11/11/2025

MET CS 767 Assignment 4: GAN's

Brendan Torok

1. How I modified the code to attempt improvement

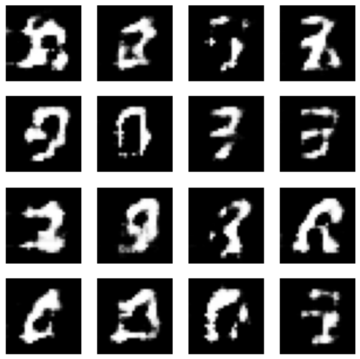
1.1 Description of what you did and reasons it *could reasonably be* an improvement

To improve the original model I first added a weight initializer to every conv/dense layer for both discriminator and generators. This is done to help stabilize learning and make the network see digits sooner. This resulted in a significant improvement in performance. I then added batch normalization to the discriminator to further stabilize activations and improve gradient flow and observed a slight improvement in performance. Based on the textbook suggestion, I experimented with using ReLU activation in all layers except for the output layer in the generator to reduce noisy and grainy textures. I did not see an improvement in performance from this change. I then experimented with decreasing dropout in the discriminator to reduce noise in the discriminators representation to help it learn faster following the addition of batch normalization. This again did not seem to make a difference in performance. I then added a sigmoid activation function to the output layer of the discriminator to test what it would do to the performance and found that performance decreased so I reverted the changes as it destabilized the training.

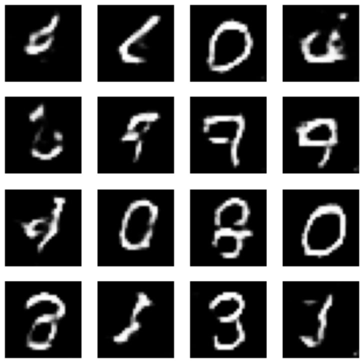
I then experimented with adding layers to both the discriminator and generator but found that it led to mode collapse very early regardless of what combination I used. Due to Google Colab GPU usage limitations and the image GAN not working as expected while using the CPUs runtime, I was only able to implement one final experiment which changed the optimizer to RMSprop. RMSprop does not remember momentum so it immediately reacts to its new data instead of using inertia of previous iterations. RMSprop therefore is less susceptible to mode collapse[1] and greatly improved the performance of the model more than any other change made.

1.2 Comparison of the result with the original output, with explanation

Starting image after 10 epochs:



Ending image after 10 epochs:



After making these changes the ending image is significantly closer to representing numbers. There are clear indications of the numbers 0, 3, 8, and 9 being detected by the model after 10 epochs. However, it appears the model continues to struggle with some more complex number shapes like 2, 4, and 5 as well as simple shapes like 1 and 7. The primary improvements appear to be in detecting thinner lines and connections between that represent numbers rather than blobs of unidentifiable shapes in the starting image. This is a marked improvement after only 10 epochs with 200 gradient descent samples each.

Additional image after 40 epochs in appendix 1. Final code is also available in appendix 1.

Link to LLM chat: https://chatgpt.com/share/69252de3-425c-8006-9b3d-511a034765ff

1.3 URL of your Colab code

https://colab.research.google.com/drive/1b_E3rh9biXPk3wAVW8HyHvHDA4FXX6Yq?usp=sharing

2. *Your* GAN Application (3 pg max)

2.1 Give 2-4 requirements for a unique, highly functional application that you will implement with a GAN

My GAN functions as an addition to an existing hybrid neural network I trained previously to predict stocks that will overperform, underperform, or have neutral performance relative to a benchmark. The input for this application is 180 days of market data for stocks in the S&P1500 and static fundamental statistics for each company. The existing neural network achieved 69% overall accuracy in the three-way classification task, but struggled with neutral data, achieving a precision of only 0.51, and recall of 0.71 for the neutral class. A GAN is used to generate synthetic data in the neutral class to improve the performance of the overall stock picking model.

After training the GAN to accurately synthesize data, the model increased the borders of the neutral class to classify more ambiguous stocks as neutral, improving its classification of overperforming stocks. This is because the GAN-driven data augmentation generates 'central' examples instead of rare borderline cases which may be more helpful in moving the classifiers boundary. The neutral class recall improved substantially, but precision decreased slightly indicating that the classifier is much better at identifying true neutral stocks by expanding the neutral decision region. However, because this region is broadened, some under/over classes near the boundary are classified as neutral. The overperform precision is also improved significantly from 0.82 to 0.90 meaning that the classifier is more conservative in assigning the overperform label. This is ideal when creating a model that will pick stocks using financial data, as it is better to have a large ambiguous neutral class but be very conservative when picking outperforming stocks.

2.2 Sample I/O

Give three varied input/outputs pairs for your implemented application that demonstrate its uniqueness and capable functionality.

Input 1: Model performance before adding the GAN to increase performance of the model, particularly in the neutral class. The starting neutral precision is 0.51 and starting neutral recall of 0.71.

Output 1: Model performance after adding the GAN to generate neutral data. Neutral precision is decreased to 0.48 but neutral recall is increased from 0.71 to 0.82. Overperform precision is increased from 0.82 to 0.90 but overperform recall is decreased from 0.68 to 0.54. This means the model has become more conservative when labeling stocks as overperform, instead assigning ambiguous stocks to the neutral class. Metrics and plot available in appendix 2.

Input 2: Min-max scaled parallel time series channel inputs for open price, close price, high price, low price, volume, VWAP, relative return vs benchmark, and z-scored relative return are fed into the GAN.

Output 2: Synthetic min-max scaled parallel time series channels generated by the GAN. The synthetic data closely resembles the real data after training the GAN. The data is available in appendix 3

Input 3: Both real and synthetic time series data are used as input to a function to determine how similar the synthetic data is to the real data

Output 3: Absolute different between the mean and standard deviation between the real and synthetic data. The differences between the two datasets are very small indicating that the GAN accurately matches the real data. Data is available in appendix 4

2.3 Data source

Explain the source of your data

The data was sourced by from several sources. The stocks included in the S&P1500 was scraped using Wikipedia and a list of the S&P1500 stocks was created. This list was then used to access the yahoo finance API to get static fundamental data for each company. To obtain timeseries data, the API rest client from massive.com was used to query 180 days of trading data for all the stocks in the S&P1500. The same API was also used to obtain 180 days of trading data for the S&P500 which was used as a benchmark. Each stocks daily performance was then compared to the baseline, and the data was then min-max scaled to use the tanh activation function in the GAN.

2.4 The GAN Architecture

GAN architecture diagrams available in Appendix 5.
The key features of the generator is this model first merges noise and label inputs, then creates a dense projection with batch normalization and LeakyReLU. There are then 2 Conv1D with batch normalization and LeakyReLU and one final Conv1D output layer with tanh activation. The model uses Conv1D because the structure is inherently 1-dimensional along the time axis.

The discriminator model concatenates the time series input and the label map then has 3 convolutional layers each with LeakyReLU. The neurons in each block increases from 32 -> 128. There is then a flatten step to flatten to a vector before a final dense output layer with sigmoid activation.

2.5 Key code

```
def build_generator(n_timesteps, n_channels, latent_dim=64):
    noise_input = layers.Input(shape=(latent_dim,), name='noise_input')
    label_input = layers.Input(shape=(1,), dtype='int32', name='class_label')

    label_embedding = layers.Embedding(input_dim=3,
output_dim=latent_dim)(label_input)
    label_vec = layers.Flatten()(label_embedding)
    merged = layers.Concatenate()([noise_input, label_vec])

    # add batch normalization and leakyReLU
    body = layers.Dense(n_timesteps * 64, use_bias=False, name='gen_dense')(merged)
    body = layers.BatchNormalization(name='gen_batch_norm_1')(body)
    body = layers.LeakyReLU()(body)
    body = layers.Reshape((n_timesteps, 64))(body)

    body = layers.Conv1D(64, kernel_size=5, padding='same', use_bias=False,
name='gen_conv_1')(body)
    body = layers.BatchNormalization(name='gen_batch_norm_2')(body)
    body = layers.LeakyReLU()(body)

    body = layers.Conv1D(32, kernel_size=3, padding='same', use_bias=False,
name='gen_conv_2')(body)
    body = layers.BatchNormalization(name='gen_batch_norm_3')(body)
    body = layers.LeakyReLU()(body)

    out = layers.Conv1D(n_channels, kernel_size=3, padding='same',
activation='tanh')(body)

    return tf.keras.Model([noise_input, label_input], out, name='generator')


def build_discriminator(n_timesteps, n_channels):
    ts_input = layers.Input(shape=(n_timesteps, n_channels))
    label_input = layers.Input(shape=(1,))

    label_embedding = layers.Embedding(input_dim=3,
output_dim=n_timesteps)(label_input)
    label_map = layers.Reshape((n_timesteps, 1))(label_embedding)

    merged = layers.Concatenate(axis=2)([ts_input, label_map])

    # add batch normalization and use leakyReLU in discriminator
    body = layers.Conv1D(32, kernel_size=5, strides=2, padding='same',
name='disc_conv_1')(merged)
    body = layers.LeakyReLU()(body)
    body = layers.Conv1D(64, kernel_size=5, strides=2, padding='same',
name='disc_conv_2')(body)
    body = layers.LeakyReLU()(body)
    # replace global average pooling with strided convoluted layer
    body = layers.Conv1D(128, kernel_size=5, strides=2, padding='same',
name='disc_conv_3')(body)
    body = layers.LeakyReLU()(body)
    # add flatten to match expected shape
    body = layers.Flatten(name='disc_flatten')(body)
    out = layers.Dense(1, activation='sigmoid')(body)

    return tf.keras.Model([ts_input, label_input], out, name='discriminator')
```
Existing hybrid model code can be seen in Appendix 6.

One difficult to capture aspect of this code is the significant refactoring that was done after one simple prompt. While working on the assignment my code became very complex with many code blocks making it difficult to correctly run each required code block after making changes. I pasted all the required code with the following prompt and was given a very simple refactoring that vastly simplified my code without losing any functionality.

Prompt:
My code is becoming quite messy now. How do I now retrain the GAN and what functions do I need to run every time in order to test the new changes? I have only added the new generate synthetic dataset function. (all code removed for this example)

Link to LLM chat: https://chatgpt.com/share/69252d4b-4c50-8006-ace4-7b330bafc773

2.6 URL of your Colab code

Github link: https://github.com/btorok-bu/METCS767_hw4

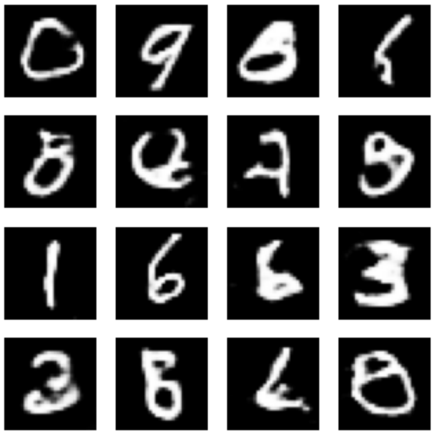
References

Show that you used a wide variety of resources by listing them below and clearly indicating in the body above where you used. Make sure to use proper referencing in your paper. We suggest using APA format, but other formats are fine as long as they clearly distinguish your work from work of others in your response. In general, observe the stated plagiarism rules.

[1] Géron, A. (2022). *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems* (3rd ed.). O'Reilly Media.

Appendix 1

Image using RMSprop after 40 epochs:



GAN model for final implementation:

```
EPOCHS = 40

cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)
def make_generator_model():
    weight_init = tf.random_normal_initializer(mean=0.0, stddev=0.02)
    model = tf.keras.Sequential()
    # 100 random values will be input, connected densely to 7*7*256 nodes
    # 256 = grayness measure
    model.add(layers.Dense(7*7*256, use_bias=False, input_shape=(100,),
                           kernel_initializer=weight_init))
    model.add(layers.Reshape((7, 7, 256)))
    model.add(layers.BatchNormalization()) # normalize each sample
    model.add(layers.ReLU()) # move
    assert model.output_shape == (None, 7, 7, 256)

    # padding='same' ensures output has same dimensions as input
    # 128 kernels
    # activation relu except for output
    model.add(layers.Conv2DTranspose(128, (5, 5), strides=(1, 1),
                                     padding='same', use_bias=False,
                                     kernel_initializer=weight_init))
    assert model.output_shape == (None, 7, 7, 128)
    model.add(layers.BatchNormalization())  # to mean 0, variance 1
    model.add(layers.ReLU())

    model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2),
                                     padding='same', use_bias=False,
                                     kernel_initializer=weight_init))
    assert model.output_shape == (None, 14, 14, 64)
    model.add(layers.BatchNormalization())
    model.add(layers.ReLU())
    model.add(layers.Conv2DTranspose(1, (5, 5), strides=(2, 2),
                                     padding='same', use_bias=False,
                                     activation='tanh',
                                     kernel_initializer=weight_init))
    assert model.output_shape == (None, 28, 28, 1)  # MNIST dimensions

    return model


def make_discriminator_model():
    # added weight init to all layers
    weight_init = tf.random_normal_initializer(mean=0.0, stddev=0.02)
    model = tf.keras.Sequential()
    model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same',
                            input_shape=[28, 28, 1],
                            kernel_initializer=weight_init))
    model.add(layers.LeakyReLU())
    # increased dropout
    model.add(layers.Dropout(0.4))

    model.add(layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same',
                            kernel_initializer=weight_init))
    # added batch normalization
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Flatten())
    model.add(layers.Dense(1))

    return model


generator = make_generator_model()
discriminator = make_discriminator_model()
# use RMSprop optimizer
gen_opt = tf.keras.optimizers.RMSprop(learning_rate=1e-4)
disc_opt = tf.keras.optimizers.RMSprop(learning_rate=1e-4)

train(train_dataset, EPOCHS, generator, discriminator, gen_opt, disc_opt)
```

Appendix 2:

Input/Output 1
Input:

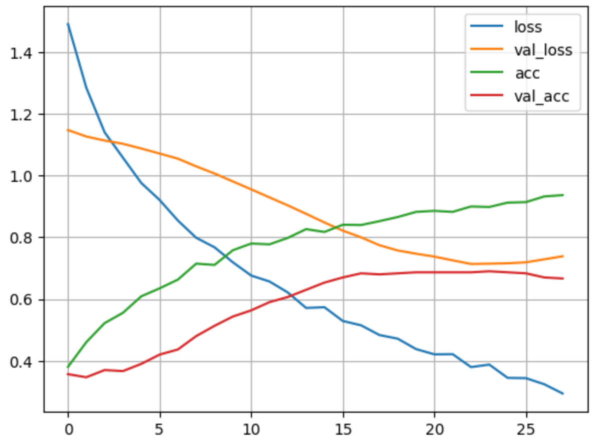
Baseline statistics before adding of hybrid convoluted model before using GAN to synthesize neutral time series data:

```
Macro F1: 0.6967
Weighted F1: 0.6999
              precision    recall  f1-score   support

       under       0.85      0.68      0.75       105
     neutral       0.51      0.71      0.59        94
        over       0.82      0.68      0.75       101

    accuracy                           0.69       300
   macro avg       0.72      0.69      0.70       300
weighted avg       0.73      0.69      0.70       300

[[71 34  0]
 [12 67 15]
 [ 1 31 69]]
```



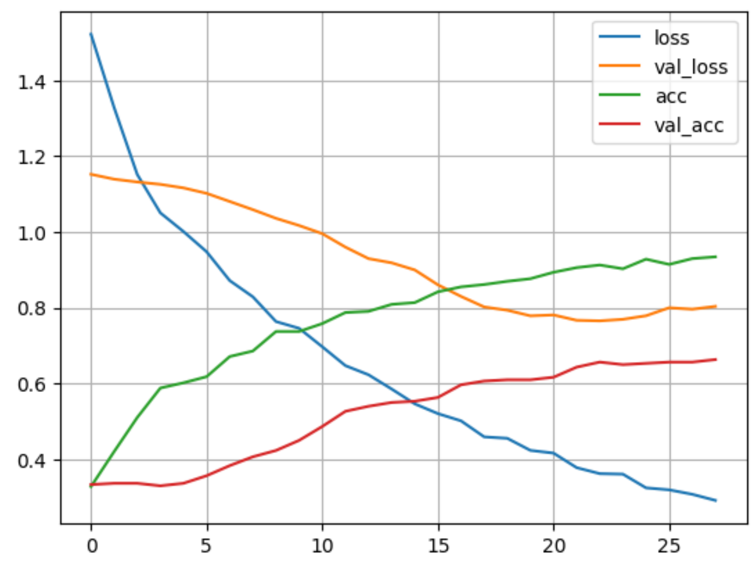
Output: Performance statistics of hybrid model after adding GAN to synthesize neutral time series data:

```
Macro F1: 0.6650
Weighted F1: 0.6671
              precision    recall  f1-score   support

       under       0.84      0.62      0.71       105
     neutral       0.48      0.82      0.60        94
        over       0.90      0.54      0.68       101

    accuracy                           0.66       300
   macro avg       0.74      0.66      0.66       300
weighted avg       0.75      0.66      0.67       300

[[65 40  0]
 [11 77  6]
 [ 1 45 55]]
```



Appendix 3:

Input/Output 2

Input: Sample of 5 days of time series data in the channels (limited to 3 of 8 channels for this example)

```
REAL sample:
       ch_0      ch_1      ch_2
0 -0.980793 -0.980470 -0.980694
1 -0.980544 -0.981294 -0.980630
2 -0.981214 -0.981085 -0.981078
3 -0.981108 -0.981051 -0.981025
4 -0.981057 -0.980599 -0.980808
```

Output: Sample of 5 days of channel data from the generated synthetic data:

```
SYNTHETIC sample:
       ch_0      ch_1      ch_2
0 -0.986915 -0.986843 -0.978241
1 -0.989752 -0.993248 -0.986490
2 -0.980104 -0.956898 -0.964601
3 -0.991609 -0.981395 -0.995365
4 -0.997407 -0.995772 -0.993446
```

Appendix 4:

Input/Output 3:

Input: Real and synthetic data in each channel (limited to 3 of 8 channels shown here)

```
       source  sample_idx  timestep      ch_0      ch_1      ch_2      ch_3
0        real           0         0 -0.913467 -0.914231 -0.913694 -0.914003
1        real           0         1 -0.916439 -0.915803 -0.916380 -0.915670
2        real           0         2 -0.914886 -0.913173 -0.914936 -0.911914
3        real           0         3 -0.914136 -0.914179 -0.914321 -0.914242
4        real           0         4 -0.913926 -0.912971 -0.913672 -0.913959
5        real           1         0 -0.999162 -0.998915 -0.998945 -0.999091
6        real           1         1 -0.998912 -0.998976 -0.998933 -0.998994
7        real           1         2 -0.998945 -0.998973 -0.998942 -0.998937
```

```
8        real         1         3 -0.998799 -0.999024 -0.998821 -0.999157
9        real         1         4 -0.999032 -0.999159 -0.999050 -0.999117
10  synthetic         0         0 -0.994939 -0.991131 -0.987608 -0.998093
11  synthetic         0         1 -0.998640 -0.999456 -0.995199 -0.997677
12  synthetic         0         2 -0.997330 -0.998317 -0.992445 -0.997511
13  synthetic         0         3 -0.994992 -0.994323 -0.993961 -0.995969
14  synthetic         0         4 -0.996548 -0.993027 -0.991621 -0.996631
15  synthetic         1         0 -0.994556 -0.989446 -0.984961 -0.996153
16  synthetic         1         1 -0.998265 -0.998842 -0.993744 -0.997937
17  synthetic         1         2 -0.996993 -0.998001 -0.993845 -0.997346
18  synthetic         1         3 -0.995373 -0.991807 -0.994566 -0.996189
19  synthetic         1         4 -0.994895 -0.987209 -0.991927 -0.996477
```

Output: The absolute difference in the mean and standard deviation between real and synthetic data for each class. This metric is used to determine how well the GAN is synthesizing realistic data. From these metrics we can see that the GAN is very effective at reproducing data as the absolute differences are small for all classes.

```
   class  mean_abs_mean_diff  mean_abs_std_diff
0      0            0.023598           0.075666
1      1            0.011188           0.019955
2      2            0.014992           0.042008
```

Appendix 5:

Generator diagram:

```
==============================
     GENERATOR (G)
==============================

 Inputs:
   - noise_input : (latent_dim,)
   - class_label : (1,) → Embedding → Flatten

 Step 1: Label Conditioning
   class_label
        │
        └──> Embedding(3 → latent_dim)
               │
               └──> Flatten → label_vec

 Step 2: Merge noise and label
   Concatenate([noise_input, label_vec]) → merged

 Step 3: Dense Projection
   Dense(n_timesteps * 64, use_bias=False)
   BatchNorm
   LeakyReLU
   Reshape → (n_timesteps, 64)
 Step 4: Convolution Blocks
   Conv1D(64, kernel_size=5, padding='same', use_bias=False)
   BatchNorm
   LeakyReLU

   Conv1D(32, kernel_size=3, padding='same', use_bias=False)
   BatchNorm
   LeakyReLU

 Step 5: Final Output Layer
   Conv1D(n_channels, kernel=3, padding='same', activation='tanh')

 Output:
   generated_timeseries → shape = (n_timesteps, n_channels)


Discriminator diagram:
================================
    DISCRIMINATOR (D)
================================

 Inputs:
   - ts_input : (n_timesteps, n_channels)
   - class_label : (1,) → Embedding → Reshape

 Step 1: Label Conditioning
   class_label
        │
        └──> Embedding(3 → n_timesteps)
                │
                └──> Reshape → (n_timesteps, 1)

 Step 2: Channel Concatenation
   Concatenate([ts_input, label_map], axis=channel)
     → shape = (n_timesteps, n_channels + 1)

 Step 3: Convolution Blocks (Downsampling)
   Conv1D(32, kernel=5, stride=2, padding='same')
   LeakyReLU

   Conv1D(64, kernel=5, stride=2, padding='same')
   LeakyReLU

   Conv1D(128, kernel=5, stride=2, padding='same')
   LeakyReLU
```

Step 4: Flatten
  Flatten → vector

Step 5: Output
  Dense(1, activation='sigmoid')
    → probability(real)

Overall GAN workflow:

```
        +--------------------+
        |    noise (z)       |
        +--------------------+
                  |
        +--------------------+
        |    class label y   |
        +--------------------+
                  |
                  ▼
           [ GENERATOR G ]
                  |
                  ▼
         generated_fake_timeseries
                  |
                  ▼
           [ DISCRIMINATOR D ]
                  |
      real_ts + real_labels ──────────▶ D (shared)
                  |
                  ▼
       Output: probability "real"
```

Appendix 6:

Existing hybrid model code:

```
layers = tf.keras.layers
metrics = tf.keras.metrics

# get the number of timesteps, channels, and static feature
n_timesteps = X_train_ts.shape[1]
n_channels = X_train_ts.shape[2]
n_static = X_train_static_proc.shape[1]


# time series branch
ts_input = tf.keras.Input(shape=(n_timesteps, n_channels), name="time_series_input")

# define the time series branch of the convoluted NN
# follow similar structure as the neural network trained in part 1
# train static branch
# add max pooling
ts_branch = layers.Conv1D(filters=32, kernel_size=5, activation='relu',
padding='causal',
                          name='convo_layer_1')(ts_input)
ts_branch = layers.BatchNormalization(name='convo_batch_norm_1')(ts_branch)
ts_branch = layers.MaxPooling1D(pool_size=2, name='max_pooling')(ts_branch)
ts_branch = layers.Conv1D(filters=16, kernel_size=3, activation='relu',
                          kernel_regularizer=tf.keras.regularizers.l2(1e-3),
padding='causal',
                          name='convo_layer_2')(ts_branch)
ts_branch = layers.BatchNormalization(name='convo_batch_norm_2')(ts_branch)
ts_branch = layers.GlobalAveragePooling1D(name='global_average_pooling')(ts_branch)
ts_branch = layers.Dropout(0.15, name='convo_dropout')(ts_branch)

# now train static branch
static_input = tf.keras.Input(shape=(n_static,), name='static_input')
# added batch normalization inside raw static input
static_branch = layers.BatchNormalization(name='static_batch_norm')(static_input)
# reduced dense layer neurons to 128
static_branch = layers.Dense(128, activation='relu',
                             kernel_regularizer=tf.keras.regularizers.l2(1e-4),
                             name='static_dense_layer')(static_branch)
static_branch = layers.Dropout(0.2, name='static_dropout')(static_branch)
# merge branches
merged = layers.Concatenate(name='merge')([ts_branch, static_branch])
# add dense layer after merging
merged = layers.Dense(64, activation='relu', name='merged_dense_layer')(merged)
merged = layers.Dropout(0.3, name='merged_dropout')(merged)
output = layers.Dense(3, activation='softmax', name='prediction')(merged)


model_conv_ts_static = tf.keras.models.Model(
    inputs=[ts_input, static_input],
    outputs=output,
    name="conv1d_ts_plus_static_classifier"
)


top2_acc_metric = tf.keras.metrics.SparseTopKCategoricalAccuracy(
    k=2,
    name="top2_accuracy"
)

# compile model
optimizer = tf.keras.optimizers.Adam(learning_rate=1e-4)
loss_fn   = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False)
```

```
model_conv_ts_static.compile(
    optimizer=optimizer,
    loss=loss_fn,
    metrics=[
        metrics.SparseCategoricalAccuracy(name='acc'),
        metrics.SparseTopKCategoricalAccuracy(k=2, name='top2_acc'),
        metrics.Precision(name='prec_under', class_id=0),
        metrics.Recall(name='rec_under', class_id=0),
        metrics.Precision(name='prec_neutral', class_id=1),
        metrics.Recall(name='rec_neutral', class_id=1),
        metrics.Precision(name='prec_over', class_id=2),
        metrics.Recall(name='rec_over', class_id=2),
    ]
)

# add neutral class weighting

class_weight = {
    0: 1.0,    # under
    1: 1.3,    # neutral (slightly up-weighted)
    2: 1.0     # over
}

early_stop = tf.keras.callbacks.EarlyStopping(
    monitor="val_loss",
    patience=5,
    restore_best_weights=True
)

# train model
history = model_conv_ts_static.fit(
    [X_train_ts, X_train_static_proc],  # both inputs
    y_train,
    validation_data=([X_test_ts, X_test_static_proc], y_test),
    epochs=30,
    batch_size=64,
    callbacks=[early_stop],
    class_weight=class_weight,
    verbose=2
)
```