How to use Neural Network to Solve PDEs?

9月工作：

编程部分：

- 复习了C++语法，学习了Eigen、GSL等专门用于科学计算的库
- 学习了CUDA编程中grid, block, thread概念，了解了并行处理的过程；
- 使用CUDA编程实现了较为简单的数值算法，对于Block，grid等应用不够熟练，总是globalIdx溢出边界。

理论部分：

- 重点了解了PFNN及其衍生出了的子方法（PFNN-2）、DGM、PINN、Ritz算法等，完成了公式推导。

暂定未来计划：

- 对本科接触过的一些经典的矩阵分解方法(SVD分解)、利用等价类思想或是方法构造置换矩阵，把矩阵分解为矩阵块，运行在不同的block/grid上。
  - 目前一大困难：在C++上定义50000阶以上的矩阵特别困难，内存溢出，应该是还没有找对方法和适用的库。

$$T_n^{(p,q)} := \begin{bmatrix} d_1 & & & a_1 & & & \\ & d_2 & & & a_2 & & \\ & & \ddots & & & \ddots & \\ & & & \ddots & & & a_{n-p} \\ & & & & \ddots & & \\ b_1 & & & & & \ddots & \\ & \ddots & & & & & \ddots \\ & & b_{n-q} & & & & d_n \end{bmatrix} \in \mathbb{R}^{n\times n}$$

$$Q^T P^T T_n^{(p,q)} P Q = \begin{cases} R_{\overline{1}} \oplus R_{\overline{2}} \oplus \cdots \oplus R_{\overline{q-p}}, & \text{if } n \text{ is even}, \\ R_{\overline{1}} \oplus R_{\overline{2}} \oplus \cdots \oplus R_{\overline{q-p}} \oplus d_p, & \text{if } n \text{ is odd}. \end{cases}$$

- 继续阅读PFNN、Ritz等研究方法后续的改进，重点关注domain decomposition部分的处理。
- 继续寻找数值代数与PDE工作的结合点。

Deep Galerkin Method (DGM)

Formula Derivation

For a parabolic PDE with d spatial dimensions:

$$
\begin{aligned}
&\frac{\partial u}{\partial t}(t,x) + \mathcal{L}u(t,x) = 0, \quad (t,x) \in [0,T] \times \Omega, \\
&u(t=0,x) = u_0(x), \\
&u(t,x) = g(t,x), \quad x \in \partial\Omega,
\end{aligned}
$$

where $x \in \Omega \subset \mathbb{R}^b$. Then the objective function can be constructed as

$$
\begin{aligned}
J(f) =& \left\| \frac{\partial u}{\partial t}(t,x;\theta) + \mathcal{L}f(t,x;\theta) \right\|^2_{[0,T]\times\Omega,\nu_1} + \|f(t,x;\theta) - g(t,x)\|^2_{[0,T]\times\partial\Omega,\nu_2} + \|f(0,x;\theta) - u_0(x)\|^2_{\Omega,\nu_3} \\
=& J_1(f) + J_2(f) + J_3(f),
\end{aligned}
$$

where $\theta \in \mathbb{R}^K$ are the neural network's parameters, ν_1 and ν_2 are the probability densities.

Then the DGM Neural Network can be summarized by four steps:

- Randomly select points $s_n = \{(t_n, x_n), (\tau_n, z_n), w_n\}$ based on the domains of $J_1(f)$, $J_2(f)$, and $J_3(f)$.
- Calculate the squared error $G(\theta_n, s_n)$ where:

$$
\begin{aligned}
G(\theta_n, s_n) =& (\frac{\partial f}{\partial t}(t_n, x_n; \theta_n) + \mathcal{L}f(t_n, x_n; \theta_n))^2 + (f(\tau_n, z_n; \theta_n) - g(\tau_n, z_n))^2 + (f(0, w_n; \theta_n) - w_0(w_n) \\
=& G_1(\theta_n, s_n) + G_2(\theta_n, s_n) + G_3(\theta_n, s_n).
\end{aligned}
$$

- Take a descent step at the random point s_n:

$$
\theta_{n+1} = \theta_n - \alpha_n \nabla_\theta G(\theta_n, s_n).
$$

- Repeat until convergence criterion is satisfied.

Additionally, for equations with complex second derivatives, the DGM algorithm establishes the assumption that the second derivatives in $\mathcal{L}f(t,x;\theta)$ is of the form $\frac{1}{2}\Sigma_{i,j=1}^{d}\rho_{i,j}\sigma_i(x)\sigma_j(x)\frac{\partial^2 f}{\partial x_i x_j}(t,x;\theta)$. Subsequently, the estimated value $\tilde{G}(\theta_n, s_n)$ of $G(\theta_n, s_n)$ is utilized as the loss function. Thus a neural network can be trained to find the optimal parameters θ.

Definition of the Network Architecture

The DGM neural network draws inspiration from the LSTM architecture, and its structure is defined as follows:

$$
\begin{aligned}
S^1 &= \sigma(W_1 \vec{x} + b^1), \\
Z^l &= \sigma(U^{z,l}\vec{x} + W_{z,l} S^l + b^{z,l}), \quad l = 1, \ldots, L, \\
G^l &= \sigma(U^{g,l}\vec{x} + W^{g,l} S^1 + b^{g,l}), \quad l = 1, \ldots, L, \\
R^l &= \sigma(U^{r,l}\vec{x} + W^{r,l} S^l + b^{r,l}), \quad l = 1, \ldots, L, \\
H^l &= \sigma(U^{h,l}\vec{x} + W^{h,l}(S^l \odot R^l) + b^{h,l}), \quad l = 1, \ldots, L, \\
S^{l+1} &= (1 - G^l) \odot H^l + Z^l \odot S^l, \quad l = 1, \ldots, L, \\
f(t,x;\theta) &= W S^{L+1} + b,
\end{aligned}
$$

where $\vec{x} = (t, x)$, the number of hidden layers is $L + 1$, and $\odot$ denotes element-wise multiplication. The network architecture is defined as follows:

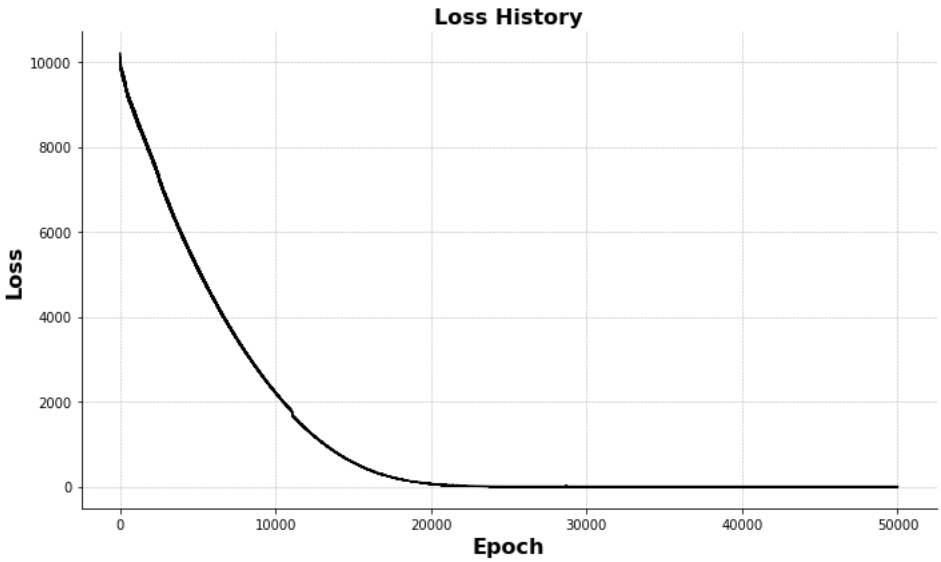
```
class DGMCell(nn.Module):
  def __init__(self, input_dim, hidden_dim, n_layers=3, output_dim=1):
    super(DGMCell, self).__init__()
    self.input_dim = input_dim
    self.hidden_dim = hidden_dim
    self.output_dim = output_dim
    self.n = n_layers

    self.sig_act = nn.Tanh()

    self.Sw = nn.Linear(self.input_dim, self.hidden_dim)

    self.Uz = nn.Linear(self.input_dim, self.hidden_dim)
    self.Wsz = nn.Linear(self.hidden_dim, self.hidden_dim)
    self.b_z = nn.Parameter(torch.zeros(self.hidden_dim))

    self.Ug = nn.Linear(self.input_dim, self.hidden_dim)
    self.Wsg = nn.Linear(self.hidden_dim, self.hidden_dim)
    self.b_g = nn.Parameter(torch.zeros(self.hidden_dim))

    self.Ur = nn.Linear(self.input_dim, self.hidden_dim)
    self.Wsr = nn.Linear(self.hidden_dim, self.hidden_dim)
    self.b_r = nn.Parameter(torch.zeros(self.hidden_dim))

    self.Uh = nn.Linear(self.input_dim, self.hidden_dim)
    self.Wsh = nn.Linear(self.hidden_dim, self.hidden_dim)
    self.b_h = nn.Parameter(torch.zeros(self.hidden_dim))

    self.Wf = nn.Linear(hidden_dim, output_dim)
    self.b  =  nn.Parameter(torch.zeros(self.output_dim))

  def forward(self, x):
    S1 = self.Sw(x)
    for i in range(self.n):
      if i==0:
        S = S1
      else:
        S = self.sig_act(out)
      Z = self.sig_act(self.Uz(x) + self.Wsz(S)+self.b_z)
      G = self.sig_act(self.Ug(x) + self.Wsg(S1)+self.b_g)
      R = self.sig_act(self.Ur(x) + self.Wsr(S)+self.b_r)
      H = self.sig_act(self.Uh(x) + self.Wsh(S*R)+self.b_h)
      out = (1-G)*H + Z*S
    out = self.Wf(out)+self.b
    return out
```

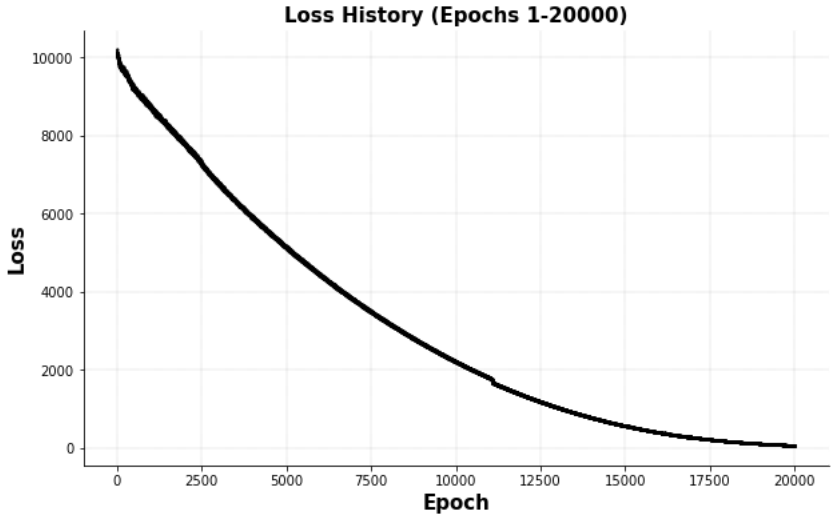
论文复现(Black-Scholes Pricing PDE)

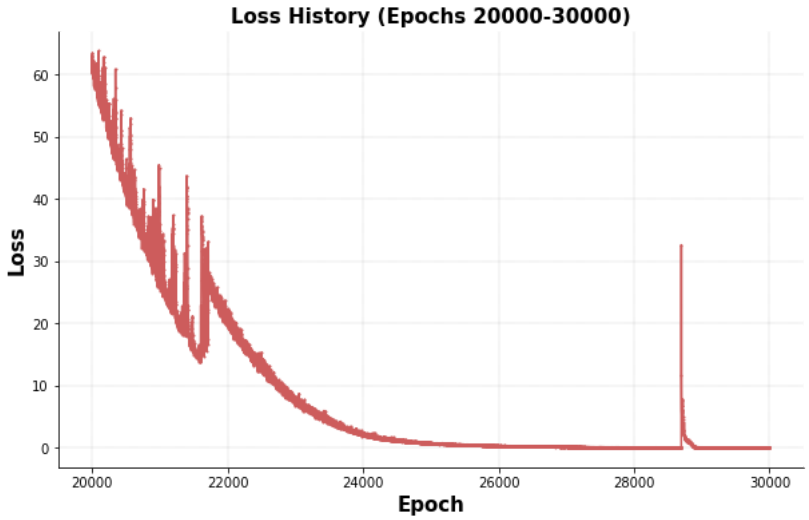
$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0$$

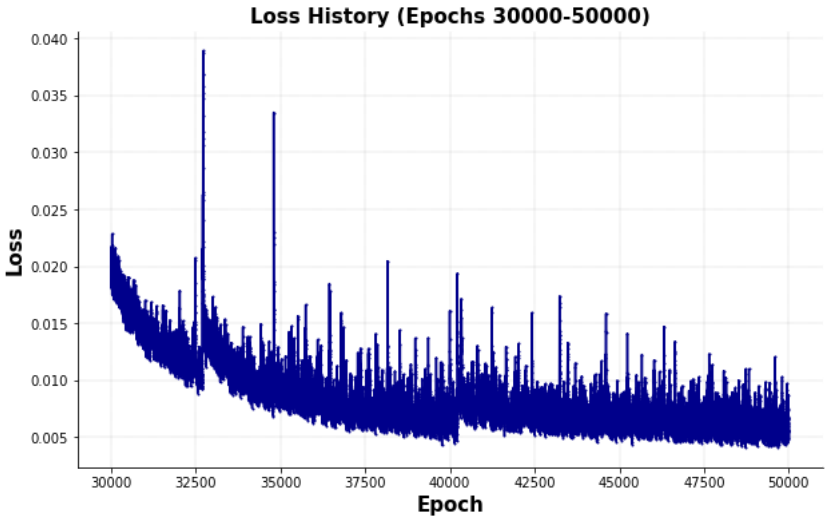
- V 是欧式期权的价格（期权的市值）。

- t 是时间。
- S 是标的资产（例如股票）的价格。
- σ 是标的资产的波动率。
- r 是无风险利率。









Penalty-Free Neural Network(PFNN)

PFNN-Original

《Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations》

$$\begin{cases} -\nabla \cdot (\rho(|\nabla u|)\nabla u) + h(u) = 0, & \text{in } \Omega \subset \mathbb{R}^d, \\ u = \varphi, & \text{on } \Gamma_D, \\ \rho(|\nabla u|)\nabla u \cdot n = \psi, & \text{on } \Gamma_N, \end{cases}$$

算法步骤

- Step1: 构造Euler-Lagrange equation of the energy functional

$$I[w] = \int_\Omega (P(w) + H(w))\, dx - \int_{\partial N} \psi w\, dx$$

$P(w) = \int_0^{|\nabla w|} \rho(s)$代表Potential Energy，$H(w) = \int_0^w h(s)\, ds$代表Hamiltonian Energy。

- Step2: 构造$w_\theta(x) = g_{\theta_1}(x) + l(x) f_{\theta_2}(x)$
- Step3: $g_{\theta_1}(x)$: 处理的是Γ_D边界

$$\Phi[g_{\theta_1}] := \sum_{x^i \in S(\Gamma_D)} (\varphi(x^i) - g_{\theta_1}(x^i))^2$$

- Step4: $f_{\theta_2}(x)$:

$$u^* = \arg\min_{w \in H} \Psi[w]$$

$$\Psi[w] := \frac{|\Omega|}{\#S(\Omega)} \sum_{x^i \in S(\Omega)} (P(w(x^i)) + H(w(x^i))) - \frac{|\Gamma_N|}{\#S(\Gamma_N)} \sum_{x^i \in S(\Gamma_N)} (\psi(x^i) w(x^i))$$

- Step5: $l(x)$: 选取了逆多重二次径向基函数来构造length factor function。

$$\begin{cases} l(x) = 0, & x \in \Gamma_D, \\ l(x) > 0, & otherwise. \end{cases}$$

$$\begin{cases} l_k(x) = 0, & x \in \gamma_k, \\ l_k(x) = 1, & x \in \gamma_{k_0}, \\ 0 < l_k(x) < 1, & \text{otherwise.} \end{cases}$$

$$l(x) = \frac{\tilde{l}(x)}{\max\limits_{x \in \Omega} \tilde{l}(x)}, \quad \tilde{l}(x) = \prod_{k \in k | \gamma_k \subset \Gamma_D} 1 - (1 - l_k(x))^u,$$

$$l_k(x) = \sum_{i=1}^{m_k} a_i \phi(x; \hat{x}^{k,i} + b \cdot x + c),$$

$$\phi(x, \tilde{x}) = (e^2 + ||x - \hat{x}||^2)^{-1/2}.$$

- Step6: 训练网络求解即可

Advantages

- 利用泛函把原方程转为了一个弱形式(weak form)，这样避免了二次求导，函数也不必在整个Ω上处处二次可导
- 没有添加任何penalty terms，训练速度较快(引入了length factor function)。
- 在较复杂的几何问题上表现也较好，length factor functionl_k可以通过从$S(\gamma_k \cup \gamma_{k_o})$中的插值节点来构造。
- 以往学者也有提出来和$w_\theta(x) = g_{\theta_1}(x) + l(x) f_{\theta_2}(x)$很类似的思路$w_\theta(x) = G(x) + L(x) f_\theta(x)$，但是其对于$G(x)$大多构造成spline interpolations，仅仅适用于低维度空间中的简单几何形状。
- 用来解决self-adjoint problems with complex geometries（在之后的改进中拓展到了non-self adjoint time-dependent differential equations）。

PFNN-2: A Domain Decomposed Penalty-Free Neural Network Method for Solving Partial Differential Equations

算法步骤-without domain decomposition

the case of a single sub-domain:

$$\begin{cases} \frac{\partial u}{\partial t} - \nabla \cdot (A \nabla u - B) + C = 0, & \text{in } \Omega \times (0, T], \\ u = r_D, & \text{on } \Gamma_D \times (0, T], \\ (A \nabla u) \cdot n = r_N, & \text{on } \Gamma_N \times (0, T], \\ u = r_0, & \text{in } \Omega \times \{0\}, \end{cases}$$

Dirichlet Boundary: $u = r_D, \text{on } \Gamma_D \times (0, T], u = r_0, \text{in } \Omega \times \{0\}$;

Neumann Boundary: $(A \nabla u) \cdot n = r_N, \text{on } \Gamma_N \times (0, T]$。

- Step1： 构造$w_\theta(x) = g_{\theta_1}(x) + l(x) f_{\theta_2}(x)$;
- Step2：使用spline functions构造length factor函数$l(x)$;

$$l(x, \mathrm{t}) = \frac{\tilde{l}(x, \mathrm{t})}{\max\limits_{(\hat{x}, \hat{\mathrm{t}}) \in \Omega \times (0, \mathrm{T}]} \tilde{l}(\hat{x}, \hat{\mathrm{t}})}, \quad \tilde{l}(x, t) = \frac{t}{T} \prod_{j=1} 1 - (1 - l_k(x))^u,$$

- Step3：构造compactly supported test functions作为公式中的v。

$$v_s(x,t;\hat{x}_s,\hat{t}_s,h_s)=\max\left(1-\frac{|t-\hat{t}_s|}{h_s},0\right)\prod_{j=1}^{d_x}\max\left(1-\frac{|x_j-\hat{x}_{s,j}|}{h_s},0\right),$$
$$s=1,2,\ldots,n_v,\quad n_v\in\mathbb{N}^+$$

- Step4：构造$g_{\theta_1}(x)$的训练函数：

$$\Psi[g_{\theta_1}]:=\sum_{(x,t)\in S(\Gamma_D\times(0,T])}(g_{\theta_1}(x,t)-r_D(x,t))^2+\sum_{(x,t)\in S(\bar{\Omega}_i)\times\{0\}}g_{\theta_1}(x,t)-r_0(x,t))^2$$

- Step5：构造$f_{\theta_2}(x)$的训练函数：

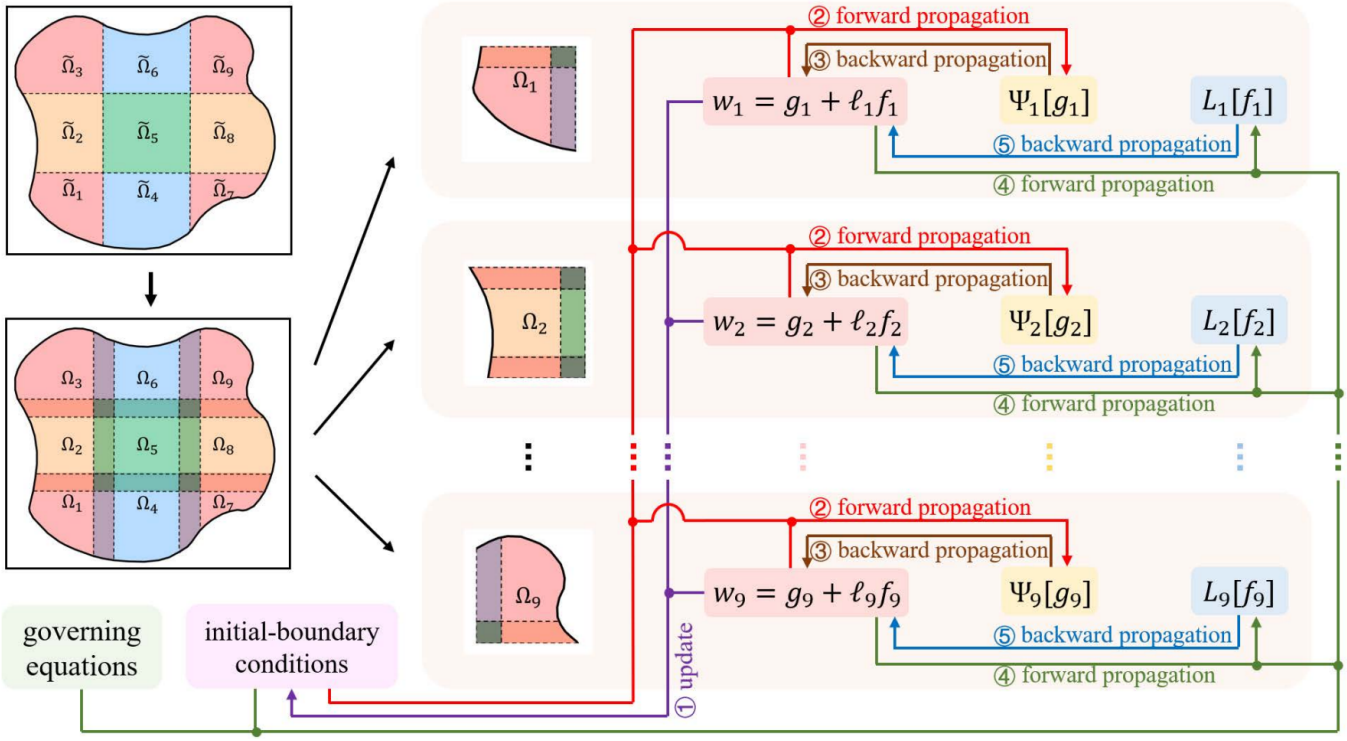
$$u^*=\arg\min_{\widetilde{w}_{\theta_2}}L[\widetilde{w}_{\theta_2}]=\arg\min_{\widetilde{w}_{\theta_2}}\sum_{s=1}^{n_v}R^2[\widetilde{w}_{\theta_2};v_s],$$

$$\begin{aligned}R[\widetilde{w}_{\theta_2};v_s]\quad:=&\frac{|\Omega|T}{\#S(\Omega\times(0,T])}\sum_{(x,t)\in S(\Omega\times[0,T])}\left[\mathcal{A}\nabla\widetilde{w}_{\theta_2}\cdot\nabla v_s+\left(\frac{\partial\widetilde{w}_{\theta_2}}{\partial t}+\nabla\cdot\mathcal{B}+\mathcal{C}\right)v_s\right](x,t)\\&-\frac{|\Gamma_N|T}{\#S(\Gamma_N\times(0,T])}\sum_{(x,t)\in S(\Gamma_N\times[0,T])}[r_N v_s](x,t)\end{aligned}\tag{2.11}$$

算法步骤-with domain decomposition

$$\begin{cases}\dfrac{\partial u_i^k}{\partial t}-\nabla\cdot(\mathcal{A}\nabla u_i^k-\mathcal{B})+\mathcal{C} &=0, \quad \text{in } \Omega_i\times(0,T],\\ u_i^k &=u^{k-1}, \quad \text{on } \Gamma_i\times(0,T],\\ u_i^k &=r_D, \quad \text{on } \Gamma_{i,D}\times(0,T],\\ (\mathcal{A}\nabla u_i^k)\cdot\boldsymbol{n} &=r_N, \quad \text{on } \Gamma_{i,N}\times(0,T],\\ u_i^k &=r_0, \quad \text{on } \overline{\Omega}_i\times\{0\}\end{cases}\tag{3.1}$$

- 将需要计算的区域切分为一组不互相重叠的子域$\{\tilde{\Omega}_i\}_{i=1}^m$；
- 再将每个子域$\{\tilde{\Omega}_i\}_{i=1}^m$拓展至一系列相互重叠的子域$\{\Omega_i\}_{i=1}^m$，$m$代表的是数量。
- 根据切分后的结果重新定义$\Gamma_{i,D}=\partial\Omega_i\cap\Gamma_D,\quad \Gamma_{i,N}=\partial\Omega_i\cap\Gamma_N,\quad \text{and}\quad \Gamma_i=\partial\Omega_i\setminus(\Gamma_D\cup\Gamma_N)$.
- 假设在第k轮的迭代时子域Ω_i上的神经网络u为u_i^k，after solving the subdomain problems for iteration k, the approximate solution defined on the whole domain is composed as $u^k|_{\tilde{\Omega}_i}=u_i^k$.



- 和之前没有进行domain decomposition时一样定义原问题的弱形式（weak form），对u进行多次训练、子域划分因此需要给出u的新定义。
- 给出$w_i = g_i + l_i f_i$中g_i和f_i的训练函数(Loss Function)，进行训练即可。

Algorithm 1: Computing the i-th local approximate solution in PFNN-2.

Input: i, m, g_i^0 f_i^0, ℓ_i, K_o (number of outer iteration), K_i (number of inner iteration).

Output: $w_i^{K_o}$.

1 Let $w_i^0 = g_i^0 + \ell_i f_i^0$.
2 **for** $k = 1,2,\cdots,K_o$ **do**
3 /* Online stage: line 4 to 13 */
4 **for** $j = 1,2,\cdots,m$ **do**
5 **if** $i < j$ **then**
6 Send $w^{k-1}|_{\Gamma_j \cap \Omega_i} = w_i^{k-1}$ to processor j.
7 Receive $w^{k-1}|_{\Gamma_i \cap \Omega_j} = w_j^{k-1}$ from processor j.
8 **end**
9 **if** $i > j$ **then**
10 Receive $w^{k-1}|_{\Gamma_i \cap \Omega_j} = w_j^{k-1}$ from processor j.
11 Send $w^{k-1}|_{\Gamma_j \cap \Omega_i} = w_i^{k-1}$ to processor j.
12 **end**
13 **end**
14 /* Offline stage: line 15 to 23 */
15 **for** $t = 1,2,\cdots,K_i$ **do**
16 Forward propagation: evaluate the loss function $\Psi_i^k[g_i]$ according to (3.4).
17 Backward propagation: calculate the gradients of $\Psi_i^k[g_i]$ and minimize $\Psi_i^k[g_i]$ to get the updated g_i^k.
18 **end**
19 **for** $t = 1,2,\cdots,K_i$ **do**
20 Forward propagation: evaluate the loss function $L_i^k[f_i]$ according to (3.5).
21 Backward propagation: calculate the gradients of $L_i^k[f_i]$ and minimize $L_i^k[f_i]$ to get the updated f_i^k.
22 **end**
23 Let $w_i^k = g_i^k + \ell_i f_i^k$.
24 **end**

公式推导

$$
\begin{aligned}
-\int_0^T \int_\Omega \nabla \cdot (A\nabla u) v \, dxdt &= -[(A\nabla u)\cdot v]_{\Omega\times(0,T]} + \int_0^T \int_\Omega A\nabla u \nabla v \, dxdt \\
&= \int_0^T \int_\Omega A\nabla u \nabla v \, dxdt
\end{aligned}
$$

代入：

$$
-\int_0^T \int_\Omega [\frac{\partial u}{\partial t}\cdot v + A\nabla u \nabla v + \nabla B \cdot v + C \cdot v] dxdt + -\int_0^T \int_{\Gamma_N} r_N v dxdt
$$

即得到了原问题的弱形式。

Advantages

1. 适用于non-self adjoint time-dependent differential equations；
2. 引入了一种重叠域分解策略(overlapping domain decomposition)，提高了训练效率(efficiency)而不降低精度(accuracy)。

3. 在with domain decomposition的算法步骤中，引入了并行计算(parallel computing)的思想，对每一个子域上求解u，训练神经网络w的时间并行。
4. 在with domain decomposition的算法步骤中，每一个子神经网络仅需要在sub-domain上采样部分点即可（Besides, the training of each network only needs data samples on part of the computing domain, which has less computational cost than training a single network with data samples from the whole domain）。
5. 在with domain decomposition和without domain decomposition问题的算法步骤中均没有出现任何penalty terms，因此更易训练，消除了penalty terms设置对于结果影响的灵敏性。

The Deep Ritz Method: A Deep Learning-Based Numerical Algorithm for Solving Variational Problems

算法步骤

$$\min_{u\in H} I(u)=\int_{\Omega}(\frac{1}{2}|\nabla(x)|^2-f(x)u(x))dx$$

1. 建立由若干个Block组成的layer，每个block里放置两个Linear transformation，activation functions以及一个残差连接（把输入和输出相结合，是一种特殊的skip-connection）（加上输出还有一层意思是哪怕再差也不可能比原有的输入更差）。
2. 建立神经网络层的block：

$$t=f_i(s)=\phi(W_{i,2}\cdot\phi(W_{i,1}s+b_{i,1})+b_{i,2})+s.$$

选取激活函数ϕ为$\phi(x)=\max\{x^3,0\}$;
$u(x;\theta)=a\cdot z_\theta(x)+b.$;

$$\min_{\theta} L(\theta)=\int_{\Omega}\frac{1}{2}|\nabla_x u(x;\theta|^2-f(x)u(x;\theta)dx$$
$$\begin{cases}u(x;\theta)=a\cdot z_\theta(x)+b,\\ z_\theta(x)=f_n\circ\cdots\circ f_1(x),\\ f_i(s)=\phi(W_{i,2}\cdot\phi(W_{i,1}s+b_{i,1})+b_{i,2})+s.\end{cases}$$

3. 运用随机梯度下降更新$L(\theta)$中的参数θ:

$$\theta^{k+1}=\theta^k-\eta\nabla_\theta\left(\frac{1}{M}\sum_{j=1}^{M}g(x_{j,k};\theta^k)\right).$$

Advantages

1. The Deep Ritz Method is naturally nonlinear, naturally adaptive and has the potential to work in rather high dimensions.
2. Fits well with the stochastic gradient descent method used in deep learning.

Disadvantages

1. 使用了数值积分的方法来离散化积分，则必须选一组固定的节点进行积分，可能会因为在该选定的节点上积分被最小化了，但是整个泛函的值并未最小化。
2. 对于一些特殊的边界条件需要引入penalty terms，加大了训练代价（处理基本边界条件没有那么简单）。
3. 最后得到的变分问题不一定是凸的；

Physics-informed neural networks

Physics-Informed Neural Network:

(1) Continuous time approach

 Consider an evolution equation:

$$\begin{gathered}\partial_t u(t,x)+\mathcal{N}[u](t,x)=0,(t,x)\in(0,T]\times\mathcal{D},\\ u(0,x)=u_0(x),x\in\mathcal{D},\\ u(t,x)=u_b(t,x),(t,x)\in(0,T]\times\partial\mathcal{D}.\end{gathered}$$

 Construct a multilayer feed-forward neural networks

$$u_\theta(z)=W^L\sigma^L(\ldots(\ldots\sigma^1(W^0z+b^0)\ldots)+b^{L-1})+b^L.$$

 Generate collocation points X^0, X^r and X^b w.r.t initial, ordinary and boundary conditions.

 Define the mean squared residual $\phi_\theta^r(X^r)$, the mean squared misfit w.r.t the initial condition $\phi_\theta^0(X^0)$ and the boundary condition $\phi_\theta^b(X^b)$ and build the loss function $\phi_\theta(X)=\phi_\theta^r(X^r)+\phi_\theta^0(X^0)+\phi_\theta^b(X^b)$.

 Train the Neural Network and minimize the loss function.

(2) Discrete time approach

 Assume the availability of N_n solution data points:

$$X^n:=\left\{(t^n,x^{n,k},u^{n,k})\right\}_{k=1}^{N_n}.$$

 Employ a Runge-Kutta method with q stages,

$$\begin{aligned}u^{n+c_i}&=u^n-\Delta t\Sigma_{j=1}^q a_{ij}\mathcal{N}[u^{n+c_j}],i=1,\ldots q,\\ u^{n+1}&=u^n-\Delta t\Sigma_{j=1}^q b_j\mathcal{N}[u^{n+c_j}],\end{aligned}$$

where $u^{n+c_j}\approx u(t^n+c_j\Delta t)$ for $j=1,\ldots,q$.

 Construct the link between the data set X^n, the PDE solution at time t^{n+1} and the unknown stages $u^{n+c_i},i=1,\ldots q$:

$$\begin{aligned}r^i(x^{n,k},u^{n,k})&:=u^{n+c_i}(x^{n,k})-u^{n,k}+\Delta t\Sigma_{j=1}^q a_{ij}\mathcal{N}\left[u^{n+c_j}(x^{n,k})\right]\approx 0,\quad i=1,\ldots,q,\\ r^{q+1}(x^{n,k},u^{n,k})&:=u^{n+1}(x^{n,k})-u^{n,k}+\Delta t\Sigma_{j=1}^q b_j\mathcal{N}\left[u^{n+c_j}(x^{n,k})\right]\approx 0.\end{aligned}$$

- Then the loss function is gained:

$$\phi(X^n) := \Sigma_{k=1}^{N_n}\Sigma_{j=1}^{q+1}|r^j(x^{n,k},u^{n,k})|^2 + +\Sigma_{i=1}^{q}(|u^{n+c_i}(-1)|^2 + |u^{n+c_i(+1)}|^2) + |u^{n+1}(-1)\}|^2.$$

- Train the network and minimize the loss function to learn the unknown mapping:

$$x \mapsto \left(u^{n+c_1}(x),\ldots,u^{n+c_q}(x),u^{n+1}(x)\right).$$

Advantages

1. Introduce three important recent approaches on high-dimensional problems: physics-informed neural networks, methods based on the Feynman–Kac formula and methods based on the solution of backward stochastic differential equations.
2. Employ the Runge-Kutta method to continue the solution to t^{n+1}.
3. The discrete time approach is easily modified to determine unknown parameters in a general nonlinear partial differential equation with unknown parameter $\partial_t u(t,x) + \mathcal{N}^\lambda[u](t,x) = 0$, $(t,x) \in (0,T] \times \mathcal{D}$.

组会补充方向

- 将物理信息Physical information作为先验信息（正则化因子Regularity Term），使用少量数据即可以进行训练，一定程度打开机器学习的black box.
- 正问题：给定方程去求解；反问题：给定答案解的一些采样，去找到方程的具体表达形式（比如一些参数）。

Title	Algorithm Framework	Contribution & Innovation	Remarks
《*Artifial Neural Networks for Solving Ordinary and Partial Differential Equations*》	1. Define the general differential equation: 2. Use the collocation method and construct a trial solution $\Psi_t(\vec{x},\vec{p})$: 3. Thus the loss function is gained: 4. Train the neural network $N(\vec{x},\vec{p})$.	1. Adopt collocation method. 2. Provide the construction formulas for neural network numerical solutions considering boundary conditions (Dirichlet and Neumann).	1. No more streamlined algorithms have been presented for higher-order differential equations. 2. No further explanation or guidance provided regarding the neural network architecture of $N(\vec{x},\vec{p})$.
《*Solving Partial Differential Equations with Neural Networks*》	1. Consider a function: 2. For all points, compute the outputs of the network $\phi(x,t)$ and the derivatives w.r.t. the inputs ϕ_t,ϕ_x,ϕ_{xx}. 3. For internal points & boundary points, build a MSE loss function. 4. Update the parameters of the NN for each loss function.	1. Provide a detailed framework for utilizing neural networks to solve partial differential equations. 2. Provide numerous numerical examples and open-source code for the learners.	1. The article is applicable to specific partial differential equations, with limited generalizability. 2. Theoretical innovation is relatively limited.

Title	Algorithm Framework	Contribution & Innovation	Remarks
*《Three ways to solve partial differential equations with neural networks—Areview**》*	Physics-Informed Neural Network: (1) Continuous time approach Consider an evolution equation: Construct a multilayer feed-forward neural networks Generate collocation points $N(\vec{x},\vec{p})$ and w.r.t initial, ordinary and boundary conditions. Define the mean squared residual $\phi(x,t), N(\vec{x},\vec{p})$, the mean squared misfit w.r.t the initial condition $\phi_t, \phi_x, \phi_{xx}$ and the boundary condition $\phi_\theta^b(X^b)$ and build the loss function . Train the Neural Network and minimize the loss function. (2) Discrete time approach Assume the availability of N_n solution data points: Employ a Runge-Kutta method with q stages, where $u^{n+c_j} \simeq u(t^n + c_j \Delta t)$ for $j = 1, \ldots, q$.. Construct the link between the data set X^n , the PDE solution at time t^{n+1} and the unknown stages $u^{n+c_i}, i = 1, \ldots, q$: Then the loss function is gained: Train the network and minimize the loss function to learn the unknown mapping:	1.Introduce three important recent approaches on high-dimensional problems: physics-informed neural networks, methods based on the Feynman–Kac formula and methods based on the solution of backward stochastic differential equations. 2.Employ the Runge-Kutta method to continue the solution to t^{n+1}.	1. No more streamlined algorithms have been presented for higher-order differential equations. 2. No further explanation or guidance provided regarding the neural network architecture of $N(\vec{x},\vec{p})$.

感兴趣的研究方向

对RFM方法做进一步的优化

- RFM方法现在已经有了TIME-DEPENDENT PROBLEMS（THE RANDOM FEATURE METHOD FOR TIME-DEPENDENT PROBLEMS）；
- RFM原始论文（Bridging Traditional and Machine Learning-based Algorithms for Solving PDEs: The Random Feature Method）；
- RFM在界面问题上的扩展（THE RANDOM FEATURE METHOD FOR SOLVING INTERFACE PROBLEMS）：值得注意的点在于该论文用的是C++中的Eigen库来进行数值计算；之后如果进行并行计算的扩展，可以继续利用数值计算语言C++实现。

尝试关注一下把RFM更多地与并行计算融合？比如提出一个并行计算的框架（类似于PFNN，PFNN的机制是先划分区域，然后每个区域采点，构建$w_i = g_i + l_i f_i$，使用l_i作为距离函数代替惩罚项，然后训练g_i和f_i，每一个区域都由两个子网络构成来逼近相关本地解）。

1.看看在Ω区域能否划分为$\{\tilde{\Omega}_i\}, i = 1, \cdots, k$，对每一个区域$\tilde{\Omega}_i$尝试使用并行算法来分解

2.比如在现有的4阶finite-difference method等方法中的矩阵系统，对矩阵进行一些分解工作，让它更容易并行计算。

3.用C++的不同grid,block,thread单位来进行并行计算，一个thread当作一个小的矩阵元素，一个block当作一个矩阵的行，一个grid当作一个矩阵。

通用矩阵乘法（GEMM）：

```
int row = blockIdx.y * blockDim.y + threadIdx.y;
int col = blockIdx.x * blockDim.x + threadIdx.x;
```

单精度矩阵乘法（SGEMM）：

```
const uint x = blockIdx.x * blockDim.x + threadIdx.x;
const uint y = blockIdx.y * blockDim.y + threadIdx.y;
```

the case of a single sub-domain:

$$\begin{cases} \frac{\partial u}{\partial t} - \nabla \cdot (A \nabla u - B) + C = 0, & \text{in } \Omega \times (0, T], \\ u = r_D, & \text{on } \Gamma_D \times (0, T], \\ (A \nabla u) \cdot n = r_N, & \text{on } \Gamma_N \times (0, T], \\ u = r_0, & \text{in } \Omega \times \{0\}, \end{cases}$$

Dirichlet Boundary: $u = r_D, \text{on } \Gamma_D \times (0, T], u = r_0, \text{in } \Omega \times \{0\}$;

Neumann Boundary: $(A \nabla u) \cdot n = r_N, \text{on } \Gamma_N \times (0, T]$。

- Step1： 构造$w_\theta(x) = g_{\theta_1}(x) + l(x) f_{\theta_2}(x)$;
- Step2：使用spline functions构造length factor函数$l(x)$;

$$l(x, \mathrm{t}) = \frac{\tilde{l}(x, \mathrm{t})}{\max\limits_{(\hat{x}, \hat{\mathrm{t}}) \in \Omega \times (0, \mathrm{T}]} \tilde{l}(\hat{x}, \hat{\mathrm{t}})}, \quad \tilde{l}(x, t) = \frac{t}{T} \prod_{j=1} 1 - (1 - l_k(x))^u,$$

- Step3：构造compactly supported test functions作为公式中的v。

$$v_s(x, t; \hat{x}_s, \hat{t}_s, h_s) = \max\left(1 - \frac{|t - \hat{t}_s|}{h_s}, 0\right) \prod_{j=1}^{d_x} \max\left(1 - \frac{|x_j - \hat{x}_{s,j}|}{h_s}, 0\right),$$
$$s = 1, 2, \ldots, n_v, \quad n_v \in \mathbb{N}^+$$

- Step4：构造$g_{\theta_1}(x)$的训练函数：

$$\Psi[g_{\theta_1}] := \sum_{(x,t) \in S(\Gamma_D \times (0,T])} (g_{\theta_1}(x, t) - r_D(x, t))^2 + \sum_{(x,t) \in S(\bar{\Omega}_i) \times \{0\}} g_{\theta_1}(x, t) - r_0(x, t))^2$$

- Step5：构造$f_{\theta_2}(x)$的训练函数

$$u^* = \arg\min_{\tilde{w}_{\theta_2}} L[\tilde{w}_{\theta_2}] = \arg\min_{\tilde{w}_{\theta_2}} \sum_{s=1}^{n_v} R^2[\tilde{w}_{\theta_2}; v_s],$$

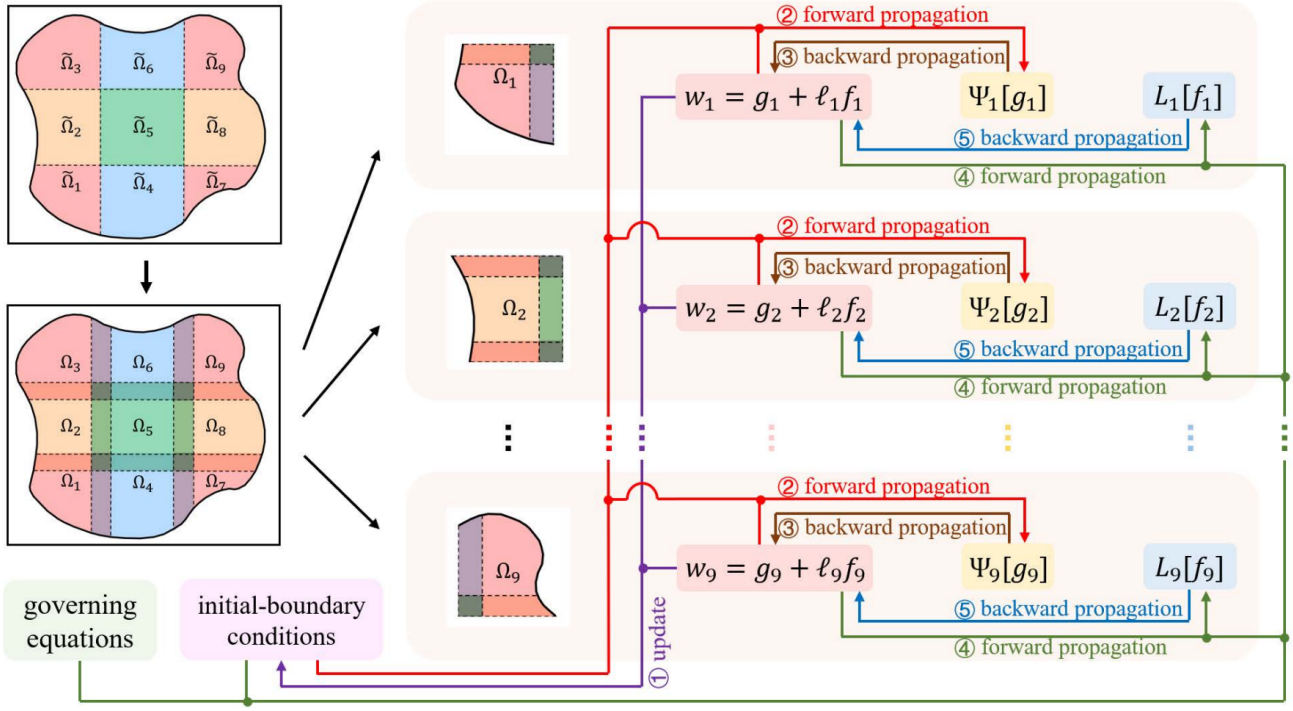
$$\begin{aligned} R[\tilde{w}_{\theta_2}; v_s] \quad := & \frac{|\Omega| T}{\#S(\Omega \times (0, T])} \sum_{(x,t) \in S(\Omega \times [0,T])} \left[\mathcal{A} \nabla \tilde{w}_{\theta_2} \cdot \nabla v_s + \left(\frac{\partial \tilde{w}_{\theta_2}}{\partial t} + \nabla \cdot \mathcal{B} + \mathcal{C}\right) v_s\right](x, t) \\ & - \frac{|\Gamma_N| T}{\#S(\Gamma_N \times (0, T])} \sum_{(x,t) \in S(\Gamma_N \times [0,T])} [r_N v_s](x, t) \end{aligned} \tag{2.11}$$

算法步骤-with domain decomposition

$$
\begin{cases}
\dfrac{\partial u_i^k}{\partial t} - \nabla \cdot \left(\mathcal{A}\nabla u_i^k - \mathcal{B}\right) + \mathcal{C} &= 0, \quad \text{in } \Omega_i \times (0,T], \\
u_i^k &= u^{k-1}, \quad \text{on } \Gamma_i \times (0,T], \\
u_i^k &= r_D, \quad \text{on } \Gamma_{i,D} \times (0,T], \\
\left(\mathcal{A}\nabla u_i^k\right) \cdot \boldsymbol{n} &= r_N, \quad \text{on } \Gamma_{i,N} \times (0,T], \\
u_i^k &= r_0, \quad \text{on } \overline{\Omega}_i \times \{0\}
\end{cases} \tag{3.1}
$$

- 将需要计算的区域切分为一组不互相重叠的子域$\{\tilde{\Omega}_i\}_{i=1}^m$；
- 再将每个子域$\{\tilde{\Omega}_i\}_{i=1}^m$拓展至一系列相互重叠的子域$\{\Omega_i\}_{i=1}^m$，$m$代表的是数量。
- 根据切分后的结果重新定义

  $\Gamma_{i,D} = \partial\Omega_i \cap \Gamma_D, \quad \Gamma_{i,N} = \partial\Omega_i \cap \Gamma_N, \quad \text{and} \quad \Gamma_i = \partial\Omega_i \setminus (\Gamma_D \cup \Gamma_N).$
- 假设在第k轮的迭代时子域Ω_i上的神经网络u为u_i^k，after solving the subdomain problems for iteration k, the approximate solution defined on the whole domain is composed as $u^k|_{\tilde{\Omega}_i} = u_i^k$.



- 和之前没有进行domain decomposition时一样定义原问题的弱形式（weak form），对u进行多次训练、子域划分因此需要给出u的新定义。
- 给出$w_i = g_i + l_i f_i$中g_i和f_i的训练函数(Loss Function)，进行训练即可。

Algorithm 1: Computing the i-th local approximate solution in PFNN-2.

Input: i, m, g_i^0 f_i^0, ℓ_i, K_o (number of outer iteration), K_i (number of inner iteration).

Output: $w_i^{K_o}$.

1 Let $w_i^0 = g_i^0 + \ell_i f_i^0$.
2 **for** $k = 1,2,\cdots,K_o$ **do**
3 /* Online stage: line 4 to 13 */
4 **for** $j = 1,2,\cdots,m$ **do**
5 **if** $i < j$ **then**
6 Send $w^{k-1}|_{\Gamma_j \cap \Omega_i} = w_i^{k-1}$ to processor j.
7 Receive $w^{k-1}|_{\Gamma_i \cap \Omega_j} = w_j^{k-1}$ from processor j.
8 **end**
9 **if** $i > j$ **then**
10 Receive $w^{k-1}|_{\Gamma_i \cap \Omega_j} = w_j^{k-1}$ from processor j.
11 Send $w^{k-1}|_{\Gamma_j \cap \Omega_i} = w_i^{k-1}$ to processor j.
12 **end**
13 **end**
14 /* Offline stage: line 15 to 23 */
15 **for** $t = 1,2,\cdots,K_i$ **do**
16 Forward propagation: evaluate the loss function $\Psi_i^k[g_i]$ according to (3.4).
17 Backward propagation: calculate the gradients of $\Psi_i^k[g_i]$ and minimize $\Psi_i^k[g_i]$ to get the updated g_i^k.
18 **end**
19 **for** $t = 1,2,\cdots,K_i$ **do**
20 Forward propagation: evaluate the loss function $L_i^k[f_i]$ according to (3.5).
21 Backward propagation: calculate the gradients of $L_i^k[f_i]$ and minimize $L_i^k[f_i]$ to get the updated f_i^k.
22 **end**
23 Let $w_i^k = g_i^k + \ell_i f_i^k$.
24 **end**