GTC 2018
Silicon Valley, California

Predictive Learning of Factor Based Strategies using Deep Neural Networks for Investment and Risk Management

Yigal Jhirad and Blay Tarnoff
March 27, 2018

GTC 2018: Table of Contents

I. Deep Learning in Finance

- Forecasting Factor Regimes
- Machine Learning Landscape
- Deep Learning + Neural Networks
- Neural Networks – ANN, RNN, LSTM
- Optimization

II. Parallel Implementation

III. Summary

IV. Author Biographies

GTC 2018: Deep Learning

- **Investment & Risk Management**
  - Forecast Market Returns, Volatility Regimes, Factor Trends, Liquidity, Economic Cycles
  - Big Data including Time Series Data, Interday, and Intraday
  - Neural Networks: Black Box/Pattern Recognition
  - Complement existing quantitative and qualitative signals

- **Challenges include state dependency and stochastic nature of markets**
  - Time series
  - Overfitting/Underfitting
  - Stochastic Nature of Data

GTC 2018: Factor Analysis

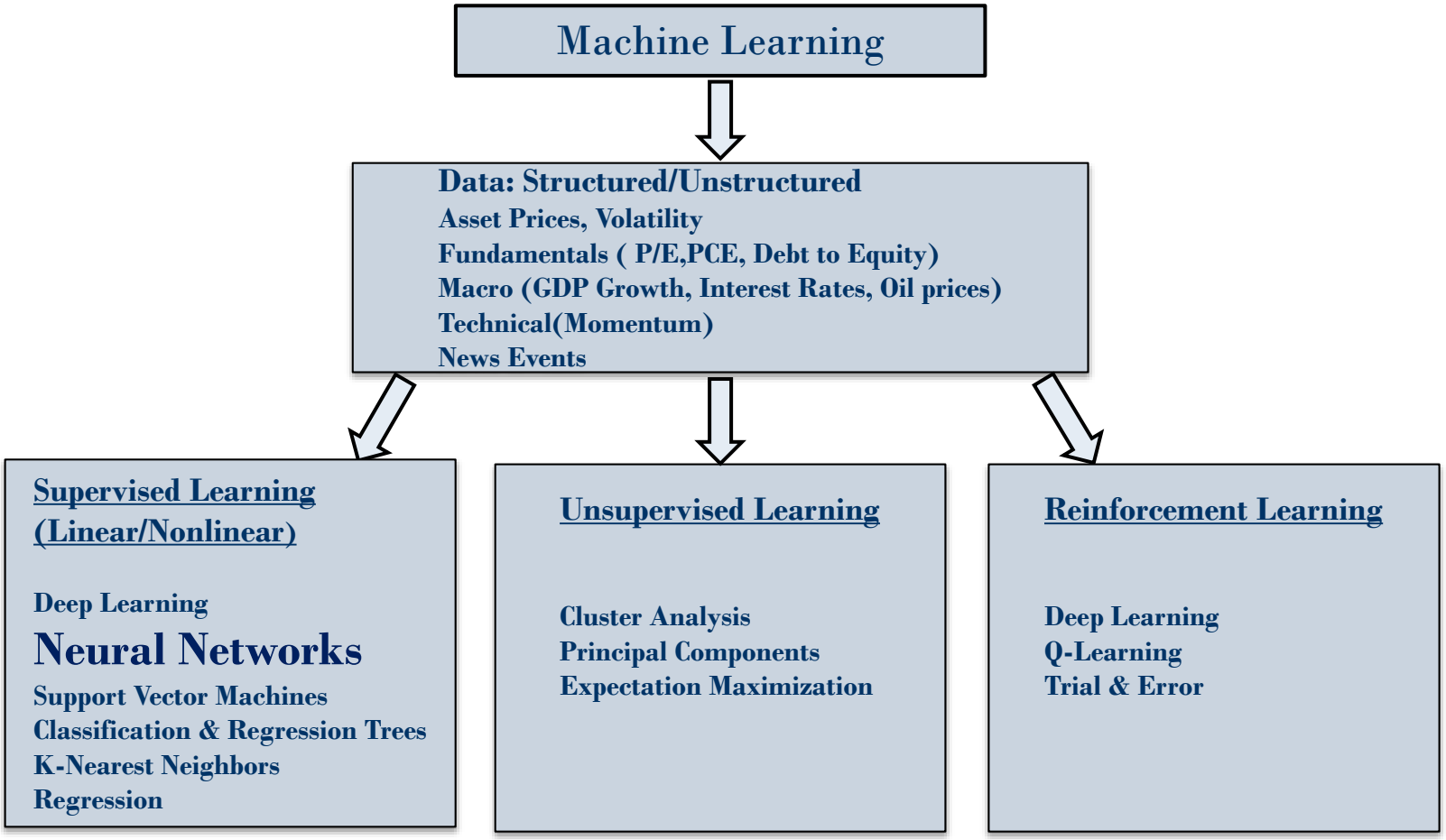
- **Factor Analysis**
  - Identify factors that are driving the market and predict relative factor performance
  - Establish a portfolio of sectors or stocks that benefits from factor performance
  - Align risk management with forecasts of volatility
- **Identifying and Assessing factors driving performance**
  - Look at factors such as Value vs. Growth, Large Cap vs. Small Cap, Volatility

Best Performing Factors
Cash/Assets
EPS Growth
Momentum
Price/Book
Market Cap (High-Low)

Worst Performing Factors
Dividend Yield
Debt/Equity
Capex/Assets
Dividend Payout
Volatility(High-Low)

Period:12/2016-12/2017

Artificial Intelligence



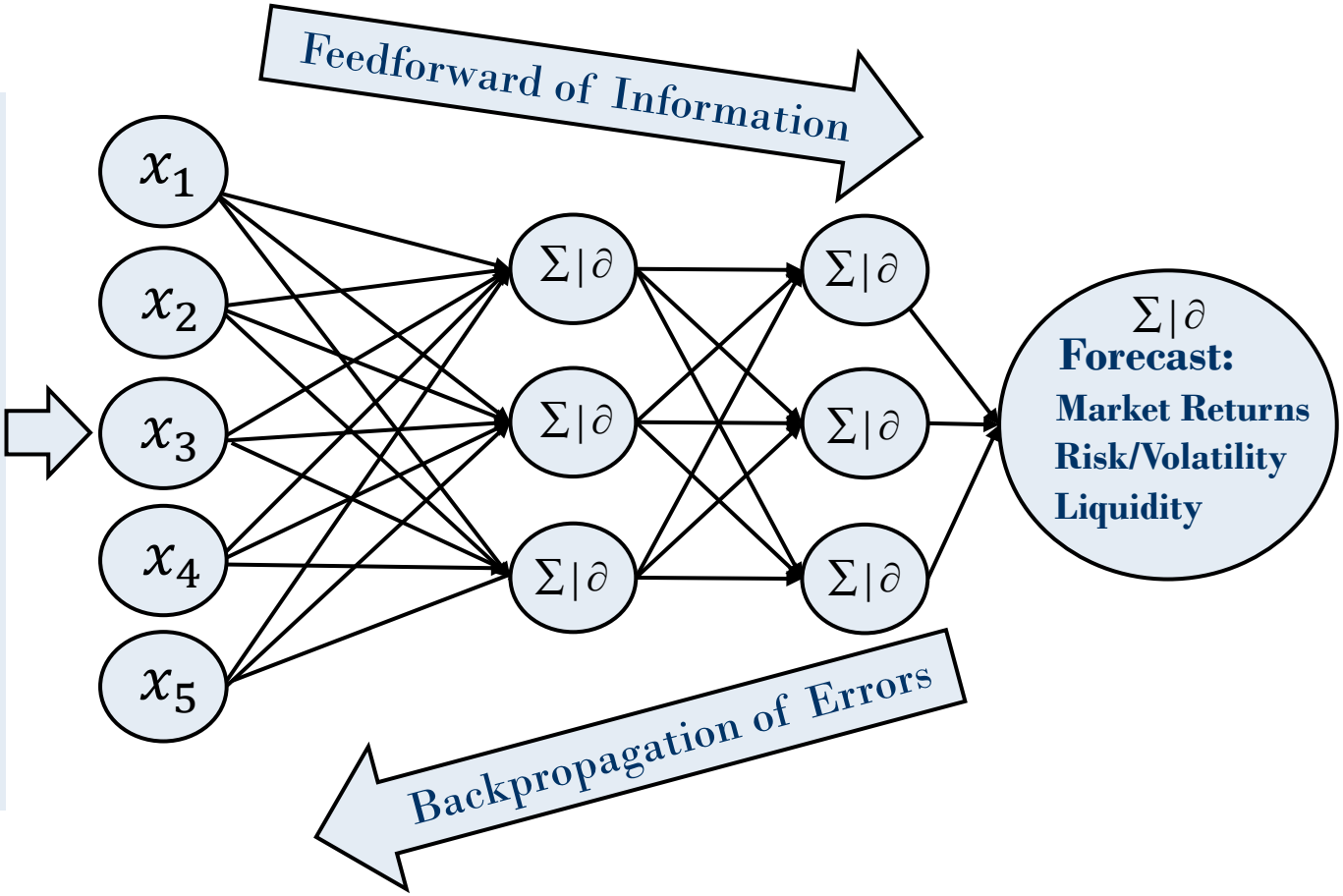
Jhirad, Yigal (2017)

Supervised Learning: Neural Networks

Feature(Factor)Identification & Regularization

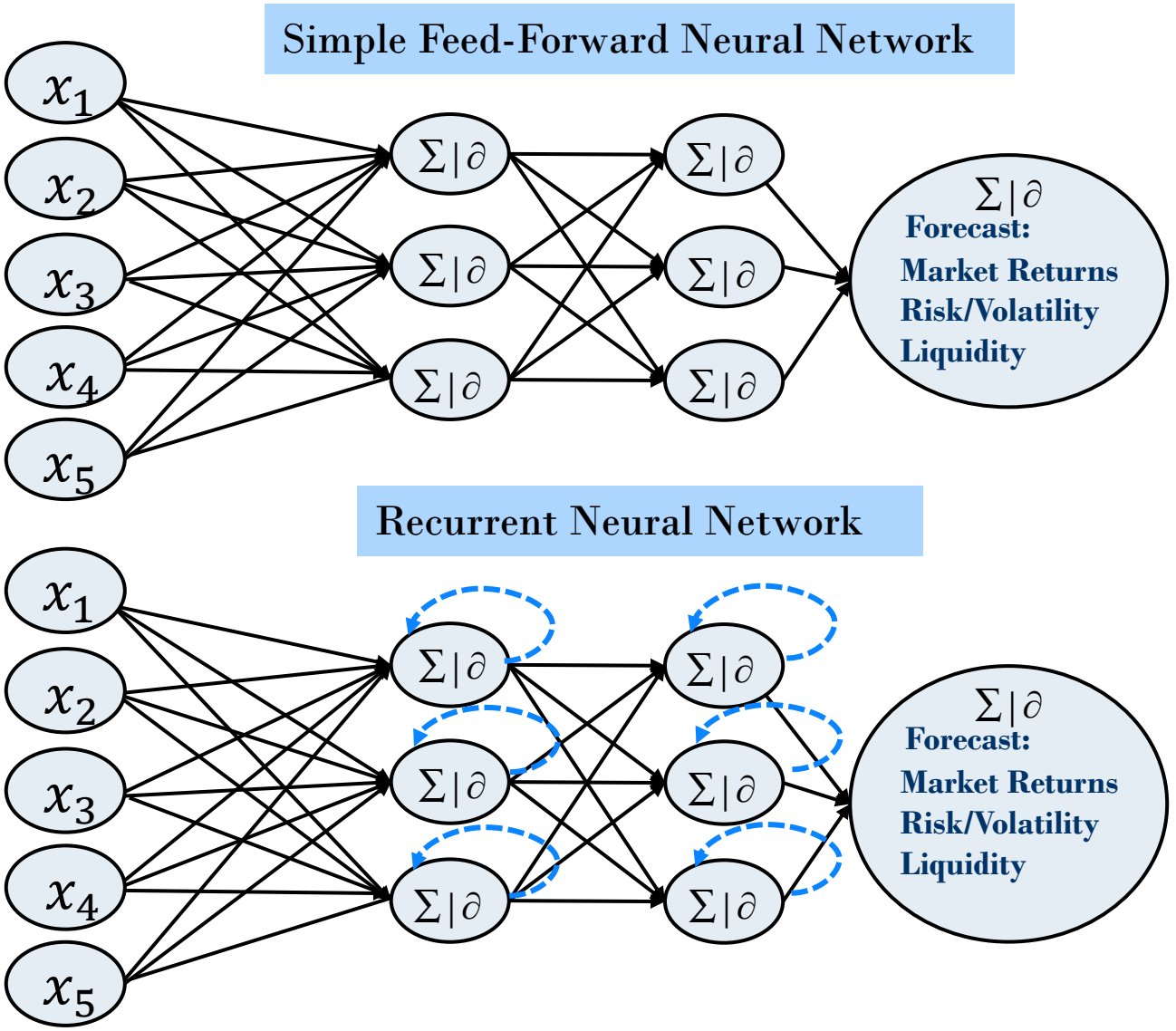
Inputs:

Fundamental/Macro/Technical
Price/Earnings
Momentum/RSI
Realized & Implied Volatility
Value vs Growth
GDP Growth/Interest Rates
Dollar Strength
Credit Spreads



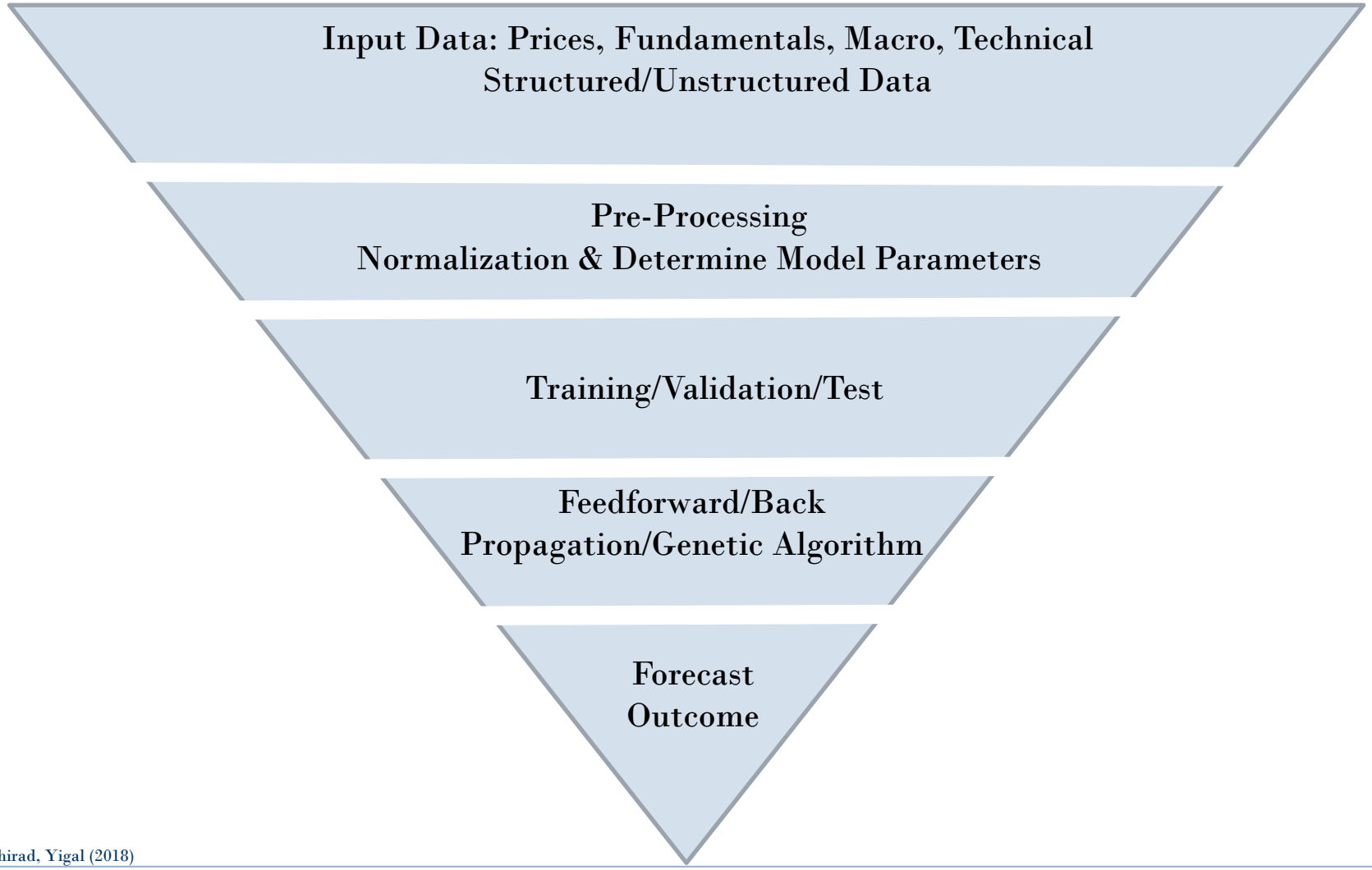
Jhirad, Yigal (2017)

Supervised Learning: Neural Networks



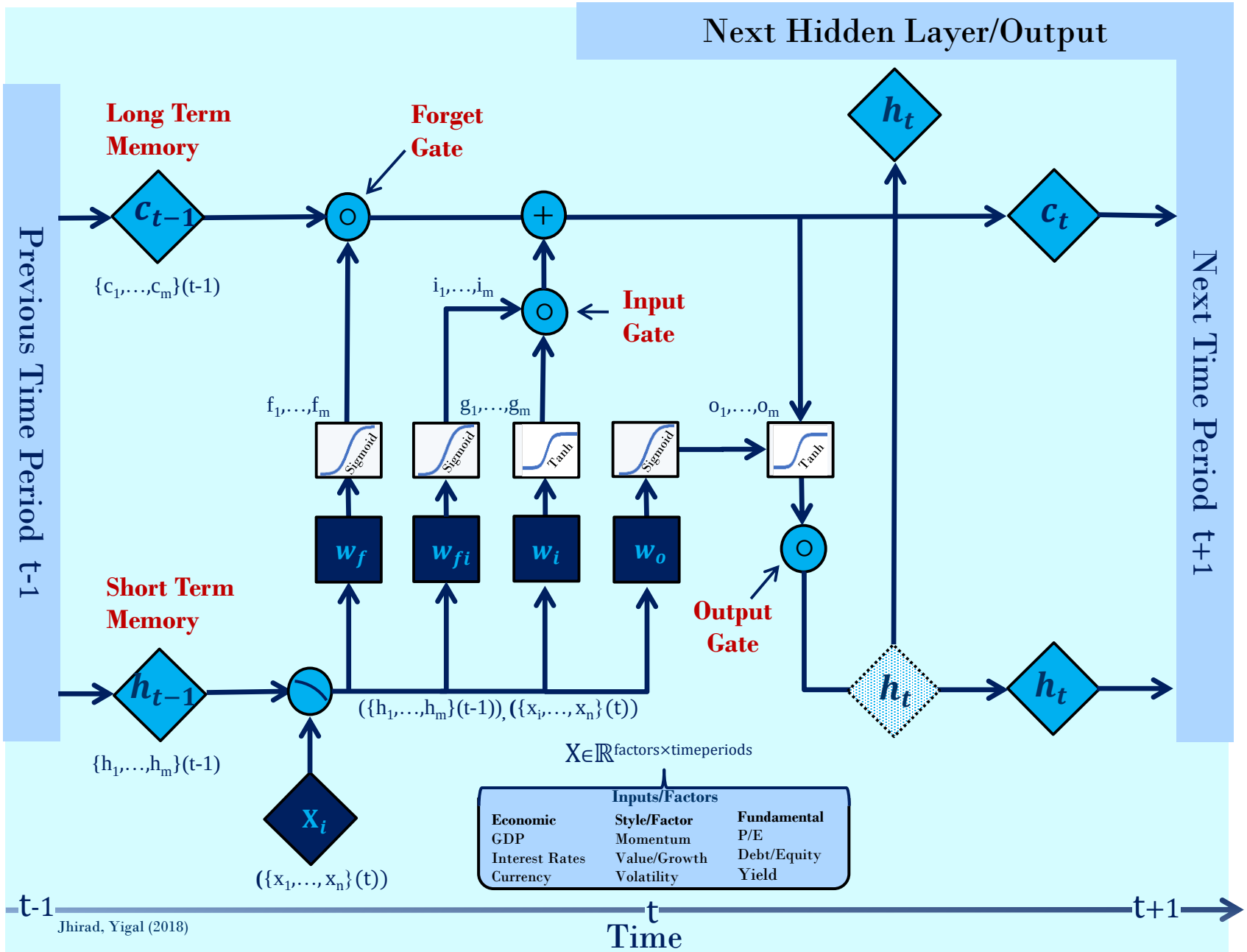
Jhirad, Yigal (2018)

Neural Network Work Flow



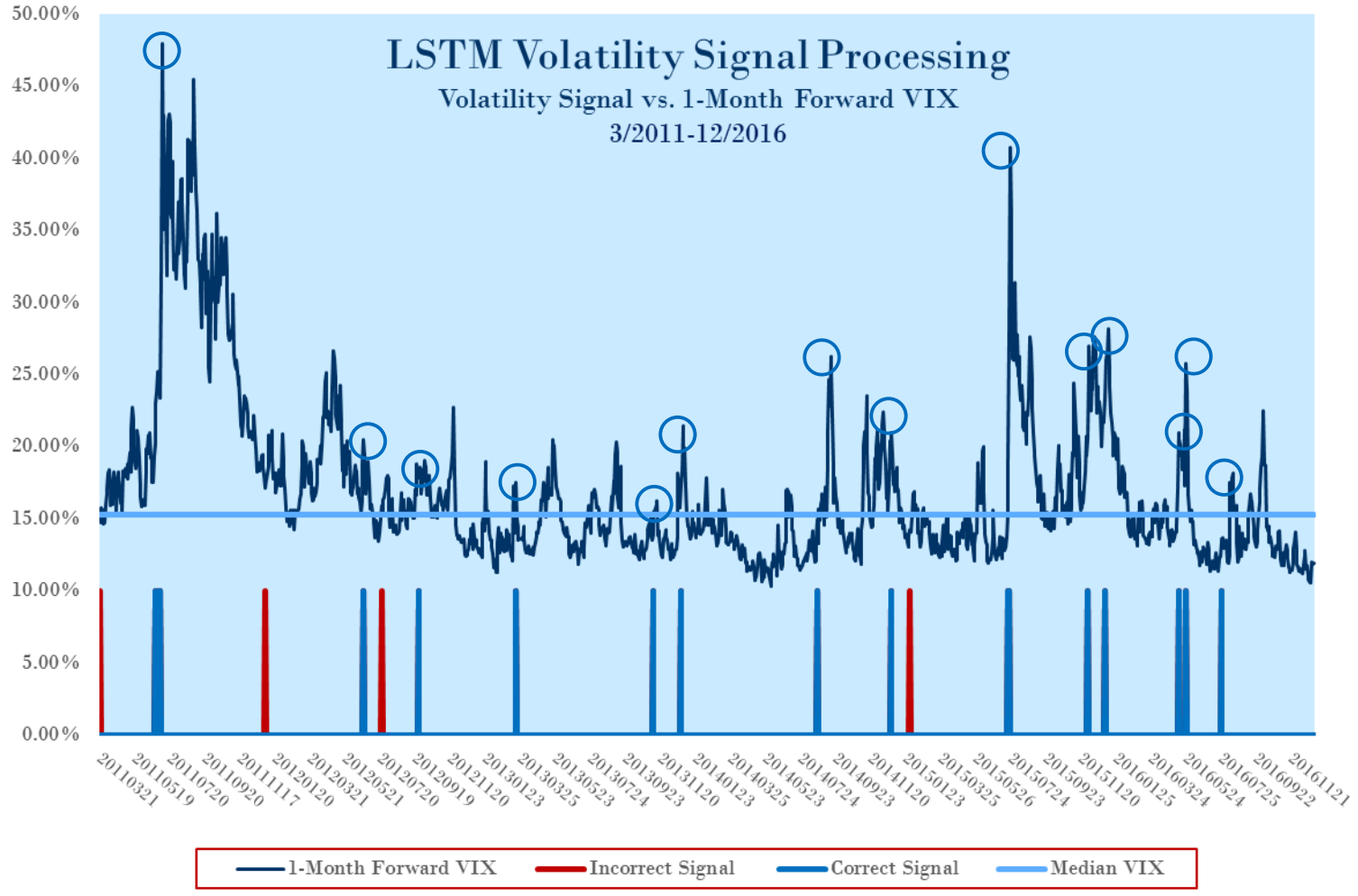
Jhirad, Yigal (2018)

GTC 2018: LSTM



Jhirad, Yigal (2018)

GTC 2018: Predicting Volatility Regimes with LSTM

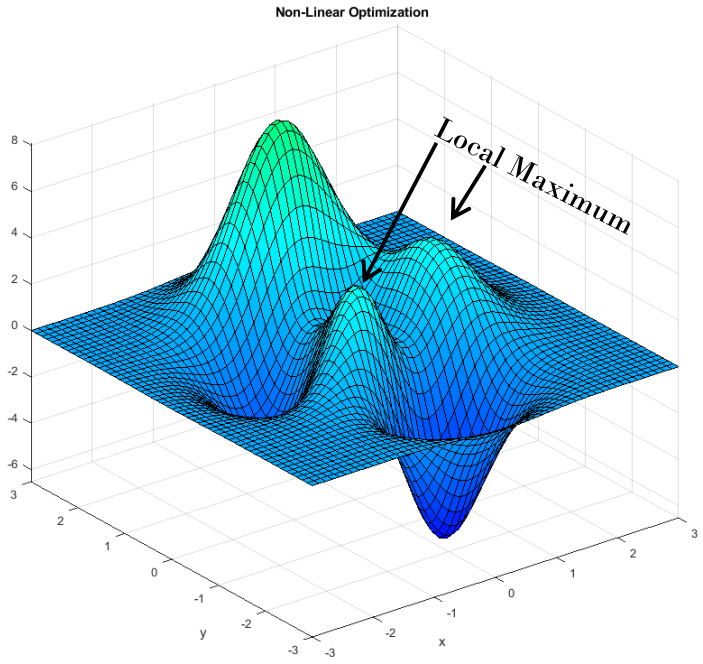


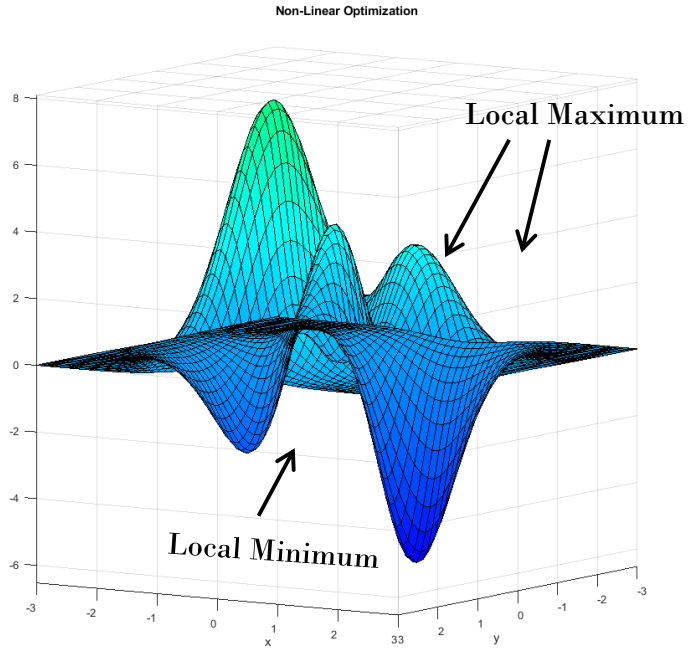
GTC 2018: Neural Networks

- **Neural Networks**
  - Feed-Forward vs. Recurrent Neural Networks
  - LSTM captures the temporal nature of financial data
  - Complement existing quantitative and qualitative signals
- **Advantages**
  - Captures non-linearity that are prevalent in financial data
  - Time Sequencing, Pattern Recognition
  - Modularity
  - Parallel Processing
- **Considerations**
  - Black Box
  - Overfitting/Underfitting
  - Optimization/Local Minima

GTC 2018: Genetic Algorithms

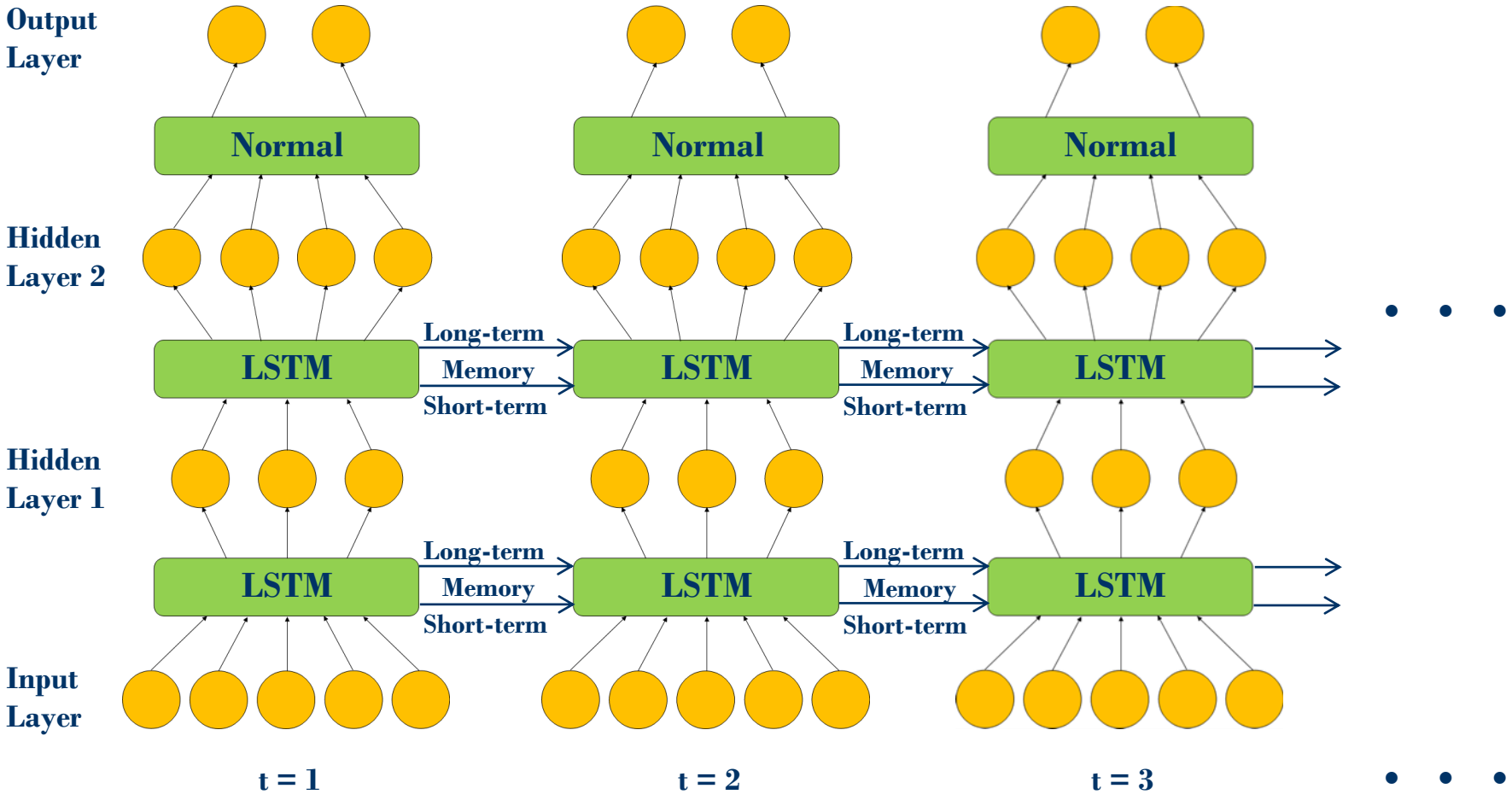
- Gradient Descent may not be efficient
- Local Minimums pose a challenge
- Genetic Algorithms complement traditional optimization techniques
- Apply the computational power within CUDA to create a more robust evolutionary algorithm to drive multi-layer Neural Networks





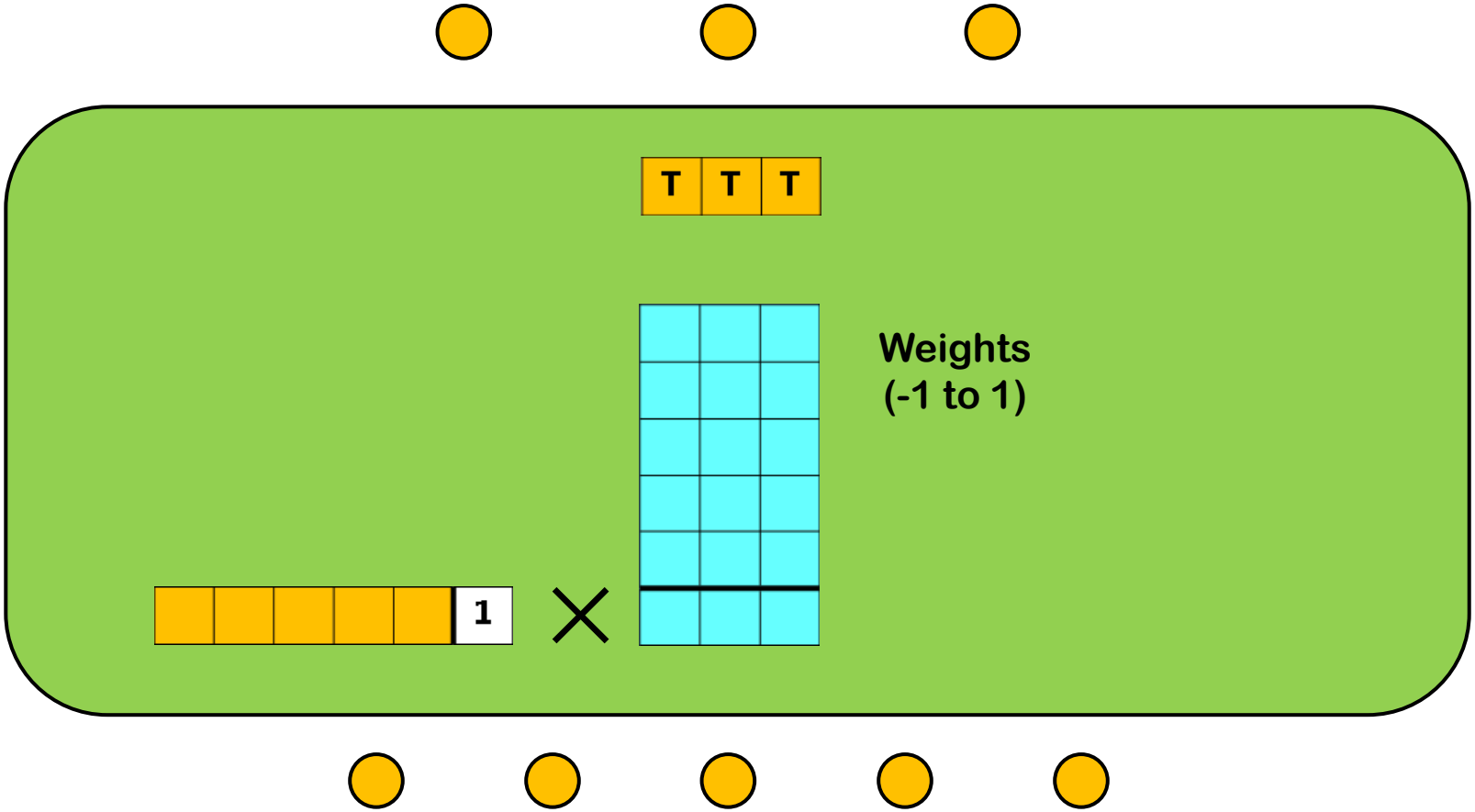
Neural Architecture

Feed forward: 4 layers: input, 2 hidden, output



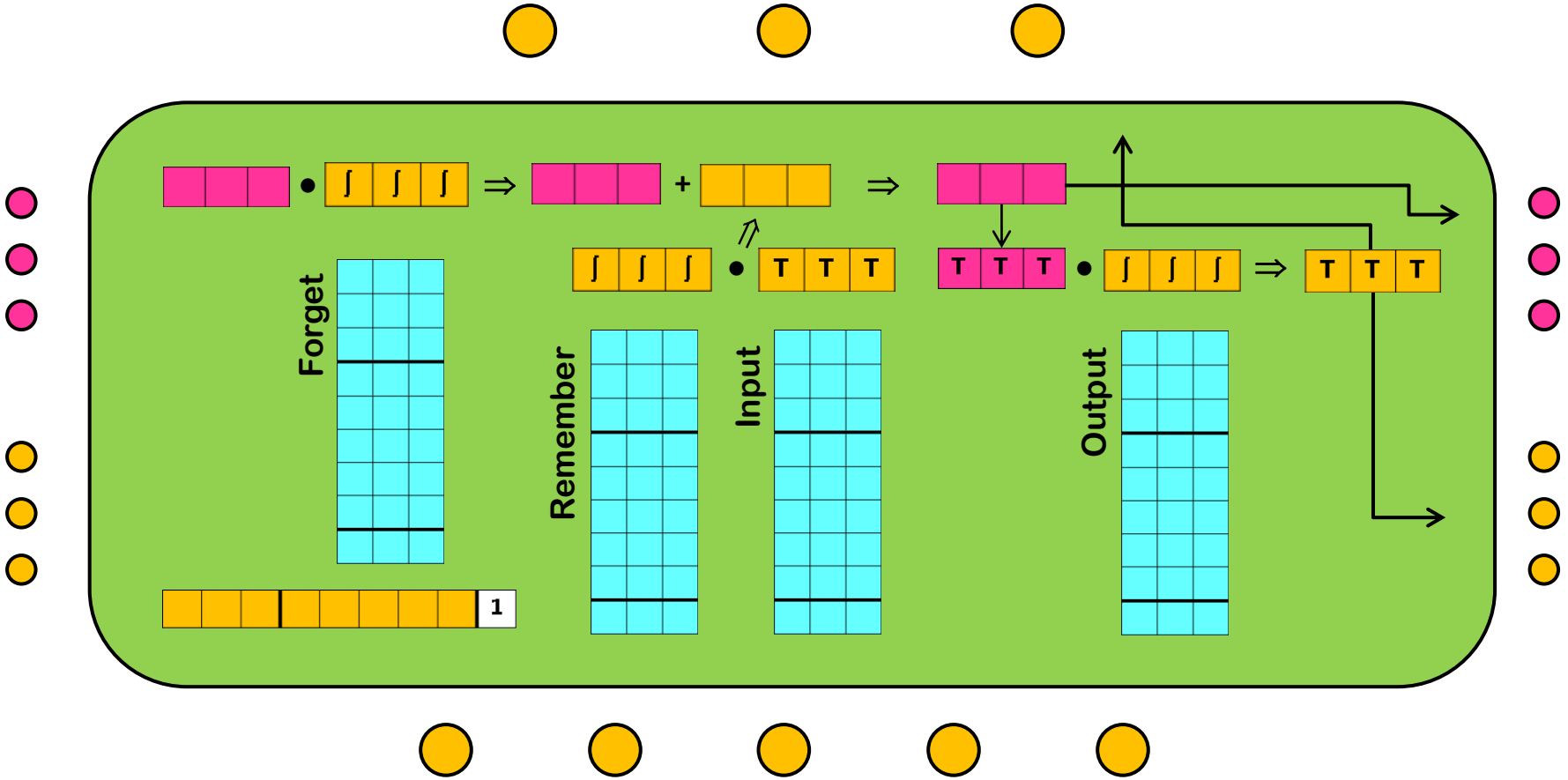
Neural Architecture

Transitional layer: Normal



Neural Architecture

Transitional layer: LSTM



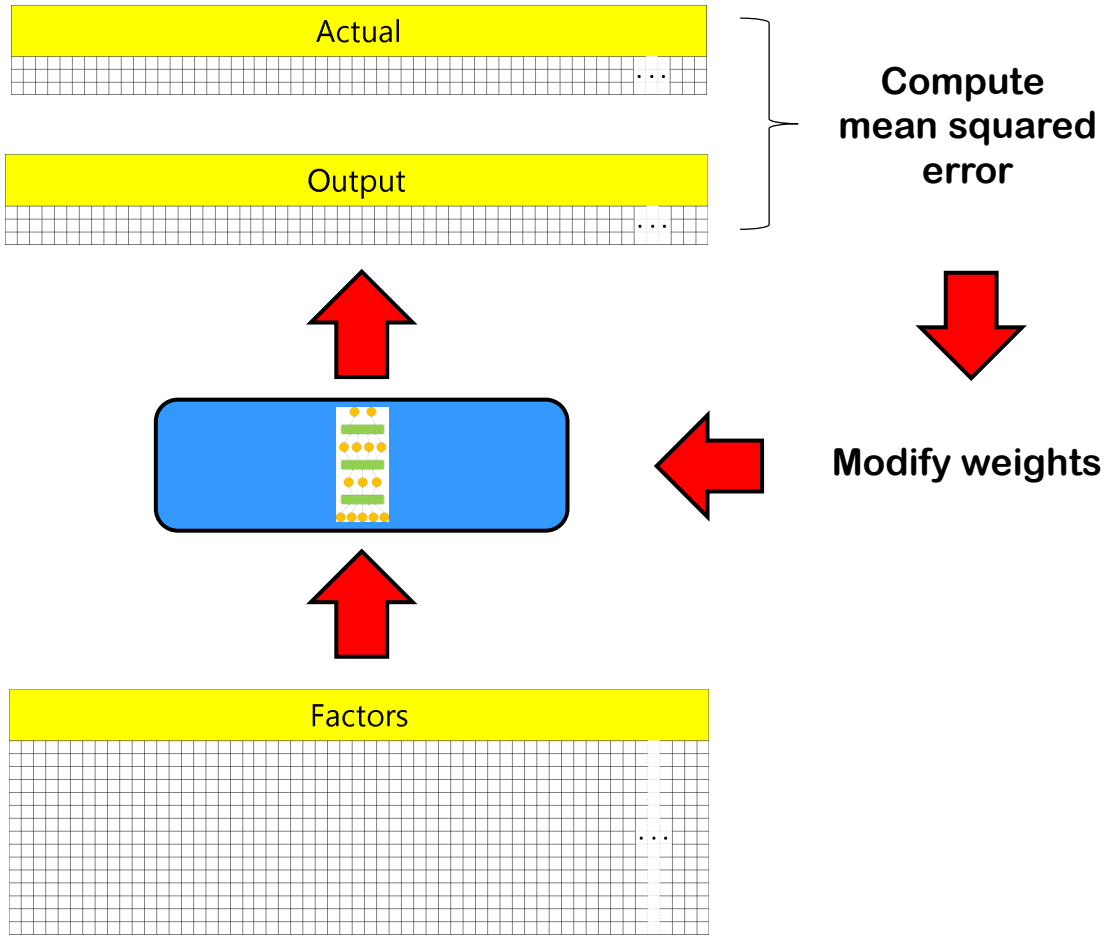
Training

Supervised

- **4 matrices of weights in each LSTM layer plus one in normal layer equals 9 weight matrices**
- **Goal of training is to find weights to populate those matrices that convert the input values to output values which most accurately reflect reality**
- **Output values are computed from input values period-by-period and compared to actual values to yield mean squared error**
- **Weight matrices are modified and process is re-run repeatedly until mean squared error ceases to improve**

Training

Supervised: feed forward



Training

Genetic algorithm: terminology

- **Gene: one matrix of weights**
- **Organism: a set of 9 weight matrices**
- **Fitness: the mean squared error generated by an organism over the timeframe**
- **Breeding population: set of organisms that have the lowest mean squared errors**
- **Mating: process of splicing the corresponding genes of two organisms in randomly selected locations to produce new organisms**
- **Mutation: the re-setting of randomly selected weights to new random values during the mating process**
- **Generation: one iteration in which the breeding population mates and produces offspring**

Training

Genetic algorithm

- **Create a set of organisms (population) by creating a set of weight matrices for each, populated with random weights**
- **Evaluate the fitness of each organism by feeding forward the input matrix through the neural network period-by-period and comparing the outputs to the matrix of actual values, yielding a mean squared error**
- **Rank the organisms by their mean squared errors**
- **Select mates for the fittest organisms and produce offspring: two new organisms**
- **Add the offspring to the population, evaluate their fitness and re-rank the population**
- **Drop the least fit organisms from the population to maintain the population size**
- **Repeat the previous three steps until no offspring survive the previous step for some number of generations**
- **Fittest organism is now a trained neural network**

Training

Genetic algorithm: mating

- **For each member of the breeding population, randomly select one of the remaining members of the population as a mate**
- **For each weight matrix (gene), randomly select a splice point between 1 and half the size of the matrix**
- **Swap the section of each mate's matrix that begins at the splice point and ends at twice the splice point with the other mate, yielding two offspring**
- **Randomly pick a set number of weights and change them to new random values (mutate)**

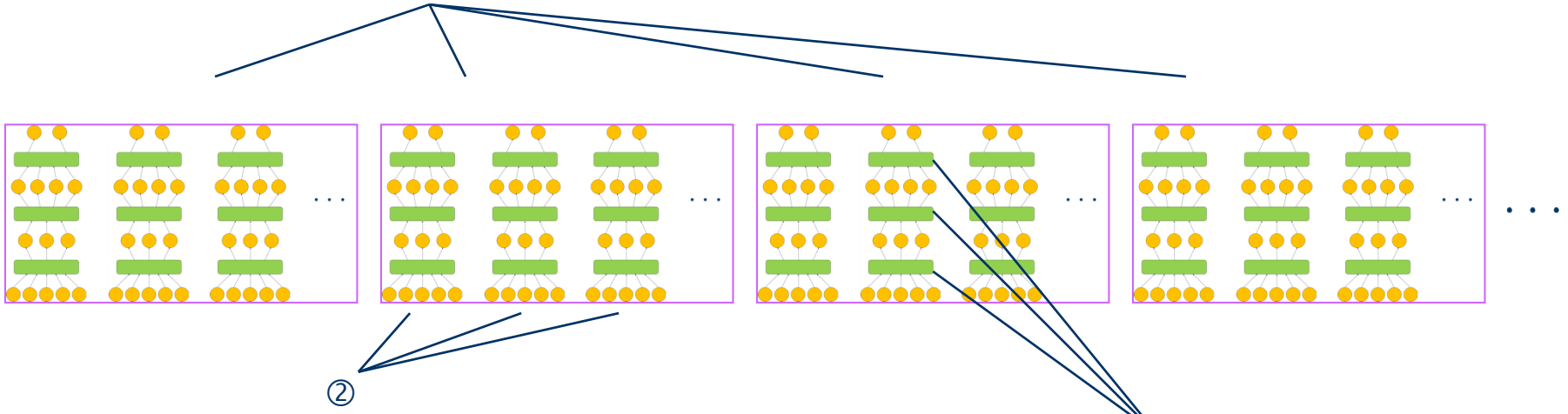
CUDA Architecture

Genetic algorithm: parallelism

Network composed entirely of non-recurrent layers enables 3 levels of parallelism

①

Each organism can be run in parallel at grid level



②

Each period can also be run in parallel at grid level, since periods are independent

③

Each matrix multiplication can be run in parallel at thread block level

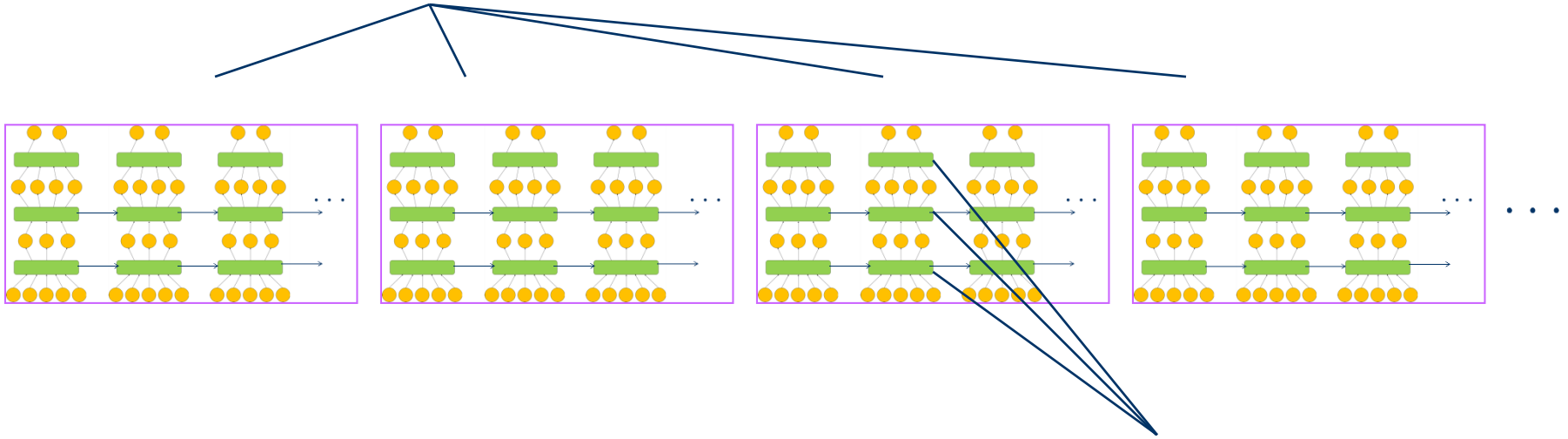
CUDA Architecture

Genetic algorithm: parallelism

Network that contains a recurrent layer loses period-level parallelism at that layer

①

Each organism can be run in parallel at grid level



②

Each matrix multiplication can be run in parallel at thread block level

CUDA Architecture

Genetic algorithm

- **Create population: For each of the 9 weight matrices:**
  - **Generate enough random numbers to populate all the organisms**
  - **For each organism, 2D pitch copy the random numbers**
  - **Launch grid of one block per organism to convert the random numbers to weights (good enough)**
- **Evaluate initial population: Launch grid of one block per organism to evaluate its fitness, each block doing the following:**
  - **For each period, feed forward the period's factors through the network and sum the squares of the differences between the output and the period's actual values**
  - **Write resulting mean squared error and final 4 long and short-term memory vectors to global**
- **Rank initial population by their mean squared errors**

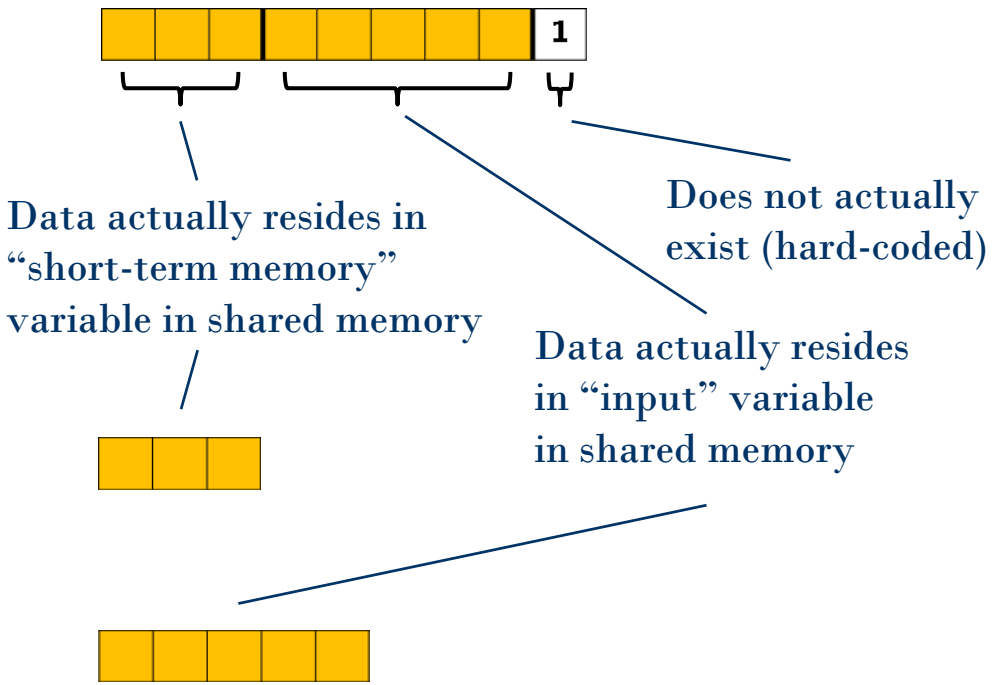
CUDA Architecture

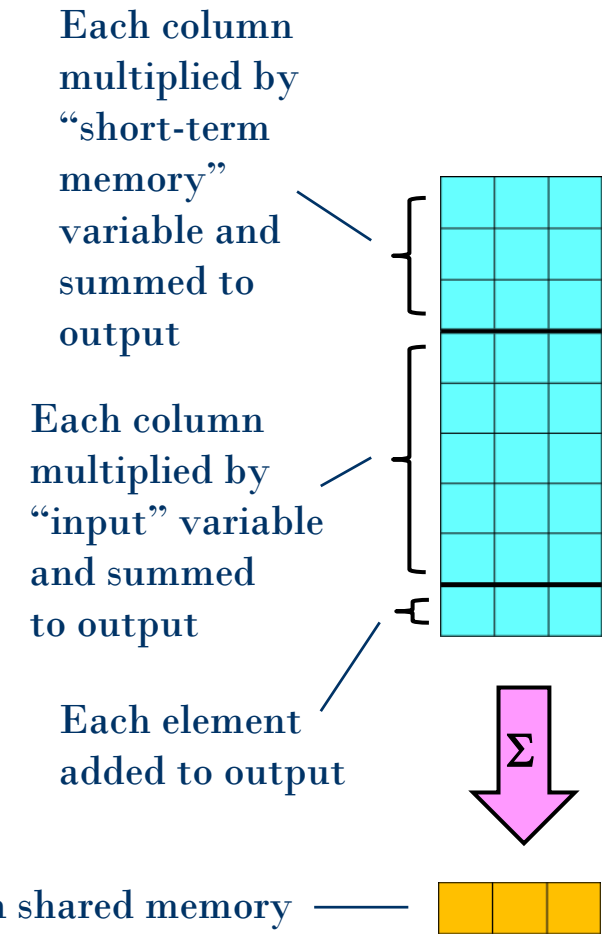
Genetic algorithm

- **Generational loop: Continues until some number of generations pass in which no offspring ranks better than the least fit organism:**
  - **Prepare to mate: Generate enough random numbers for each weight matrix of each breeding organism to select a mate, select a gene splice location, select which weights to mutate, and produce the mutated weights**
  - **Mate: For each of the 9 weight matrices, launch grid of one block per breeding organism that randomly selects a mate organism, swaps a randomly selected section of their weights, mutates some weights, and writes those resulting weight matrices in place of two of the lowest ranked organisms**
  - **Evaluate offspring: Launch grid of one block per breeding organism to evaluate its fitness, as in step prior to generational loop, writing the resulting mean squared errors and memory vectors in place of two of the lowest ranking organisms**
  - **Re-rank the population by their mean squared errors**
- **Write the weight matrices and memory vectors of the fittest organism to the host**

CUDA Architecture

Neural network: weight matrix structure

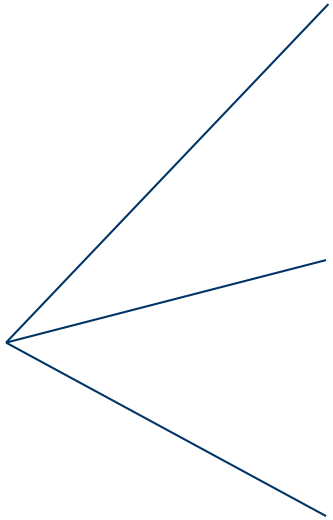


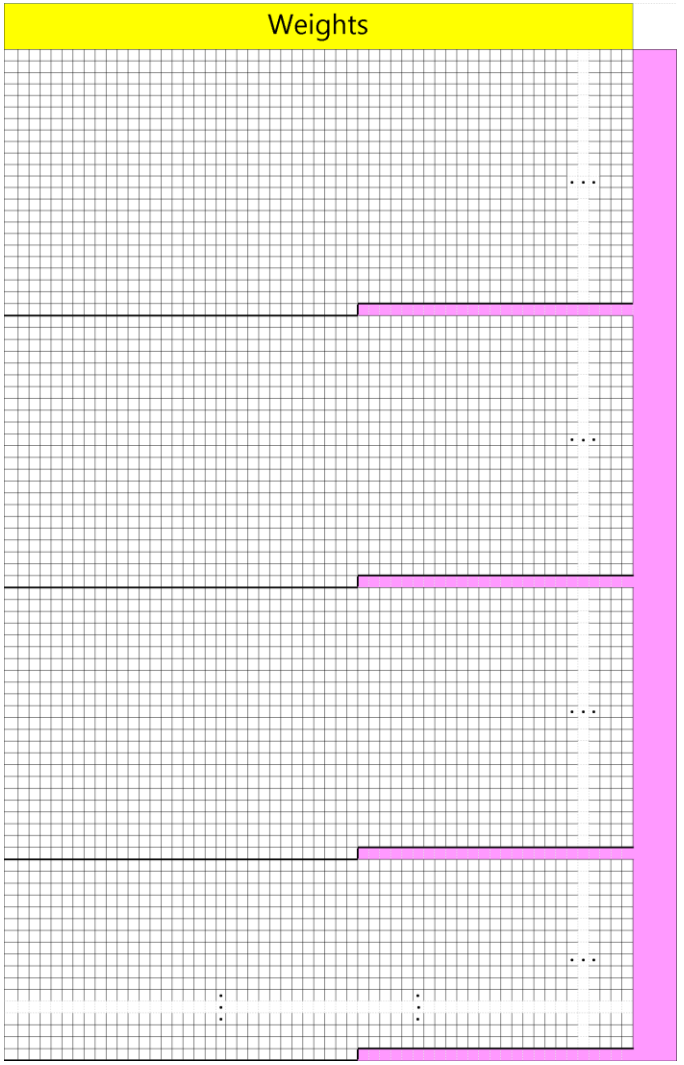


CUDA Architecture

Neural network: weight matrix structure

Organisms





CUDA Architecture

Neural network: weight matrix structure

Thread block size: 128 (binary number, multiple of warp size)

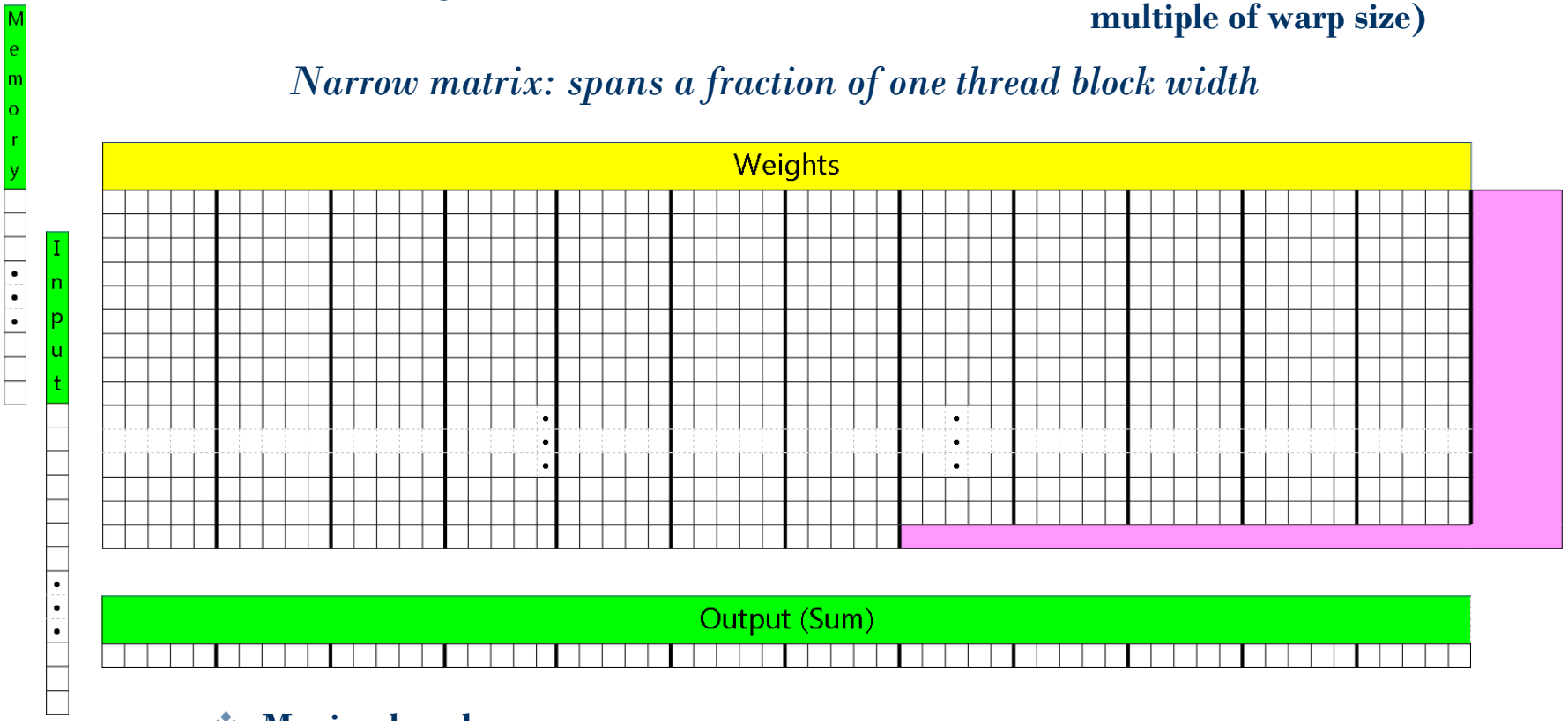
Wide matrix: spans many thread block widths

CUDA Architecture

Neural network: weight matrix structure

Thread block size: 128 (binary number, multiple of warp size)

Narrow matrix: spans a fraction of one thread block width



- **Maximal coalescence**
  - **No bank conflicts (broadcast, except in rare circumstances)**
    - **No __syncthreads necessary (until final summation of narrow matrix)**

CUDA Architecture

Neural network: matrix multiplication simplified code

```
npk = number of logical rows of weight matrix per block (always 1 for wide matrices)
nk = total number of blocks, i.e. rows of Weights (last block will usually be short for narrow matrices)
nO = # output neurons (= # elements in Memory vector, = logical width of weight matrix)
nI = # input neurons (= # elements in Input vector)

// Multiplication of logical input vector by logical weight matrix
k: loop vertically, i.e. by one block of logical matrix rows at a time
    o: loop horizontally (strided), i.e. by one thread block width at a time, starting at threadIdx.x
        i = k * npk + o / nO                    // index to input vectors
        if i < nO:                  n = Memory[i]
        else if i < nO + nI:        n = Input[i - nO]
        else:                       n = 1
        Output[o] = fmaf(Weights[k][o], n, Output[o])

// Final summation of Output
s: loop by diminishing strides (start at nO * blockDim.x / 2 and halve on each loop until 0)
    if s < npk * nO && s * 2 > warpSize:    __syncthreads
    if threadIdx.x + s < npk * nO:
        Output[threadIdx.x] = fadd(Output[threadIdx.x], Output[threadIdx.x + s])])
```

Note that the outer (k) loop is trivial for wide matrices and the inner (o) loop is trivial for narrow

Note that the second term is always 0 for wide matrices, in which case i = k, the outer loop counter

Summary

- **Utilize an LSTM Neural Network to identify market regimes**
- **Propose an Augmented LSTM Process that can help drive deep learning by identifying appropriate factors across market regimes**
  - Enhance construction by utilizing Optimization with Constraints function instead of penalty function
  - Utilize Genetic algorithms
- **CUDA leverages GPU Hardware providing computational power to drive optimization algorithms and Deep Learning**
- **Application in Investment and Risk Management**

Author Biographies

- **Yigal D. Jhirad,** Senior Vice President, is Director of Quantitative and Derivatives Strategies and a Portfolio Manager for Cohen & Steers' options and real assets strategies. Mr. Jhirad heads the firm's Investment Risk Committee. He has 30 years of experience. Prior to joining the firm in 2007, Mr. Jhirad was an executive director in the institutional equities division of Morgan Stanley, where he headed the company's portfolio and derivatives strategies effort. He was responsible for developing, implementing and marketing quantitative and derivatives products to a broad array of institutional clients, including hedge funds, active and passive funds, pension funds and endowments. Mr. Jhirad holds a BS from the Wharton School. He is a Financial Risk Manager (FRM), as Certified by the Global Association of Risk Professionals.

- **Blay A. Tarnoff** is a senior applications developer and database architect. He specializes in array programming and database design and development. He has developed equity and derivatives applications for program trading, proprietary trading, quantitative strategy, and risk management. He is currently a consultant at Cohen & Steers and was previously at Morgan Stanley.