A New Deep Learning Method for Solar Flare Prediction

Authors: Yash Chaudhary, Yasser Abduallah, Jason T. L. Wang
Institution: New Jersey Institute of Technology **Date:** August 6, 2025

---

1. Overview

This package contains the code and data for a new deep learning framework for the prediction of solar flares. The model is designed to forecast the probability of a $\geq$M-class solar flare within a 24-hour window using time-series data from NASA's Solar Dynamics Observatory.

This package provides the full model architecture, the training and testing scripts, and the processed data necessary to replicate the model's state-of-the-art performance, which achieves a **True Skill Statistic (TSS) of 0.90.**

---

2. Environment Setup

The following cell imports all necessary libraries. Please ensure they are installed, for example, by using the included `requirements.txt` file (`pip install -r requirements.txt`).

In [2]:
```
# --- Core Libraries ---
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
import numpy as np
import os
from tqdm.notebook import tqdm
from sklearn.metrics import confusion_matrix

print("Libraries imported successfully.")
```

```
Libraries imported successfully.
```

---

3. Model Architecture

The core of our predictor is a new deep learning model based on a transformer architecture. The model is defined in the class below.

Key Architectural Concepts:

- **Patching:** The input time-series (24 hours) is first divided into non-overlapping 'patches' (2 hours each). This allows the model to learn local patterns within each short time window.
- **Embedding:** Each patch is linearly projected into a higher-dimensional space, creating a rich representation for the transformer to process.
- **Transformer Encoder:** A standard Transformer Encoder then learns the long-range dependencies and relationships between these patches.
- **Classification Head:** Finally, the information from all patches is aggregated to produce a single probability for the 24-hour forecast window.

In [3]:
```
class Patching(nn.Module):
    def __init__(self, patch_length):
        super().__init__()
        self.patch_length = patch_length

    def forward(self, x):
        num_patches = x.shape[1] // self.patch_length
        return x.view(x.shape[0], num_patches, self.patch_length, x.shape[2])

class MTST(nn.Module):
    def __init__(self, num_features, time_steps, patch_length, d_model, num_heads, num_encoder_layers, dim_feedforward, dropout=0.1):
        super().__init__()
        assert time_steps % patch_length == 0, "Time steps must be divisible by patch length"
        self.patching = Patching(patch_length)
        num_patches = time_steps // patch_length

        # Correctly calculate input dimension for the linear projection
        patch_input_dim = patch_length * num_features
        self.linear_projection = nn.Linear(patch_input_dim, d_model)

        self.positional_encoding = nn.Parameter(torch.zeros(1, num_patches, d_model))

        encoder_layer = nn.TransformerEncoderLayer(d_model=d_model, nhead=num_heads, dim_feedforward=dim_feedforward, dropout=dropout,
```

```
        self.transformer_encoder = nn.TransformerEncoder(encoder_layer, num_layers=num_encoder_layers)

        self.output_aggregation_pool = nn.AdaptiveAvgPool1d(1)
        self.classification_output_layer = nn.Linear(d_model, 1)

    def forward(self, x):
        x = self.patching(x)
        B, N, P, F = x.shape
        x = x.reshape(B, N, P * F)
        x = self.linear_projection(x)
        x = x + self.positional_encoding
        x = self.transformer_encoder(x)
        x = x.permute(0, 2, 1)
        x = self.output_aggregation_pool(x).squeeze(-1)
        output_logits = self.classification_output_layer(x)
        return output_logits

print("MTST model architecture defined.")
```

```
MTST model architecture defined.
```

---

4. Model Training

This cell executes the full training pipeline. It will:

1. Load the training data from the `data/processed/` directory.
2. Initialize the model, loss function, and optimizer.
3. Run the training loop for 50 epochs.
4. Save the final trained model weights to `models/mtst_flare_model.pth`.

In [4]:
```
# --- Configuration ---
PROCESSED_DATA_DIR = 'data/processed'
MODELS_DIR = 'models'
MODEL_SAVE_PATH = os.path.join(MODELS_DIR, 'mtst_flare_model.pth')
os.makedirs(MODELS_DIR, exist_ok=True)

# --- Hyperparameters ---
NUM_FEATURES = 16
BATCH_SIZE = 64
NUM_EPOCHS = 50
LEARNING_RATE = 0.0001
WEIGHT_DECAY = 2e-5

# --- Data Loading ---
print("Loading training data...")
X_train = np.load(os.path.join(PROCESSED_DATA_DIR, 'X_train.npy'))
y_train = np.load(os.path.join(PROCESSED_DATA_DIR, 'y_train.npy'))
train_dataset = TensorDataset(torch.tensor(X_train, dtype=torch.float32), torch.tensor(y_train, dtype=torch.float32).unsqueeze(1))
train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True)
print("Data loaded.")

# --- Initialization ---
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = MTST(num_features=16, time_steps=24, patch_length=2, d_model=128, num_heads=8, num_encoder_layers=3, dim_feedforward=256, drop
criterion = nn.BCEWithLogitsLoss()
optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE, weight_decay=WEIGHT_DECAY)

# --- Training Loop ---
print(f"\n--- Starting Model Training on {device} ---")
model.train()
for epoch in range(NUM_EPOCHS):
    total_loss = 0
    for features, labels in tqdm(train_loader, desc=f"Epoch {epoch+1}/{NUM_EPOCHS}"):
        features, labels = features.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(features)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    print(f"Epoch [{epoch+1}/{NUM_EPOCHS}], Average Loss: {total_loss / len(train_loader):.4f}")

# --- Save Model ---
torch.save(model.state_dict(), MODEL_SAVE_PATH)
print(f"\n--- Training complete! Model saved to {MODEL_SAVE_PATH} ---")
```

```
Loading training data...
Data loaded.

--- Starting Model Training on cuda ---
Epoch 1/50:   0%|          | 0/1057 [00:00<?, ?it/s]
Epoch [1/50], Average Loss: 0.0967
Epoch 2/50:   0%|          | 0/1057 [00:00<?, ?it/s]
Epoch [2/50], Average Loss: 0.0784
Epoch 3/50:   0%|          | 0/1057 [00:00<?, ?it/s]
Epoch [3/50], Average Loss: 0.0703
Epoch 4/50:   0%|          | 0/1057 [00:00<?, ?it/s]
Epoch [4/50], Average Loss: 0.0615
Epoch 5/50:   0%|          | 0/1057 [00:00<?, ?it/s]
Epoch [5/50], Average Loss: 0.0541
Epoch 6/50:   0%|          | 0/1057 [00:00<?, ?it/s]
Epoch [6/50], Average Loss: 0.0471
Epoch 7/50:   0%|          | 0/1057 [00:00<?, ?it/s]
Epoch [7/50], Average Loss: 0.0421
Epoch 8/50:   0%|          | 0/1057 [00:00<?, ?it/s]
Epoch [8/50], Average Loss: 0.0369
Epoch 9/50:   0%|          | 0/1057 [00:00<?, ?it/s]
Epoch [9/50], Average Loss: 0.0331
Epoch 10/50:   0%|          | 0/1057 [00:00<?, ?it/s]
Epoch [10/50], Average Loss: 0.0299
Epoch 11/50:   0%|          | 0/1057 [00:00<?, ?it/s]
Epoch [11/50], Average Loss: 0.0281
Epoch 12/50:   0%|          | 0/1057 [00:00<?, ?it/s]
Epoch [12/50], Average Loss: 0.0253
Epoch 13/50:   0%|          | 0/1057 [00:00<?, ?it/s]
Epoch [13/50], Average Loss: 0.0240
Epoch 14/50:   0%|          | 0/1057 [00:00<?, ?it/s]
Epoch [14/50], Average Loss: 0.0238
Epoch 15/50:   0%|          | 0/1057 [00:00<?, ?it/s]
Epoch [15/50], Average Loss: 0.0229
Epoch 16/50:   0%|          | 0/1057 [00:00<?, ?it/s]
Epoch [16/50], Average Loss: 0.0224
Epoch 17/50:   0%|          | 0/1057 [00:00<?, ?it/s]
Epoch [17/50], Average Loss: 0.0200
Epoch 18/50:   0%|          | 0/1057 [00:00<?, ?it/s]
Epoch [18/50], Average Loss: 0.0199
Epoch 19/50:   0%|          | 0/1057 [00:00<?, ?it/s]
Epoch [19/50], Average Loss: 0.0205
Epoch 20/50:   0%|          | 0/1057 [00:00<?, ?it/s]
Epoch [20/50], Average Loss: 0.0193
Epoch 21/50:   0%|          | 0/1057 [00:00<?, ?it/s]
Epoch [21/50], Average Loss: 0.0192
Epoch 22/50:   0%|          | 0/1057 [00:00<?, ?it/s]
Epoch [22/50], Average Loss: 0.0182
Epoch 23/50:   0%|          | 0/1057 [00:00<?, ?it/s]
Epoch [23/50], Average Loss: 0.0183
Epoch 24/50:   0%|          | 0/1057 [00:00<?, ?it/s]
Epoch [24/50], Average Loss: 0.0171
Epoch 25/50:   0%|          | 0/1057 [00:00<?, ?it/s]
Epoch [25/50], Average Loss: 0.0166
Epoch 26/50:   0%|          | 0/1057 [00:00<?, ?it/s]
Epoch [26/50], Average Loss: 0.0163
Epoch 27/50:   0%|          | 0/1057 [00:00<?, ?it/s]
Epoch [27/50], Average Loss: 0.0162
Epoch 28/50:   0%|          | 0/1057 [00:00<?, ?it/s]
Epoch [28/50], Average Loss: 0.0154
Epoch 29/50:   0%|          | 0/1057 [00:00<?, ?it/s]
Epoch [29/50], Average Loss: 0.0160
Epoch 30/50:   0%|          | 0/1057 [00:00<?, ?it/s]
Epoch [30/50], Average Loss: 0.0154
Epoch 31/50:   0%|          | 0/1057 [00:00<?, ?it/s]
Epoch [31/50], Average Loss: 0.0142
Epoch 32/50:   0%|          | 0/1057 [00:00<?, ?it/s]
Epoch [32/50], Average Loss: 0.0149
Epoch 33/50:   0%|          | 0/1057 [00:00<?, ?it/s]
Epoch [33/50], Average Loss: 0.0145
Epoch 34/50:   0%|          | 0/1057 [00:00<?, ?it/s]
Epoch [34/50], Average Loss: 0.0143
Epoch 35/50:   0%|          | 0/1057 [00:00<?, ?it/s]
Epoch [35/50], Average Loss: 0.0137
Epoch 36/50:   0%|          | 0/1057 [00:00<?, ?it/s]
Epoch [36/50], Average Loss: 0.0140
Epoch 37/50:   0%|          | 0/1057 [00:00<?, ?it/s]
Epoch [37/50], Average Loss: 0.0136
Epoch 38/50:   0%|          | 0/1057 [00:00<?, ?it/s]
Epoch [38/50], Average Loss: 0.0126
Epoch 39/50:   0%|          | 0/1057 [00:00<?, ?it/s]
Epoch [39/50], Average Loss: 0.0128
Epoch 40/50:   0%|          | 0/1057 [00:00<?, ?it/s]
```

```
Epoch [40/50], Average Loss: 0.0127
Epoch 41/50:   0%|          | 0/1057 [00:00<?, ?it/s]
Epoch [41/50], Average Loss: 0.0135
Epoch 42/50:   0%|          | 0/1057 [00:00<?, ?it/s]
Epoch [42/50], Average Loss: 0.0130
Epoch 43/50:   0%|          | 0/1057 [00:00<?, ?it/s]
Epoch [43/50], Average Loss: 0.0123
Epoch 44/50:   0%|          | 0/1057 [00:00<?, ?it/s]
Epoch [44/50], Average Loss: 0.0123
Epoch 45/50:   0%|          | 0/1057 [00:00<?, ?it/s]
Epoch [45/50], Average Loss: 0.0120
Epoch 46/50:   0%|          | 0/1057 [00:00<?, ?it/s]
Epoch [46/50], Average Loss: 0.0121
Epoch 47/50:   0%|          | 0/1057 [00:00<?, ?it/s]
Epoch [47/50], Average Loss: 0.0110
Epoch 48/50:   0%|          | 0/1057 [00:00<?, ?it/s]
Epoch [48/50], Average Loss: 0.0118
Epoch 49/50:   0%|          | 0/1057 [00:00<?, ?it/s]
Epoch [49/50], Average Loss: 0.0118
Epoch 50/50:   0%|          | 0/1057 [00:00<?, ?it/s]
Epoch [50/50], Average Loss: 0.0116

--- Training complete! Model saved to models/mtst_flare_model.pth ---
```

---

5. Model Evaluation

Finally, this cell loads the trained model from the previous step and evaluates its performance on the held-out test set. It calculates and prints the final Confusion Matrix and True Skill Statistic (TSS).

In [5]:
```python
# --- Data Loading ---
print("Loading test data...")
X_test = np.load(os.path.join(PROCESSED_DATA_DIR, 'X_test.npy'))
y_test = np.load(os.path.join(PROCESSED_DATA_DIR, 'y_test.npy'))

# --- Load Trained Model ---
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
eval_model = MTST(num_features=16, time_steps=24, patch_length=2, d_model=128, num_heads=8, num_encoder_layers=3, dim_feedforward=256,
eval_model.load_state_dict(torch.load(MODEL_SAVE_PATH, map_location=device))
eval_model.to(device)
eval_model.eval()
print(f"Model loaded and ready for evaluation.")

# --- Inference ---
print("Generating predictions...")
with torch.no_grad():
    logits = eval_model(torch.tensor(X_test, dtype=torch.float32).to(device))
    predictions = (torch.sigmoid(logits).squeeze().cpu().numpy() >= 0.5).astype(int)

# --- Metrics ---
tn, fp, fn, tp = confusion_matrix(y_test, predictions).ravel()
sensitivity = tp / (tp + fn) if (tp + fn) > 0 else 0.0
specificity = tn / (tn + fp) if (tn + fp) > 0 else 0.0
tss = sensitivity + specificity - 1

print("\n--- Final Model Performance Metrics ---")
print(f"Confusion Matrix (TN, FP, FN, TP): ({tn}, {fp}, {fn}, {tp})")
print(f"True Skill Statistic (TSS): {tss:.4f}")
```

```
Loading test data...
Model loaded and ready for evaluation.
Generating predictions...

--- Final Model Performance Metrics ---
Confusion Matrix (TN, FP, FN, TP): (16313, 56, 49, 493)
True Skill Statistic (TSS): 0.9062
```

The model was trained and validated on a high-performance computing cluster. The primary metric for validation is the True Skill Statistic (TSS), as it is well-suited for the imbalanced classification task of flare prediction. The trained model achieves a TSS of 0.9062.

How to Replicate the Results

1. **Install Dependencies:** `pip install -r requirements.txt`
2. **Train the model:** `python train.py` (This will create `models/mtst_flare_model.pth`)
3. **Evaluate the model:** `python test.py` (This will load the saved model and print the final TSS score.)