

深度学习讲故事

JW Seminar*

日期: 2024 年 11 月 25 日

蒲俊才的博士论文 [pu-thesis], 使用的理论框架为对 physics-informed neural network(简称 PINN) 进行改进, 为此我们需要先简要了解一下 M. Raissi, P. Perdikaris 和 G.E. Karniadakis 的 PINN 模型 [PINN]

Remark. 如无特别声明, 本文认为神经网络, 网络, 模型是一个东西, 且混用可能机器学习和深度学习, 但本文只有深度学习.

1 背景介绍

在正式开始之前, 需要解释一下基础名词, 这一部分比较简单, 没有什么比较难理解的知识. 实际上神经网络也没什么难理解的, 主要是算法实现和算力限制.

定义 1.1. 万能拟合定律 如果有足够的多数据, 足够多的网络层和神经元, 可以近似任意函数, 解决任何问题. 即通用人工智能 (Artificial general intelligence, 简称 AGI).

1.1 神经网络

定义 1.2. 虽然前面说了有 PINN, 但它仍然是神经网络的一种, 原理如下图所示, 神经网络最左侧称为输入层 (第 0 层), 最右侧称为输出层, 其余为隐藏层. 每层神经网络之间存在偏置 $b^{(k)}$, 权重因子 ($w_{ij}^{(k)}$) 输入信号 x 和激活函数 σ .

如果我们记为 $w_{ij}^{(k)}$ 第 k 层第 j 个神经元指向第 $k+1$ 层第 i 个神经元, 并假设 k 层有 n 个神经元, 则

$$\begin{aligned} a_j^{(k+1)} &= b_j^{(k)} + \sum_{i=1}^n w_{ij}^{(k)} z_i^{(k)} \\ z_i^{(k)} &= h(a_j^{(k)}) \end{aligned} \quad (1)$$

其中偏置和激活函数是需要决定的, 权重因子 $w_{ij}^{(k)}$ 是给定初值, 网络自动更新取得最优值.

这样的话, 如果第 k 层的矩阵综合起来记为 $W^{(k)}$, 就会有

$$W^{(k)} = \begin{pmatrix} W_{11}^{(k)} & W_{12}^{(k)} & \cdots & W_{1n}^{(k)} \\ W_{21}^{(k)} & W_{22}^{(k)} & \cdots & W_{2n}^{(k)} \\ \vdots & \vdots & \ddots & \vdots \\ W_{n1}^{(k)} & W_{n2}^{(k)} & \cdots & W_{nn}^{(k)} \end{pmatrix} \quad (2)$$

则第 k 个网络层的计算可以视为

$$\mathcal{L}_k = W^{(k)} \mathbf{x}^{(d-1)} + \mathbf{b}^{(k)} \quad (3)$$

整个网络层的计算可以视为

$$Output = (\mathcal{L}_d \circ \sigma \circ \mathcal{L}_{d-1} \circ \sigma \cdots \circ \mathcal{L}_2 \circ \sigma \circ \mathcal{L}_1)(\mathbf{x}) \quad (4)$$

*<https://github.com/IceySwan/JW-seminar>

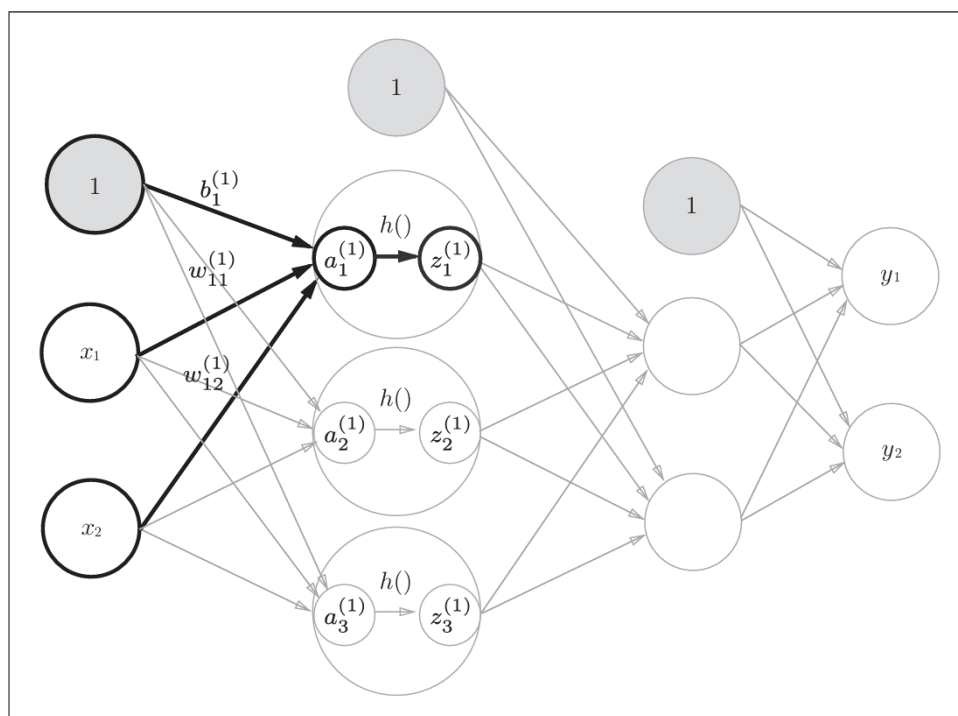


图 1: 常见神经网络示意图

Remark. 输入层为第 0 层, 网络的层数 = 隐藏层 + 输出层. (因为必须有计算才算层数, 输入层仅仅导入数据, 所以不算层数.)

Remark. 实际上神经网络比这个更复杂, 神经元里面那个小圆 a_1 是神经元, 小圆 z_1 是激活函数. 大圆叫做感知机, 它可以接受多个信号, 然后仅输出 0, 1 两种信号. 类似 Sigma 函数. 我们后面的损失函数也是仿照着它设计的. 此外现在还有 Batch-normalization 层, 所以实际上神经网络的每一层应该是 Affine -> Batch Normalization -> Sigma

Remark. Lu Zhou 和 Pu Hongming 的论文 [lu2017expressivepowerneuralnetworks] 论证了神经网络层数和神经元的关系, 即浅层网络可以通过增加神经元的方式达到和深层网络差不多的效果, 但是需要指数级的增加. 这也很好理解, 假设 A 是一个 10 层的网络, 每层 100 个神经元; B 是 100 层的网络, 每层 10 个神经元. 假设每个神经元花费的内存和算力相同, 那么 A 消耗的算力是 100^{10} , B 是 10^{100} . 所以对于深度学习而言, 增加网络层是比增加神经元更好的方法.

定义 1.3. 由于数据量过小, 或者训练方法不对, 只能解决某个特殊问题, 称为过拟合 (Overfitting).

Remark. 虽然深度学习的目标是 AGI, 追求泛化能力, 但是在建立模型的时候还是先追求过拟合, 然后再抑制过拟合, 追求泛化能力. 一般使用 L_2 来抑制过拟合, 如果是特别复杂的网络, 会使用 Dropout 来抑制过拟合.

1.2 激活函数

激活函数一般是神经网络中激活神经元的一种非线性函数, 常用 σ 表示, 一般而言神经网络每层的激活函数都相同, 唯有输出层为 Softmax 映射 (归一化), 常见的激活函数如下所示.

首先是应用最广泛的 Sigmoid 函数和蒲的论文使用的 tanh 函数, 以及归一化的 Softmax 函数.

$$\begin{aligned}\text{sigmoid}(x) &= \frac{1}{1 + e^{-x}}, \\ \tanh(x) &= \frac{e^x - e^{-x}}{e^x + e^{-x}} \\ \text{softmax}(a_k) &= \frac{\exp a_k}{\sum_{i=1}^n \exp a_i}\end{aligned}\tag{5}$$

易见

$$\tanh(x) = 2\text{sigmoid}(2x) - 1\tag{6}$$

且他们的导函数分别为

$$\text{sigmoid}'(x) = \frac{e^{-x}}{(1 + e^{-x})} = \text{sigmoid}(x)(1 - \text{sigmoid}(x))\tag{7}$$

$$\tanh'(x) = \frac{4}{(e^x + e^{-x})^2} = \text{sech}^2(x) = 1 - \tanh^2(x)\tag{8}$$

它们的函数图像见图 (2)

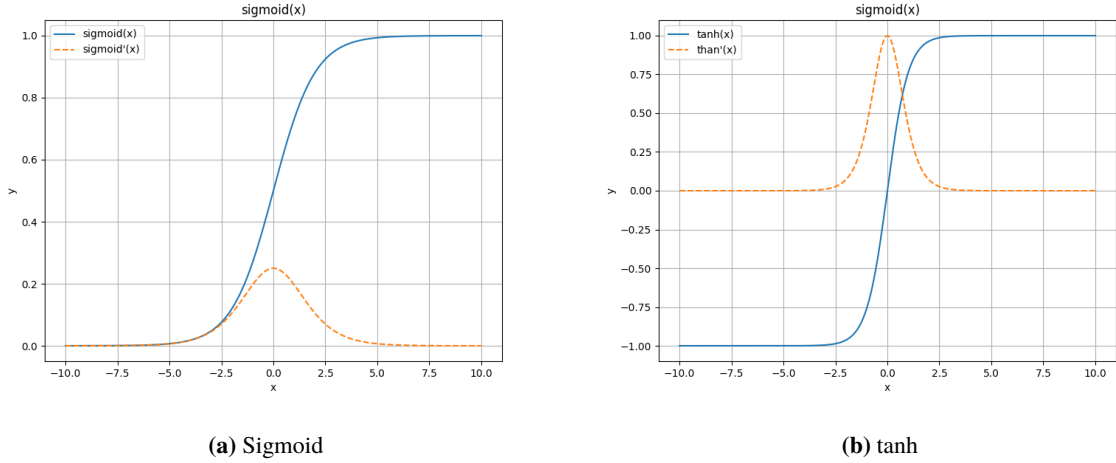


图 2: Sigmoid 和 tanh 函数图

Sigmoid 函数和 tanh 函数在两端都比较平坦, 即其对应的导数趋向于 0. 当使用这些饱和和激活函数 (如 sigmoid) 的神经网络进行反向传播时, 那些输出值在 0 或 1 附近的神经元梯度几乎为 0, 这些神经元通常被称为饱和神经元. 相应地, 这些神经元的权重将无法得到更新, 且与之相连的神经元的权重更新也非常缓慢. 在某些情况下, 最终训练好的网络会收敛到一个很差的局部极小值. 这就是所谓的梯度消失问题.

Remark. tanh 比 sigmoid 更好的点在于它关于原点中心对称, 这种良好的对称性使其对网络训练有极好的效果

1.3 损失函数

损失函数用来衡量网络性能的“恶劣程度”, 所以损失函数越小越好. 常见的损失函数有均方误差 (mean squared error, 简称 MSE) 平方和误差 (Error Sum of Squares, 简称 SSE) 和交叉熵误差 (cross entropy error, 简称 CEE)

$$\begin{aligned}MSE &= \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2 \\ CEE &= - \sum_{i=1}^n [y_i \ln(\hat{y}_i) + (1 - y_i) \ln(1 - \hat{y}_i)]\end{aligned}\tag{9}$$

其中 y_i 是实际值, \hat{y}_i 是预测值, n 是样本数量. 这里面 MSE 一般用作隐藏层 (Affine 层) 的损失函数, CEE 一般用作输出层损失函数, 使得反向传播的结果 “非常简单”. 一般而言, 想要达到 95% 以上的精度, 需要损失函数在 10^{-4} 的量级.

Remark. 这里大致介绍一下反应网络的几个指标

1. Loss: 用来训练网络, 使其达到好结果
2. \mathbb{L}_2 : 抑制过拟合, 一般比 Loss 大一两个量级
3. 正确率: 网络预测结果/真实结果
4. 精度: 最小 “步长”, 如 $1e-4$, $1e-8$ 等.

1.4 反向传播算法和自动微分

如果说前向 (前馈) 传播是由因及果的话, 那反向传播就是由果及因. 前向传播阶段将训练数据输入网络, 经过一系列的矩阵乘法和非线性变换操作, 得到最终的预测结果. 反向传播阶段根据预测结果与真实结果之间的误差, 利用反向自动微分和链式法则, 从后往前逐层计算每个神经元的误差贡献, 以此更新神经元的权重.

在讲反向传播之前, 我们还需要先简要了解一下自动微分和它的缺点. 我们都知道

定义 1.4. 函数 f 导数的定义是:

$$\frac{df}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h} \quad (10)$$

实际上计算机并不能取到无穷小, 如果我们设 $h = 10^{-10}$, 那么在 float32 类型下计算会因为精度问题输出为 0, 这时候就会出现梯度消失, 因为 float32 的精度约为 10^{-7} . 为此, 我们常用 float64 类型来提高精度, 但它占用的内存和计算开销通常是 float32 的两倍. 在机器学习中, 由于 float32 精度已经足够, 因此通常使用 float32 类型. 当然, 我们也可以使用中心求导等方式减小误差. 为此我们选用反向传播算法, 它从原理上越过了舍入误差, 并且极大的加快了训练速度.

Remark. 我曾经测试过这两个算法的区别, 反向传播使用 300MB, 7 min 完成推理, 数值微分直接内存跑到了 16 GB 关机了. 后来多次调整之后, 数值微分 15.5 GB 内存, 1 h 完成的项目, 反向传播只需要 227 MB, 7 min 完成.

图 (3) 是误差反向传播算法的原理示意图

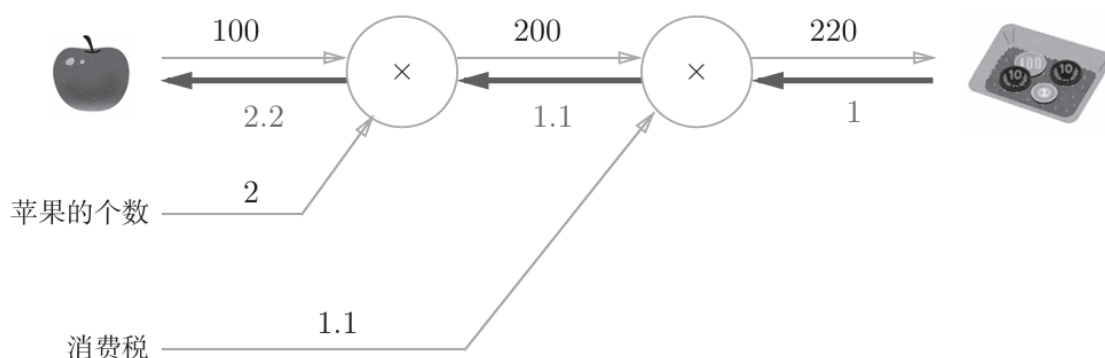


图 3: 反向传播原理

反向传播和常规求导不同在于, 常规求导为 $y' = f(x)$, 反向传播使用 $y = F(y)$ 表示, 如上面的 (7) 式, 另外如果设 $Y = XW + B$ 为矩阵, L 为 Y 的损失函数 (标量), 则反向传播为

$$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y} W^T, \quad \frac{\partial L}{\partial W} = X^T \frac{\partial L}{\partial Y} \quad (11)$$

Remark. 注意这里并不是数学意义上的求导, 不妨假设 $X_{m \times n}, W_{n \times k}, Y_{m \times k}$, 则 $\frac{\partial L}{\partial X}$ 为 $m \times n$ 维矩阵, $\frac{\partial L}{\partial W}$ 为 $n \times k$ 阶矩阵, 这个形式是为了满足维度.

同样的, 机器学习里面的“梯度下降”也不是我们数学上的梯度下降, 而是

$$x_0 = x_0 - \eta \frac{\partial f}{\partial x_0}, \quad x_1 = x_1 - \eta \frac{\partial f}{\partial x_1} \quad (12)$$

其中 η 表示更新量, 在神经网络的学习中, 称为学习率 (learning rate). 学习率决定在一次学习中, 应该学习多少, 以及在多大程度上更新参数. (12) 式是表示更新一次的式子, 这个步骤会反复执行. 也就是说, 每一步都按上更新变量的值, 通过反复执行此步骤, 逐渐减小函数值. 虽然这里只展示了有两个变量时的更新过程, 但是即便增加变量的数量, 也可以通过类似的式子 (各个变量的偏导数) 进行更新. 学习率需要事先确定为某个值, 比如 0.01 或 0.001. 然后根据训练结果逐步调整. 一般而言, 这个值过大或过小, 都无法抵达一个“好的位置”. 一般的取值范围为 $[10^{-3}, 10^{-2}]$.

而且哪怕 lr 取得比较好, 梯度法也不一定能达到最小值, 我们都知道梯度指向的是函数下降的方向, 它必然是极小值但不一定是最小值, 此外, 当函数很复杂且呈扁平状时, 还会有“鞍点”, 这就是梯度消失. 此时学习会进入一个 (几乎) 平坦的地区, 陷入被称为“学习高原”的无法前进的停滞期。

1.5 Batch 和 mini-Batch

这样设想一下, 我们想要训练的数据量称为 Batch, 假设有 100 个数据, 很轻松就能直接训练完看结果如何. 但是假设有一千万乃至一亿数据, 还是一下都训练完的话, 很可能需要数天乃至数月的时间, 所以我们想要先试出来一个比较好的学习率 lr, 然后再进行训练. 所以需要抽样, 也就是 mini-Batch. 比方我们想要先抽 100 个, 那么 mini-Batch 就是 100.

说到这里, 差不多前置知识就说完了, 总结一下, 对于深度学习, 只需要确定激活函数, 学习率和 mini-Batch (也成为超参数), 其余参数均交给网络自己决定.

1.6 权重初始值与 Xavier 正则化

接下来我们继续想办法解决梯度消失/爆炸问题. Xavier 是目前常用的正则化方法, 其基本思想是通过适当的随机初始化 (一般为 Gauss 分布), 使得该层输入个数不影响每层输出的方差, 该层输出个数也不影响每层梯度的方差, 从而加快网络训练速度并提高网络训练精度. Xavier 初始化的具体实现为假设 W 为 $n_{in} \times n_{out}$ 维的权重矩阵, 其中 n_{in} 和 n_{out} 分别表示该层输入输出. 则 W 的初始化应该满足

$$Var(M) = \frac{2}{n_{in} + n_{out}} \quad (13)$$

其中 Var 表示方差. 蒲的论文也是用的 Xavier 正则化.

1.7 优化器

接下来说一下怎么更新网络信息, 寻找最小 Loss, 即优化器. 时间紧迫, 我们只说一个 Adam[Adam] 优化器, 这也是目前业界主流的优化器, 它结合了 Momentum 和 Adagrad 的优点, 通过一阶和二阶矩¹的信息来调整学习率, 提升训练速度和收敛效果.

Adam 算法核心思想是根据历史梯度的平均值和方差来自适应地调整学习率. 具体地, 对于每个参数, Adam 算法维护分别表示梯度的一阶矩和二阶矩估计的两个变量 \mathbf{m} 和 \mathbf{v} . 这里 \mathbf{m} 和 \mathbf{v} 的初始值设置为 0, 在每次迭代中先计算梯度 g_t , 然后再分别更新 \mathbf{m} 和 \mathbf{v} , 具体过程如下

$$\mathbf{m}_t \leftarrow \beta_1 \mathbf{m}_{t-1} + (1 - \beta_1) g_t, \quad \mathbf{v}_t \leftarrow \beta_2 \mathbf{v}_{t-1} + (1 - \beta_2) g_t^2 \quad (14)$$

¹这里一阶和二阶矩为均值和方差

其中 β_1, β_2 分别表示指数加权移动平均的衰减率 (通常取值为 0.9 和 0.999), g_t 表示在当前迭代中计算得到的梯度. 为了减少偏差, 上述更新公式 m 和 v 的初始值都需要进行偏置修正后再来计算参数更新量, 具体表达式如下:

$$\hat{m}_t \leftarrow \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t \leftarrow \frac{v_t}{1 - \beta_2^t}, \quad \theta_{t+1} \leftarrow \theta_t - \frac{\alpha \cdot \hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \quad (15)$$

其中 η 表示学习率, ϵ 是一个很小的常数, 用于防止分母为 0 的情况发生. 相比于其他算法, Adam 算法具有以下优点

- 自适应学习率: Adam 对每个参数采用不同的学习率, 自适应地调整学习率。
- 收敛性好: 使用了动量项², Adam 可以加速收敛, 减少振荡.³
- 适合稀疏梯度: Adam 在高维空间和稀疏数据 (如文本或图像) 上有较好表现
- 计算效率高: Adam 的计算效率较高, 内存需求低.

2 PINN

虽然叫物理信息神经网络, 但是既不物理, 也不信息. 并没有什么创新的原理, 一般的网络只计算数据 (取样点) 引起的误差, PINN 首次将函数本身作为误差的一部分. 即

$$Loss = Loss_{data} + Loss_f \quad (16)$$

这样的好处在于可以使用很少的样本点 (几十到上百) 即可以高精度的预测 PDE 的解和模型的位置参数, 而一般的网络的样本却需要几万起步. 但是 PINN 的缺点是必须知道方程 (的形式), 以及初边值条件.

通过上面损失函数我们可以把 PINN 网络看成两部分, 即通用神经网络和包含方程信息的物理信息网络. 以下我们通过这个来自 [PINN] 的例子解释一下 PINN 的结构和优势

2.1 实例

例 2.1. The nonlinear Schrödinger equation along with periodic boundary conditions is given by

$$\begin{cases} ih_t + 0.5h_{xx} + |h|^2h = 0, & x \in [-5, 5], t \in [0, \pi/2], \\ h(0, x) = 2\text{sech}(x), \\ h(t, -5) = h(t, 5), \\ h_x(t, -5) = h_x(t, 5). \end{cases} \quad (17)$$

Here, $h(t, x)$ is the complex-valued solution. Let us define $f(x, t)$ to be given by:

$$f := ih_t + 0.5h_{xx} + |h|^2h. \quad (18)$$

A neural network prior is placed on $h(t, x)$, with $h(t, x) = [u(t, x), v(t, x)]$, where $u(t, x)$ and $v(t, x)$ represent the real and imaginary parts, respectively. The shared parameters are optimized by minimizing the mean squared error loss:

$$MSE = MSE_0 + MSE_b + MSE_f, \quad (19)$$

²这里的动量项也不是物理上的动量, 而是 Momentum 项

³在计算梯度时不仅考虑当前的梯度, 还考虑历史梯度的平均值; 使用了自适应的二阶矩估计, 即在计算梯度时不仅考虑历史梯度和当前梯度的平均值, 而且考虑了二阶矩的平均值.

where:

$$\begin{aligned} \text{MSE}_0 &= \frac{1}{N_0} \sum_{i=1}^{N_0} |h(0, x_i^0) - h_i^0|^2, \\ \text{MSE}_b &= \frac{1}{N_b} \sum_{i=1}^{N_b} \left(|h(t_i^b, -5) - h(t_i^b, 5)|^2 + |h_x(t_i^b, -5) - h_x(t_i^b, 5)|^2 \right), \\ \text{MSE}_f &= \frac{1}{N_f} \sum_{i=1}^{N_f} |f(t_i^f, x_i^f)|^2. \end{aligned}$$

Here, $\{x_0^i, h_0^i\}_{i=1}^{N_0}$ denotes the initial data. $\{t_b^i\}_{i=1}^{N_b}$ corresponds to the collocation points on the boundary, and $\{t_f^i, x_f^i\}_{i=1}^{N_f}$ represents the collocation points on $f(t, x)$. Consequently, MSE_0 corresponds to the loss on the initial data, MSE_b enforces the periodic boundary conditions, and MSE_f penalizes the Schrödinger equation not being satisfied on the collocation points.

2.1.1 Neural Network Setup

To evaluate the accuracy of our method, we simulated equation (17) using conventional spectral methods to generate a high-resolution dataset. The setup includes:

1. Initial state: $h(0, x) = 2 \text{sech}(x)$.
2. Periodic boundary conditions: $h(t, -5) = h(t, 5)$, $h_x(t, -5) = h_x(t, 5)$.
3. Numerical methods: Integration up to $t = \pi/2$ using the Chebfun package with:
 - Spectral Fourier discretization with 256 modes.
 - Fourth-order explicit Runge-Kutta time integrator.
 - Time step $\Delta t = \pi/2 \times 10^{-6}$.

The training set includes:

- Initial data $\{x_i^0, h_i^0\}_{i=1}^{N_0}$ on $h(0, x)$, where $N_0 = 50$,
- Boundary collocation points $\{t_b^i\}_{i=1}^{N_b}$, where $N_b = 50$,
- Collocation points $\{t_f^i, x_f^i\}_{i=1}^{N_f}$ with $N_f = 20,000$ for enforcing the equation (17) inside the domain.

Here all point locations were generated using the Latin Hypercube Sampling.

The spatio-temporal solution $h(t, x)$ is represented as $h(t, x) = u(t, x) + iv(t, x)$ using a 5-layer neural network with 100 neurons per layer, and setting $\tanh x$ as the activation function.

2.1.2 Results and Discussion

Fig. 4 summarizes the experimental results, and $\mathbb{L}_2 = 1.97 \times 10^{-3}$.

最后说一下 PINN 的缺点:

1. Loss 既是成功也是失败, 把物理方程引入 Loss, 会导致 Loss 变大 (相比只有 data), 但是又不知道哪一部分变大. 而且一般如果写成 $h = u + iv$, 一般 u, v 中必有一个收敛慢
2. 精度差, 理论极限在 10^{-6}
3. 对超参数敏感, 需要不断调试得到合适的超参数
4. 对于非对称区域, 尤其是怪波拟合并不好. (使用 IPINN)

代码展示 (tf1 to torch)

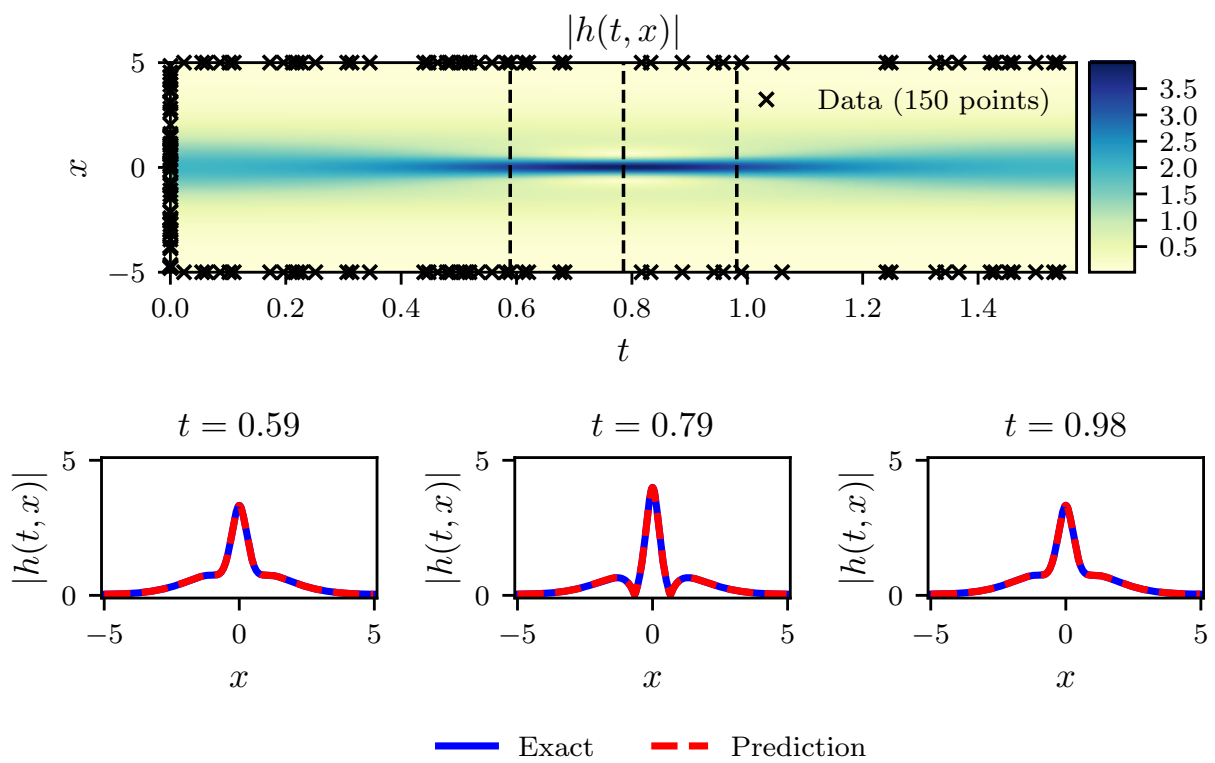


图 4: PiNN NLS