Guardians of Carbon - II

ECO-AI Hackthon Final Report – a single frame training strategy

Zhenkai (Josh) Bo, Heng Wang, Muhammad Ali, Nortier Bertrand

Ali

1. Segmentation problem

- Manipulate the target
- Loss function

2. Replace U-net with viT

```
│    │    │    └─GroupNorm (norm2)          [1, 16, 512, 128]    [1, 16, 512, 128]    32
│    │    │    └─Dropout2d (dropout2)       [1, 16, 512, 128]    [1, 16, 512, 128]    --
├─Conv2d (conv_out)                         [1, 16, 512, 128]    [1, 3, 512, 128]     435
==========================================================================================
```

```
Epoch 1/2: 100%|██████████| 177/177 [00:54<00:00,  3.28it/s, train_loss=-37214.750271]
Epoch 1/2:
Train Loss: -37214.750271
Val Loss: -794578.789402
Learning Rate: 0.000051
Saved new best model with validatio
----------------------------------
Epoch 2/2: 100%|██████████| 177/177
Epoch 2/2:
Train Loss: -175763303.630297
Val Loss: -1197841382.956522
Learning Rate: 0.000001
Saved new best model with validation loss: -1197841382.956522
--------------------------------------------------
```

```
OutOfMemoryError: CUDA out of memory. Tried to allocate 128.00 MiB. GPU 0 has a total capacity of 15.89 GiB of which 123.12 MiB is free. Process 6052 has 15.77 GiB memory in use. Of the allocated memory 15.42 GiB is allocated by PyTorch, and 54.39 MiB is reserved by PyTorch but unallocated. If reserved but unallocated memory is large try setting PYTORCH_CUDA_ALLOC_CONF=expandable_segments:True to avoid fragmentation.  See documentation for Memory Management  (https://pytorch.org/docs/stable/notes/cuda.html#environment-variables)
```

Claude

I'll create a Vision Transformer (ViT) model that follows a similar architecture and training loop structure. I'll break this down into two parts: the mo training function.

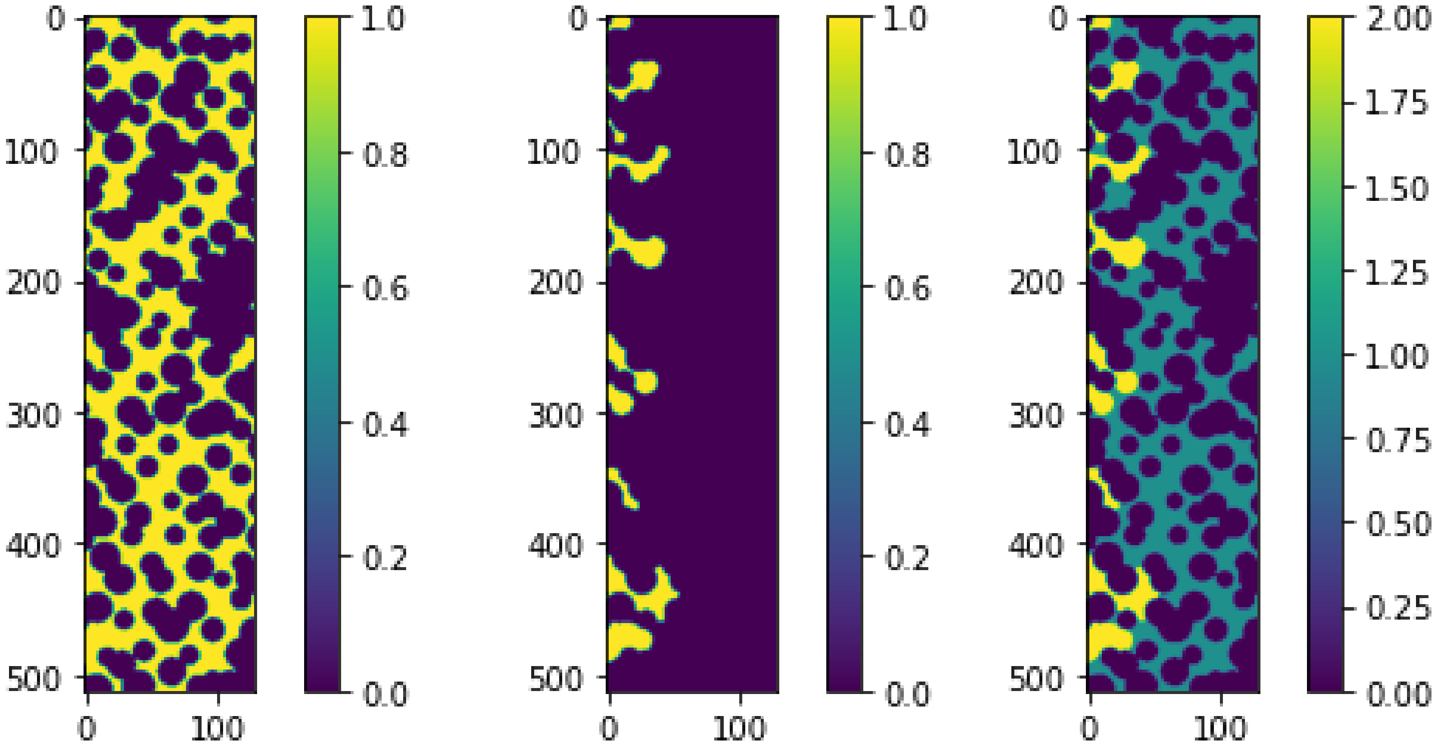
Vision Transformer M
Click to open code

Key differences and features of t implementation:

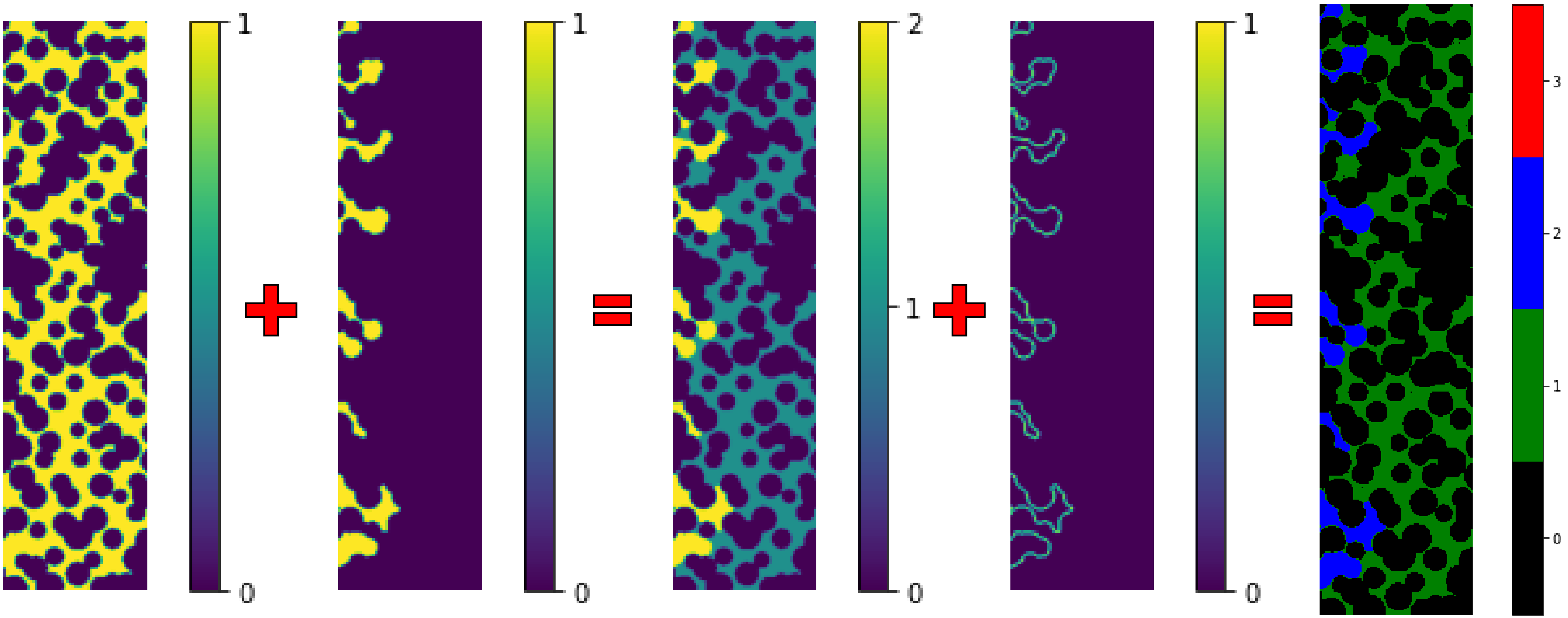
1. Model Architecture:
   - Uses `timm` library for a p Transformer base model
   - Supports multi-input (d

Vision Transformer Model for Multi-Input Training

```python
import torch
import torch.nn as nn
import timm
import math

class ViTMultiInputModel(nn.Module):
    def __init__(
        self,
        domain_input_channels=3,
        time_input_channels=1,
        num_classes=1,
        embed_dim=768,
        num_heads=12,
        depth=12,
```



Multi-class segmentation



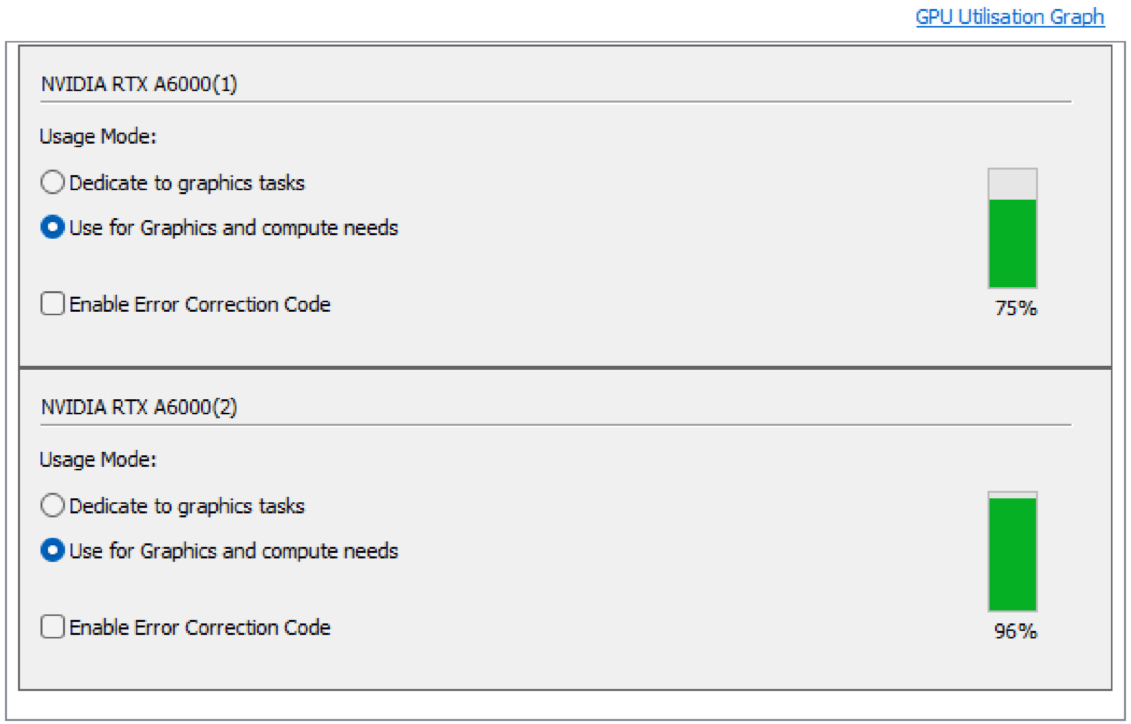
Domain

Invaded Domain

Invasion Boundary

GPU Utilization

Multi-class segmentation



Baseline

Metrics

Loaded model from epoch: 15

Best validation loss: 0.1266841

Average Train MSE: 0.074407

Average Val MSE: 0.087164

Final Time Step Train MSE: 0.155112

Final Time Step Val MSE: 0.178702

Loaded model from epoch: 17

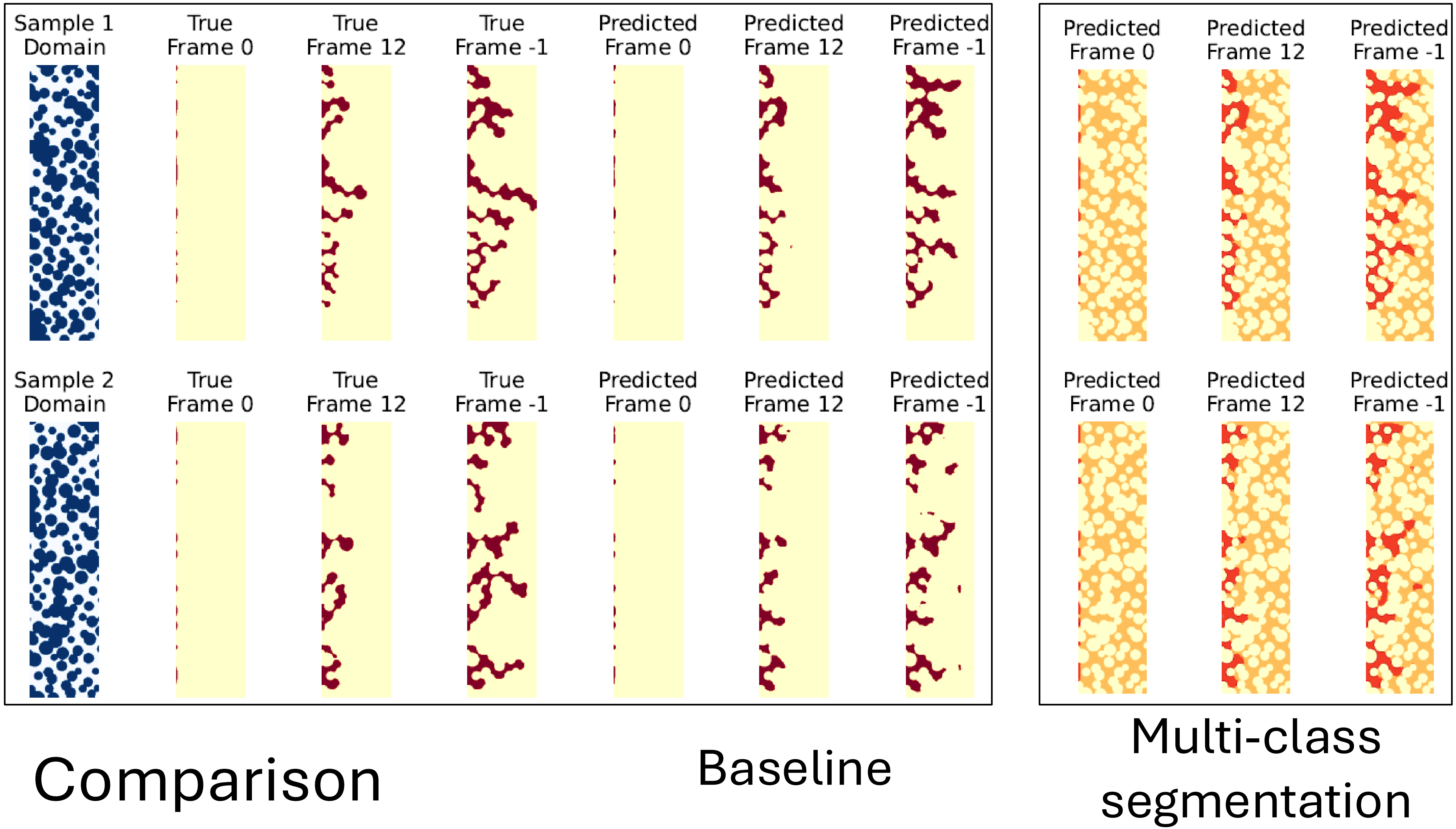
Best validation loss: 0.1993547

Average Train MSE: 0.070234

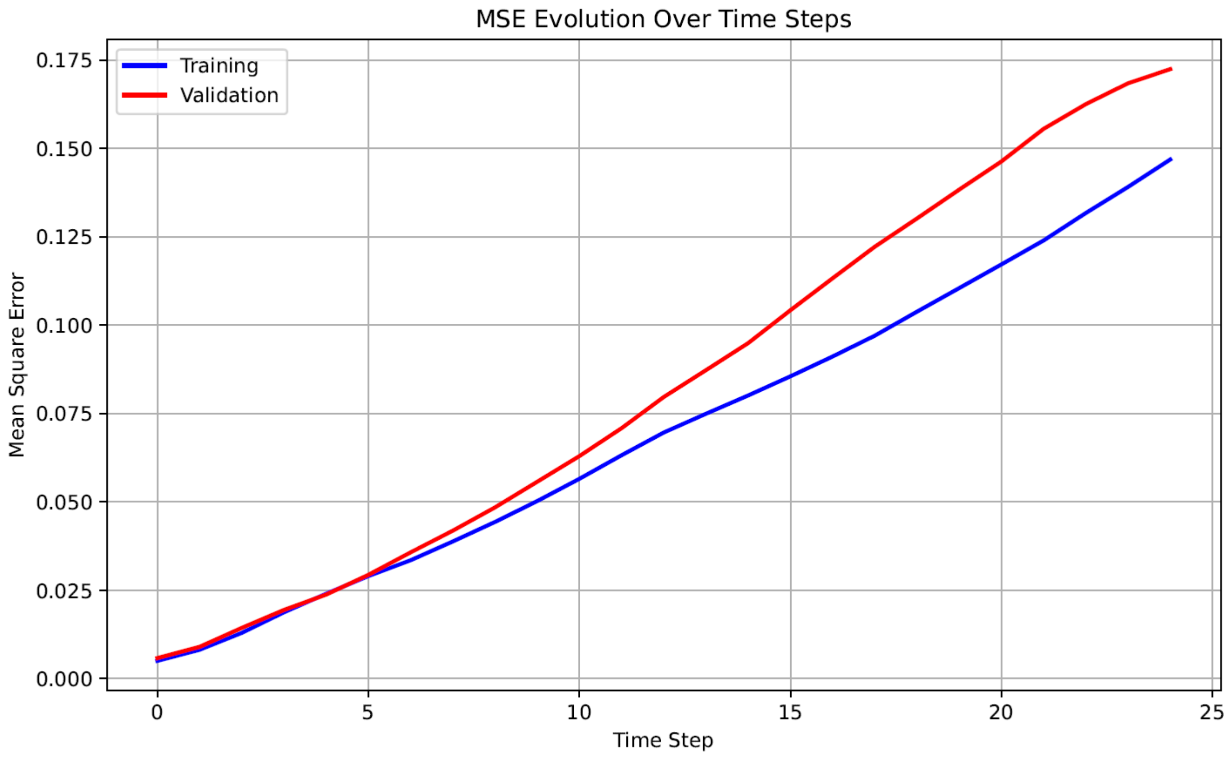
Average Val MSE: 0.083692

Final Time Step Train MSE: 0.146832

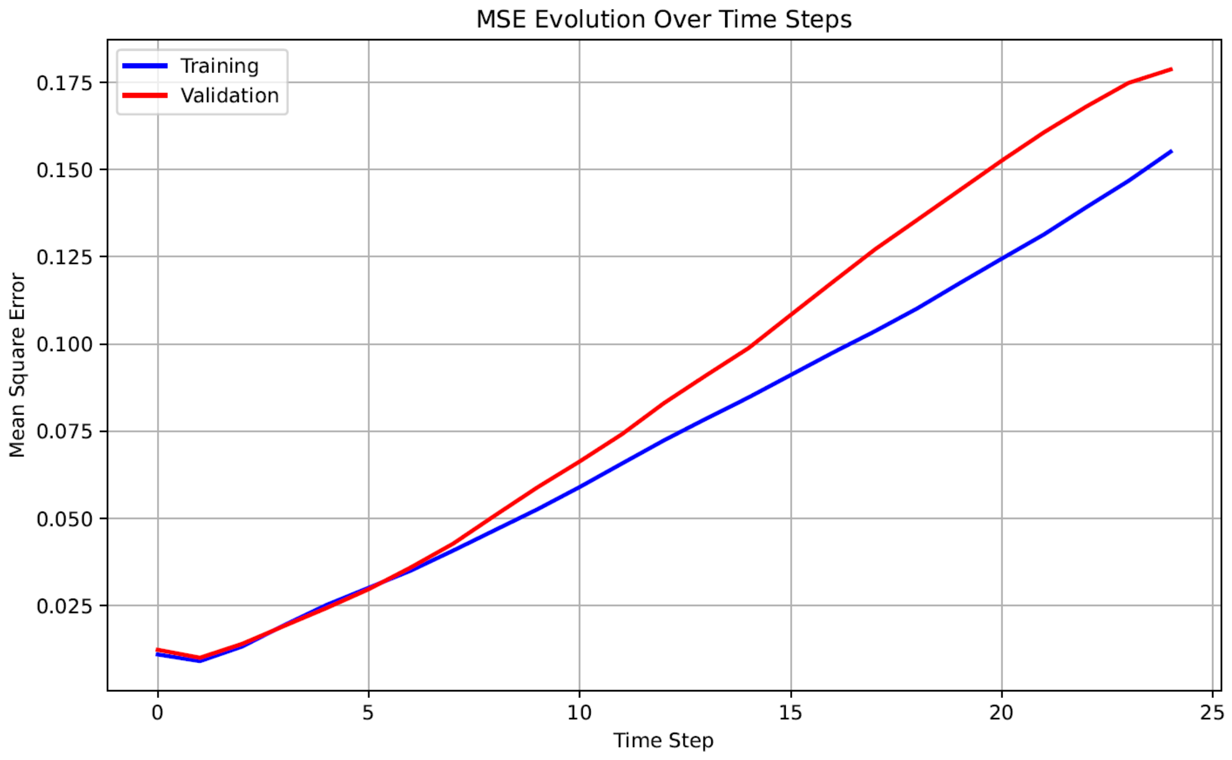
Final Time Step Val MSE: 0.172383


Sample 1 Domain
True Frame 0
True Frame 12
True Frame -1
Predicted Frame 0
Predicted Frame 12
Predicted Frame -1
Predicted Frame 0
Predicted Frame 12
Predicted Frame -1
Sample 2 Domain
True Frame 0
True Frame 12
True Frame -1
Predicted Frame 0
Predicted Frame 12
Predicted Frame -1
Predicted Frame 0
Predicted Frame 12
Predicted Frame -1
Comparison
Baseline
Multi-class segmentation

MSE *over* time



Baseline



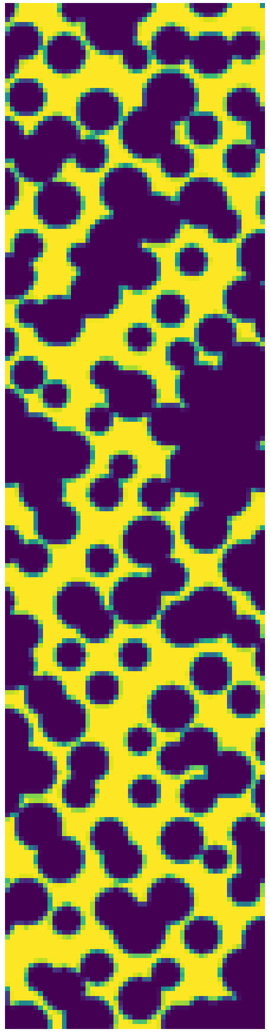
Multi-class segmentation

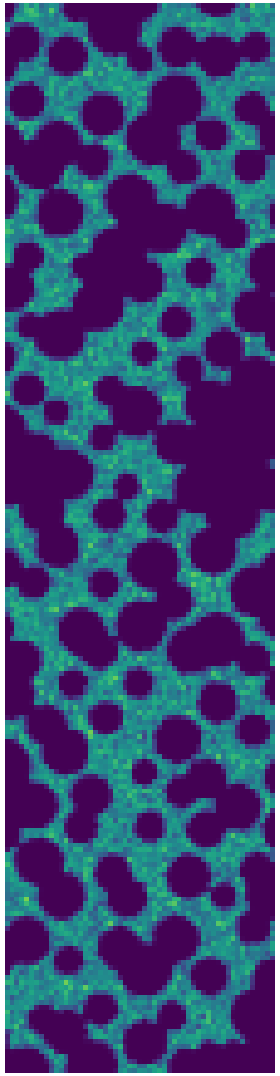
** With and without class weights

Ali

Segmentation problem – further improvements

- Manipulate the input (domain) as well
- Loss function (IoU + Dice loss)

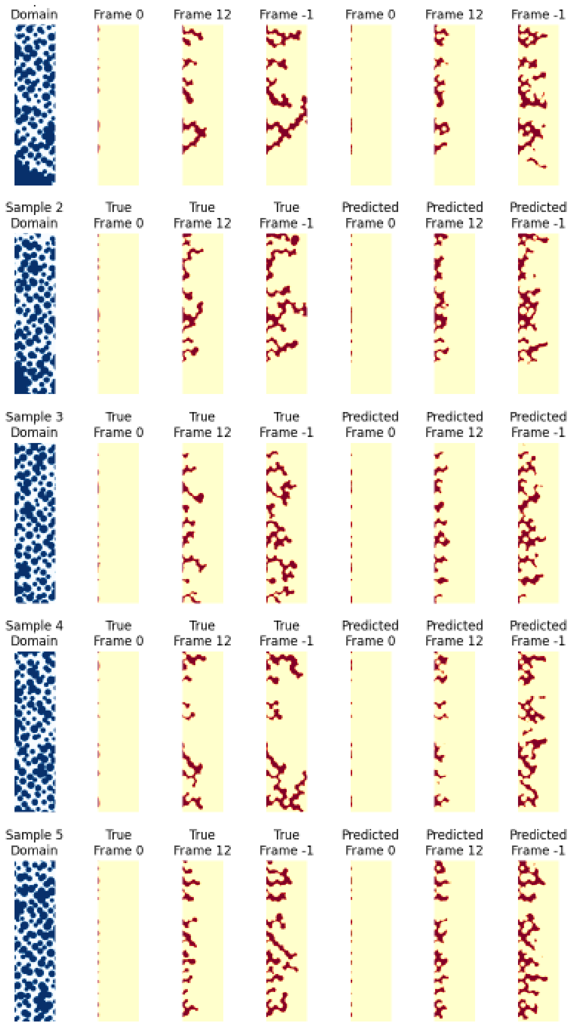




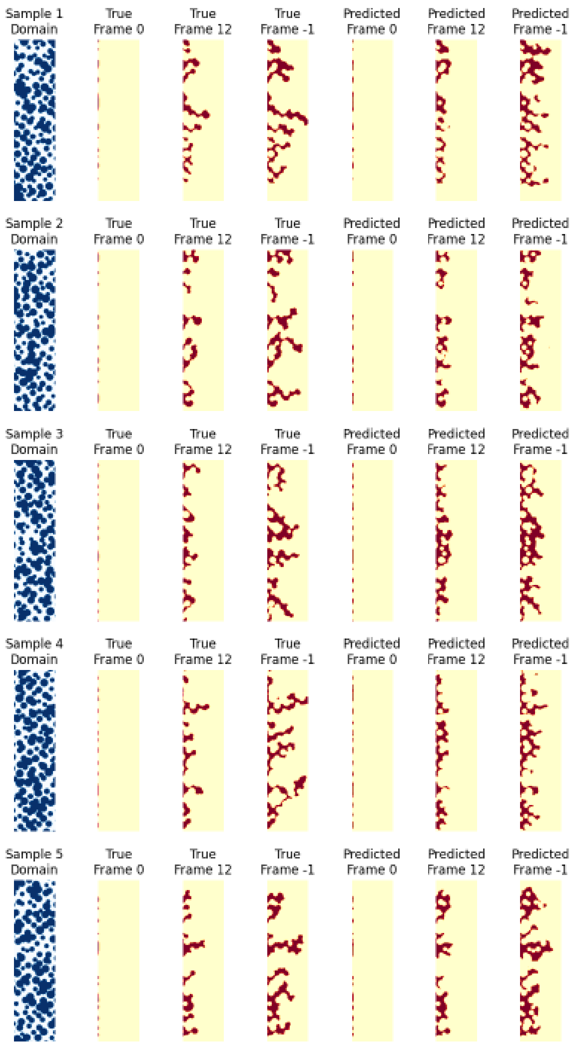
Failure reasons:

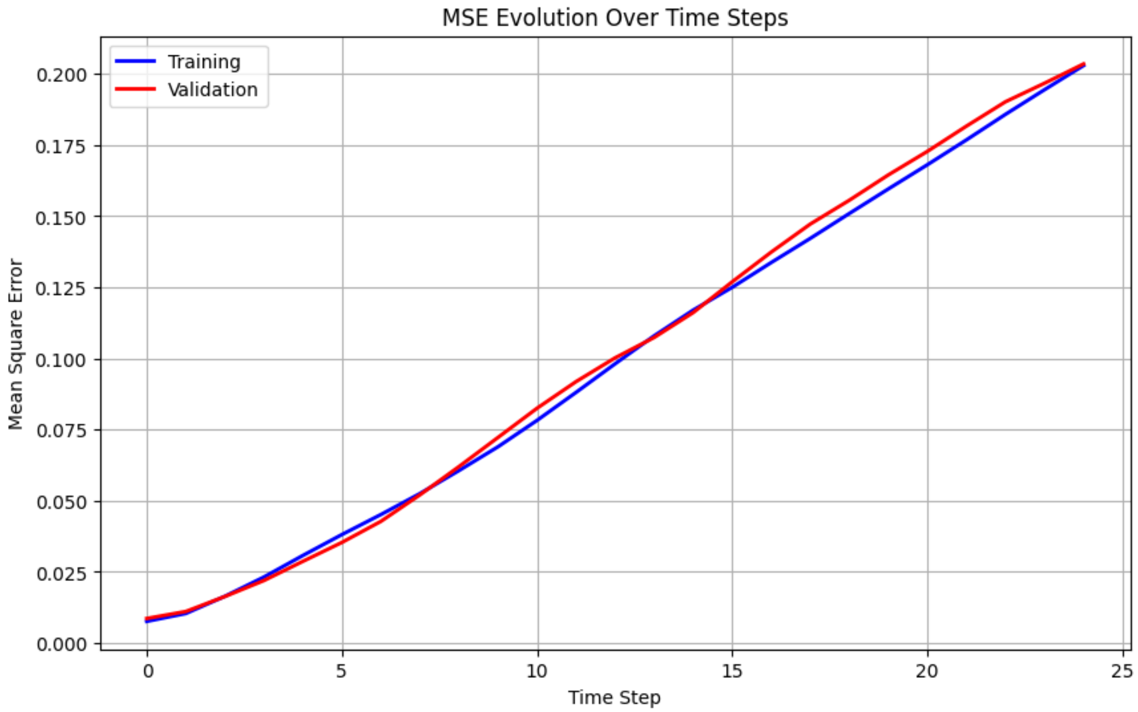
- Fixated on the geometry of the domain
- No improvements in time embedding

Adding Time to the encoder: worse that without time in encoder



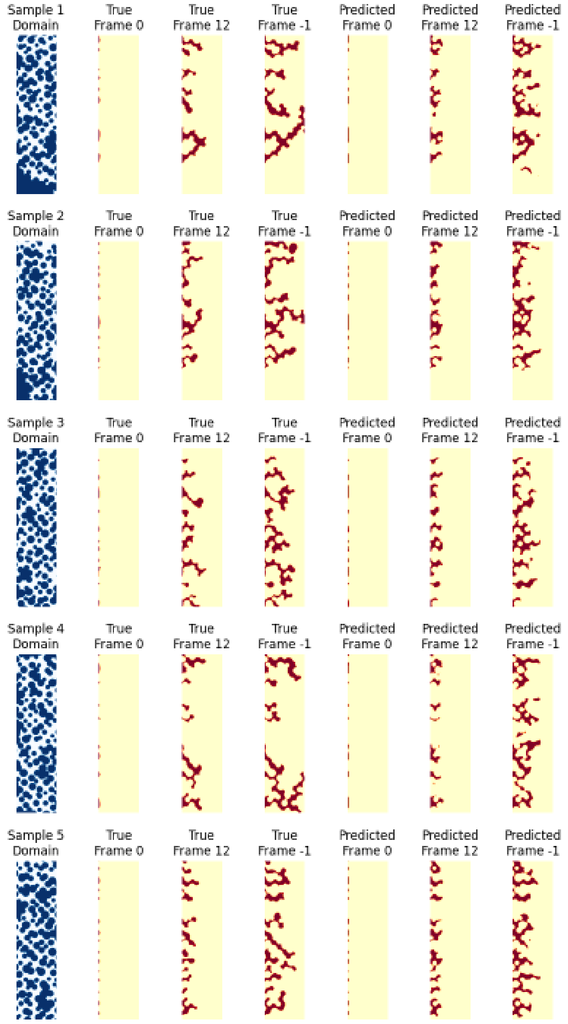
True vs Predicted Invasion Fields: Validation



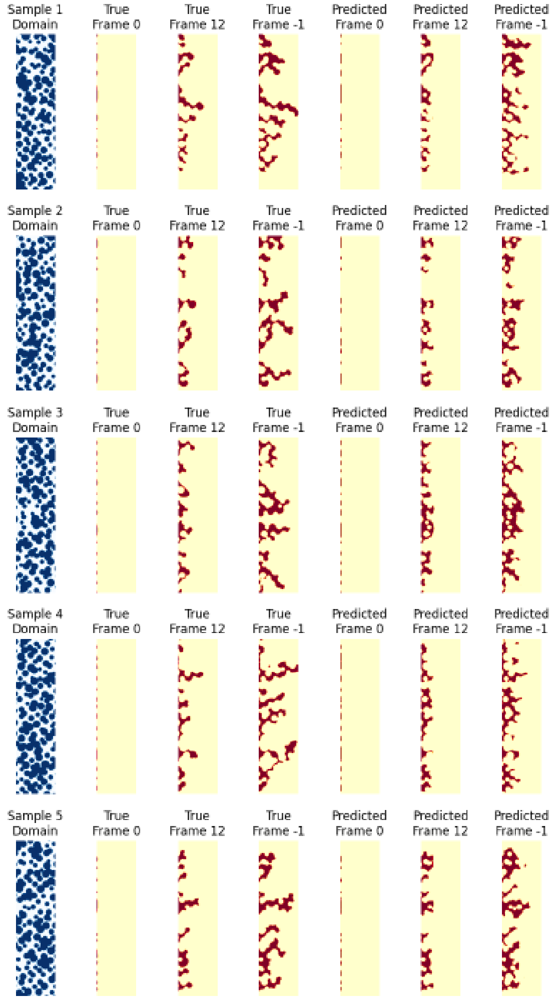


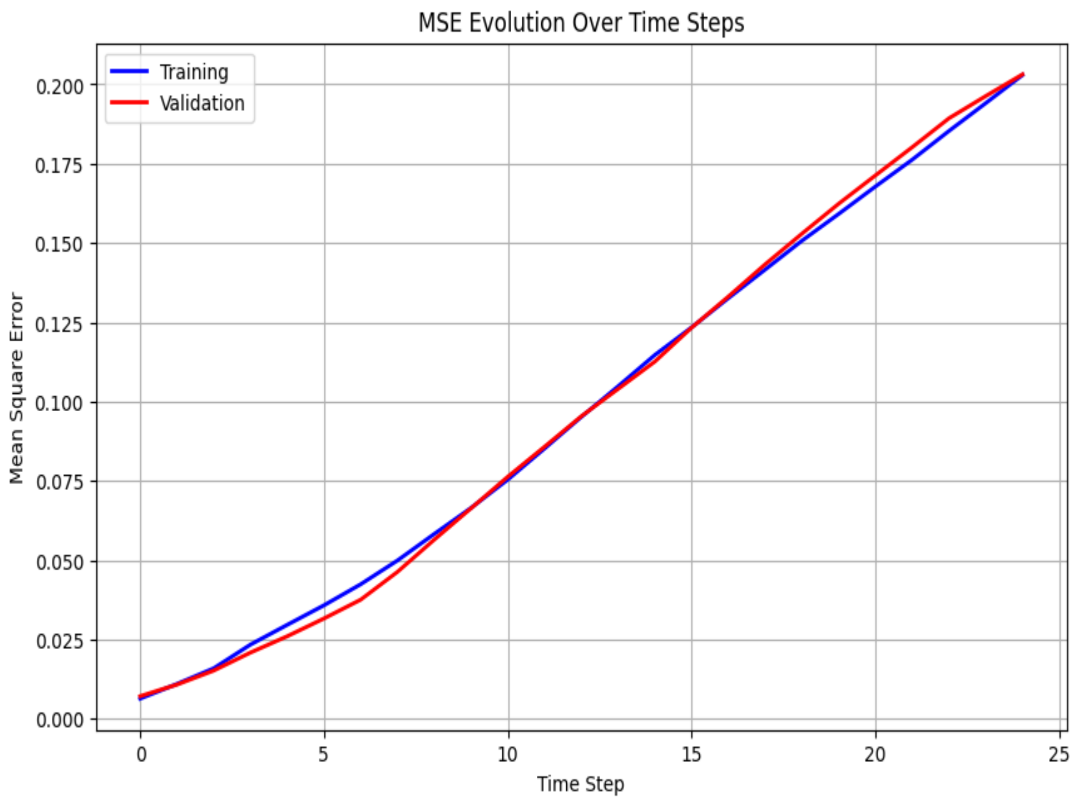

```
Average Train MSE: 0.099329
Average Val MSE: 0.101053
Final Time Step Train MSE: 0.202997
Final Time Step Val MSE: 0.203549
```

Time in encoder + self-attention mechanism in the bottleneck of the U-net: slightly better average MSE than with time in encoder alone.



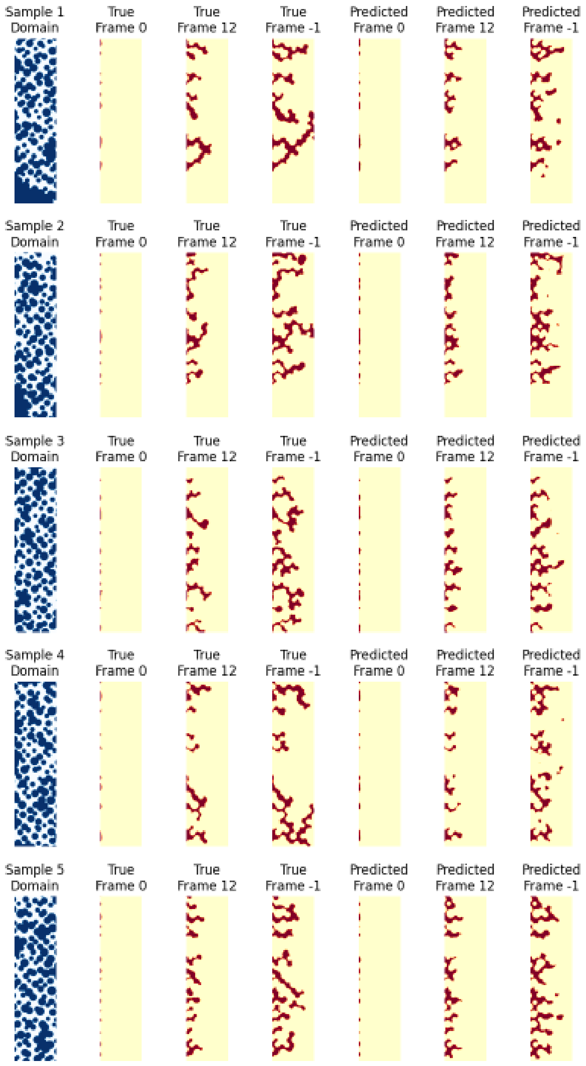
True vs Predicted Invasion Fields: Validation



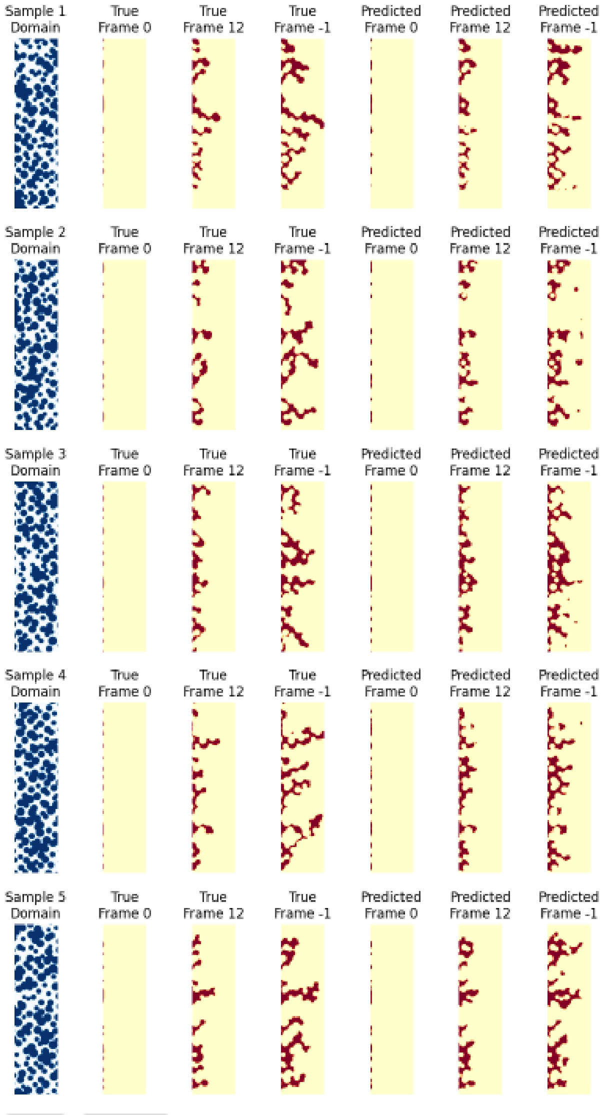


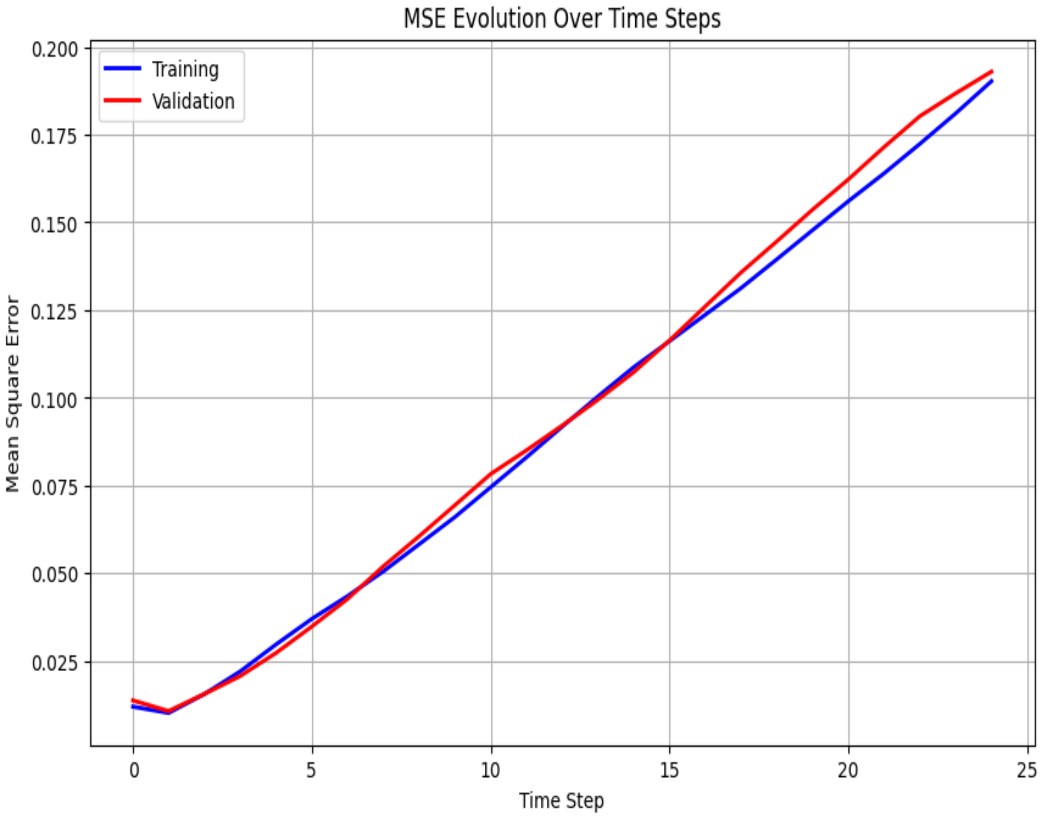

```
Average Train MSE: 0.097925
Average Val MSE: 0.097899
Final Time Step Train MSE: 0.202946
Final Time Step Val MSE: 0.203151
```

Self-attention mechanism in the bottleneck of the U-net without time in encoder: no significant improvement



True vs Predicted Invasion Fields: Validation





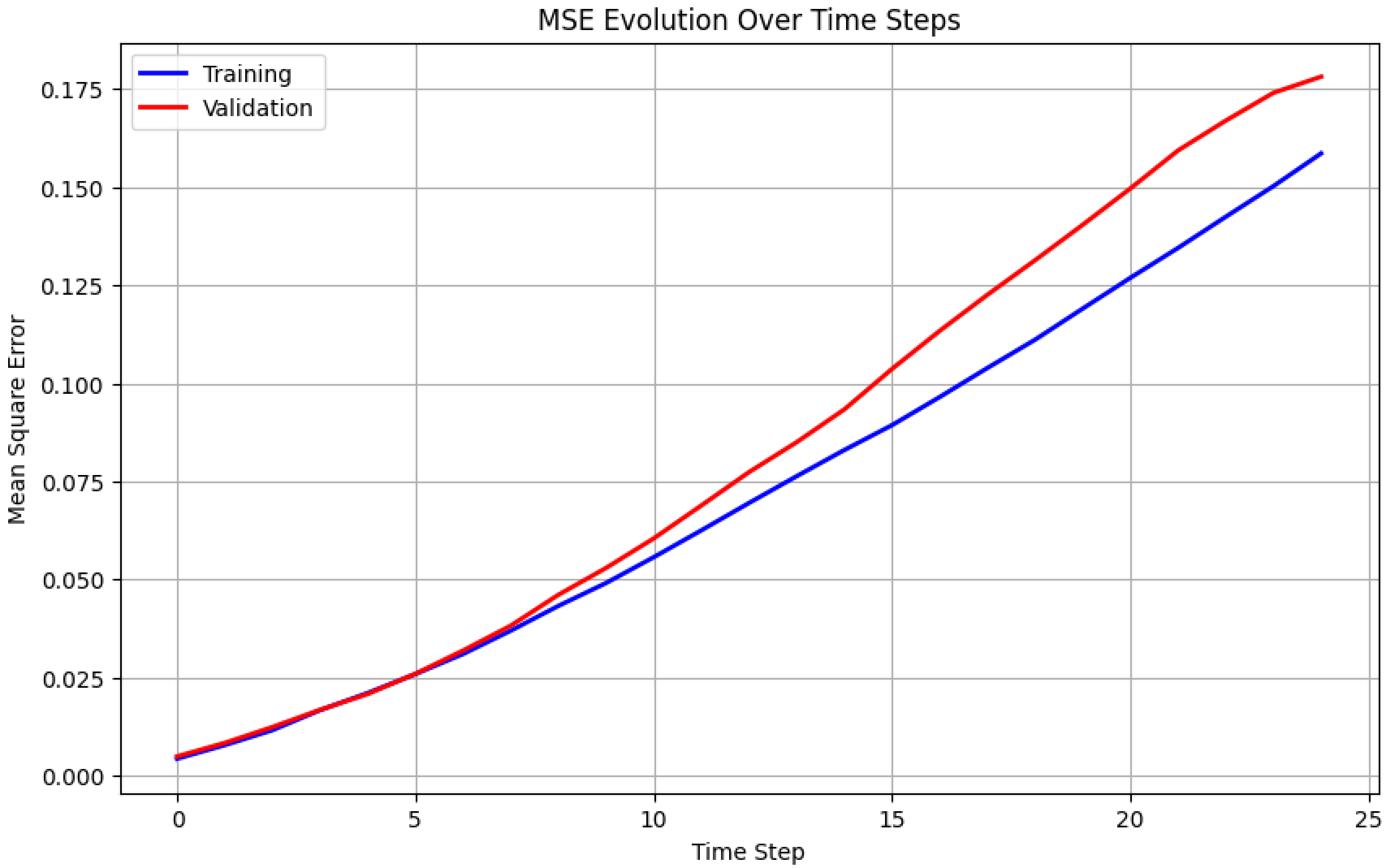
Average Train MSE: 0.093052
Average Val MSE: 0.095228
Final Time Step Train MSE: 0.190260
Final Time Step Val MSE: 0.192968

Loss function Dice loss function

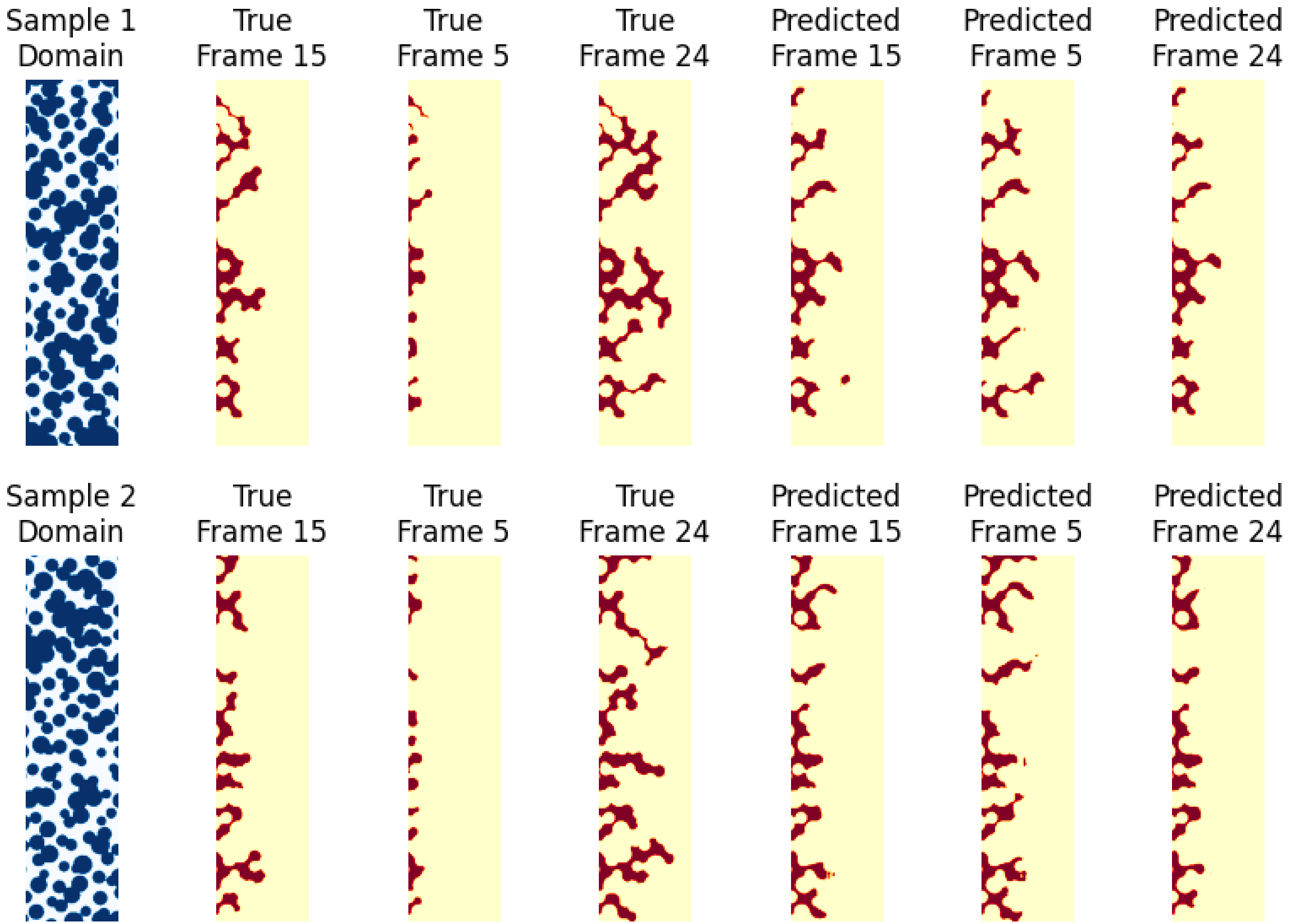
```python
# PART 2: Dice loss
# Get probability predictions for the invasion class
pred_softmax = F.softmax(predictions, dim=1)
pred_probs = pred_softmax[:, 1]  # Take channel 1 (invasion)

# Extract target invasion masks and apply fluid mask
target_masked = targets.squeeze(1) * fluid_mask  # Only consider valid pixels
pred_masked = pred_probs * fluid_mask            # Only consider valid pixels

# Calculate Dice coefficient (smooth factor added to prevent division by zero)
smooth = 1e-6
intersection = (pred_masked * target_masked).sum(dim=(1, 2))
pred_sum = pred_masked.sum(dim=(1, 2))
target_sum = target_masked.sum(dim=(1, 2))

# Dice coefficient formula: 2*intersection / (pred_sum + target_sum)
dice_coeff = (2.0 * intersection + smooth) / (pred_sum + target_sum + smooth)

# Convert to Dice loss (1 - Dice coefficient)
dice_loss = 1.0 - dice_coeff
```

```python
# PART 3: Volume difference loss
# Count positive pixels in ground truth and predictions (within fluid mask)
gt_positive_pixels = gt_positive_pixels = target_masked.sum(dim=(1, 2))

# For predictions, binarize using threshold of 0.5 to count positive pixels
pred_binary = (pred_probs > 0.5).float() * fluid_mask
pred_positive_pixels = pred_binary.sum(dim=(1, 2))

# Calculate absolute difference in volume normalized by total fluid mask pixels
volume_diff = torch.abs(gt_positive_pixels - pred_positive_pixels) / (valid_pixels + 1e-6)

# Combine the two loss components
# You can adjust the weight between the two components
alpha = 1  # Weight for Dice loss
combined_loss = alpha * sample_losses.mean() #+ alpha * dice_loss.mean() + alpha * volume_diff.mean()
```

Loss function Dice loss function



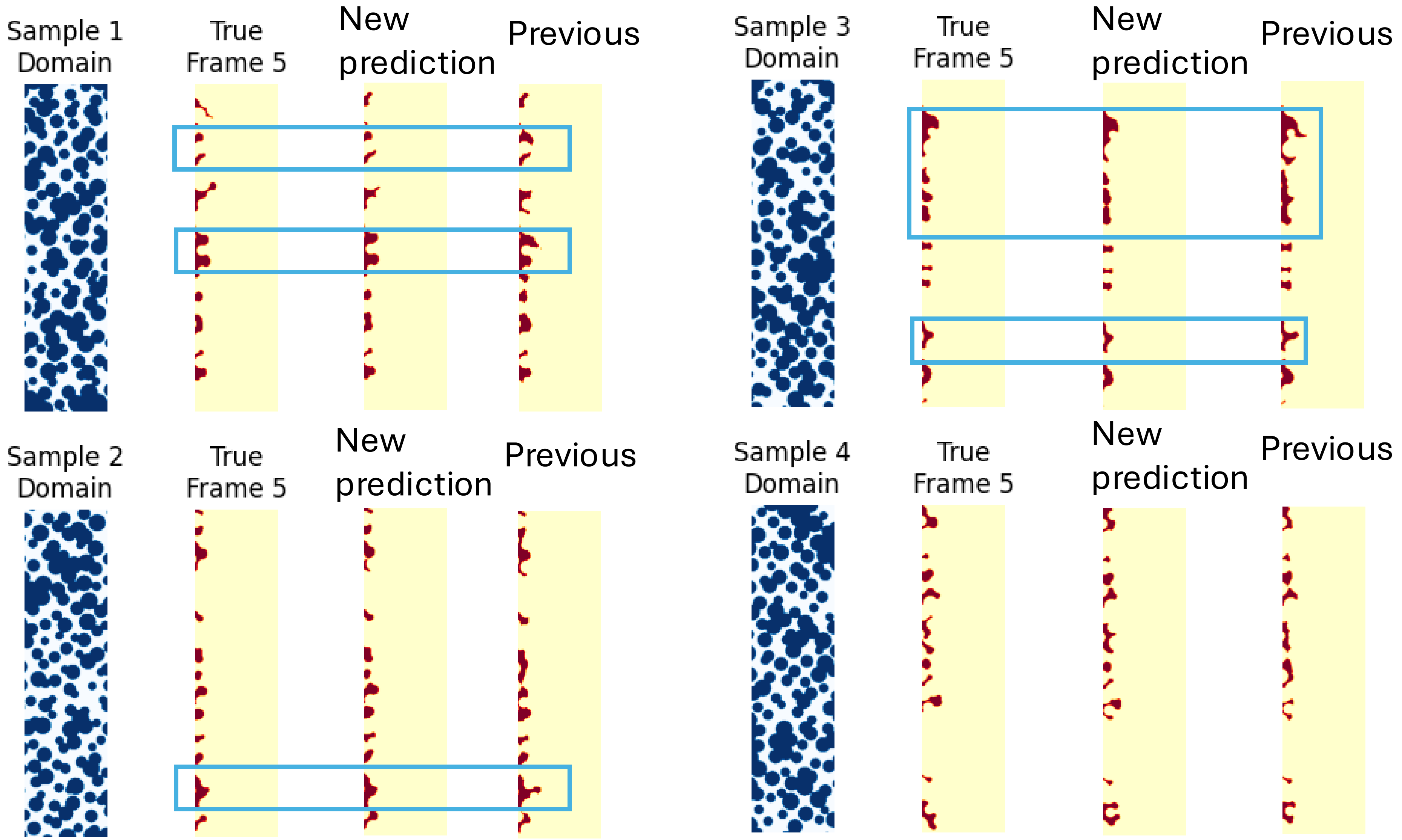

```
Average Train MSE: 0.073092
Average Val MSE: 0.083312
Final Time Step Train MSE: 0.158557
Final Time Step Val MSE: 0.178103
```



Each frame has unique mapping with the input

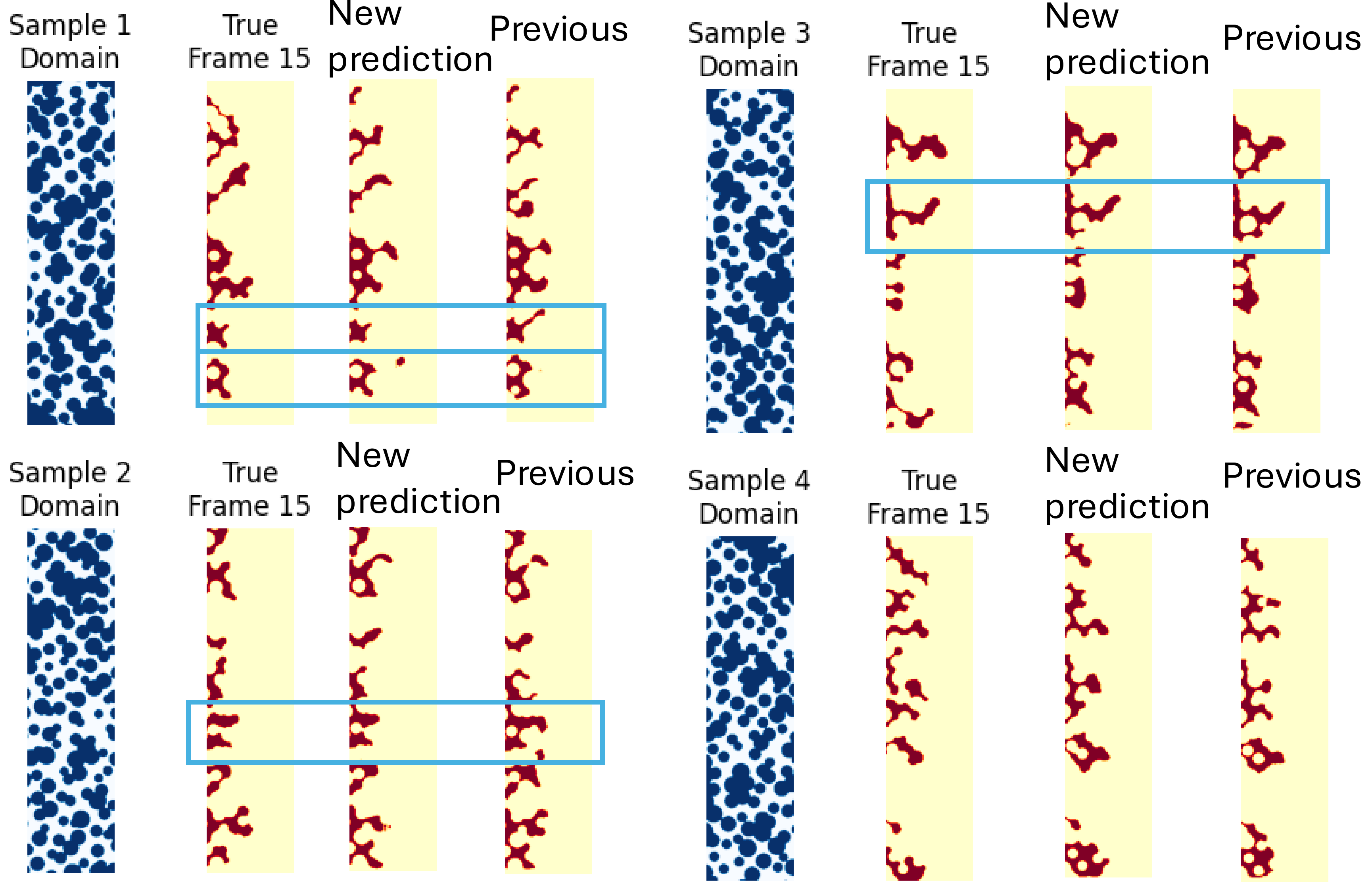
Prediction generated by network which trained with: Dropout=0.15, unique network for frame 15

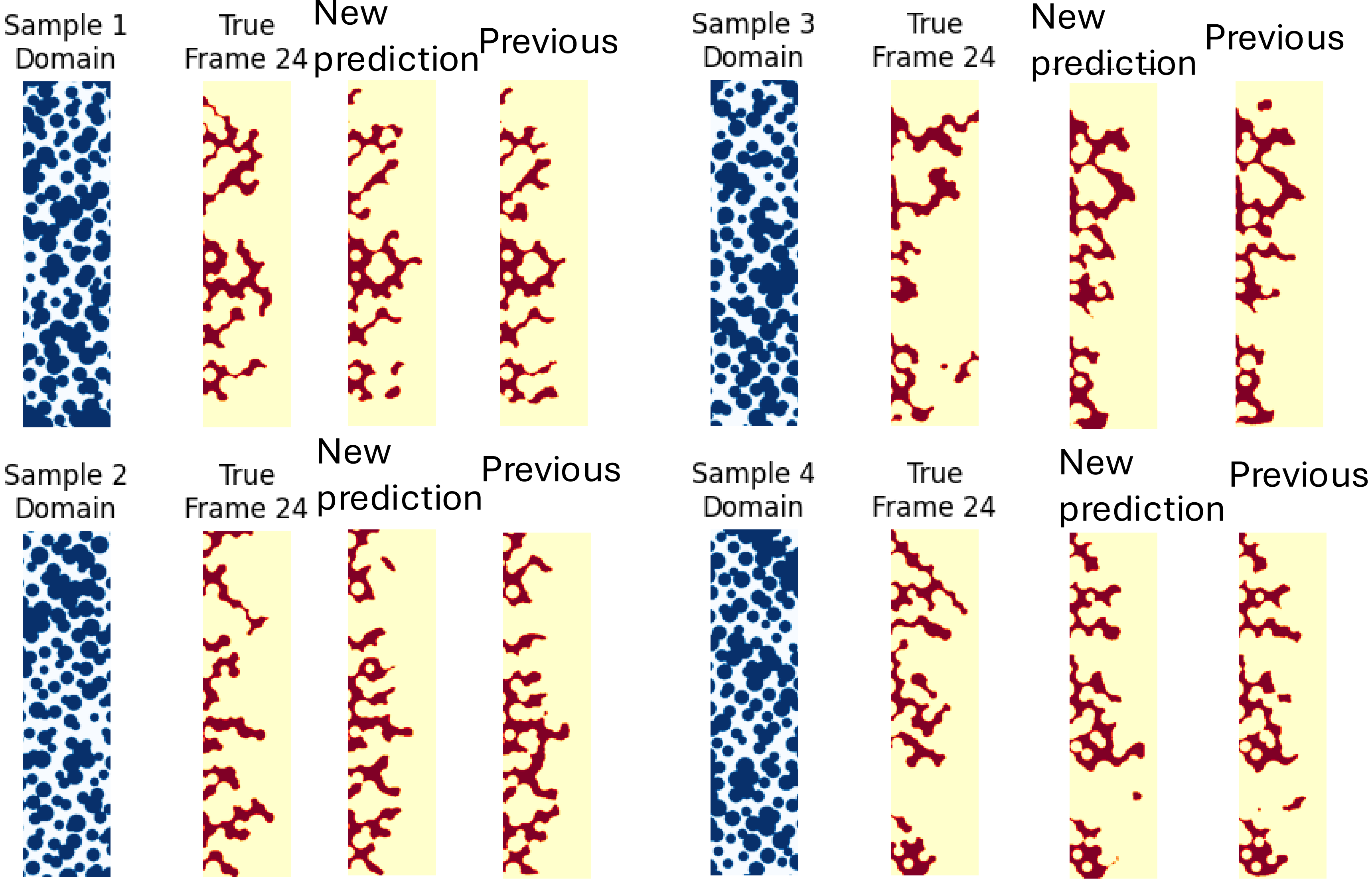
Originnally prediction generated by network which trained with: Dropout=0.30, time embedding network



The middle prediction generated by unique network

The rightside prediction generated by original network


Sample 1
Domain
True
Frame 15
New
prediction
Previous
Sample 3
Domain
True
Frame 15
New
prediction
Previous
Sample 2
Domain
True
Frame 15
New
prediction
Previous
Sample 4
Domain
True
Frame 15
New
prediction
Previous


Sample 1
Domain
True
Frame 24
New
prediction
Previous
Sample 3
Domain
True
Frame 24
New
prediction
Previous
Sample 2
Domain
True
Frame 24
New
prediction
Previous
Sample 4
Domain
True
Frame 24
New
prediction
Previous