Transformer_Time_series_v4

July 26, 2020

```
In [1]: import numpy as np
        import tensorflow as tf
        import os
        import pandas as pd

        plot_path = "plots/"

        # Real server data

        root_path = "Data/Ant_202007/"


        cif = pd.read_json(root_path+'cif.json', orient='index')
        paycore = pd.read_json(root_path+'paycore.json', orient='index')
        paydecision = pd.read_json(root_path+'paydecision.json', orient='index')
        paydecision2 = pd.read_json(root_path+'paydecision2.json', orient='index')
        paydecision3 = pd.read_json(root_path+'paydecision3.json', orient='index')

        df = pd.DataFrame()
        df["time_stamp"] = cif.index
        df["cif"] = cif[0].values
        df["paycore"] = paycore[0].values
        df["paydecision"] = paydecision[0].values
        df["paydecision2"] = paydecision2[0].values
        df["paydecision3"] = paydecision3[0].values

        # Optional
        if False:
            df.to_csv(root_path+"fusion.csv")


        # convert time stamp
        df['time_stamp'] = pd.to_datetime(df['time_stamp'])
        names_array = np.array(df.keys()[1:],dtype="str")
        os.listdir(root_path)


        if True:
```

```
    # calculate previous hour high low:
    # convert to seconds
    temp = df['time_stamp'] - min(df['time_stamp'])
    temp = temp.dt.total_seconds().astype(int)
    df["hours"] = temp//3600

    h_max = max(df["hours"])+1

    for n in range(len(names_array)):
        df[names_array[n]+"_open"] = df[names_array[n]]
        df[names_array[n]+"_close"] = df[names_array[n]]
        df[names_array[n]+"_max"] = df[names_array[n]]
        df[names_array[n]+"_min"] = df[names_array[n]]

    for j in range(1,h_max):
        mask_j = df["hours"]==j-1
        max_val = df[mask_j][names_array].max(axis=0).values
        min_val = df[mask_j][names_array].max(axis=0).values
        open_val = df[mask_j][names_array].values[0,:]
        close_val = df[mask_j][names_array].values[-1,:]
        mask_i = df["hours"]==j
        r = df[mask_i][names_array].shape[0]
        df.loc[mask_i,[r+"_open" for r in names_array]] = np.tile(open_val,(r,1))
        df.loc[mask_i,[r+"_close" for r in names_array]] = np.tile(close_val,(r,1))

        df.loc[mask_i,[r+"_max" for r in names_array]] = np.tile(max_val,(r,1))
        df.loc[mask_i,[r+"_min" for r in names_array]] = np.tile(min_val,(r,1))
```

```
In [2]: # scale dot attention:
        def scaled_dot_product_attention(q, k, v, mask):
            matmul_qk = tf.matmul(q, k, transpose_b=True)
            # Dimension of k
            dk = tf.cast(tf.shape(k)[-1], tf.float32)
            scaled_attention_logits = matmul_qk / tf.math.sqrt(dk)
            if mask is not None:
                scaled_attention_logits += (mask * -1e9)
            # calculate attention weight:
            attention_weights = tf.nn.softmax(scaled_attention_logits, axis=-1)
            output = tf.matmul(attention_weights, v)
            return output, attention_weights

        # Multi-head Attention:
        # This is what we use
        class MultiHeadAttention(tf.keras.layers.Layer):
            def __init__(self, d_model, num_heads):
```

```python
        # Always use Super to inheriatte and avoid extra code.
        assert d_model%num_heads==0
        super(MultiHeadAttention, self).__init__()
        self.num_heads = num_heads
        self.d_model = d_model
        # sanity check:
        assert d_model % self.num_heads == 0
        self.depth = d_model // self.num_heads
        # Q K W:
        self.wq = tf.keras.layers.Dense(d_model)
        self.wk = tf.keras.layers.Dense(d_model)
        self.wv = tf.keras.layers.Dense(d_model)

        self.dense = tf.keras.layers.Dense(d_model)
    def split_heads(self, x, batch_size):
        # Transpose the result such that the shape is (batch_size, num_heads, seq_len,
        x = tf.reshape(x, (batch_size, -1, self.num_heads, self.depth))
        return tf.transpose(x, perm=[0, 2, 1, 3])
    def call(self, v, k, q, mask):
        batch_size = tf.shape(q)[0]
        q = self.wq(q)  # (batch_size, seq_len, d_model)
        k = self.wk(k)  # (batch_size, seq_len, d_model)
        v = self.wv(v)  # (batch_size, seq_len, d_model)

        q = self.split_heads(q, batch_size)  # (batch_size, num_heads, seq_len_q, depth
        k = self.split_heads(k, batch_size)  # (batch_size, num_heads, seq_len_k, depth
        v = self.split_heads(v, batch_size)  # (batch_size, num_heads, seq_len_v, depth

        # scaled_attention.shape == (batch_size, num_heads, seq_len_q, depth)
        # attention_weights.shape == (batch_size, num_heads, seq_len_q, seq_len_k)

        scaled_attention, attention_weights = scaled_dot_product_attention(q, k, v, mas
        # https://www.tensorflow.org/api_docs/python/tf/transpose : perm
        scaled_attention = tf.transpose(scaled_attention, perm=[0, 2, 1, 3])  # (batch_
        concat_attention = tf.reshape(scaled_attention,
                                      (batch_size, -1, self.d_model))  # (batch_size, seq_
        output = self.dense(concat_attention)  # (batch_size, seq_len_q, d_model)
        return output, attention_weights
```

```python
# check our Multi-head attention:
# D_model must be divided by num_head

n_d_model=32
temp_mha = MultiHeadAttention(d_model=n_d_model, num_heads=8)
```

```
y = tf.random.uniform((16, 60, n_d_model))  # (batch_size, encoder_sequence, d_model)
out, attn = temp_mha(y, k=y, q=y, mask=None)
out.shape, attn.shape
```

Out[2]: (TensorShape([16, 60, 32]), TensorShape([16, 8, 60, 60]))

In [3]:
```
## Encoder decoder for Time series:

# pointwise feed forward network
def point_wise_feed_forward_network(d_model, dff):
    # Two FC layers:
    return tf.keras.Sequential([
      tf.keras.layers.Dense(dff, activation='relu'),  # (batch_size, seq_len, dff)
      tf.keras.layers.Dense(d_model)  # (batch_size, seq_len, d_model)
  ])

# Change embedding since it's not int anymore:
class EmbeddingLayer(tf.keras.layers.Layer):
    def __init__(self,embedding_size):
        super(EmbeddingLayer,self).__init__()
        self.embedding_size=embedding_size

    def build(self,input_shape):
        with tf.name_scope('embedding'):
            self.shared_weights=self.add_weight(name='weights',
                                                shape=[input_shape[-1],self.embedding_s
                                                initializer=tf.random_normal_initialize

        super(EmbeddingLayer,self).build(input_shape)


    def call(self,x):
        y=tf.einsum('bsf,fk->bsk',x,self.shared_weights)
        return y


class EncoderLayer(tf.keras.layers.Layer):
    # Here we use a 0.1 dropout rate as default
    def __init__(self, d_model, num_heads, dff, rate=0.1):
        super(EncoderLayer, self).__init__()
        self.mha = MultiHeadAttention(d_model, num_heads)
        self.ffn = point_wise_feed_forward_network(d_model, dff)

        self.layernorm1 = tf.keras.layers.LayerNormalization(epsilon=1e-6)
        self.layernorm2 = tf.keras.layers.LayerNormalization(epsilon=1e-6)

        self.dropout1 = tf.keras.layers.Dropout(rate)
```

```
        self.dropout2 = tf.keras.layers.Dropout(rate)

    def call(self, x, training, mask):
        attn_output, _ = self.mha(x, x, x, mask)  # (batch_size, input_seq_len, d_mode
        attn_output = self.dropout1(attn_output, training=training)

        out1 = self.layernorm1(x + attn_output)  # (batch_size, input_seq_len, d_model
        ffn_output = self.ffn(out1)  # (batch_size, input_seq_len, d_model)
        ffn_output = self.dropout2(ffn_output, training=training)

        out2 = self.layernorm2(out1 + ffn_output)  # (batch_size, input_seq_len, d_mod

        return out2
sample_encoder_layer = EncoderLayer(512, 8, 2048)

sample_encoder_layer_output = sample_encoder_layer(tf.random.uniform((64, 43, 512)), Fa

print(sample_encoder_layer_output.shape)  # (batch_size, input_seq_len, d_model)

class DecoderLayer(tf.keras.layers.Layer):
    def __init__(self, d_model, num_heads, dff, rate=0.1):
        super(DecoderLayer, self).__init__()

        self.mha1 = MultiHeadAttention(d_model, num_heads)
        self.mha2 = MultiHeadAttention(d_model, num_heads)

        self.ffn = point_wise_feed_forward_network(d_model, dff)

        self.layernorm1 = tf.keras.layers.LayerNormalization(epsilon=1e-6)
        self.layernorm2 = tf.keras.layers.LayerNormalization(epsilon=1e-6)
        self.layernorm3 = tf.keras.layers.LayerNormalization(epsilon=1e-6)

        self.dropout1 = tf.keras.layers.Dropout(rate)
        self.dropout2 = tf.keras.layers.Dropout(rate)
        self.dropout3 = tf.keras.layers.Dropout(rate)

    def call(self, x, enc_output, training, look_ahead_mask, padding_mask):
        # enc_output.shape == (batch_size, input_seq_len, d_model)

        attn1, attn_weights_block1 = self.mha1(x, x, x, look_ahead_mask)  # (batch_siz
        attn1 = self.dropout1(attn1, training=training)
        out1 = self.layernorm1(attn1 + x)

        attn2, attn_weights_block2 = self.mha2(
            enc_output, enc_output, out1, padding_mask)  # (batch_size, target_seq_len
        attn2 = self.dropout2(attn2, training=training)
        out2 = self.layernorm2(attn2 + out1)  # (batch_size, target_seq_len, d_model)
```

```
        ffn_output = self.ffn(out2)  # (batch_size, target_seq_len, d_model)
        ffn_output = self.dropout3(ffn_output, training=training)
        out3 = self.layernorm3(ffn_output + out2)  # (batch_size, target_seq_len, d_mo

        return out3, attn_weights_block1, attn_weights_block2

sample_decoder_layer = DecoderLayer(512, 8, 2048)

sample_decoder_layer_output, _, _ = sample_decoder_layer(
    tf.random.uniform((64, 50, 512)), sample_encoder_layer_output,
    False, None, None)

print(sample_decoder_layer_output.shape)  # (batch_size, target_seq_len, d_model)
```

```
(64, 43, 512)
(64, 50, 512)
```

In [4]:
```
def get_angles(pos, i, d_model):
    angle_rates = 1 / np.power(10000, (2 * (i//2)) / np.float32(d_model))
    return pos * angle_rates

def positional_encoding(position, d_model):
    angle_rads = get_angles(np.arange(position)[:, np.newaxis],
                            np.arange(d_model)[np.newaxis, :],
                            d_model)

    # apply sin to even indices in the array; 2i
    angle_rads[:, 0::2] = np.sin(angle_rads[:, 0::2])

    # apply cos to odd indices in the array; 2i+1
    angle_rads[:, 1::2] = np.cos(angle_rads[:, 1::2])
    pos_encoding = angle_rads[np.newaxis, ...]
    return tf.cast(pos_encoding, dtype=tf.float32)

class Encoder(tf.keras.layers.Layer):
    def __init__(self, num_layers, d_model, num_heads, dff, input_vocab_size,
                 maximum_position_encoding, rate=0.1):
        super(Encoder, self).__init__()

        self.d_model = d_model
        self.num_layers = num_layers

        self.embedding = tf.keras.layers.Embedding(input_vocab_size, d_model)
        self.pos_encoding = positional_encoding(maximum_position_encoding,
                                                self.d_model)
```

```
        self.enc_layers = [EncoderLayer(d_model, num_heads, dff, rate)
                           for _ in range(num_layers)]

        self.dropout = tf.keras.layers.Dropout(rate)

    def call(self, x, training, mask):
        seq_len = tf.shape(x)[1]

        # adding embedding and position encoding.
        #print("Check",x.shape)
        x = self.embedding(x)  # (batch_size, input_seq_len, d_model)
        #x = tf.keras.layers.Dense(self.d_model)(x)
        #print("check 2",x.shape)
        x *= tf.math.sqrt(tf.cast(self.d_model, tf.float32))
        x += self.pos_encoding[:, :seq_len, :]
        #print("check 3",x.shape)

        x = self.dropout(x, training=training)
        #print("check 4",x.shape)

        for i in range(self.num_layers):
            x = self.enc_layers[i](x, training, mask)
        return x  # (batch_size, input_seq_len, d_model)

class Decoder(tf.keras.layers.Layer):
    def __init__(self, num_layers, d_model, num_heads, dff, target_vocab_size,
                 maximum_position_encoding, rate=0.1):
        super(Decoder, self).__init__()
        self.d_model = d_model
        self.num_layers = num_layers

        self.embedding = tf.keras.layers.Embedding(target_vocab_size, d_model)
        self.pos_encoding = positional_encoding(maximum_position_encoding, d_model)

        self.dec_layers = [DecoderLayer(d_model, num_heads, dff, rate)
                           for _ in range(num_layers)]
        self.dropout = tf.keras.layers.Dropout(rate)

    def call(self, x, enc_output, training, look_ahead_mask, padding_mask):
        seq_len = tf.shape(x)[1]
        attention_weights = {}

        x = self.embedding(x)  # (batch_size, target_seq_len, d_model)
        #x = tf.keras.layers.Dense(self.d_model)(x)

        x *= tf.math.sqrt(tf.cast(self.d_model, tf.float32))
        x += self.pos_encoding[:, :seq_len, :]
```

```
        x = self.dropout(x, training=training)
        for i in range(self.num_layers):
            x, block1, block2 = self.dec_layers[i](x, enc_output, training,
                                                   look_ahead_mask, padding_mask)

            attention_weights['decoder_layer{}_block1'.format(i+1)] = block1
            attention_weights['decoder_layer{}_block2'.format(i+1)] = block2

        # x.shape == (batch_size, target_seq_len, d_model)
        return x, attention_weights
```

In [47]:
```
class Transformer(tf.keras.Model):
    def __init__(self, num_layers, d_model, num_heads, dff, input_vocab_size,
                 target_vocab_size, pe_input, pe_target, rate=0.1):
        super(Transformer, self).__init__()

        self.encoder = Encoder(num_layers, d_model, num_heads, dff,
                               input_vocab_size, pe_input, rate)

        self.decoder = Decoder(num_layers, d_model, num_heads, dff,
                               target_vocab_size, pe_target, rate)

        self.final_layer = tf.keras.layers.Dense(target_vocab_size)

        # calculate final value:
        self.mu = tf.keras.layers.Dense(1, kernel_initializer=tf.keras.initializers.Ra
        self.sigma = tf.keras.layers.Dense(1, activation='softplus',
                                           kernel_initializer=tf.keras.initializers.Ra

    def call(self, inp, tar, training, enc_padding_mask,
             look_ahead_mask, dec_padding_mask):
        enc_output = self.encoder(inp, training, enc_padding_mask)  # (batch_size, in
        #print("check encoder size",enc_output.shape)


        # dec_output.shape == (batch_size, tar_seq_len, d_model)
        dec_output, attention_weights = self.decoder(
            tar, enc_output, training, look_ahead_mask, dec_padding_mask)

        #print("check decoder size",dec_output.shape)

        final_output = self.final_layer(dec_output)  # (batch_size, tar_seq_len, targ
        #print("check output",self.mu(dec_output).shape)

        return final_output, attention_weights
        #return self.mu(dec_output), attention_weights
```

```
In [48]: # Change pe_target to output delta_t
         # change pe_input to input delta_t
         # position = np.arange(seq_len): length of a sliding window, which is pe_input and pe_
         # Here pe_input and

         # For GTX 1060 we can set batch=16 and use 4X batch size for Tesla P40

         batch = 8

         sample_transformer = Transformer(
             num_layers=2, d_model=512, num_heads=8, dff=2048,
             input_vocab_size=1000, target_vocab_size=1000,
             pe_input=1440, pe_target=240)

         # input: batch+sequence length
         # biggest length for in/out put is pe_input,  pe_target
         temp_input = tf.random.uniform((batch, 720), dtype=tf.int64, minval=0, maxval=1000)
         temp_target = tf.random.uniform((batch, 1), dtype=tf.int64, minval=0, maxval=1000)

         #temp_input = tf.cast(temp_input,dtype=tf.float32)
         #temp_target = tf.cast(temp_target,dtype=tf.float32)

         fn_out, _ = sample_transformer(temp_input, temp_target, training=False,
                                        enc_padding_mask=None,
                                        look_ahead_mask=None,
                                        dec_padding_mask=None)

         print("final output size",fn_out.shape)  # (batch_size, tar_seq_len, target_vocab_siz
```

```
final output size (8, 1, 1000)
```

```
In [50]: # sample_transformer.summary()
```

```
In [9]: # Load data:names_array


        temp = df["cif"]
        # Normalize to 0-1000

        temp = (temp-min(temp))/(max(temp)-min(temp))

        lower, upper = 0, 999
        temp = [lower + (upper - lower) * x for x in temp]
        temp = np.array(temp,dtype=int)
        delta_t = 720
```

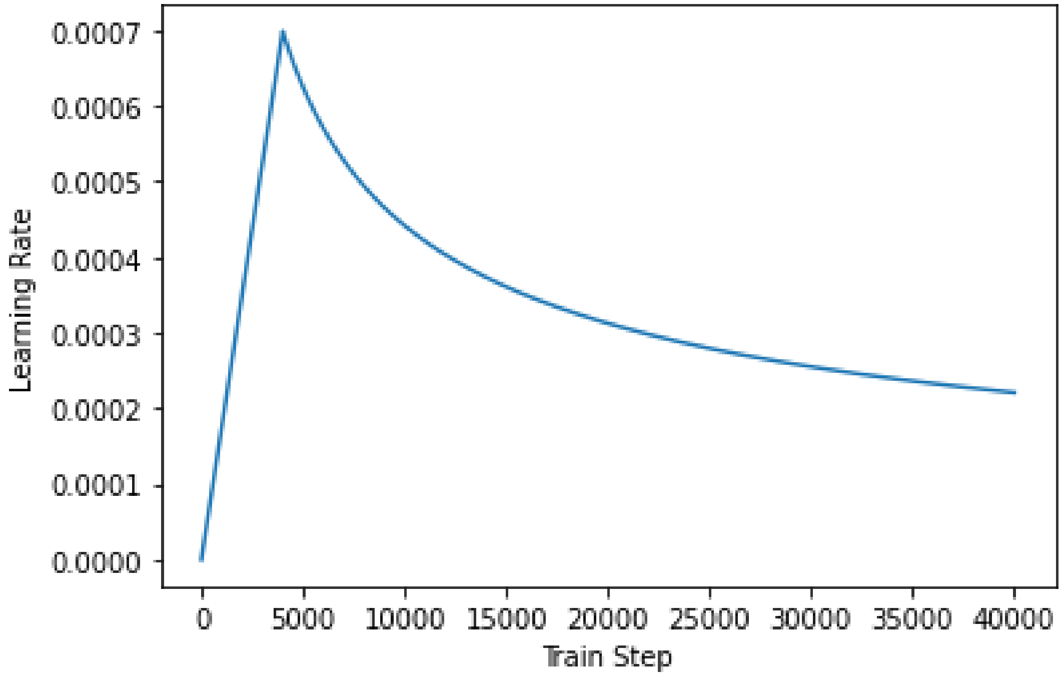
```
        X = np.zeros((temp.shape[0]-delta_t,delta_t,1),dtype=int)
        y = temp[delta_t:]

        for i in range(len(y)):
            if i%10000==0:
                print("Prepare data %.2f percent"%(100*i/len(y)))
            X[i,:,:] = np.atleast_2d(temp[i:i+delta_t]).T

        train_dataset_TS = tf.data.Dataset.from_tensor_slices((X,y))
```

```
Prepare data 0.00 percent
Prepare data 24.37 percent
Prepare data 48.73 percent
Prepare data 73.10 percent
Prepare data 97.46 percent
```

In [ ]:

In [51]:
```
## Optimizer:
import matplotlib.pyplot as plt

d_model=512

class CustomSchedule(tf.keras.optimizers.schedules.LearningRateSchedule):
    def __init__(self, d_model, warmup_steps=4000):
        super(CustomSchedule, self).__init__()

        self.d_model = d_model
        self.d_model = tf.cast(self.d_model, tf.float32)

        self.warmup_steps = warmup_steps

    def __call__(self, step):
        arg1 = tf.math.rsqrt(step)
        arg2 = step * (self.warmup_steps ** -1.5)

        return tf.math.rsqrt(self.d_model) * tf.math.minimum(arg1, arg2)

learning_rate = CustomSchedule(d_model)

optimizer = tf.keras.optimizers.Adam(learning_rate, beta_1=0.9, beta_2=0.98,
                                     epsilon=1e-9)
# Learning rate curve:
temp_learning_rate_schedule = CustomSchedule(d_model)

plt.plot(temp_learning_rate_schedule(tf.range(40000, dtype=tf.float32)))
plt.ylabel("Learning Rate")
plt.xlabel("Train Step")
```

Out[51]: Text(0.5, 0, 'Train Step')



In [52]:
```python
# loss and metric
loss_object = tf.keras.losses.SparseCategoricalCrossentropy(
    from_logits=True, reduction='none')


def loss_function(real, pred):
    mask = tf.math.logical_not(tf.math.equal(real, 0))
    loss_ = loss_object(real, pred)

    mask = tf.cast(mask, dtype=loss_.dtype)
    loss_ *= mask

    return tf.reduce_sum(loss_)/tf.reduce_sum(mask)




train_loss = tf.keras.metrics.Mean(name='train_loss')
train_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(
    name='train_accuracy')
```

In [53]:
```python
def create_padding_mask(seq):
    seq = tf.cast(tf.math.equal(seq, 0), tf.float32)
```

```python
    # add extra dimensions to add the padding
    # to the attention logits.
    return seq[:, tf.newaxis, tf.newaxis, :]  # (batch_size, 1, 1, seq_len)

def create_look_ahead_mask(size):
    mask = 1 - tf.linalg.band_part(tf.ones((size, size)), -1, 0)
    return mask  # (seq_len, seq_len)

def create_masks(inp, tar):
    # Encoder padding mask
    enc_padding_mask = create_padding_mask(inp)

    # Used in the 2nd attention block in the decoder.
    # This padding mask is used to mask the encoder outputs.
    dec_padding_mask = create_padding_mask(inp)

    # Used in the 1st attention block in the decoder.
    # It is used to pad and mask future tokens in the input received by
    # the decoder.
    look_ahead_mask = create_look_ahead_mask(tf.shape(tar)[1])
    dec_target_padding_mask = create_padding_mask(tar)
    combined_mask = tf.maximum(dec_target_padding_mask, look_ahead_mask)


    return enc_padding_mask, combined_mask, dec_padding_mask
```

In [54]:
```python
batch = 8

sample_transformer = Transformer(
    num_layers=2, d_model=512, num_heads=8, dff=2048,
    input_vocab_size=1000, target_vocab_size=1000,
    pe_input=1440, pe_target=240)


transformer = sample_transformer
"""



train_step_signature = [
    tf.TensorSpec(shape=(None, None), dtype=tf.int64),
    tf.TensorSpec(shape=(None, None), dtype=tf.int64),
]

@tf.function(input_signature=train_step_signature)
def train_step(inp, tar):

    tar_inp = tar[:, :-1]
```

```
        tar_real = tar[:, 1:]

        enc_padding_mask, combined_mask, dec_padding_mask = create_masks(inp, tar_inp)

        with tf.GradientTape() as tape:
            predictions, _ = transformer(inp, tar_inp,
                                         True,
                                         enc_padding_mask,
                                         combined_mask,
                                         dec_padding_mask)
            loss = loss_function(tar_real, predictions)

        gradients = tape.gradient(loss, transformer.trainable_variables)
        optimizer.apply_gradients(zip(gradients, transformer.trainable_variables))

        train_loss(loss)
        train_accuracy(tar_real, predictions)
        #return loss



    """
```

```
Out[54]: '\n\n\ntrain_step_signature = [\n    tf.TensorSpec(shape=(None, None), dtype=tf.int64
```

```
In [55]: loss_object = tf.keras.losses.SparseCategoricalCrossentropy(
             from_logits=True, reduction='none')



         # remove the mask inside the loss

         def loss_function(real, pred):
             #mask = tf.math.logical_not(tf.math.equal(real, 0))
             # No MAE for now
             #loss_ = tf.keras.losses.MAE(y_true=real, y_pred=pred)
             loss_ = loss_object(real, pred)
             #mask = tf.cast(mask, dtype=loss_.dtype)
             #loss_ *= mask

             return tf.reduce_sum(loss_)
```

```
In [56]: if False:

             @tf.function(input_signature=train_step_signature)
             def train_step(inp, tar):
                 train_step_signature = [
                 tf.TensorSpec(shape=(None, None), dtype=tf.int64),
```

```
        tf.TensorSpec(shape=(None, None), dtype=tf.int64),]

        with tf.GradientTape() as tape:
            predictions, _ = transformer(inp, tar,
                                         True,
                                         None,
                                         None,
                                         None)
            loss = loss_function(tar, predictions)

        gradients = tape.gradient(loss, transformer.trainable_variables)
        optimizer.apply_gradients(zip(gradients, transformer.trainable_variables))

        train_loss(loss)
        train_accuracy(tar, predictions)

        return loss
```

In [57]:
```
checkpoint_path = "./checkpoints/train_TS"

ckpt = tf.train.Checkpoint(transformer=transformer,
                           optimizer=optimizer)

ckpt_manager = tf.train.CheckpointManager(ckpt, checkpoint_path, max_to_keep=5)

# if a checkpoint exists, restore the latest checkpoint.
if ckpt_manager.latest_checkpoint:
    ckpt.restore(ckpt_manager.latest_checkpoint)
    print ('Latest checkpoint restored!!')




if True:


    train_step_signature = [
        tf.TensorSpec(shape=(None, None), dtype=tf.int64),
        tf.TensorSpec(shape=(None, None), dtype=tf.int64),
    ]

    @tf.function(input_signature=train_step_signature)

    def train_step(inp, tar):
        #tar_inp = tar[:, :-1]
        #tar_real = tar[:, 1:]
        tar_inp = tar
```

```
            tar_real = tar

            enc_padding_mask, combined_mask, dec_padding_mask = create_masks(inp, tar_inp
            with tf.GradientTape() as tape:
                predictions, _ = transformer(inp, tar_inp,
                                             True,
                                             enc_padding_mask,
                                             combined_mask,
                                             dec_padding_mask)
                loss = loss_function(tar_real, predictions)
            gradients = tape.gradient(loss, transformer.trainable_variables)
            optimizer.apply_gradients(zip(gradients, transformer.trainable_variables))

            train_loss(loss)
            train_accuracy(tar_real, predictions)
```

In [ ]:
```
import time

EPOCHS = 10
train_dataset = tf.data.Dataset.from_tensor_slices((X,y))
batch=16

N = len(y)

for epoch in range(EPOCHS):
    start = time.time()

    train_loss.reset_states()
    train_accuracy.reset_states()
    for i in range(N//batch):
        inp, tar=X[batch*i:min(batch*i+batch,N),:,0],y[batch*i:min(batch*i+batch,N)]
        tar = np.atleast_2d(tar).T
        lo = train_step(inp, tar)
        if i%200==0:
            print("Doing %d (%d) batch in epoch %d "%(i,N//batch,epoch))
            print("Loss",train_loss.result(), "Accuracy",train_accuracy.result())
```

In [ ]:
```
#  Prediction:
# Predict like batch*1*1000. For classification categorical. one of 1000 classes with 
```

In [70]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, shuffle=Fals
```

In [72]:
```
# testing:
N_test = len(y_test)

y_pred_all = []
```

```
    for i in range(N_test//batch):
        if i%200==0:
            print("Doing %d (%d)"%(i,N_test//batch))

        inp, tar=X_test[batch*i:min(batch*i+batch,N),:,0],y_test[batch*i:min(batch*i+batc
        tar = np.atleast_2d(tar).T
        tar_inp = tar
        tar_real = tar

        enc_padding_mask, combined_mask, dec_padding_mask = create_masks(inp, tar_inp)

        predictions, attention_weights = transformer(inp,
                                                     tar,
                                                     False,
                                                     enc_padding_mask,
                                                     combined_mask,
                                                     dec_padding_mask)
        y_pred_all.extend(tf.cast(tf.argmax(predictions, axis=-1), tf.int32))



    # I'm just being lazy :)
    y_pred_all = np.array(y_pred_all).ravel()
    y_test = y_test[:len(y_pred_all)]
```

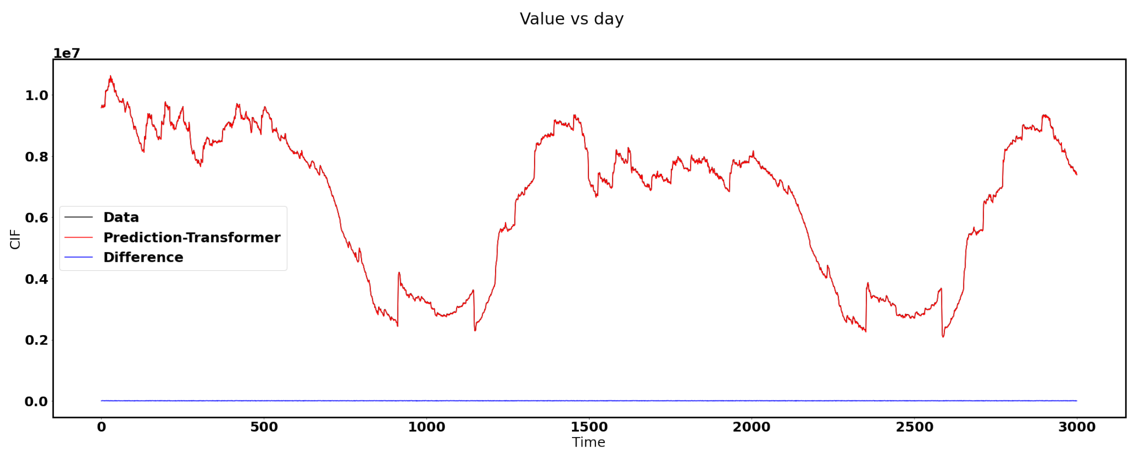
```
Doing 0 (769)
Doing 200 (769)
Doing 400 (769)
Doing 600 (769)
```

!!!!!!! One thing SparseCategoricalCrossentropy is not good is that if label is 500, predict to be 499 and 999 are definitely equal but it's not. So we should also consider MAE as our loss...

```
In [ ]: # normalize back

        temp = df["cif"]

        #y_test = (y_test-lower)/(upper - lower)
        y_pred_all = (y_pred_all-lower)/(upper - lower)
        #y_test = (max(temp)-min(temp))*y_test+min(temp)
        y_pred_all = (max(temp)-min(temp))*y_pred_all+min(temp)


        lower, upper = 0, 999
        temp = [lower + (upper - lower) * x for x in temp]
        temp = np.array(temp,dtype=int)
```

```
In [121]: shift = N_test-N_test//batch*batch
          v4 = df["cif"].values[-len(y_pred_all)-shift:-shift]
```

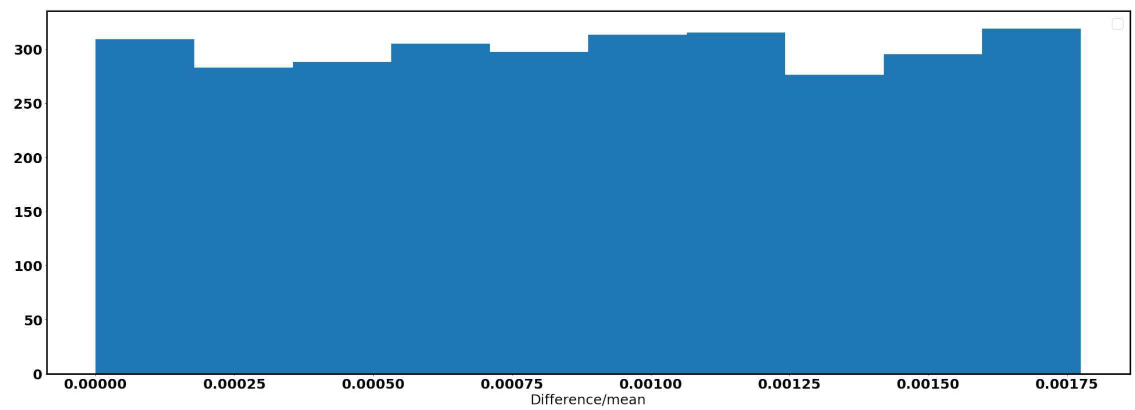
```
import matplotlib
from matplotlib.pylab import rc


font = {'family': 'normal','weight': 'bold',
        'size': 25}

matplotlib.rc('font', **font)
rc('axes', linewidth=3)

plt.plot(v4[:3000],"k",label="Data")
plt.plot(y_pred_all[:3000],"r",label="Prediction-Transformer")
diff = v4[:3000]-y_pred_all[:3000]
plt.plot(diff,"b",label="Difference")

plt.xlabel("Time")
plt.ylabel(r"CIF")
plt.suptitle("Value vs day")

fig = matplotlib.pyplot.gcf()
plt.legend()

fig.set_size_inches(35,12)

save_path = plot_path + "Transformer_CIF" + ".png"

fig.savefig(save_path, dpi=200)
```




```
In [126]: font = {'family': 'normal','weight': 'bold',
                  'size': 25}
```

```
    matplotlib.rc('font', **font)
    rc('axes', linewidth=3)

    plt.hist(diff/np.nanmean(v4))
    plt.xlabel("Difference/mean")


    fig = matplotlib.pyplot.gcf()
    plt.legend()

    fig.set_size_inches(35,12)

    save_path = plot_path + "Transformer_CIF_diff_hist" + ".png"

    fig.savefig(save_path, dpi=200)
```

```
No handles with labels found to put in legend.
```




```
In [ ]: # Let's do some lone term prediction like 60 minutes?
        # testing:
        N_test = len(y_test)

        y_pred_all_long = []
        for i in range(N_test//batch):
            if i%200==0:
                print("Doing %d (%d)"%(i,N_test//batch))

            inp, tar=X_test[batch*i:min(batch*i+batch,N),:,0],y_test[batch*i:min(batch*i+batch
            tar = np.tile(tar,(60,1)).T

            tar_inp = tar
            tar_real = tar
```

```
            enc_padding_mask, combined_mask, dec_padding_mask = create_masks(inp, tar_inp)

            predictions, attention_weights = transformer(inp,
                                                         tar,
                                                         False,
                                                         enc_padding_mask,
                                                         combined_mask,
                                                         dec_padding_mask)
            y_pred_all_long.extend(tf.cast(tf.argmax(predictions, axis=-1), tf.int32))



        # I'm just being lazy :)
        y_pred_all = np.array(y_pred_all).ravel()
        y_test = y_test[:len(y_pred_all)]
```

In [136]:
```
temp_target = tf.random.uniform((batch, 60), dtype=tf.int64, minval=0, maxval=1000)
predictions, attention_weights = transformer(inp,
                                             temp_target,
                                             False,
                                             None,
                                             None,
                                             None)
```

In [138]:
```
predicted_id=tf.cast(tf.argmax(predictions, axis=-1), tf.int32)
```

In [144]:

Out[144]: (60,)

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]: