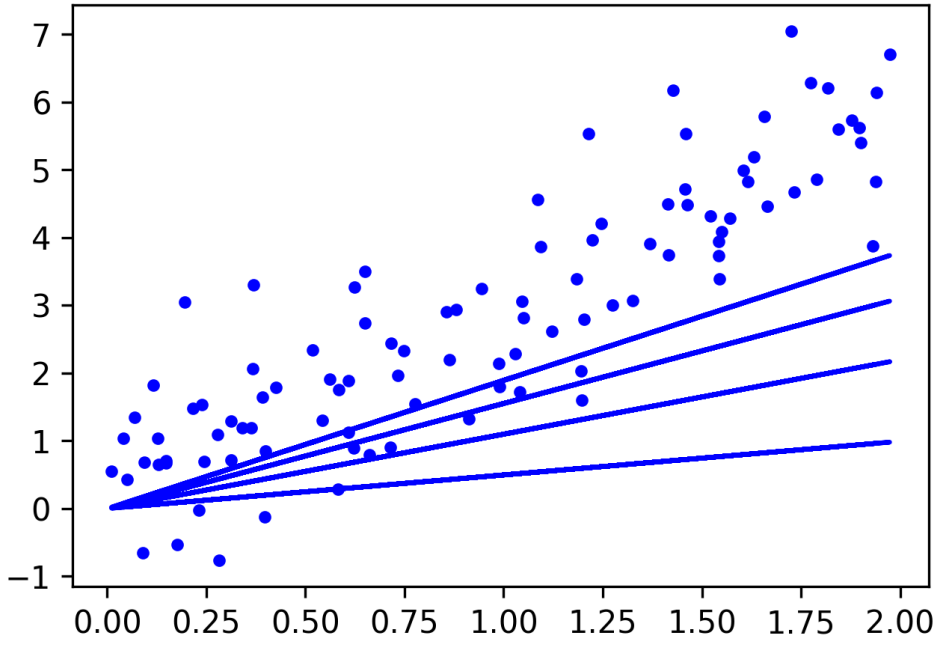


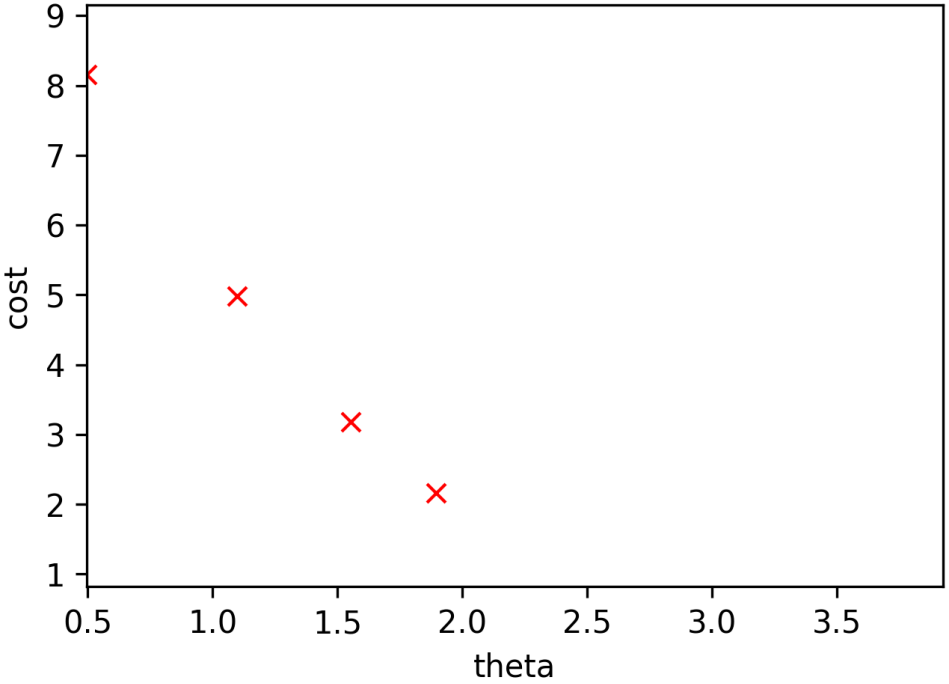
Linear Regression:BatchGradientDescent



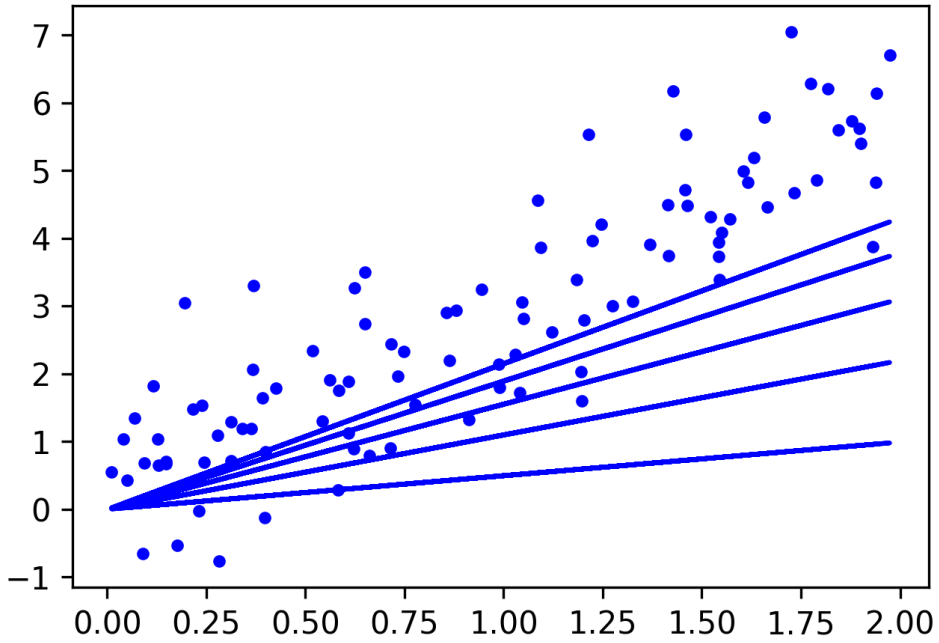


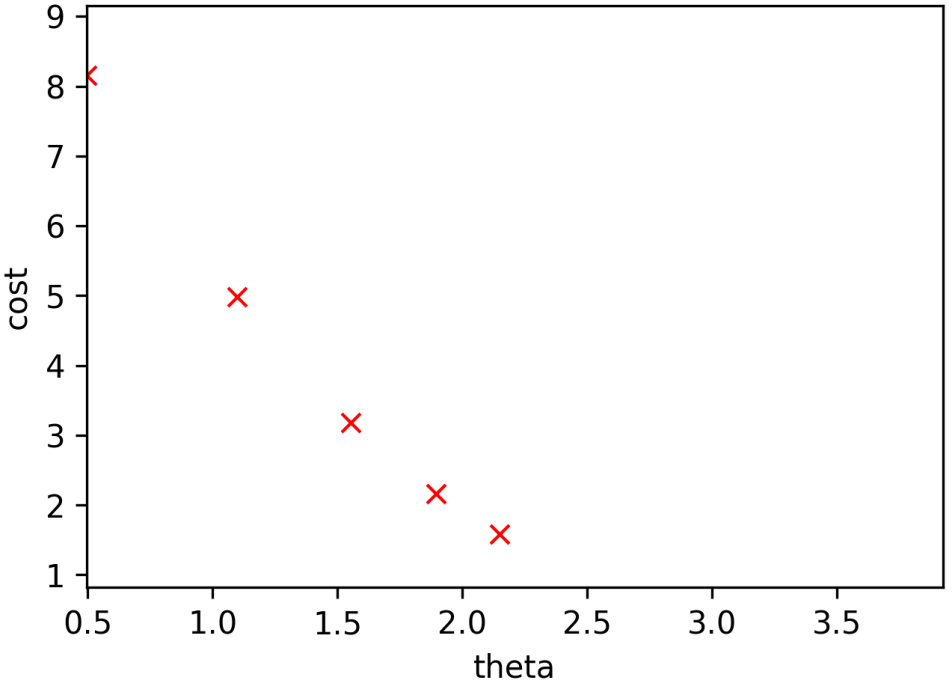
Linear Regression:BatchGradientDescent



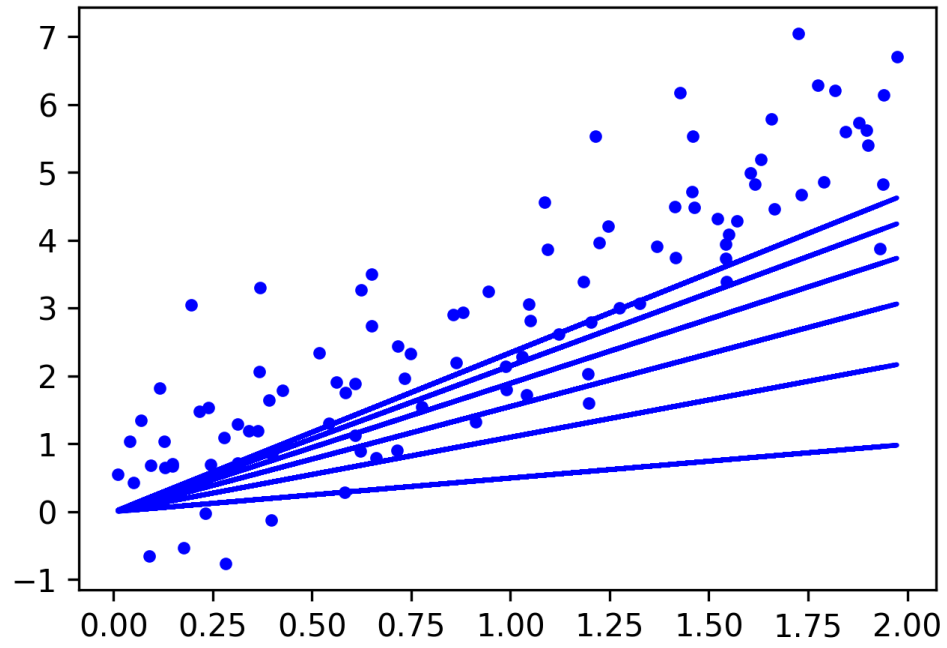


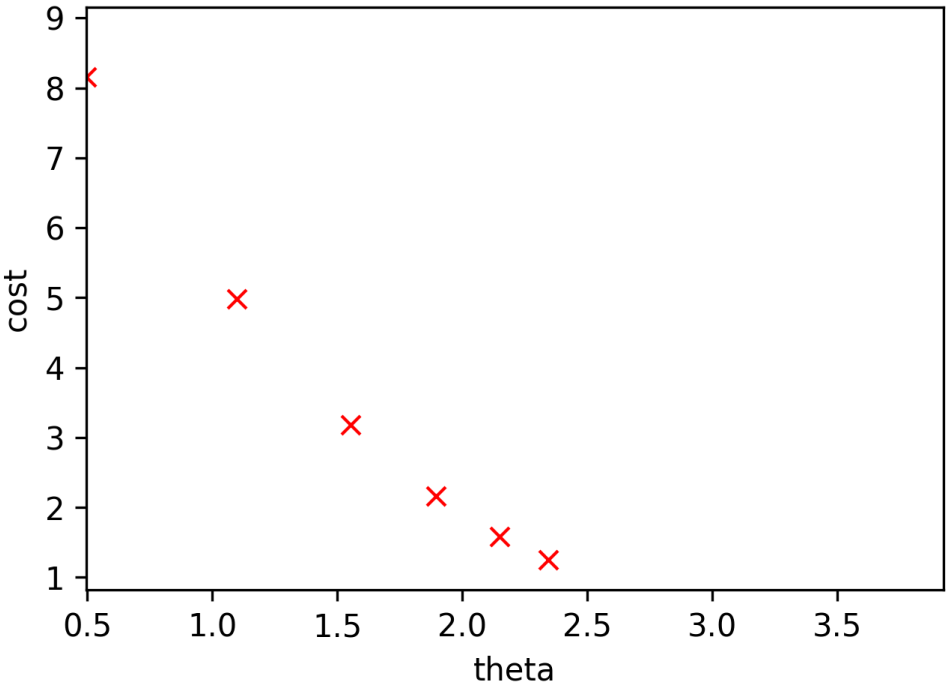
Linear Regression:BatchGradientDescent



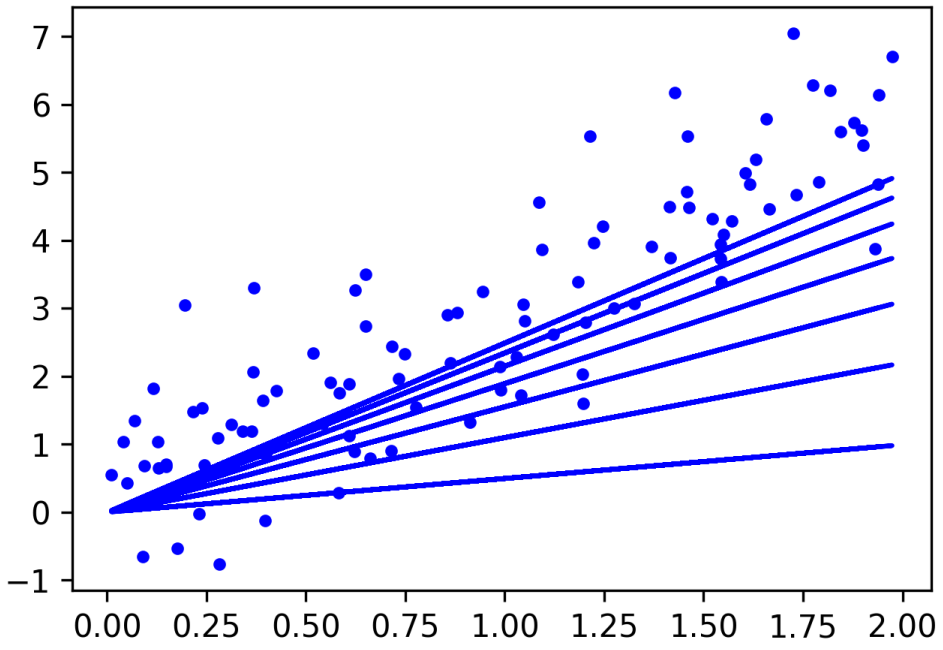


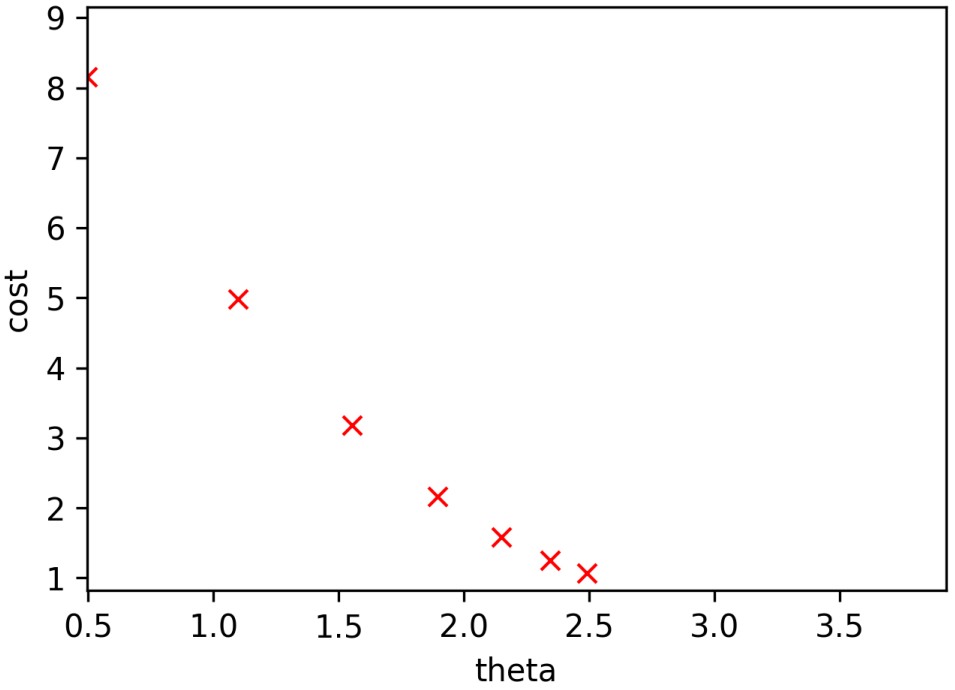
Linear Regression:BatchGradientDescent



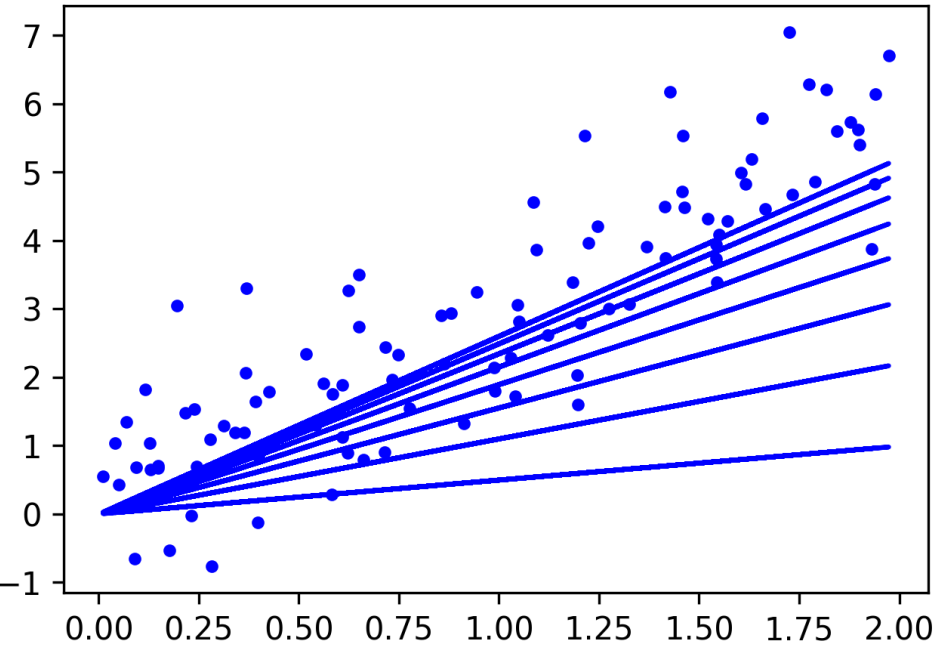


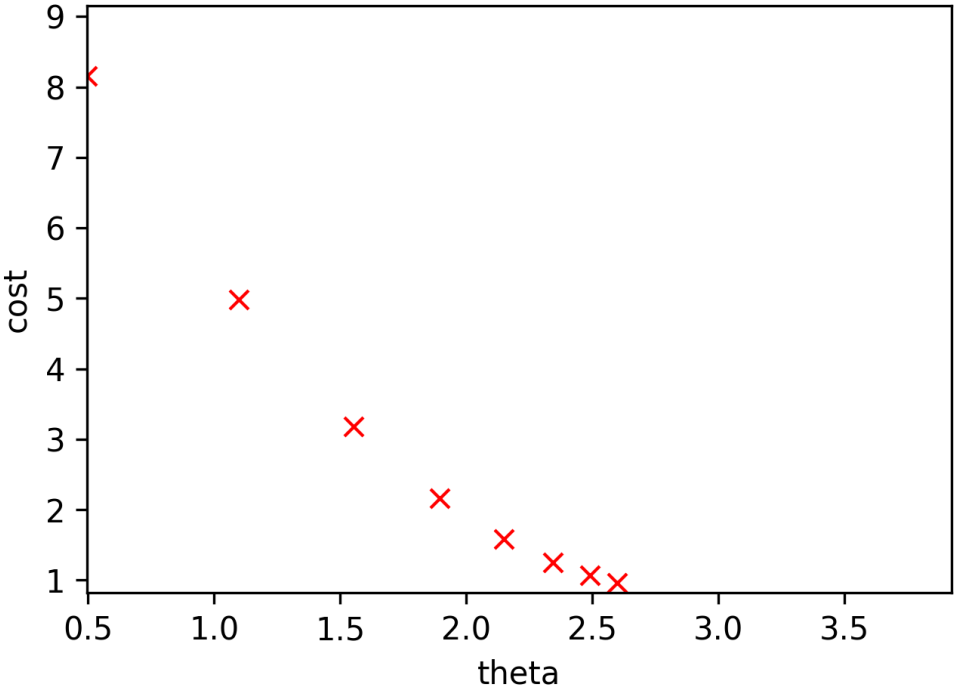
Linear Regression:BatchGradientDescent



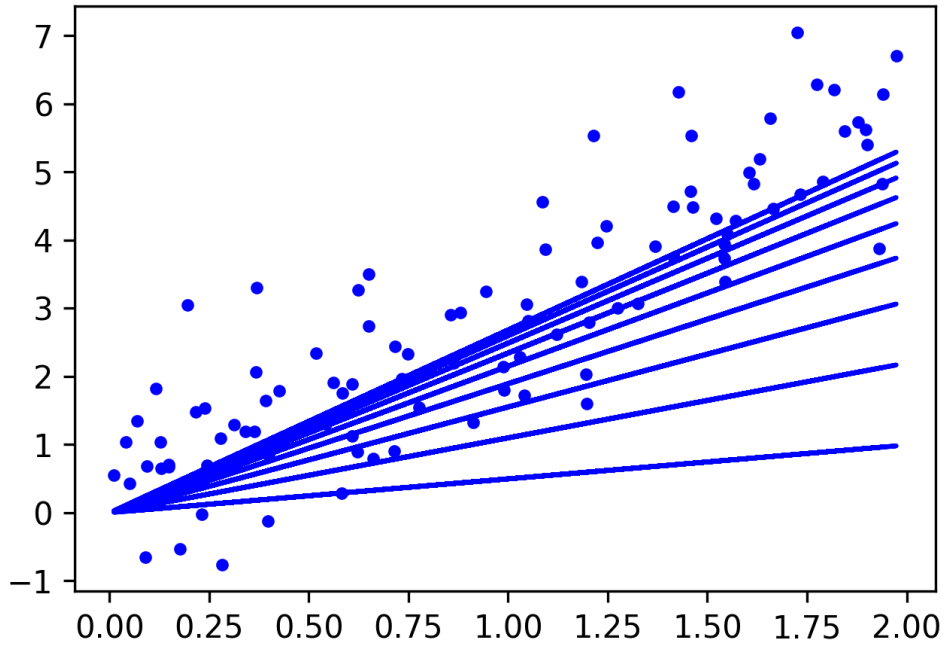


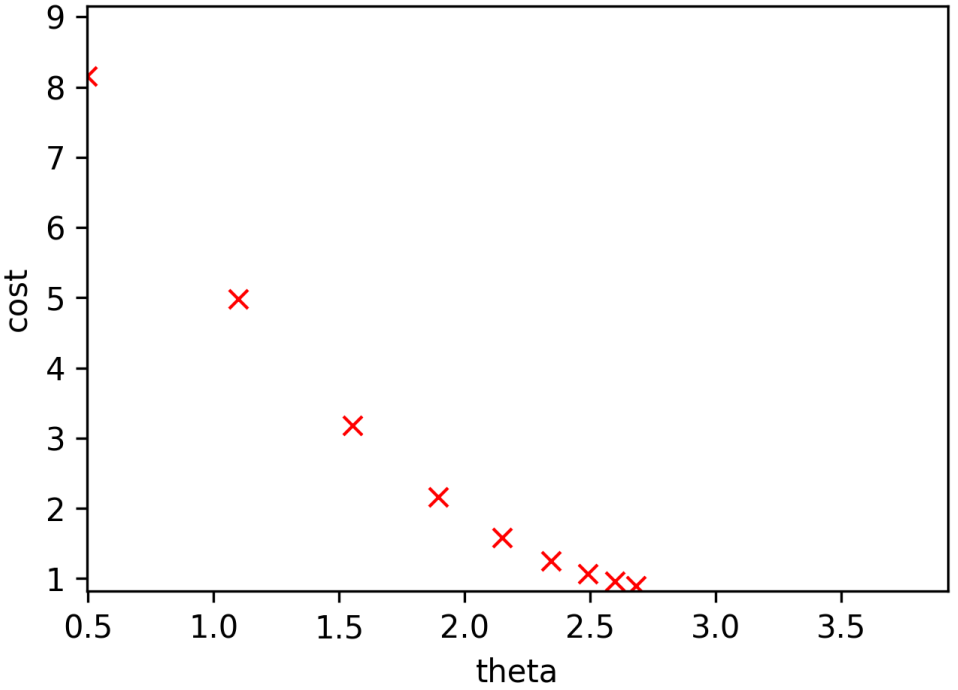
Linear Regression:BatchGradientDescent



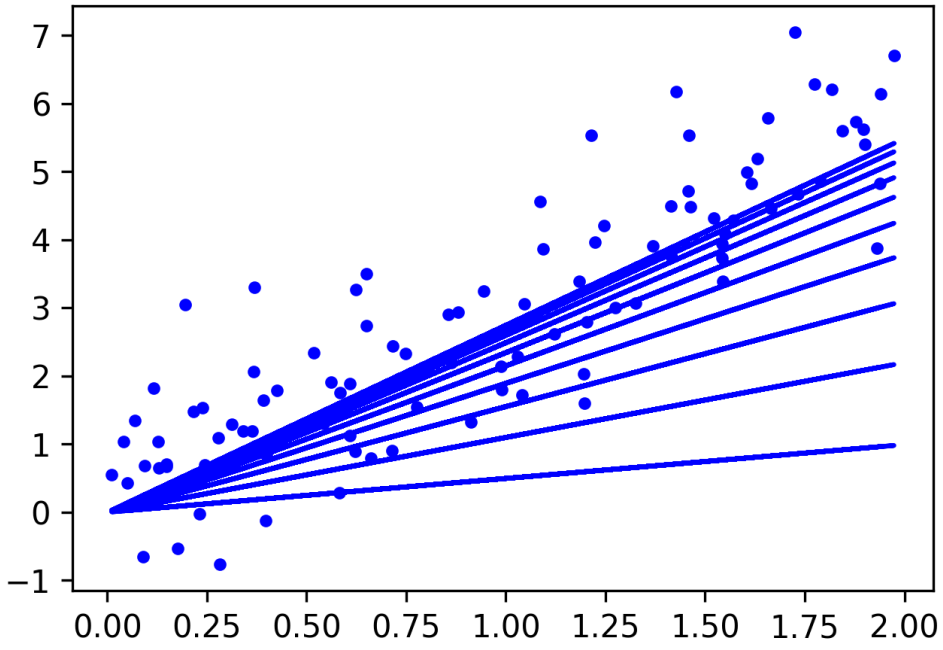


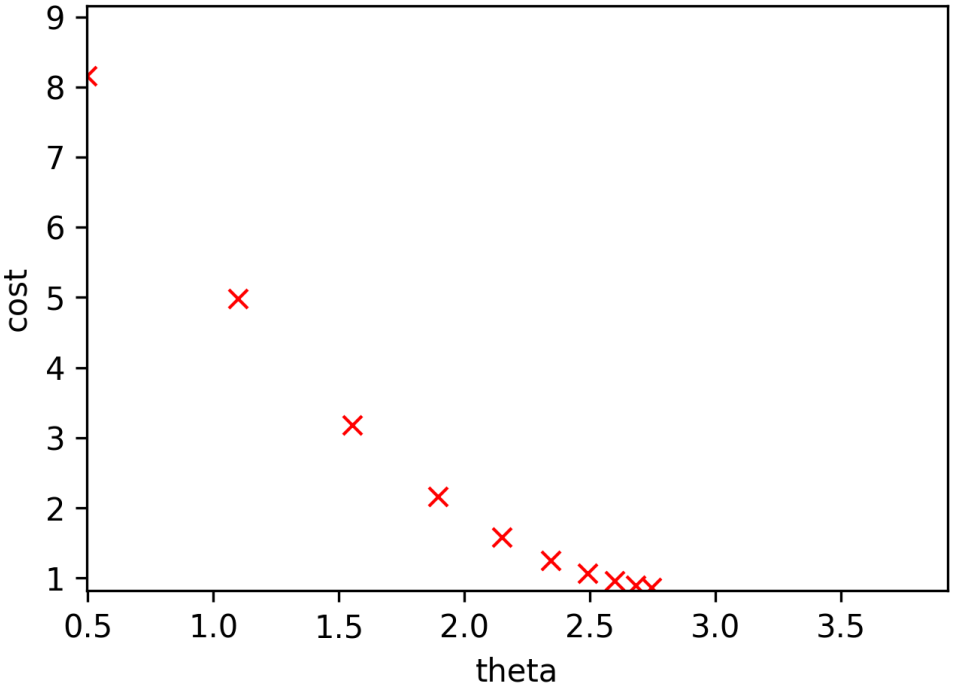
Linear Regression:BatchGradientDescent





Linear Regression:BatchGradientDescent





Linear Regression:BatchGradientDescent

$$J(\theta_0, \theta_1) = 1/m \sum_{i=1}^{m} (h(x)^i - y^i)^2$$

$$= 1/m \sum_{i=0}^{m} (\theta_0 + \theta_1 x^i - y^i)^2$$

$$\partial/\partial_{\theta_1} J(\theta_0, \theta_1) = 2/m \sum_{i=1}^{m} (\theta_0 + \theta_1 x^i - y) \cdot x^i$$

$$\theta_j := \theta_j - \alpha \, \partial/\partial_{\theta_j} J(\theta_0, \theta_1)$$

```
def costfunc(X,theta):
    m = len(X)
    y_predict = X.dot(theta)
    return np.sum(((y-y_predict)**2)/m)

def thetaTransposeX(X,theta):
    return X.dot(theta)

def gradient(X,theta):
    m = len(X)
    return 2/m * X_b.T.dot(X_b.dot(theta) - y)

def plot_gradient_descent(theta, eta, theta_path=None):
    m = len(X_b)
    n_iterations = 1000
    for iteration in range(n_iterations):
        if iteration < iterations_to_track:
            y_predict=thetaTransposeX(X_b,theta)
            cost=costfunc(X_b,theta)
            if theta_path_bgd_10 is not None:
                theta_path_bgd_10.append(theta[0][0])
            if cost_path_bgd_10 is not None:
                cost_path_bgd_10.append(cost)
            if y_predict_bgd_10 is not None:
                y_predict_bgd_10.append(y_predict)
        gradients = gradient(X_b,theta)
        theta = theta - eta * gradients
    print("theta best:",theta)

rnd.seed(42)
theta = rnd.randn(1,1)  # random initialization
plot_gradient_descent(theta, eta=0.1)
minx=0.0
maxx=10.0
```

Linear Regression:BatchGradientDescent

```
def plot_progress(xvec,ttxvec,theta,cost):
    fig = plt.figure(figsize=(10, 4))
    sub1 = plt.subplot(1, 2, 1)
    sub2 = plt.subplot(1, 2, 2)
    style = "b-"
    sub1.plot(X, y, "b.")
    xmin=np.amin(theta)
    xmax=np.amax(theta)+1
    ymin=np.amin(cost)
    ymax=np.amax(cost)+1
    for i,ttx in enumerate(ttxvec):
        sub1.plot(X, ttx.T[0], style)
        plt.xlabel("theta")
        plt.ylabel("cost")
        axes = plt.gca()
        axes.set_xlim([xmin,xmax])
        axes.set_ylim([ymin,ymax])
        sub2.plot(theta[i], cost[i], "rx")
        fl.save_fig("progress"+str(i))
```

plots figure-1
1. x against y
2. and overlays predicted y.

plots figure-2
1. theta against cost.
2. easier to plot because θ is scalar.

Linear Regression:Gradient Descent Algorithm

hypothesis $\Rightarrow$ $h(x) = \theta_0 + \theta_1 x$

cost $\Rightarrow$ $J(\theta_0, \theta_1) = \frac{1}{m} \sum_{i=1}^{m} (h(x^{(i)}) - y^{(i)})^2$

Gradient Descent $\Rightarrow$ repeat until convergence {

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$$

for $j = 1$, $j = 0$

}

Linear Regression:Gradient Descent Algorithm

1. $J(\theta_0, \theta_1) = \frac{1}{m} \sum_{i=1}^{m} (h(x)^{(i)} - y^{(i)})^2 \Rightarrow \frac{1}{m} \sum_{i=1}^{m} (f(\theta_0, \theta_1)^{(i)})^2$

2. $f(\theta_0, \theta_1)^{(i)} = \theta_0 + \theta_1 x - y$

3. $J(f(\theta_0, \theta_1)^{(i)}) = \frac{1}{m} \sum_{i=1}^{m} (\theta_0 + \theta_1 x - y^{i})^2$

4. $\frac{\partial}{\partial \theta_0} (J\theta_0 \theta_1) = \frac{\partial}{\partial_0} \frac{1}{m} \sum_{i=1}^{m} (f(\theta_0, \theta_1)^{i})^2 = 2/m \sum_{i=1}^{m} (f(\theta_0, \theta_1)^{(i)})^{(2-1)}$

- treating function f as opaque

- It is like finding a chain of derivatives. The first one is the cost function and the second one is the hypothesis.

Linear Regression:Gradient Descent Algorithm

5. $$\frac{\partial}{\partial\theta_0} f(\theta_0,\theta_1)^i = \frac{\partial}{\partial\theta_0}(\theta_0+\theta_1 x^{(i)}-y^{(i)})$$

$$= \frac{\partial}{\partial\theta_0}(\theta_0 + [a\ num][a\ num] - [a\ num]$$

$$= \theta_0' = 1\times\theta^{1-1} = 1$$

6 $$\frac{\partial}{\partial\theta_0} g(f(\theta_0\theta_1)^{(i)}) = \frac{\partial}{\partial\theta_0} g(\theta_0\theta_1) f(\theta_0\theta_1)$$

$$= 2/m \sum_{i=1}^{m} (\theta_0+\theta_1 x^{(i)}-y^{(i)}) \times 1$$

Linear Regression:Gradient Descent Algorithm

With respect to θ_1

7. $$\frac{\partial}{\partial \theta_1} f(\theta_0 \theta_1)^i = \frac{\partial}{\partial \theta_1} (\theta_0 + \theta_1 x - y)$$

$$= \frac{\partial}{\partial \theta_1} ([\text{a num}] + \theta_1 [\text{a num}] - [\text{a num}])$$

$$= 0 + \theta_1 x^i - 0$$

$$= 0 + 1 \times \theta_1^{i-1} \times x^i - 0$$

$$= x^i$$

8. $$\frac{\partial}{\partial \theta_1} g(f(\theta_0 \theta_1)^i) = \frac{\partial}{\partial \theta_1} g(\theta_0 \theta_1) \frac{\partial}{\partial \theta_1} f(\theta_0 \theta_1)^i$$

$$= \boxed{2/m \sum_{i=1}^{m} (\theta_0 + \theta_1 x^i - y^i)} \times \boxed{x^i}$$

Agenda:

Deep Learning: Introduction

Foundation:Linear Regression

Deep Learning: Going Deep:Intuition

Deep Learning: Going Deep:Forward pass

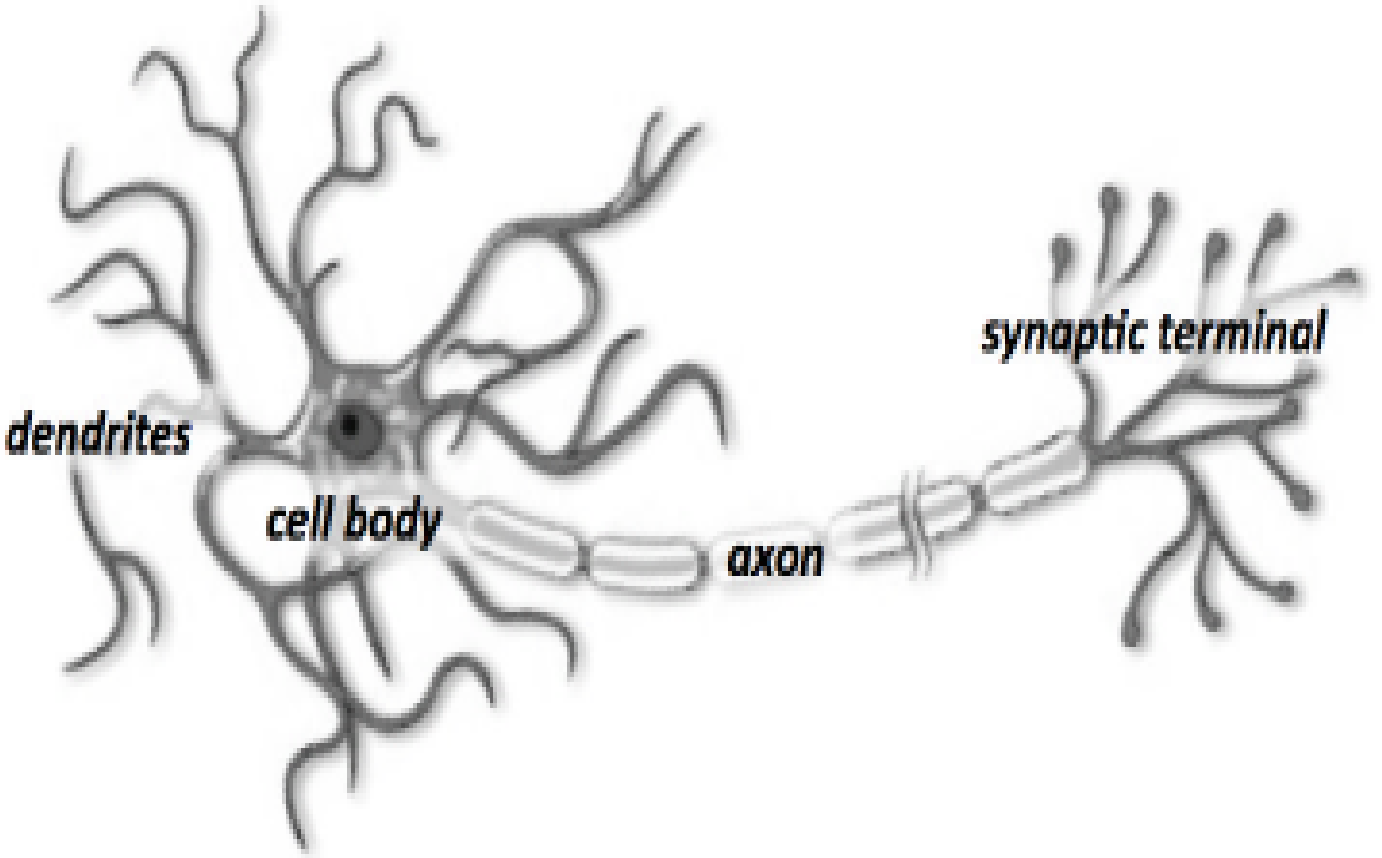
Deep Learning: Going Deep:Back Prop

Deep Learning: Architectures

Deep Learning: Fintechs

Deep Learning: GPU and TPU

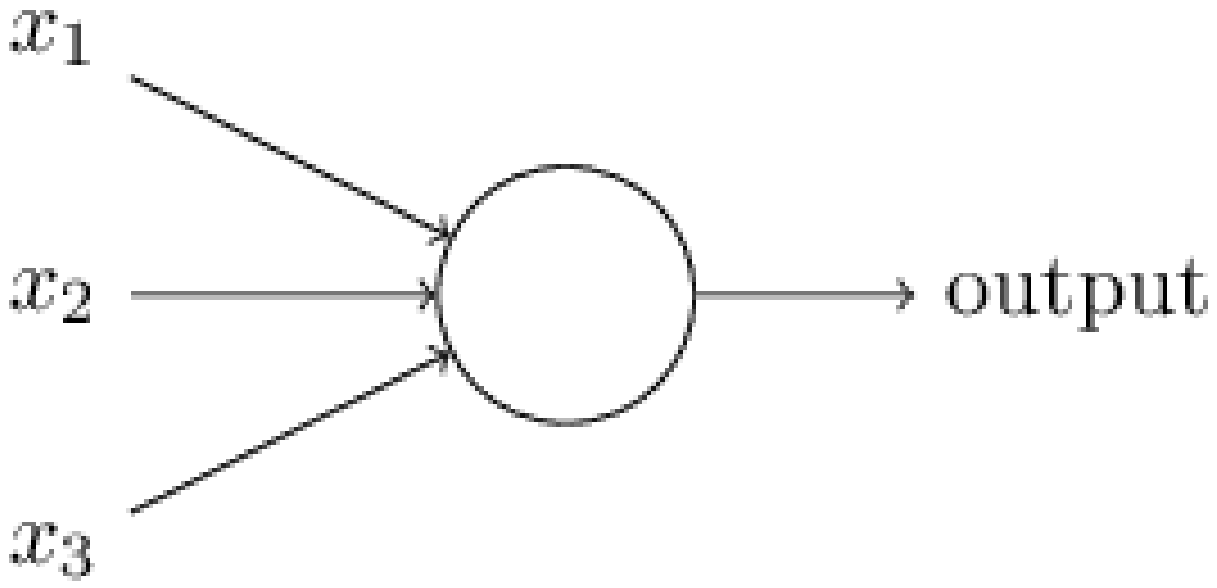
Neurons in the brain



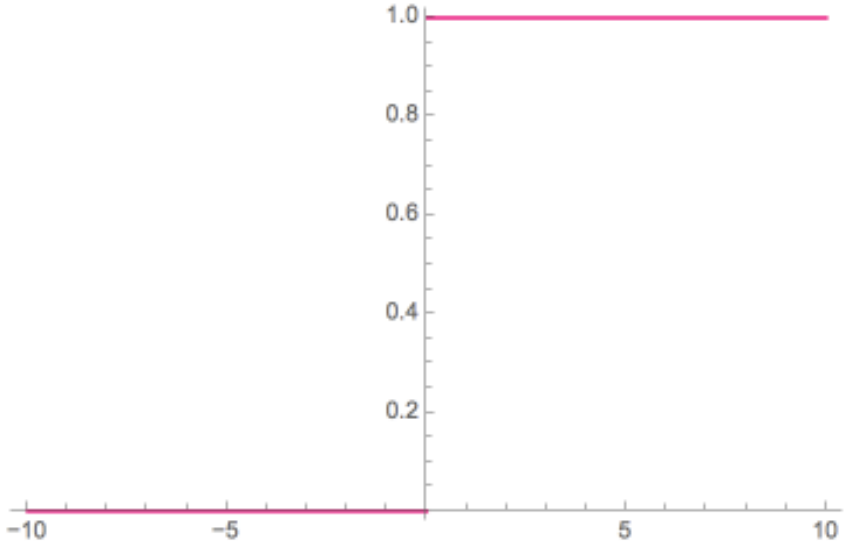
Artificial Neurons:Perceptrons

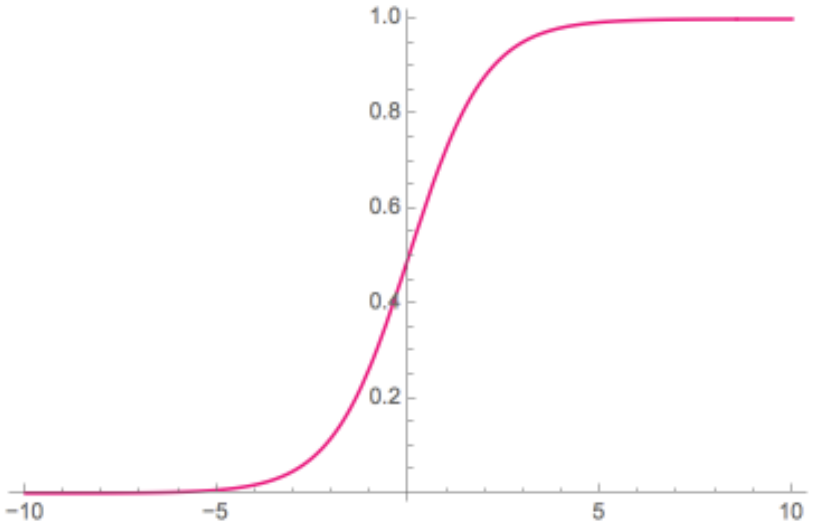
- Suppose the weekend is coming up, and you've heard that there's going to be a food festival in your city.
- You might make your decision by weighing up three factors:
  - Is the weather good?
  - Does your boyfriend or girlfriend want to accompany you?
  - Is the festival near public transit? (You don't own a car).

Artificial Neurons:Perceptrons

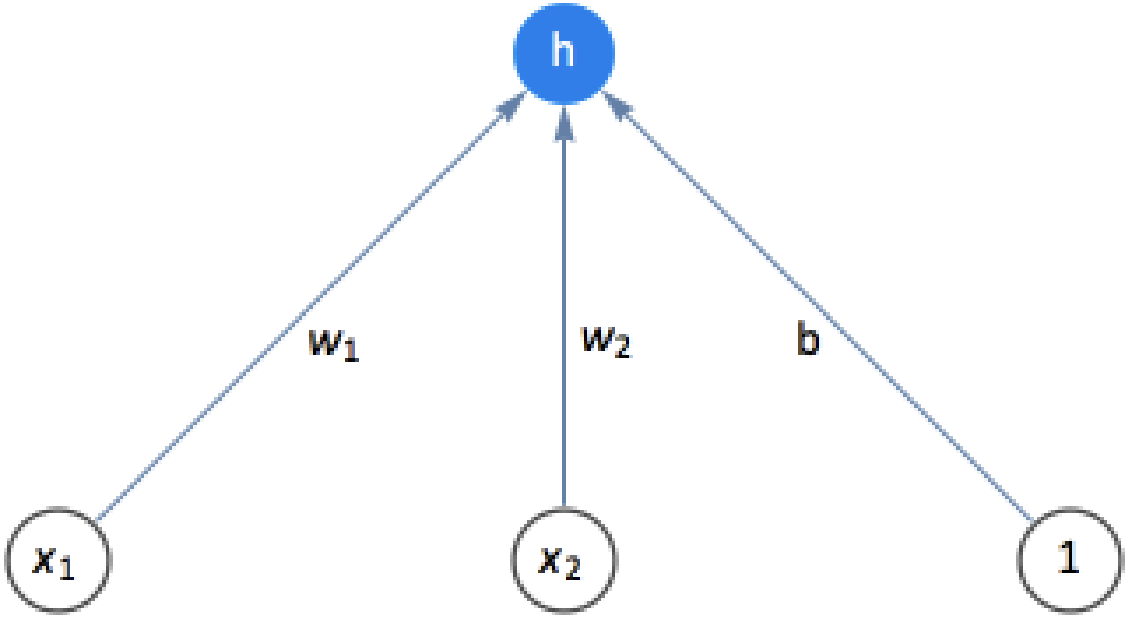


Artificial Neurons:Sigmoid Neuron



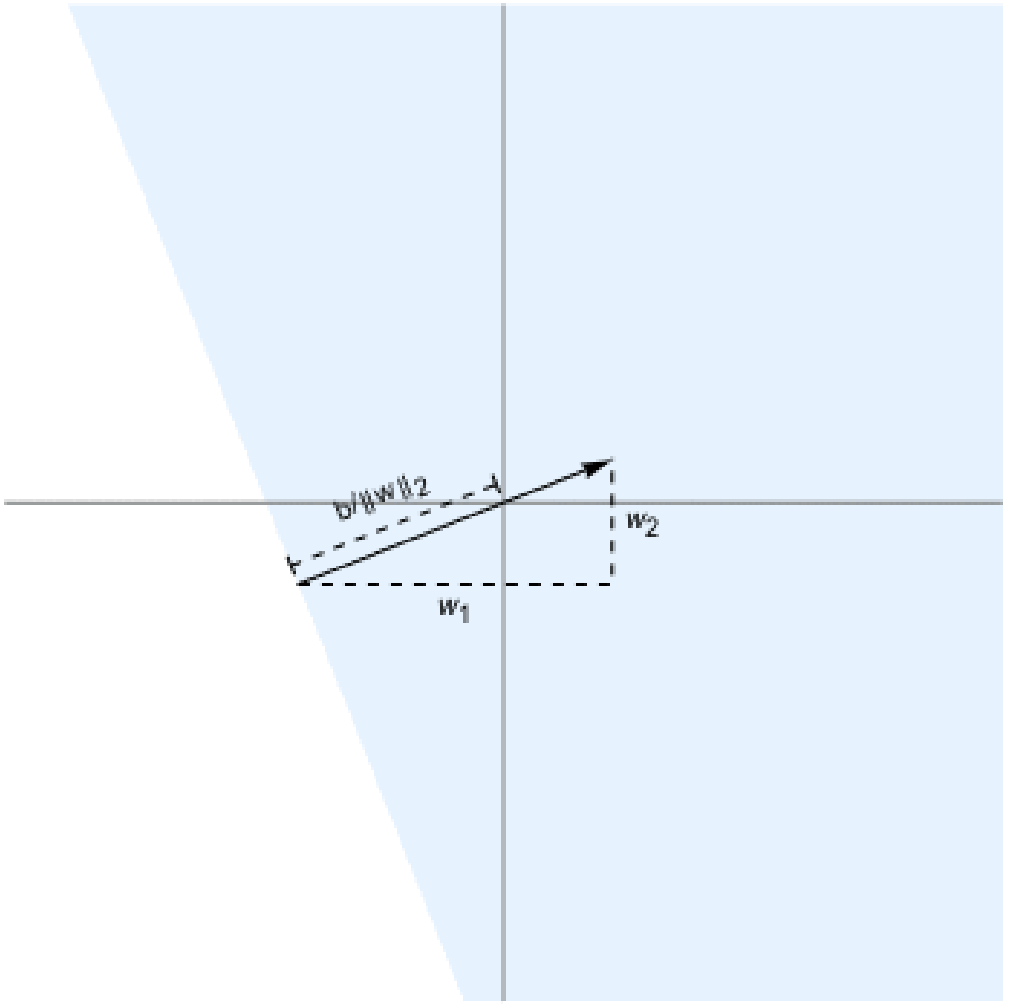


Artificial Neurons:Sigmoid Neuron

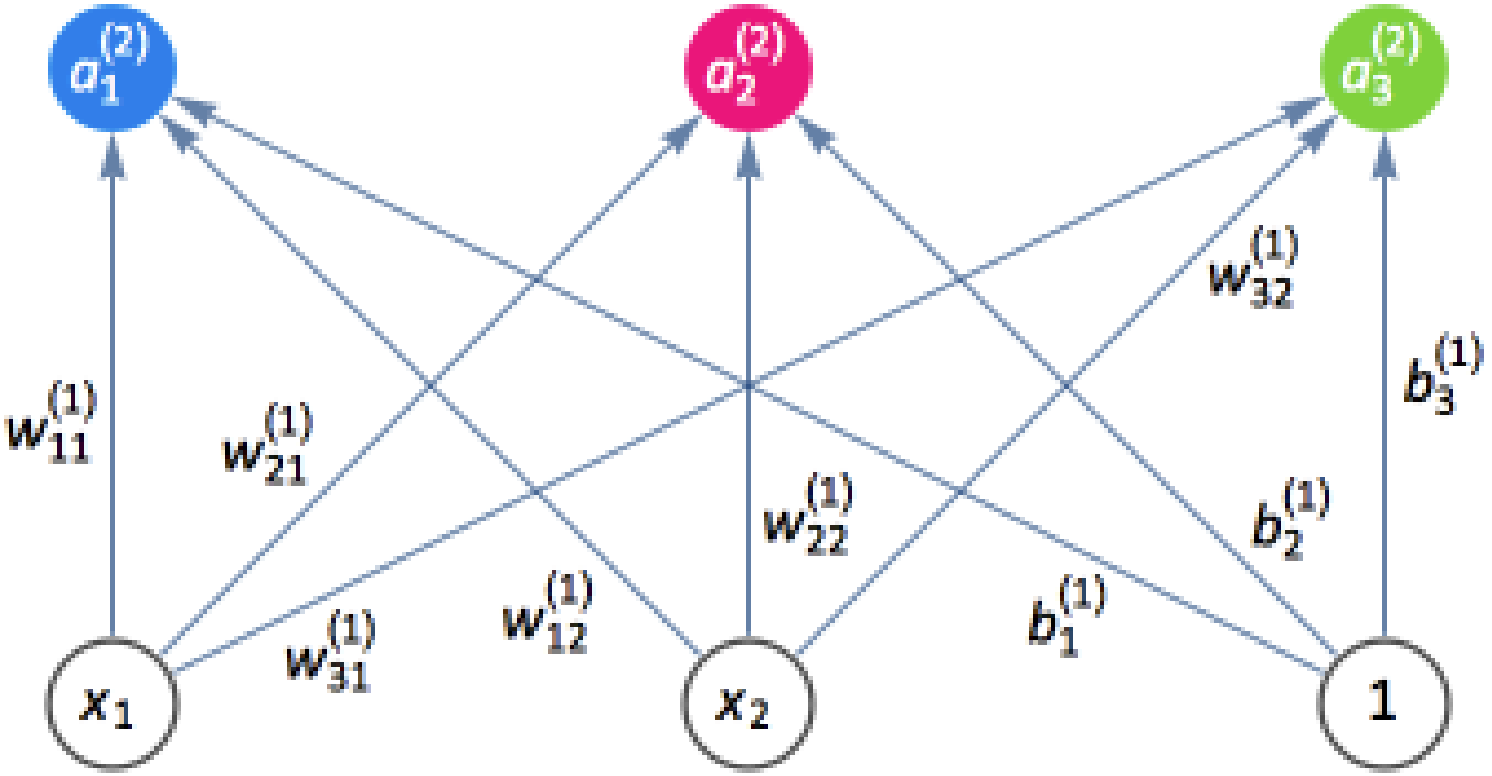


$$h = f\left(w_1 x_1 + w_2 x_2 + b\right)$$

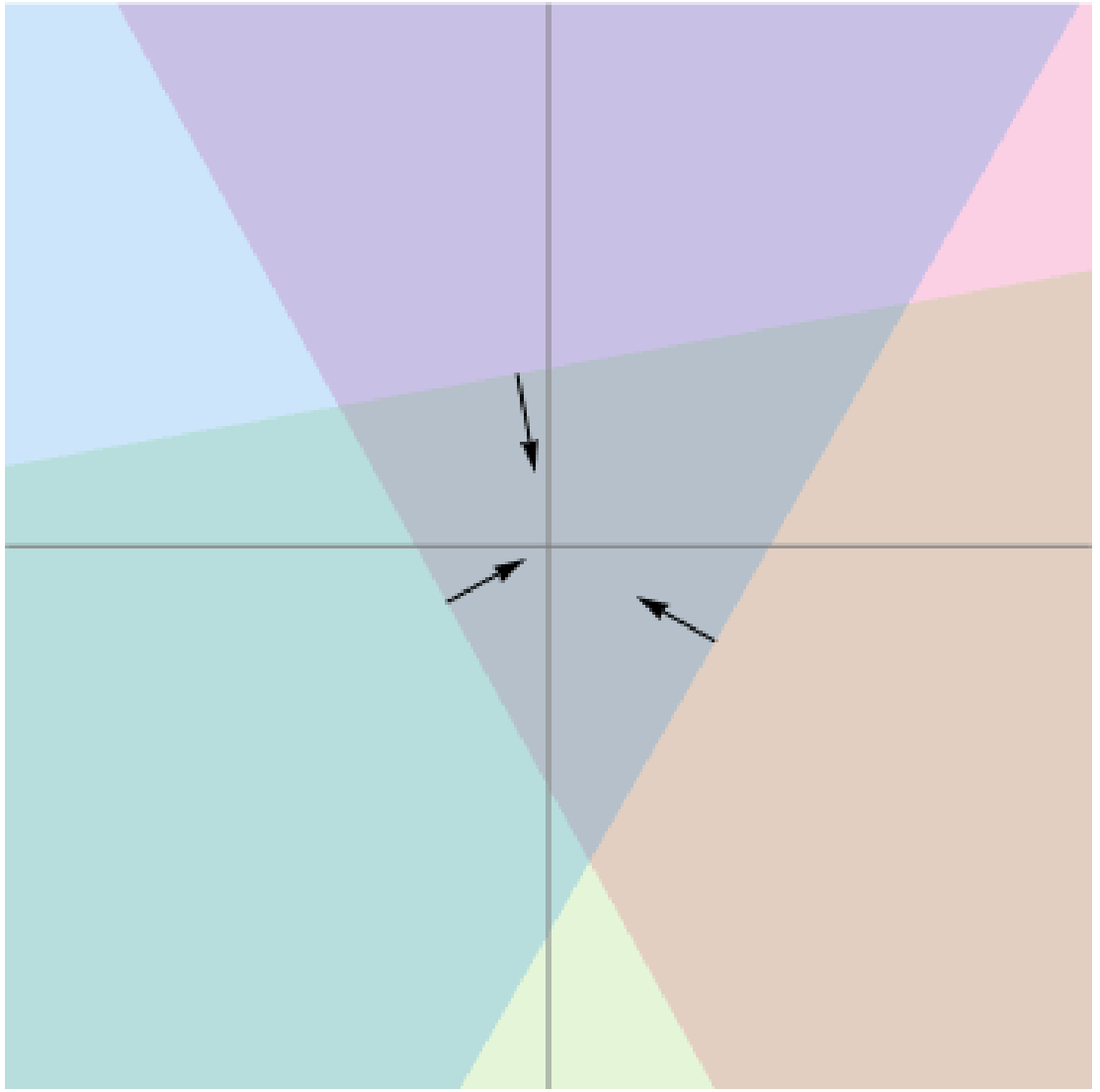
Artificial Neurons:Sigmoid Neuron



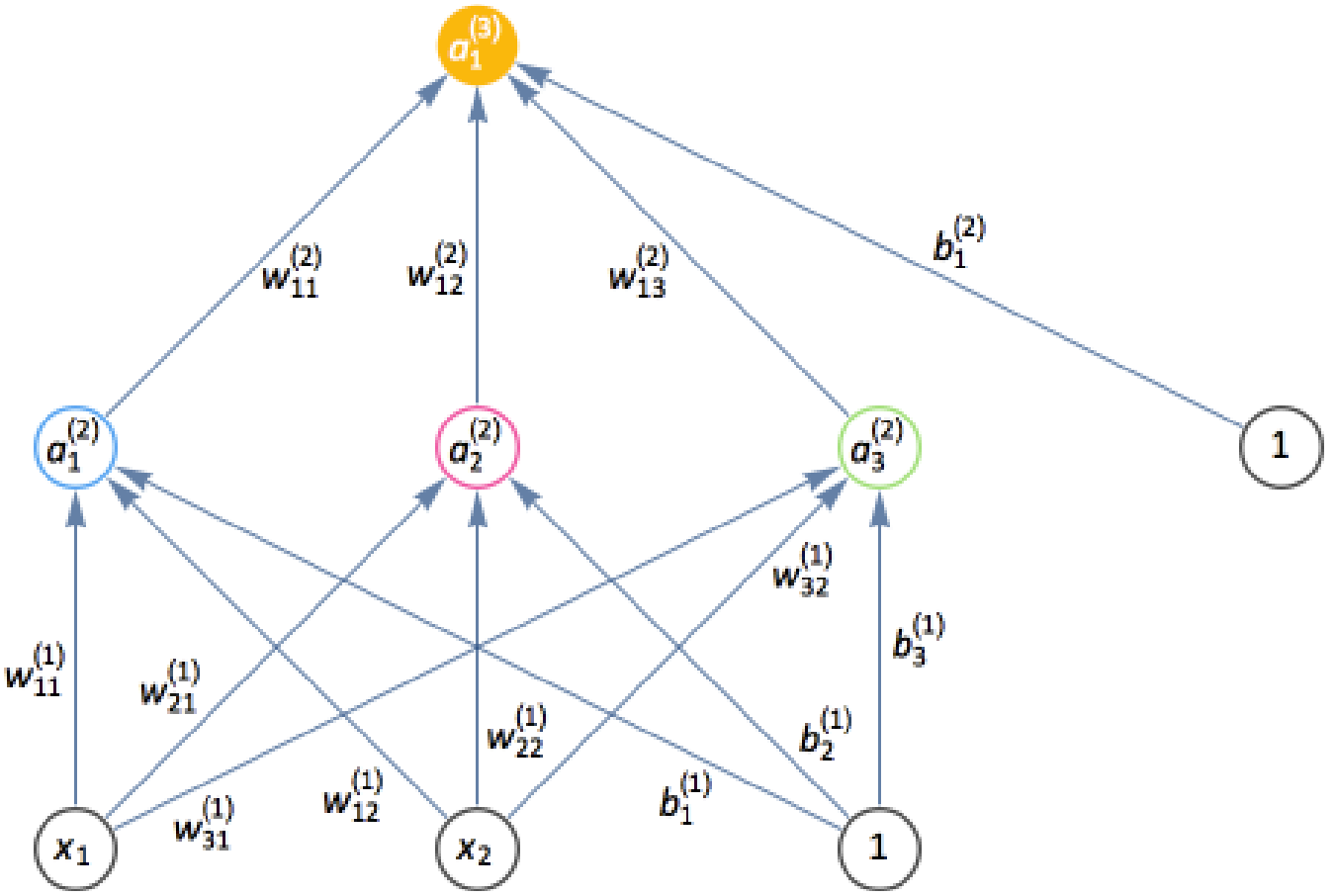
Deep Neural Network



Deep Neural Network

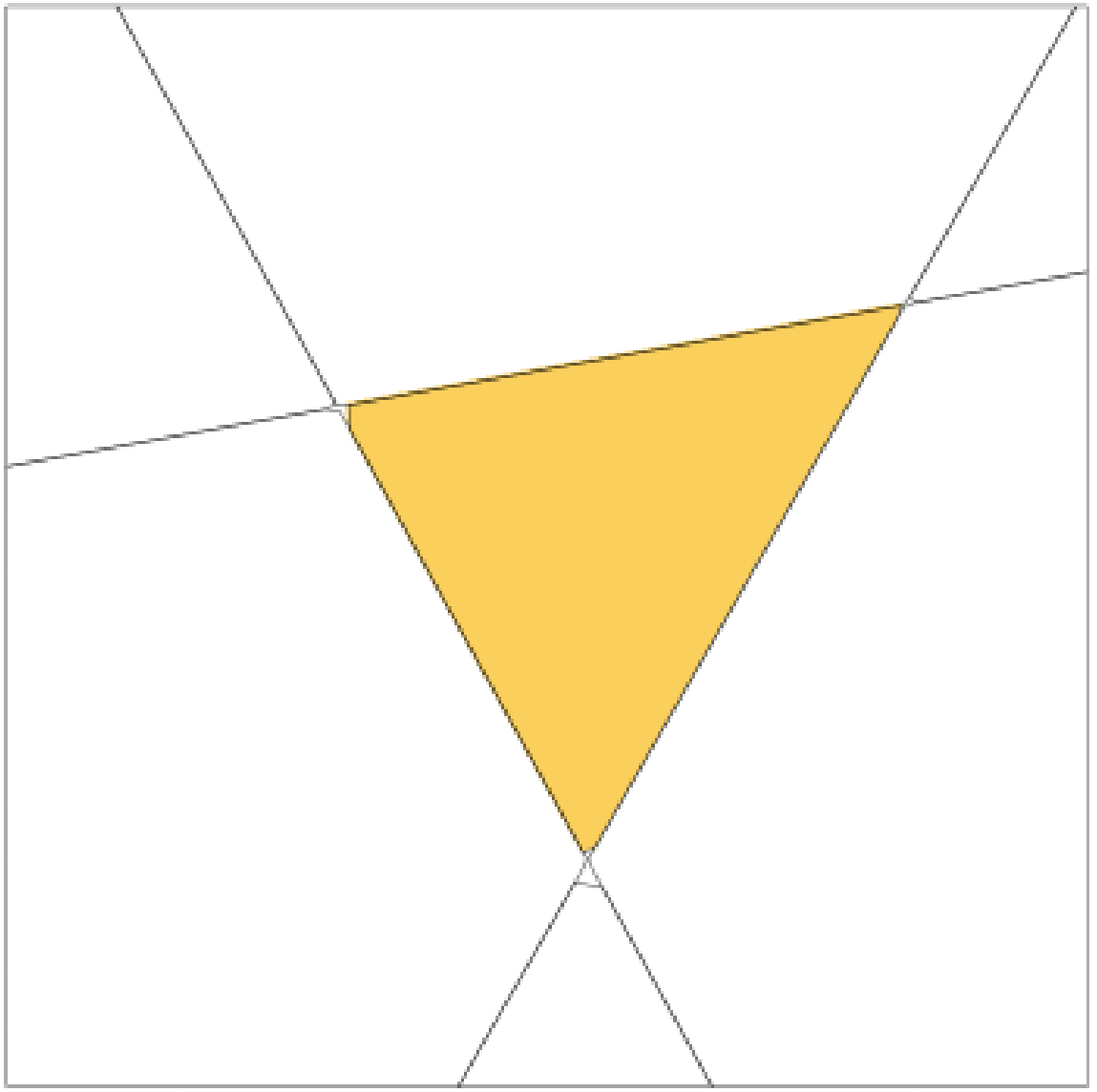


Deep Neural Network

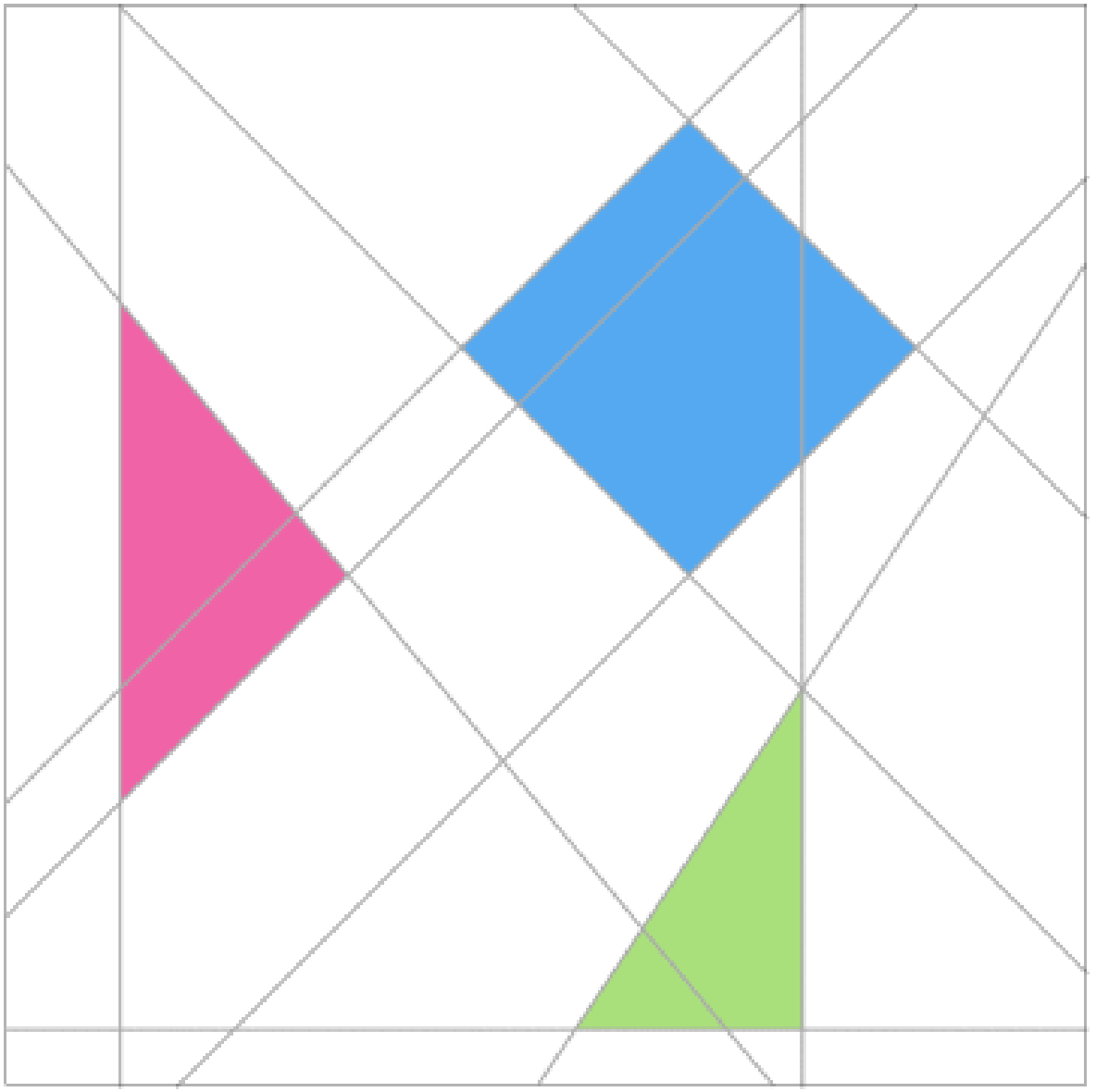


$$a_1^{(3)} = f\left(w_{11}^{(2)} a_1^{(2)} + w_{12}^{(2)} a_2^{(2)} + w_{13}^{(2)} a_3^{(2)} + b_1^{(2)}\right)$$

Deep Neural Network



Deep Neural Network



Deep Neural Network

Agenda:

Deep Learning: Introduction

Foundation:Linear Regression

Deep Learning: Going Deep:Intuition

Deep Learning: Going Deep:Forward pass

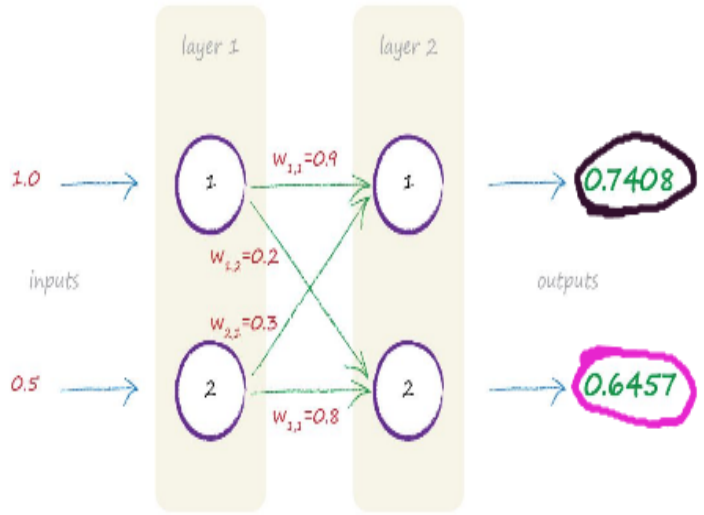
Deep Learning: Going Deep:Back Prop

Deep Learning: Architectures

Deep Learning: Fintechs

Deep Learning: GPU and TPU

Multi Layer Perceptron:Following the signals



x_{21} = (1st Node × link weight) + (2nd Node × link weight)

= (1.0 × 0.9) + (0.5 × 0.3)

= 0.9 + 0.15

= 1.05

$\sigma(x) = 1/(1+e^{-1.05}) = 1/(1+0.39) = .7408$

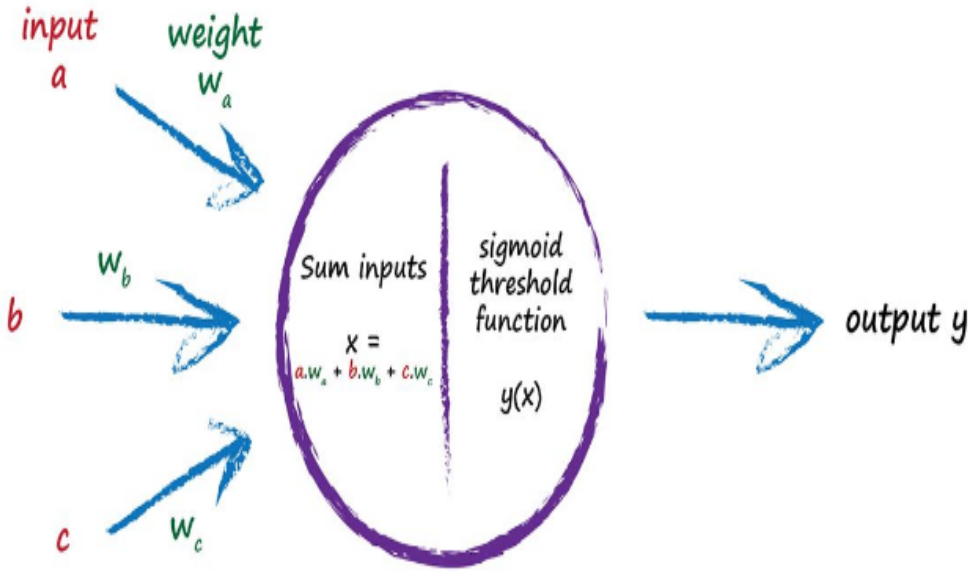
x_{22} = (1.0 × 0.2) + (0.6 × 0.8)

= (0.2 + 0.4)

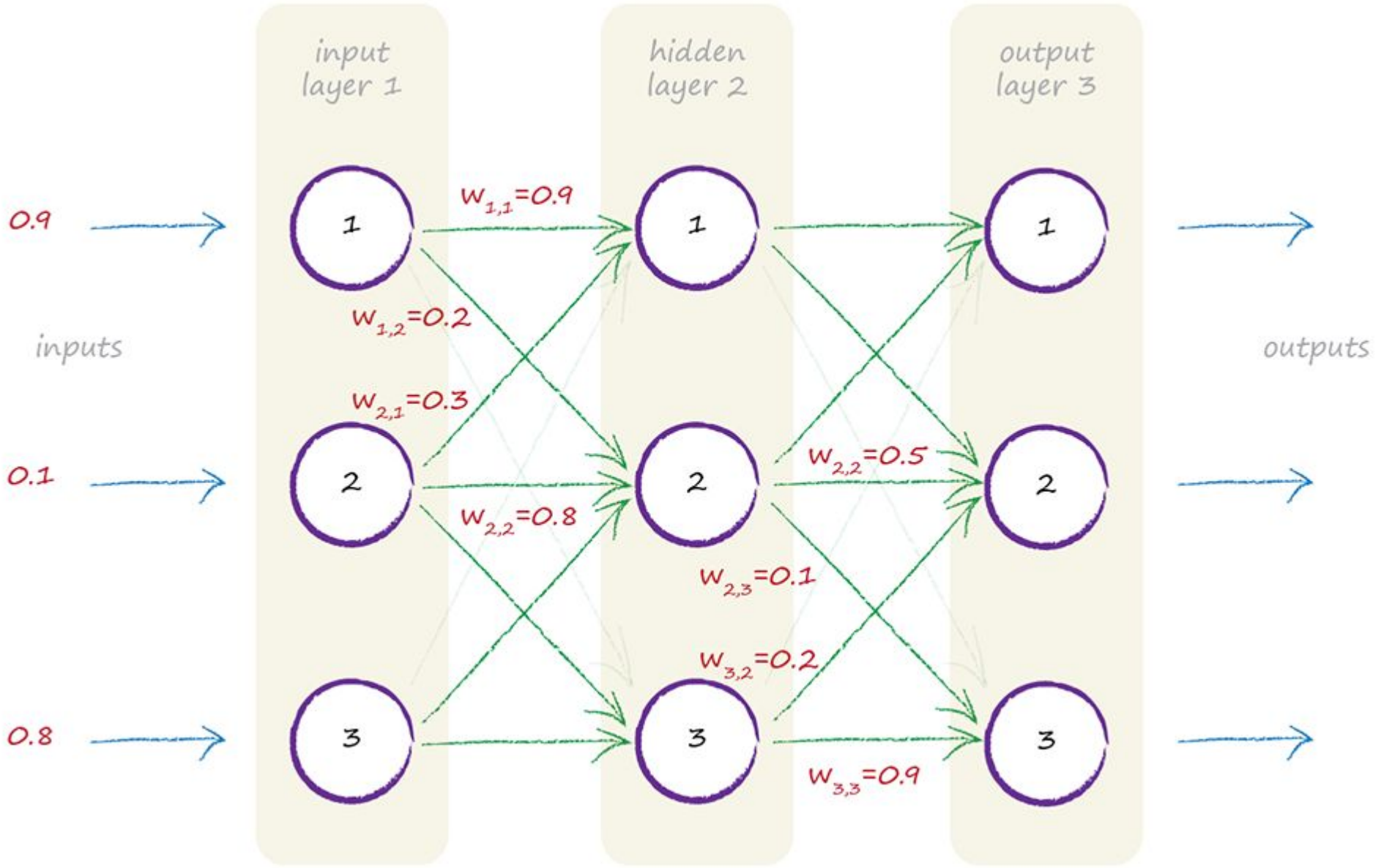
= 0.6

$\sigma_{(x)} = 1/1+e^{-0.6} = 1/(1+0.5488)$

= 0.6457



Multi Layer Perceptron: 3 Layer



Multi Layer Perceptron: 3 Layer Neural Network:Initialization

```
class neuralNetwork:


    # initialise the neural network
    def __init__(self, inputnodes, hiddennodes, outputnodes, learningrate,wih=None,who=None):
        # set number of nodes in each input, hidden, output layer
        self.inodes = inputnodes
        self.hnodes = hiddennodes
        self.onodes = outputnodes

        # link weight matrices, wih and who
        # weights inside the arrays are w_i_j, where link is from node i to node j in the next layer
        # w11 w21
        # w12 w22 etc
        if wih is None:
            self.wih = numpy.random.normal(0.0, pow(self.inodes, -0.5), (self.hnodes, self.inodes))
            self.who = numpy.random.normal(0.0, pow(self.hnodes, -0.5), (self.onodes, self.hnodes))
        else:
            self.wih=wih
            self.who=who

        # learning rate
        self.lr = learningrate

        # activation function is the sigmoid function
        self.activation_function = lambda x: scipy.special.expit(x)
        # reverse of sigmoid to work backwards
        self.inverse_activation_function = lambda x: scipy.special.logit(x)
        pass
```

- number of neurons in each layer
- random initialization of weights
- mean of 0.0 and float as type
- standard dev(σ) of $3^{-0.5}$ (0.577) from normal
- shape of weights
- Sigmoid function
- reverse

Multi Layer Perceptron: 3 Layer Neural Network:

```
def train(self, inputs_list, targets_list):
    # convert inputs list to 2d array
    inputs = numpy.array(inputs_list, ndmin=2).T
    targets = numpy.array(targets_list, ndmin=2).T

    # calculate signals into hidden layer
    hidden_inputs = numpy.dot(self.wih, inputs)
    # calculate the signals emerging from hidden layer
    hidden_outputs = self.activation_function(hidden_inputs)

    # calculate signals into final output layer
    final_inputs = numpy.dot(self.who, hidden_outputs)
    # calculate the signals emerging from final output layer
    final_outputs = self.activation_function(final_inputs)

    # output layer error is the (target - actual)
    output_errors = targets - final_outputs
    # hidden layer error is the output_errors, split by weights, recombined at hidden nodes
    hidden_errors = numpy.dot(self.who.T, output_errors)

    # update the weights for the links between the hidden and output layers
    self.who += self.lr * numpy.dot((output_errors * final_outputs * (1.0 - final_outputs)), numpy.transpose(hidden_outputs))

    # update the weights for the links between the input and hidden layers
    self.wih += self.lr * numpy.dot((hidden_errors * hidden_outputs * (1.0 - hidden_outputs)), numpy.transpose(inputs))

    pass
```

- first set of weights
- second set of weights

```
./DL/ANN/manualDnninternals.py
self.wih: [
[ 0.9  0.3  0.4]
 [ 0.2  0.8  0.2]
 [ 0.1  0.5  0.6]]
self.who: [
[ 0.3  0.7  0.5]
 [ 0.6  0.5  0.2]
 [ 0.8  0.1  0.9]]
(matrix([
[ 1.16],
 [ 0.42],
 [ 0.62]]),
matrix([[ 0.76133271],
[ 0.60348325],
 [ 0.65021855]]),
matrix([[ 0.97594736],
[ 0.88858496],
 [ 1.25461119]]),
matrix([[ 0.72630335],
[ 0.70859807],
 [ 0.77809706]]))
```

Agenda:

Deep Learning: Introduction

Foundation:Linear Regression

Deep Learning: Going Deep:Intuition

Deep Learning: Going Deep:Forward pass

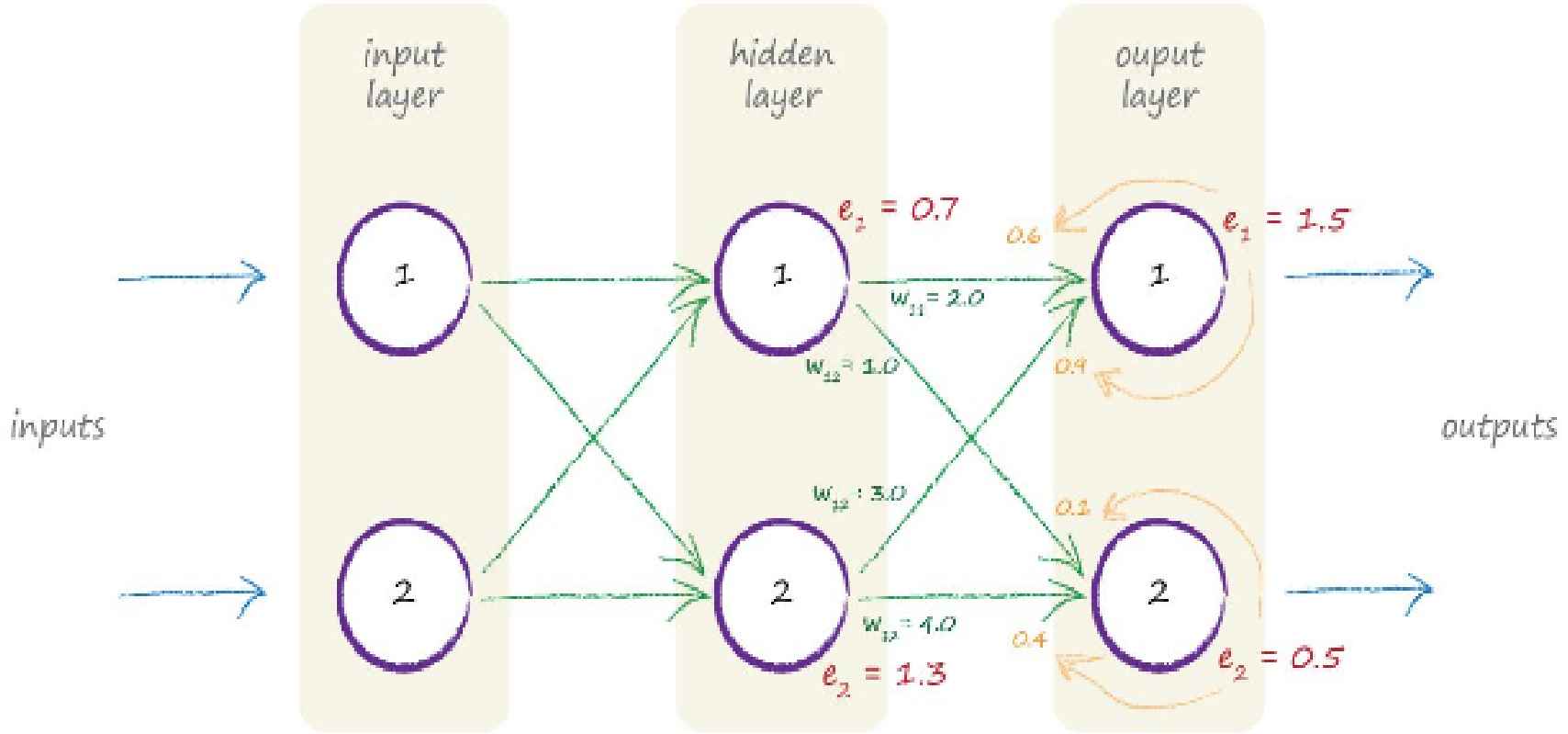
Deep Learning: Going Deep:Back Prop

Deep Learning: Architectures

Deep Learning: Fintechs

Deep Learning: GPU and TPU

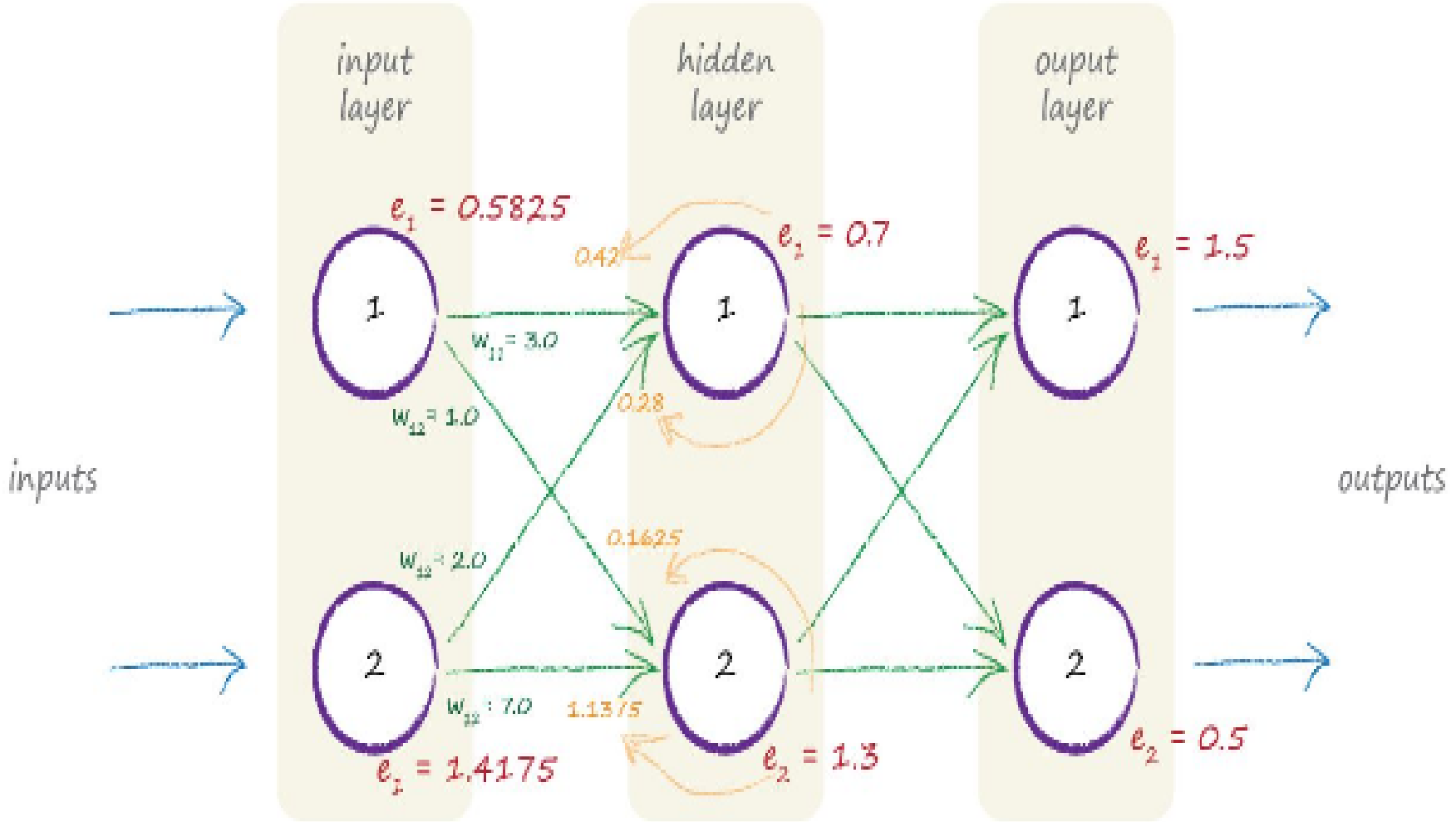
Multi Layer Perceptron: Back Propagation: Multiple Layers



$e_{hidden,1}$ = sum of split errors on links w_{11} and w_{12}

$$= e_{output,1} * \frac{w_{11}}{w_{11} + w_{21}} + e_{output,2} * \frac{w_{12}}{w_{12} + w_{22}}$$

Multi Layer Perceptron: Back Propagation: Multiple Layers



Multi Layer Perceptron:Gradient Descent

Error or the cost function:

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial}{\partial w_{jk}} \sum_n (t_n - o_n)^2$$

- Errors are not related for neurons at the output layer so we can get rid of the sum term simplifying it further.

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial}{\partial w_{jk}} (t_n - o_n)^2$$

Multi Layer Perceptron:Gradient Descent

1. $$\frac{\partial E}{\partial W_{JK}} = \frac{\partial E}{\partial O_K} \cdot \frac{\partial O_K}{\partial W_{JK}}$$

2. $$\frac{\partial E}{\partial O_k} = \frac{\partial}{\partial O_K} (t_k - O_k)^2$$

$$= \frac{\partial}{\partial O_K} (t_k^2 + O_k^2 - 2 t_k O_k)$$

$$= (0 + 2O_k - 2t_k)$$

$$= \boxed{-2(t_k - O_k)}$$

3. $$\frac{\partial O_k}{\partial W_{JK}} = \frac{\partial}{\partial W_{JK}} \text{sigmoid}(\Sigma W_{JK} \cdot O_J)$$

4. $$\frac{\partial}{\partial (x)} \text{sigmoid}(x) = \frac{\partial}{\partial x} \left(\frac{1}{1+e^{-x}}\right)$$

(substitute u for $(1+e^{-x})$

$$\frac{\partial}{\partial u}(1/u) = \frac{\partial}{\partial u} u^{-1}$$

$$= (-1) u^{-2}$$

$$\frac{\partial}{\partial x} \text{sigmoid}(x) = \frac{\partial \text{sigmoid}(x)}{\partial u} \cdot \frac{\partial u}{\partial x}$$

$$= -u^{-2} \cdot \frac{\partial (1+e^{-x})}{\partial x}$$

Multi Layer Perceptron:Gradient Descent

$$= -u^{-2} \cdot \frac{\partial\, 1+e^{-x}}{\partial x}$$

$$= -u^{-2} \cdot (0 + -e^{-x}) = u^{-2} \cdot e^{-x}$$

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

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

$$= \frac{1}{1+e^{-x}} \left(1 - \frac{1}{1+e^{-x}}\right)$$

$$= \text{sigmoid}(x)(1 - \text{sigmoid}(x))$$

Multi Layer Perceptron:Gradient Descent

3. $\frac{\partial}{\partial W_{jk} O_j} \text{sigmoid}(\Sigma W_{jk} \cdot O_j) = \text{sigmoid}(\Sigma W_{jk} \cdot O_j) \cdot (1 - \text{sigmoid}(\Sigma W_{jk} \cdot O_j) \cdot \frac{\partial}{\partial W_{jk}} (\Sigma W_{jk} \cdot O_j)$

$= \text{sigmoid}(\Sigma W_{jk} \cdot O_j) \cdot (1 - \text{sigmoid}(\Sigma W_{jk} \cdot O_j) \cdot O_j$

1. $\frac{\partial E}{\partial W_{jk}} = \frac{\partial E}{\partial O_k} \cdot \frac{\partial O_k}{\partial W_{jk}}$

$= -2(t_k - O_k) \cdot \text{sigmoid}(\Sigma W_{jk} \cdot O_j) \cdot (1 - \text{sigmoid}(\Sigma W_{jk} \cdot O_j) \cdot O_j$

- Eq-1
- Eq-2

- as a simplification the constant 2 can be gotten rid of
- denoted as e_k where k signifies the layer
- these subscripts change depending on the layer as the weights have to be updated for every neuron on every layer.

Multi Layer Perceptron:Gradient Descent:Back Propagation

```
def train(self, inputs_list, targets_list):
    # convert inputs list to 2d array
    inputs = numpy.array(inputs_list, ndmin=2).T
    targets = numpy.array(targets_list, ndmin=2).T

    # calculate signals into hidden layer
    hidden_inputs = numpy.dot(self.wih, inputs)
    # calculate the signals emerging from hidden layer
    hidden_outputs = self.activation_function(hidden_inputs)

    # calculate signals into final output layer
    final_inputs = numpy.dot(self.who, hidden_outputs)
    # calculate the signals emerging from final output layer
    final_outputs = self.activation_function(final_inputs)

    # output layer error is the (target - actual)
    output_errors = targets - final_outputs
    # hidden layer error is the output_errors, split by weights, recombined at hidden nodes
    hidden_errors = numpy.dot(self.who.T, output_errors)

    # update the weights for the links between the hidden and output layers
    self.who += self.lr * numpy.dot((output_errors * final_outputs * (1.0 - final_outputs)), numpy.transpose(hidden_outputs))

    # update the weights for the links between the input and hidden layers
    self.wih += self.lr * numpy.dot((hidden_errors * hidden_outputs * (1.0 - hidden_outputs)), numpy.transpose(inputs))

    pass
```

$$new\ w_{jk} = old\ w_{jk} - \alpha \cdot \frac{\partial E}{\partial w_{jk}}$$

$$\Delta w_{jk} = \alpha * E_k * sigmoid(O_k) * (1 - sigmoid(O_k)) \cdot O_j^T$$

- Notice the transpose

update the weights for the links between the hidden and output layers
self.who += self.lr * numpy.dot((output_errors * final_outputs * (1.0 - final_outputs)), numpy.transpose(hidden_outputs))

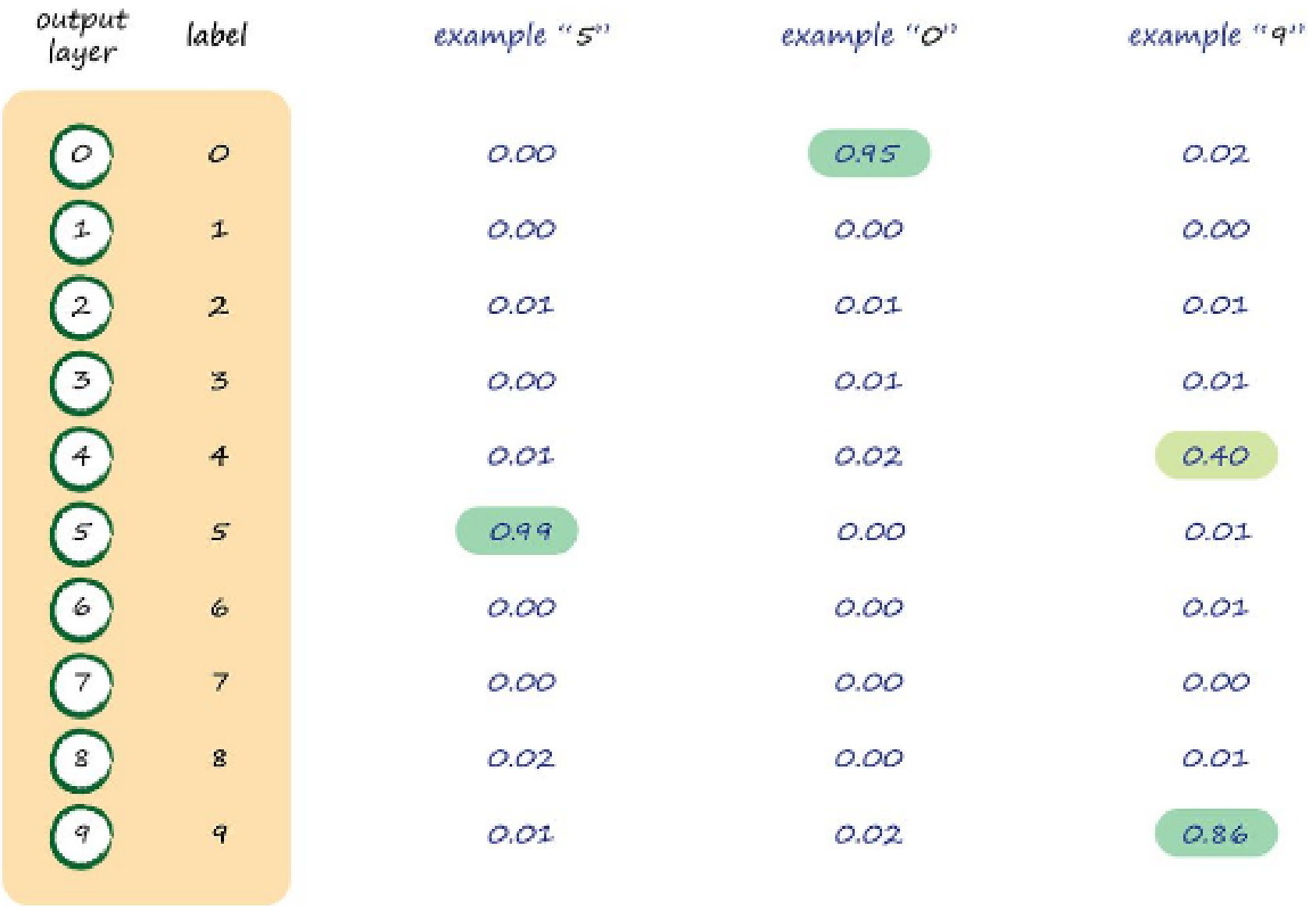
update the weights for the links between the input and hidden layers
self.wih += self.lr * numpy.dot((hidden_errors * hidden_outputs * (1.0 - hidden_outputs)), numpy.transpose(inputs))

Multi Layer Perceptron:Preparing Data:Input

In [19]:
```
# scale input to range 0.01 to 1.00
scaled_input = (numpy.asfarray(all_values[1:]) / 255.0 * 0.99) + 0.01
print(scaled_input)
```

```
[ 0.01       0.01       0.01       0.01       0.01       0.01       0.01
  0.01       0.01       0.01       0.01       0.01       0.01       0.01
  0.01       0.01       0.01       0.01       0.01       0.01       0.01
  0.01       0.01       0.01       0.01       0.01       0.01       0.01
  0.01       0.01       0.01       0.01       0.01       0.01       0.01
  0.01       0.01       0.01       0.01       0.01       0.01       0.01
  0.01       0.01       0.01       0.01       0.01       0.01       0.01
  0.01       0.01       0.01       0.01       0.01       0.01       0.01
  0.01       0.01       0.01       0.01       0.01       0.01       0.01
  0.01       0.01       0.01       0.01       0.01       0.01       0.01
  0.01       0.01       0.01       0.01       0.01       0.01       0.01
  0.01       0.01       0.01       0.01       0.01       0.01       0.01
  0.01       0.01       0.01       0.01       0.01       0.01       0.01
  0.01       0.01       0.01       0.01       0.01       0.01       0.01
  0.01       0.01       0.01       0.01       0.01       0.01       0.01
  0.01       0.01       0.01       0.01       0.01       0.01       0.01
  0.01       0.208      0.62729412 0.99223529 0.62729412 0.20411765
  0.01       0.01       0.01       0.01       0.01       0.01       0.01
  0.01       0.01       0.01       0.01       0.01       0.01       0.01
  0.01       0.01       0.01       0.01       0.01       0.01       0.01
  0.01       0.19635294 0.934      0.98835294 0.98835294 0.98835294
  0.93011765 0.01       0.01       0.01       0.01       0.01       0.01
  0.01       0.01       0.01       0.01       0.01       0.01       0.01
  0.01       0.01       0.01       0.01       0.01       0.01       0.01
  0.01       0.21964706 0.89129412 0.99223529 0.98835294 0.93788235
  0.91458824 0.98835294 0.23129412 0.03329412 0.01       0.01       0.01
```

Multi Layer Perceptron:Preparing Data:Output



Multi Layer Perceptron:Preparing Data:MNIST:Test

```
# test the neural network
# scorecard for how well the network performs, initially empty
scorecard = []

# go through all the records in the test data set
for record in test_data_list:
    # split the record by the ',' commas
    all_values = record.split(',')
    # correct answer is first value
    correct_label = int(all_values[0])
    # scale and shift the inputs
    inputs = (numpy.asfarray(all_values[1:]) / 255.0 * 0.99) + 0.01
    # query the network
    outputs = n.query(inputs)
    # the index of the highest value corresponds to the label
    label = numpy.argmax(outputs)
    # append correct or incorrect to list
    if (label == correct_label):
        # network's answer matches correct answer, add 1 to scorecard
        scorecard.append(1)
    else:
        # network's answer doesn't match correct answer, add 0 to scorecard
        scorecard.append(0)
        pass

    pass


# calculate the performance score, the fraction of correct answers
scorecard_array = numpy.asarray(scorecard)
print ("performance = ", scorecard_array.sum() / scorecard_array.size)
```

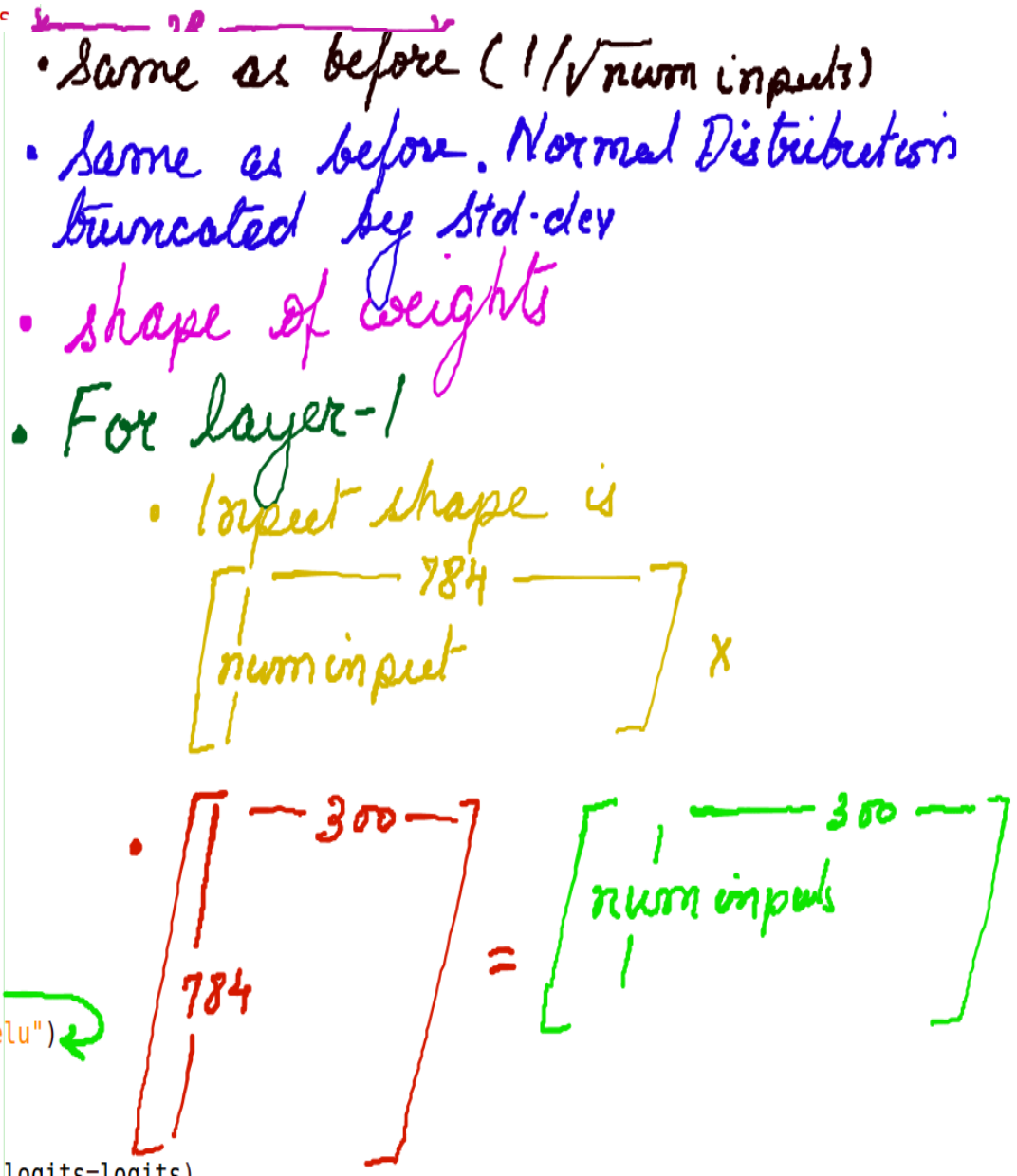
Multi Layer Perceptron:Preparing Data:MNIST:Test

```
(ml_home) mohit@nomind:~/Work/ArtificialIntelligence$ ./DL/ANN/manualdnn.py
input/mnist/mnist_train_100.csv
performance =  0.6
(ml_home) mohit@nomind:~/Work/ArtificialIntelligence$
(ml_home) mohit@nomind:~/Work/ArtificialIntelligence$
(ml_home) mohit@nomind:~/Work/ArtificialIntelligence$ ./DL/ANN/manualdnn.py -r input/mnist/mnist_train.csv -t input/mnist/mnist_test.csv
input/mnist/mnist_train.csv
performance =  0.9722
```

- Trained on only 100 samples.
- Trained on complete set.
- Mind you this is hand rolled vanilla neural net with no ensemble.

Multi Layer Perceptron:Back Query

```
# run the network backwards, given a label, see what image it produces
#Construction Phase
def neuron_layer(X, n_neurons, name, activation=None):
    with tf.name_scope(name):
        n_inputs = int(X.get_shape()[1])
        stddev = 1 / np.sqrt(n_inputs)
        init = tf.truncated_normal((n_inputs, n_neurons), stddev=stddev)
        W = tf.Variable(init, name="weights")
        print("Shape of input:",X.get_shape())
        print("Shape of Weights:",W.get_shape())
        b = tf.Variable(tf.zeros([n_neurons]), name="biases")
        Z = tf.matmul(X, W) + b
        if activation=="relu":
            return tf.nn.relu(Z)
        else:
            return Z

tf.reset_default_graph()
n_inputs = 28*28  # MNIST
n_hidden1 = 300
n_hidden2 = 100
n_outputs = 10
learning_rate = 0.01

X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
y = tf.placeholder(tf.int64, shape=(None), name="y")

with tf.name_scope("dnn"):
    hidden1 = neuron_layer(X, n_hidden1, "hidden1", activation="relu")
    hidden2 = neuron_layer(hidden1, n_hidden2, "hidden2", activation="relu")
    logits = neuron_layer(hidden2, n_outputs, "output")

with tf.name_scope("loss"):
    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y, logits=logits)
    loss = tf.reduce_mean(xentropy, name="loss")

with tf.name_scope("train"):
    optimizer = tf.train.GradientDescentOptimizer(learning_rate)
    training_op = optimizer.minimize(loss)
```



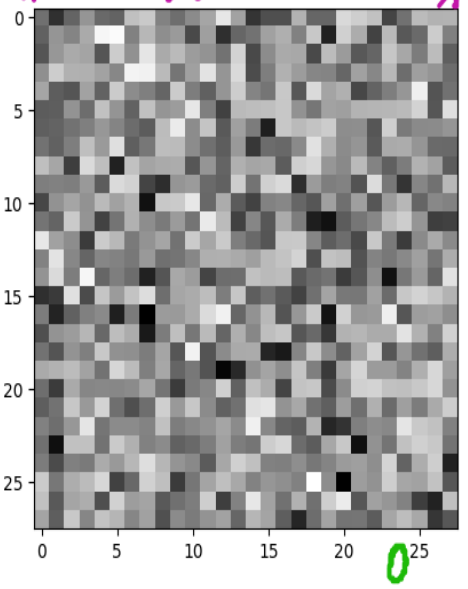
Multi Layer Perceptron:Back Query

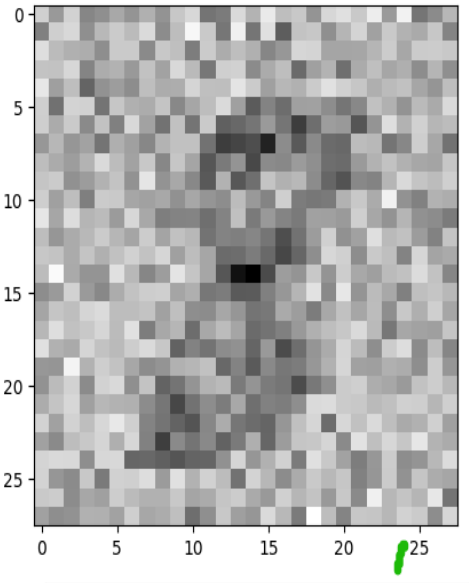
```
# run the network backwards, given a label, see what image it produces
# label to test
label = 8
# create the output signals for this label
targets = numpy.zeros(output_nodes) + 0.01
# all_values[0] is the target label for this record
targets[label] = 0.99
print(targets)

# get image data
image_data = n.backquery(targets)

# plot image data
matplotlib.pyplot.imshow(image_data.reshape(28,28), cmap='Greys', int
matplotlib.pyplot.show()
```

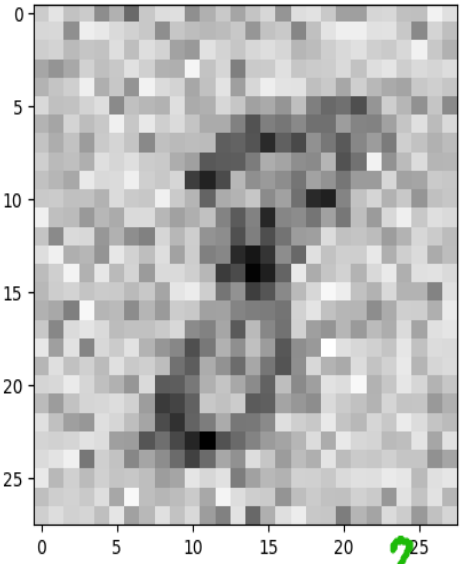
• Why?

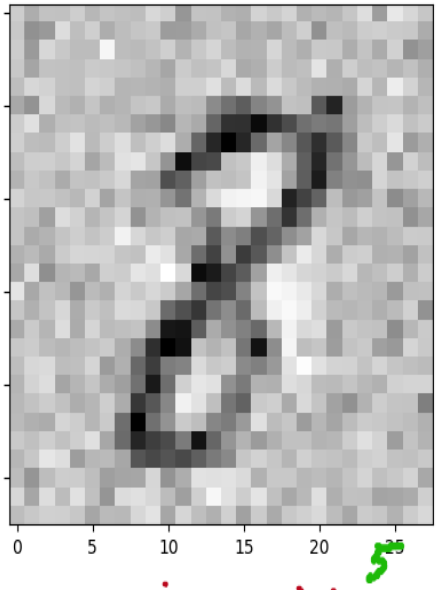
28

28









```
def backquery(self, targets_list):
    # transpose the targets list to a vertical array
    final_outputs = numpy.array(targets_list, ndmin=2).T

    # calculate the signal into the final output layer
    final_inputs = self.inverse_activation_function(final_outputs)

    # calculate the signal out of the hidden layer
    hidden_outputs = numpy.dot(self.who.T, final_inputs)
    # scale them back to 0.01 to .99
    hidden_outputs -= numpy.min(hidden_outputs)
    hidden_outputs /= numpy.max(hidden_outputs)
    hidden_outputs *= 0.98
    hidden_outputs += 0.01

    # calculate the signal into the hideen layer
    hidden_inputs = self.inverse_activation_function(hidden_outputs)

    # calculate the signal out of the input layer
    inputs = numpy.dot(self.wih.T, hidden_inputs)
    # scale them back to 0.01 to .99
    inputs -= numpy.min(inputs)
    inputs /= numpy.max(inputs)
    inputs *= 0.98
    inputs += 0.01

    return inputs
```

0 1 2 5

• Working backwards with reverse sigmoid function also called the logit.

• learned weights progressively.

Agenda:

Deep Learning: Introduction

Foundation:Linear Regression

Deep Learning: Going Deep:Intuition

Deep Learning: Going Deep:Forward pass

Deep Learning: Going Deep:Back Prop

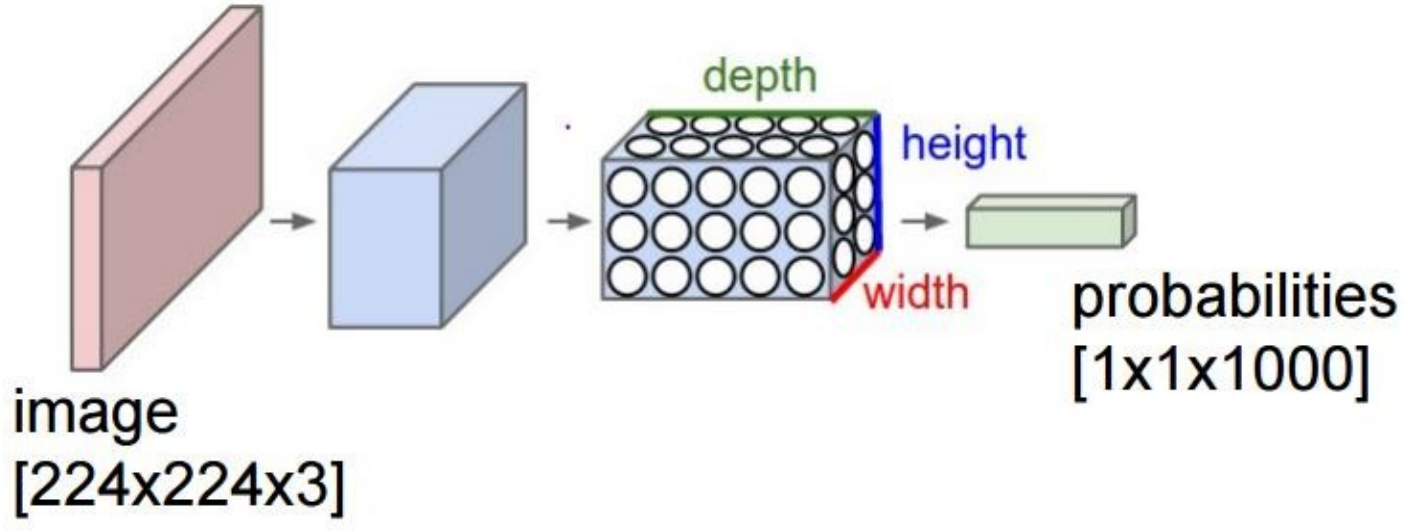
Deep Learning: Architectures

Deep Learning: Fintechs

Deep Learning: GPU and TPU

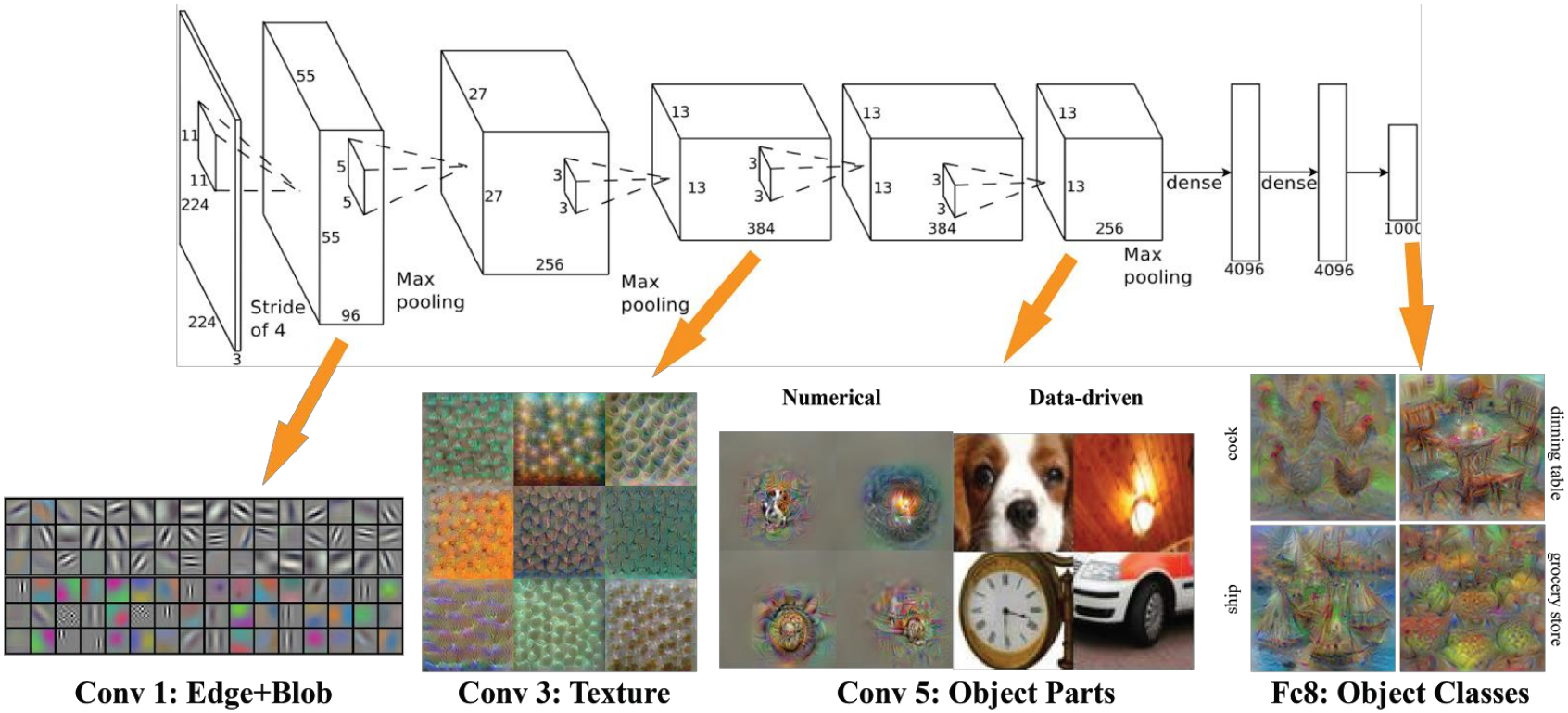
Architectures:CNN

Convolutional Neural Networks learn a complex representation of visual data using vast amounts of data. They are inspired by the human visual system and learn multiple layers of transformations, which are applied on top of each other to extract a progressively more sophisticated representation of the input.



Every layer of a CNN takes a 3D volume of numbers and outputs a 3D volume of numbers. E.g. Image is a 224*224*3 (RGB) cube and will be transformed to 1*1000 vector of probabilities.

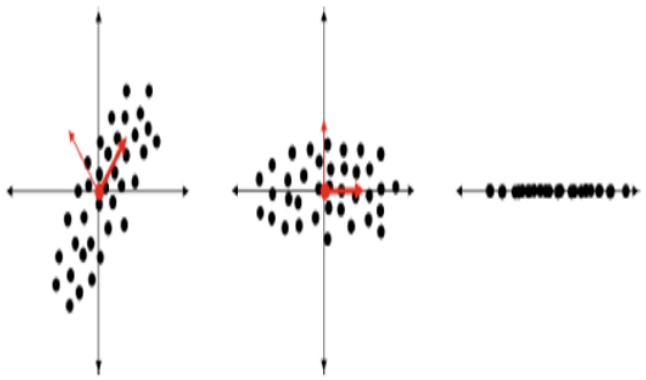
Architectures:CNN



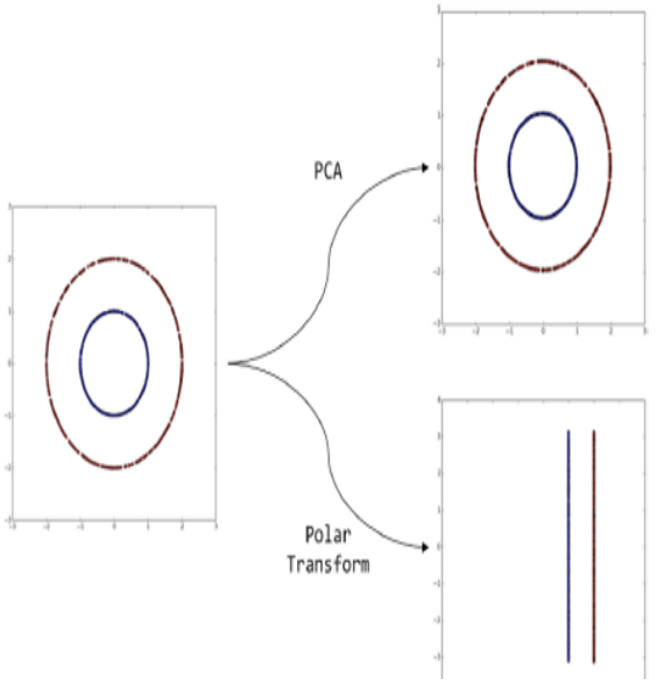
Convolution layer is a feature detector that automagically learns to filter out not needed information from an input by using convolution kernel.

Pooling layers compute the max or average value of a particular feature over a region of the input data *(downsizing of input images).* Also helps to detect objects in some unusual places and reduces memory size.

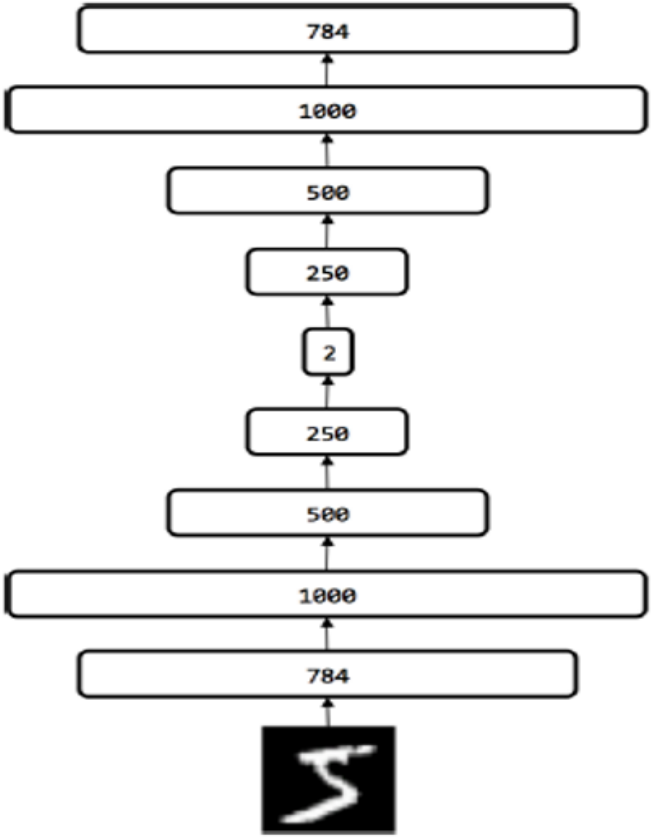
Architectures:AE

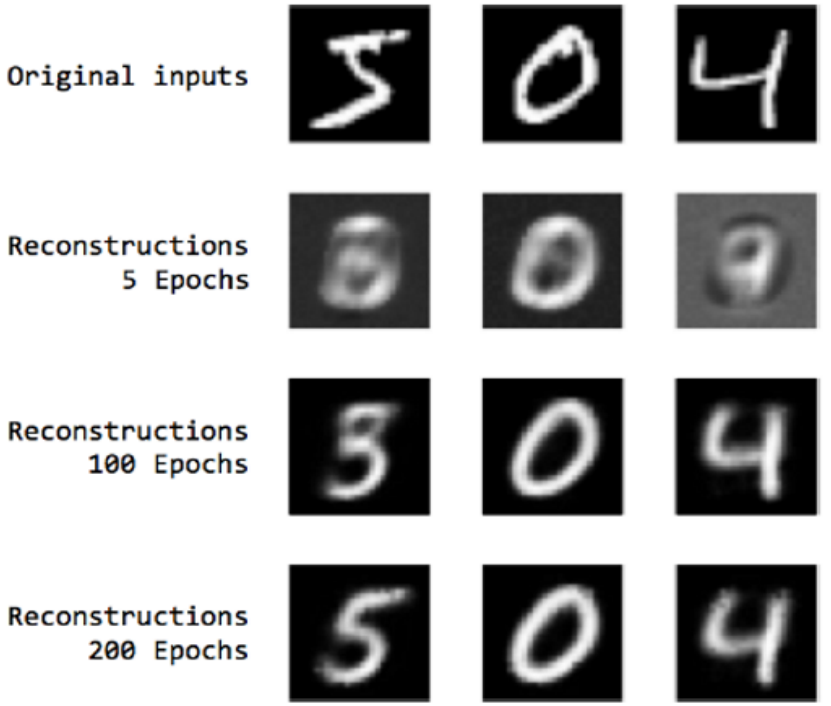


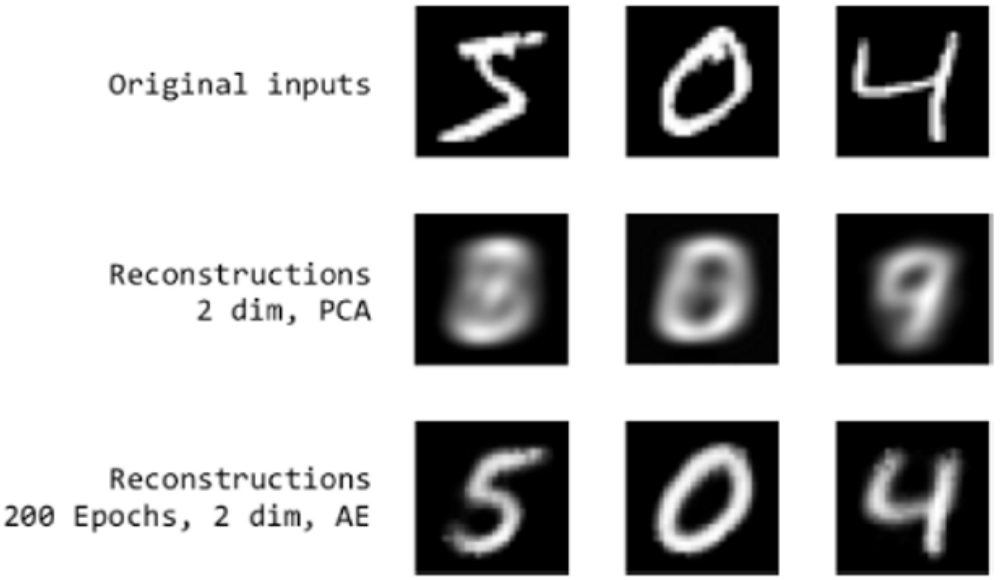
. While PCA is good at finding dominant dimensions, it is not so effective when the dominant-dimension is not linear.



Architectures:AE

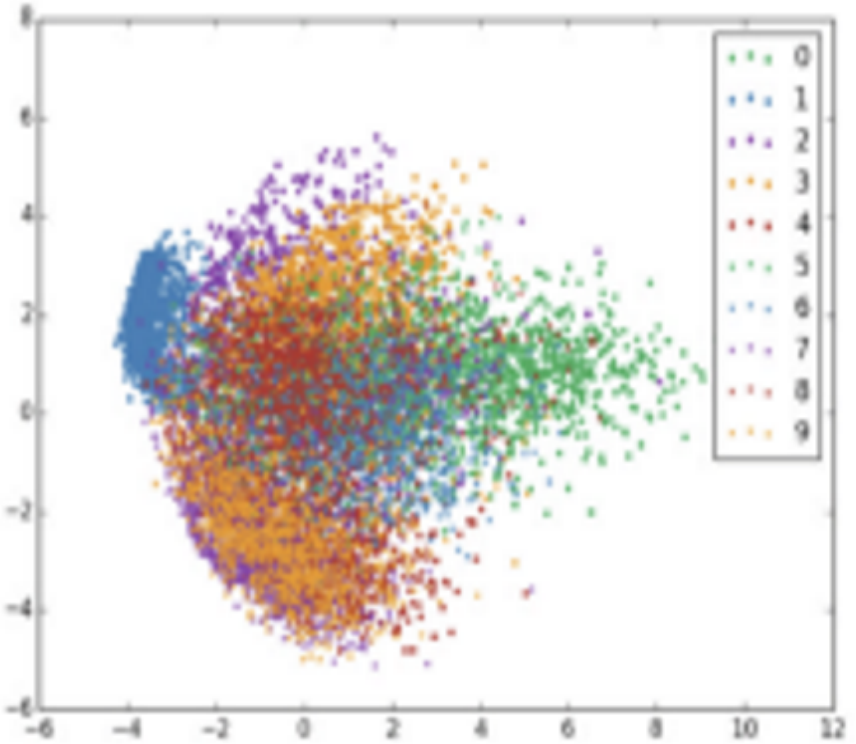




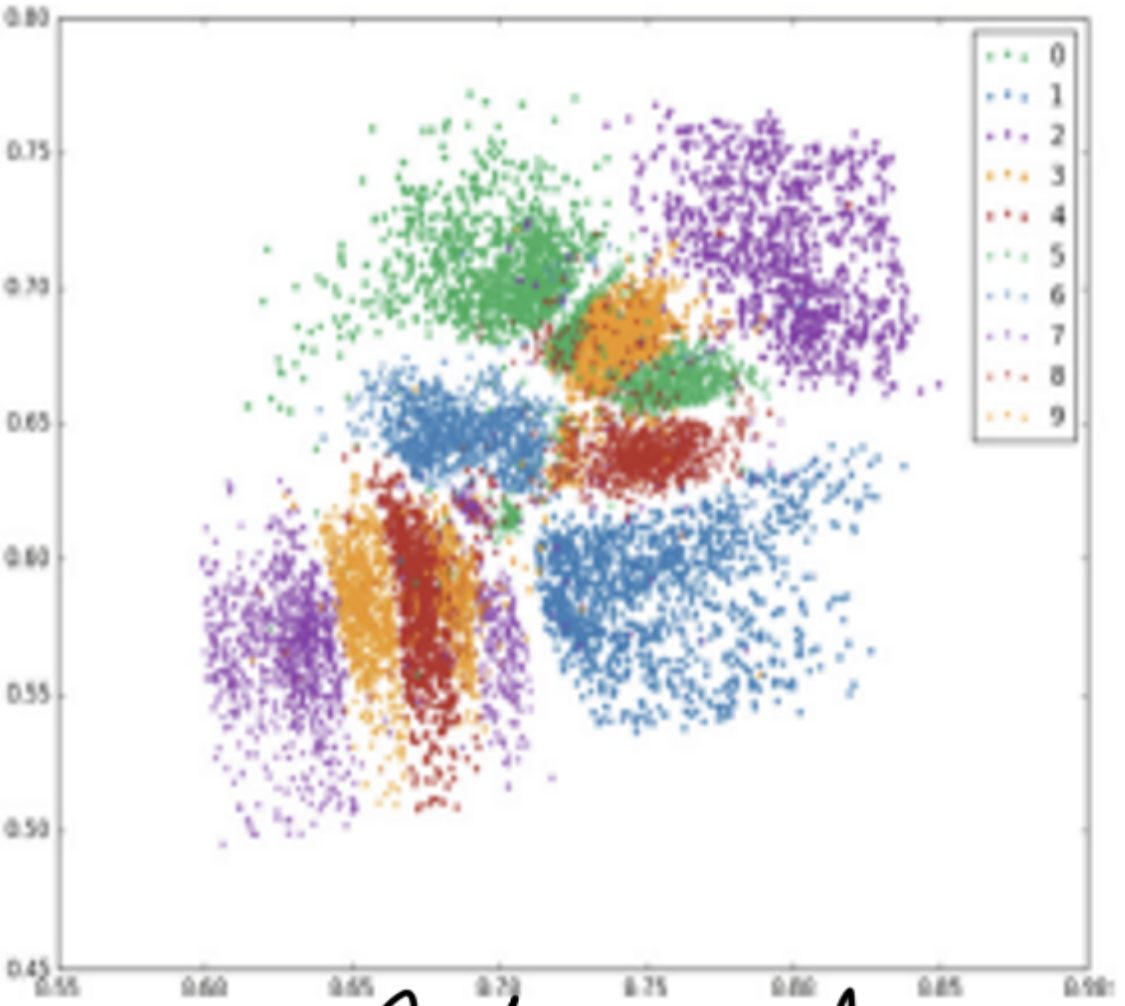


. There is only one winner here.

Architectures:AE



PCA



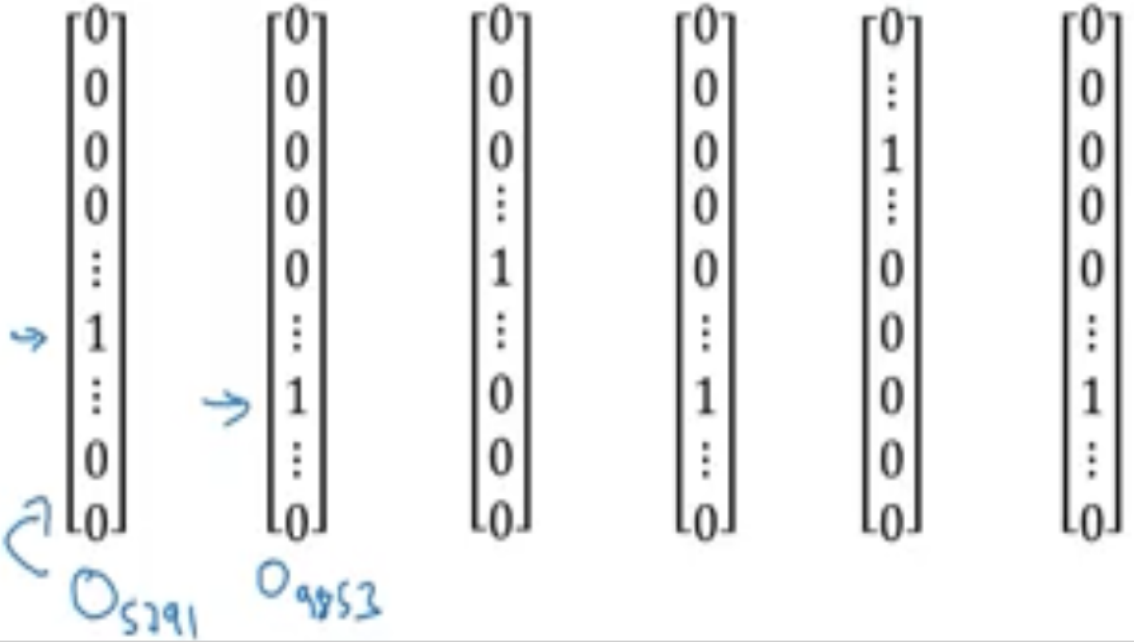
Autoencoder

- Visualization of 2 dimension encoding done by PCA and autoencoder.

Architectures:AE:Usecase

1-hot representation

Man (5391)	Woman (9853)	King (4914)	Queen (7157)	Apple (456)	Orange (6257)

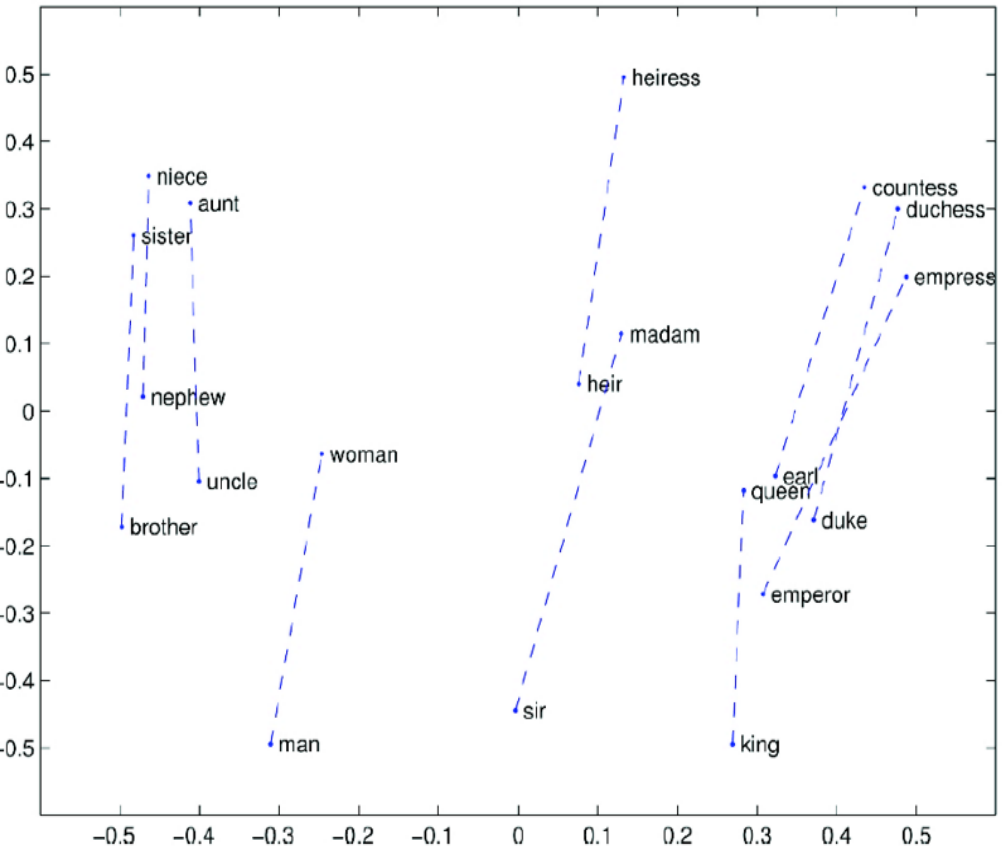


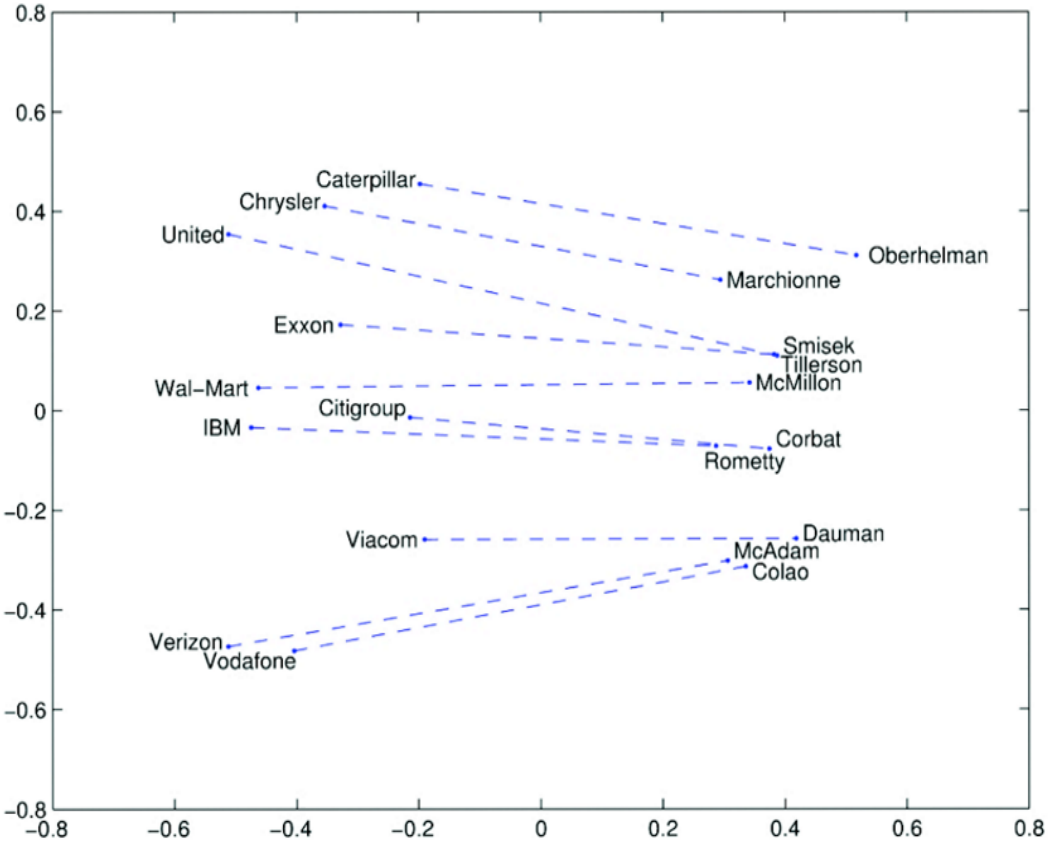
O_{5391} O_{9853}

I want a glass of orange juice.

I want a glass of apple ?.

Architectures:AE:Usecase





- If these are considered as dimensions of data then they can be learnt through auto encoders.
- Many are available off the shelf

- Some times however you train them for a specific vocabulary.

Architectures:RNN:Applications

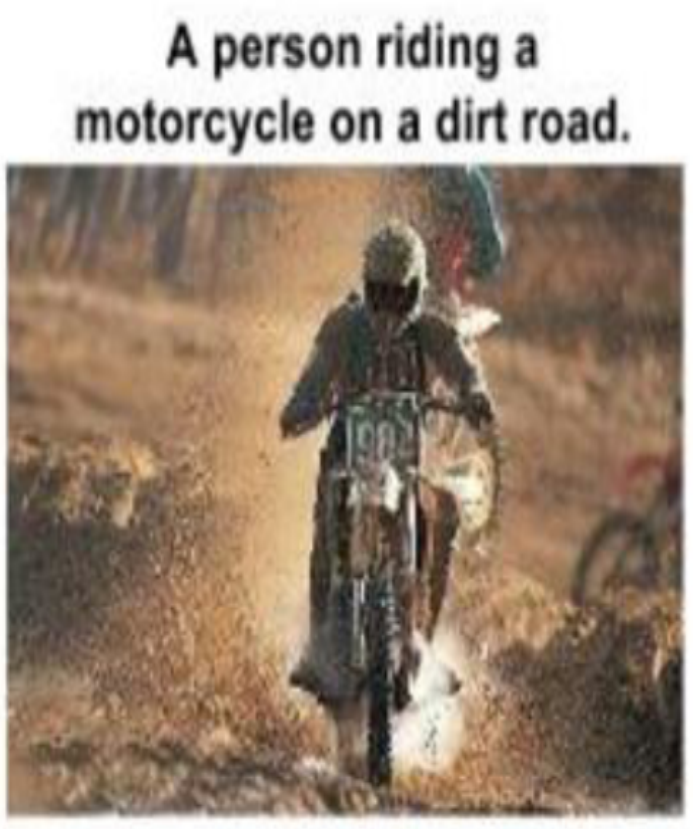
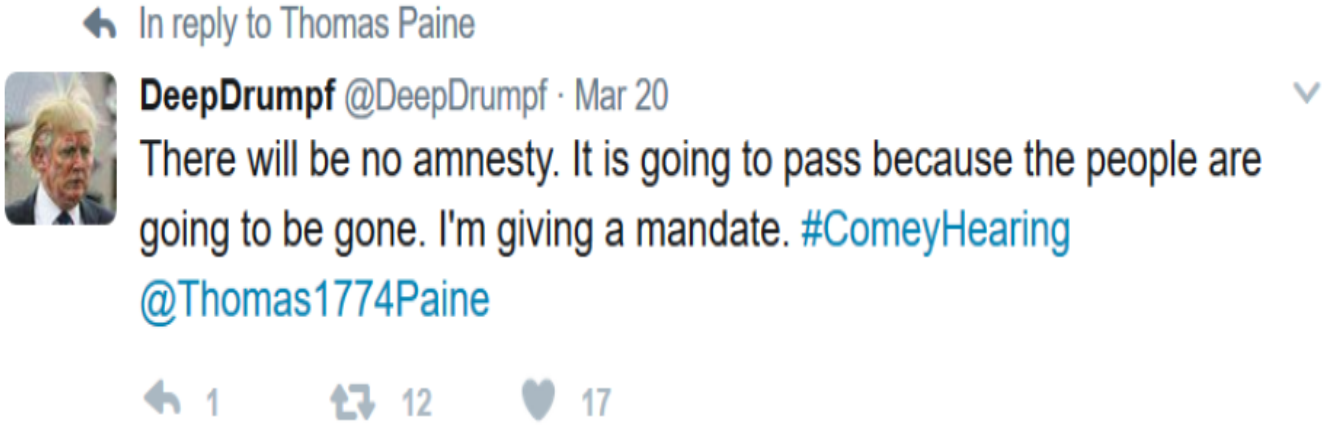


Image Captioning [reference]

... and more!

VIOLA:
Why, Salisbury must find his flesh and thought
That which I am not aps, not a man and in fire,
To show the reining of the raven and the wars
To grace my hand reproach within, and not a fair are hand,
That Caesar and my goodly father's world;

Write like Shakespeare [reference]



.. and Trump [reference]

Architectures:RNN:Applications

Technically, an RNN models sequences

Time series

Natural Language, Speech

We can even convert non-sequences to sequences, eg: feed an image as a sequence of pixels!

Architectures:RNN:Applications

Can model sequences having variable length

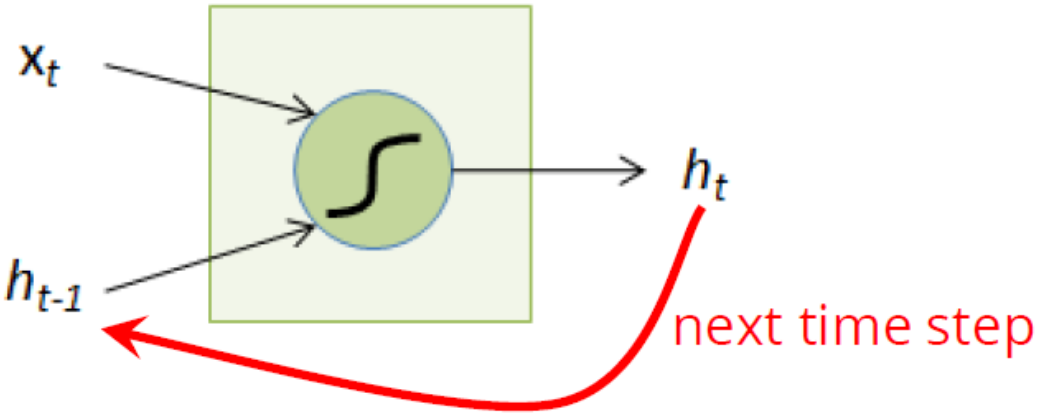
Efficient: Weights shared across time-steps

They work!

SOTA in several speech, NLP tasks

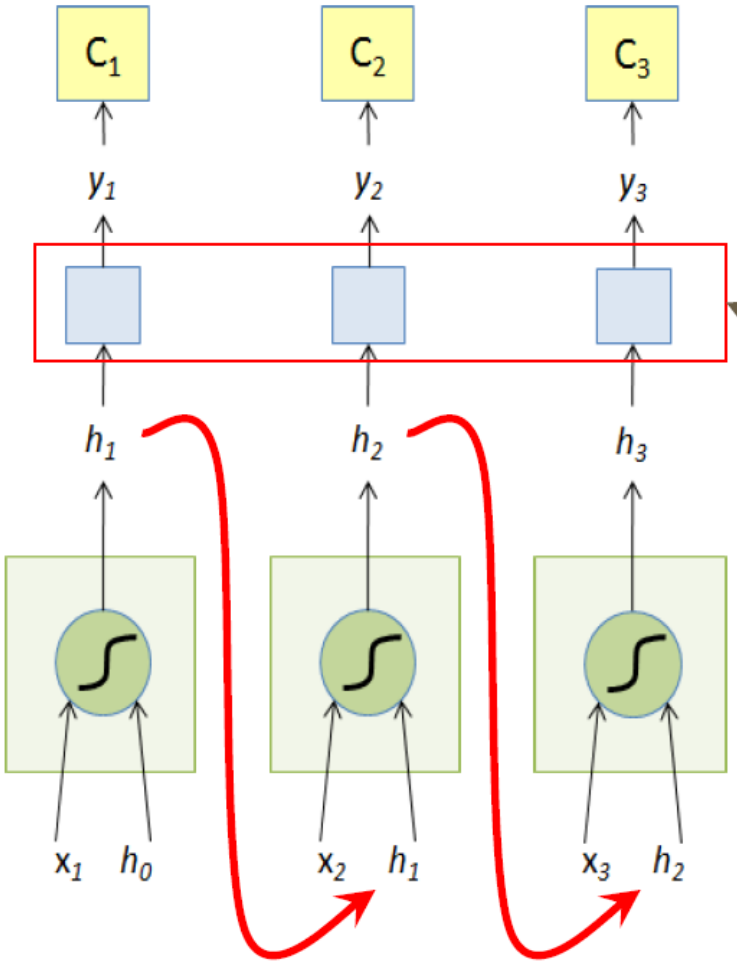
Architectures:RNN:Applications

- x_t: Input at time t
- h_{t-1}: State at time t-1



$$h_t = f(W_h h_{t-1} + W_x x_t)$$

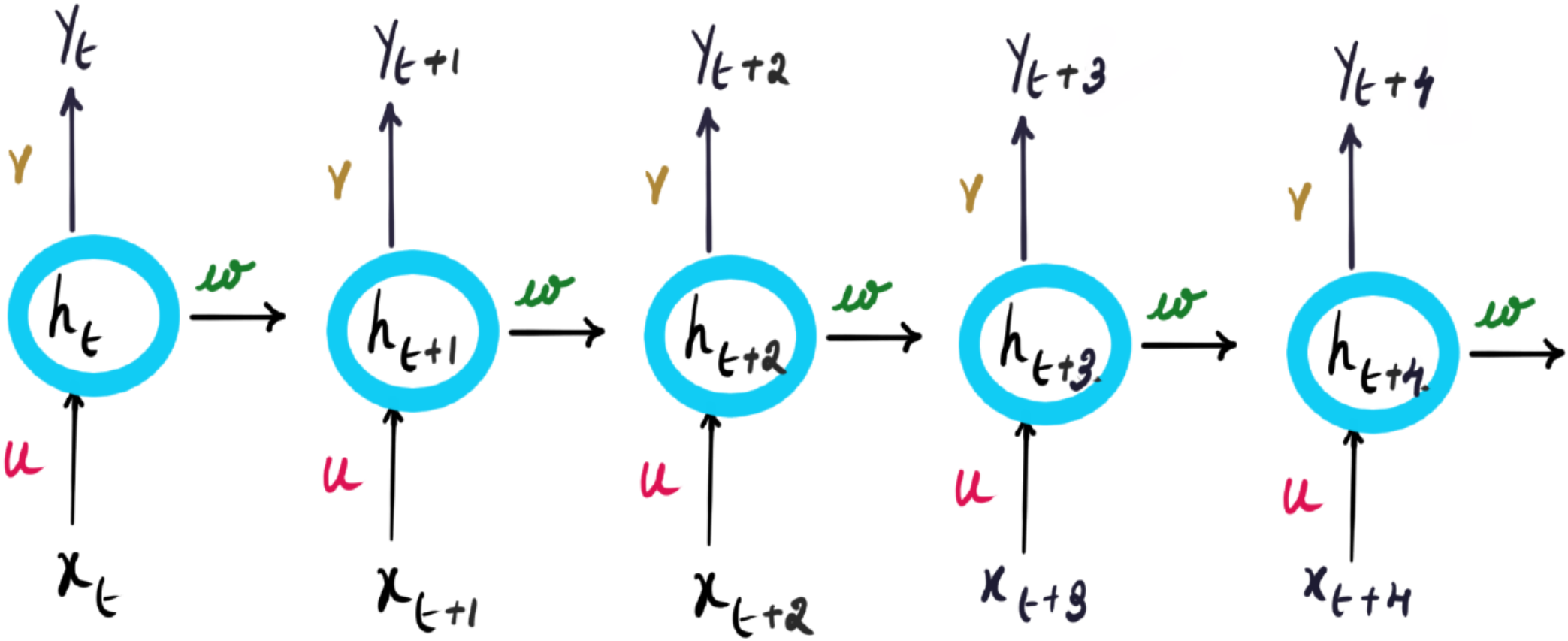
Architectures:RNN:Applications



$$h_t = f(W_h h_{t-1} + W_x x_t)$$

Weights shared over time!

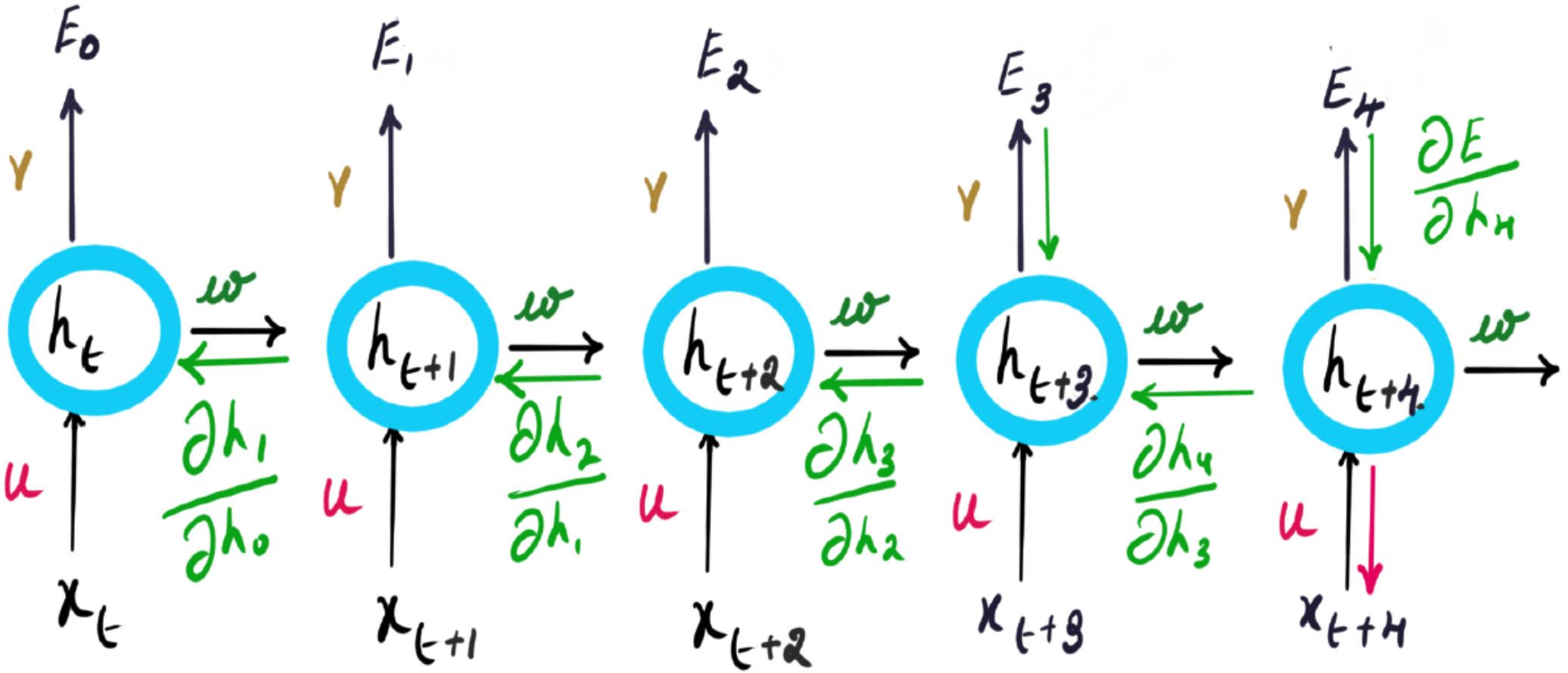
Architectures:RNN:Applications



$$h_t = \tanh(u \cdot x_t + w \cdot h_{(t-1)} + bh)$$

$$y_t = \text{softmax}(v \cdot h_t)$$

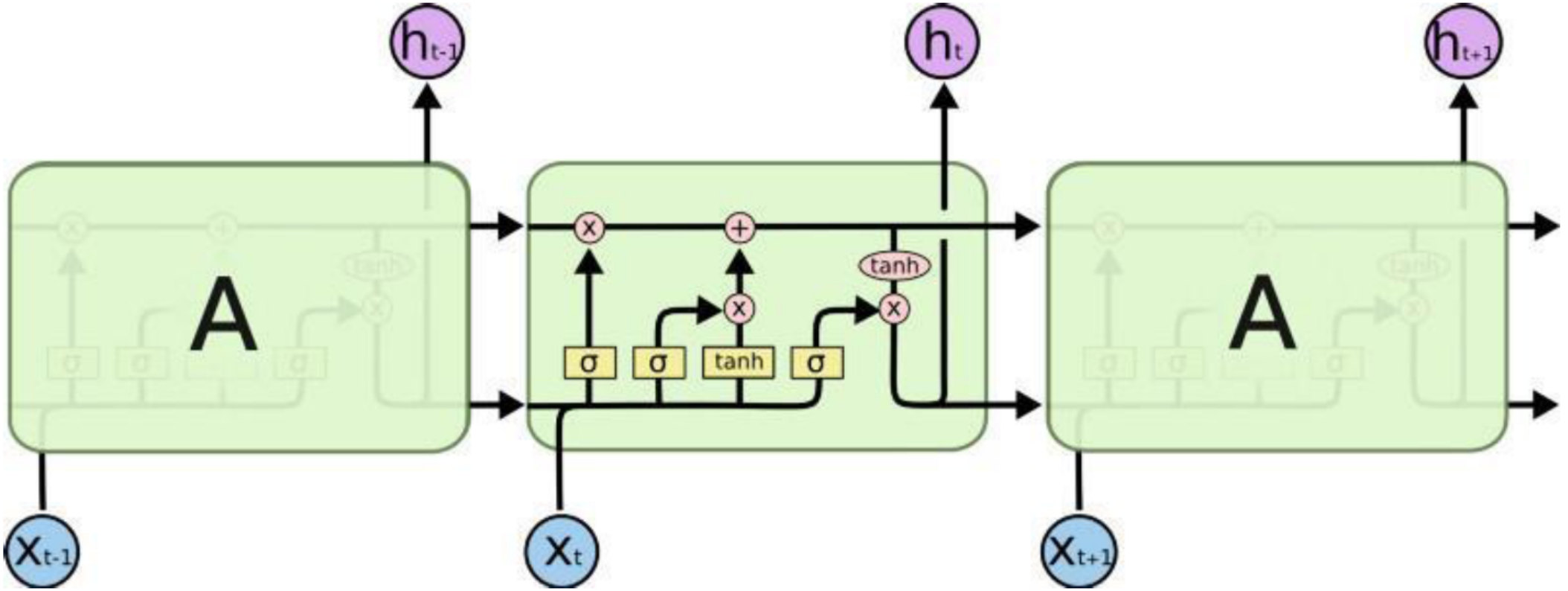
Architectures:RNN:Applications



- Derivatives have to be calculated w.r.t all the weights and biases.
- Derivatives to calculated

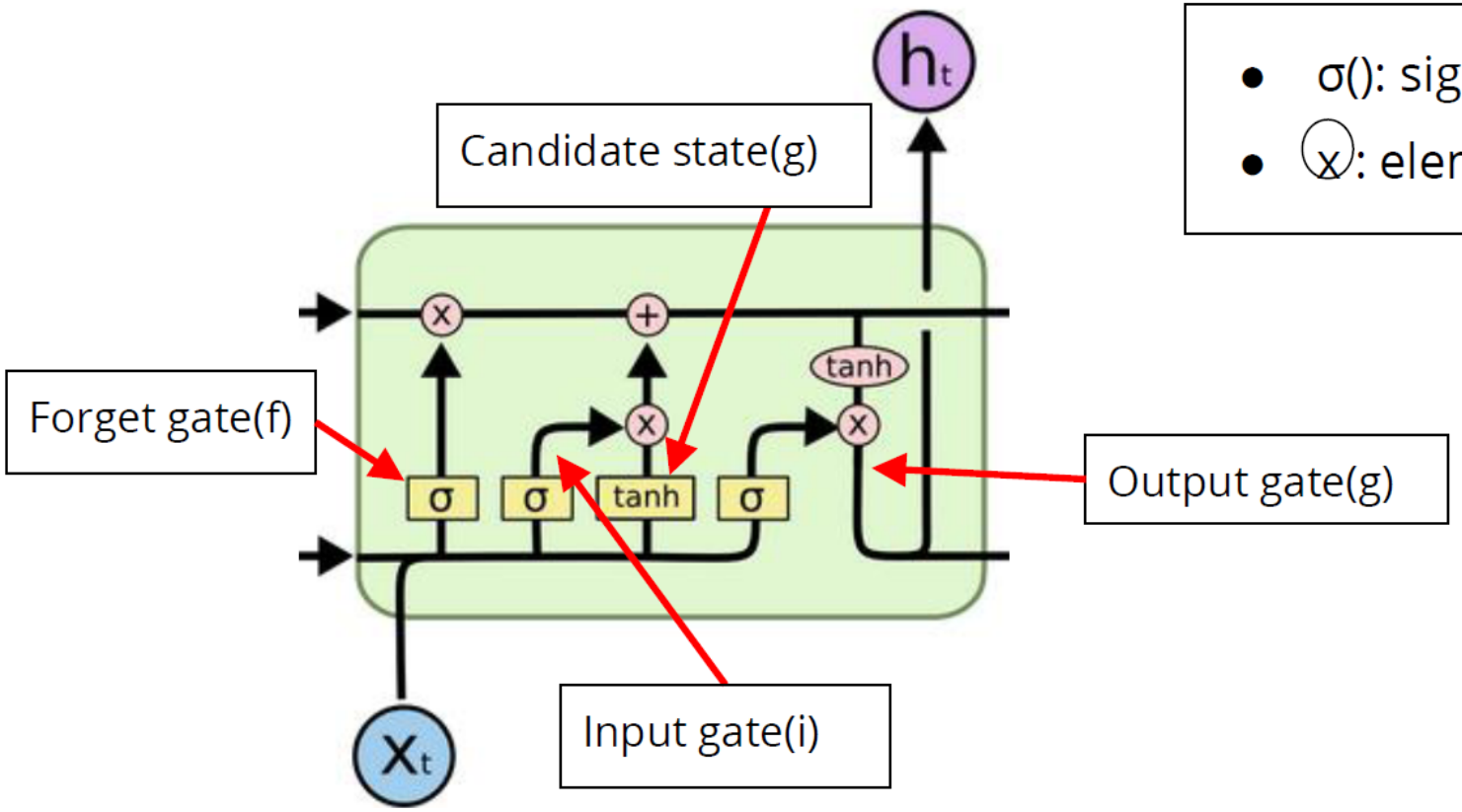
$$\frac{\partial E}{\partial u}, \frac{\partial E}{\partial w}, \frac{\partial E}{\partial r}, \frac{\partial E}{\partial bh}, \frac{\partial E}{\partial by}$$

Architectures:RNN:Gated



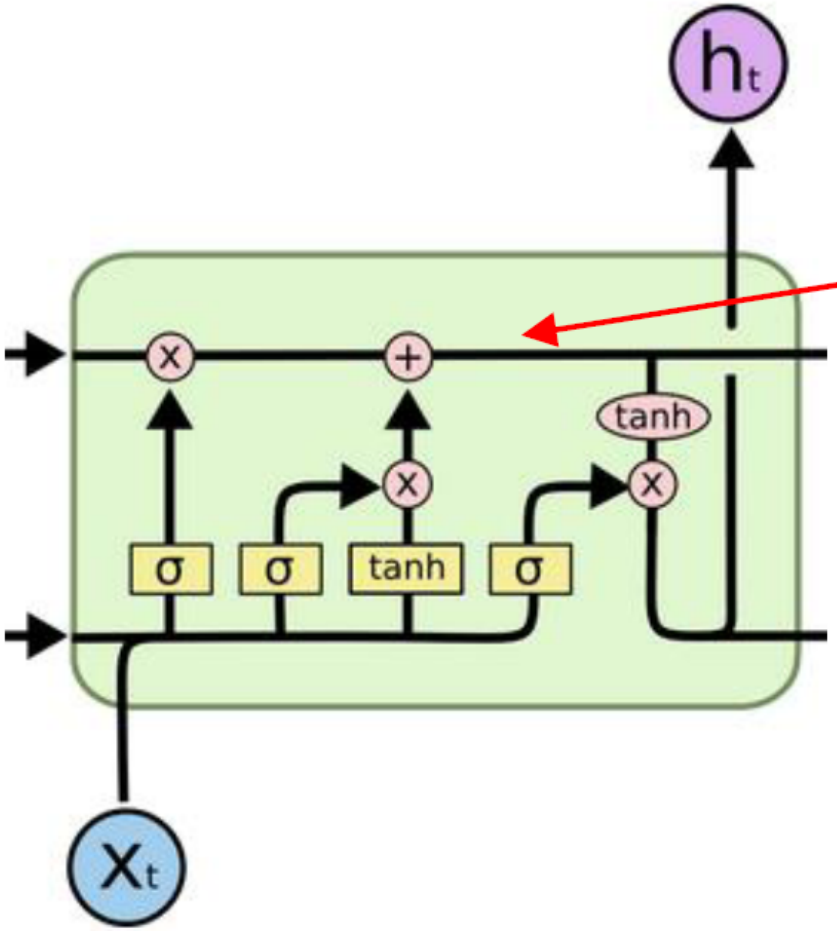
Source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Architectures:RNN:gated



- σ(): sigmoid non-linearity
- ⊗: element-wise multiplication

Architectures:RNN:gated



$$\begin{pmatrix} \tilde{\mathbf{f}}_t \\ \tilde{\mathbf{i}}_t \\ \tilde{\mathbf{o}}_t \\ \tilde{\mathbf{g}}_t \end{pmatrix} = \mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}_x \mathbf{x}_t + \mathbf{b}$$

$$\mathbf{c}_t = \sigma(\tilde{\mathbf{f}}_t) \odot \mathbf{c}_{t-1} + \sigma(\tilde{\mathbf{i}}_t) \odot \tanh(\tilde{\mathbf{g}}_t)$$

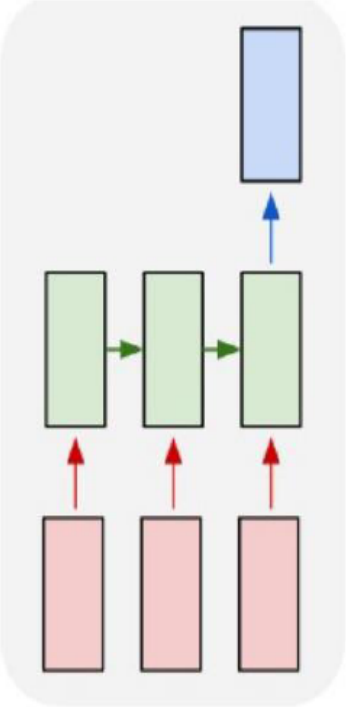
$$\mathbf{h}_t = \sigma(\mathbf{o}_t) \odot \tanh(\mathbf{c}_t),$$

Forget old state

Remember new state

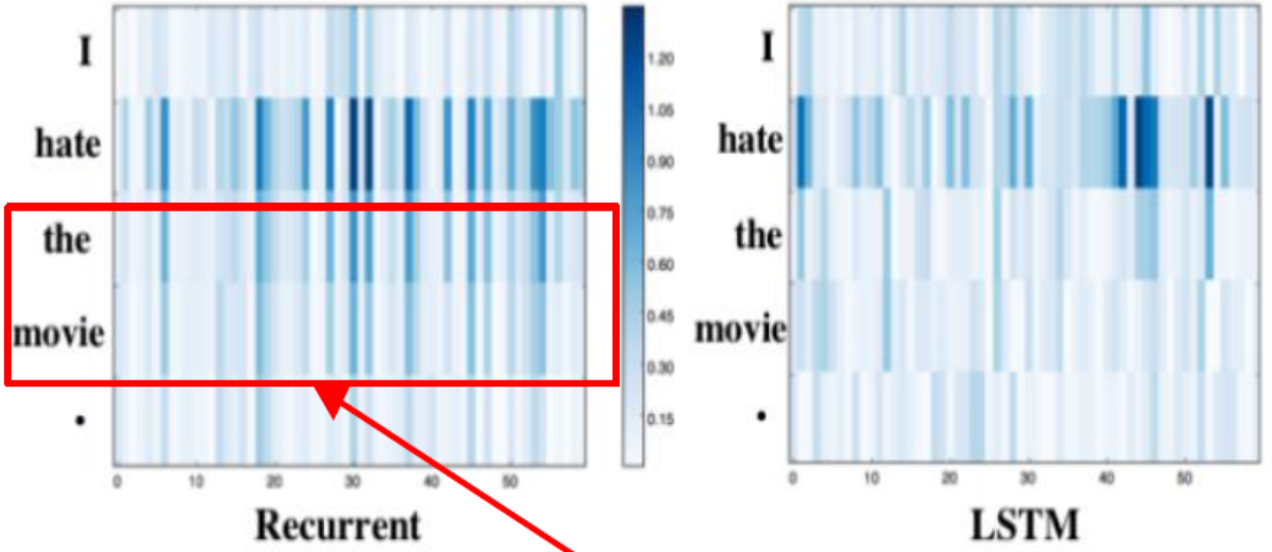
Architectures:RNN:gated

Sentiment Analysis



Many-one network

Saliency Heatmap

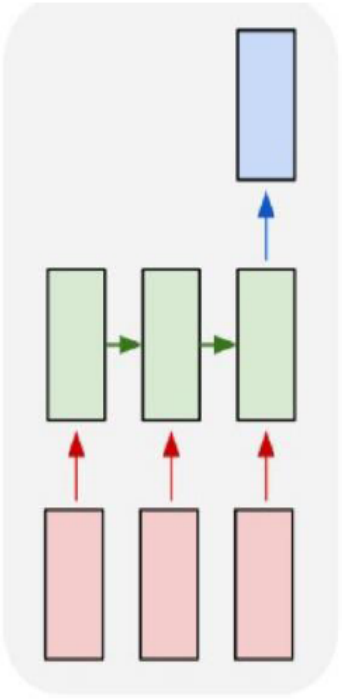


Recent words more salient

LSTM captures long term dependencies

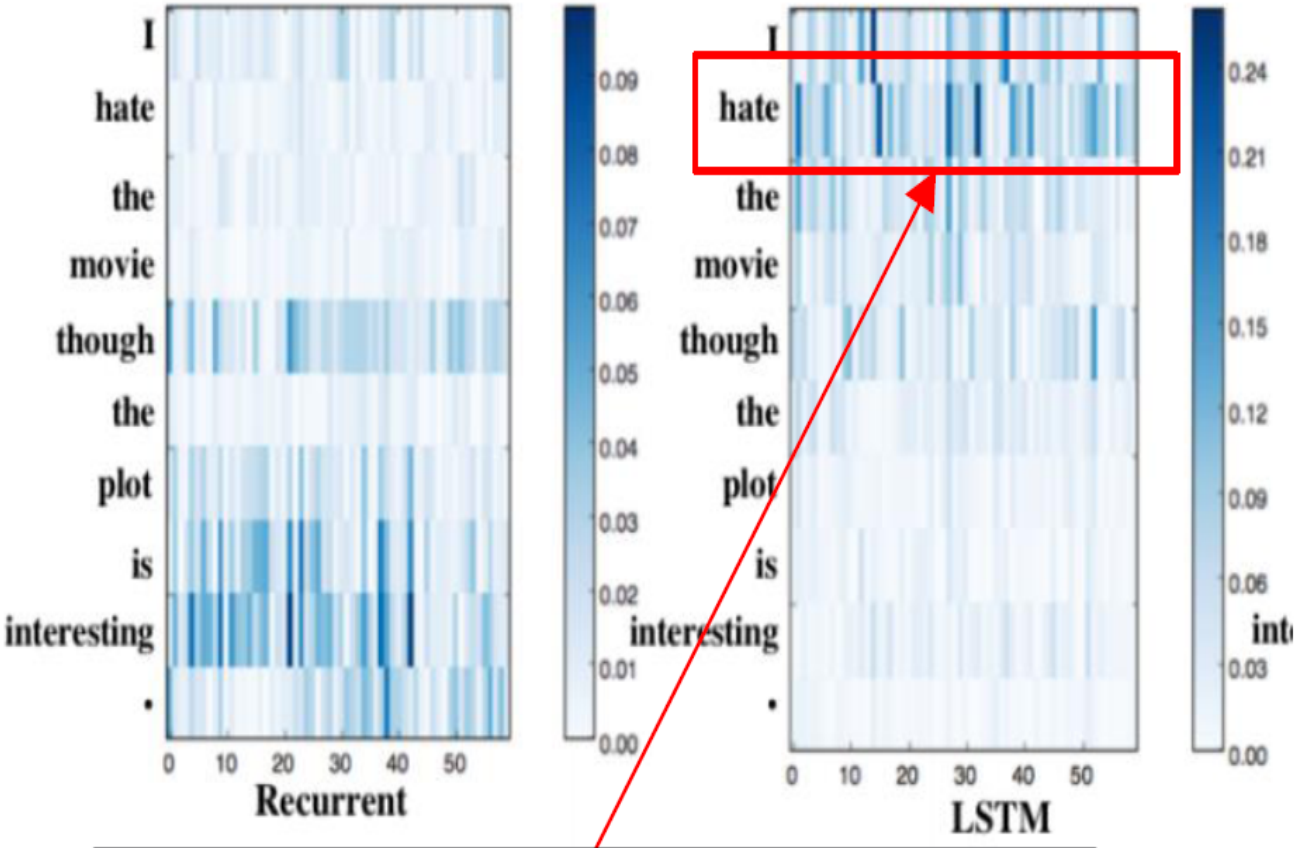
Architectures:RNN:gated

Sentiment Analysis



Many-one network

Saliency Heatmap

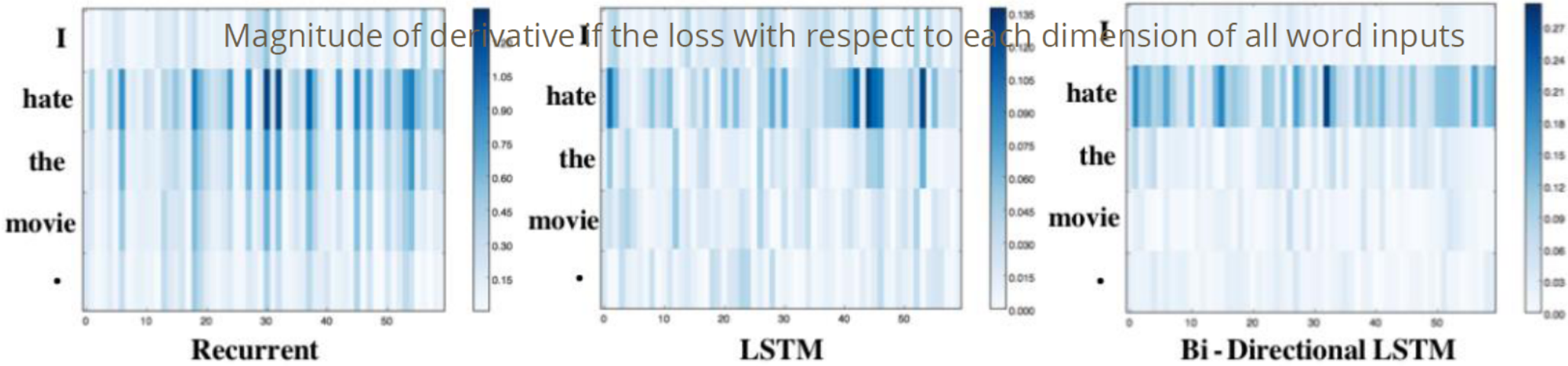


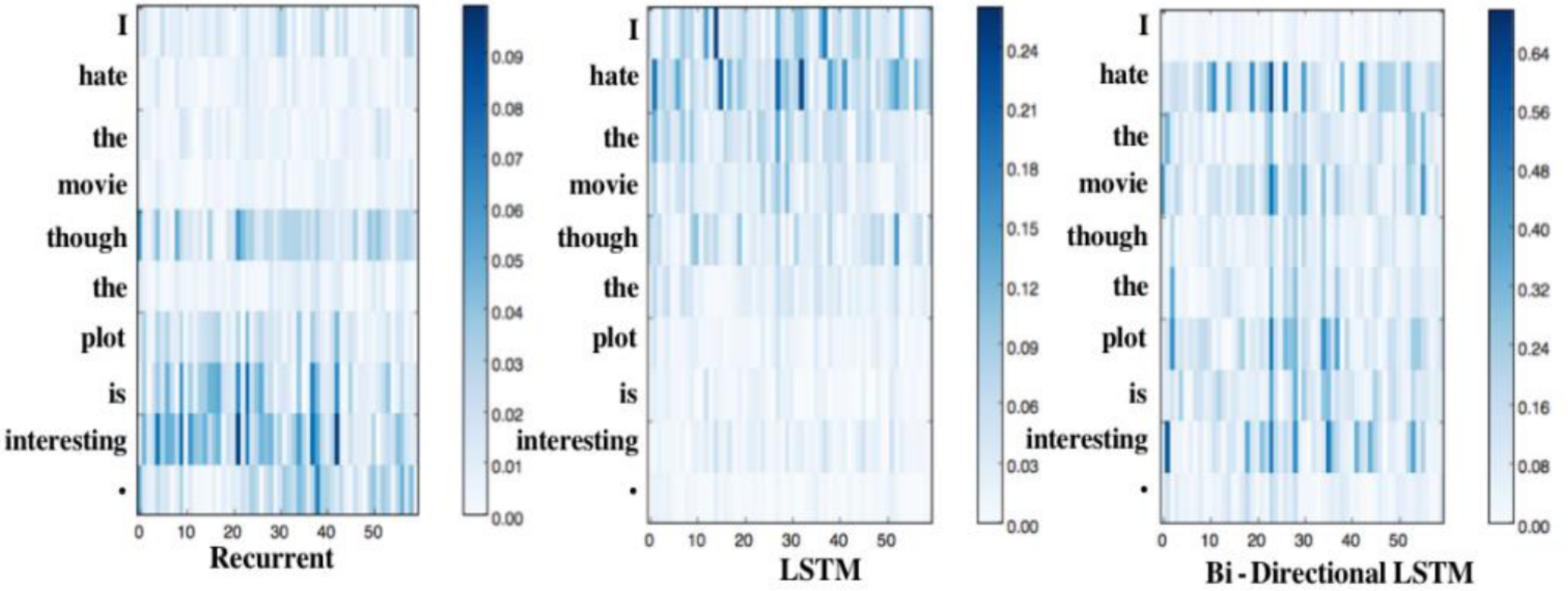
LSTM captures long term dependencies

Architectures:RNN:gated

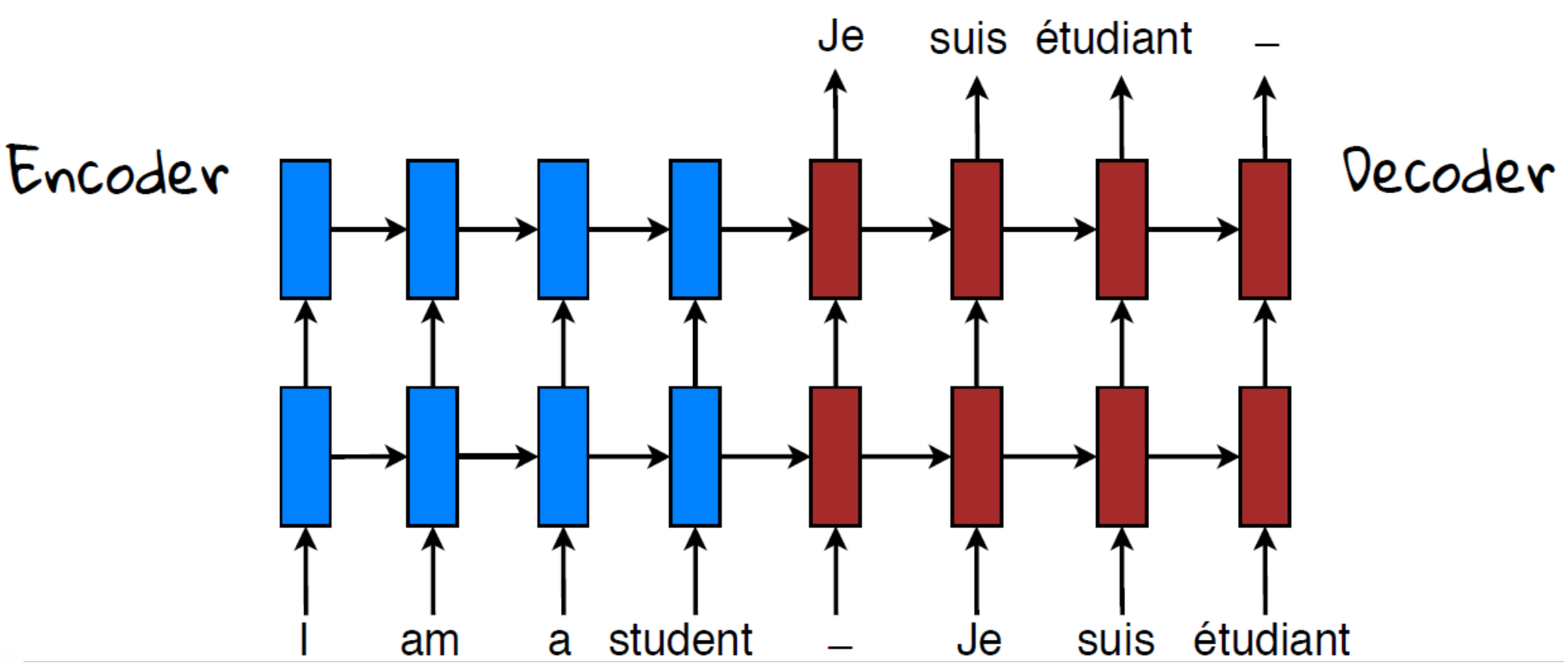
How much each unit contributes to the decision ?

Magnitude of derivative of the loss with respect to each dimension of all word inputs

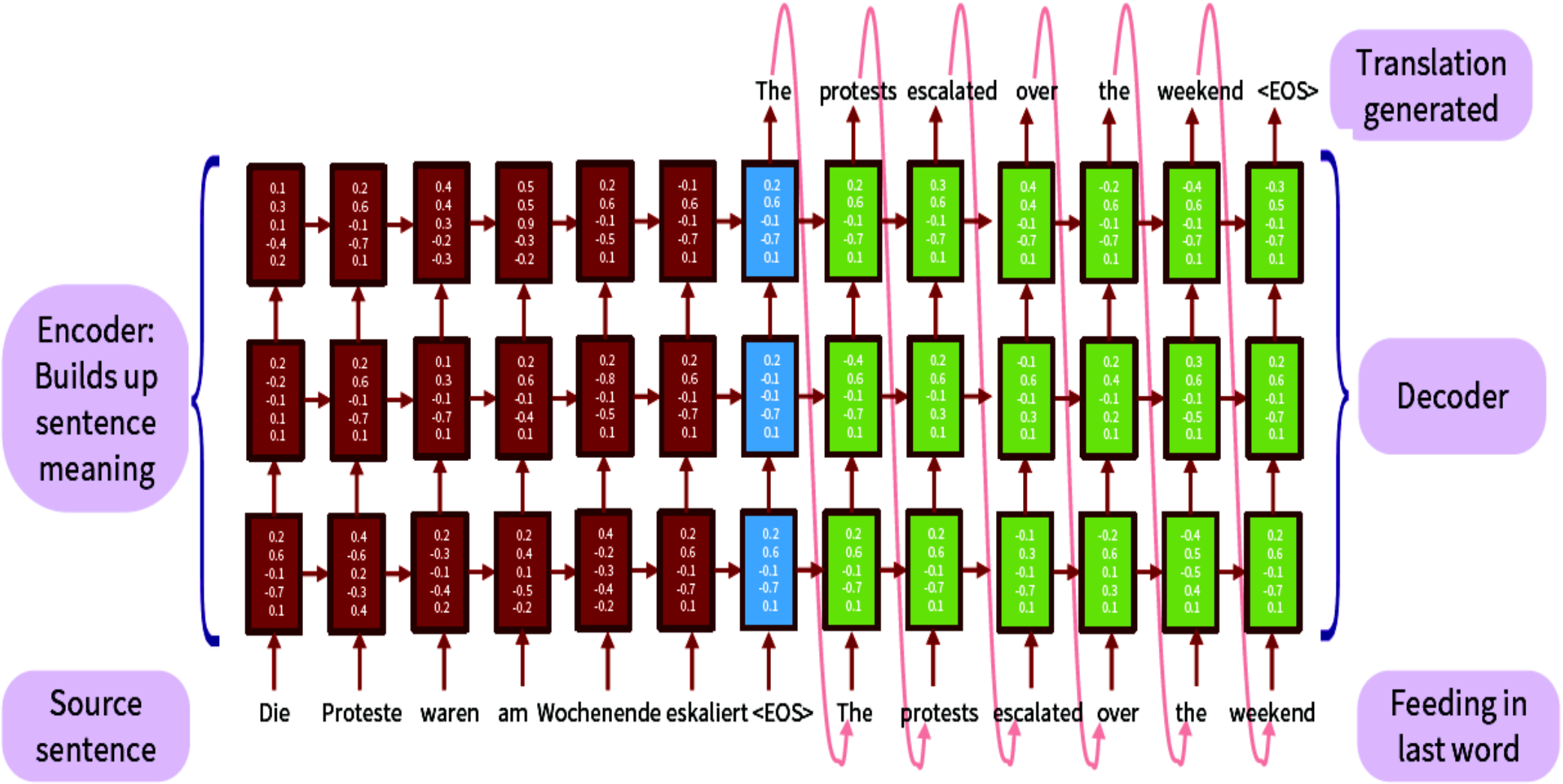




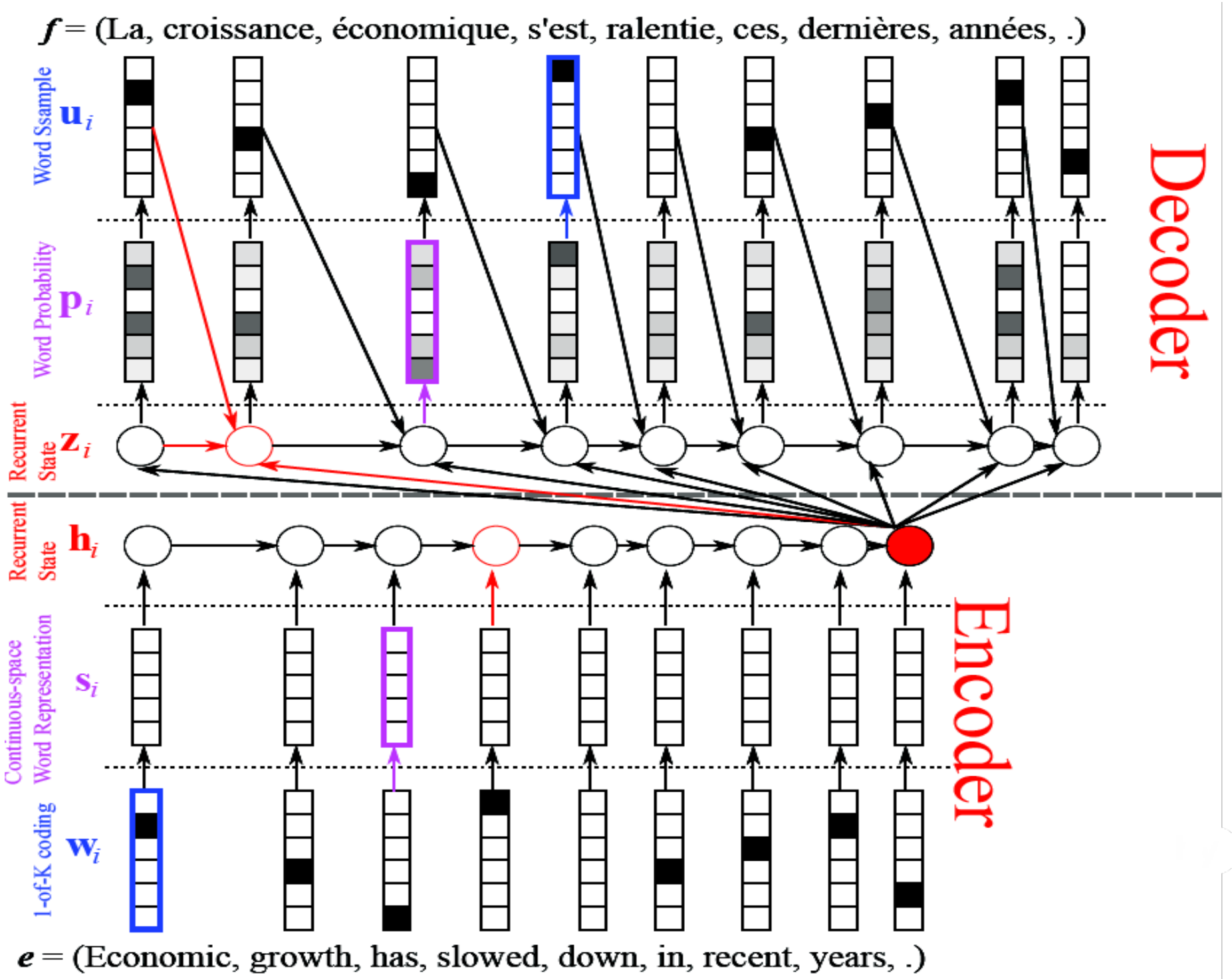
Architectures:RNN:gated:NMT(Neural Machine Translation)



Architectures:RNN:gated:NMT(Neural Machine Translation)

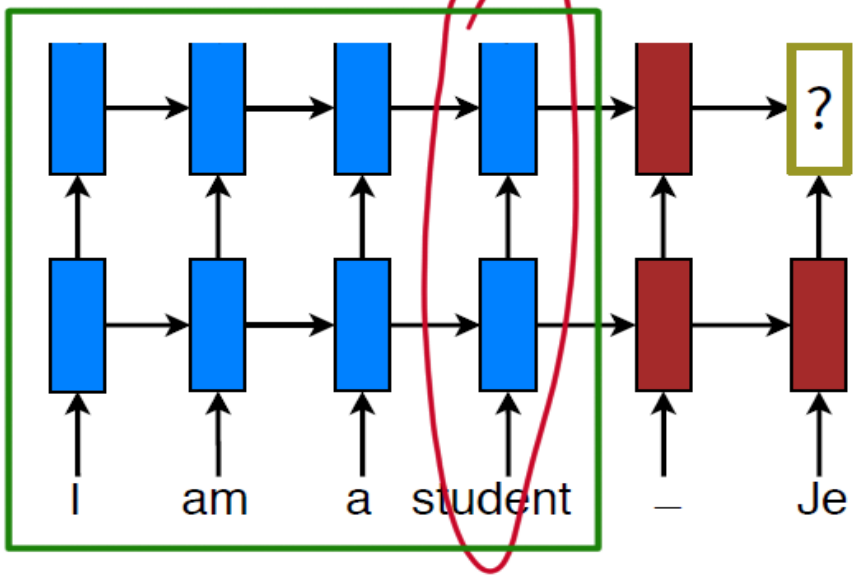


Architectures:RNN:gated:NMT(Neural



Architectures:RNN:gated:NMT+Attention

problem: fixed dimensional representation



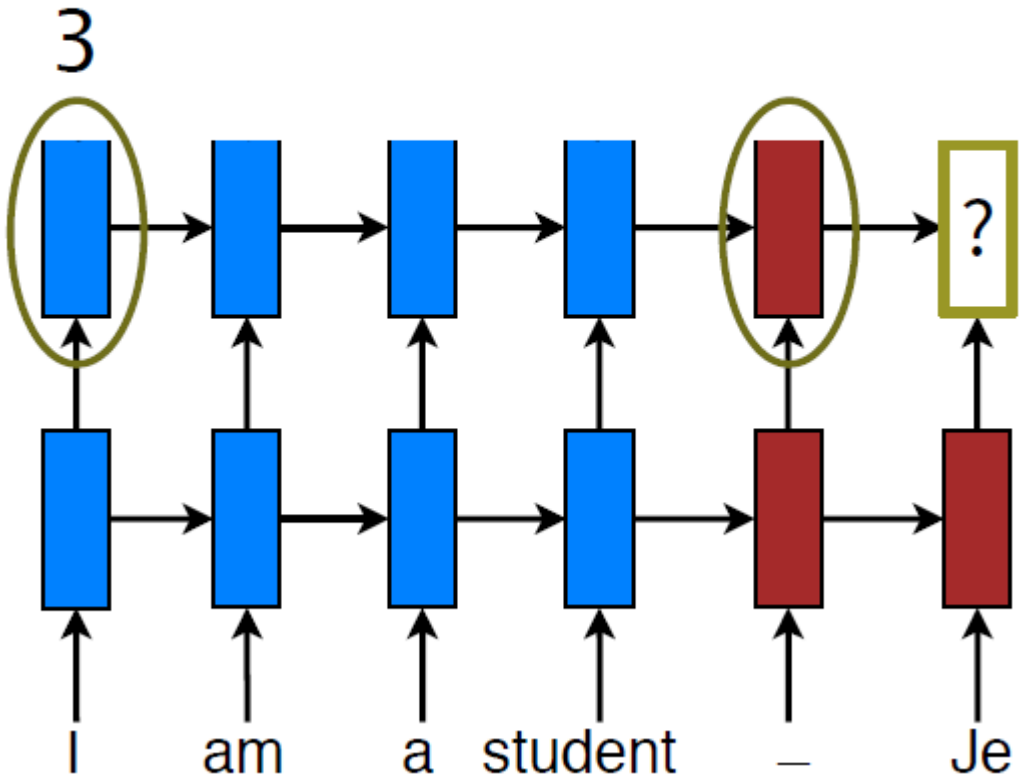
Solution: Random Access Memory

- retrieve as needed.

Simplified version of (Bahdanau et al) 2015

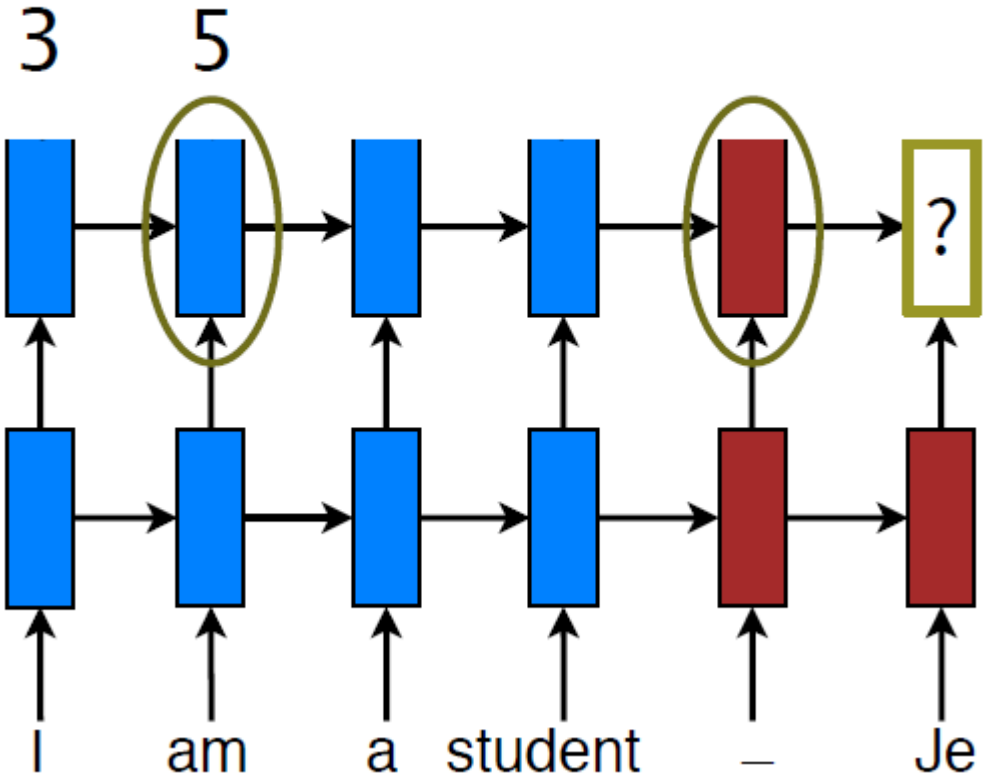
Architectures:RNN:gated:NMT+Attention

$$\text{score}(\boldsymbol{h}_{t-1}, \bar{\boldsymbol{h}}_s)$$



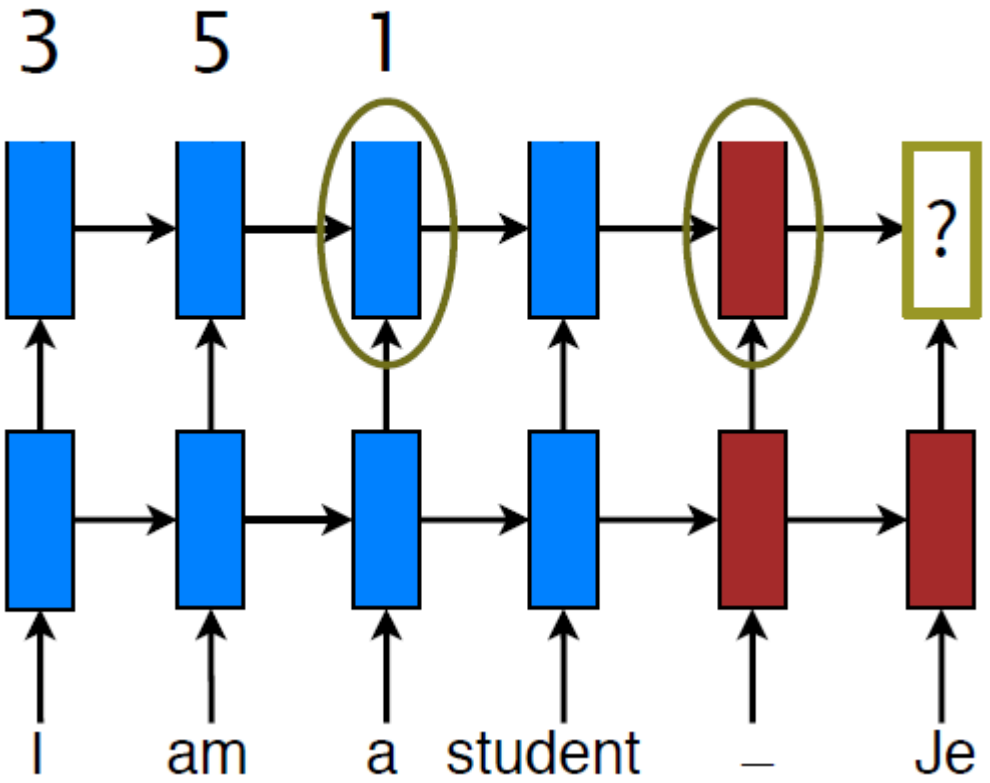
Architectures:RNN:gated:NMT+Attention

$$\text{score}(\boldsymbol{h}_{t-1}, \bar{\boldsymbol{h}}_s)$$



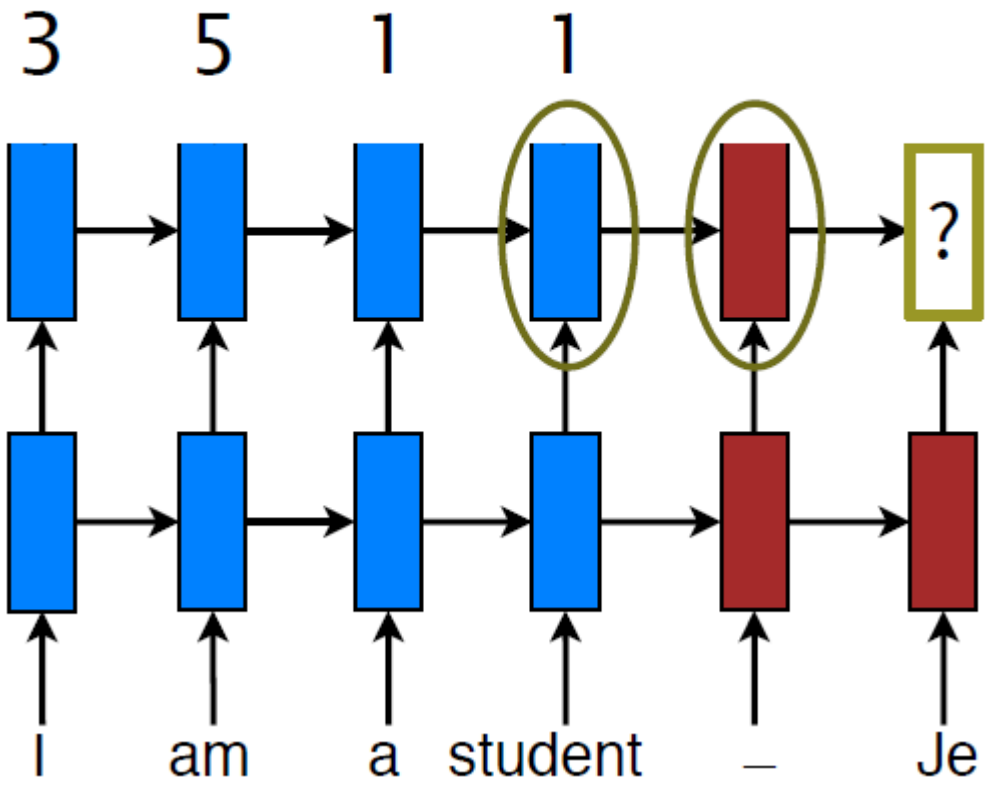
Architectures:RNN:gated:NMT+Attention

$$\text{score}(\boldsymbol{h}_{t-1}, \bar{\boldsymbol{h}}_s)$$



Architectures:RNN:gated:NMT+Attention

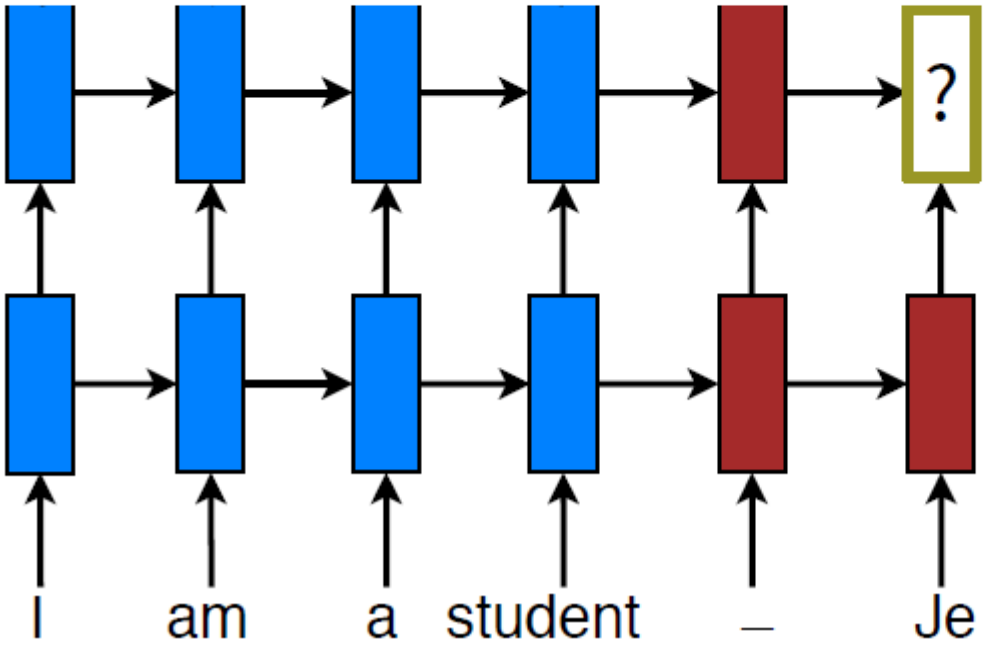
$$\text{score}(\boldsymbol{h}_{t-1}, \bar{\boldsymbol{h}}_s)$$



Architectures:RNN:gated:NMT+Attention

$$\boldsymbol{a}_t(s) = \frac{\mathrm{e}^{\mathrm{score}(s)}}{\sum_{s'} \mathrm{e}^{\mathrm{score}(s')}}$$

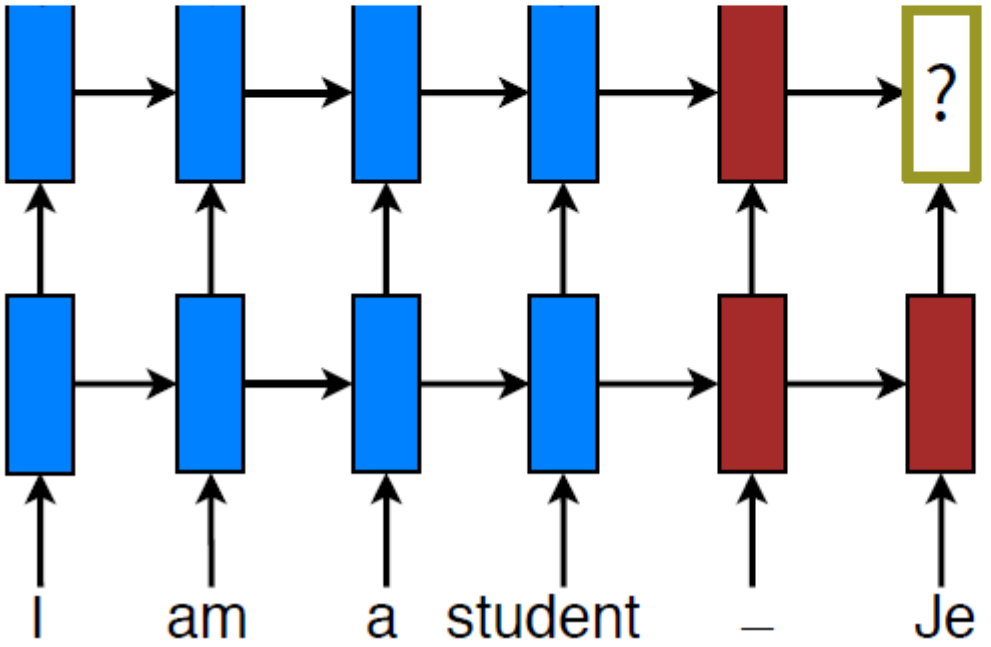
$\boldsymbol{a}_t$ 0.3 0.5 0.1 0.1



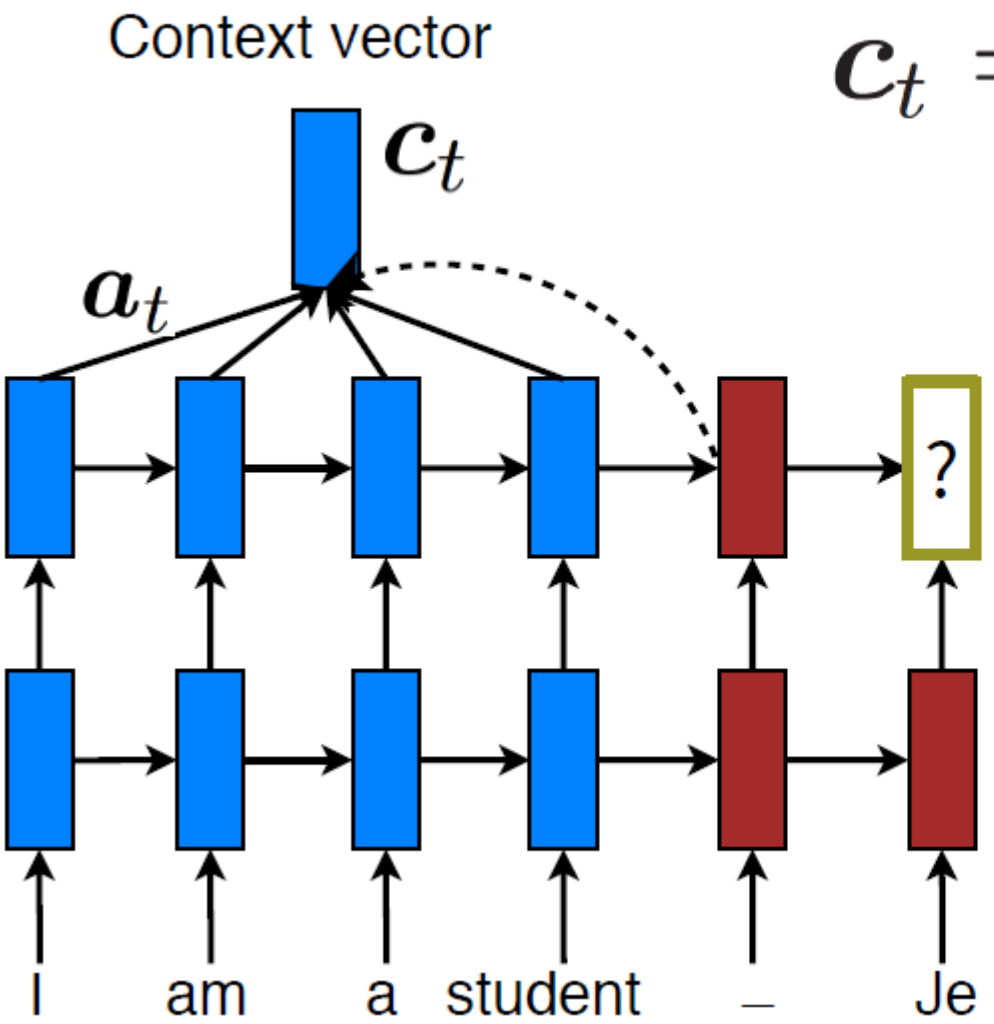
Architectures:RNN:gated:NMT+Attention

$$\boldsymbol{a}_t(s) = \frac{\mathrm{e}^{\mathrm{score}(s)}}{\sum_{s'} \mathrm{e}^{\mathrm{score}(s')}}$$

$\boldsymbol{a}_t$ 0.3 0.5 0.1 0.1

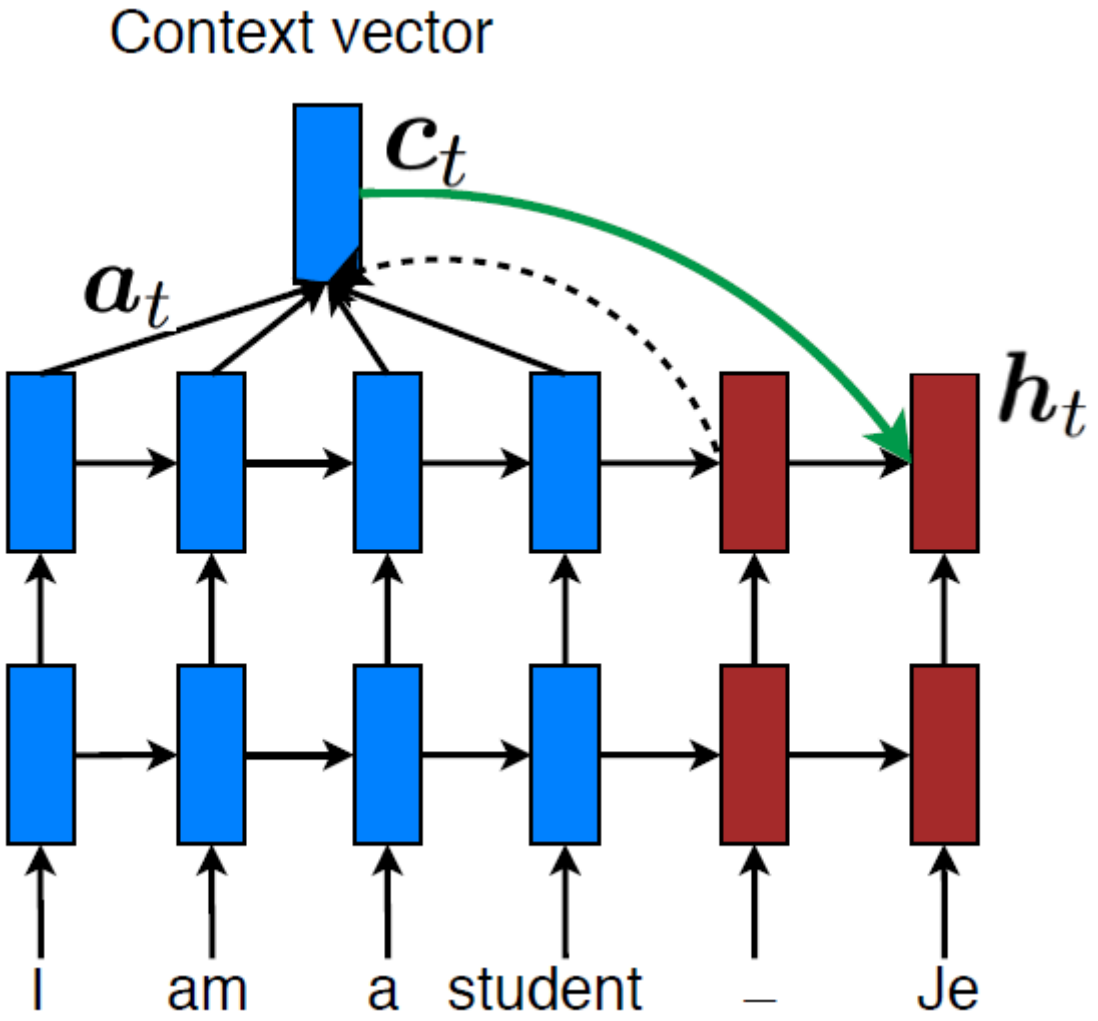


Architectures:RNN:gated:NMT+Attention

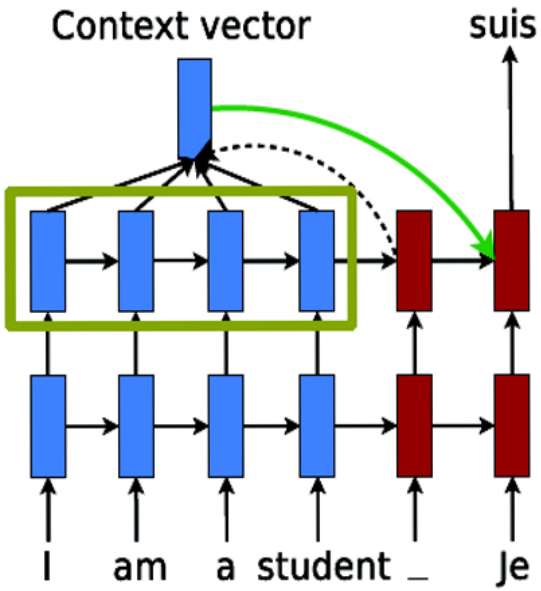


$$c_t = \sum_s a_t(s)\bar{h}_s$$

Architectures:RNN:gated:NMT+Attention



Architectures:RNN:gated:NMT+Attention

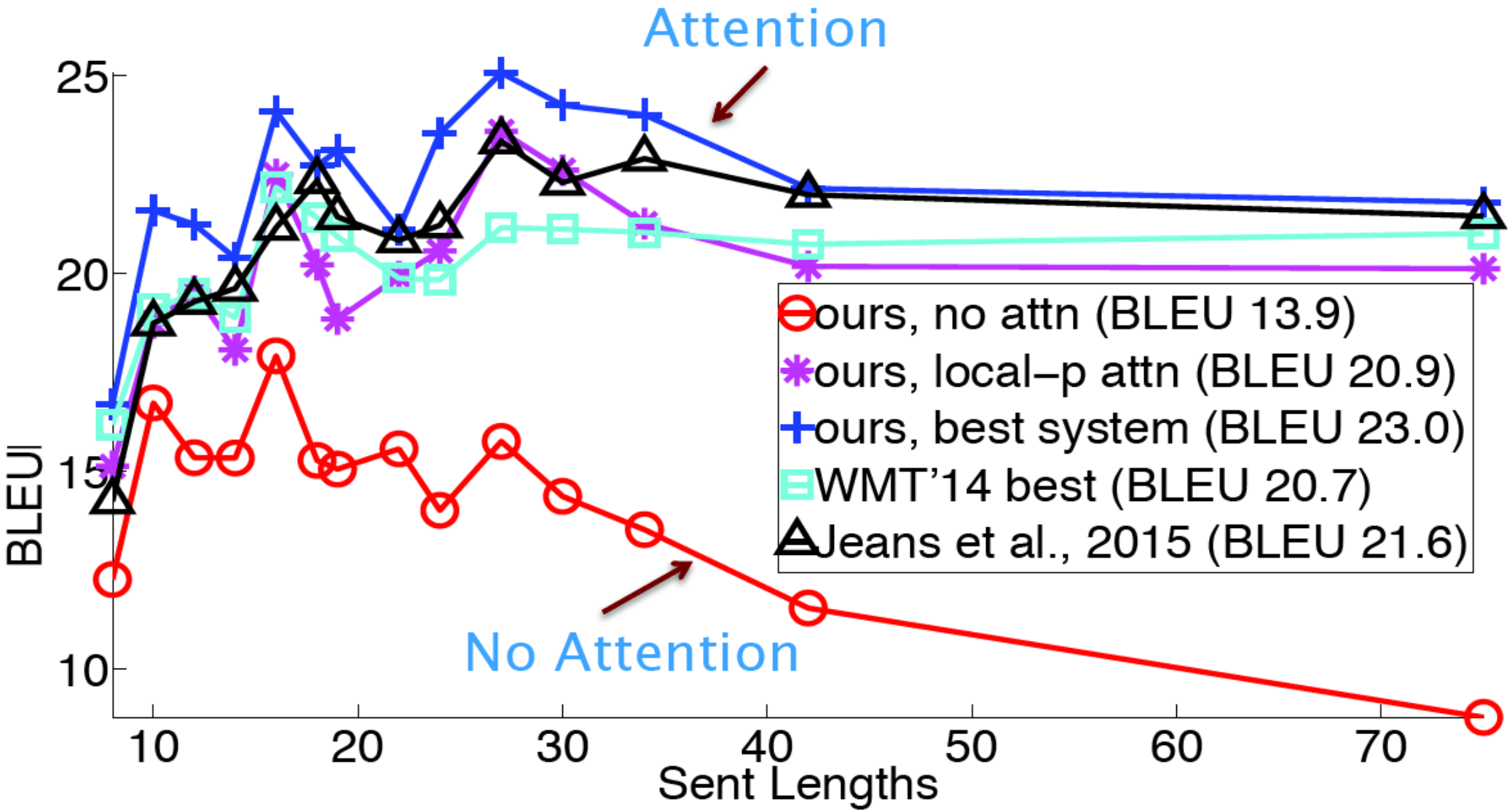


$$\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s) = \begin{cases} \boldsymbol{h}_t^\top \bar{\boldsymbol{h}}_s \\ \boldsymbol{h}_t^\top \boldsymbol{W}_a \bar{\boldsymbol{h}}_s \\ \boldsymbol{v}_a^\top \tanh\left(\boldsymbol{W}_a [\boldsymbol{h}_t; \bar{\boldsymbol{h}}_s]\right) \end{cases}$$

widely adopted, the one we use

proposed model by montreal

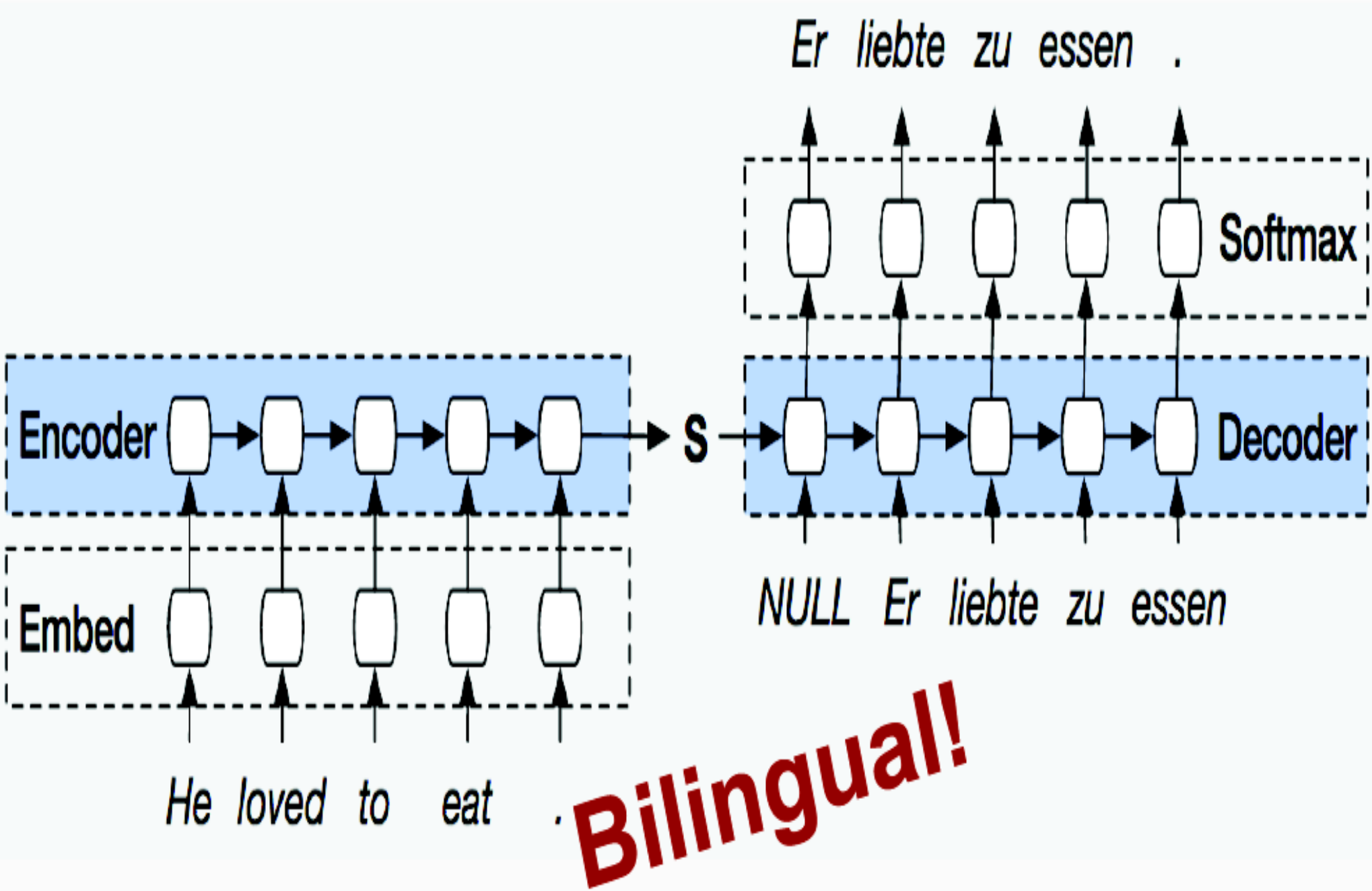
Architectures:RNN:gated:NMT+Attention



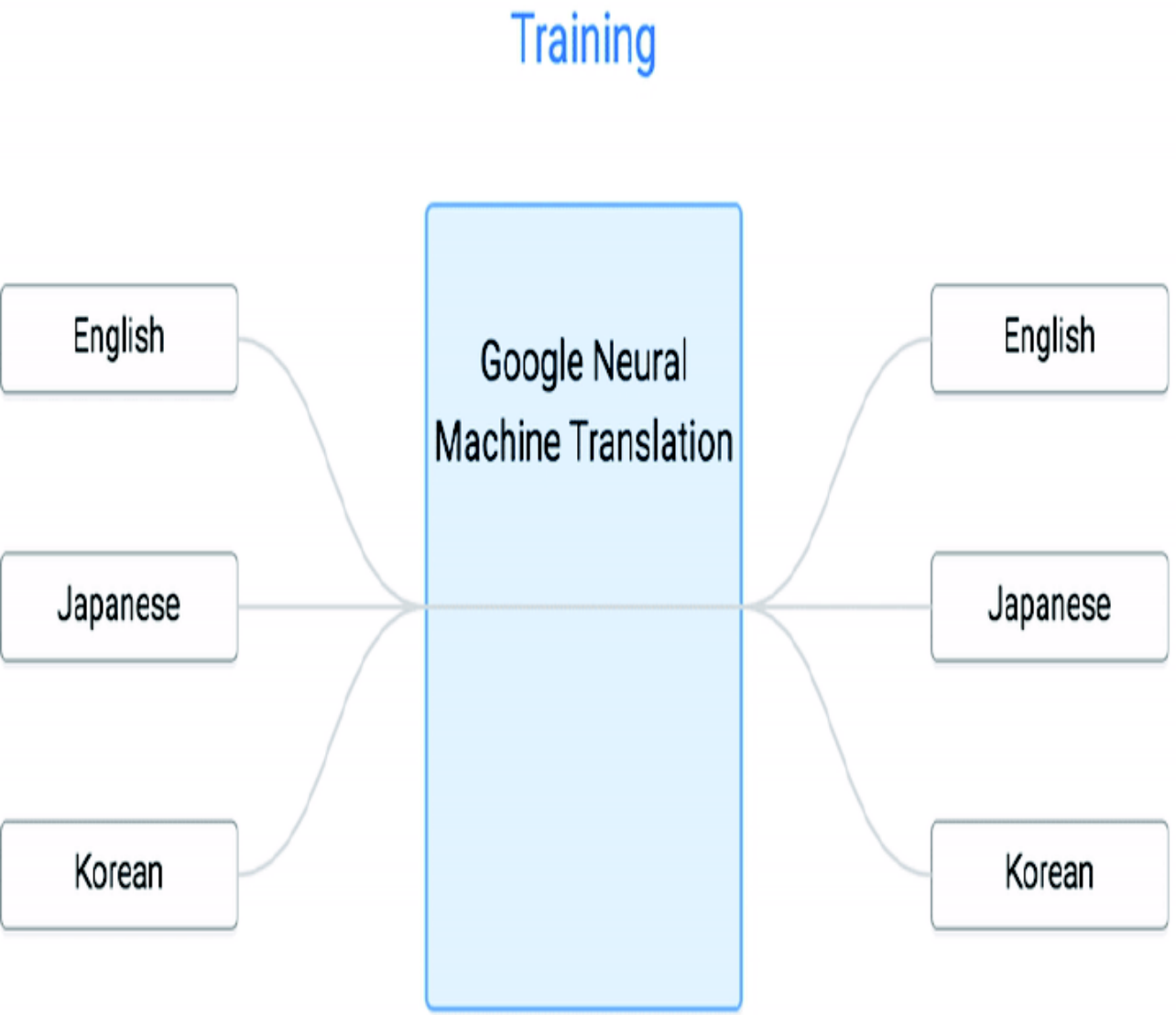
Architectures:RNN:gated:NMT+Attention

source	Orlando Bloom and ***Miranda Kerr*** still love each other
human	Orlando Bloom und **Miranda Kerr** lieben sich noch immer
+attn	Orlando Bloom und **Miranda Kerr** lieben einander noch immer .
base	Orlando Bloom und **Lucas Miranda** lieben einander noch immer .

Architectures:RNN:gated:G(Google)NMT

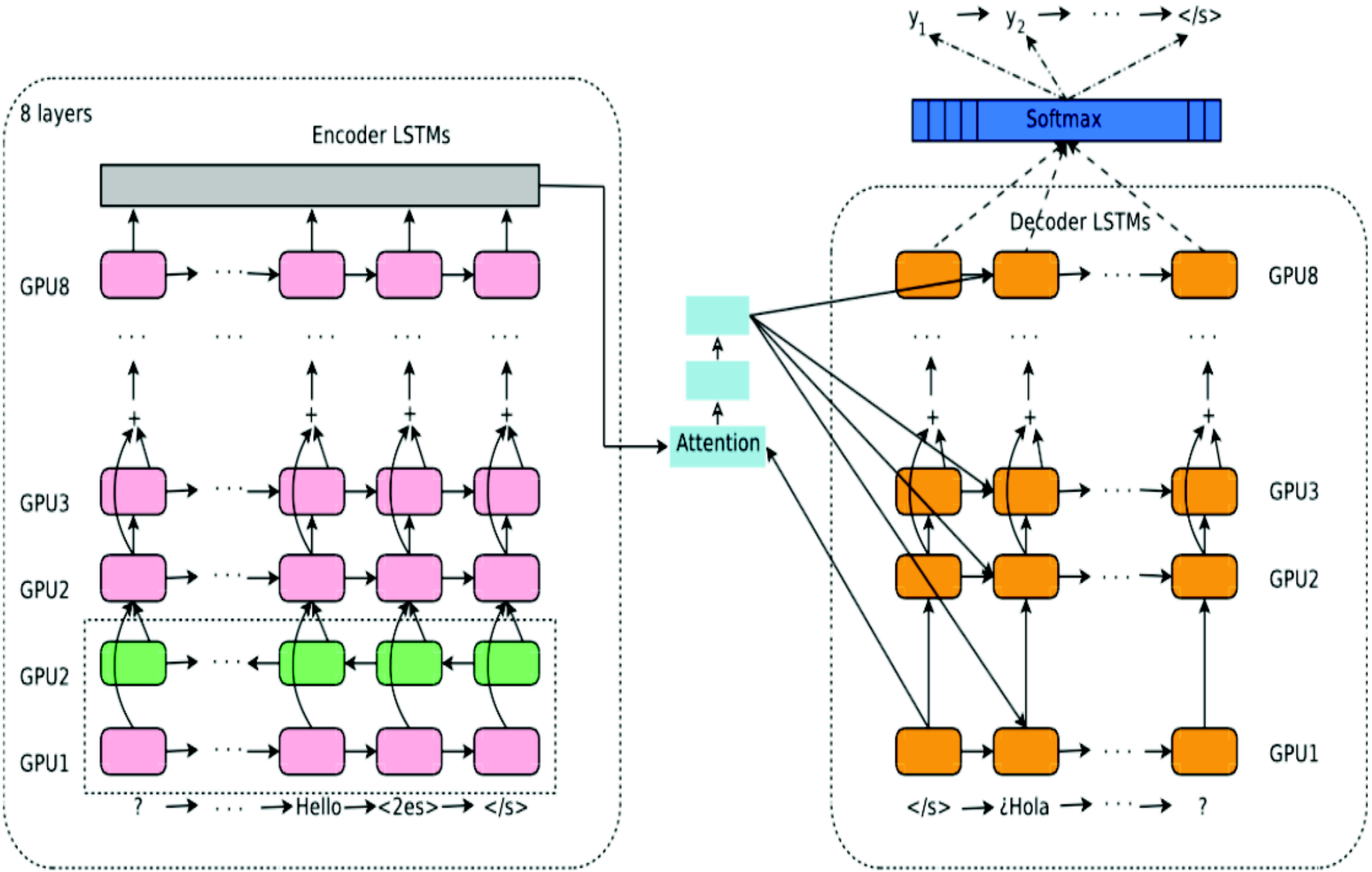


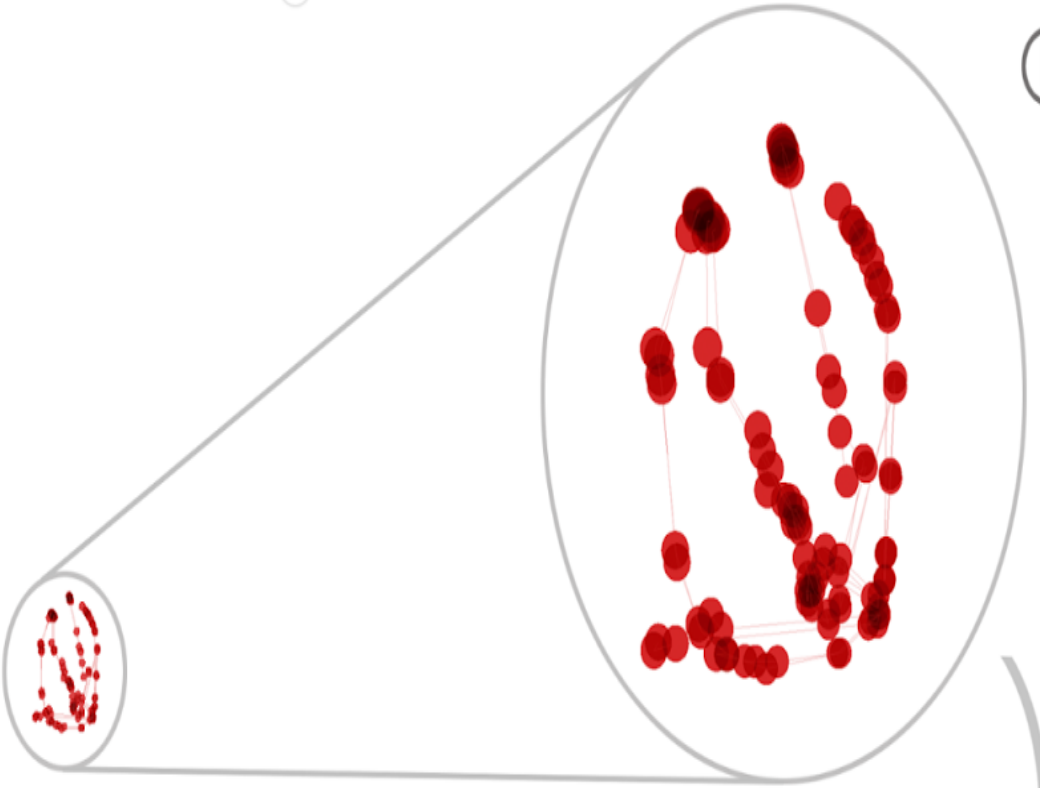
Architectures:RNN:gated:G(Google)NMT



- **Simplicity**: single model
- **Low-resource language improvements**
- **Zero-shot translation**

Architectures:RNN:gated:G(Google)NMT





ENGLISH

The stratosphere extends from about 10km to about 50km in altitude.

KOREAN

성층권은 고도 약 10km부터 약 50km까지 확장됩니다.

JAPANESE

成層圏は、高度 10km から 50km の範囲にあります.

a

b

c

Agenda:

Deep Learning: Introduction

Foundation:Linear Regression

Deep Learning: Going Deep:Intuition

Deep Learning: Going Deep:Forward pass

Deep Learning: Going Deep:Back Prop

Deep Learning: Architectures

Deep Learning: Fintechs

Deep Learning: GPU and TPU

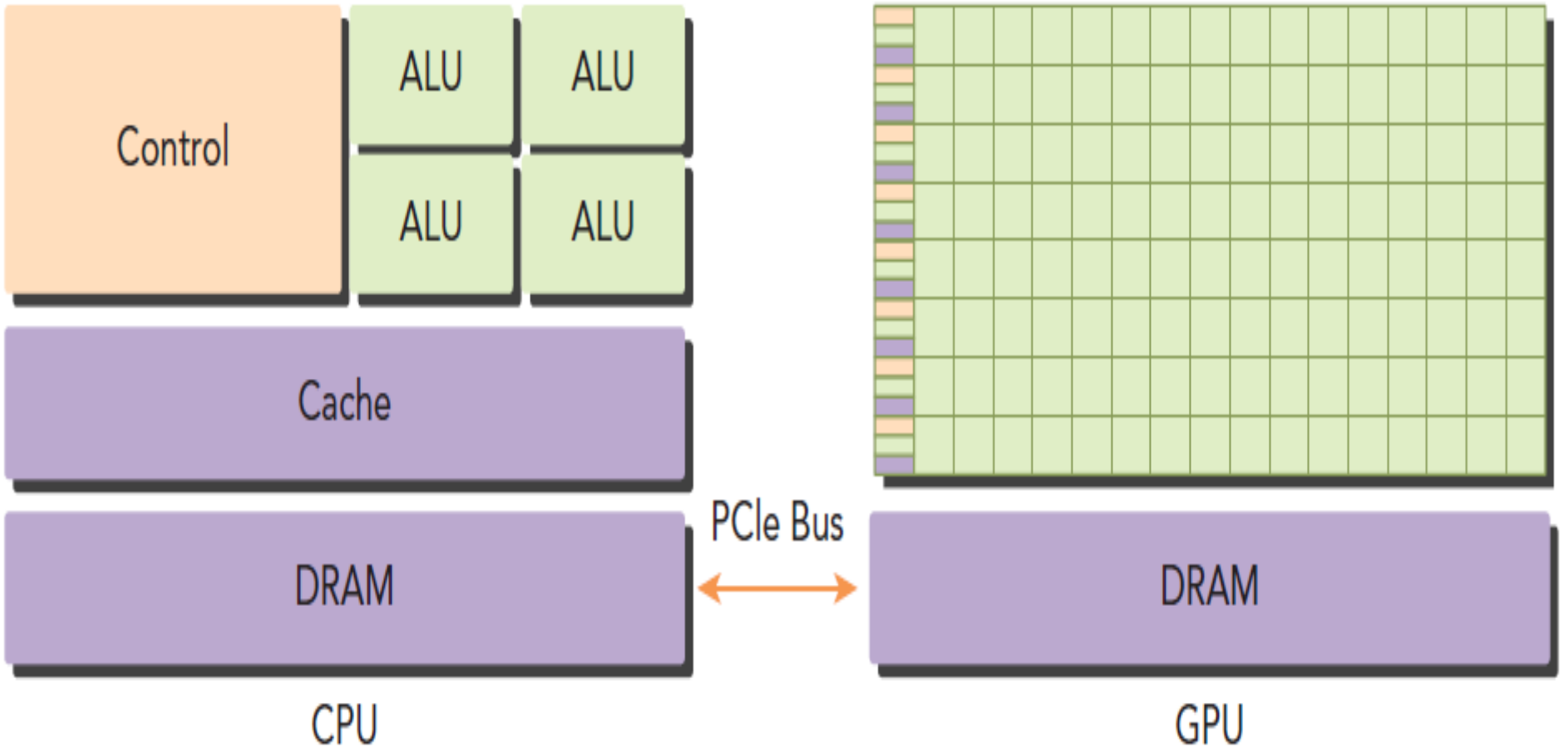
CPU VS GPU

- Which is faster?
  - 300km/hour Ferrari that ferries 2 people at a time
  - 100km/hour Volvo Bus that ferries 50 people at a time

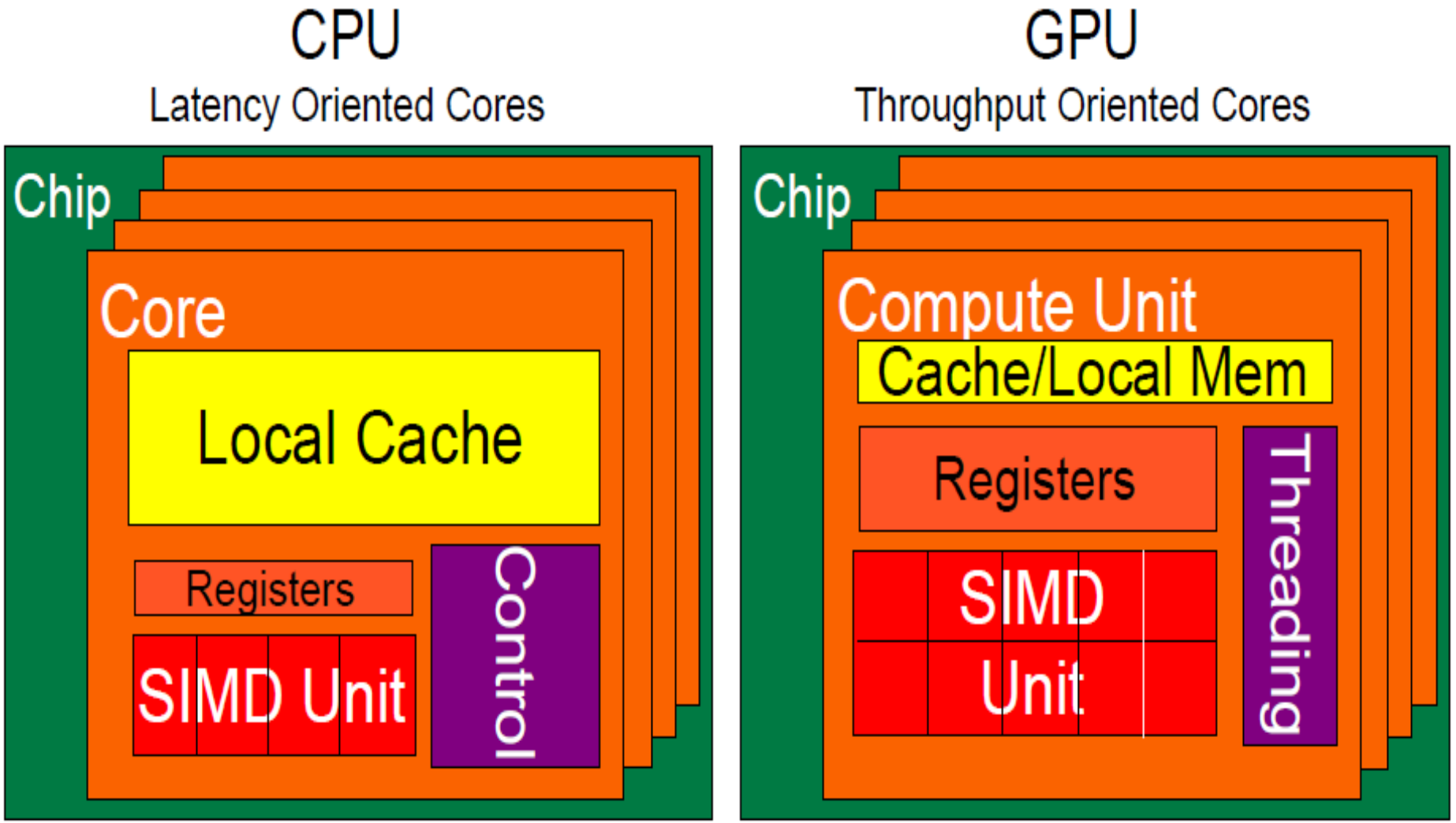
CPU VS GPU

- Which is faster?
  - 300km/hour Ferrari that ferries 2 people at a time
    - Optimize on latency
  - 100km/hour Volvo Bus that ferries 50 people at a time
    - Optimize on throughput

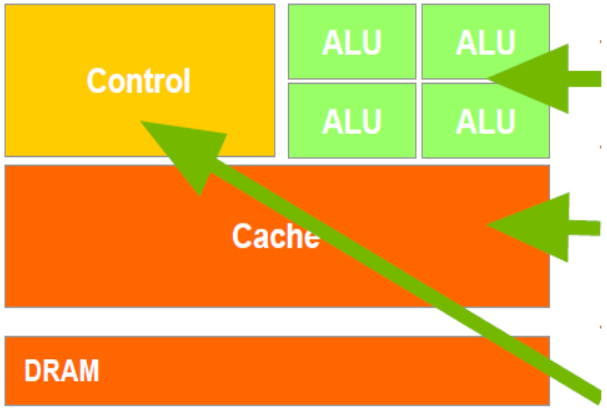
CPU VS GPU



Parallel Computing:Computer Architecture:CPU vs GPU

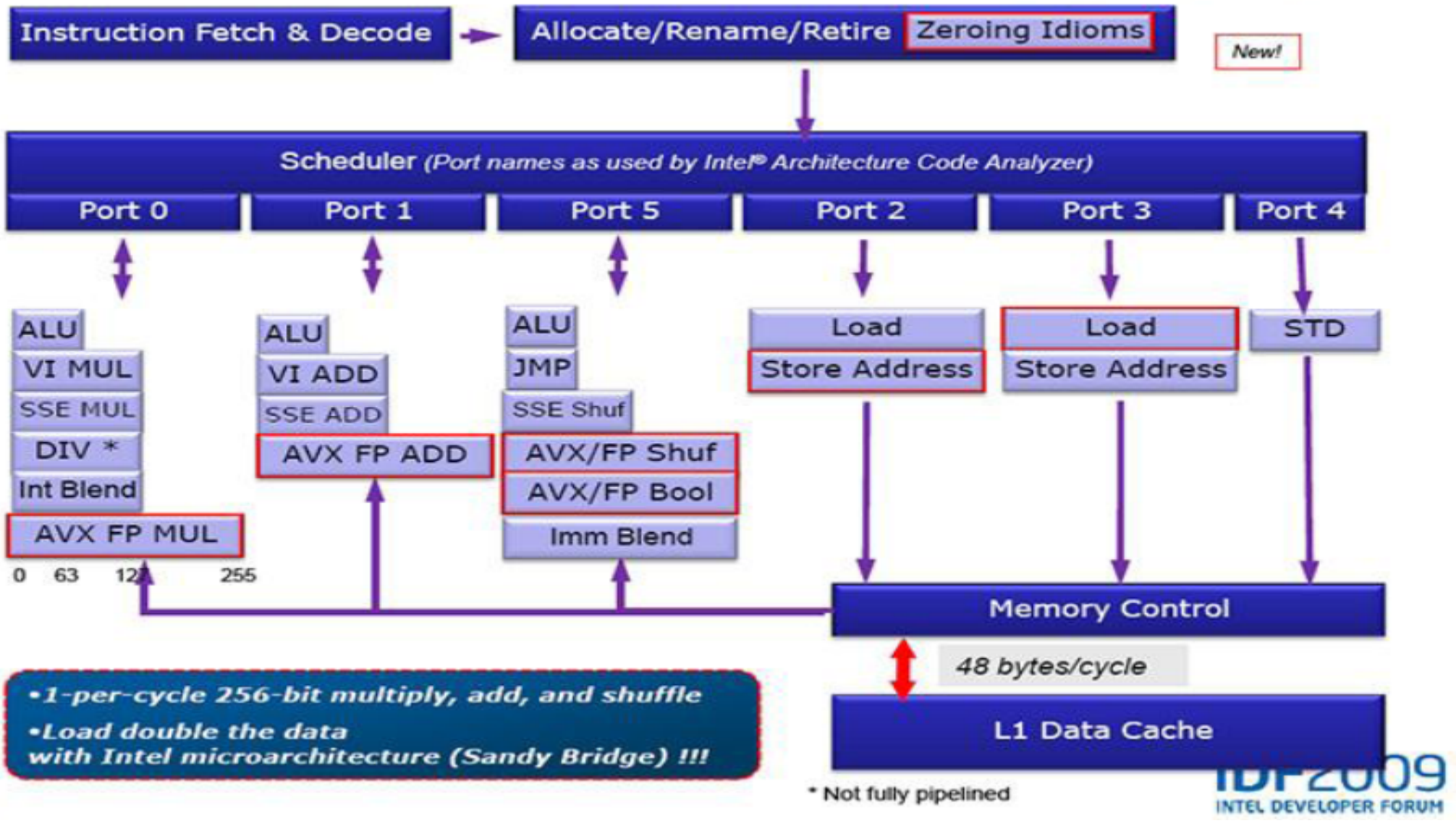


Parallel Computing:Computer Architecture:CPU vs GPU

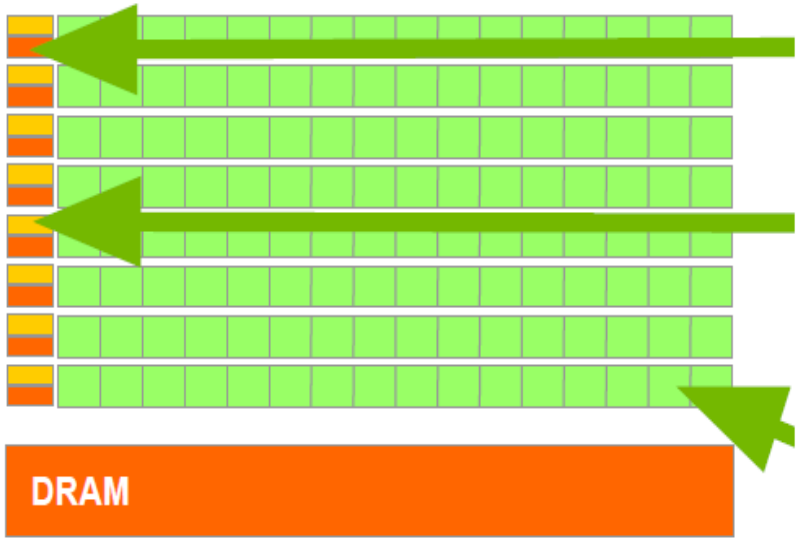


1. Powerful ALU
   - Reduces operational latency
2. Large caches
   - Converts long latency memory to short latency cache
3. Sophisticated control
   - Branch prediction for reduced latency
   - Data forwarding for reduced data latency.

Parallel Computing:Computer Architecture:CPU vs GPU

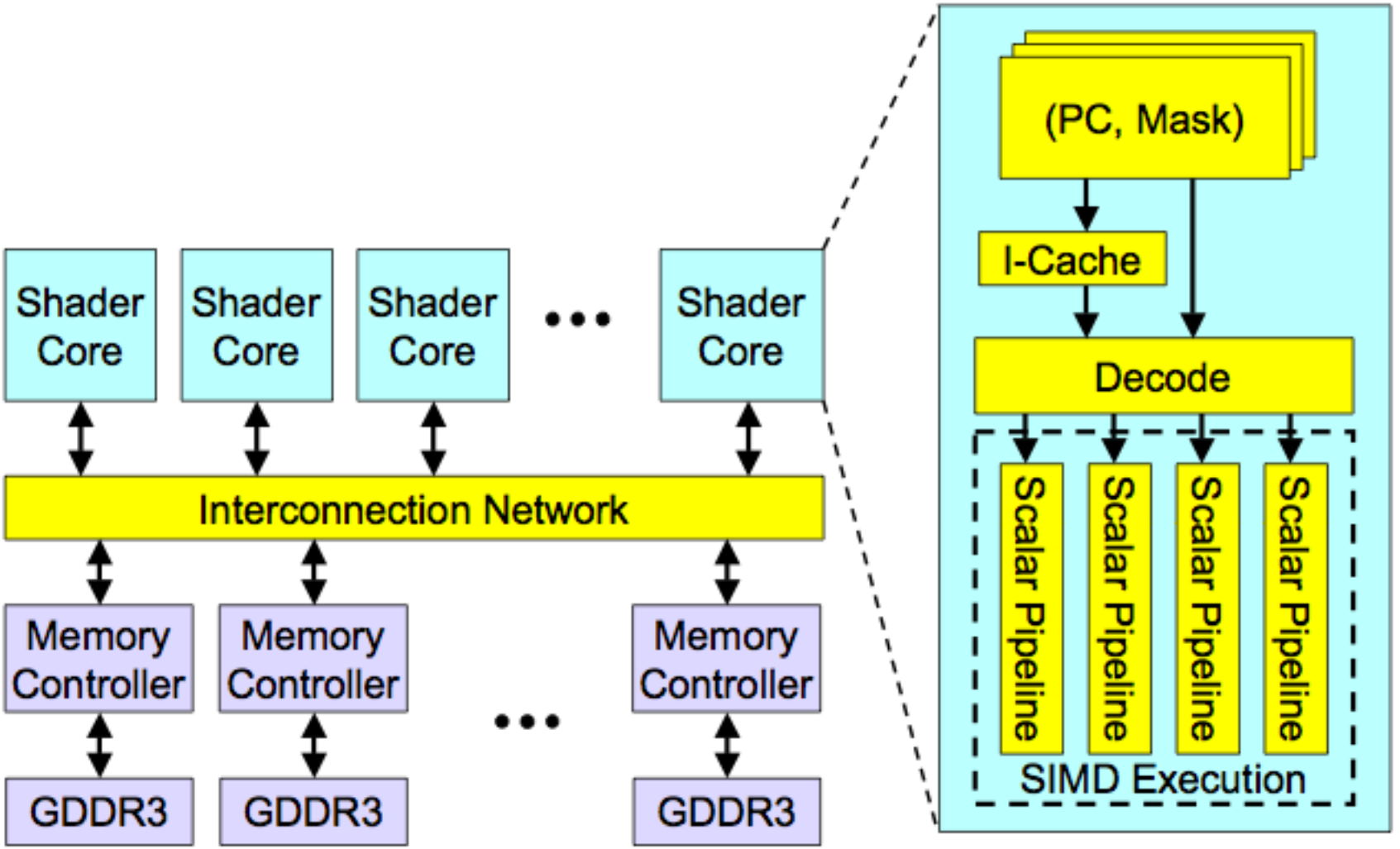


Parallel Computing:Computer Architecture:CPU vs GPU



1. Small caches
   - boost memory throughput
2. Very simple control
   - No Branch Predictors
   - No Data forwarding
3. Energy Efficient ALU
   - Many Long Latency but heavily pipelined for high throughput
4. Require massive number of threads to tolerate latencies
   - Threading logic
   - Thread state

Parallel Computing:Computer Architecture:CPU vs GPU



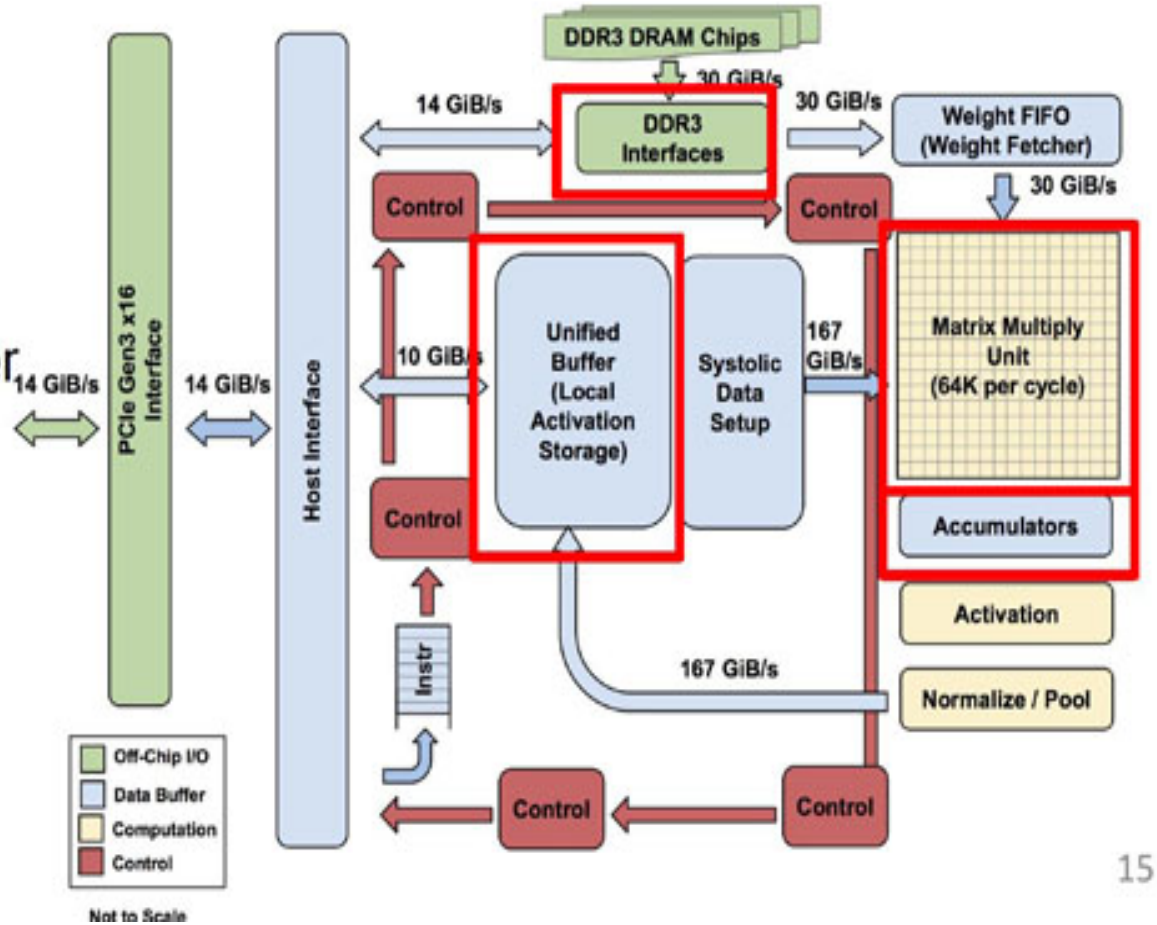
Conclusions

- Computationally dense processors (like GPUs) required
- Programmability
  - We don't know the algorithms of the future
- Lower precision
  - But not too low
  - Interesting algorithm/dataset engineering here
- We need better support for multi-GPU
  - E.g. Atomics between GPUs, collectives
  - Looking forward to NVLink

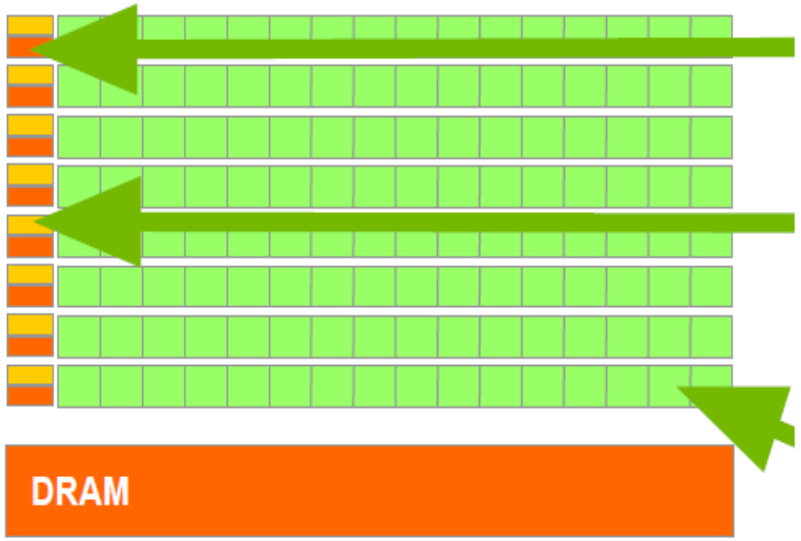
Google's TPU

- The Matrix Unit: 65,536 (256x256) 8-bit multiply-accumulate units
- 700 MHz clock rate
- Peak: 92T operations/second
  - 65,536 * 2 * 700M
- >25X as many MACs vs GPU
- >100X as many MACs vs CPU
- 4 MiB of on-chip Accumulator memory
- 24 MiB of on-chip Unified Buffer (activation memory)
- 3.5X as much on-chip memory vs GPU
- Two 2133MHz DDR3 DRAM channels
- 8 GiB of off-chip weight DRAM memory

TPU: High-level Chip Architecture

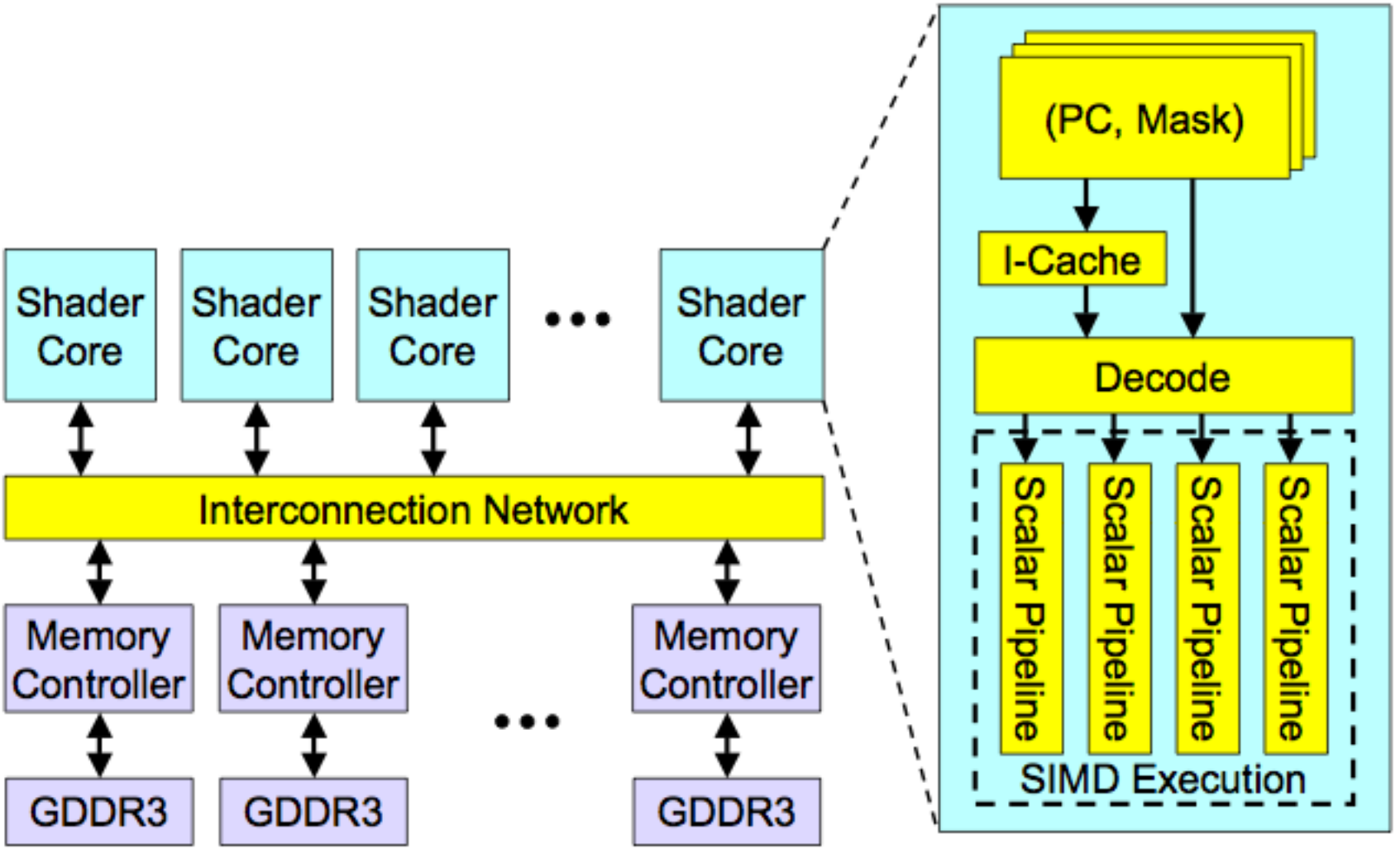


Parallel Computing:Computer Architecture:CPU vs GPU



1. Small caches
   - boost memory throughput
2. Very simple control
   - No Branch Predictors
   - No Data forwarding
3. Energy Efficient ALU
   - Many Long Latency but heavily pipelined for high throughput
4. Require massive number of threads to tolerate latencies
   - Threading logic
   - Thread state

Parallel Computing:Computer Architecture:CPU vs GPU



Conclusions

- Computationally dense processors (like GPUs) required
- Programmability
  - We don't know the algorithms of the future
- Lower precision
  - But not too low
  - Interesting algorithm/dataset engineering here
- We need better support for multi-GPU
  - E.g. Atomics between GPUs, collectives
  - Looking forward to NVLink

Conclusions

- Deep Learning is solving many hard problems
- Training deep neural networks is an HPC problem
- Scaling brings AI progress!

Conclusions

- Deep Learning is solving many hard problems
- Training deep neural networks is an HPC problem
- Scaling brings AI progress!

Agenda:

Deep Learning: Introduction

Foundation:Linear Regression

Deep Learning: Going Deep:Intuition

Deep Learning: Going Deep:Forward pass

Deep Learning: Going Deep:Back Prop

Deep Learning: Architectures

Deep Learning: Fintechs

Deep Learning: GPU and TPU

Deep Learning:Fintechs:Use cases

- SPOOFING SIMILARITY
  - The Spoofing similarity model identifies various forms of market abuse that involve false or misleading order activity known as spoofing.
  - Simple Spoofing
  - Spoofing with Layering
  - Spoofing with Vacuuming
  - Collapsing of Layers
  - Flipping
  - Spread Squeeze

Deep Learning:Fintechs:Use cases

- ABUSIVE MESSAGING DETECTION
  - The Abusive Messaging Detection model identifies patterns of disruptive or excessive order activity.
  - Quote Stuffing
  - Microburst Quote Stuffing
  - Malfunctioning Algos

Deep Learning:Fintechs:Use cases

- MOMENTUM IGNITION DETECTION
  - The Momentum Ignition Detection model identifies behavior designed to initiate rapid market movement at the expense of other participants.
  - Igniters and Directional Manipulation

Deep Learning:Fintechs:Use cases

- SUSPICIOUS PRICE MOVEMENT DETECTION
  - The Suspicious Price Movement model identifies executions during unusual market movements which may indicate abusive behavior or erroneous trades.
  - Aberrant Pricing

Deep Learning:Fintechs:Use cases

- PINGING AND PHISHING DETECTION
  - The Pinging and Phishing Detection model detects activity designed to take advantage of hidden volume at the expense of slower market participants.
  - Pinging and Phishing

Deep Learning:Fintechs:Use cases

- WASH TRADE DETECTION
  - The Wash Trade Detection model identifies executions with no change in beneficial ownership.
  - Wash Sales and Churning

Deep Learning:Fintechs:Use cases

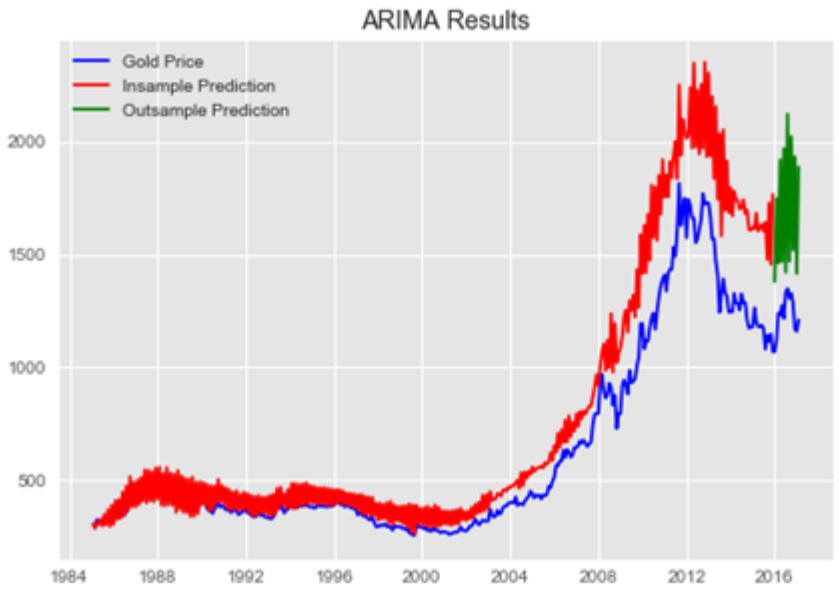
- CROSS TRADE DETECTION
  - The Cross Trade Detection model identifies potential cross trades without sufficient delay between order entries.
  - Crossing

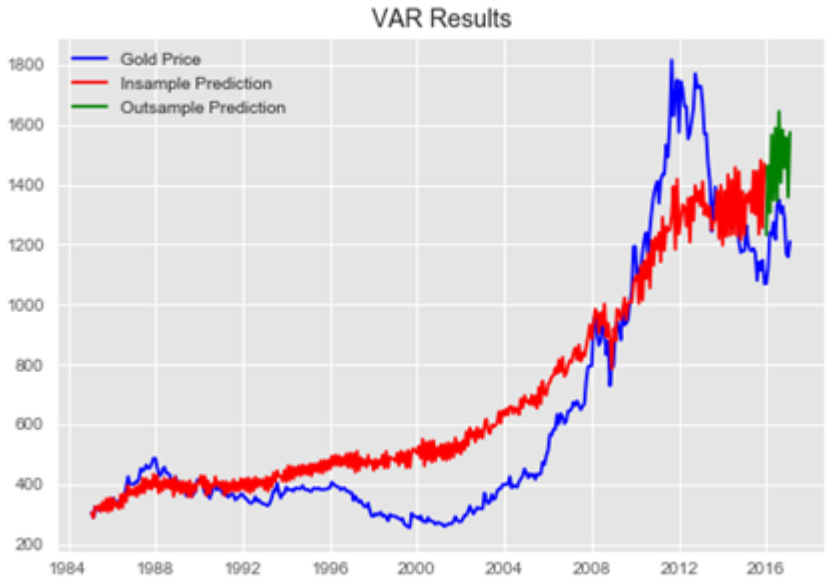
Deep Learning:Fintechs:Use cases

- CLOSING PERIOD ABUSE DETECTION
  - The Closing Period Abuse Detection model identifies orders placed with the intent to manipulate market price at or near the close.
  - Closing Period Abuse

Deep Learning:Fintechs:Use cases

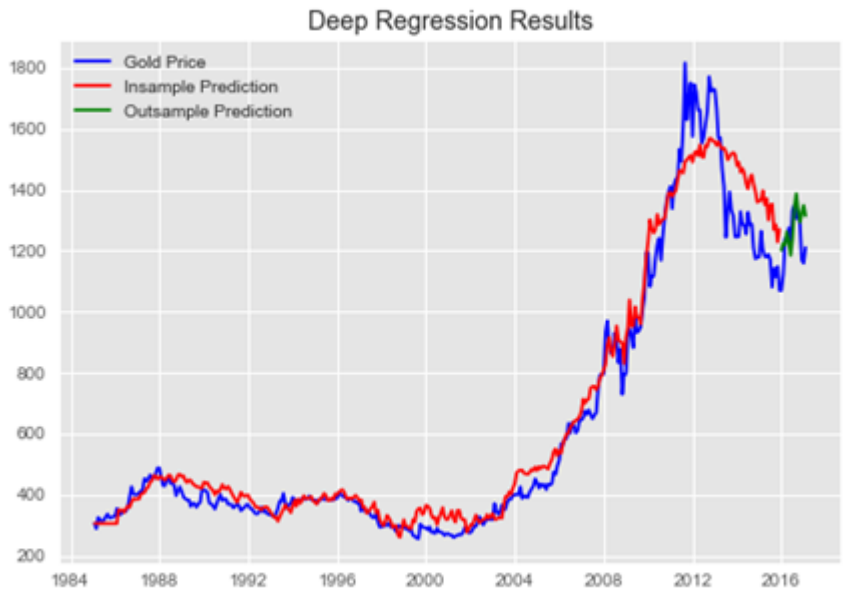
- Return Prediction
  - Taking the sample problem of predicting daily Gold Prices, we first look at the traditional methods.

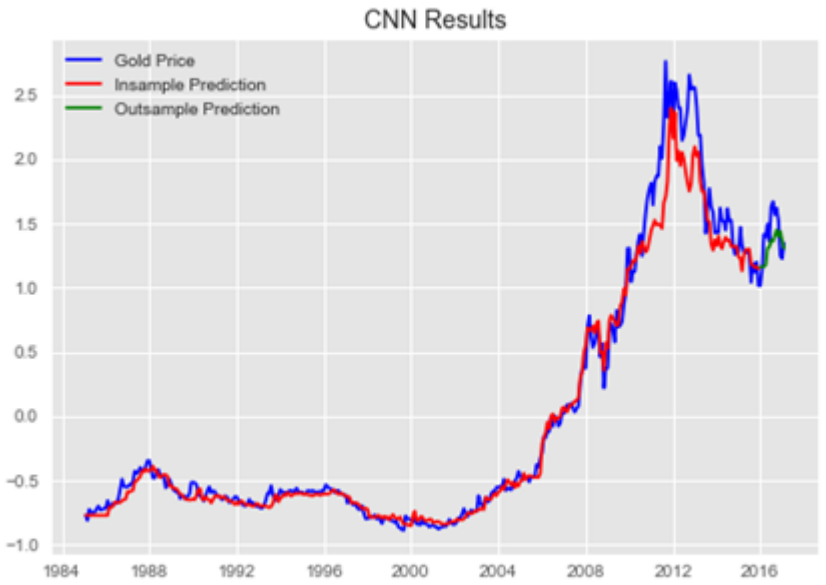




Deep Learning:Fintechs:Use cases

- Return Prediction
  - Taking the sample problem of predicting daily Gold Prices, we first look at the traditional methods.





Deep Learning:Fintechs:Use cases

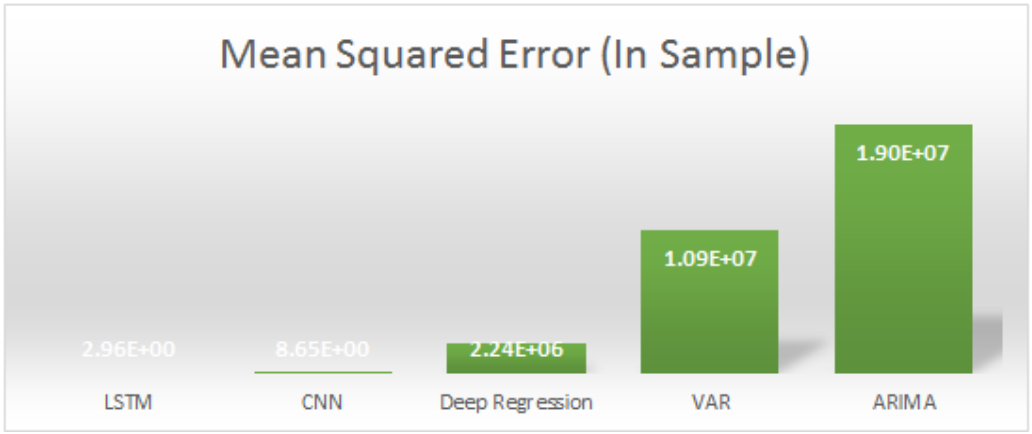
- Return Prediction
  - BOOM

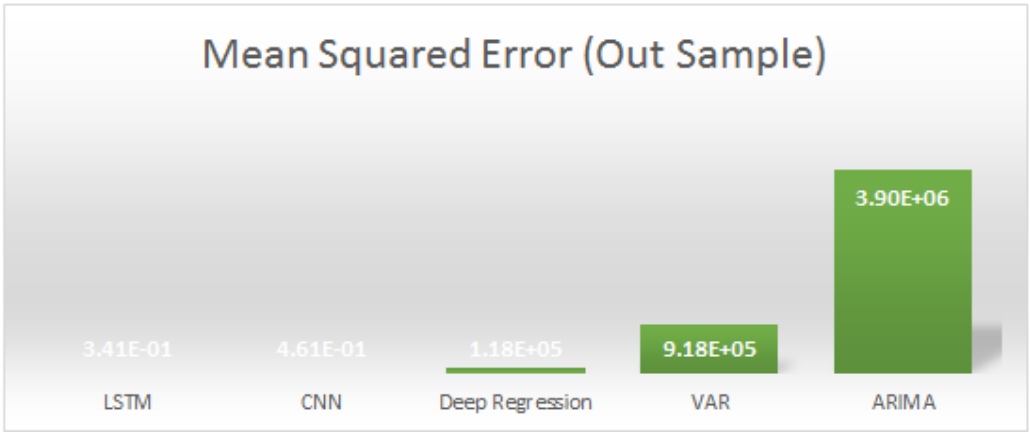
Deep Learning:Fintechs:Use cases

- Return Prediction
  - BOOM

Model	Mean Squared Error (In Sample)
LSTM	2.96E+00
CNN	8.65E+00
Deep Regression	2.24E+06
VAR	1.09E+07
ARIMA	1.90E+07

Model	Mean Squared Error (Out Sample)
LSTM	3.41E-01
CNN	4.61E-01
Deep Regression	1.18E+05
VAR	9.18E+05
ARIMA	3.90E+06





Deep Learning:Fintechs:Use cases

- Mortgage Risk
  - A deep learning model of multi-period mortgage risk and use it to analyze an unprecedented dataset of origination and monthly performance records for over 120 million mortgages originated across the Canada between 1995 and 2014.
  - Our nonparametric estimators of term structures of conditional probabilities of prepayment, foreclosure and various states of delinquency incorporate the dynamics of a large number of loan-specific as well as economic and demographic variables at national, state,county and zip-code levels.

How can we help fintechs?

- Discover regulatory and compliance issues before violations happen, preventing fines and enforcement investigations.
- Advanced pattern recognition detects activity likely to be high-risk, flagging hidden threats well before disaster strikes.
- Examines profitability, cost drivers, and industry-specific risk factors to incentivize behaviors and improve firm performance.
- Finds outlier events, by understanding historical trends, environmental context and correlation within firm activity.
- We show financial institutions the true power of artificial intelligence by delivering clearly prioritized calls to action. This empowers decision makers to solve challenging business problems.