

I asked ChatGPT:

How could I convert this RNN to a transformer? # Flatten CNN and concat with other features

```
zack_hack_div_2 = 0
if cnn_rnn_zack:
    zack_hack_div_2 = zack_hack // 2
    cnn_output = tf.slice(cnn_output, [0, zack_hack_div_2, 0, 0], [-1, rnn_nunroll, -1, -1])
    nfeats_conv = reduce(lambda x, y: x * y, [int(x) for x in cnn_output.get_shape()[-2:]])
else:
    nfeats_conv = reduce(lambda x, y: x * y, [int(x) for x in cnn_output.get_shape()[-3:]])
    feats_conv = tf.reshape(cnn_output, [batch_size * rnn_nunroll, nfeats_conv])
    nfeats_tot = nfeats_conv + nfeats
    feats_all = tf.concat([feats_conv, feats_other], axis=1)
    print('feats_cnn: {}'.format(feats_conv.get_shape()))
    print('feats_all: {}'.format(feats_all.get_shape()))

# Project to RNN size
rnn_output = feats_all
rnn_output_size = nfeats_tot
if do_rnn:
    with tf.variable_scope('rnn_proj'):
        rnn_proj_w = tf.get_variable('W', [nfeats_tot, rnn_size],
initializer=tf.uniform_unit_scaling_initializer(factor=1.0, dtype=dtype), dtype=dtype)
        rnn_proj_b = tf.get_variable('b', [rnn_size], initializer=tf.constant_initializer(0.0),
dtype=dtype)

    rnn_inputs = tf.nn.bias_add(tf.matmul(feats_all, rnn_proj_w), rnn_proj_b)
    rnn_inputs = tf.reshape(rnn_inputs, [batch_size, rnn_nunroll, rnn_size])
    rnn_inputs = tf.split(rnn_inputs, rnn_nunroll, axis=1)
    rnn_inputs = [tf.squeeze(input_, [1]) for input_ in rnn_inputs]

    if rnn_cell_type == 'rnn':
        cell_fn = tf.nn.rnn_cell.BasicRNNCell
    elif rnn_cell_type == 'gru':
        cell_fn = tf.nn.rnn_cell.GRUCell
    elif rnn_cell_type == 'lstm':
        cell_fn = tf.nn.rnn_cell.BasicLSTMCell
    else:
        raise NotImplementedError()
    cell = cell_fn(rnn_size)
```

```

if mode == 'train' and rnn_keep_prob < 1.0:
    cell = tf.nn.rnn_cell.DropoutWrapper(cell, output_keep_prob=rnn_keep_prob)

if rnn_nlayers > 1:
    cell = tf.nn.rnn_cell.MultiRNNCell([cell] * rnn_nlayers)

initial_state = cell.zero_state(batch_size, dtype)

# RNN
# TODO: weight init
with tf.variable_scope('rnn_unroll'):
    state = initial_state
    outputs = []
    for i in range(rnn_nunroll):
        if i > 0:
            tf.get_variable_scope().reuse_variables()
            (cell_output, state) = cell(rnn_inputs[i], state)
            outputs.append(cell_output)
        final_state = state

    rnn_output = tf.reshape(tf.concat(outputs, axis=1), [batch_size * rnn_nunroll, rnn_size])
    rnn_output_size = rnn_size
    print('rnn_output: {}'.format(rnn_output.get_shape()))

```

ChatGPT responded with:

To convert this RNN-based architecture to a **Transformer-based** architecture, you need to replace the RNN unrolling and cell logic with a Transformer encoder block that processes the sequence of inputs in parallel.

What you're replacing

This part:

```

# RNN unroll logic
for i in range(rnn_nunroll):

```

...

becomes a **Transformer encoder** that processes the entire sequence (`[batch_size, rnn_nunroll, rnn_size]`) at once.

Steps to Replace RNN with Transformer

1. Keep CNN + other features as-is

This block stays mostly unchanged:

```
feats_conv = tf.reshape(cnn_output, [batch_size * rnn_nunroll, nfeats_conv])
feats_all = tf.concat([feats_conv, feats_other], axis=1)
```

But instead of reshaping and splitting for RNN, you reshape back to `[batch_size, rnn_nunroll, dim]`.

2. Project to Transformer model dimension

```
# Project to transformer dimension
with tf.variable_scope('transformer_proj'):
    proj_w = tf.get_variable('W', [nfeats_tot, transformer_dim],
        initializer=tf.glorot_uniform_initializer(), dtype=dtype)
    proj_b = tf.get_variable('b', [transformer_dim], initializer=tf.zeros_initializer(), dtype=dtype)
    feats_proj = tf.nn.bias_add(tf.matmul(feats_all, proj_w), proj_b)
    feats_proj = tf.reshape(feats_proj, [batch_size, rnn_nunroll, transformer_dim])
```

3. Add positional encoding

Transformer needs positional information:

```
pos_encoding = positional_encoding(rnn_nunroll, transformer_dim)
feats_proj += pos_encoding # shape [1, rnn_nunroll, transformer_dim]
```

You can use sinusoidal or learnable positional encoding. Here's a basic sinusoidal one:

```
def positional_encoding(length, depth):
    angle_rates = 1 / np.power(10000, (2 * (np.arange(depth)[np.newaxis, :] // 2)) /
np.float32(depth))
    position = np.arange(length)[:, np.newaxis]
    angle_rads = position * angle_rates
    angle_rads[:, 0::2] = np.sin(angle_rads[:, 0::2])
    angle_rads[:, 1::2] = np.cos(angle_rads[:, 1::2])
    return tf.constant(angle_rads[np.newaxis, ...], dtype=tf.float32)
```

4. Replace RNN with Transformer encoder

Use something like this:

```
transformer_layer = tf.keras.layers.MultiHeadAttention(
    num_heads=transformer_heads,
    key_dim=transformer_dim // transformer_heads,
    dropout=dropout_rate
)
attn_output = transformer_layer(feats_proj, feats_proj)
attn_output = tf.keras.layers.LayerNormalization()(feats_proj + attn_output)

ffn_output = tf.keras.layers.Dense(transformer_ff_dim, activation='relu')(attn_output)
ffn_output = tf.keras.layers.Dense(transformer_dim)(ffn_output)
transformer_output = tf.keras.layers.LayerNormalization()(attn_output + ffn_output)
```

To stack layers:

```
for _ in range(transformer_nlayers):
    ...
```

5. Reshape output for downstream

```
# Flatten for classification or next processing
transformer_output = tf.reshape(transformer_output, [batch_size * rnn_nunroll,
transformer_dim])
```

Summary

- Keep CNN + feature merging logic.
- Instead of projecting to RNN and unrolling, you:
 - Project to transformer dimension
 - Add positional encodings
 - Use Transformer encoder blocks
 - Output has shape similar to the original RNN flattened output

Would you like me to rewrite the full block with a working Transformer version in TensorFlow 1.x or 2.x?