Milestone 4

Varuni Gang, Andrew Greene, Scott Shaffer, Liang-Liang Zhang

April 25, 2017

```
library('ggplot2')
```

As always, our code is online at https://github.com/amgreene/cs109b

Our initial model

We started by creating a fairly "vanilla" model based on a pre-trained image model. The Python code is below.

Notes about the input data

Our image data, as described in previous milestones, has been resized to 48x48 and converted if necessary to RGB. We have divided the data into cohorts based on the last digit in the TMDb id, and for this milestone we are using cohort 0 as our training set and cohort 2 as our test set.

(This is a change from previous weeks, when we used cohort 6 as our test set; there were some data quality issues with cohort 6 and since we had not used cohort 2 for the Keras project, we considered it "untainted" for this week's purposes.)

For use with Keras, we have pickled a numpy array for each of the image data and genre label data for each cohort. Keras can use these in memmap mode, reducing the memory strain on our processing.

We also created "augmented" image data by reflecting each image left-to-right, although we have not yet had time to train on that data.

Multi-label becomes multi-model single-class

We decided early on that we wanted to properly approach this as a multi-label problem. However, because the labels are imbalanced at different rates, if we were to train a single neural network with one output for each label, it would be impossible to properly assign `class_weight` values – after all, we are *not* treating this as a multi-class problem! This therefore forces us to build a separate model for each genre, with a simple binary classifier. For that kind of model, `class_weight` works quite well.

Of course, that is the approach we had to take as well for our traditional models: The Random Forest, the Support Vector Classifier, and even Logistic Regression can't properly handle a multi-label problem with a single classifier.

In principle, we could try to share weights for one or both of our FC layers among our genre-specific models. In practice, we did not have time to pursue that approach.

Description of our initial model

We start with the `VGG16` model, setting the input size and omitting the top layer; we also freeze all its weights.

Next we take the output of the VGG16 model, add a 2-D global average pooling layer, and then add two fully-connected (FC) layers using `relu` activation. For our initial model, these are initialized using `he_normal` and the second layer has L1 regularization applied with $\lambda = 10^{-5}$.

We finally have a single-node output layer with `sigmoid` activation.

When compiling our initial model, we made the following choices:

Our optimizer is the `Adam()` optimizer, with its default parameters.

For our loss function, while we originally used the `binary_crossentropy` loss function provided by Keras, we decided to create our own `fscore_loss` method instead. Binary Cross-Entropy is, of course, the customary choice for this sort of case: We have a binary classifier, after all. But since we are trying to maximize the F-Score of our model, we thought it would be more appropriate to select as our objective function the very function we are trying to optimize! (To treat the F-Score as a loss, we subtract it from one.)

While in the early days of neural networks, it was necessary to choose an objective function that could be analytically differentiable in order to compute the gradiant and the Hessian, today we can use numerical differentiation and choose an objective function based on its usefulness, not just its nice mathematical properties.

(This also gives us the opportunity to explore writing custom loss functions in Keras, which is one of our "additional exploratory idea[s]" for this milestone.)

So, for our initial model, we chose our `fscore_loss` function.

We also include our `fscore` method in the list of metrics to report; this will populate the graphs presented later in this report.

In our framework code, we also choose the parameters to `model.fit` as follows.

The class weights are computed based on the class frequencies. Our initial batch size is 256, which means there are on the rough order of 100 batches per epoch for our data set. We train each model for 10 epochs, which is enough for us to get a rough sense of how each model compares with the others. (Since we are evaluating several hundred variations on these models, this was a time tradeoff. Some early experiments helped us choose 10 as being enough epochs *in most cases* to show whether a model was headed toward doing a reasonable job. We acklnowledge that there may be some models that will be short-changed with this approach, but our goal here is a general overview so we feel that is a worthwhile tradeoff. Later in this paper we explore a few cases of using more training epochs.)

Every one of our models is trained on cohort 0 and validated using cohort 2.

Most of the decisions described above will be tested by comparison with the alternatives later in this report. We do not assert that these choices are the best – they are relatively neutral choices which will let us explore the hyperparameter space from our "home base".

```
'''  To run this:

     Log in to your AWS instance
     mkdir data
     pip install h5py

     While the pip install is running, from your home machine
     scp data/*.npy to your AWS instance's data directory

     Back on your AWS instance:
     launch python
     copy and paste this code
'''
```

```python
import keras
from keras import backend as K
from keras import regularizers
from keras.applications.vgg16 import VGG16
from keras.datasets import mnist
from keras.layers import Dense, Activation, Conv2D, MaxPooling2D, Flatten, GlobalAveragePooling2D
from keras.models import Model, Sequential
from keras.optimizers import SGD, Adam
from keras.preprocessing import image

import numpy as np
import pandas as pd

# For the base models, we want to do all genres,
# but for some of our explorations, we want to skip
# those that are so rare that they can't be reasonably
# modeled:

skipping_lightweights = True

def do_one_genre(single_genre_index,
                 fc_size_1,
                 fc_size_2,
                 reg
):

    data = np.load('data/img_rgb_color_0x.npy',
                   mmap_mode='r')
    labels = np.load('data/genre_cohorts_0x.npy',
                     mmap_mode='r')[:,single_genre_index]
    m = labels.mean()
    if skipping_lightweights and m < 0.03:
        return
    class_weights = {0: 1/(1-m),
                     1: 1/m,
                     } # single-label

    validation_data = np.load('data/img_rgb_color_2x.npy',
                              mmap_mode='r')
    validation_labels = np.load('data/genre_cohorts_2x.npy',
                                mmap_mode='r')[:,single_genre_index]

    # We want to optimize fscore. So we'll need a method
    # that can be used both as a metric (so we can report
    # on it after each epoch) and as an objective function
    # (so we can try to optimize for it)
    #
    # K.sum(K.minimum(T1, T2) is an efficient way to
    # compute (T1 element-wise-and T2).sum() on Keras tensors
    #
    def fscore(y_true, y_pred):
        TruePos = K.sum(K.minimum(y_pred, y_true))
        TrueNeg = K.sum(K.minimum((1-y_pred), (1-y_true)))
    FalsePos = K.sum(K.minimum(y_pred, (1-y_true)))
    FalseNeg = K.sum(K.minimum((1-y_pred), y_true))
    Precision = TruePos / (TruePos + FalsePos + 1e-10)
    Recall = TruePos / (TruePos + FalseNeg + 1e-10)
    F_Score = 2 * (Precision * Recall) / (Precision + Recall)
    return F_Score

# We want to maximize fscore which means we want to
# minimize -fscore
def fscore_loss(y_true, y_pred):
    return 1-fscore(y_true, y_pred)

# ==== BEGIN MODEL DEFINITION =====
#
# CNN Based on pre-trained
#

# Start with VGG16
base_model = VGG16(weights='imagenet',
                   input_shape=(48, 48, 3),
                   include_top=False)
# Freeze it
for layer in base_model.layers:
    layer.trainable = False

# Next section based on
# https://keras.io/applications/#usage-examples-for-image-classification-models

# Take the output features from the VGG16 model...
x = base_model.output

# ... add a pooling layer ...
x = GlobalAveragePooling2D()(x)

# ... and now put a couple of fully-connected layers
# on top of that ...
x = Dense(fc_size_1, activation='relu',
          kernel_initializer='he_normal',
          #kernel_regularizer=regularizers.l2(1e-6),
          #activity_regularizer=regularizers.l1(1e-6)
)(x)
x = Dense(fc_size_2, activation='relu',
          kernel_initializer='he_normal',
          # kernel_regularizer=regularizers.l2(1e-6),
          activity_regularizer=regularizers.l2(reg)
)(x)

# ... and a logistic layer to extract our prediction
predictions = Dense(1, activation='sigmoid')(x)

# this is the model we will train
model = Model(input=base_model.input, output=predictions)
model.compile(optimizer=keras.optimizers.Adam(), # Try various ones
                  # loss='binary_crossentropy',
                  loss=fscore_loss,
                  metrics=['binary_accuracy', fscore])
    model.summary()

    # now train it

    history = model.fit(data,      # training data
                        labels,    # training labels
                        class_weight = class_weights,
                        batch_size=256,
                        epochs=10,
                        verbose=1,
                        validation_data=
                        (validation_data,
                         validation_labels),
                        )

    # Different variations of this script stash
    # different metadata in the filename
    lr_str = ('%.1e' % (reg)).replace('.', '_').replace('-', '_')

    basename = 'model-%d-%d-%d-l2_%s' % (
        single_genre_index,
        fc_size_1,
        fc_size_2,
        lr_str
    )
    # model.save(basename + '.h5')

    o = open(basename + '-history.txt', 'w')
    o.write(str(history.history) + '\n')

    # class weight took care of imbalance
    Predictions = (model.predict(data) >= .5).flatten()
    TruePos = (Predictions & labels).sum()
    TrueNeg = ((1-Predictions) & (1-labels)).sum()
    FalsePos = (Predictions & (1-labels)).sum()
    FalseNeg = ((1-Predictions) & labels).sum()

    # Print confusion matrix
    o.write('Train conf matrix: %d %d %d %d\n' %
            (TruePos, TrueNeg, FalsePos, FalseNeg,))

    Precision = TruePos / (TruePos + FalsePos + 1e-10)
    Recall = TruePos / (TruePos + FalseNeg + 1e-10)
    F_Score = 2 * (Precision * Recall) / (Precision + Recall)

    o.write('Train p r f: %f %f %f\n' % (Precision, Recall, F_Score))

    # Compute F-score using validation data
    # class weight took care of imbalance
    Predictions = (model.predict(validation_data) >= .5).flatten()
```

```
    TruePos = (Predictions & validation_labels).sum()
    TrueNeg = ((1-Predictions) & (1-validation_labels)).sum()
    FalsePos = (Predictions & (1-validation_labels)).sum()
    FalseNeg = ((1-Predictions) & validation_labels).sum()

    # Print confusion matrix
    o.write('Train conf matrix: %d %d %d %d\n' %
            (TruePos, TrueNeg, FalsePos, FalseNeg,))

    Precision = TruePos / (TruePos + FalsePos + 1e-10)
    Recall = TruePos / (TruePos + FalseNeg + 1e-10)
    F_Score = 2 * (Precision * Recall) / (Precision + Recall)

    o.write('Train p r f: %f %f %f\n' %
            (Precision, Recall, F_Score))

    o.close()

# Loop over all FC combos
for fc1_size in (256, 1024, 2048):
    for fc2_size in (256, 64, 1024):
        if fc1_size == 1024 and fc2_size == 256:
            continue # already did those by accident
        for single_genre_index in range(18):
            do_one_genre(single_genre_index, fc1_size, fc2_size, 1e-5)

# Loop over regularization lambdas:
for reg in (1e-5, 1e-6, 1e-4, 1e-2, 1e-3):
    for single_genre_index in range(18):
        do_one_genre(single_genre_index,
                     256,
                     256,
                     reg
        )
```

Our evaluation framework

After we created the basic Python script above, each of us took the hyperparameters for which we are responsible and created our own variant of it. The framework iterates over the combination of hyperparameters, creates, trains, and evaluates the resulting model, and writes the history and result to a text file. We collect those text files from our various AWS instances and run an extraction script to create a `tsv` file suitable for bringing into this Rmd notebook for exploration and explanation:

```
''' Convert history as captured in the txt files
    into a tidy tsv suitable for ggplot in our report
'''

import os
import os.path
import re

exception_messages = []
```

```
o = open('model_perf.tsv', 'w')
o.write('\t'.join([
    'Regularizer',
    'FC1',
    'FC2',
    'genre',
    'variant',
    'epoch',
    'train_loss',
    'train_accuracy',
    'train_fscore',
    'test_loss',
    'test_accuracy',
    'test_fscore'
    ]) + '\n')

genre_names= [
    # missing 2 for now
    'Animation',
    'Comedy',
    'Crime',
    'Documentary',
    'Drama',
    'Family',
    'Fantasy',
    'Foreign',
    'History',
    'Horror',
    'Music',
    'Mystery',
    'Romance',
    'Science_Fiction',
    'TV_Movie',
    'Thriller',
    'War',
    'Western'
]

nan = 'NaN'

for filename in os.listdir('model_logs'):
    if 'history' not in filename:
        continue
    with open(os.path.join('model_logs', filename), 'r') as f:
        try:
            line_1 = f.readline()
            history = eval(line_1.split('Train conf')[0])
            FC1 = 256  # unless overridden
            FC2 = 256  # unless overridden
            if 'BCE-NoReg' in filename:
                (genre_num, FC1, FC2) = re.search(
                    r'model-(\d+)-(\d+)-(\d+)',
                    filename).groups()
            linreg = 'NoReg'
            variant = 'BCE loss No Reg'
        elif 'BCE' in filename:
            (genre_num, FC1, FC2) = re.search(
                r'model-(\d+)-(\d+)-(\d+)',
                filename).groups()
            linreg = 'L1=1e-5'
            variant = 'BCE loss'
        elif 'NoReg' in filename:
            (genre_num, FC1, FC2) = re.search(
                r'model-(\d+)-(\d+)-(\d+)',
                filename).groups()
            linreg = 'NoReg'
            variant = 'regularizer'
        elif '-l1' in filename:
            (genre_num, FC1, FC2, linreg) = re.search(
                r'model-(\d+)-(\d+)-(\d+)-(l1_[0-9_e]+)',
                filename).groups()
            linreg = linreg.replace('e_', 'e-')
            linreg = linreg.replace('_', '.')
            linreg = linreg.replace('l1.', 'L1=')
            linreg = linreg.replace('.0', '')
            linreg = linreg.replace('e-0', 'e-')
            linreg = linreg.replace('e+0', 'e+')
            variant = 'regularizer L1'
        elif '-l2' in filename:
            (genre_num, FC1, FC2, linreg) = re.search(
                r'model-(\d+)-(\d+)-(\d+)-(l2_[0-9_e]+)',
                filename).groups()
            linreg = linreg.replace('e_', 'e-')
            linreg = linreg.replace('_', '.')
            linreg = linreg.replace('l2.', 'L2=')
            linreg = linreg.replace('.0', '')
            linreg = linreg.replace('e-0', 'e-')
            linreg = linreg.replace('e+0', 'e+')
            variant = 'regularizer L2'
        elif 'scratch_' in filename:
            (genre_num, FC1, FC2) = re.search(
                r'scratch_model-(\d+)-(\d+)-(\d+)',
                filename).groups()
            linreg = 'L1=1e-5'
            variant = 'scratch'
        elif '_LRS_' in filename:
            (genre_num, FC1, FC2) = re.search(
                r'model-(\d+)-(\d+)-(\d+)_LRS_',
                filename).groups()
            linreg = 'L1=1e-5'
            variant = 'LR decay'
        elif '_RLROP_100' in filename:
            if 'drama' in filename:
                genre_num = 4
            elif 'comedy' in filename:
                genre_num = 1
                elif 'anima' in filename:
                    genre_num = 0
                else:
                    raise Exception("Unknown genre")
                FC1 = '256'
                FC2 = '256'
                linreg = 'L1=1e-5'
                variant = '100 epochs Reduce LR on Plateau'
            elif '_RLROP_' in filename:
                (genre_num, FC1, FC2) = re.search(
                    r'model-(\d+)-(\d+)-(\d+)_RLROP_',
                    filename).groups()
                linreg = 'L1=1e-5'
                variant = 'Reduce LR on Plateau'
            elif '-VGG16-pretrain-' in filename:
                (genre_num, frozen) = re.search(
                    r'model-(\d+)-VGG16-pretrain-(\d+)-history',
                    filename).groups()
                linreg = 'L1=1e-5'
                variant = 'VGG w/' + frozen + ' frozen layers'
            else:
                (genre_num, FC1, FC2) = re.search(
                    r'model-(\d+)-(\d+)-(\d+)',
                    filename).groups()
                if FC1 == '256' and FC2 == '1024':
                    continue # don't have a full set of these
                if FC1 == '1024' and FC2 == '4':
                    continue # don't have a full set of these
                linreg = 'L1=1e-5'
                variant = 'base'
            genre = genre_names[int(genre_num)]
            print FC1, FC2, genre
            for epoch in range(len(history['loss'])):
                o.write('\t'.join([str(v) for v in [
                    linreg,
                    FC1,
                    FC2,
                    genre,
                    variant,
                    epoch + 1,
                    history['loss'][epoch],
                    history['binary_accuracy'][epoch],
                    history['fscore'][epoch],
                    history['val_loss'][epoch],
                    history['val_binary_accuracy'][epoch],
                    history['val_fscore'][epoch],
                ]]) + '\n')
        except Exception as e:
            print filename, e
            exception_messages += [filename + ' ' + str(e),]

print "DONE!"
print exception_messages
```

```r
# Load the performance data
cnn_perf <- read.table('model_perf.tsv', header=T, sep='\t')
```

A note about excluded genres

Due to a programming error when generating our cohort labels, the two genres that are alphabetically first (`Action` and `Adventure`) were accidentally omitted. If time permits, we hope to restore them for Milestone 5.

We also noted, while assessing the performance of our models, that there are some genres whose frequency is so rare that the neural nets were simply unable to develop a model for them. This is stronger than the imbalanced data problem; it is an insufficiency with the *absolute numbers* of images available for training. It is possible that if we were to train on more cohorts, or if we used augmented images, we might be able to overcome this, but again we have had to prioritize our time.

Since these rare genres produce models whose F-Scores are meaningless (more formally, they have no true positives and therefore both their precision and recall are 0), we begin to omit them from our later explorations so as to not waste processing time.

In particular, any genre whose base rate in the training data is < 0.03 has been omitted from most of these comparisons. These are `Fantasy`, `Foreign`, `History`, `Mystery`, `TV_Movie`, `War`, and `Western`.

Here are the precomputed base rates for our training set:

```r
genre_names <- c(
    'Animation',
    'Comedy',
    'Crime',
    'Documentary',
    'Drama',
    'Family',
    'Fantasy',
    'Foreign',
    'History',
    'Horror',
    'Music',
    'Mystery',
    'Romance',
    'Science_Fiction',
    'TV_Movie',
    'Thriller',
    'War',
    'Western'
)
train.baserate <- c(0.04844114, 0.201396,   0.05262913,
                    0.09781294, 0.28129362, 0.04271754,
                    0.02815263, 0.02680316, 0.01703118,
                    0.06454165, 0.04648674, 0.02680316,
                    0.08115403, 0.03476035, 0.01447185,
                    0.07915309, 0.01675198, 0.01912517)
test.baserate <- c(0.04988222,  0.19611103, 0.05223777,
                   0.09824027,  0.28077225, 0.04447832,
                   0.02780472,  0.02886703, 0.01704309,
                   0.06872662,  0.04480163, 0.02706572,
                   0.08059674,  0.0322387 , 0.01404092,
                   0.08175142,  0.01708928, 0.0159346)
```

```r
baserates <- data.frame(genre=factor(genre_names,
                                     levels=levels(cnn_perf$genre)),
                        train=train.baserate,
                        test=test.baserate)
```

Evaluation of the initial model

Let's look at training and test F-scores. Where there is no visible line for part or all of the test or training data, that is because at that epoch the F-score is undefined because there are no true positives in the predictions. (See for example the `Animation` facet, which has no test line, or the `Mystery` facet, where even the training line has gaps.) These "deficient" genres are those discussed above, the ones that are so rare (base rate < 0.03) that we cannot meaningfuly measure the resulting models, rendering comparisons useless. (We include them here merely to illustrate that point.)

Dashed lines are the base rate for the training and test sets (if only the training set is visible, that is because it is overlapping the training set); this would be the F-score for a dummy model that randomly predicts `1` in the same proportion as the base rate. All of our initial CNN models exceed this baseline for both training and test data.

```r
fscore.plot <- function(FC1, FC2, LReg) {
  ggplot(cnn_perf[cnn_perf$FC1==FC1 &
                    cnn_perf$FC2==FC2 &
                    cnn_perf$variant=='base' &
                    cnn_perf$Regularizer==LReg
                  ,]) +
    geom_line(aes(x=epoch, y=train_fscore, color='Train')) +
    geom_line(aes(x=epoch, y=test_fscore, color='Test'))  +
    geom_hline(data=baserates,
               aes(yintercept=train, color='Train'),
               linetype="dashed") +
    geom_hline(data=baserates,
               aes(yintercept=test, color='Test'),
               linetype="dashed") +
    facet_wrap(~ genre) +
    scale_x_continuous(breaks=seq(0, 10, 2)) +
    labs(title=paste0('F-Score over 10 epochs; FC1=', FC1,
                      ' FC2=', FC2, ' ', LReg),
         y='F score')
}
```

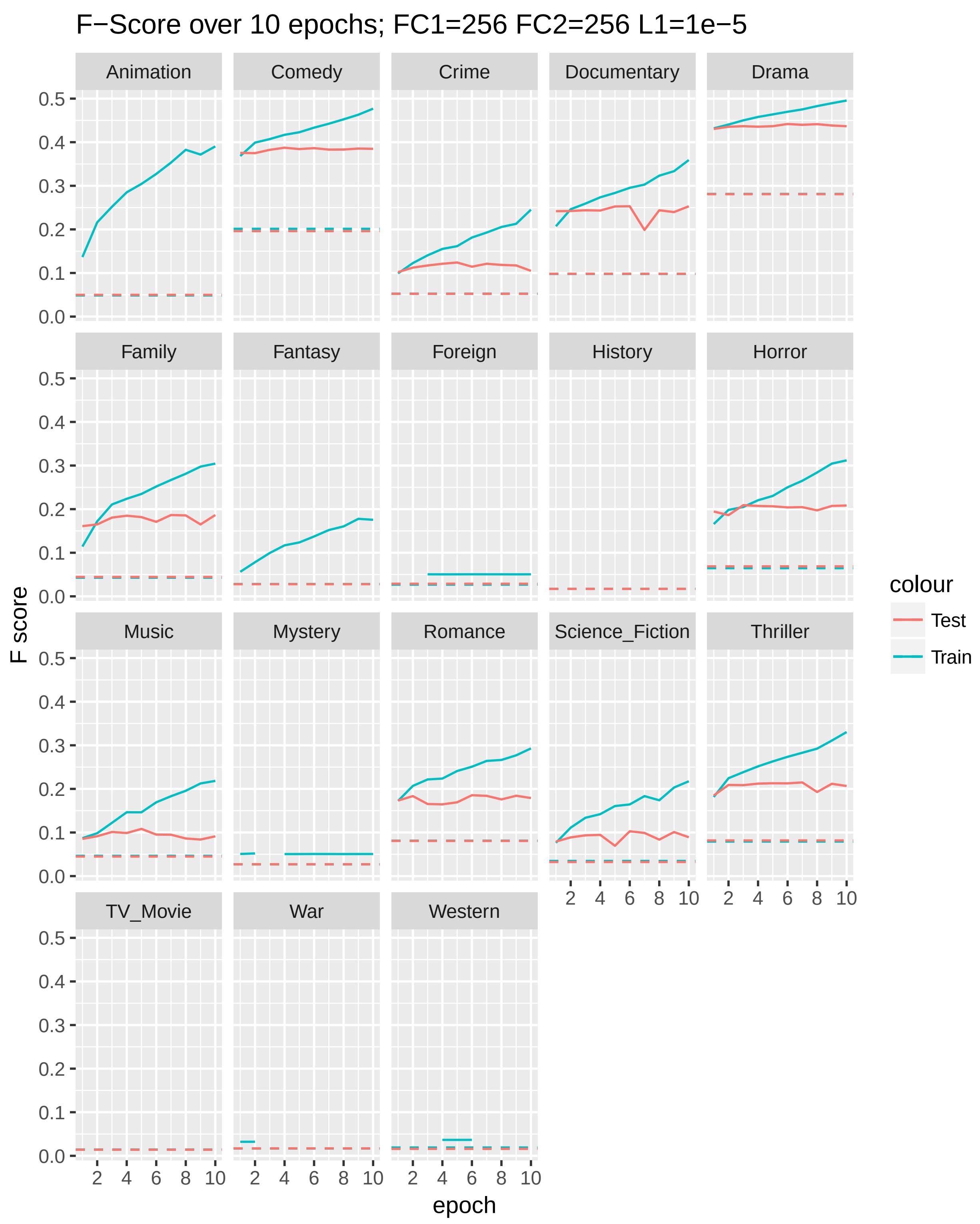
```r
# Function to display small multiples, so we can compare models that
# differ only in the value of a single hyperparameter

fscore.plot.grid <- function(variant, what.to.compare) {
  perf_data <- cnn_perf[cnn_perf$FC1==256 &
                          cnn_perf$FC2==256 &
                          cnn_perf$variant %in% c(variant, 'base') &
                          cnn_perf$genre %in%
                          baserates$genre[baserates$train>=0.03]
                  ,]
  # Exception: If we're comparing the FC sizes on the base model,
  # then that's the only criteron we care about (and we want to
  # include the rare genres as well)
  if (what.to.compare == 'FC1 + FC2') {
    perf_data <- cnn_perf[cnn_perf$variant=='base' & cnn_perf$epoch<11,]
  }
  ggplot(perf_data) +
    geom_line(aes(x=epoch, y=train_fscore, color='Train')) +
    geom_line(aes(x=epoch, y=test_fscore, color='Test'))  +
    geom_hline(data=baserates[baserates$train>=0.03,],
               aes(yintercept=train, color='Train'),  linetype="dashed") +
    geom_hline(data=baserates[baserates$train>=0.03,],
               aes(yintercept=test, color='Test'),  linetype="dashed") +
    facet_grid(as.formula(paste('genre ~', what.to.compare))) +
    scale_x_continuous(breaks=seq(0, 10, 2)) +
    scale_y_continuous(breaks=c(0, 0.5), limits = c(0, 0.6)) +
    labs(title=paste0('Comparing F-Score per ', what.to.compare),
         y='F score') +
    theme(strip.text.y = element_text(angle=0)) # unrotate facet labels
}
```

```r
fscore.plot(256, 256, 'L1=1e-5')
```

```
## Warning: Removed 4 rows containing missing values (geom_path).
```

```
## Warning: Removed 40 rows containing missing values (geom_path).
```



In all but the rare cases, the training F-Score increases with each epoch, while the test F-Score stays close to the value it had at the end of the first epoch. We make the following observations:

- The network is, in fact, “learning” to maximize the training F-Score (because the training F-Score is increasing with each epoch)
- The network is not overfitting (because if it were, the test F-Score would decrease)

- There is *some* value to the model (because the test F-Score is noticeably higher than the base rate, and is in general only slightly below the Random Forest and SVC F-Scores from our previous milestone), but it is not extreme (because the test F-Score does not exceed the traditional models and does not increase after the first epoch).

Comparisons

Our starting point is the VGG16-based CNN described above. For the next stage of our project, we decided to alter one hyperparameter at a time, evaluating the resulting model for several different values of that hyperparameter (at least one value other than our initial model, and when time permitted, more than one alternate value). We use the following R function to create a small-multiple plot for each hyperparameter, allowing us to compare for each viable genre how the model behaves for the first ten epochs as the hyperparameter is varied.

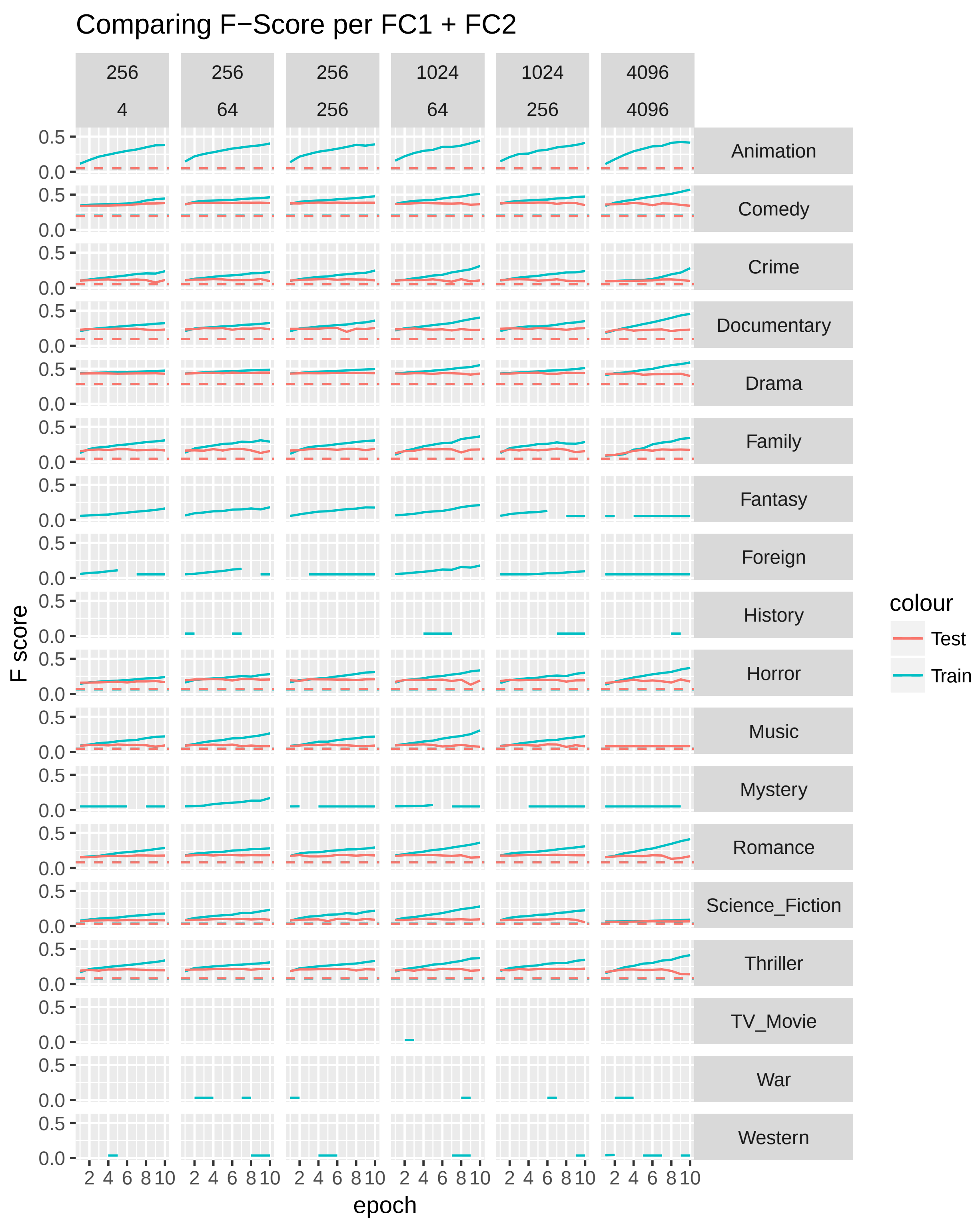
Given more time, we would like to try more values and allow the models to train over more than ten epochs.

Varying the number of FC nodes

The first hyperparameters we investigate are the number of nodes in our two fully-connected (FC) layers.

```r
fscore.plot.grid('FC', 'FC1 + FC2')
```

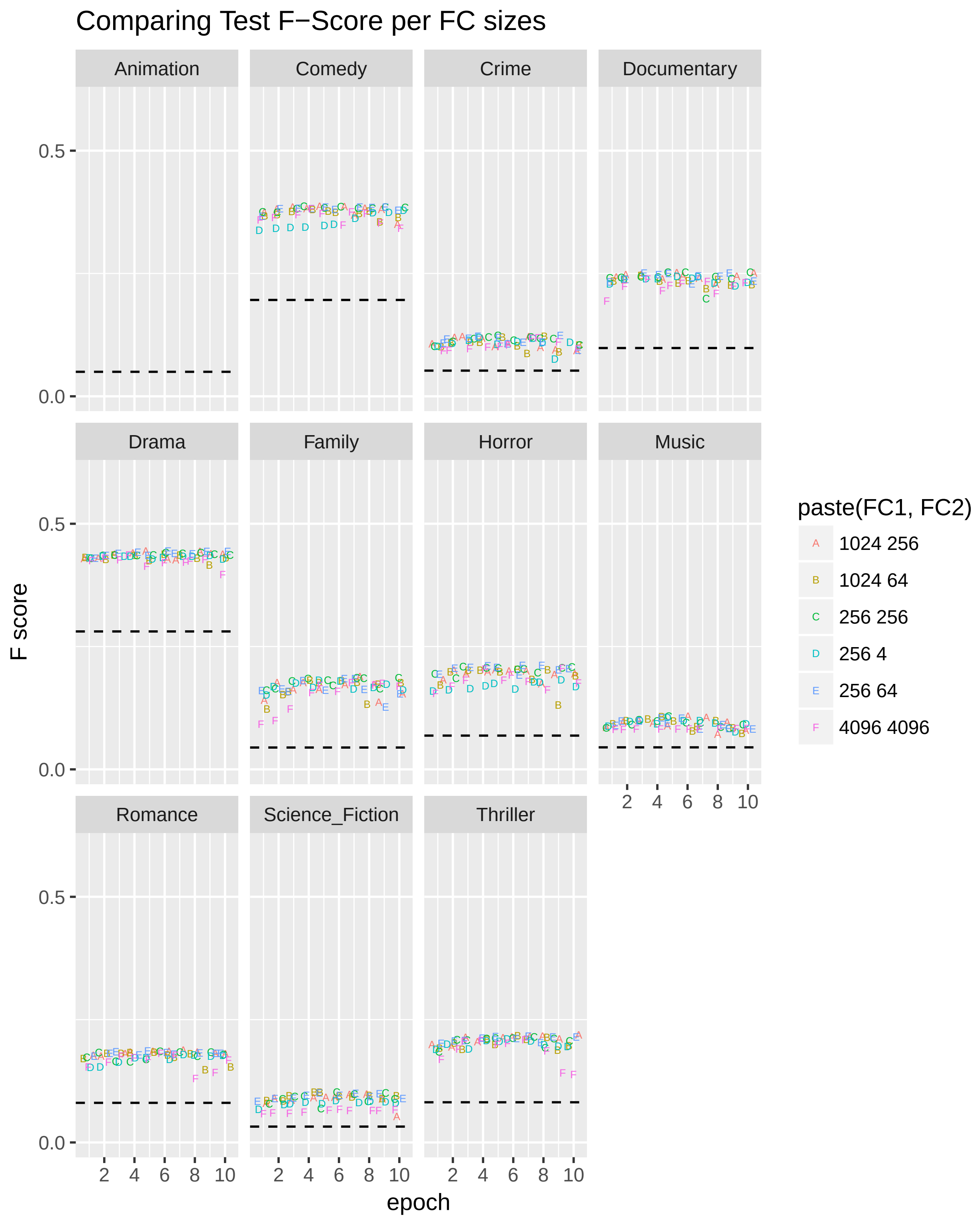
```
## Warning: Removed 240 rows containing missing values (geom_path).
```



Another view of the same data:

```r
perf_data <- cnn_perf[cnn_perf$variant=='base' &
                      cnn_perf$epoch < 11 &
                      cnn_perf$genre %in%
```

```
                          cnn_perf$genre[cnn_perf$test_fscore > 0]
                        ,
                        ]
ggplot(perf_data) +
  geom_jitter(aes(x=epoch, y=test_fscore,
                  shape=paste(FC1, FC2),
                  color=paste(FC1, FC2)),
              height=0) +
  scale_shape_manual(values=LETTERS) +
  geom_hline(data=baserates[baserates$train>=0.03,],
             aes(yintercept=test),  linetype="dashed") +
  facet_wrap(~genre) +
  scale_x_continuous(breaks=seq(0, 10, 2)) +
  scale_y_continuous(breaks=c(0, 0.5), limits = c(0, 0.6)) +
  labs(title='Comparing Test F-Score per FC sizes',
       y='F score') +
  theme(strip.text.y = element_text(angle=0)) # unrotate facet labels
```



We can see here the impact of the number of FC nodes. For relatively infrequent genres like `Comedy` and `Horror`, the `256x4` FC does notably worse at extracting the weak signal. For the relatively frequent genres like `Drama`, the effect of layer size is less pronounced. However, in general, the smaller networks seem to have underperformed throughout. Interestingly, a very large network (4096 nodes in both FC layers) does not improve things – it is rarely if ever the best performing of our models at any epoch, and in fact, for several genres (e.g., `Comedy`) the 4096x4096 model is the worst performer after ten epochs! This implies that the training set is overfitting, and increasing the λ for regularization might be helpful.

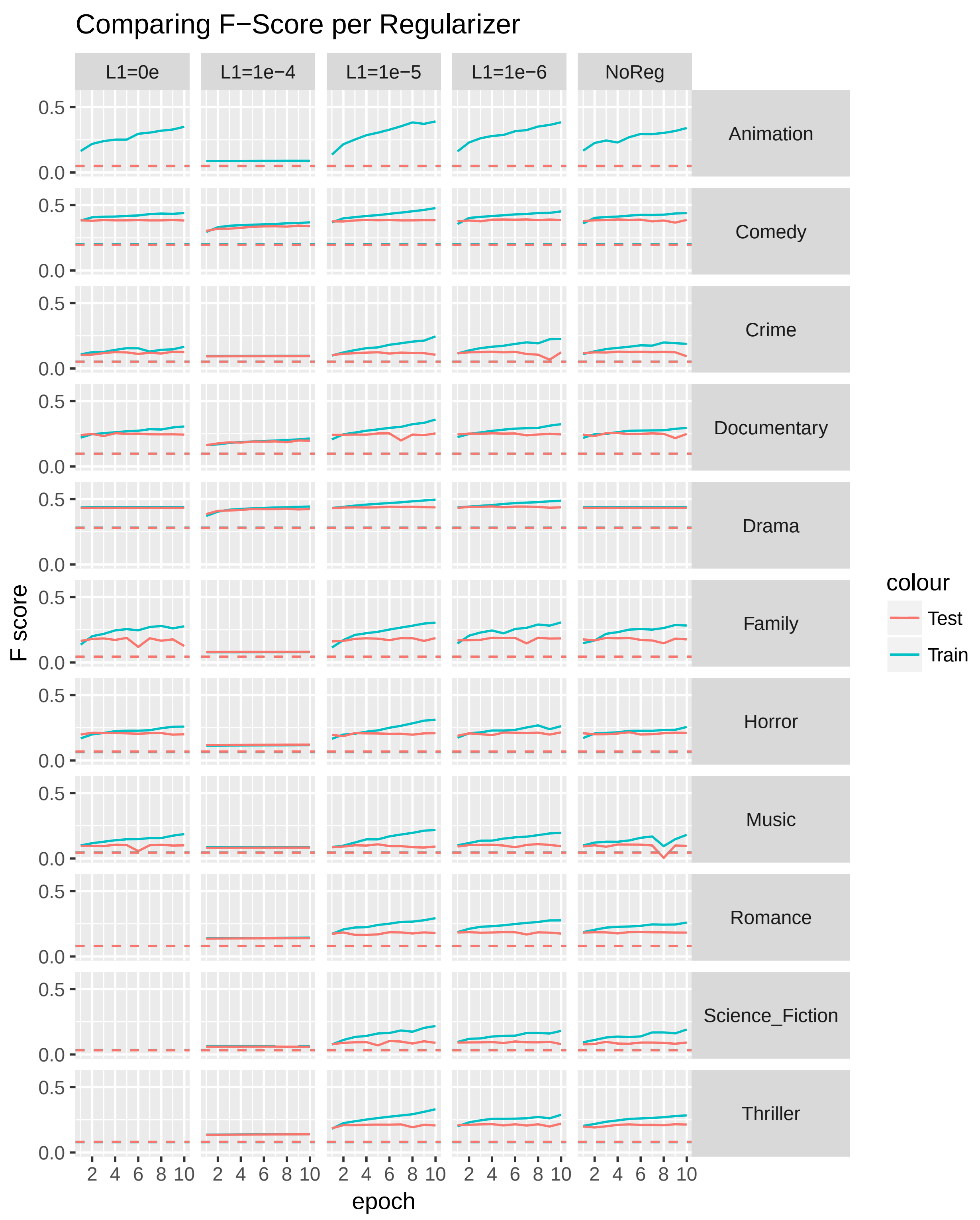
Regularization

It quickly became clear that we would need to regularize our models. With about 21,000 training observations and nearly 200,000 tunable parameters in our starting-point model, it did not take long to overfit and get "perfect" accuracy on the training set, with the test set score going way down.

We experimented first with L1 regularization, and in the following chart we compare the F-scores across several values:

```r
fscore.plot.grid(c('regularizer', 'regularizer L1'), 'Regularizer')
```

```
## Warning: Removed 50 rows containing missing values (geom_path).
```



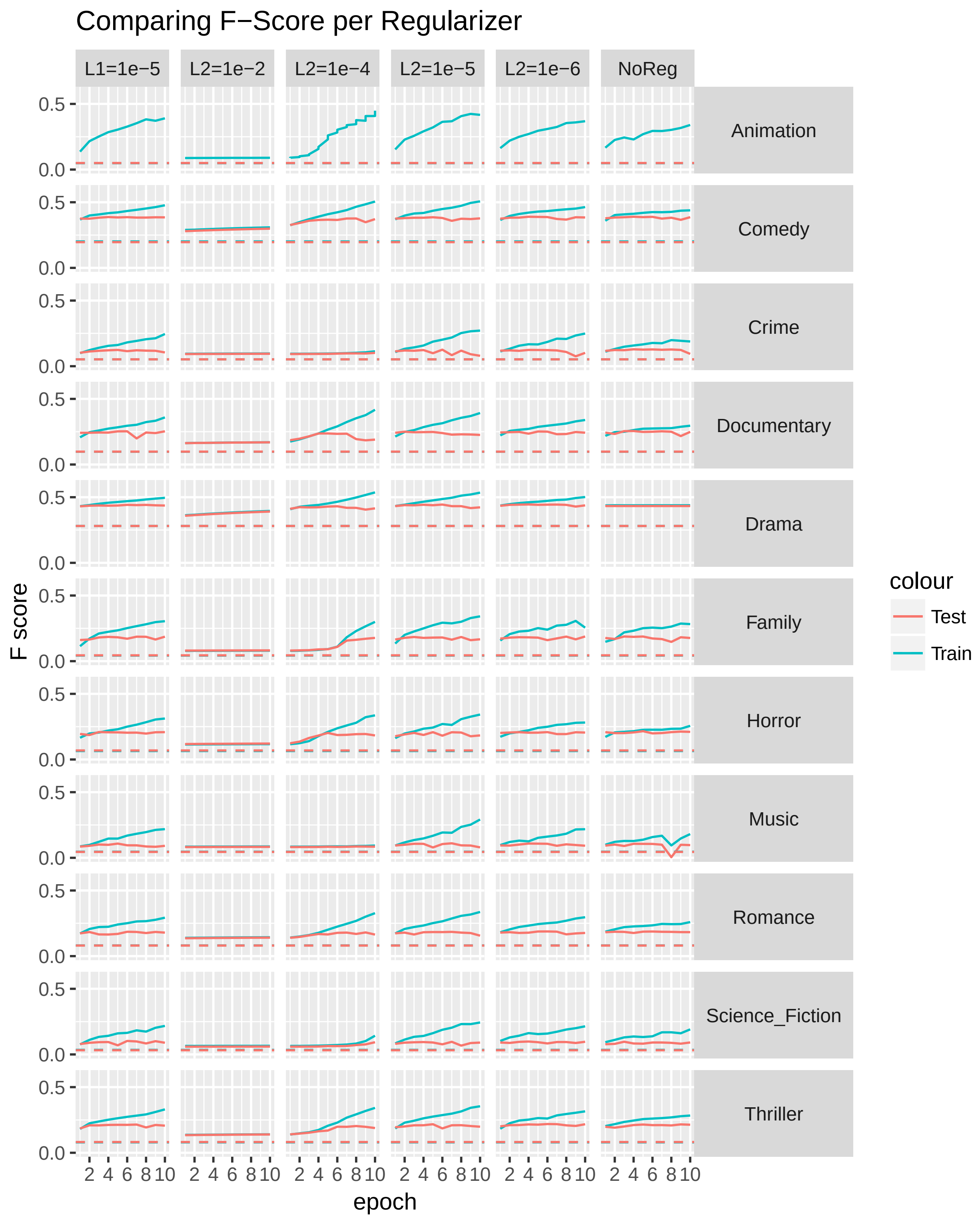
When the L1 parameter λ is as high as 10^{-4}, we can see that the F-Score stays flat at approximately the base rate and does not improve with each epoch. The regularization term dominates the data.

With $\lambda = 10^{-5}$ we see the training set improve, but not at the expense of the test set. Things remain reasonable at $\lambda = 10^{-6}$ as well. With an unregularized model, we see some distinct problems: For example, the `Music` genre at epoch 8 *underperforms* a stratified (random choice) model on the test data.

Now let's look at L2. (Note that for comparison with the initial model, this chart will also contain `L1=1e-5`)

```
fscore.plot.grid(c('regularizer', 'regularizer L2'), 'Regularizer')
```

```
## Warning: Removed 70 rows containing missing values (geom_path).
```



For $\lambda = 10^{-5}$, the L2 models perform slightly worse than L1 (looking at the test F score). We benefit more from pruning of connections than from limiting their strength. Given the large number of weights and the (relatively) small number of observations, this makes sense.

For $\lambda = 10^{-4}$, the L2 models appear to be in the "sweet spot". For several genres (most notably `Family` and `Horror`), both the train and test F-scores start off lower than they do in the models with less aggressive regularization; they rise together for several epochs (for `Family`, for example, this occurs between epochs 5 through 7); then the training score continues to rise quickly while the test score rises slowly or even falls. This is the classic behavior that indicates that the model is learning well, that the training results generalize to the test results, and that we can identify the point at which overfitting begins. The test score in most of these models plateaus at about the same level as for the less-regularized models, but it seems more trustworthy. It should be possible to fine-tune the λ value for the L2 case.

For $\lambda = 10^{-2}$ we see definitive flattening of the curve, as we saw for $\lambda = 10^{-4}$ in the L1 models.

Objective function

As explained above, we chose to write our own loss function, which corresponds to the F-Score:

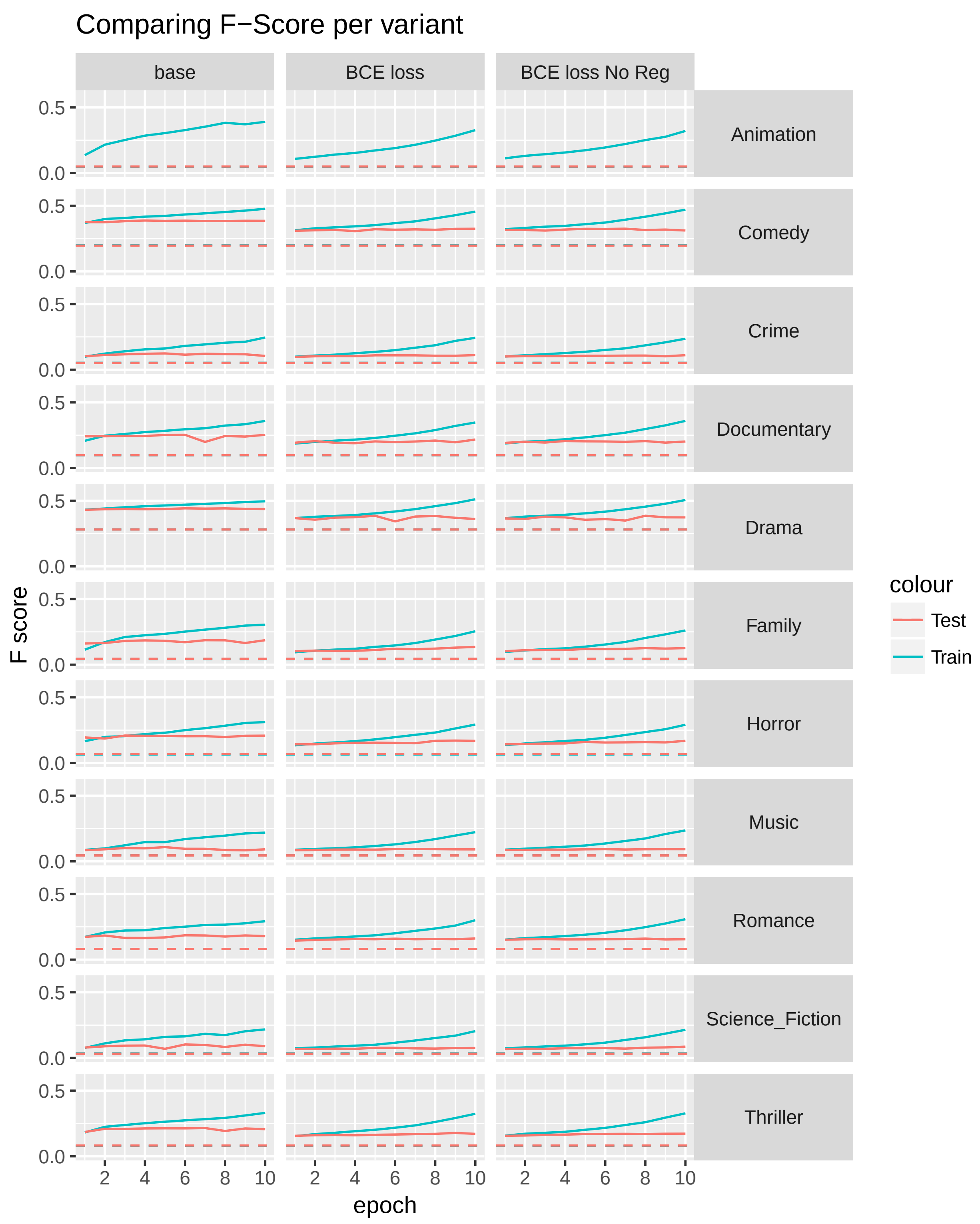
```
# We want to optimize fscore. So we'll need a method
# that can be used both as a metric (so we can report
# on it after each epoch) and as an objective function
# (so we can try to optimize for it)
def fscore(y_true, y_pred):
    # y_pred = K.transpose(y_pred)
    TruePos = K.sum(K.minimum(y_pred, y_true))
    TrueNeg = K.sum(K.minimum((1-y_pred), (1-y_true)))
    FalsePos = K.sum(K.minimum(y_pred, (1-y_true)))
    FalseNeg = K.sum(K.minimum((1-y_pred), y_true))
    Precision = TruePos / (TruePos + FalsePos + 1e-10)
    Recall = TruePos / (TruePos + FalseNeg + 1e-10)
    F_Score = 2 * (Precision * Recall) / (Precision + Recall)
    return F_Score

# We want to maximize fscore which means we want to
# minimize -fscore
def fscore_loss(y_true, y_pred):
    return 1-fscore(y_true, y_pred)
```

We wish to compare this loss function to the built-in `binary_crossentropy` loss function:

```
fscore.plot.grid(c('BCE loss', 'BCE loss No Reg'),
                 'variant')
```

```
## Warning: Removed 30 rows containing missing values (geom_path).
```



Clearly, the `fscore_loss` function gives better results:

- The F-scores at the end of the first epoch are higher than those for BCE for both the training and test sets
- The training F-scores increase faster under `fscore_loss` than under BCE for the first ten epochs. By the end of the tenth epoch, however, BCE training F-score has, in general, "caught up" with the

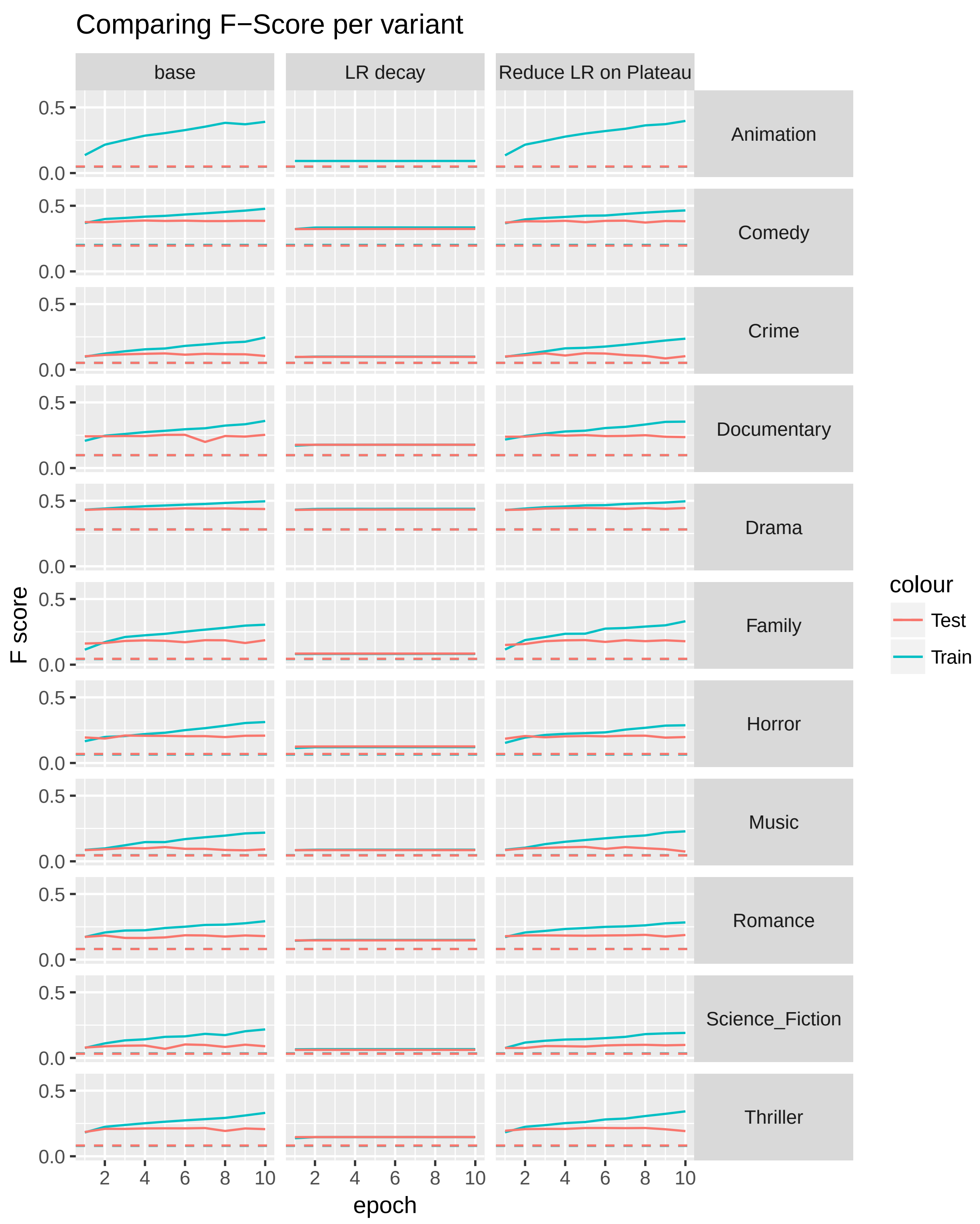
`fscore_loss` F-score. We believe that if we continued running, the BCE would continue to appear to improve, but with no improvement in the test F-score; that is, the BCE has started to overfit here.

- The test F-scores are flat for both loss functions, so the advantages listed above do not come at the cost of hurting the results on the test cohort, which is our true measure of success.

Learning Rate and Batch Size

```
fscore.plot.grid(c('LR decay', 'Reduce LR on Plateau'),
                 'variant')
```

```
## Warning: Removed 30 rows containing missing values (geom_path).
```



Learning rate is the most important parameter in the whole optimization process. Up until now we only had fixed learning rate, which is a reasonable choice but not the most suitable. We tried the following two approaches to tune this parameter.

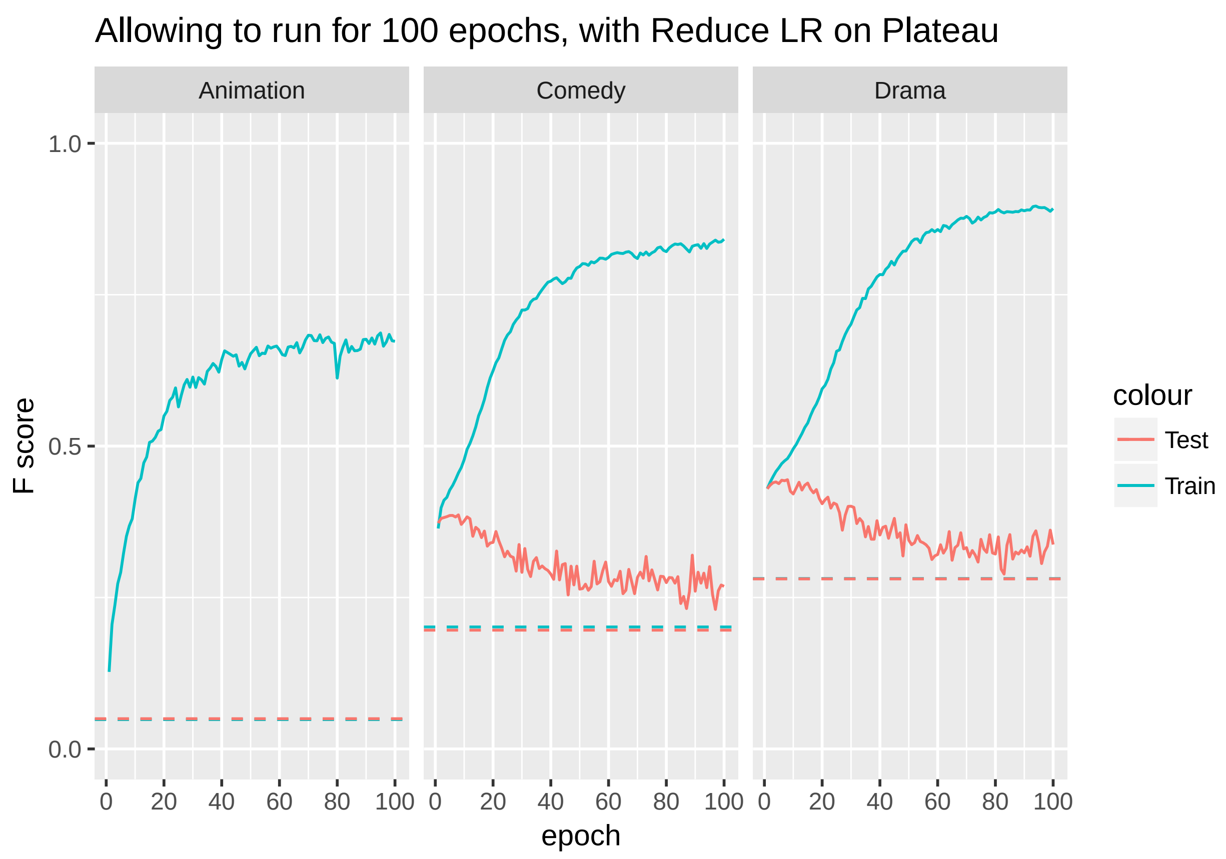
A) Learning rate decay: In this approach we started with a very large learning rate for initial epochs to get the filters (convolution layers) running and then decreased it slowly. Since we are running the model with only 10 epochs, we decided to set the patience counter at 5 epochs with a decrease in LR by `0.1 / (2. ** epoch)` on the rest of the epochs. From the above graph we can see that this did not not have any effect on either the training or test set. This is entirely plausible, as in this case the decrease in learning rate is solely based on the number of epochs and has not being decided by the training or test set accuracy or loss.

B) Reduce LR on Plateau: This is very simmilar to constant learning rate decay differing in how and when we choose to decay the learning rate. In this aproach we are constantly testing on the validation set and receiving feedback based on validation loss. This allows us to decide when (and by how much) to decrease the learning rate. We chose an aggresive patience of just 1 epoch and decreased the learning rate by a factor of .2 with a minimum lr of .001. As you can see from the above graph, this gives much better results than constant learning rate decay for almost all genres. This can be attributed to the fact that the learning rate is adjusted based on the feedback recieved from actual data (validation set).

Let's take the most promising of those, the *Reduce LR on Plateau* for the `Animation`, `Comedy`, and `Drama` genres, and let it run for 100 epochs:

```
perf_data <- cnn_perf[cnn_perf$variant=='100 epochs Reduce LR on Plateau'
                      ,]
ggplot(perf_data) +
  geom_line(aes(x=epoch, y=train_fscore, color='Train')) +
  geom_line(aes(x=epoch, y=test_fscore, color='Test'))  +
  geom_hline(data=baserates[baserates$genre %in% c('Animation', 'Comedy', 'Drama'),],
             aes(yintercept=train, color='Train'),  linetype="dashed") +
  geom_hline(data=baserates[baserates$genre %in% c('Animation', 'Comedy', 'Drama'),],
             aes(yintercept=test, color='Test'),  linetype="dashed") +
  scale_x_continuous(breaks=seq(0, 100, 20)) +
  scale_y_continuous(breaks=c(0, 0.5, 1.0), limits = c(0, 1.0)) +
  facet_grid(~ genre) +
  labs(title=paste0('Allowing to run for 100 epochs, with Reduce LR on Plateau'),
       y='F score')
```

```
## Warning: Removed 100 rows containing missing values (geom_path).
```



One can see from the above graph that the F-score approaches the theoretical maximum on the training set while for the test set, it decreases to just above the random-choice of F-score discussed previously. This is quite interesting as it shows that decreasing the learning rate over a larger number of epochs certainly provides a better model fit. However, in our case the model is defintely overfitting on the training set and underperforming on the test set. There is still much work to be done to identify the appropriate values for patience and factors. In addition, we think that using a stronger regularizer might improve performance of the model.

For this milestone we did not have the time to tune the batch size. After understanding the effect the number of epochs had on model fit, we believe that decreasing the batch size will likely remove some overfitting.

Starting from Scratch

For our "from Scratch" model we used a much simpler flow: "conv2D -> Maxpool -> flatten -> Dense (activation = relu) -> Dense (activation = sigmoid)". Hyper-parameters were chosen to correspond as closely as possible with our initial model, to facilitate comparison:

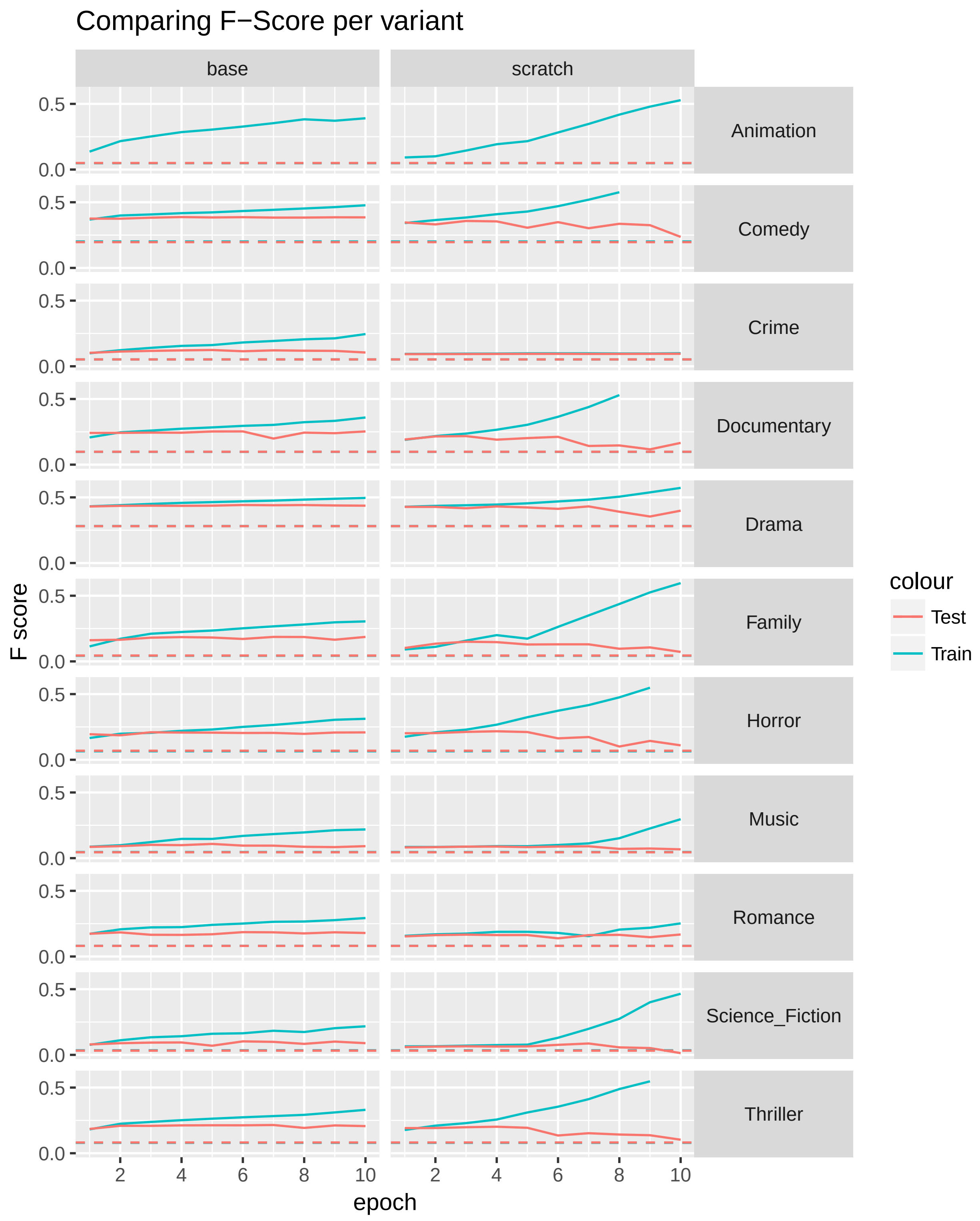
1. Set L1 kernel regularizer to `1e-5`
2. Set the number of filters in the Conv2D layer to 256, corresponding to the number of nodes in our base model's first FC layer
3. Set the number of nodes in our FC later to 256, corresponding to the size of our base model's second FC layer
4. Use F-score to evaluate performance of each model and use `fscore_loss` as our objective function
5. Use kernel initializer "`he_normal`"

```
model = Sequential()
model.add(Conv2D(fc_size_1,
                     kernel_size=(3, 3),
                     data_format='channels_last',
                     input_shape=(48, 48, 3),
                     activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Flatten())
    model.add(Dense(fc_size_2,
                    activation='relu',
                    kernel_initializer='he_normal',
                    activity_regularizer=regularizers.l1(1e-5)
                   ))
    model.add(Dense(1,
                    activation='sigmoid',
                    kernel_initializer='he_normal',
                    activity_regularizer=regularizers.l1(1e-5)
                   ))
    model.compile(optimizer=keras.optimizers.Adam(),
                  loss=fscore_loss,
                  metrics=['accuracy','binary_accuracy', fscore])
    model.summary()
```

```
fscore.plot.grid('scratch', 'variant')
```

```
## Warning: Removed 1 rows containing missing values (geom_path).

## Warning: Removed 20 rows containing missing values (geom_path).
```



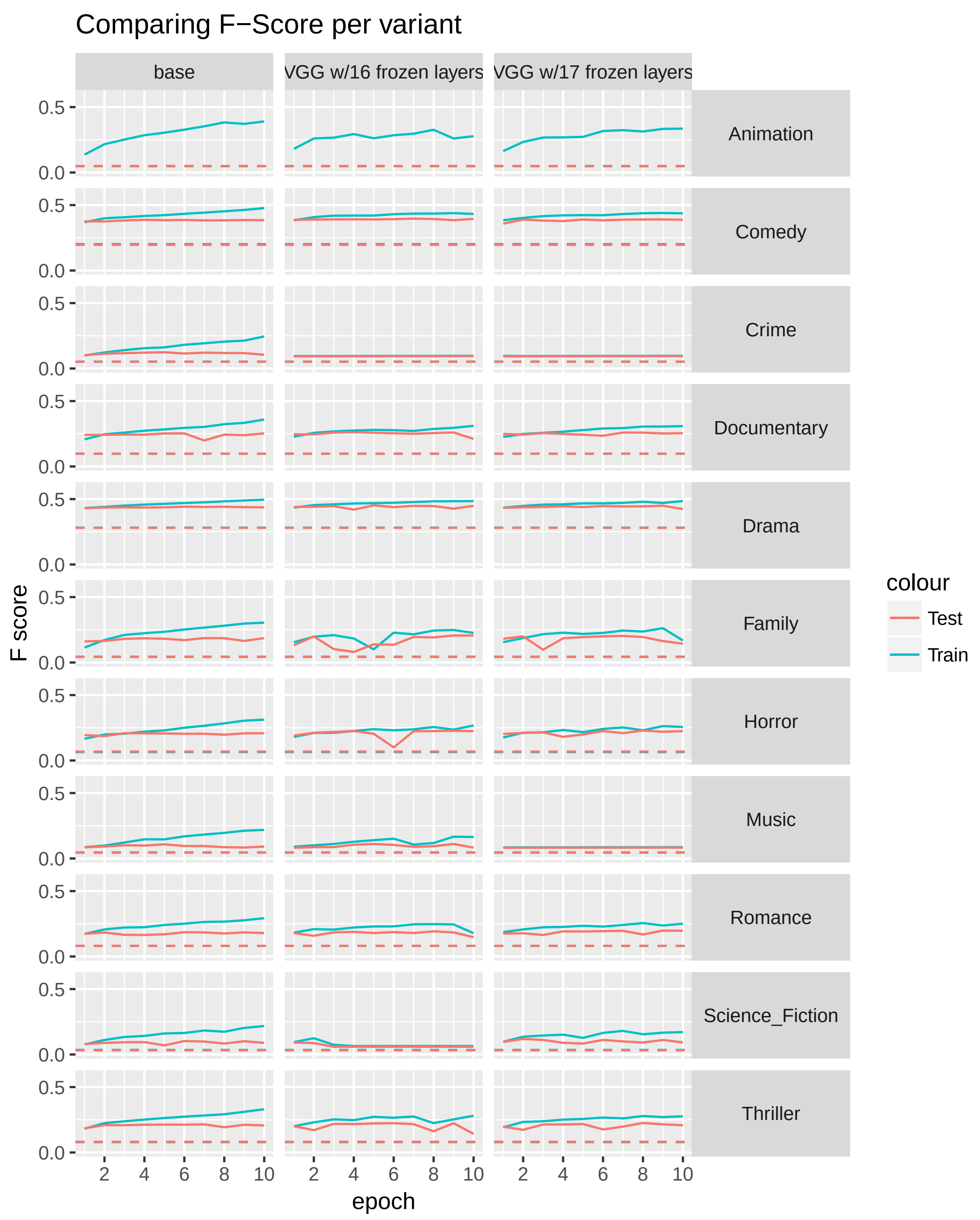
Performance comparison between base model and scratch model: * For most genres, except "Crime", the training F-scores keep increasing with epoch and the increase rate is higher than for the base model. By the end of the 10th epoch, the "from-scratch" model's training F-scores are higher than those of base model.
* The "from-scratch" model's test F-scores are at similar level to the initial model's for the first few epochs, and then decrease in most genres. This indicates over-fitting in the training dataset occurs after a few epochs.

The final test F-scores after 10 epochs for the “from-scratch” model are lower than base model’s scores in most genres. * The specific “from-scratch” model tested here in Milestone 4, tends to be over-fitted more easily than base model. The pre-trained model, with its frozen layers, is constrained to work with features that are known to be meaningful in related contexts; our “from-scratch” model, on the other hand, is susceptible to overfitting in the convolutional filters. Stronger regularization, different layers and FC size could be tested in future work.

Re-Training some layers of VGG-16

```
fscore.plot.grid(c('VGG w/16 frozen layers',
                   'VGG w/17 frozen layers'),
                 'variant')
```

```
## Warning: Removed 30 rows containing missing values (geom_path).
```



We explored training a portion of the later layers of the VGG-16 model, i.e., layers closer to its last layer, in addition to training our fully connected layers that take the VGG-16 model's output. The motivation for this approach is that the early layers (closer to the input layer) of the VGG-16 model capture general features that would be common to both the ImageNet images the VGG-16 model was trained on **and** to the movie poster images, such as line edges and color blobs, and therefore be more useful for our modeling. The later layers of the VGG-16 model, however, would be increasingly more specific to the 1000 ImageNet image categories the VGG-16 model was trained on, but less relevant to the movie poster images. By re-training the later layers, the hope is that they would become more specific to the features of our movie poster images and thereby improve classification performance.

Since our dataset is large (21,000 training observations) and we consider the movie poster images to be somewhat similar to the ImageNet images VGG-16 was trained on, we decided it was worthwhile to explore the effect of re-training (unfreezing) only the later VGG-16 layers. We compared three models: the base model, with all (19) VGG-16 layers frozen (none re-trained), VGG-16 with 16 frozen layers (3 re-trained), and VGG-16 with 17 frozen layers (2 re-trained).

Summary of plot comparing F-scores:

- All base model training F-score plots are almost exclusively monotonically increasing, while the other two models with some re-training of the VGG-16 layers are less so, and, on average, they achieve lower F-scores compared to the base model, which is unexpected. We expected the two models with unfrozen layers to train better for the reason stated previously.

- For the most part, the test F-score plot are much flatter for all three models and they all achieve similar F-scores by the last epoch run. Again, we expected the unfrozen models would have yielded, on average, higher F-scores.

Time permitting, it would be worthwhile to do the following:

- Try smaller learning rates for the unfrozen layers. Since the weights for the pre-trained layers should already be fairly good, they should be varied less during retraining. Even better might be using a smaller learning rate for the unfrozen layer weights of the VGG-16 and a higher rate for the fully-connected layers we appended to the VGG-16.
- Compare results to those obtained using higher resolution images. We downsampled our images to 48x48 which may be discarding useful features.
- Unfreeze more layers. Based on what we've observed with unfreezing some layers, we wouldn't expect unfreezing more layers to help. However, it would be worth trying for completeness.
- Explore other pre-trained models, such as ResNet50 and InceptionV3. A first attempt at using these models generated an error due to the resolution of our images (48x48); higher resolution is required by both of these models.

Discussion of the results

For most genres, our initial model outperforms a baseline of a stratified dummy classifier (i.e., one which randomly predicts `1` at the base rate). We explored the impact of various hyperparameters on this model.

Some of them provided incremental improvements (for example, switching to L2 regularization and tuning λ, and adjusting the learning rate dynamically.)

In some cases, our initial choices (`fscore_loss`) proved better than the alternative.

And some parameters (e.g., the batch size) we did not yet have the opportunity to tune at all.

With the limited tuning that we were able to do we came close to but did not surpass our best models from Milestone 3. However, it is important to note that Milestone 3 used metadata about the movies, and for this Milestone we *only* used image data. The fact that our models did as well as they did is gratifying.

An important next step will be to build a model that merges the output of the VGG16 CNN with the metadata features, allowing the FC layers to benefit from both.

Additional ideas

Custom loss function

As described above, we defined our own `fscore_loss` function to serve as our objective function. We list that here because we consider it one of the "additional ideas" required for this milestone.

Merging non-image data

In our "results" section, we mentioned that all our Neural-Net models were built using only the pixel data from the poster images, while in previous weeks we extracted features from the data in TMDb and IMDb. One important approach would be to combine the learned image features with the explicitly extracted non-image features.

This could be done with the `keras.layers.merge.Concatenate` which would allow us to combine the features at the output of the VGG-16 model with the non-image features for the corresponding movie before feeding the concatenated tensor in to the first of our fully-connected layers.

We didn't have time to execute on this but we did test the Keras Merge functionality on a simple, prototype model. if time permits, we would like to explore a merged model as part of Milestone 5.

Since the results of the image model and non-image model were of similar strength (based on the per-genre F-scores), we would hope that combining the two would provide even stronger predictive power.

A simpler way of merging the two would be by building an ensemble model that combines the probabilities yielded by the CNN and non-CNN models. Such an ensemble could combine multiple varieties of traditional models, including our Random Forest and SVC classifiers. Once again we see the benefit of structuring a multi-label problem as a collection of individual binary classifiers.

Artificially balanced cohorts

Once we decided to model each label in the multi-label problem by constructing a separate binary clasifier, the difficulties of creating resampled cohorts went away. We could re-approach the imbalance problem by creating balanced resampled cohorts for each genre out of our training set, possibly using our augmented images to bolster the extremely underrepresented genres.