An Intuitive Guide to KFAC for Physics-Informed Neural Networks

Understanding the Mechanics for 2D Poisson's Equation with JAX Concepts

May 17, 2025

Abstract

Physics-Informed Neural Networks (PINNs) have shown remarkable promise in solving differential equations, but their training, especially with first-order optimizers, can be challenging and slow. Second-order methods, like those based on the Gauss-Newton (GGN) matrix, can drastically improve accuracy and convergence. However, the $\mathcal{O}(D^3)$ cost of forming and inverting full GGN matrices for networks with D parameters restricts their use. *Kronecker-Factored Approximate Curvature* (KFAC) offers a scalable alternative by approximating layer-wise blocks of the GGN matrix as $B \otimes A$, allowing efficient inversion. This document aims to provide an intuitive yet detailed understanding of how KFAC, particularly the formulation by Dangel et al. (2024)[1], is adapted for PINNs. We use the 2D Poisson equation as a running example and discuss conceptual JAX implementations, drawing from recent literature to build a comprehensive picture of this advanced optimization technique.

1 The Challenge: Optimizing Physics-Informed Neural Networks

PINNs embed physical laws, typically expressed as partial differential equations (PDEs), into the neural network's loss function. For a function $u(\mathbf{x})$ governed by $\mathcal{L}_{\text{op}}(u(\mathbf{x})) = f(\mathbf{x})$ in domain Ω with boundary conditions $\mathcal{B}_{\text{op}}(u(\mathbf{x})) = g(\mathbf{x})$ on $\partial\Omega$, a neural network $u_\theta(\mathbf{x})$ is trained by minimizing:

$$\mathcal{L}(\theta) = \mathcal{L}_\Omega(\theta) + \mathcal{L}_{\partial\Omega}(\theta) \tag{1}$$

where $\mathcal{L}_\Omega(\theta)$ penalizes PDE residuals and $\mathcal{L}_{\partial\Omega}(\theta)$ penalizes boundary condition mismatches. Training PINNs effectively is non-trivial due to potentially stiff or ill-conditioned loss landscapes [16, 21], challenges in balancing disparate loss terms, and the computational cost of evaluating differential operators and their gradients with respect to network parameters.

Our Pedagogical PDE: 2D Poisson's Equation We focus on:

$$\begin{aligned} -\Delta u(x,y) &= 2\pi^2 \sin(\pi x)\sin(\pi y) && \text{for } \mathbf{x} = (x,y) \in \Omega := [0,1]^2 \\ u(x,y) &= 0 && \text{for } \mathbf{x} = (x,y) \in \partial\Omega \end{aligned}$$

The analytic solution is $u^\star(x,y) = \sin(\pi x)\sin(\pi y)$. The PINN loss (matching Eq. (1) in Dangel et al.[1]) is:

$$\mathcal{L}(\theta) = \underbrace{\frac{1}{2N_\Omega}\sum_{n=1}^{N_\Omega}\big(-\Delta u_\theta(\mathbf{x}_n) - f(\mathbf{x}_n)\big)^2}_{\mathcal{L}_\Omega(\theta)} + \underbrace{\frac{1}{2N_{\partial\Omega}}\sum_{n=1}^{N_{\partial\Omega}}\big(u_\theta(\mathbf{x}_n^{\text{bdy}}) - 0\big)^2}_{\mathcal{L}_{\partial\Omega}(\theta)}. \tag{2}$$

2 The Ideal: Second-Order Optimization with Gauss-Newton

Second-order methods like Energy Natural Gradient Descent (ENGD) (Müller & Zeinhofer, 2023)[3] often employ a Gauss-Newton (GGN) approximation to the Hessian. The update rule is $\theta_{k+1} = \theta_k - \eta G(\theta_k)^{-1}\nabla\mathcal{L}(\theta_k)$. For the PINN loss (**??**), the GGN matrix $G(\theta)$ is approximately (see Eq. (2) in Dangel et al.[1]):

$$G(\theta) \approx G_\Omega(\theta) + G_{\partial\Omega}(\theta)$$

$$G_\Omega(\theta) = \frac{1}{N_\Omega}\sum_{n=1}^{N_\Omega}\left(J_\theta(-\Delta u_\theta(\mathbf{x}_n))\right)^\top\left(J_\theta(-\Delta u_\theta(\mathbf{x}_n))\right) \tag{3}$$

$$G_{\partial\Omega}(\theta) = \frac{1}{N_{\partial\Omega}}\sum_{n=1}^{N_{\partial\Omega}}\left(J_\theta u_\theta(\mathbf{x}_n^{\text{bdy}})\right)^\top\left(J_\theta u_\theta(\mathbf{x}_n^{\text{bdy}})\right) \tag{4}$$

where $J_\theta(\cdot)$ is the Jacobian of the respective network output with respect to all network parameters θ. The $\mathcal{O}(D^3)$ cost of forming and inverting $G(\theta)$ makes full GGN impractical.

3 The Scalable Solution: Kronecker-Factored Approximate Curvature (KFAC)

KFAC (Martens & Grosse, 2015)[2] approximates $G(\theta)$ with a block-diagonal matrix (one block per layer l), where each $G^{(l)}(\theta) \approx B^{(l)} \otimes A^{(l)}$. This allows efficient inversion: $(B^{(l)} \otimes A^{(l)})^{-1} = (B^{(l)})^{-1} \otimes (A^{(l)})^{-1}$.

3.1 The Core Idea for PINNs: KFAC on an Effective Weight-Shared Network via Taylor-Mode AD

The primary challenge for applying KFAC to PINNs is that $\mathcal{L}_\Omega(\theta)$ involves differential operators. Dangel et al.[1] address this by leveraging **Taylor-mode Automatic Differentiation (AD)** as a conceptual and mathematical framework [5, 19]. Taylor-mode AD propagates Taylor series coefficients of a function (which encode its derivatives) through its computation graph. For PINNs, this means the computation of u_θ *and* its derivatives (e.g., for $-\Delta u_\theta$) can be described as a single, unified forward pass through an "augmented network."

- **Augmented State Propagation:** Instead of just propagating scalar activations $z^{(l-1)}$, this augmented network propagates an "augmented state vector." For the 2D Poisson problem, this state at the input of layer l is $Z_{in}^{(l-1)} = (z^{(l-1)}, \partial_x z^{(l-1)}, \partial_y z^{(l-1)}, \Delta z^{(l-1)})$.

Each component $z^{(l-1)}, \partial_x z^{(l-1)}$, etc., is itself a vector of the same dimension as the standard layer activations.

- **Shared Weights and KFAC Application:** The original network weights $W^{(l)}$ of layer l are used to transform *all components* of this augmented state (e.g., $\partial_x z^{(l)} = W^{(l)}(\partial_x z^{(l-1)})$, while $z^{(l)} = W^{(l)} z^{(l-1)} + b^{(l)}$). This constitutes "weight sharing" across the components of the augmented state. KFAC can then be applied to this effective weight-shared network, drawing on general formulations for KFAC with weight sharing (Eschenhagen et al. [6]). The KFAC factors $A_\Omega^{(l)}$ and $B_\Omega^{(l)}$ effectively average statistics over these "virtual" parallel computations performed by the shared weights on the different components of the augmented state.

For specific (often linear) PDE operators like the Laplacian, this Taylor-mode AD concept can be implemented very efficiently using the **Forward Laplacian** framework (Li et al. [4], generalized by Dangel et al.[1]). This framework provides direct propagation rules for $(u, \nabla u, \Delta u)$ through network layers, avoiding the full overhead of general Taylor-mode AD tools (like ‘jax.experimental.jet‘) and costly repeated full Hessian computations. JAX implementations of Forward Laplacian (e.g., ‘microsoft/folx‘ [7], ‘y1xiaoc/fwdlap‘ [8]) often achieve this efficiency via custom JAX primitive rules or by interpreting ‘jaxpr‘ representations.

KFAC Factors for the Boundary Term $\mathcal{L}_{\partial\Omega}$ (Standard Regression): For a layer l with input activations $z^{(l-1)}$ and output pre-activations $s^{(l)} = W^{(l)} z^{(l-1)} + b^{(l)}$:

$$A_{\partial\Omega}^{(l)} = \mathbb{E}_{\mathbf{x}^{\text{bdy}}}\left[z^{(l-1)}(z^{(l-1)})^\top\right] \quad \text{(Covariance of layer inputs)}$$

$$B_{\partial\Omega}^{(l)} = \mathbb{E}_{\mathbf{x}^{\text{bdy}}}\left[(\nabla_{s^{(l)}}\mathcal{L}_{\partial\Omega})\,(\nabla_{s^{(l)}}\mathcal{L}_{\partial\Omega})^\top\right] \quad \text{(Covariance of loss gradients w.r.t. pre-activations)}$$

KFAC Factors for the PDE Operator Term $\mathcal{L}_\Omega$ (Augmented Network): Using the augmented state components, the factors are (see Eq. (8) & (9) in Dangel et al.[1], which average over S components):

$$A_\Omega^{(l)} = \mathbb{E}_{\mathbf{x}}\left[\frac{1}{S}\sum_{s=1}^{S}(Z_{in}^{(l-1)})_s((Z_{in}^{(l-1)})_s)^\top\right]$$
$$B_\Omega^{(l)} = \mathbb{E}_{\mathbf{x}}\left[\frac{1}{S}\sum_{s=1}^{S}(\nabla_{(S_{out}^{(l)})_s}\mathcal{L}_\Omega)((\nabla_{(S_{out}^{(l)})_s}\mathcal{L}_\Omega))^\top\right]$$

Here, $S = d + 2 = 4$ for 2D Poisson. $(Z_{in}^{(l-1)})_s$ is the s-th component of the augmented input activation vector to layer l. $(S_{out}^{(l)})_s$ is the s-th component of the augmented *pre-activation* output vector of layer l. $A_\Omega^{(l)}$ captures statistics of these augmented inputs. $B_\Omega^{(l)}$ captures statistics of how the PDE loss $\mathcal{L}_\Omega$ (which depends on the final $-\Delta u_\theta$) changes with respect to these intermediate augmented pre-activations.

Combining and Inverting Factors: The total KFAC preconditioner for layer l combines these. Dangel et al.[1] (Eq. before §3.4 Gradient Preconditioning) sum the damped individual preconditioners:

$$\tilde{G}^{(l)-1} \approx ((\bar{A}_\Omega^{(l)} + \lambda_\Omega I)^{-1} \otimes (\bar{B}_\Omega^{(l)} + \lambda_\Omega I)^{-1}) + ((\bar{A}_{\partial\Omega}^{(l)} + \lambda_{\partial\Omega} I)^{-1} \otimes (\bar{B}_{\partial\Omega}^{(l)} + \lambda_{\partial\Omega} I)^{-1})$$

where $\bar{A}, \bar{B}$ are EMA-updated factors. The action of this inverse sum of Kronecker products on a gradient vector is computed efficiently using eigendecompositions of all four small factor matrices (see Appendix I of Dangel et al.[1]).

4 Algorithm Outline (§3.4)

At each KFAC optimization step t:

1. **Factor Statistics Collection Phase** (Done every $factor_update_freq$ steps, on a dedicated batch if desired):
   (a) **Augmented & Standard Forward Pass:**
      - For $\mathcal{L}_\Omega$: On interior points $\{\mathbf{x}_n\}$, perform the "augmented forward pass" (e.g., Forward Laplacian). For each layer l, collect augmented input activations $(Z_{in}^{(l-1)})_{n,s}$ (for $A_\Omega^{(l)}$) and augmented pre-activation outputs $(S_{out}^{(l)})_{n,s}$. Store the final operator output, e.g., $-\Delta u_\theta(\mathbf{x}_n)$.
      - For $\mathcal{L}_{\partial\Omega}$: On boundary points $\{\mathbf{x}_n^{\text{bdy}}\}$, perform a standard forward pass. For each layer l, collect input activations $z_n^{(l-1)}$ (for $A_{\partial\Omega}^{(l)}$) and pre-activation outputs $s_n^{(l)}$. Store the final network output $u_\theta(\mathbf{x}_n^{\text{bdy}})$.

   (b) **Backward Pass for B-Factor Gradients:**
      - Compute gradients $\nabla_{(S_{out}^{(l)})_{n,s}} \mathcal{L}_\Omega$ for $B_\Omega^{(l)}$. This involves taking the gradient of the sampled $\mathcal{L}_\Omega$ (formed using the stored $-\Delta u_\theta(\mathbf{x}_n)$) with respect to the collected $(S_{out}^{(l)})_{n,s}$ via VJP.
      - Compute gradients $\nabla_{s_n^{(l)}} \mathcal{L}_{\partial\Omega}$ for $B_{\partial\Omega}^{(l)}$ similarly, from sampled $\mathcal{L}_{\partial\Omega}$ and stored $u_\theta(\mathbf{x}_n^{\text{bdy}})$.

   (c) **Update Kronecker Factor EMAs:** For each layer l, compute current mini-batch estimates $\hat{A}_\Omega^{(l)}, \hat{B}_\Omega^{(l)}, \hat{A}_{\partial\Omega}^{(l)}, \hat{B}_{\partial\Omega}^{(l)}$. Update EMA-smoothed factors: $\bar{A}_{\Omega,t}^{(l)} \leftarrow \rho \bar{A}_{\Omega,t-1}^{(l)} + (1-\rho)\hat{A}_\Omega^{(l)}$, etc.

2. **Gradient Computation and Preconditioning Phase** (Done every $grad_update_freq$ steps, typically more frequent):
   (a) **Compute Full Gradient:** On a (potentially different/smaller) mini-batch, compute the standard gradient of the total loss: $\nabla_{W^{(l)}} \mathcal{L}(\theta_t)$ for all layers l.

   (b) **Damp Factors and Precondition Gradient:** Apply damping to the current EMA factors: $\tilde{A}_\Omega^{(l)} = \bar{A}_{\Omega,t}^{(l)} + \lambda_\Omega I$, etc. Compute the preconditioned gradient $\Delta W^{(l)}$ by applying the inverse of the combined KFAC preconditioner to $\nabla_{W^{(l)}} \mathcal{L}(\theta_t)$.

3. **Apply Parameter Update**: Use ΔW with an outer learning rate η_t (and potentially momentum, or KFAC* heuristics) to update: $\theta_{t+1} = \theta_t - \eta_t \Delta W$.

5 Conceptual JAX + Equinox Implementation Notes

Listing 1: Further conceptual details for KFAC in JAX + Equinox.

```
import jax, jax.numpy as jnp, equinox as eqx, optax
from typing import Tuple, List, Any, Callable, Dict

```

```
# --- Network Definition (MLP from previous draft) ---
class MLP(eqx.Module): ... # (As before)

# --- Augmented State & Factor Management ---
AugmentedState = Tuple[jnp.ndarray, jnp.ndarray, jnp.ndarray, jnp
   ↪ .ndarray] # val, dx, dy, lap
ActivationDerivs = Tuple[Callable, Callable, Callable] # sigma,
   ↪ sigma', sigma''

@eqx.filter_pytree_node
class KFACFactorsLayer(eqx.Module):
    A_omega: jnp.ndarray; B_omega: jnp.ndarray
    A_boundary: jnp.ndarray; B_boundary: jnp.ndarray
    # Actual shapes depend on layer_input_dim, layer_output_dim

    def __init__(self, layer_input_dim: int, layer_output_dim:
       ↪ int, init_val: float = 1e-2):
        # Initialize factors as scaled identities for stability
        self.A_omega = jnp.eye(layer_input_dim) * init_val
        self.B_omega = jnp.eye(layer_output_dim) * init_val
        self.A_boundary = jnp.eye(layer_input_dim) * init_val
        self.B_boundary = jnp.eye(layer_output_dim) * init_val

class KFACState(eqx.Module):
    layer_factors_ema: List[KFACFactorsLayer] # PyTree of
       ↪ KFACFactorsLayer
    ema_rho: float; damping_omega: float; damping_boundary: float
    factor_update_freq: int; grad_update_freq: int
    factor_update_counter: jnp.ndarray; grad_step_counter: jnp.
       ↪ ndarray

def get_activation_derivatives(activation_fn_name: str) ->
   ↪ ActivationDerivs: ... # (As before)

# --- Augmented Propagation for one layer ---
# (propagate_augmented_one_linear_activation_layer from previous
   ↪ draft)
# Input aug_state_in: (val_in, dx_val_in, dy_val_in, lap_val_in)
   ↪ where val_in is activation from prev layer.
# Output: ( (val_out, dx_val_out, dy_val_out, lap_val_out), #
   ↪ After activation
#           (s_val, s_dx, s_dy, s_lap),                   # Pre-
   ↪ activation, for B-factors
#           aug_state_in                                  # Input
   ↪  activations, for A-factors
#         )

# --- Initial Augmented State at Network Input ---
def initial_augmented_state(point_coords: jnp.ndarray,
   ↪ input_dim_first_layer: int) -> AugmentedState:
    # point_coords is the (x,y) input, e.g., shape (2,)
    # Dangel et al. Eq 5a-c: z(0)=x, dx_i z(0)=e_i, d2_ij z(0)=0
    # This means the "value" component of the augmented state at
       ↪ input is point_coords itself.
    # The "derivative" components are the derivatives of this
       ↪ input vector w.r.t. spatial coords.

    val_z0 = point_coords # This is what the first layer sees as
       ↪ its "value" input

    # Assuming point_coords = [x_coord, y_coord] and
       ↪ input_dim_first_layer = 2
    if input_dim_first_layer != 2 or point_coords.shape[0] != 2:
        raise ValueError("Mismatch in initial_augmented_state
           ↪ dimensions for 2D Poisson.")

    dx_z0 = jnp.array([1.0, 0.0]) # d(val_z0)/dx_coord = d[x,y]/
       ↪ dx = [1,0]
    dy_z0 = jnp.array([0.0, 1.0]) # d(val_z0)/dy_coord = d[x,y]/
       ↪ dy = [0,1]
    lap_z0 = jnp.array([0.0, 0.0])# Laplacian of (x,y) vector is
       ↪ (0,0)
    return (val_z0, dx_z0, dy_z0, lap_z0)

# --- Full Forward Pass for Factor Statistics (Conceptual,
   ↪ vmappable over batch) ---
def collect_factor_statistics_pass_vmappable(model: MLP,
   ↪ single_spatial_point: jnp.ndarray):
    # single_spatial_point: e.g., [x_coord, y_coord]
    # This function traces one sample through the net, collecting
       ↪  all necessary quantities.

    # Initialize for augmented pass (Omega term)
    # Assuming first layer of MLP takes input of same dim as
       ↪ single_spatial_point
    first_layer_input_dim = model.layers[0].weight.shape[1]
    current_aug_state = initial_augmented_state(
       ↪ single_spatial_point, first_layer_input_dim)

    # Store layer-wise stats: PyTrees matching model structure
    # stats_A_omega_terms[l] = list of (Z_in^(l-1))_s components
       ↪ for layer l
    # stats_B_omega_targets[l] = list of (S_out^(l))_s components
       ↪  for layer l
    # ... and similar for boundary terms
    # For simplicity, let's assume they are dicts keyed by layer
       ↪ index or layer object.

    # This function would iterate through model.layers, call
    # propagate_augmented_one_linear_activation_layer, and store:
    # 1. For A_omega: The `aug_state_in` to each layer.
    # 2. For B_omega_targets: The ‘aug_linear_pre_act‘ from each
       ↪ layer.
    # 3. For A_boundary: The standard ‘current_std_activation‘
       ↪ input to each layer.
    # 4. For B_boundary_targets: The standard ‘pre_act_std‘ from
       ↪ each layer.
    # It also needs to return the final_neg_lap_u and final_u_val
       ↪  for this sample.
    pass # Placeholder for detailed layer-wise iteration and stat
       ↪  collection

# --- Loss and Gradient for KFAC Factors ---
f_rhs = lambda x_vec: 2 * jnp.pi**2 * jnp.sin(jnp.pi * x_vec[0])
   ↪ * jnp.sin(jnp.pi * x_vec[1])

# This function computes the loss based on intermediate pre-
   ↪ activations.
# It's differentiated w.r.t these pre-activations to get B-factor
   ↪  gradients.
# def loss_from_preactivations_fn(
#       model_outputs_from_preacts: Dict, # Contains
   ↪ final_neg_lap_u, final_u_val
#       targets: Dict # Contains f_rhs_targets, boundary_targets
   ↪ (0 for Poisson)
#   ):
#   loss_omega = 0.5 * jnp.mean((model_outputs_from_preacts['
   ↪ neg_lap_u'] - targets['f_rhs'])**2)
#   loss_boundary = 0.5 * jnp.mean((model_outputs_from_preacts['
   ↪ u_val_bdy'] - targets['g_bdy'])**2)
#   return loss_omega + loss_boundary
# Then, jax.grad(loss_from_preactivations_fn, argnums=preact_args
   ↪ ) is used.

# --- KFAC Optimizer (Conceptual Class Structure) ---
class KFACOptimizer:
    def __init__(self, model_example: MLP, kfac_hyperparams: Dict
       ↪ ):
        # Initialize KFACState (EMA factors, counters) based on
           ↪ model_example structure
        # Initialize outer optimizer (e.g., Adam)
        pass

    # @eqx.filter_jit # Jit the step function
    def step(self, model: MLP, kfac_state: KFACState,
       ↪ outer_opt_state: Any,
             interior_batch: jnp.ndarray, boundary_batch: jnp.
                ↪ ndarray
             ) -> Tuple[MLP, KFACState, Any, float]:

        # --- 1. Compute Full Gradient of L(theta) ---
        # (loss_val, full_grads) = eqx.filter_value_and_grad(
          ↪ loss_fn_for_pinn_total)(model,...)

        # --- 2. Update KFAC Factors (if factor_update_freq
          ↪ condition met) ---
        #   a. Vmap collect_factor_statistics_pass_vmappable over
          ↪  batches
        #      to get Z_in_omega, S_out_omega, z_in_bdy,
          ↪ s_out_bdy for all layers, all samples.
        #   b. Define a function that takes these S_out/s_out and
          ↪  computes the loss.
        #      Differentiate this function w.r.t S_out/s_out to
          ↪ get B-factor gradients.
        #      (This requires careful handling of closures and
          ↪ jax.grad argnums).
        #   c. Compute A_hat, B_hat estimates (average over batch
          ↪  and 's' components for Omega).
        #   d. Update kfac_state.layer_factors_ema using EMA.
        #   e. Increment kfac_state.factor_update_counter.

        # --- 3. Precondition Gradient and Update Model (if
          ↪ grad_update_freq condition met) ---
        #   a. Damp EMA factors from kfac_state.
        #   b. preconditioned_grads = apply_kfac_preconditioner(
          ↪ full_grads, damped_ema_factors).
        #      This involves the inverse of sum of Kronecker
          ↪ products per layer.
        #   c. updates, new_outer_opt_state = outer_optimizer.
          ↪ update(preconditioned_grads, ...)
        #   d. new_model = eqx.apply_updates(model, updates)
        #   e. Increment kfac_state.grad_step_counter.
        # else: new_model = model; new_outer_opt_state =
          ↪ outer_opt_state (no model update)

        # return new_model, new_kfac_state, new_outer_opt_state,
          ↪ loss_val
        pass # Placeholder
```

6 Hyper-parameter Cheatsheet & Practical Tips

(Table from previous draft)

Practical Tips Refined:

- **Network Initialization**: Careful weight initialization (e.g., SIREN-style [22] or appropriate variance for tanh) is crucial for PINNs, affecting gradient flow and KFAC's performance.
- **Factor Initialization**: Scaled identities (αI, $\alpha \approx 0.01 - 0.1$) or zero factors with strong initial damping.
- **Adaptive Damping**: Essential. Layer-wise adaptive damping (e.g., Levenberg-Marquardt,

scaling with trust ratio $\|\Delta\theta\|/\|\nabla\mathcal{L}\|$) or separate $\lambda_\Omega, \lambda_{\partial\Omega}$. KFAC* in Dangel et al.[1] uses heuristics.

- **Numerical Stability**: $jax.default_matmul_precision = "high"$. Ensure factor positive definiteness (add jitter if using low precision). Mixed precision (Lin, Dangel, et al. [10]) can save memory.
- **Gradient Scaling/Clipping/Normalization**: Normalize loss terms or clip raw gradients, especially if loss components have disparate magnitudes or during early training.
- **Factor Update Scheduling & Batching**: Update factors less frequently. Larger, distinct batches for factor estimation can be beneficial.
- **Warm-up**: Initial epochs with Adam before KFAC can stabilize early PINN training.
- **Spectral Bias**: PINNs often struggle to learn high-frequency components. While KFAC improves curvature estimation, architectural choices (e.g., Fourier features [23], SIRENs [22]) are also important and may interact with KFAC.

7 Going Beyond Poisson: Challenges and Considerations

- **Time-dependent problems** ($u_t + \mathcal{L}_{\text{op}} u = f$): Time t is an input. Augmented state includes $\partial_t z^{(l-1)}$.
- **Higher-order operators** (e.g., biharmonic $\Delta^2 u$): Augmented state includes all derivatives up to PDE order. Forward propagation rules become more complex.
- **Nonlinear PDEs** ($\mathcal{L}_{\text{op}}(u, \nabla u, \dots) = f$): As per Dangel et al.[1] (Eq. 10, 12, 14), the GGN involves Jacobians of the PDE residual $\Psi(u, Du, \dots)$ w.r.t. $u, Du, \dots$. The augmented state and propagation rules become more involved. KFAC applies to the GGN of this nonlinear least-squares problem.
- **Convection-Dominated Problems**: These are notoriously hard for PINNs. KFAC might help by better capturing sharp gradients, but may require very careful damping and stabilization.
- **Stochastic PDEs**: KFAC's expectation-based factors might naturally extend to certain types of SPDEs where expectations are already part of the loss or solution process, but this is an advanced topic.
- **Systems of PDEs**: For vector u_θ, augmented state and KFAC factors become block matrices.

8 Limitations and Future Directions

- **Implementation Complexity**: General, robust KFAC for diverse PINNs is a major engineering task, especially the efficient, differentiable augmented forward/backward passes for arbitrary operators.
- **Hyperparameter Sensitivity**: KFAC introduces critical hyperparameters. Advanced adaptive strategies are key [16, 28, 14].
- **Memory for Factors**: While better than full GGN, factor storage can be an issue. KFLR [10, 20] or structured inverse-free methods [10] offer solutions. For very deep networks, accumulation of approximation errors in factors can also be a concern.
- **Scalability to Complex Operators/Geometries**: Efficiency of "Forward Laplacian" is operator-specific. Arbitrary operators or complex geometries needing many

points can strain augmented pass/factor computation.
- **Theoretical Understanding for PINNs**: KFAC's convergence for non-convex, ill-conditioned [11, 16, 21] PINN losses, especially with the augmented network view, needs more research.
- **Interaction with other PINN Improvements**: Synergies with curriculum learning, adaptive sampling/re-weighting [1, 8, 15, 24], domain decomposition, specialized architectures [22, 23] are open research areas. KFAC with domain decomposition could be powerful for large-scale problems.
- **Curse of Dimensionality**: While KFAC improves optimization, PINNs themselves can still suffer from the curse of dimensionality for very high-dimensional PDEs, impacting sampling requirements and generalizability.
- **Broader Scientific Machine Learning**: Efficient and scalable second-order methods like KFAC are crucial for advancing SciML, enabling more complex models and faster solutions for digital twins, inverse problems, and scientific discovery [25].

Bibliography

References

[1] Dangel, F., Müller, J., & Zeinhofer, M. (2024). Kronecker-Factored Approximate Curvature for Physics-Informed Neural Networks. *Advances in Neural Information Processing Systems (NeurIPS)*, 37.

[2] Martens, J., & Grosse, R. (2015). Optimising Neural Networks with Kronecker-Factored Approximate Curvature. *Proceedings of the 32nd International Conference on Machine Learning (ICML)*, 37, 1140-1148.

[3] Müller, J., & Zeinhofer, M. (2023). Achieving High Accuracy with PINNs via Energy Natural Gradient Descent. *Proceedings of the 40th International Conference on Machine Learning (ICML)*, 202, 25471-25485.

[4] Li, R., Ye, H., Jiang, D., Wen, X., Wang, C., Li, Z., ... Ren, W. (2023). Forward Laplacian: A New Computational Framework for Neural Network-based Variational Monte Carlo. *arXiv preprint arXiv:2301.06504*.

[5] Bettencourt, J., Johnson, M. J., Duvenaud, D. (2019). Taylor-mode automatic differentiation for higher-order derivatives in JAX. *NeurIPS Workshop on Program Transformations for ML.*

[6] Eschenhagen, R., Immer, A., Turner, R. E., Schneider, F., Hennig, P. (2023). Kronecker-Factored Approximate Curvature for Modern Neural Network Architectures. *Advances in Neural Information Processing Systems (NeurIPS)*, 36.

[7] Microsoft. (Accessed 2024). folx: Implementation of Forward Laplacian algorithm in JAX. *GitHub Repository.* `https://github.com/microsoft/folx`.

[8] Xiao, Y. (Accessed 2024). fwdlap: Forward mode laplacian implemented in JAX tracer. *GitHub Repository.* `https://github.com/y1xiaoc/fwdlap`.

[9] Google Research. (Accessed 2024). KFAC-JAX: Kronecker-factored Approximate Curvature in JAX. *GitHub Repository.* `https://github.com/google-research/kfac-jax`.

[10] Lin, W., Dangel, F., Eschenhagen, R., Neklyudov, K., Kristiadi, A., Turner, R. E., Makhzani, A. (2024). Structured Inverse-Free Natural Gradient: Memory-Efficient Numerically-Stable KFAC. *International Conference on Machine Learning (ICML).*

[11] Krishnapriyan, A. S., Gholami, A., Zhe, S., Kirby, R. M., Mahoney, M. W. (2021). Characterizing possible failure modes in physics-informed neural networks. *Advances in Neural Information Processing Systems*, 34, 26548-26560.

[12] Gangloff, H., Jouvin, N., et al. (2024). jinns: a JAX Library for Physics-Informed Neural Networks. *arXiv preprint arXiv:2402.08286.*

[13] Dangel, F., et al. (2024). Enhancing Computational Efficiency in Physics-Informed Neural Operators (using Taylor-mode AD). *NeurIPS 2024 Machine Learning and the Physical Sciences Workshop.*

[14] Sun, M., et al. (2023). NKFAC: A Fast and Stable KFAC Optimizer for Deep Neural Networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence.*

[15] Hao, Z., et al. (2023). PINNacle: A Comprehensive Benchmark of Physics-Informed Neural Networks for Solving PDEs. *Advances in Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track.*

[16] Benning, M., et al. (2024). Challenges in Training PINNs: A Loss Landscape Perspective. *arXiv preprint arXiv:2402.10414.*

[17] Yang, G., Hu, E. (2021). Tensor Programs V: Tuning Large Neural Networks via Zero-Shot Hyperparameter Transfer. *International Conference on Machine Learning (ICML).*

[18] Sharma, A., et al. (2024). Towards a Foundation Model for Physics-Informed Neural Networks: Multi-PDE Learning with Active Sampling. *arXiv preprint arXiv:2402.07119.*

[19] Griewank, A., Walther, A. (2008). *Evaluating derivatives: principles and techniques of algorithmic differentiation* (2nd ed.). SIAM.

[20] Botev, A., Ritter, H., Barber, D. (2017). Practical Gauss-Newton optimisation for deep learning. *Proceedings of the 34th International Conference on Machine Learning (ICML)*, 70, 557-565.

[21] Wang, S., Teng, Y., Perdikaris, P. (2021). Understanding and mitigating gradient flow pathologies in physics-informed neural networks. *SIAM Journal on Scientific Computing*, 43(5), A3055-A3081.

[22] Sitzmann, V., Martel, J. N. P., Bergman, A. W., Lindell, D. B., Wetzstein, G. (2020). Implicit Neural Representations with Periodic Activation Functions. *Advances in Neural Information Processing Systems (NeurIPS)*, 33.

[23] Tancik, M., Srinivasan, P. P., Mildenhall, B., Fridovich-Keil, S., Raghavan, N., Singhal, U., ... Ng, R. (2020). Fourier Features Let Networks Learn High Frequency Functions in Low Dimensional Domains. *Advances in Neural Information Processing Systems (NeurIPS)*, 33.

[24] McClenny, L., Braga-Neto, U. M. (2023). Self-Adaptive Physics-Informed Neural Networks. *Journal of Computational Physics.*

[25] Kissas, G., et al. (2020). Machine learning in cardiovascular flows: Aortic blood pressure and wall shear stress prediction from 4D flow MRI. *Computer Methods and Programs in Biomedicine.*