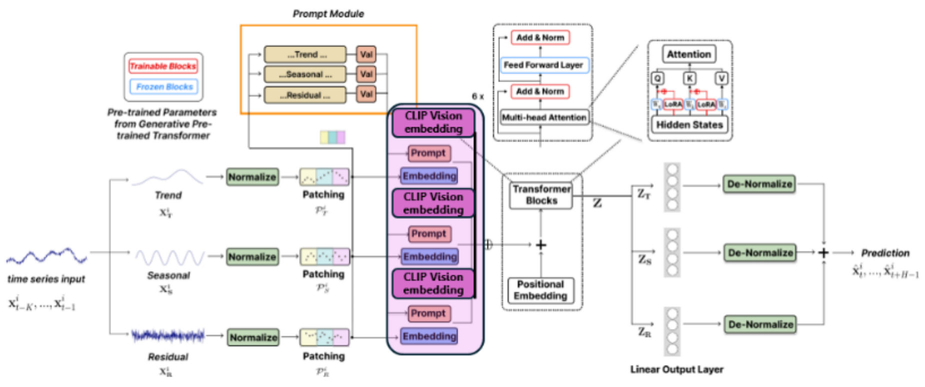
VisionaryTimes - Summary of the main progress in the BGU hackathon 17/01/25

The main goal of the hackathon day was to successfully integrate vision into the tempo model.

Add vision to TEMPO Architecture

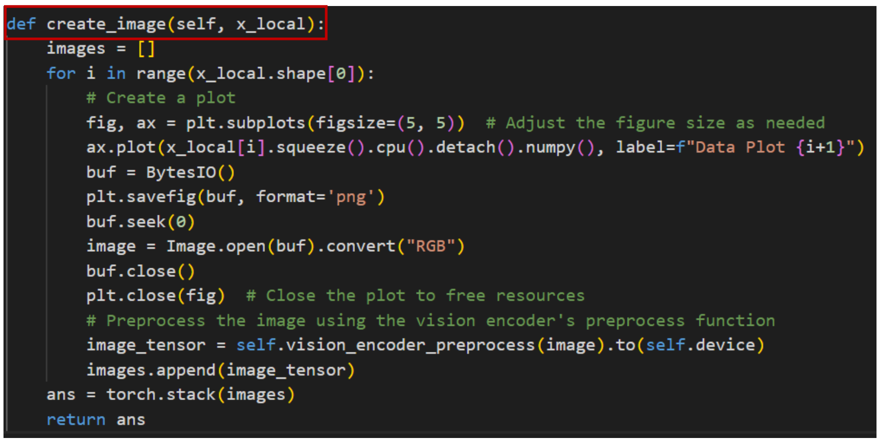


As part of the hackathon, we performed the following steps:

1. We understood the existing embedding creation process, in order to understand where to integrate the vision extension.
2. We understood how the clip model can be used to generate images. And we performed experiments to run this model.
3. Implemented key adjustments, including modifying the TEMPO class (__init__, create_image, vision_embed) and updating the forward function to incorporate vision embeddings.
4. We trained the model with vision adjustments.

Code adjustments explanations:

create_image function takes a sequence or series of values as input, creates visual representations (plots) of that data, processes these plots into image tensors using a vision encoder preprocessing step, and finally returns a stacked tensor containing these pre-processed image representations.



```python
def create_image(self, x_local):
    images = []
    for i in range(x_local.shape[0]):
        # Create a plot
        fig, ax = plt.subplots(figsize=(5, 5))  # Adjust the figure size as needed
        ax.plot(x_local[i].squeeze().cpu().detach().numpy(), label=f"Data Plot {i+1}")
        buf = BytesIO()
        plt.savefig(buf, format='png')
        buf.seek(0)
        image = Image.open(buf).convert("RGB")
        buf.close()
        plt.close(fig)  # Close the plot to free resources
        # Preprocess the image using the vision encoder's preprocess function
        image_tensor = self.vision_encoder_preprocess(image).to(self.device)
        images.append(image_tensor)
    ans = torch.stack(images)
    return ans
```

This function gets x_local which is the TS data patched.

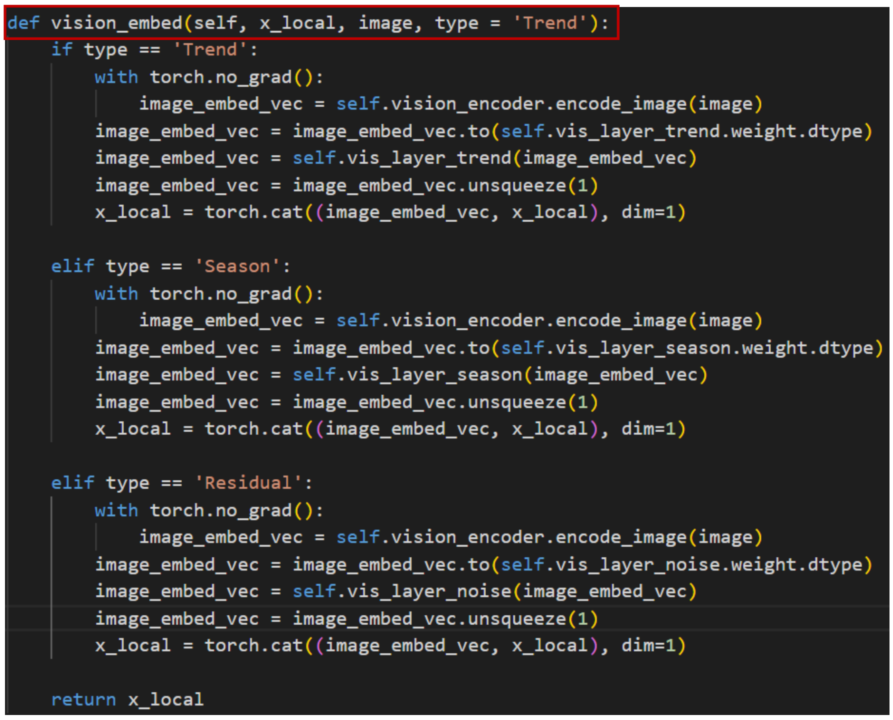
Loops over each instance (row) in x_local. x_local.shape[0] indicates the batch size or the number of data sequences to process.

Creating the plot image.

Passes the image through a preprocessing function specific to a vision encoder (e.g., CLIP), converting the image into a tensor.

Stacks all the image tensors in the images list along a new dimension, creating a single tensor that combines all processed images. Returns the stacked tensor

vision_embed function processes visual data (image) using a vision encoder to generate embeddings. Based on the type of parameter, it applies different transformations to these embeddings (e.g., "Trend," "Season," or "Residual"). Finally, the processed embeddings are concatenated with an input tensor, x_local, to incorporate visual features into the existing data.



```python
def vision_embed(self, x_local, image, type = 'Trend'):
    if type == 'Trend':
        with torch.no_grad():
            image_embed_vec = self.vision_encoder.encode_image(image)
        image_embed_vec = image_embed_vec.to(self.vis_layer_trend.weight.dtype)
        image_embed_vec = self.vis_layer_trend(image_embed_vec)
        image_embed_vec = image_embed_vec.unsqueeze(1)
        x_local = torch.cat((image_embed_vec, x_local), dim=1)

    elif type == 'Season':
        with torch.no_grad():
            image_embed_vec = self.vision_encoder.encode_image(image)
        image_embed_vec = image_embed_vec.to(self.vis_layer_season.weight.dtype)
        image_embed_vec = self.vis_layer_season(image_embed_vec)
        image_embed_vec = image_embed_vec.unsqueeze(1)
        x_local = torch.cat((image_embed_vec, x_local), dim=1)

    elif type == 'Residual':
        with torch.no_grad():
            image_embed_vec = self.vision_encoder.encode_image(image)
        image_embed_vec = image_embed_vec.to(self.vis_layer_noise.weight.dtype)
        image_embed_vec = self.vis_layer_noise(image_embed_vec)
        image_embed_vec = image_embed_vec.unsqueeze(1)
        x_local = torch.cat((image_embed_vec, x_local), dim=1)

    return x_local
```

x_local: A tensor that contains embeddings (of the patch and prompt) which will be updated with the visual embeddings.
image: The input visual data (image) to be encoded.
type: 'Trend', 'Season', or 'Residual'.

torch.no_grad() disables gradient computation, optimizing the process as this is likely for inference only.

Create the embedding - The vision encoder (self.vision_encoder) is used to encode the input image, producing an embedding vec.

Ensures the embedding vector (image_embed_vec) matches the data type of the weights in vis_layer_trend (dimension d).
unsqueeze(1) - Adds a singleton dimension to the embedding vector to make it compatible for concatenation with x_local. This likely converts the embedding vector to the shape [batch_size, 1, dim].

Concatenates the processed embedding vector of the vision with x_local (patch and prompt embd)

adjust the class TEMPO __init__ :

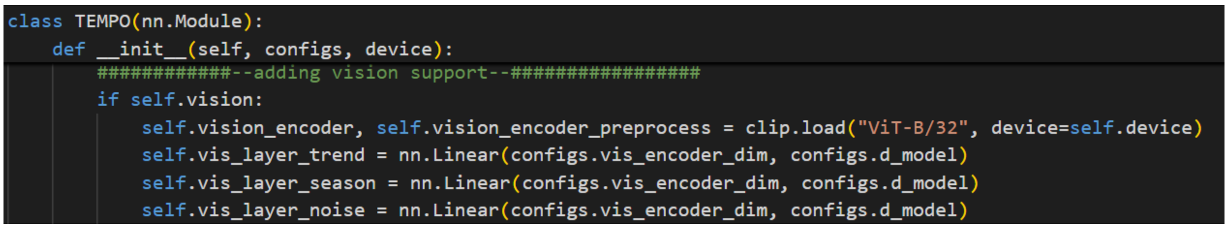
if self.vision – This block of code is executed only if vision support is required. Vision support allows the model to process and incorporate visual data (e.g., images) alongside time series data.

self.vision_encoder - Defines the CLIP model configurations. clip.load("ViT-B/32"): Loads a Vision Transformer (ViT) model that processes images, specifically the ViT-B/32 variant.

Three separate linear layers are defined for mapping vision features to the model's required dimensions:

- configs.vis_encoder_dim: The dimensionality of the features produced by the vision encoder (output of CLIP).
- configs.d_model: The dimensionality of the transformer model's internal representations.

These layers ensure that visual data features can be integrated seamlessly with the time series data components.

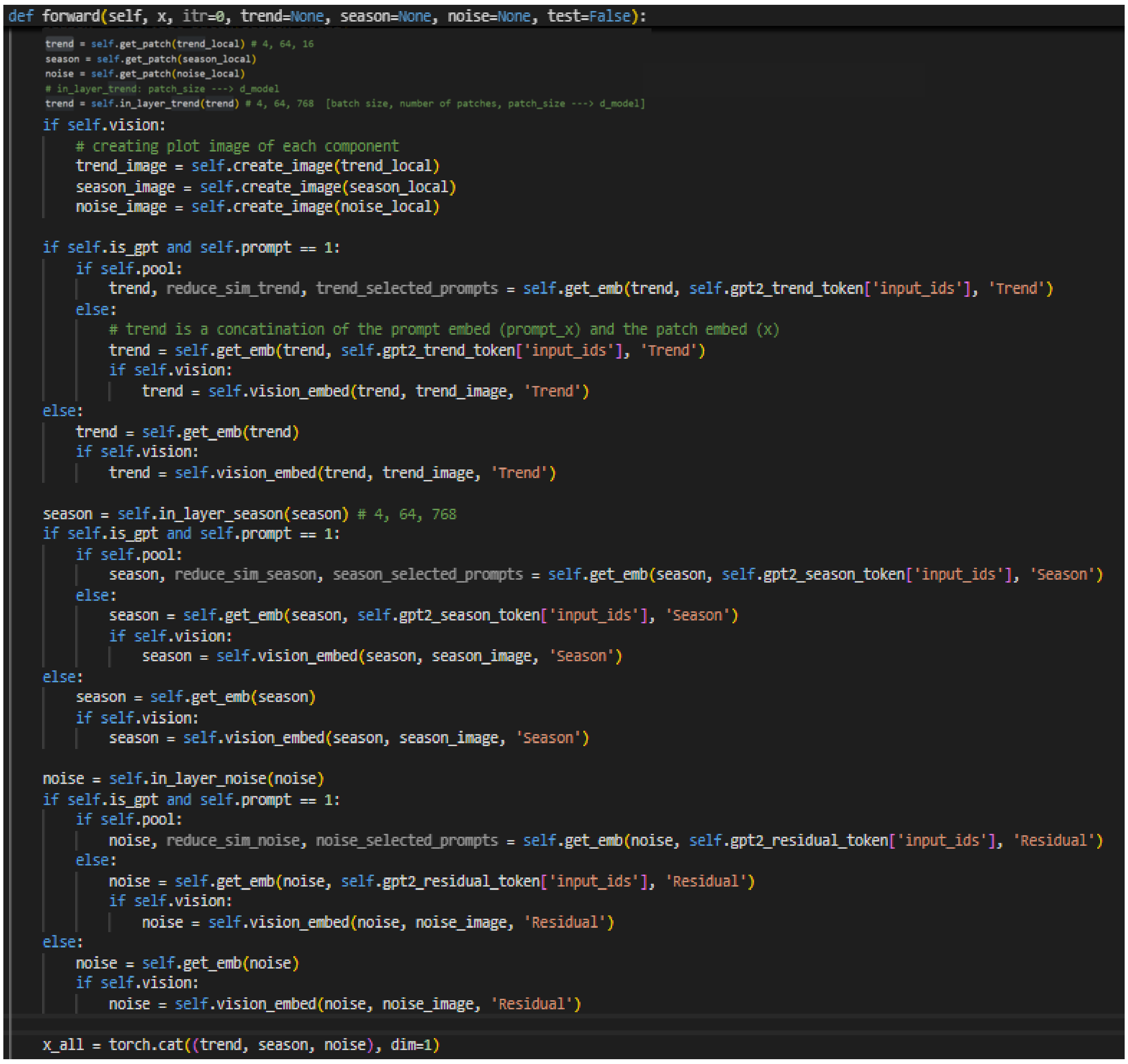


```python
class TEMPO(nn.Module):
    def __init__(self, configs, device):
        ############--adding vision support--#################
        if self.vision:
            self.vision_encoder, self.vision_encoder_preprocess = clip.load("ViT-B/32", device=self.device)
            self.vis_layer_trend = nn.Linear(configs.vis_encoder_dim, configs.d_model)
            self.vis_layer_season = nn.Linear(configs.vis_encoder_dim, configs.d_model)
            self.vis_layer_noise = nn.Linear(configs.vis_encoder_dim, configs.d_model)
```

Adjusting the forward function to support vision embedding:

We use the create_image function to generate plot images.

After extracting patch embeddings using get_patch and combining them with prompt embeddings via get_emb, we integrate vision embeddings by applying the vision_embed function.



```python
def forward(self, x, itr=0, trend=None, season=None, noise=None, test=False):
        trend = self.get_patch(trend_local) # 4, 64, 16
        season = self.get_patch(season_local)
        noise = self.get_patch(noise_local)
        # in_layer_trend: patch_size ---> d_model
        trend = self.in_layer_trend(trend) # 4, 64, 768  [batch size, number of patches, patch_size ---> d_model]
        if self.vision:
            # creating plot image of each component
            trend_image = self.create_image(trend_local)
            season_image = self.create_image(season_local)
            noise_image = self.create_image(noise_local)

        if self.is_gpt and self.prompt == 1:
            if self.pool:
                trend, reduce_sim_trend, trend_selected_prompts = self.get_emb(trend, self.gpt2_trend_token['input_ids'], 'Trend')
            else:
                # trend is a concatination of the prompt embed (prompt_x) and the patch embed (x)
                trend = self.get_emb(trend, self.gpt2_trend_token['input_ids'], 'Trend')
                if self.vision:
                    trend = self.vision_embed(trend, trend_image, 'Trend')
        else:
            trend = self.get_emb(trend)
            if self.vision:
                trend = self.vision_embed(trend, trend_image, 'Trend')

        season = self.in_layer_season(season) # 4, 64, 768
        if self.is_gpt and self.prompt == 1:
            if self.pool:
                season, reduce_sim_season, season_selected_prompts = self.get_emb(season, self.gpt2_season_token['input_ids'], 'Season')
            else:
                season = self.get_emb(season, self.gpt2_season_token['input_ids'], 'Season')
                if self.vision:
                    season = self.vision_embed(season, season_image, 'Season')
        else:
            season = self.get_emb(season)
            if self.vision:
                season = self.vision_embed(season, season_image, 'Season')

        noise = self.in_layer_noise(noise)
        if self.is_gpt and self.prompt == 1:
            if self.pool:
                noise, reduce_sim_noise, noise_selected_prompts = self.get_emb(noise, self.gpt2_residual_token['input_ids'], 'Residual')
            else:
                noise = self.get_emb(noise, self.gpt2_residual_token['input_ids'], 'Residual')
                if self.vision:
                    noise = self.vision_embed(noise, noise_image, 'Residual')
        else:
            noise = self.get_emb(noise)
            if self.vision:
                noise = self.vision_embed(noise, noise_image, 'Residual')

        x_all = torch.cat((trend, season, noise), dim=1)
```