EP24040 - Computational Physics

Kirit Makwana
Department of Physics, IIT Hyderabad

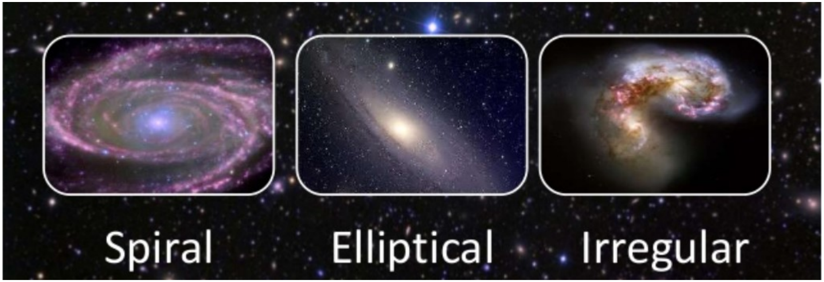
Neural networks

Motivation

- There are a whole slew of numerical methods that aim to approximate the exact solution of equations with some discretization procedure
- In the case of Monte-Carlo techniques, there is a randomization procedure
- The approximation can also be done using spectral methods - methods in which the problem is transformed from real space to another more suitable space
- Fourier transforms, Hermite transform, Chebyshev transforms, etc
- Further still, Monte Carlo methods have also been modified to make computations faster using greedy algorithms
- All these techniques are extensively used in Physics, as well as in other fields

Classification

- A new class of problem has arisen due to large availability of data
- The problem is classification of data
- Often times we have lots of data and need to organize it into human relevant classes
- For example, identifying different animals from image data, or converting a signal into an alphabet (speech recognition)
- Classifying the types of galaxies (spiral, elliptical)



Classification

- Analyzing noisy data, removing or reducing noise, predicting disruptive events
- In these problems we have a lot of input data, and from past experience we know what output we get in the case of input data
- We need to make a model such that given some input data, we can predict the outcome
- For example, sample image data and identify the animal in that

Input output vectors

- Any form of input can be converted into a vector
- Pixels from a photo can be arranged in an input vector $\boldsymbol{x} \in \mathbb{R}^n$
- The output can be another vector $\boldsymbol{y} \in \mathbb{R}^m$
- The output can be very simple such as a two element vector, signifying whether the animal is cat or dog

$$\begin{bmatrix}1\\0\end{bmatrix} \implies \text{Cat or } \begin{bmatrix}0\\1\end{bmatrix} \implies \text{Dog} \tag{1}$$

Linear transformation

- A linear model takes a linear transformation from $\boldsymbol{x} \in \mathbb{R}^n$ to $\boldsymbol{y} \in \mathbb{R}^m$
- This can be done by simply matrix multiplication

$$\boldsymbol{y} = A\boldsymbol{x} \tag{2}$$

- where A is an $m \times n$ matrix which we are interested in finding
- Lets say we have 3 input vector $\boldsymbol{x}_1 = [1, -1]$, $\boldsymbol{x}_2 = [0, 1]$, and $\boldsymbol{x}_3 = [1, 1]$
- Lets say for these cases, we have the output $\boldsymbol{y}_1 = [0, 1]$, $\boldsymbol{y}_2 = [0, 1]$, $\boldsymbol{y}_3 = [1, 0]$ respectively
- We can try to find an A which will be a 2×2 matrix such that

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} 1 \\ -1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \text{ and } \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{3}$$

Linear model

- We can see that A is

$$A = \begin{bmatrix} 0 & 0 \\ 2 & 1 \end{bmatrix} \tag{4}$$

- This matrix is uniquely determined by the first two conditions
- But then it does not satisfy the third condition
- This means a linear model will not work in this case
- In general, as the number of cases increases, you will get more conditions, while the matrix will have maximum rank of n
- So the system will be overdetermined and we cannot fit the linear model exactly
- One way to overcome this drawback is to go to a nonlinear model

Nonlinear model

- Take the output of a linear model and apply a nonlinear function to it

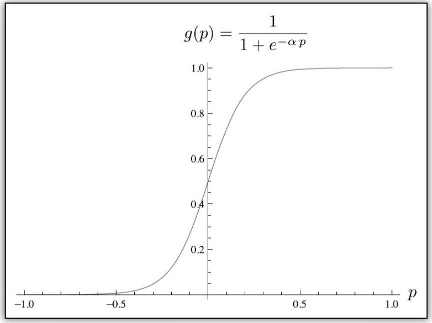
$$\boldsymbol{z} = g(\boldsymbol{y}) = \begin{bmatrix} g(y_1) \\ g(y_2) \\ . \\ . \\ g(y_m) \end{bmatrix} \tag{5}$$

- Defining $g(p)$ gives extra flexibility to fit all the required conditions, ex. the third condition above
- The input and outputs can be standardized for ease of operation
- Set $g(p)$ to lie between 0 and 1

Sigmoid function

- The sigmoid function is a continuous version of the step function

$$g(p) = \frac{1}{1+e^{-\alpha p}} \tag{6}$$



- α sets the width of the sigmoid function, higher α makes the function sharper

Setting α

- The elements of the nonlinear output $\boldsymbol{z}$ are

$$z_i = g((A\boldsymbol{x})_i) = g[\sum_{j=1}^{n} A_{ij}x_j] \tag{7}$$

- Now the argument of function g should span the range of the width of the sigmoid function, so that the output g takes the full range of 0 to 1
- Assume that the elements of matrix A will be of the order unity
- Then the maximum element of $A\boldsymbol{x}$ will be approximately $n\max(x_i)$
- Then take α as

$$\alpha = \frac{10}{n\max(x_i)} \tag{8}$$

- Also, we can shift the input numbers $x_i \rightarrow x_i - med(x_i)$ such that the maximum and minimum are equal and opposite in signs
- This way we can handle input with any magnitude $\boldsymbol{x}$

Training

- Now the next step is to find the matrix A
- This process is called as training. In this we provide training examples to the model
- These examples are basically pairs of $[\boldsymbol{x}^k, \boldsymbol{y}^k]_{k=1}^T$ where $\boldsymbol{x}^k$ is the k-th input vector and $\boldsymbol{y}^k$ is its desired output vector
- There are T training examples, labelled by k, and our goal is to set A such that we get $\boldsymbol{y}^k$ for given $\boldsymbol{x}^k$
- For this we define an error function f

$$f^k = ||g(A\boldsymbol{x}^k) - \boldsymbol{y}^k||^2 \text{ for each case} j = 1, 2, ..., T \tag{9}$$

- The norm of a vector can be defined as the square root of the sum of the squares of its elements, i.e.,

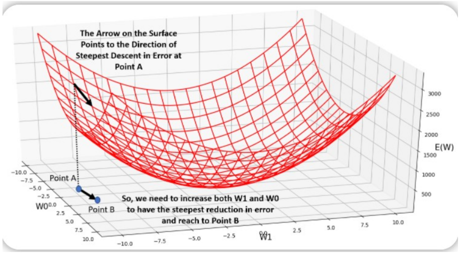
$$f^k = \sum_{i=1}^{m} [g(\sum_{s=1}^{n} A_{is} x_s^k) - y_i^k]^2 \tag{10}$$

Training

- This function f can be thought of as a function of the matrix A_{ij} because that is the only variable, rest all quantities come from the training set
- Hence, the objective then is the adjust A_{ij} such that we can minimize f for each training example
- f is a positive definite quantity, and this problem is of minimization of f
- A well-known method is known as the steepest descent (gradient descent) method

Gradient descent

- Consider a function of 2 variables, this can be represented as a surface
- This function contains has a minima



- If we have a function $f(\boldsymbol{x})$ of which we want to find minimum, then starting from some position $\boldsymbol{x}_0 = (x_0, y_0)$

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n - \eta \nabla f \tag{11}$$

- η is the step size, also called as "rate of learning"

Steepest descent

- Ex. find the minimum of $f(x, y) = x^2 + 4y^2 - 4x - 24y + 40$
- This actually comes from $(x-2)^2 + 4(y-3)^2$, so the minimum is at $(x, y) = (2, 3)$
- Learning rate can be faster with higher η
- However, too high η might lead to method not converging or instability
- Also called steepest descent because change is maximum in direction of gradient
- Convergence is determined when the steps become smaller than some threshold

Training model

- Going back to the case of finding the matrix A in our model
- The function to be minimized is

$$f(A_{ij}) = \sum_{i=1}^{m} \left[g(\sum_{j=1}^{n} A_{ij} x_j) - y_i \right]^2 \tag{12}$$

- given any training pair $(\boldsymbol{x}^k, \boldsymbol{y}^k)$ - suppressing the superscript k for simplicity
- Define $b_i = \sum_{j=1}^{n} A_{ij} x_j$ and $z_i = g(b_i)$, i goes from 1 to m
- Now we want to find the values of matrix elements A_{pq} which minimizes the error f
- For using gradient descent we need

$$\frac{\partial f}{\partial A_{pq}} = \sum_{i=1}^{m} 2(z_i - y_i) \frac{\partial z_i}{\partial A_{pq}} \tag{13}$$

Training model

- We have

$$\frac{\partial z_i}{\partial A_{pq}} = g'(b_i)\frac{\partial b_i}{\partial A_{pq}} \tag{14}$$

$$\frac{\partial b_i}{\partial A_{pq}} = \frac{\partial}{\partial A_{pq}}\sum_{j=1}^{m} A_{ij}x_i \tag{15}$$

$$= \sum_{j=1}^{m} \delta_{ip}\delta_{jq}x_i \tag{16}$$

$$= \delta_{ip}x_q \tag{17}$$

- Also can be shown that

$$g'(p) = \alpha g(p)(1 - g(p)) \tag{18}$$

Training model

- Putting this together we get

$$\frac{\partial f}{\partial A_{pq}} = \sum_{i=1}^{m} 2(z_i - y_i)\alpha g(b_i)(1 - g(b_i))\delta_{ip} x_q \tag{19}$$

- which gives

$$\frac{\partial f}{\partial A_{pq}} = 2(z_p - y_p)\alpha z_p(1 - z_p)x_q \tag{20}$$

- This makes the calculation of the matrix element easy

Algorithm

- Initialize random A of order unity
- Loop over each training pair
- Calculate z using the existing A matrix
- Loop over all elements of A_{pq} and update with

$$A_{pq} \to A_{pq} - \eta\left[2(z_p - y_p)\alpha z_p(1 - z_p)x_q\right] \tag{21}$$

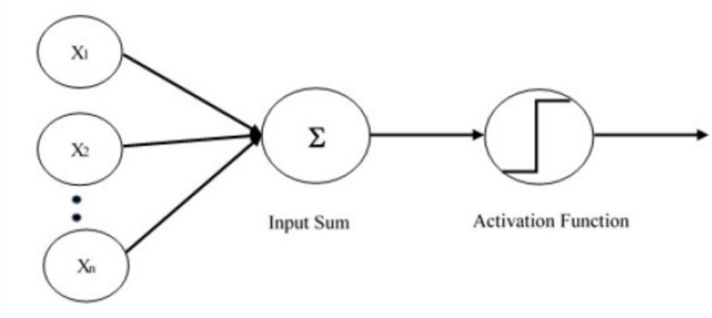
- Taking one gradient descent step over all the training examples is called an "epoch"
- Run the training for several epochs till the error is reduced below some threshold or the accuracy of the model reaches desired level
- The accuracy can be defined as the percentage of predictions it gets right
- After going through all training pairs and epochs, the model is trained and can be used for new data

Training model

- This training is not expected to produce exact results
- The outcome of the training will depend on the order in which the test cases are given
- Ideally after giving the training examples multiple times and in random order, the neural network A should converge
- However, this may not be the case always and it may not give good agreement for all the training examples
- After the model is trained on the training set, it can be tested on a testing set to check how well it performs
- If the performance if good, it can be used on new and untested data

Hidden layers

- If we are given an input vector with 10 elements, i.e., $n = 10$ and the output if just a single element, i.e., $m = 1$ then
- The matrix A will be 10×1 matrix with 10 parameters to adjust
- This can be represented diagramatically as



Hidden layers

- If there is a lot of data which is making the problem overconstrained, then we require more freedom in the neural network model
- We can add an intermediate layer of size k
- The input $\boldsymbol{x} \in R^n$ gets multiplied by matrix B which is $k \times n$ giving a $\tilde{\boldsymbol{z}} \in R^k$

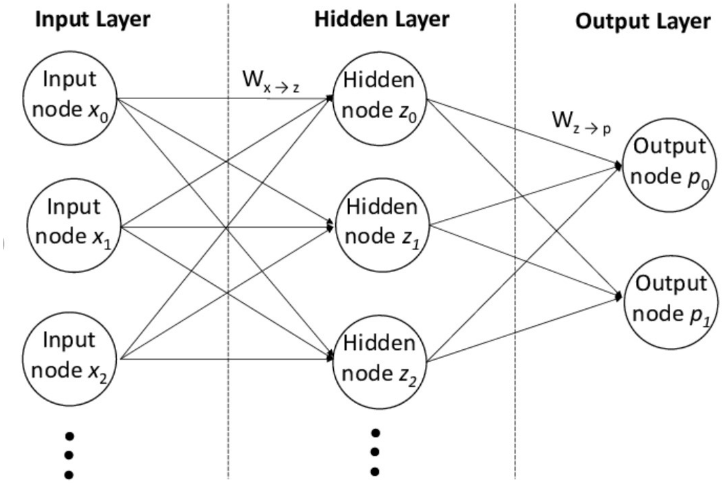
$$\tilde{\boldsymbol{z}} = g(B\boldsymbol{x}) - \frac{1}{2} \tag{22}$$

- Since the nonlinear function g is applied here, the $-1/2$ changes the mapping from $0-1$ to $-1/2$ to $+1/2$
- Now the matrix A of size $m \times k$ is applied on $\tilde{\boldsymbol{z}}$ to get the final output

$$\boldsymbol{z} = \bar{g}(A\tilde{\boldsymbol{z}}) \tag{23}$$

Hidden layers

- The hidden layers can be diagrammatically represented as



Training

- Now both the B and A matrices have to found by training the model
- The error is again defined as

$$f(A_{ls}, B_{ij}) = \sum_{l=1}^{m} (z_l - y_l)^2 \tag{24}$$

- Using the steepest gradient technique as before, we can calculate the partial derivative of this w.r.t. the matrix elements A_{ls} and B_{ij}
- It can be shown that

$$\frac{\partial f}{\partial A_{pq}} = 2\bar{\alpha}(z_p - y_p) z_p (1 - z_p) \tilde{z}_q \tag{25}$$

Training

- Similarly, differentiating w.r.t. B_{pq} it can be shown that

$$\frac{\partial f}{\partial B_{pq}} = \sum_{l=1}^{m} \sigma_l A_{lp} \alpha \left(\frac{1}{2} + \tilde{z}_p\right)\left(\frac{1}{2} - \tilde{z}_p\right) x_q \tag{26}$$

- where

$$\sigma_l = 2\bar{\alpha}(z_l - y_l) z_l (1 - z_l) \tag{27}$$

- Now we can just modify the update algorithm

$$A_{pq} \to A_{pq} - \eta \frac{\partial f}{\partial A_{pq}} \tag{28}$$

$$B_{pq} \to B_{pq} - \eta \frac{\partial f}{\partial B_{pq}} \tag{29}$$

- More generally, the gradient w.r.t. the weights of the neural network nodes are estimated by a method of backward propagation

Different models

- A model with a large number of hidden layers is a deep network
- To generate mode data, noise can be added to the data
- The images can also be tranformed, like left-right inversion, or time inversion, etc.
- Convolutional neural network - image data is processed through some convolution in order to account for neighboring/related pixels
- Taking a large hidden layer (high k) versus many hidden layers - mostly based on experience and experimentation
- Steepest descent leads to the local minimim, might not always work or find the best possible network - therefore need to try many different starting points and train multiple times

Examples

- Toy problem - `https://machinelearningmastery.com/develop-your-first-neural-network-with-pytorch-step-by-st`
- Classify signals into frequency - this can be used for analyzing time dependent data
- Classification of galaxies - `https://medium.com/analytics-vidhya/multiclass-image-classification-problem-convolutional-neu`
- Identifying and classifying phases of matter based on experimental or simulation results

Example

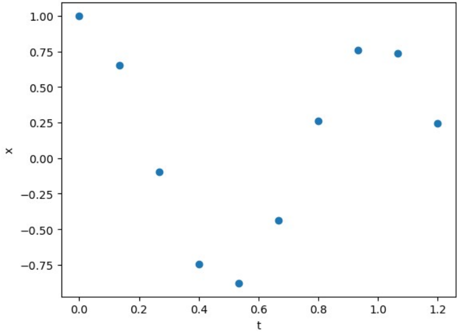
- Consider the input and output examples

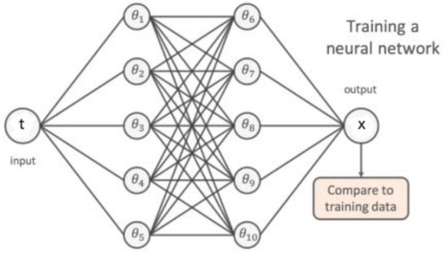
$$
\begin{align}
[0.45, 0.6, -0.27] &\rightarrow 1 \tag{30}\\
[0.85, -0.57, 0.14] &\rightarrow 0 \tag{31}\\
[0.02, 0.96, -0.15] &\rightarrow 1 \tag{32}\\
[-0.63, 0.79, -0.25] &\rightarrow 0 \tag{33}\\
[-0.82, 0.15, 0.45] &\rightarrow 1 \tag{34}\\
[0.48, 0.75, 0.64] &\rightarrow 0 \tag{35}\\
& \tag{36}
\end{align}
$$

- Train a neural network on this data and try to predict the output for new data

Physics informed neural networks

- Neural networks can also be used to solve differential equations describing physical laws
- For example - suppose we experimentally measure the displacement of a particle as a function of time





Training a neural network
t
input
θ_1 θ_2 θ_3 θ_4 θ_5
θ_6 θ_7 θ_8 θ_9 θ_{10}
output
x
Compare to training data

Figure: Ben Mosley

- Now we need to design a neural network that can fit a function $f(t)$ that will approximately give us the value of the displacement $x(t)$

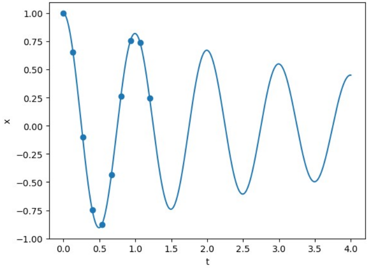
$$f(t) \approx u(t) \tag{37}$$

Physics informed neural networks

- With limited data, the neural network will not be able to extrapolate the displacement to later times
- Rather, if we know that the displacement obeys a differential equation then we can get the full solution

$$m\frac{d^2x}{dt^2} + \mu\frac{dx}{dt} + kx = 0 \tag{38}$$

- This is the equation of a damped harmonic oscillator (SHM if $\mu = 0$) and its solution looks as below



PINN

- Consider the damped harmonic oscillator differential equation

$$m\frac{d^2x}{dt^2} + \mu\frac{dx}{dt} + kx = 0 \tag{39}$$

- with the initial condition $x(t=0) = 1$ and $x'(t=0) = 0$, i.e. an oscillator starting from the peak displacement
- Setup a neural network NN that takes t as an input and gives x as output
- We have to train the network weight factors $\boldsymbol{\theta}$ such that

$$NN(t, \boldsymbol{\theta}) \approx x(t) \tag{40}$$

- We do not need any experimental data to train this network
- The neural network will start generating data itself and the differential equation and boundary conditions will help in training

Loss function

- The network is initialized with random weights
- In each epoch a few input collocation times (t_i, i=1,2,..,N_p) are chosen randomly over the range of interest (including the boundary point) - it is important to train the model over random time points in order to avoid bias towards early or later times
- The network makes predictions $NN(t_i, \boldsymbol{\theta})$
- The loss function consists of two parts, the boundary loss term

$$f_b = \lambda_1 \left(NN(t=0,\theta) - 1 \right)^2 + \lambda_2 \left[\frac{dNN}{dt}(t=0,\theta) - 0 \right]^2 \qquad (41)$$

- and the differential equation loss is (also called physics-loss)

$$f_p = \frac{\lambda_3}{N_p} \sum_{i=1}^{N_p} \left[m \frac{d^2 NN(t_i)}{dt^2} + \mu \frac{dNN(t_i)}{dt} + kNN(t_i) \right]^2 \qquad (42)$$

Grad function

- The weights of the different loss terms $\lambda_1, \lambda_2, \lambda_3$ can be adjusted
- The derivatives dNN/dt and d^2NN/dt^2 need to be calculated

```
from torch.autograd import Variable
t_collocation = np.random.uniform(0,4,size=(500,1)
pt_t_collocation = Variable(torch.from_numpy\n
(t_collocation).float(), requires_grad=True)
```

- After defining a network $NN(time)$ which takes an input *time* tensor, it's derivative can be calculated

```
xn=NN(time)
dxndt=torch.autograd.grad(xn,time)
```

- Similarly, getting the second derivative will involve taking one more grad on the *dxndt* variable

Wave equation

- In order to solve the wave equation for a quantity $u(x,t)$

$$\frac{\partial^2 u}{\partial x^2} - \frac{1}{c^2}\frac{\partial^2 u}{\partial t^2} = 0 \tag{43}$$

- For this we will define a $NN(x,t)$ and the physics loss will be

$$f_p = \frac{1}{N_p}\sum_i \left[\frac{\partial^2 NN(x_i,t_i)}{\partial x^2} - \frac{1}{c^2}\frac{\partial^2 NN(x_i,t_i)}{\partial t^2}\right] \tag{44}$$

- Where the summation is over all collocation points
- For periodic boundary condition the loss is

$$f_b = \sum_j [NN(x_j,t_j) - NN(x_j+L,t_j)]^2 \tag{45}$$

- where x_j is a boundary points and L is the period

Epilogue

- Computational physics and scientific computing is a relatively young field
- However, its growth is very rapid and it is continuing to develop faster
- Not just numerical techniques but hardware also gets more advanced
- Massively parallel computing, graphical processing units, cloud computing
- Parallelization with OpenMP, MPI, GPU, OpenACC
- Quantum computing
- Other courses
  - Computational Solid state physics
  - Computational Particle Physics
  - Computational Modeling of Biological Systems
  - Data Science Analysis