A Technical Deep Dive: Implementing Oscillatory and Physics-Informed Networks for Intracranial Aneurysm Detection

1. Executive Summary

1.1 Purpose and Scope

This report provides a deep technical guide for the implementation of two novel, high-risk, high-reward components proposed in the "Master Blueprint" architecture for the RSNA 2025 Intracranial Aneurysm Detection challenge. The primary objective is to de-risk the implementation of these state-of-the-art modules by translating complex theoretical foundations into practical, actionable engineering strategies. The analysis and recommendations herein are tailored to the specific constraints of the competition, particularly the 16-24GB VRAM limitation and the Kaggle notebook submission environment. The scope is tightly focused on the two primary research directives: the integration of Artificial Kuramoto Oscillatory Neuron (AKOrN) blocks for 3D feature analysis and the application of a Physics-Informed Neural Network (PINN) as an auxiliary regularizer to enforce hemodynamic constraints.

1.2 Key Findings for Directive A (AKOrN)

The integration of AKOrN blocks into a 3D Transformer backbone like WaveFormer is a promising strategy for modeling the complex dynamics of feature interactions. The analysis concludes that the most viable architectural pattern is to insert the AKOrN layer sequentially after the standard MLP block within each Transformer layer. This placement allows the AKOrN to act as a specialized refinement module, operating on features already processed by the attention mechanism. A critical implementation challenge is the conversion of high-dimensional 3D feature maps into a format suitable for the Kuramoto simulation. The recommended approach is to use adaptive 3D average pooling to create a manageable number of "regional" oscillators, balancing spatial summarization with computational feasibility. A key finding from the original AKOrN literature is a non-intuitive performance degradation when the oscillator dimension (analogous to channel depth) becomes too large; therefore, experimentation should prioritize smaller dimensions (e.g., 8, 16) to preserve the crucial feature-binding property of the system. The final synchronization state, quantified by the Kuramoto order parameter, offers a powerful, interpretable metric for model confidence.

1.3 Key Findings for Directive B (PINN)

The use of a PINN as an auxiliary "auditor" head is a theoretically sound method for injecting physical priors into the main classification model. The recommended architecture involves a lightweight, separate Multi-Layer Perceptron (MLP) that takes spatial coordinates as input and is trained to satisfy the Navier-Stokes equations for incompressible blood flow. The gradients from

this physics-based loss regularize the main backbone, encouraging it to learn feature representations that are consistent with plausible hemodynamics. The primary implementation challenge is not the calculation of the physics residual itself—which can be handled robustly using torch.autograd.grad—but the stabilization of the multi-task training dynamics. Analysis of PINN failure modes, particularly from a Neural Tangent Kernel (NTK) perspective, reveals that imbalances in the convergence rates of the data-fitting loss and the physics loss are the principal cause of training failure. Consequently, a naive summation of losses is insufficient. An advanced, automatic loss balancing strategy is required.

1.4 Overarching Strategic Recommendation

Both the AKOrN and PINN modules introduce significant computational overhead. However, their implementation is deemed feasible within the 16-24GB VRAM constraint through meticulous architectural design and hyperparameter selection. The foundational "Master Blueprint" architecture, built upon highly efficient components like SegFormer3D and WaveFormer, provides the necessary computational budget for these advanced modules. For successful implementation, it is imperative to prioritize computationally lean strategies: employing adaptive pooling for AKOrN input, limiting the number of Kuramoto simulation steps, and utilizing an efficient loss-balancing technique like Uncertainty Weighting for the PINN head. A phased implementation plan, starting with a strong baseline and incrementally adding each new module, is strongly recommended to isolate variables and systematically debug these complex components.

2. Analysis of the "Master Blueprint" Architecture

2.1 Foundational Strategy: Efficiency and Domain-Specificity

The proposed "Master Blueprint" architecture is predicated on a robust and well-established two-stage pipeline for object detection: candidate generation followed by classification. This division of labor allows for the development of specialized models at each stage, optimizing for recall in the first and precision in the second. The specific choice of components within this pipeline demonstrates a coherent and strategically sound design philosophy that prioritizes computational efficiency and domain-specificity, which are critical for success under the stringent hardware and time constraints of the competition.

Stage 1: Candidate Generation

The selection of SegFormer3D as the backbone for candidate generation is an exemplary choice for this task.3 As a hierarchical Transformer, it effectively captures multi-scale volumetric features, which is essential for detecting aneurysms of varying sizes. Its most compelling advantage is its extreme efficiency; with up to 33 times fewer parameters and a 13-fold reduction in GFLOPs compared to other state-of-the-art 3D segmentation models, it is

exceptionally well-suited for a fast and memory-light first-pass analysis.3 This efficiency is achieved through an efficient self-attention mechanism and a simple all-MLP decoder, avoiding the complexity of traditional U-Net style decoders.7 The proposal to enhance this architecture with principles from

Vessel Graph Networks (VGN) is a crucial step toward domain-aware detection.[9] Standard segmentation models are topologically agnostic; they may identify a blob-like structure as a candidate even if it is not connected to the vascular tree. By incorporating graph-based reasoning, which explicitly models the connectivity and curvilinear structure of blood vessels, the model can enforce a strong prior that a valid aneurysm candidate must be topologically connected to a vessel. This has the potential to drastically reduce the number of false positives passed to the second stage, thereby improving the overall pipeline's efficiency and accuracy.[11]

Stage 2: Classification Backbone

For the more computationally intensive classification stage, the selection of WaveFormer as the backbone is a strategic continuation of the efficiency-first principle.13 Standard 3D Transformers are often prohibitively expensive due to the cubic growth of tokens with spatial resolution. WaveFormer mitigates this by employing a Discrete Wavelet Transform (DWT) to decompose feature maps into low-frequency (global context) and high-frequency (local detail) sub-bands.14 The computationally demanding self-attention mechanism is then applied only to the compact, low-frequency representation, significantly reducing the token count and, consequently, the VRAM and computational requirements.15 The use of an Inverse DWT (IDWT) for the decoder further enhances efficiency by replacing parameter-heavy upsampling layers.16

Pre-training this powerful backbone with **SparK** aligns with modern best practices in deep learning, particularly for medical imaging where labeled data is scarce.[17] SparK is a masked image modeling (MIM) approach that extends the success of masked autoencoders to convolutional and hybrid architectures by using sparse convolutions to efficiently process only the visible (unmasked) patches.[18] This self-supervised pre-training strategy allows the WaveFormer to learn robust and generalizable representations of cerebrovascular anatomy from a large corpus of unlabeled 3D medical images before being fine-tuned on the specific aneurysm detection task. Studies have shown that SparK is particularly robust to reductions in the size of the fine-tuning dataset, making it an ideal choice for this application.[17]

2.2 Deeper Implications of Architectural Choices

The selection of these specific components is not merely a collection of high-performing models but reflects a deeply integrated design philosophy. The consistent prioritization of VRAM and computational efficiency at every level—from the lightweight SegFormer3D to the token-reducing WaveFormer and the sparse-convolution-based SparK—is a prerequisite for creating

the necessary computational headroom to accommodate the advanced AKOrN and PINN modules. Without this efficient foundation, the proposed novel components would be computationally infeasible within the 16GB VRAM limit.

Furthermore, the architecture of WaveFormer introduces a physically meaningful dichotomy in its feature representations. The DWT inherently separates features into low-frequency components, which capture global context, shape, and overall structure, and high-frequency components, which capture local details, textures, and fine-grained boundaries.[13] This is a powerful architectural prior not present in standard Transformers. This separation can be strategically exploited by the subsequent novel modules. For instance, the "binding" mechanism of AKOrN, which seeks to find consensus among features, may be more effective and computationally tractable when applied to the compressed, global, low-frequency features that define the aneurysm's morphology. Similarly, the PINN's hemodynamic constraints are macroscopic physical laws governing overall flow patterns, making them more conceptually aligned with the low-frequency representation of the vessel and aneurysm shape rather than high-frequency texture details. This natural feature separation within the backbone will inform the optimal placement and application of the AKOrN and PINN modules discussed in the following sections.

3. Directive A - Deep Dive on Artificial Kuramoto Oscillatory Neuron (AKOrN) for 3D Volumetric Analysis

3.1 Core Concepts and Mathematical Formulation

To understand the application of Artificial Kuramoto Oscillatory Neurons (AKOrN) to 3D image analysis, it is essential to first grasp its foundations in the Kuramoto model of synchronization.

From Kuramoto to AKOrN

The classic Kuramoto model describes a population of coupled oscillators, each with its own intrinsic frequency, that can spontaneously synchronize their phases when their mutual coupling strength exceeds a critical threshold.19 The state of the system can be summarized by a complex order parameter,

reiψ=N1∑j=1Neiθj, where N is the number of oscillators and θj is the phase of oscillator j.[20] The magnitude of this parameter,

r, serves as a direct measure of global synchrony: r≈0 corresponds to an incoherent state where phases are uniformly distributed, while r=1 signifies a perfectly synchronized state where all oscillators share the same phase.[21]

The AKOrN model, introduced by Miyato et al., generalizes this concept from a scalar phase $\theta_j \in This allows for a richer representation of each oscillator's state. The dynamics of each oscillator i are governed by a generalized Kuramoto differential equation:

$$\dot{x}_i = \Omega_i x_i + \text{Proj}_{x_i}(c_i + \sum_j J_{ij} x_j)$$

where:

- x_i is the N-dimensional state vector of oscillator i.
- Ω_i is a learnable N×N anti-symmetric matrix that determines the natural frequency and rotation of the oscillator.
- c_i is the external stimulus or conditional input. This term acts as a "bias direction" or a "symmetry breaking" field, anchoring the oscillator's dynamics to external data. This is the primary mechanism through which image features are injected into the dynamical system.
- J_{ij} is the learnable coupling matrix (or tensor) that defines the strength and nature of the interaction between oscillator i and oscillator j.
- $\text{Proj}_{x_i}(y) = y - \langle y, x_i \rangle x_i$ is a projection operator that ensures the update direction is tangential to the sphere, thus preserving the unit norm of x_i.

Formulation for Image Tasks

For application to image-based tasks, the oscillators are arranged on a spatial grid, mirroring the structure of a convolutional feature map. The interaction term $\sum_j J_{ij} x_j$ is implemented as a convolution. The state of an oscillator at channel c, height h, and width w is updated based on its local neighborhood. The update direction $y_{c,h,w}$ is given by:

$$y_{c,h,w} := c_{c,h,w} + d \sum_{h', w' \in R} J_{c,d,h',w'} x_{d,(h+h'),(w+w')}$$

Here, $J_{c,d,h',w'}$ are the weights of a convolutional kernel, d indexes the input channels, and the sum over h',w' covers the spatial extent of the kernel R.22 In this formulation, the learnable coupling matrix

J is effectively a convolutional filter, and the external stimuli C are derived from the input feature map provided to the AKOrN layer.

3.2 Architectural Integration into a 3D Transformer Backbone

A standard block in a Vision Transformer architecture like WaveFormer consists of Layer Normalization, a Multi-Head Self-Attention module, another Layer Normalization, and an MLP

(Feed-Forward Network) block, with residual connections around the attention and MLP blocks. The ThreeDimensionalAKOrNLayer can be integrated into this structure in several ways.

Strategy 1: MLP Replacement

In this direct approach, the MLP block is entirely replaced by the AKOrN layer. The output of the attention mechanism, after the first residual connection and layer normalization, is fed into the AKOrN layer. This layer's output is then passed through the second residual connection.

- Diagram:

!(https://i.imgur.com/8Qp4l7C.png)

- **Rationale:** This strategy assumes that the dynamical synchronization process of AKOrN can serve as a more powerful and expressive feature transformation than a standard MLP. It is the most parameter-efficient of the proposed strategies if the AKOrN's internal layers are kept small.

Strategy 2: Sequential Augmentation (Recommended)

Here, the AKOrN layer is placed sequentially after the MLP block, before the final residual connection. The standard Transformer block first performs its feature extraction via attention and MLP, and the AKOrN layer then acts as a subsequent refinement module, encouraging feature binding and consensus.

- Diagram:

!(https://i.imgur.com/Fw5j5yF.png)

- **Rationale:** This approach is less disruptive to the standard Transformer architecture. It leverages the known strengths of the MLP for feature transformation while adding the unique capabilities of AKOrN as a specialized processing step. The original AKOrN paper demonstrates that increasing the number of simulation steps at test time can improve performance on reasoning tasks, suggesting an iterative refinement process that aligns well with this sequential design.

Strategy 3: Parallel Gating

This strategy places the MLP and AKOrN layers in parallel. Both modules process the same input from the attention block. Their outputs are then combined, for example, through a weighted sum controlled by a learnable gating mechanism.

- Diagram:

!(https://i.imgur.com/488v2Yg.png)

- **Rationale:** This offers the most flexibility, allowing the network to learn how to balance standard feature transformation (MLP) with dynamical refinement (AKOrN) for different inputs. However, it is the most computationally and parametrically expensive option.

Given the exploratory nature of this module, the **Sequential Augmentation** strategy is recommended as the primary approach. It preserves the integrity of the well-understood Transformer block while adding the AKOrN's functionality as a distinct, analyzable stage.

3.3 From 3D Feature Maps to Sequential Dynamics

A fundamental engineering challenge is converting the 5D feature map tensor of shape `` (Batch, Channels, Depth, Height, Width) from the WaveFormer backbone into a 1D sequence of oscillators required for the Kuramoto simulation.

Method 1: Voxel-wise Flattening

This method treats each spatial voxel as an independent oscillator. The feature map is reshaped from to, where M=D×H×W is the total number of voxels. The channel dimension C becomes the oscillator state dimension N, and M becomes the number of oscillators. While this preserves maximum spatial information, the number of oscillators M becomes extremely large even for moderately sized feature maps (e.g., 16×32×32=16384), making the pairwise interactions computationally intractable and certain to exceed the 16GB VRAM limit.23

Method 2: Adaptive 3D Pooling (Recommended)

To manage the computational complexity, an adaptive pooling layer (e.g., torch.nn.AdaptiveAvgPool3d) can be applied to the spatial dimensions of the feature map before it enters the AKOrN layer. For example, the tensor could be pooled to, where D’,H’,W’ are small, fixed integers (e.g., 4). This reduces the number of oscillators to a manageable M=D’×H’×W’ (e.g., 4×4×4=64). These 64 "super-voxels" represent regional summaries of the feature map. This approach aggressively reduces dimensionality and loses fine-grained spatial detail but makes the simulation computationally feasible.24 It introduces a critical trade-off between spatial resolution and computational cost.

Method 3: Channel-wise Oscillators

An alternative interpretation is to treat the channel dimension C as the number of oscillators. The tensor is reshaped from to, where N=D×H×W. Here, there are C oscillators, and the state of each is a very high-dimensional vector representing the entire spatial feature map for that channel. This is conceptually misaligned with the convolutional formulation in the AKOrN paper 22 and is likely less effective, but remains a theoretical possibility.

Given the project's constraints, **Adaptive 3D Pooling** is the most pragmatic and recommended approach. The size of the pooled grid (e.g., 4×4×4 vs. 8×8×8) becomes a key hyperparameter to tune.

Strategy	Architectural Placement	Feature Conversion	VRAM Impact	Computational Cost	Conceptual Pros	Conceptual Cons
MLP Replacement	Replaces MLP block in Transformer layer	Adaptive Pooling	Medium	Medium	Parameter-efficient; direct integration of dynamics.	Disrupts standard Transformer block; may underperform if MLP is crucial.
Sequential Augmentation (Recommended)	After MLP block in Transformer layer	Adaptive Pooling	High	High	Preserves standard block; acts as a feature refiner; aligns with iterative nature.	Increases layer depth and computational latency; highest parameter count.
Parallel Gating	Parallel to MLP block	Adaptive Pooling	High	High	Highly flexible; network learns to balance	Most complex; adds gating parameters; potential

Strategy	Architectural Placement	Feature Conversion	VRAM Impact	Computational Cost	Conceptual Pros	Conceptual Cons
					standard vs. dynamic features.	for vanishing gradients in one path.
Voxel-wise Flattening	Any	Direct Flattening	Prohibitive	Prohibitive	Preserves all spatial information.	Computationally intractable for typical 3D feature maps; will not fit in VRAM.

3.4 Conceptual PyTorch Implementation

The following conceptual code snippet illustrates the implementation of a ThreeDimensionalAKOrNLayer module, incorporating the recommended sequential augmentation and adaptive pooling strategies.

Python

```
import torch

import torch.nn as nn

import torch.nn.functional as F

# Assuming MONAI or other library provides 3D layers

from monai.networks.layers import Conv


class ThreeDimensionalAKOrNLayer(nn.Module):
```

"""

Implements the Artificial Kuramoto Oscillatory Neuron layer for 3D feature maps.

Assumes input tensor of shape.

"""

```
def __init__(self, in_channels, oscillator_dim, num_oscillators_per_axis=4,
             kernel_size=3, num_steps=16, dt=0.1):
    """
    Args:
        in_channels (int): Number of channels in the input feature map.
        oscillator_dim (int): The dimension 'N' of each oscillator state vector.
                              This is a critical hyperparameter.
        num_oscillators_per_axis (int): The target spatial dimension after pooling.
                                        Total oscillators will be this value cubed.
        kernel_size (int): The kernel size for the convolutional coupling 'J'.
        num_steps (int): Number of simulation steps 'T'.
        dt (float): Time step for the Euler integration.
    """
    super().__init__()
    self.in_channels = in_channels
    self.oscillator_dim = oscillator_dim
    self.num_oscillators_total = num_oscillators_per_axis ** 3
    self.num_steps = num_steps
    self.dt = dt

    # Adaptive pooling to create a fixed number of regional oscillators
    self.pool = nn.AdaptiveAvgPool3d((num_oscillators_per_axis,) * 3)
```

```
        # Linear layers to project pooled features into initial oscillator states (X)
        # and conditional stimuli (C).
        self.proj_x = nn.Linear(in_channels, self.oscillator_dim)
        self.proj_c = nn.Linear(in_channels, self.oscillator_dim)

        # Learnable coupling 'J' implemented as a 3D convolution.
        # The groups parameter makes it a depthwise-like convolution over the oscillator dimension.
        self.conv_J = Conv(
            in_channels=self.oscillator_dim,
            out_channels=self.oscillator_dim,
            kernel_size=kernel_size,
            padding=kernel_size // 2,
            groups=self.oscillator_dim, # Depthwise-like operation
            bias=False
        )

        # Learnable natural frequency matrix 'Omega' for each oscillator.
        # It must be anti-symmetric. We learn a general matrix and make it anti-symmetric.
        self.W_omega = nn.Parameter(torch.randn(self.num_oscillators_total,
                                                self.oscillator_dim, self.oscillator_dim))

    def forward(self, x_in):
        # x_in shape:
        batch_size = x_in.shape
```

```
# 1. Pool features to create regional "super-voxels"
pooled_features = self.pool(x_in) # Shape:

# 2. Reshape for processing: flatten spatial dims
# Shape: ->
pooled_features = pooled_features.flatten(2).permute(0, 2, 1)

# 3. Project to get initial oscillator states and stimuli
# Shape of x and c:
x = self.proj_x(pooled_features)
c = self.proj_c(pooled_features)

# Normalize initial states to lie on the (N-1)-sphere
x = F.normalize(x, p=2, dim=-1)

# 4. Run the Kuramoto simulation loop
# Make Omega anti-symmetric
omega = self.W_omega - self.W_omega.transpose(1, 2)

for _ in range(self.num_steps):
    # Calculate natural frequency term: Omega * x
    # Unsqueeze for batch dimension broadcasting
    natural_freq_term = torch.einsum('mab,bma->ba', omega, x)

    # Calculate coupling term: sum(J_ij * x_j)
```

```
    # Reshape for convolution: ->
    x_reshaped = x.permute(0, 2, 1).view(
        batch_size, self.oscillator_dim,
        self.pool.output_size, self.pool.output_size, self.pool.output_size
    )
    coupling_term_conv = self.conv_J(x_reshaped)
    # Reshape back: ->
    coupling_term = coupling_term_conv.flatten(2).permute(0, 2, 1)

    # Total force before projection
    y = c + coupling_term

    # Projection step to stay on the tangent plane of the sphere
    proj_y = y - (torch.einsum('bmn,bmn->bm', y, x).unsqueeze(-1) * x)

    # Euler integration step
    dx = (natural_freq_term + proj_y) * self.dt
    x = F.normalize(x + dx, p=2, dim=-1)

# 5. Reshape final states back to a 5D tensor format to match input
# This step depends on the desired output format. One option is to upsample.
# Here, we return the processed pooled features.
# Shape: -> ->
x_out_pooled = x.permute(0, 2, 1).view(
    batch_size, self.oscillator_dim,
    self.pool.output_size, self.pool.output_size, self.pool.output_size
```

```
        )

        # Upsample to original spatial dimensions to be added in residual connection
        x_out = F.interpolate(x_out_pooled, size=x_in.shape[2:], mode='trilinear',
align_corners=False)

        return x_out
```

3.5 Interpretability and Model Explainability

A significant advantage of using a dynamics-based module like AKOrN is its inherent interpretability, which can provide valuable insights into the model's decision-making process.

Kuramoto Order Parameter (r) as a Confidence Score

The global level of synchronization of the oscillators for a given input can be calculated from their final states. For each sample in the batch, the order parameter r is the magnitude of the mean vector of all its oscillator states:

r= M1i=1∑Mxi 2

where M is the total number of oscillators. This scalar value, ranging from 0 to 1, provides a direct measure of the "consensus" or "binding" achieved by the features representing the input candidate. A high value of r (e.g., > 0.8) suggests that the oscillators, driven by the input features, have converged to a coherent, synchronized state. This could be interpreted as the model having high confidence in its extracted representation. Conversely, a low value of r might indicate an ambiguous or out-of-distribution case where the features failed to produce a consensus. By analyzing the distribution of r for true positives, false positives, and false negatives, it may be possible to establish a correlation between synchronization and prediction correctness, providing a powerful tool for error analysis and uncertainty quantification.

Learned Coupling Matrix (J) as an Anatomical Prior

The weights of the convolutional kernel that implements the coupling matrix J are learned during training. These weights determine how neighboring regional features influence each other to promote or inhibit synchronization. Visualizing these learned 3D kernels can reveal the spatial patterns the model has identified as important for feature binding.25 For instance, the model might learn a kernel with strong positive weights along a particular axis, effectively learning to bind features that form a continuous, vessel-like structure. It might learn inhibitory

connections for patterns that are anatomically implausible. This provides a window into the model's learned anatomical priors, moving beyond a simple "black box" and toward a model whose internal mechanisms can be inspected and understood.

A critical risk to highlight, however, is the finding from the original AKOrN paper that performance can suddenly and catastrophically drop if the oscillator dimension N is too large. The authors suggest this is because the essential binding property is lost in very high-dimensional spaces. This is a crucial, non-obvious departure from typical deep learning practice, where increasing channel depth (analogous to N) is a standard method for increasing model capacity. Therefore, the implementation strategy must involve careful experimentation with small oscillator dimensions (e.g., N in {4, 8, 16, 32}) and not default to larger values common in other architectures.

4. Directive B - Deep Dive on Physics-Informed Neural Networks (PINN) as an Auxiliary Regularizer

4.1 Core Concepts and Best Practices in Medical Imaging

Physics-Informed Neural Networks (PINNs) represent a paradigm shift from purely data-driven models to hybrid models that integrate domain knowledge in the form of governing physical laws.[26] In the context of this project, the PINN is not used to solve a partial differential equation (PDE) from scratch, but rather to act as a regularizer for a primary computer vision model. The core principle is to add the residual of a relevant PDE—in this case, the Navier-Stokes equations for blood flow—as a penalty term in the total loss function.[28] By minimizing this physics loss, the main network is encouraged to learn feature representations that are not only effective for the classification task but also consistent with the fundamental laws of fluid dynamics. This approach can enhance model robustness, improve generalization with fewer data, and increase interpretability.[29]

The application of PINNs in medical imaging is a growing field, with successful uses in tasks like MRI reconstruction, image registration, and biomechanical parameter estimation.[31] These applications demonstrate the value of incorporating physics priors related to imaging processes and tissue properties into the learning framework.

However, the training of PINNs is notoriously challenging. A seminal work by Wang et al. analyzed these training difficulties through the lens of the Neural Tangent Kernel (NTK). The key finding is that PINNs often fail to train due to a severe imbalance in the training dynamics between the different components of the loss function. The data-fitting term (e.g., boundary conditions or measurements) and the physics-residual term often have vastly different magnitudes and associated gradient dynamics, leading to a discrepancy in their convergence rates.[34] One loss term can easily dominate the training, causing the optimizer to become

trapped in a poor local minimum where the other loss term is high. This fundamental pathology explains many of the observed training failures.

Failure Mode	Underlying Cause (NTK Perspective)	Primary Mitigation Strategy	Secondary Strategies
Physics loss stagnates or fails to decrease.	**Convergence Rate Imbalance:** The data-fitting loss converges much faster than the physics loss, causing its gradients to dominate and the optimizer to ignore the physics residual.	**Dynamic Loss Balancing:** Implement an automated weighting scheme (e.g., Uncertainty Weighting) to adaptively balance the influence of each loss term during training.[35]	Use curriculum learning: train on data loss first, then gradually introduce the physics loss.
Model converges to a trivial or non-physical solution (e.g., zero velocity field).	**Stiff/Complex Loss Landscape:** High-order derivatives in the PDE residual can create a complex, non-convex loss landscape with many poor local minima.[37]	**Gradient-Enhanced PINNs:** Include gradient information of the solution in the loss function to provide a stronger, more direct signal to the optimizer.[38]	Use transfer learning from a simpler, related physical problem to achieve a better weight initialization.[39]
Unstable or oscillating loss curves during training.	**Gradient Pathologies:** The backpropagated gradients from high-order derivative terms in the PDE can become noisy or "contaminated," leading	**Sequential Training:** Break the problem into smaller, sequential time or spatial steps, using the solution of one step to initialize the	Employ adaptive activation functions or modified network architectures designed to have better gradient properties.

Failure Mode	Underlying Cause (NTK Perspective)	Primary Mitigation Strategy	Secondary Strategies
	to unstable training steps.	next, which stabilizes training.[41]	

4.2 Architectural Integration as an Auxiliary Head

The PINN regularizer is integrated as an auxiliary "auditor" head that operates in parallel with the primary classification head. The architecture ensures that the PINN can influence the shared backbone without interfering with the primary task's output structure.

Information and Gradient Flow:

The diagram below illustrates the flow of information and, crucially, the flow of gradients.

!(https://i.imgur.com/z0m257P.png)

1. **Shared Backbone (WaveFormer):** The input 3D image volume is processed by the WaveFormer backbone, which generates a rich, multi-scale feature map. This feature map is then pooled into a feature vector and passed to the primary classification head for aneurysm prediction. This constitutes the main task path.
2. **PINN Head (Auditor):** A separate, lightweight MLP is defined as the PINN head. This network does not directly see the image or the full feature map. Instead, its inputs are randomly sampled spatial coordinates (x, y, z) from the domain of the input image volume.
3. **Feature Interpolation:** At each sampled coordinate (x, y, z), the corresponding feature vector from the backbone's final feature map is extracted via trilinear interpolation. This interpolated feature vector is concatenated with the coordinates and serves as the full input to the PINN head.
4. **Physics Prediction:** The PINN head takes this combined input and predicts the physical quantities of interest at that location: the velocity vector (u,v,w) and the pressure p.
5. **Physics Loss Calculation:** Using these predictions, the residual of the Navier-Stokes equations is calculated via automatic differentiation. This residual forms the physics_loss.
6. **Gradient Backpropagation:** The total loss is a weighted sum of the primary classification_loss and the auxiliary physics_loss. When total_loss.backward() is called,

gradients from the physics_loss flow back through the PINN head and, importantly, back into the shared backbone via the interpolated feature vectors. This acts as a regularization signal, penalizing the backbone if it produces feature representations that lead to physically implausible predictions by the PINN head. The PINN effectively "audits" the backbone's features for physical consistency.

4.3 Practical Implementation of the Navier-Stokes Physics Loss

The core of the PINN module is the function that calculates the residual of the governing equations. For intracranial blood flow, a reasonable approximation is the steady-state, incompressible Navier-Stokes equations. The following conceptual PyTorch code demonstrates how to compute this residual using automatic differentiation.

Python

```
import torch
import torch.nn as nn


def calculate_navier_stokes_residual(coords, pinn_head, rho=1060., mu=0.0035):
    """
    Calculates the residual of the incompressible Navier-Stokes equations.

    Args:
        coords (torch.Tensor): A tensor of shape [N, 3] for (x, y, z) coordinates.
                               Requires gradients to be enabled.
        pinn_head (nn.Module): The MLP that maps coordinates to (u, v, w, p).
        rho (float): Density of blood (kg/m^3).
        mu (float): Dynamic viscosity of blood (Pa·s).

    Returns:
        torch.Tensor: The mean squared error of the PDE residuals.
    """
```

```
    # Ensure coordinates require gradients for differentiation
    coords.requires_grad_(True)

    # Predict velocity (u, v, w) and pressure (p)
    predictions = pinn_head(coords)
    u, v, w, p = predictions[:, 0], predictions[:, 1], predictions[:, 2], predictions[:, 3]

    # Use torch.autograd.grad to compute first-order derivatives
    # We compute derivatives of each output component (u, v, w, p)
    # with respect to each input component (x, y, z).
    grad_u = torch.autograd.grad(u, coords, grad_outputs=torch.ones_like(u),
create_graph=True)
    du_dx, du_dy, du_dz = grad_u[:, 0], grad_u[:, 1], grad_u[:, 2]

    grad_v = torch.autograd.grad(v, coords, grad_outputs=torch.ones_like(v), create_graph=True)
    dv_dx, dv_dy, dv_dz = grad_v[:, 0], grad_v[:, 1], grad_v[:, 2]

    grad_w = torch.autograd.grad(w, coords, grad_outputs=torch.ones_like(w),
create_graph=True)
    dw_dx, dw_dy, dw_dz = grad_w[:, 0], grad_w[:, 1], grad_w[:, 2]

    grad_p = torch.autograd.grad(p, coords, grad_outputs=torch.ones_like(p),
create_graph=True)
    dp_dx, dp_dy, dp_dz = grad_p[:, 0], grad_p[:, 1], grad_p[:, 2]

    # Compute second-order derivatives (Laplacian components)
```

```
    d2u_dx2 = torch.autograd.grad(du_dx, coords, grad_outputs=torch.ones_like(du_dx), create_graph=True)[:, 0]

    d2u_dy2 = torch.autograd.grad(du_dy, coords, grad_outputs=torch.ones_like(du_dy), create_graph=True)[:, 1]

    d2u_dz2 = torch.autograd.grad(du_dz, coords, grad_outputs=torch.ones_like(du_dz), create_graph=True)[:, 2]

    laplacian_u = d2u_dx2 + d2u_dy2 + d2u_dz2


    d2v_dx2 = torch.autograd.grad(dv_dx, coords, grad_outputs=torch.ones_like(dv_dx), create_graph=True)[:, 0]

    d2v_dy2 = torch.autograd.grad(dv_dy, coords, grad_outputs=torch.ones_like(dv_dy), create_graph=True)[:, 1]

    d2v_dz2 = torch.autograd.grad(dv_dz, coords, grad_outputs=torch.ones_like(dv_dz), create_graph=True)[:, 2]

    laplacian_v = d2v_dx2 + d2v_dy2 + d2v_dz2


    d2w_dx2 = torch.autograd.grad(dw_dx, coords, grad_outputs=torch.ones_like(dw_dx), create_graph=True)[:, 0]

    d2w_dy2 = torch.autograd.grad(dw_dy, coords, grad_outputs=torch.ones_like(dw_dy), create_graph=True)[:, 1]

    d2w_dz2 = torch.autograd.grad(dw_dz, coords, grad_outputs=torch.ones_like(dw_dz), create_graph=True)[:, 2]

    laplacian_w = d2w_dx2 + d2w_dy2 + d2w_dz2


    # Assemble the residuals for the three momentum equations and the continuity equation

    # Momentum in x: rho * (u*du/dx + v*du/dy + w*du/dz) = -dp/dx + mu * laplacian(u)

    residual_x = rho * (u * du_dx + v * du_dy + w * du_dz) + dp_dx - mu * laplacian_u


    # Momentum in y: rho * (u*dv/dx + v*dv/dy + w*dv/dz) = -dp/dy + mu * laplacian(v)
```

```
    residual_y = rho * (u * dv_dx + v * dv_dy + w * dv_dz) + dp_dy - mu * laplacian_v

    # Momentum in z: rho * (u*dw/dx + v*dw/dy + w*dw/dz) = -dp/dz + mu * laplacian(w)
    residual_z = rho * (u * dw_dx + v * dw_dy + w * dw_dz) + dp_dz - mu * laplacian_w

    # Continuity (incompressibility): du/dx + dv/dy + dw/dz = 0
    residual_continuity = du_dx + dv_dy + dw_dz

    # Calculate the mean squared error of all residuals
    loss_x = torch.mean(residual_x**2)
    loss_y = torch.mean(residual_y**2)
    loss_z = torch.mean(residual_z**2)
    loss_continuity = torch.mean(residual_continuity**2)

    total_physics_loss = loss_x + loss_y + loss_z + loss_continuity
    return total_physics_loss
```

4.4 State-of-the-Art Strategies for Balancing Multi-Task Losses

As established by the NTK analysis, simply adding the classification and physics losses (L_total = L_class + L_physics) is a recipe for training failure. The disparate nature and scale of these two loss terms necessitate a sophisticated balancing mechanism.

Method 1: Uncertainty Weighting (Loss-based, Recommended)

This method, proposed by Kendall et al., frames multi-task learning in a probabilistic context by weighting each task's loss by its learned homoscedastic uncertainty.44 The total loss function for two tasks becomes:

$$L_{total} = \frac{2}{\sigma_1^2} L_{class} + \frac{2}{\sigma_2^2} L_{physics} + \log(\sigma_1) + \log(\sigma_2)$$

Here, σ_1 and σ_2 are learnable scalar parameters representing the uncertainty of the classification and physics tasks, respectively. The network learns to down-weight the loss of a task with high uncertainty (large σ) while the $\log(\sigma)$ terms prevent the network from trivially

learning infinitely large uncertainties. This approach is computationally efficient, as it only adds two learnable parameters to the optimizer and does not require multiple backward passes or direct gradient manipulation.[45]

Method 2: Gradient Balancing (Gradient-based)

This family of methods directly addresses gradient conflicts between tasks.46 Algorithms like GradNorm, PCGrad, and MGDA operate by computing the per-task gradients on a shared layer of the network. They then modify these gradients—by normalizing their magnitudes (GradNorm), projecting them to remove conflicting components (PCGrad), or finding a common descent direction (MGDA)—before applying the final update to the shared weights. While often yielding slightly better performance by explicitly managing gradient interference, these methods incur a significant computational cost. They typically require storing per-task gradients and/or performing multiple backward passes at each step, which substantially increases both training time and VRAM usage.48

Method Family	Specific Algorithm	Core Principle	Computational Cost	VRAM Overhead	Pros	Cons
Manual Weighting	Simple Weighted Sum	Manually tune fixed weights λ_1, λ_2.	Very Low	None	Simple to implement.	Brittle; requires extensive tuning; static weights cannot adapt to training dynamics.
Loss-based (Recommended)	**Uncertainty Weighting**	Weight losses by their learned homoscedastic	**Low**	**Minimal** (2 extra parameters)	**Adapts automatically; computationally**	Only applicable if task uncertainty can be

Method Family	Specific Algorithm	Core Principle	Computational Cost	VRAM Overhead	Pros	Cons
		uncertainty.			**cheap; robust.**	modeled; may be slightly less performant than gradient methods.
Gradient-based	GradNorm, PCGrad, MGDA	Directly manipulate per-task gradients to resolve conflicts.	**High**	**High** (stores multiple gradients)	Can achieve SOTA performance; directly resolves gradient conflicts.	**High computational and memory cost; complex to implement; may not be feasible within Kaggle limits.**

Recommendation:

Given the strict computational budget of the Kaggle competition environment, Uncertainty Weighting is the strongly recommended approach. Its low overhead in both VRAM and computation makes it a practical and robust choice for balancing the classification and physics losses. Gradient balancing methods should be considered a secondary, high-cost alternative to be explored only if the performance of Uncertainty Weighting proves insufficient and the hardware budget permits.

5. Synthesis and Strategic Recommendations

5.1 Integrated Challenges and Interdependencies

The successful implementation of the "Master Blueprint" requires navigating the combined computational burden of its most novel components. The AKOrN module introduces a simulation loop within the forward pass, adding a computational cost that scales linearly with the number of simulation steps. The PINN module requires repeated calculations of high-order derivatives via automatic differentiation, which, while efficient, is more intensive than a standard loss calculation.

However, a potential synergistic relationship exists between these two modules. The PINN regularizer forces the backbone network to learn feature representations that are smooth and consistent with the laws of fluid dynamics. This could result in a more structured and physically plausible feature space. A well-structured feature space, in turn, might be an ideal substrate for the AKOrN oscillators. The synchronization process could converge more quickly and to a more stable state when operating on features that already encode physical priors, potentially reducing the required number of simulation steps and mitigating some of the AKOrN's computational cost. This hypothesized synergy—where physics-informing leads to more stable dynamics—is a novel research direction that could be explored during model development.

5.2 Phased Implementation and Experimentation Plan

To manage the complexity of this architecture, a phased, incremental implementation and validation plan is essential. This approach allows for systematic debugging and isolates the performance contribution of each component.

- **Phase 1 (Establish Baseline):** The first priority is to train the core classification architecture—the WaveFormer backbone pre-trained with SparK—without the AKOrN or PINN modules. The goal is to establish a strong, reliable performance baseline on the competition metric. This step is crucial for quantifying the true impact of the subsequent, more experimental modules.
- **Phase 2 (AKOrN Integration):** Implement the ThreeDimensionalAKOrNLayer using the recommended Sequential Augmentation strategy. Integrate it into the baseline model from Phase 1. The primary focus of experimentation in this phase should be on tuning the AKOrN-specific hyperparameters: the number of simulation steps T, the oscillator dimension N (starting with small values), and the adaptive pooling grid size.
- **Phase 3 (PINN Integration):** Starting again from the Phase 1 baseline, integrate the auxiliary PINN head and the Uncertainty Weighting loss mechanism. The key experiments in this phase will involve tuning the initial values of the uncertainty

parameters (σ1,σ2) and determining the optimal sampling strategy and number of collocation points for calculating the physics loss.

- **Phase 4 (Full Model Integration):** Combine the validated components from Phases 2 and 3 into the final "Master Blueprint" architecture. This final phase will involve end-to-end training and a final hyperparameter sweep, focusing on the learning rate and the interaction between the different model components.

5.3 Final Recommendations for Kaggle Environment

The Kaggle platform imposes strict constraints on runtime and resources, which must be central to the implementation strategy.

- **Prioritize Efficiency:** At every stage, default to the most computationally efficient options. Begin with a small number of AKOrN simulation steps (e.g., 8-12), a coarse adaptive pooling grid (e.g., 4×4×4), and the Uncertainty Weighting method for the PINN. Complexity should only be increased if performance gains are demonstrated and the time/memory budget allows.
- **Pre-computation and Caching:** The Kaggle environment often has time limits on inference. Any part of the pipeline that can be pre-computed and saved to disk (e.g., vessel segmentation masks from Stage 1, pre-processed NIfTI files) should be. This minimizes the work that needs to be done within the time-limited submission notebook.
- **Model Ensembling:** A single, complex model is unlikely to be the winning solution. A more robust strategy is to train several variants of the "Master Blueprint" (e.g., with and without AKOrN/PINN, with different backbones) and ensemble their predictions. The interpretability features of the AKOrN (order parameter) and PINN (physics loss magnitude) can even be used as features for a second-level meta-model that combines the predictions of the primary models.
- **Robust Validation:** A rigorous cross-validation scheme that respects the patient-level grouping of the data is paramount. Overfitting to the public leaderboard is a common pitfall, and a stable local CV score is the most reliable indicator of final performance.