Deep learning and macro finance

Goutham Gopalakrishna

Rotman School of Management, University of Toronto

Princeton Initiative 2023

September 9, 2023

Part-I: Introduction

Introduction

- The basic idea of machine learning goes back to Rosenblatt (1958) who introduced the idea of perceptron
- The progress halted during the 1990s
- Forces behind the revival
  - Big data
  - Cheap computational power
  - Advancements in algorithms
- Popularity in industry: packages in Python, Tensorflow, Pytorch etc.
- Strong community support for packages $\implies$ better tools in the future
- Coding and compiling deep learning algorithms is easy thanks to the rich ecosystem provided by Pytorch, Tensorflow, Keras etc.

Deep learning introduction

- The goal is to approximate a function $y = f(\boldsymbol{x})$, where y is some scalar and $\boldsymbol{x}$ is a vector of inputs
- In basic econometrics, this is a regression problem. In macroeconomics, f can be a value function, policy function, pricing kernel etc.
- y can also be a vector (vector of value functions, probability distribution etc.)

Deep learning introduction

- An artifical neural network (ANN) as an approximation to the function $f(\boldsymbol{x})$ takes the form

$$y = f(\boldsymbol{x}) \approx \sigma\left(\sum_{i=1}^{L} w_i x_i\right)$$

- The most fundamental unit of deep neural network is called an artificial neuron

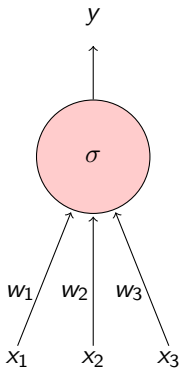
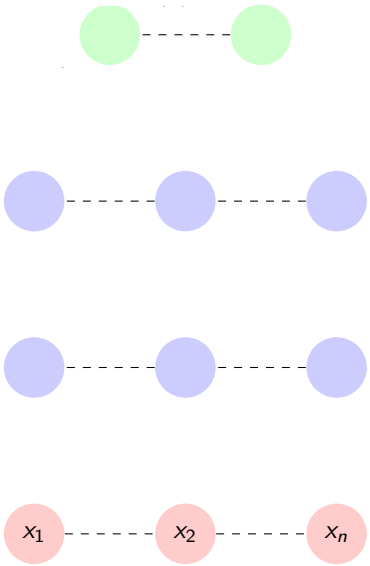


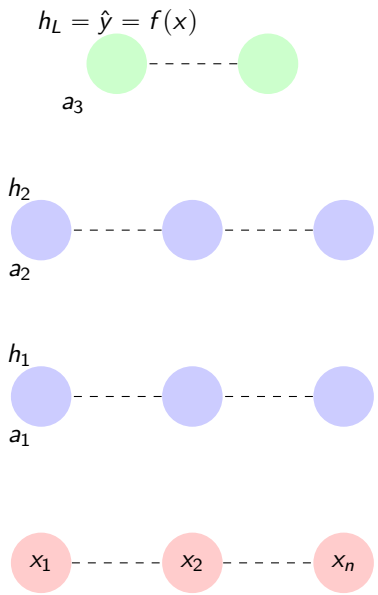
Figure: Artificial Neuron

Feed forward neural network



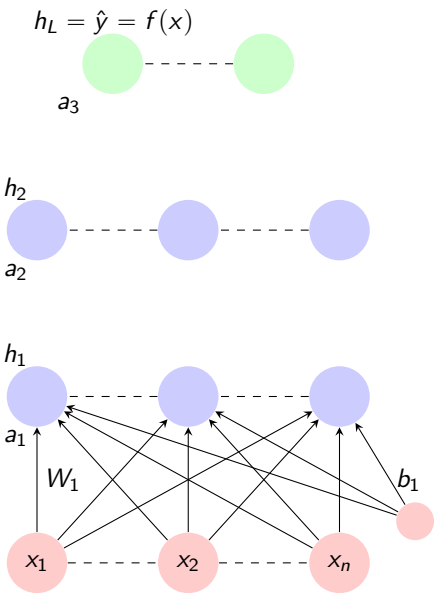
- The input is an n-dimensional vector
- The network contains $L-1$ hidden layers (2, in this case) having $\boldsymbol{n}$ neurons
- The input layer is called 0^{th} layer and the output layer is L^{th} layer
- Finally, there is one output layer containing $\boldsymbol{k}$ neurons
- Each neuron in the hidden layers can be separted into two parts: aggregation (a) and activation (h)
- The parameters for the hidden layers are weights $W_i \in \mathbb{R}^{n \times n}$ and biases $b_i \in \mathbb{R}^n$ for $0 < i < L$
- The parameters for the output layers are weights $W_L \in \mathbb{R}^{n \times k}$ and $b_L \in \mathbb{R}^k$

Feed forward neural network



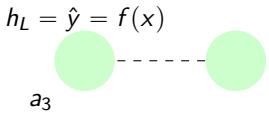
- The input is an n-dimensional vector
- The network contains $L-1$ hidden layers (2, in this case) having $\boldsymbol{n}$ neurons
- The input layer is called 0^{th} layer and the output layer is L^{th} layer
- Finally, there is one output layer containing $\boldsymbol{k}$ neurons
- Each neuron in the hidden layers can be separted into two parts: aggregation (a) and activation (h)
- The parameters for the hidden layers are weights $W_i \in \mathbb{R}^{n \times n}$ and biases $b_i \in \mathbb{R}^n$ for $0 < i < L$
- The parameters for the output layers are weights $W_L \in \mathbb{R}^{n \times k}$ and $b_L \in \mathbb{R}^k$

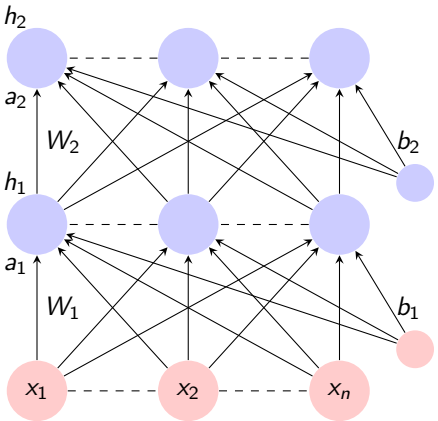
Feed forward neural network



- The input is an n-dimensional vector
- The network contains $L-1$ hidden layers (2, in this case) having $\boldsymbol{n}$ neurons
- The input layer is called 0^{th} layer and the output layer is L^{th} layer
- Finally, there is one output layer containing $\boldsymbol{k}$ neurons
- Each neuron in the hidden layers can be separted into two parts: aggregation (a) and activation (h)
- The parameters for the hidden layers are weights $W_i \in \mathbb{R}^{n \times n}$ and biases $b_i \in \mathbb{R}^n$ for $0 < i < L$
- The parameters for the output layers are weights $W_L \in \mathbb{R}^{n \times k}$ and $b_L \in \mathbb{R}^k$

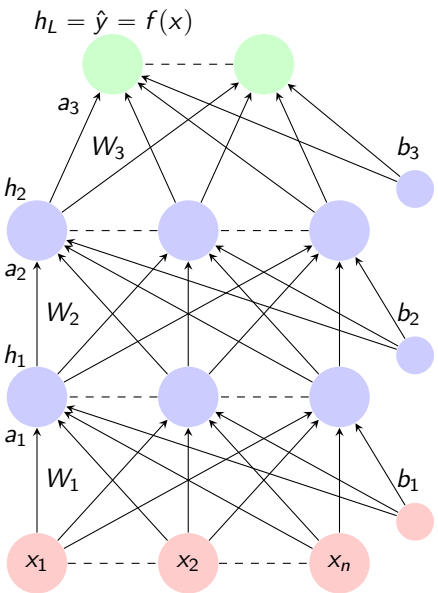
Feed forward neural network





- The input is an n-dimensional vector
- The network contains $L-1$ hidden layers (2, in this case) having $\boldsymbol{n}$ neurons
- The input layer is called 0^{th} layer and the output layer is L^{th} layer
- Finally, there is one output layer containing $\boldsymbol{k}$ neurons
- Each neuron in the hidden layers can be separted into two parts: aggregation (a) and activation (h)
- The parameters for the hidden layers are weights $W_i \in \mathbb{R}^{n \times n}$ and biases $b_i \in \mathbb{R}^n$ for $0 < i < L$
- The parameters for the output layers are weights $W_L \in \mathbb{R}^{n \times k}$ and $b_L \in \mathbb{R}^k$

Feed forward neural network: Mathematical representation



- The aggregation in layer i is given by

$$a_i(x) = b_i + W_i h_{i-1}(x)$$

- The activation in layer i is given by

$$h_i(x) = \sigma(a_i(x))$$

where g is called as the activation function

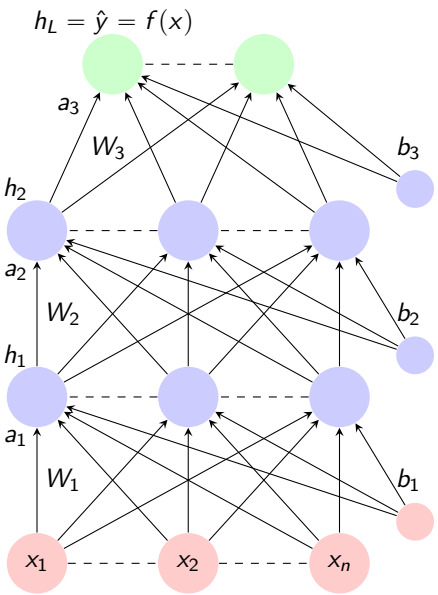
- The activation at the final layer is given by

$$\hat{y}(x) = O(a_L(x))$$

where O is the activation function on the final layer

- For simplicity, we will denote a_i and h_i

Typical problem



- Data: $\{\boldsymbol{x}^j, \boldsymbol{y}^j\}$
- Model:

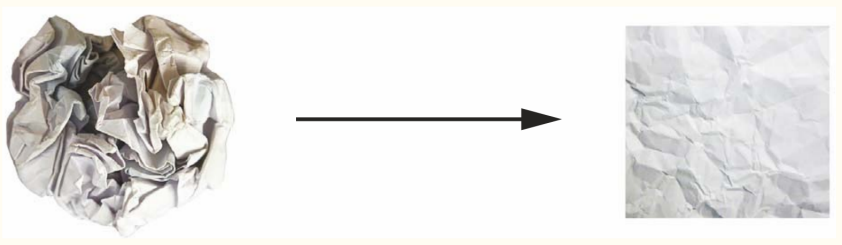
$$\begin{aligned}\hat{\boldsymbol{y}}^j &= f^{DNN}(\boldsymbol{x}^j) \\ &= O\big(W_3\sigma(W2\sigma(W_1\boldsymbol{x}^j + b_1) + b_2) + b_3\big)\end{aligned}$$

- The type of neural network, number of layers, number of neurons in each layer, and activation function constitute architecture of a particular neural network
- Parameters: $\theta = (W_1, ..., W_L; b_1, ..., b_L)$ where $L = 3$
- Goal is to learn the optimal parameters θ using an efficient algorithm

Why deep learning works?

1. Finds representations of data that is informationally efficient
2. Convenient representation of geometry in high-dimensional manifold
   - Deep neural networks are chains of affine transformations- makes affine transformation followed by non-linear transformations sequentially
   - The chains of affine transformations ends up transforming the geometry of the state space
   - Optimizing in transformed geometry is often simpler

Geometric transformation



Source: François Chollet

Why deep learning works?

- Deep neural network is represented mathematically as

$$\hat{y} = f^{DNN}(\boldsymbol{x}) = O\big(W_3\sigma(W2\sigma(W_1\boldsymbol{x} + b_1) + b_2) + b_3\big)$$

  where the parameter vector is $\theta = (W_1, ..., W_L; b_1, .., b_L)$ and O and σ are activation functions
- Comparing this with a standard projection method

$$\hat{y} = f^{Proj}(\boldsymbol{x}) = \sum_{i=1}^{L} b_i\phi_i(\boldsymbol{x})$$

  where the parameter vector is $(b_1., , b_L)$ and ϕ_i is a Chebychev polynomial
- Deep neural networks contain lots of parameters but with simple basis functions. Why is this useful? Because the sequence of affine and non-linear transformations ends up changing the geometry of the state space
- Finding convenient geometric representations of the data is more important than finding the right basis functions for approximation problems. This is where deep learning shines!

Comparison to other methods

Note that other methods can also approximate Borel-measurable functions well but DNNs

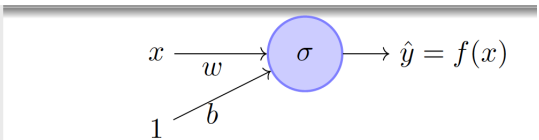
- can also approximate functions with discontinuities. No assumptions about continuity or differentiability required (Universal approximation theorem- Hornik, Stinchcombe, and White (1989))
- can approximate high dimensional functions with better accuracy

	High dimensions	Non-convex state space	Big data	Discontinuous functions	Global dynamics
Projection method	✓	✗	✓	✗	✓
Gaussian processes	✓	✓	✗	✗	✓
Adaptive sparse grid	✓	✗	✓	✓	✓
Deep learning: simulation	✓	✓	✓	✓	✗
Deep learning: active learning	✓	✓	✓	✓	✓

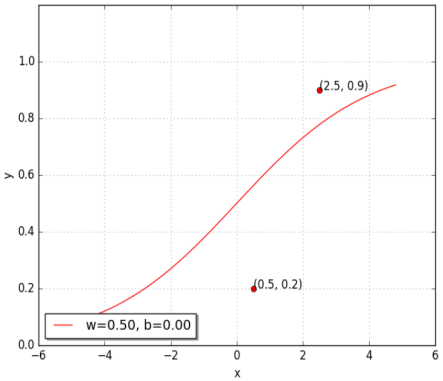
Source: Simon Scheidegger

Typical problem

- The problem at hand is to find the approximation $\hat{y} = f^{ANN}(\boldsymbol{x}; \theta)$
- Assume that f^{ANN} is a simple single layer network with activation $\sigma(\cdot) = \frac{1}{exp(-(wx+b))}$
- Consider a simple one dimensional problem. That is, the goal is to fit $(x, y) = (0.5, 0.2)$ and $(x, y) = (2.5, 0.9)$
- That is, the at the end of training the network, we would like to find θ^* such that $f^{ANN}(0.5) = 0.2$ and $f^{ANN}(2.5) = 0.9$
- The parameter vector $\theta = [w, b]$ contain the weight and bias of the neuron activated σ
- The loss function is given by $\mathcal{L}(w, b) = \sum_{i=1}^{2} \left(y_i - f^{ANN}(x_i)\right)$



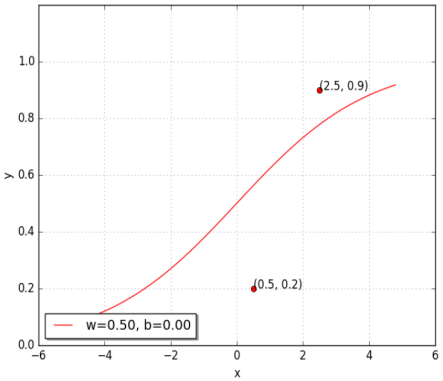
Learning by trial and error



- Can we try to find w^*, b^* manually?

$$\sigma(x) = \frac{1}{1 + e^{-(wx+b)}}$$

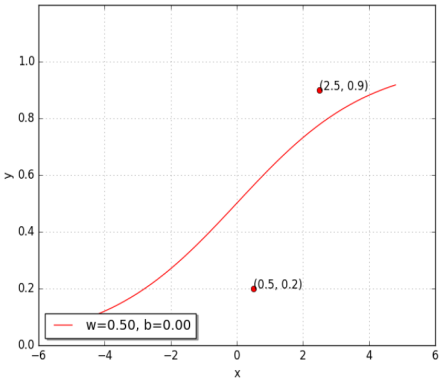
Learning by trial and error



- Can we try to find w^*, b^* manually?
- Let us use a random guess ($w = 0.5, b = 0$)

$$\sigma(x) = \frac{1}{1 + e^{-(wx+b)}}$$

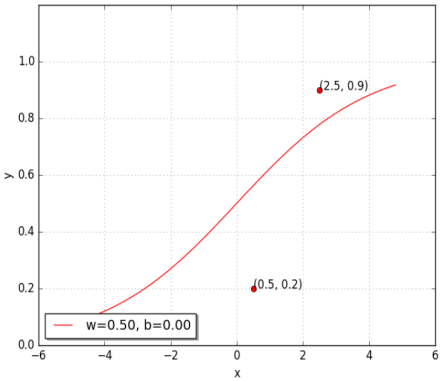
Learning by trial and error



- Can we try to find w^*, b^* manually?
- Let us use a random guess $(w = 0.5, b = 0)$
- Does not seem a great fit. How can we quantify how terrible $(w = 0.5, b = 0)$ is?

$$\sigma(x) = \frac{1}{1 + e^{-(wx+b)}}$$

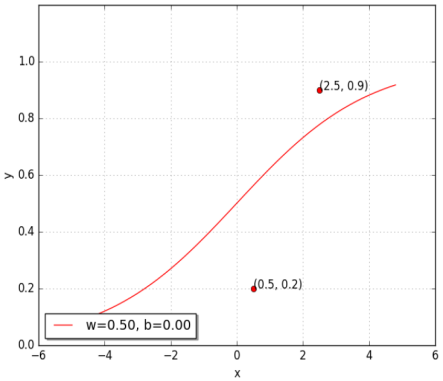
Learning by trial and error



- Can we try to find w^*, b^* manually?
- Let us use a random guess $(w = 0.5, b = 0)$
- Does not seem a great fit. How can we quantify how terrible $(w = 0.5, b = 0)$ is?
- Compute the loss using the loss function $\mathcal{L}(w, b) = \sum_{i=1}^{2} \left(y_i - f^{ANN}(x_i)\right)$

$$\sigma(x) = \frac{1}{1 + e^{-(wx+b)}}$$

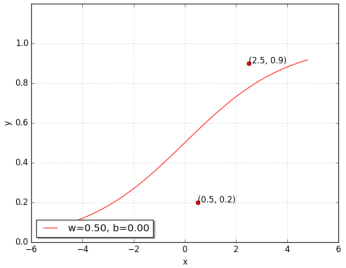
Learning by trial and error



- Can we try to find w^*, b^* manually?
- Let us use a random guess $(w = 0.5, b = 0)$
- Does not seem a great fit. How can we quantify how terrible $(w = 0.5, b = 0)$ is?
- Compute the loss using the loss function $\mathcal{L}(w, b) = \sum_{i=1}^{2} \left(y_i - f^{ANN}(x_i)\right)$
- $\mathcal{L}(0.5, 0) = 0.073$
- The goal is to make $\mathcal{L}(w, b)$ as close to zero as possible

$$\sigma(x) = \frac{1}{1 + e^{-(wx+b)}}$$

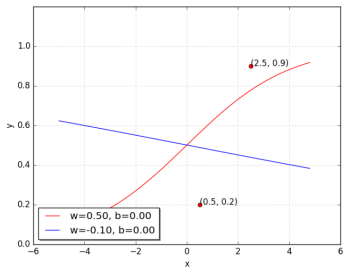
Learning by trial and error



Let us try some other values of w, b

w	b	$\mathscr{L}(w, b)$
0.50	0.00	0.0730

Learning by trial and error

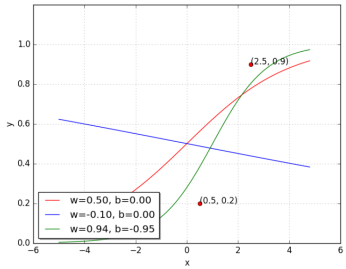


Let us try some other values of w, b

w	b	$\mathscr{L}(w, b)$
0.50	0.00	0.0730
-0.10	0.00	0.1481

It has made things worse. Perhaps it would help to push w and b in the other direction.

Learning by trial and error

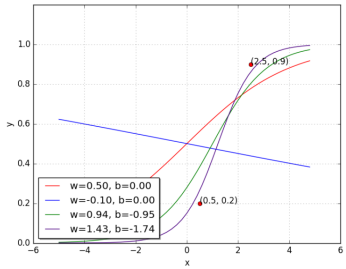


Let us try some other values of w, b

w	b	$\mathscr{L}(w, b)$
0.50	0.00	0.0730
-0.10	0.00	0.1481
0.94	-0.94	0.0214

Much better. Let us keep going in this direction (i.e., increase w and decrease b)

Learning by trial and error

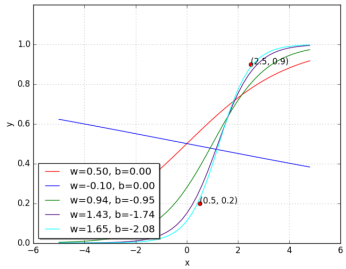


Let us try some other values of w, b

w	b	$\mathscr{L}(w,b)$
0.50	0.00	0.0730
-0.10	0.00	0.1481
0.94	-0.94	0.0214
1.42	-1.73	0.0028

Much better. Let us keep going in this direction (i.e., increase w and decrease b)

Learning by trial and error

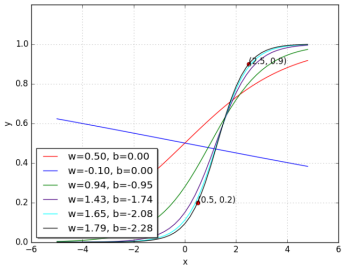


Let us try some other values of w, b

w	b	$\mathscr{L}(w, b)$
0.50	0.00	0.0730
-0.10	0.00	0.1481
0.94	-0.94	0.0214
1.42	-1.73	0.0028
1.65	-2.08	0.0003

Much better. Let us keep going in this direction (i.e., increase w and decrease b)

Learning by trial and error



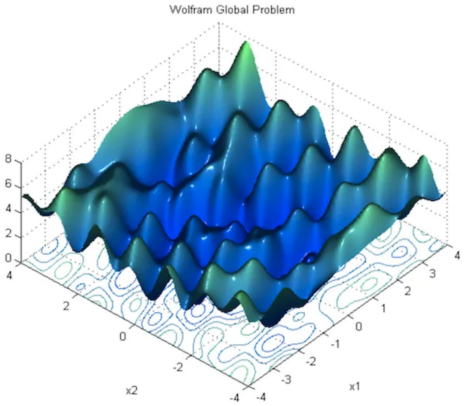
Let us try some other values of w, b

w	b	$\mathscr{L}(w,b)$
0.50	0.00	0.0730
-0.10	0.00	0.1481
0.94	-0.94	0.0214
1.42	-1.73	0.0028
1.65	-2.08	0.0003
1.78	-2.27	0.0000

More principled way of doing this guesswork is what learning is all about!

Why deep neural networks?

- It seems like a single layer is enough to approximate the function well. Why do we need hidden layers?
- Complex problems require deep neural networks



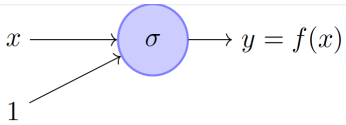
Source: Yoshua Bengio.

Gradient descent algorithm

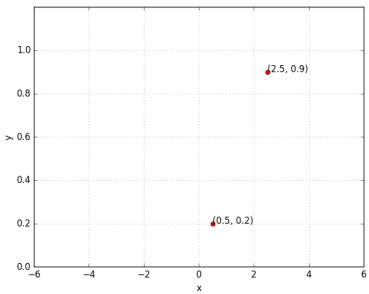
$t \leftarrow 0$
$max\_iter \leftarrow 1000$
while $t < max\_iter$ **do**
  $w_{t+1} \leftarrow w_t - \eta \nabla w_t$
  $b_{t+1} \leftarrow b_t - \eta \nabla b_t$
  $t \leftarrow t + 1$
end

How to obtain ∇w_t and ∇b_t?

Gradient descent



$$f(x) = \frac{1}{1+e^{-(w \cdot x + b)}}$$



- Let's assume that there is only one point to fit (x, y)

$$\mathcal{L}(w, b) = 0.5 * (f^{ANN}(x) - y)^2$$

$$\nabla w = \frac{\partial \mathcal{L}}{\partial w} = \frac{\partial}{\partial w}(0.5 * (f^{ANN}(x) - y)^2)$$

$$\ldots$$

$$\nabla w = (f^{ANN}(x) - y) * f^{ANN}(x) * (1 - f^{ANN}(x)) * x$$

- For two points,

$$\nabla w = \sum_{i=1}^{2} (f^{ANN}(x_i) - y_i) * f^{ANN}(x_i) * (1 - f^{ANN}(x_i)) * x_i$$

$$\nabla b = \sum_{i=1}^{2} (f^{ANN}(x_i) - y_i) * f^{ANN}(x_i) * (1 - f^{ANN}(x_i))$$

Gradient descent

```
X = [0.5, 2.5]
Y = [0.2, 0.9]

def f(w,b,x) : #sigmoid with parameters w,b
    return 1.0 / (1.0 + np.exp(-(w*x + b)))

def error (w, b) :
    err = 0.0
    for x,y in zip(X,Y) :
        fx = f(w,b,x)
        err += 0.5 * (fx - y) ** 2
    return err
```

Gradient descent

```
X = [0.5, 2.5]
Y = [0.2, 0.9]

def f(w,b,x) : #sigmoid with parameters w,b
    return 1.0 / (1.0 + np.exp(-(w*x + b)))

def error (w, b) :
    err = 0.0
    for x,y in zip(X,Y) :
        fx = f(w,b,x)
        err += 0.5 * (fx - y) ** 2
    return err

def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

def grad_w(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx) * x
```

Gradient descent

```
X = [0.5, 2.5]
Y = [0.2, 0.9]

def f(w,b,x) : #sigmoid with parameters w,b
    return 1.0 / (1.0 + np.exp(-(w*x + b)))

def error (w, b) :
    err = 0.0
    for x,y in zip(X,Y) :
        fx = f(w,b,x)
        err += 0.5 * (fx - y) ** 2
    return err

def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

def grad_w(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx) * x

def do_gradient_descent() :
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    for i in range(max_epochs) :
        dw, db = 0, 0
        for x,y in zip(X, Y) :
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)
        w = w - eta * dw
        b = b - eta * db
```

Momemtum gradient descent

- Navigating plateaus take a lot of time since gradients are small
- Momentum based gradient descent fixes the problem
- If you are being repeatedly asked to move in the same direction, then it is a good idea to take bigger steps in that direction

$$u_t = \beta u_{t-1} + \nabla w_t$$
$$w_{t+1} = w_t - \eta u_t$$

After some algebra, we have

$$u_t = \sum_{\tau=0}^{t} \beta^{t-\tau} \nabla w_\tau$$

That is, u_t is the exponentially weighted average of current and all past gradients

Stochastic gradient descent

```
X = [0.5, 2.5]
Y = [0.2, 0.9]

def f(w, b, x): #sigmoid with parameters w,b
    return 1.0 / (1.0 + np.exp(-(w*x +b)))

def error(w, b):
    err = 0.0
    for x,y in zip(X,Y):
        fx = f(w,b,x)
        err += 0.5* (fx - y) ** 2
    return err

def grad_b(w, b, x, y):
    fx = f(w, b, x)
    return (fx - y) * fx * (1 - fx)

def grad_w(w, b, x, y):
    fx = f(w, b, x)
    return (fx - y) * fx * (1 - fx) * x

def do_gradient_descent():
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    for i in range(max_epochs):
        dw, db = 0, 0
        for x, y in zip(X, Y):
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)
        w = w - eta * dw
        b = b - eta * db
```

- In gradient descent, the gradients are computed as the summation of gradients at all points
- Updating the parameters this way is costly especially in large datasets
- An alternative is to update for each data point
  $\Longrightarrow$ Stochastic gradient descent

Stochastic gradient descent

```
def do_stochastic_gradient_descent():
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    for i in range(max_epochs):
        dw, db = 0, 0
        for x, y in zip(X, Y):
            dw = grad_w(w, b, x, y)
            db = grad_b(w, b, x, y)
            w = w - eta * dw
            b = b - eta * db
```

```
def do_gradient_descent() :
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    for i in range(max_epochs) :
        dw, db = 0, 0
        for x,y in zip(X, Y) :
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)
        w = w - eta * dw
        b = b - eta * db
```

- Notice that in the stochastic gradient descent, the parameters are updated for each data point
- The computed gradients are therefore approximations
- This makes the descent stochastic. This is because at each point, the parameters are updated in the direction most favourable to it, without being concerned about other points
- There is no guarantee that at each step the loss is reduced
- Sometimes, the oscillations can be wild. How can we reduce these oscillations? We can use mini-batch gradient descent

Mini-batch gradient descent

```python
def do_mini_batch_gradient_descent() :
    w, b, eta =-2, -2, 1.0
    mini_batch_size, num_points_seen = 2, 0
    for i in range(max_epochs) :
        dw, db, num_points = 0, 0, 0
        for x,y in zip(X, Y) :
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)
            num_points_seen +=1

            if num_points_seen % mini_batch_size == 0 :
                # seen one mini_batch
                w = w - eta * dw
                b = b - eta * db
                dw, db = 0, 0 #reset gradients
```

```python
def do_stochastic_gradient_descent():
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    for i in range(max_epochs):
        dw, db = 0, 0
        for x, y in zip(X, Y):
            dw = grad_w(w, b, x, y)
            db = grad_b(w, b, x, y)
            w = w - eta * dw
            b = b - eta * db
```

- In gradient descent, the the parameters are updated after seeing all data points
- In stochastic gradient descent, the parameters are updated for each data point
- In mini-batch gradient descent, the parameters are updated after seeing mini-batch number of data points

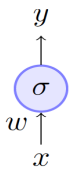
More variants

- Adagrad , RMSProp, Adam: Adjust the learning rate to make sure that parameters pertaining to sparse features get updated properly

Update rule for Adam

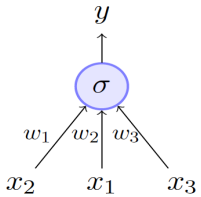
$$
\begin{aligned}
m_t &= \beta_t * m_{t-1} + (1 - \beta_t) * \nabla w_t \\
v_t &= \beta_2 * v_{t-1} + (1 - \beta_2) * (\nabla w_t)^2 \\
\hat{m}_t &= \frac{m_t}{1 - \beta_1^t} \qquad \hat{v}_t = \frac{v_t}{1 - \beta_2^t} \\
w_{t+1} &= w_t - \frac{\eta_t}{\sqrt{\hat{v}_t + \epsilon}} * \hat{m}_t
\end{aligned}
$$

Backpropagation



- We saw how to train a network with no hidden layers and only one neuron

$$
\begin{aligned}
w &= w - \eta \nabla w \\
\nabla w &= \frac{\partial \mathcal{L}(\boldsymbol{w})}{\partial w} \\
&= (f(\boldsymbol{x}) - y) * f(\boldsymbol{x}) * (1 - f(\boldsymbol{x})) * x
\end{aligned}
$$



- Extension to a network with multiple input is straightforward

$$
\begin{aligned}
w_1 &= w_1 - \eta \nabla w_1 \\
w_2 &= w_2 - \eta \nabla w_2 \\
w_3 &= w_3 - \eta \nabla w_3 \\
\nabla w_i &= (f(\boldsymbol{x}) - y) * f(\boldsymbol{x}) * (1 - f(\boldsymbol{x})) * x_i
\end{aligned}
$$

Functional approximation

- **Universal approximation theorem** (Hornik, Stinchcombe, and White (1989)): A neural network with at least one hidden layer can approximate any Borel measureable function to any degree of accuracy
- However, having non-linear activation function in the hidden layers is important
  - ➢ Question: what happens when the activation functions are linear in a deep neural network?
- Once activation function is $\sigma(x) = \frac{1}{1+exp(-(wx+b))}$
- Another popular activation function is the Rectified Linear Unit (ReLU)
  $\sigma(x) = \max\{0, x\}$

Limitations

Obviously, there are some limitations

- Deep neural networks require lots of data to work with
  - ➢ Not a problem for the task at our hand since we will use simulated data
- No theoretical guidance for choosing the right architecture
- Learning can be slow without access to a high performance cluster

Under the hood

- Right choice of architecture and optimizers are important
- Lots of options to choose from
  1. Architectures: Feed-forward, Recurrent, LSTMs, Gated, LLMs etc.
  2. Optimizers: ReLu, SeLu, ELu, Tanh, Sigmoid, Swish, and so on.
- Under the hood details including tensorflow implementation can be found in my mini course available online here.

Part-II: Application

ALIENs: What is it about?

- **ENs**: Use neural network to solve general equilibrium continuous time finance models to capture global dynamics (portfolio choice, macro-finance, monetary policy)

1. Portfolio Choice: Merton (1971), Cochrane et al (2008), Martin (2013)
2. Macro-Finance: He and Krishnamurthy (2013), Brunnermeier and Sannikov (2014), Di Tella (2017), Li (2019), Krishnamurthy and Li (2022)
3. Monetary Theory: Silva (2020), Brunnermeier and Sannikov (2016), Drechsler, Savov, and Schnabl (2018)

ALIENs: What is it about?

- **I**: Encode economic information as regularizer
- **ENs**: Use neural network to solve general equilibrium continuous time finance models to capture global dynamics (portfolio choice, macro-finance, monetary policy)

1. Portfolio Choice: Merton (1971), Cochrane et al (2008), Martin (2013)
2. Macro-Finance: He and Krishnamurthy (2013), Brunnermeier and Sannikov (2014), Di Tella (2017), Li (2019), Krishnamurthy and Li (2022)
3. Monetary Theory: Silva (2020), Brunnermeier and Sannikov (2016), Drechsler, Savov, and Schnabl (2018)

ALIENs: What is it about?

- **AL**: Actively learn about state space with stark non-linearity/large prediction error
- **I**: Encode economic information as regularizer
- **ENs**: Use neural network to solve general equilibrium continuous time finance models to capture global dynamics (portfolio choice, macro-finance, monetary policy)

1. Portfolio Choice: Merton (1971), Cochrane et al (2008), Martin (2013)
2. Macro-Finance: He and Krishnamurthy (2013), Brunnermeier and Sannikov (2014), Di Tella (2017), Li (2019), Krishnamurthy and Li (2022)
3. Monetary Theory: Silva (2020), Brunnermeier and Sannikov (2016), Drechsler, Savov, and Schnabl (2018)

General setup

- $$U_t = E_t\Big[\int_t^{\infty} f(c_s, U_s)ds\Big] \tag{1}$$

General setup

- $$U_t = E_t\Big[\int_t^{\infty} f(c_s, U_s)ds\Big] \tag{1}$$

- Exogenous dividend process of risky asset

$$\frac{dy_t}{y_t} = gdt + \sigma \underbrace{dZ_t}_{\text{Brownian shock}} \tag{2}$$

General setup

$$U_t = E_t\Big[\int_t^{\infty} f(c_s, U_s)ds\Big] \tag{1}$$

- Exogenous dividend process of risky asset

$$\frac{dy_t}{y_t} = gdt + \sigma \underbrace{dZ_t}_{\text{Brownian shock}} \tag{2}$$

- There is also a risk free debt market (pays return r). Risky asset has price of risk ζ_t, and volatility σ_t^R
- Problem of the agent is

$$\sup_{\hat{c},\theta} \quad U_t \tag{3}$$

$$\text{s.t} \quad \frac{dw_t}{w_t} = (r + \underbrace{\theta_t}_{\text{port. choice}} \underbrace{\zeta_t}_{\text{price of risk}} - \hat{c}_t)dt + \theta_t \underbrace{\sigma_t^R}_{\text{ret. volatility}} dZ_t \tag{4}$$

- If g, σ, r are time varying, then we have a multi-dimensional problem

HJB

- HJB is

$$\sup_{\hat{c}_t, \theta_t} f(c_t, U_t) + E_t(dU_t) = 0$$

- Conjecturing $U = \frac{Jw^{1-\gamma}}{1-\gamma}$, where J is the stochastic opportunity process and γ is the risk aversion, the HJB equation reduces to

$$\mu^J(\boldsymbol{x}, J)J = \sum_{i=1}^{d} \mu^{x_i}(\boldsymbol{x}, J)\frac{\partial J}{\partial x_i} + \sum_{i,j=1}^{d} b^{i,j}(\boldsymbol{x}, J)\frac{\partial^2 J}{\partial x_i \partial x_j} \tag{5}$$

  1. State variables are x. Could be high-dimensional (large d)
  2. μ^J, μ^x, and $b^{i,j}$ are linear, advection, and diffusion coefficients

- PDE (5) can be highly non-linear elliptical PDE depending on the problem
- Past literature: Convert it into **quasi-linear parabolic PDE** and use finite difference $\rightarrow$ slowly introduce non-linearity through

$$\mu^J(\boldsymbol{x}, J^{old})J = \frac{\partial J}{\partial t} + \sum_{i=1}^{d} \mu^{x_i}(\boldsymbol{x}, J^{old})\frac{\partial J}{\partial x_i} + \sum_{i,j=1}^{d} b^{i,j}(\boldsymbol{x}, J^{old})\frac{\partial^2 J}{\partial x_i \partial x_j} \tag{6}$$

- Works well in low dimensions, but breaks down in high dimensions

Methodology overview

- Focus of this part is to introduce a technique to solve macro models involving PDEs of type (5) in high dimensions
  1. Benchmark model (BS2016 with recursive preference)
  2. Capital misallocation model with productivity shock (Gopalakrishna 2021)

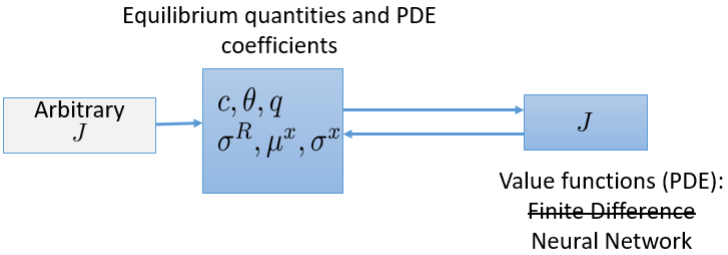


Figure: Overview of methodology.

Neural network solution method

$$f := \frac{\partial \hat{J}}{\partial t} + \sum_{i}^{d} \mu^{i}(\boldsymbol{x}) \frac{\partial \hat{J}}{\partial x_i} + \sum_{i,j=1}^{d} b^{i,j}(\boldsymbol{x}) \frac{\partial^2 \hat{J}}{\partial x_i \partial x_j} - \mu^{J} \hat{J} = 0;$$

$$\forall (t, \boldsymbol{x}) \in [T - k\Delta t, T - (k-1)\Delta t] \times \Omega$$

$$\hat{J} = \tilde{J}_0 \quad \forall (t, \boldsymbol{x}) \in (T - (k-1)\Delta t) \times \Omega;$$

where $\hat{J}$ is a neural network object with parameters Θ, and f is the PDE residual.

Neural network solution method

$$f := \frac{\partial \hat{J}}{\partial t} + \sum_{i}^{d} \mu^{i}(\boldsymbol{x}) \frac{\partial \hat{J}}{\partial x_i} + \sum_{i,j=1}^{d} b^{i,j}(\boldsymbol{x}) \frac{\partial^2 \hat{J}}{\partial x_i \partial x_j} - \mu^{J} \hat{J} = 0;$$

$$\forall (t, \boldsymbol{x}) \in [T - k\Delta t, T - (k-1)\Delta t] \times \Omega$$

$$\hat{J} = \tilde{J}_0 \quad \forall (t, \boldsymbol{x}) \in (T - (k-1)\Delta t) \times \Omega;$$

where $\hat{J}$ is a neural network object with parameters Θ, and f is the PDE residual. Can be seen as a classical constrained optimization problem

Optimization

$$\Theta^* = \underset{\Theta}{argmin} \quad \hat{J} - \tilde{J}_0$$

$$\text{s.t.} \quad f = 0$$

Neural network solution method

$$f := \frac{\partial \hat{J}}{\partial t} + \sum_{i}^{d} \mu^{i}(\boldsymbol{x}) \frac{\partial \hat{J}}{\partial x_i} + \sum_{i,j=1}^{d} b^{i,j}(\boldsymbol{x}) \frac{\partial^2 \hat{J}}{\partial x_i \partial x_j} - \mu^{J} \hat{J} = 0;$$

$$\forall (t, \boldsymbol{x}) \in [T - k\Delta t, T - (k-1)\Delta t] \times \Omega$$

$$\hat{J} = \tilde{J}_0 \quad \forall (t, \boldsymbol{x}) \in (T - (k-1)\Delta t) \times \Omega;$$

Can be seen as an classical constrained optimization problem

Optimization

$$\Theta^* = \underset{\Theta}{argmin} \quad \hat{J} - \tilde{J}_0$$

$$\text{s.t.} \quad \int_t \int_{\boldsymbol{x}} |f|^2 dt d\boldsymbol{x} = 0$$

Neural network solution method

$$f := \frac{\partial \hat{J}}{\partial t} + \sum_{i}^{d} \mu^{i}(\boldsymbol{x}) \frac{\partial \hat{J}}{\partial x_i} + \sum_{i,j=1}^{d} b^{i,j}(\boldsymbol{x}) \frac{\partial^2 \hat{J}}{\partial x_i \partial x_j} - \mu^{J} \hat{J} = 0;$$

$$\forall (t, \boldsymbol{x}) \in [T - k\Delta t, T - (k-1)\Delta t] \times \Omega$$

$$\hat{J} = \tilde{J}_0 \quad \forall (t, \boldsymbol{x}) \in (T - (k-1)\Delta t) \times \Omega$$

- Mesh free since we can randomly sample from the state space $(t, \boldsymbol{x})$ to train the neural network

Neural network solution method

$$f := \frac{\partial \hat{J}}{\partial t} + \sum_{i}^{d} \mu^{i}(\boldsymbol{x}) \frac{\partial \hat{J}}{\partial x_i} + \sum_{i,j=1}^{d} b^{i,j}(\boldsymbol{x}) \frac{\partial^2 \hat{J}}{\partial x_i \partial x_j} - \mu^{J} \hat{J} = 0;$$

$$\forall (t, \boldsymbol{x}) \in [T - k\Delta t, T - (k-1)\Delta t] \times \Omega$$

$$\hat{J} = \tilde{J}_0 \quad \forall (t, \boldsymbol{x}) \in (T - (k-1)\Delta t) \times \Omega$$

- Mesh free since we can randomly sample from the state space $(t, \boldsymbol{x})$ to train the neural network
- Sparse training points in region of importance leads to instability in future iterations. Solution: Track subdomain Ω_c and sample more points from there

$$f = 0 \quad \forall (t, \boldsymbol{x}) \in [T - k\Delta t, T - (k-1)\Delta t] \times \Omega_c;$$

$$\hat{J} = \tilde{J}_0 \quad \forall (\boldsymbol{x}, t) \in (T - (k-1)\Delta t) \times \Omega_c;$$

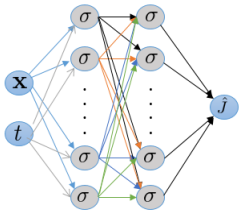
- The subdomain Ω_c is found by inspecting the PDE coefficients which are determined using previous value $\tilde{J}$

Neural network solution method

$$f := \frac{\partial \hat{J}(\boldsymbol{x}|\Theta)}{\partial t} + \sum_{i}^{d} \mu^{i}(\mathbf{x}) \frac{\partial \hat{J}(\boldsymbol{x}|\Theta)}{\partial x_i} + \sum_{i,j=1}^{d} b^{i,j}(\mathbf{x}) \frac{\partial^2 \hat{J}(\boldsymbol{x}|\Theta)}{\partial x_i \partial x_j} - \mu^{J} \hat{J}(\boldsymbol{x}|\Theta) = 0;$$

$$\forall (t, \mathbf{x}) \in [T - k\Delta t, T - (k-1)\Delta t] \times \Omega$$

$$\hat{J}(\boldsymbol{x}|\Theta) = \tilde{J}_0; \quad \forall (t, \boldsymbol{x}) \in (T - (k-1)\Delta t) \times \Omega;$$



$\mathbf{x} \in \mathbb{R}^d$ Space dimension

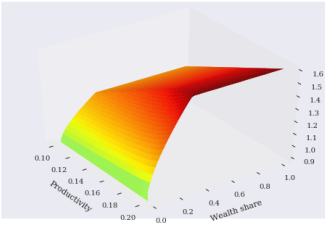
$t \in [0, T]$ Time dimension

σ Tanh activation function. $\sigma(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

$\hat{J}(x \mid \Theta)$ Output from neural network

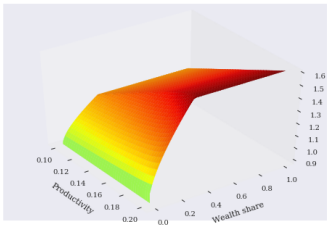
Active learning

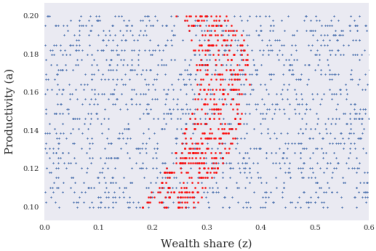
Example from Gopalakrishna (2021): Macro-finance model with 2 state variables (productivity, wealth share)



Active learning

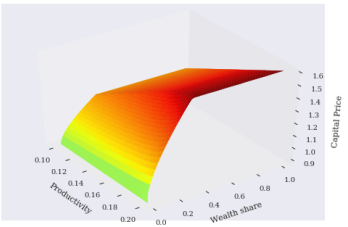
Example from Gopalakrishna (2021): Macro-finance model with 2 state variables (productivity, wealth share)

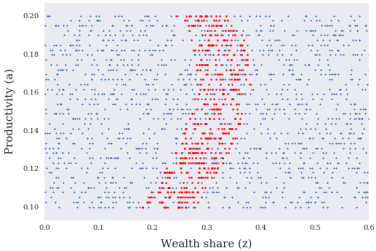


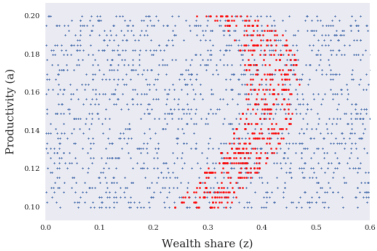


Active learning

Example from Gopalakrishna (2021): Macro-finance model with 2 state variables (productivity, wealth share)







Solution technique: ALIENs

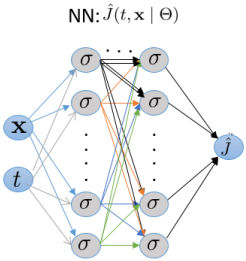


Figure: Methodology.

Solution technique: ALIENs

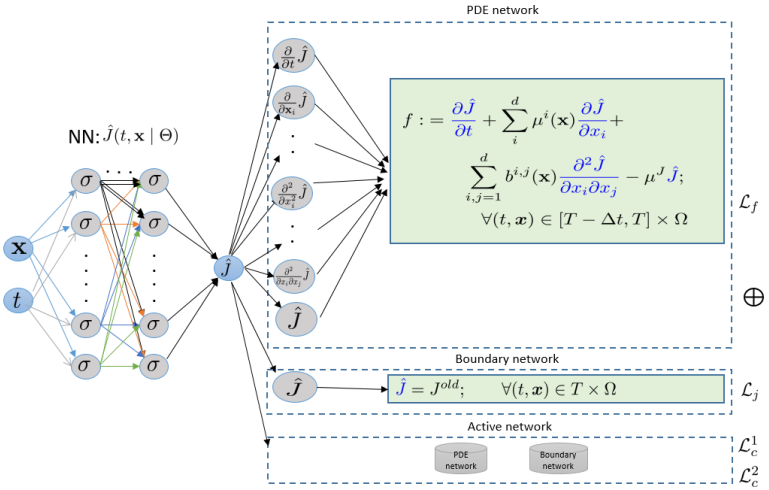


Figure: Methodology.

Solution technique: ALIENs

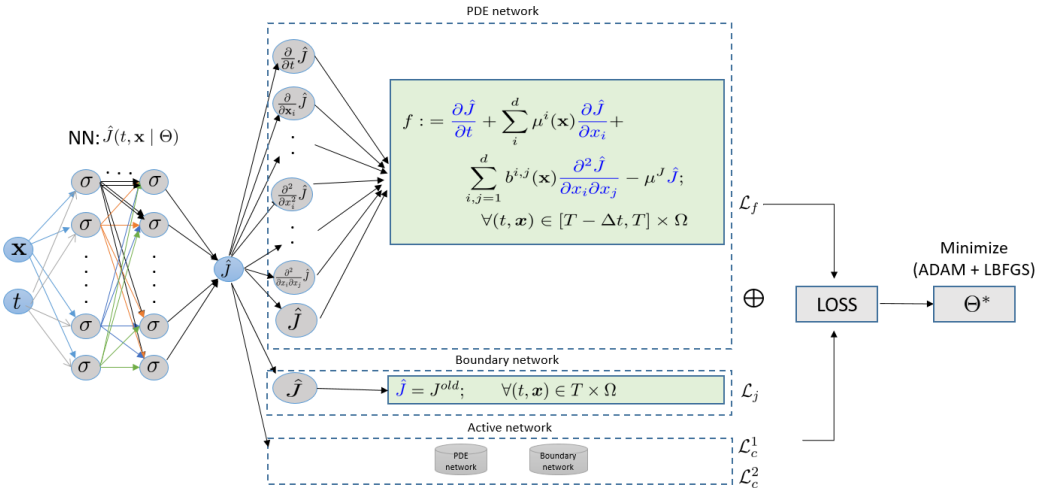


Figure: Methodology.

ALIENs

$$\mathcal{L} = \lambda_f \mathcal{L}_f + \lambda_j \mathcal{L}_j + \lambda_b \mathcal{L}_b + \lambda_c^1 \mathcal{L}_c^1 + \lambda_c^2 \mathcal{L}_c^2 \tag{7}$$

where

$$\text{PDE loss} \qquad \mathcal{L}_f = \frac{1}{N_f} \sum_{i=1}^{N_f} |f(\boldsymbol{x}_f^i, t_f^i)|^2$$

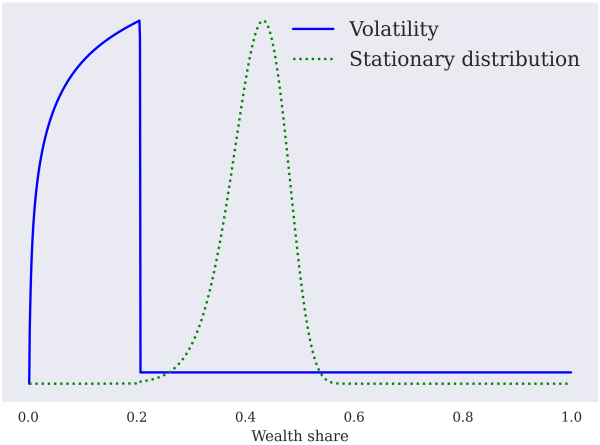
$$\text{Bounding loss-1} \qquad \mathcal{L}^j = \frac{1}{N_j} \sum_{i=1}^{N_j} |\hat{J}(\boldsymbol{x}_j^i, t_j^i) - \tilde{J}_0^i|^2$$

$$\text{Active loss-1} \qquad \mathcal{L}_c^2 = \frac{1}{N_c} \sum_{i=1}^{N_c} |f(\boldsymbol{x}_c^i, t_c^i)|^2$$

$$\text{Active loss-2} \qquad \mathcal{L}_c^1 = \frac{1}{N_c} \sum_{i=1}^{N_c} |\hat{J}(\boldsymbol{x}_c^i, t_c^i) - \tilde{J}_0^i|^2$$

Active Learning vs Simulation method

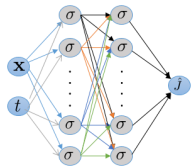
- ALIENs actively learn the region of sharp transition and samples more points → faster convergence
- Sampling procedure is complementary to simulation based methods (Azinovic et al (2018), Villaverde et al (2020)), but also works for models with rare events and financial constraints that bind far away from the steady state



Automatic differentiation in practice

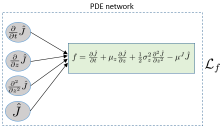
Approximating J using a neural network

```
def J(z,t):
        J = neural_net(tf.concat([z,t],1),weights,biases)
        return J
```



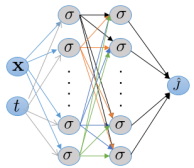
Constructing regularizer: 1D model

```
def f(z,t):
        J = J(z,t)
        J_t = tf.gradients(J,t)[0]
        J_z = tf.gradients(J,z)[0]
        J_zz = tf.gradients(J_z,z)[0]
        f = J_t + advection * J_z + diffusion * J_zz - linearTerm * J
        return f
```



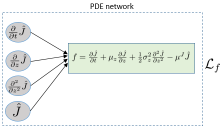
Automatic differentiation in practice

Approximating J using a neural network

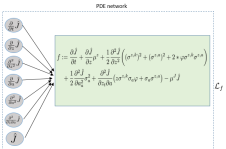



```
def J(z,t):
        J = neural_net(tf.concat([z,t],1),weights,biases)
        return J
```

Constructing regularizer: 1D model




```
def f(z,t):
        J = J(z,t)
        J_t = tf.gradients(J,t)[0]
        J_z = tf.gradients(J,z)[0]
        J_zz = tf.gradients(J_z,z)[0]
        f = J_t + advection * J_z + diffusion * J_zz - linearTerm * J
        return f
```




```
def f(z,a,t):
        J = J(z,a,t)
        J_t = tf.gradients(J,t)[0]
        J_z = tf.gradients(J,z)[0]
        J_a = tf.gradients(J,a)[0]
        J_zz = tf.gradients(J_z,z)[0]
        J_aa = tf.gradients(J_a,a)[0]
        J_az = tf.gradients(J_a,z)[0]
        f = J_t + advection_z * J_z + advection_a * J_a + diffusion_z * J_zz +
                                  diffusion_a * J_aa  + crossTerm * J_az - linearTerm * J
        return f
```

Horovod

- Data parallelism as opposed to Model parallelism
- Horovod uses ringAllReduce operation to average gradients (improves efficiency)

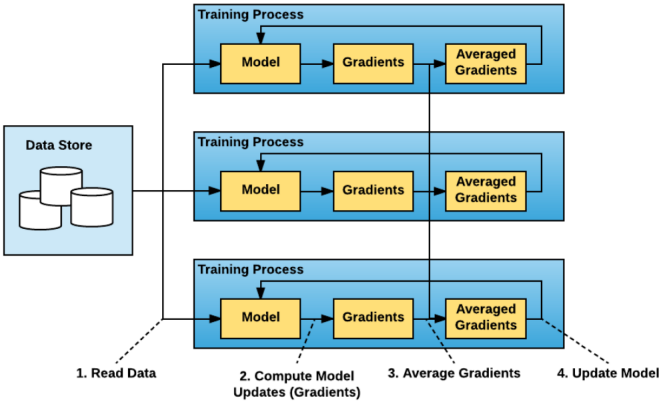


Figure: Source: https://eng.uber.com/horovod/

Horovod

```
def J():
        ...
def f():
        ...

hvd.init() #initialize Horovod
config = tf.ConfigProto() #pin GPUs to processes
config.gpu_options.visible_device_list = str(hvd.local_rank()) #assign chief worker
config.gpu_options.allow_growth = True #enable GPU
sess= tf.Session(config=config) #Configure tensorflow
if hvd.rank()==0:
        ... #assign a piece of data to chief worker
else:
        while hvd.rank() < hvd.size():
                ... #assign a piece of data to each worker

def build_model():
        #initialize parameters using Xavier initialization
        #parametrize the function J using J()
        #buld loss function using net_f()
        #set up tensorflow optimizer in the variable name opt
        optimizer = hvd.DistributedOptimizer(opt)
        #minimize loss
        #initialize Tensorflow session
        bcast = hvd.broadcast_global_variables(0) #Broadcast parameters to all workers
        sess.run(bcast)
        #train the deep learning model
```

HPC

Interactive mode

Sinteract -q gpu -p gpu -g gpu -m 12G -t 10:00:00
virtualenv –system-site-packages venv-for-tf
source ./venv-for-tf/bin/activate
pip install –user –no-cache-dir tensorflow-gpu==2.7.0

ipythonCores: 1
Tasks: 1
Time: 10:00:00
Memory: 128G
Partition: gpu
Account: sfi-pcd
Jobname: interact
Resource: gpu
QOS: gpu
salloc: job 124415 allocated

References: Part-I

- Textbooks:
  1. Raul Rojas. Neural Networks: A Systematic Introduction. 1996
  2. Ian Goodfellow, Yoshua Bengio and Aaron Courville. Deep Learning. An MIT Press book. 2016
- Other sources
  1. Dive into deep learning (interactive learning material)
  2. CSCS - USI Summer school 2020 by Simon Scheidegger
  3. Machine learning for macroeconomics (teaching slides) by Jesús Fernández-Villaverde
  4. Neural networks (teaching slides) by Hugo Larochelle
  5. Deep learning CS6910 (teaching slides) by Mitesh Khapra

References: Part-II

- Goutham Gopalakrishna. ALIENs and Continuous Time Economies. 2021. SSRN Working paper.
- Justin Sirignano and Konstantinos Spiliopoulos. DGM: A deep learning algorithm for solving partial differential equations. 2018a. Journal of Computational Physics.
- Victor Duarte. Machine Learning for Continuous-Time Economics. 2017. SSRN Working paper.
- Jesús Fernández-Villaverde, Samuel Hurtado, and Galo Nuno. Financial Frictions and the Wealth Distribution. 2022. Econometrica (forthcoming).
- Princeton Mini course materials (slides and code) by Goutham Gopalakrishna: Github page.