---

title: "Project 3 - Group 2"

output: html_notebook

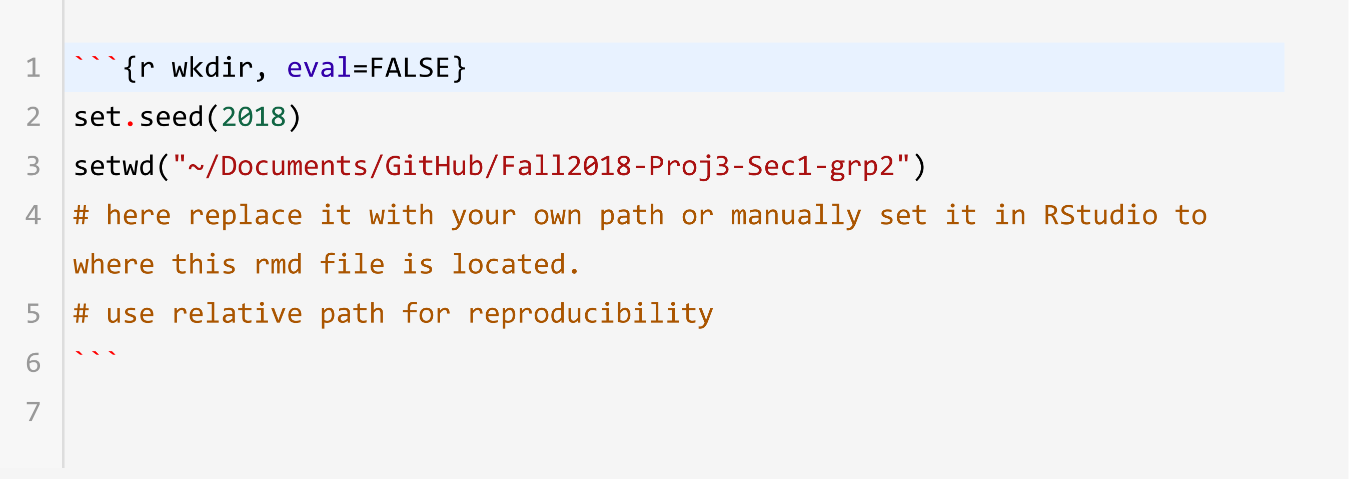
In your final repo, there should be an R markdown file that organizes **all computational steps** for evaluating your proposed image classification framework.

This file is currently a template for running evaluation experiments of image analysis (or any predictive modeling). You should update it according to your codes but following precisely the same structure.

~~~
```{r}
if(!require("EBImage")){
  source("https://bioconductor.org/biocLite.R")
  biocLite("EBImage")
}

if(!require("gbm")){
  install.packages("gbm")
}

library("EBImage")
library("gbm")
```

~~~

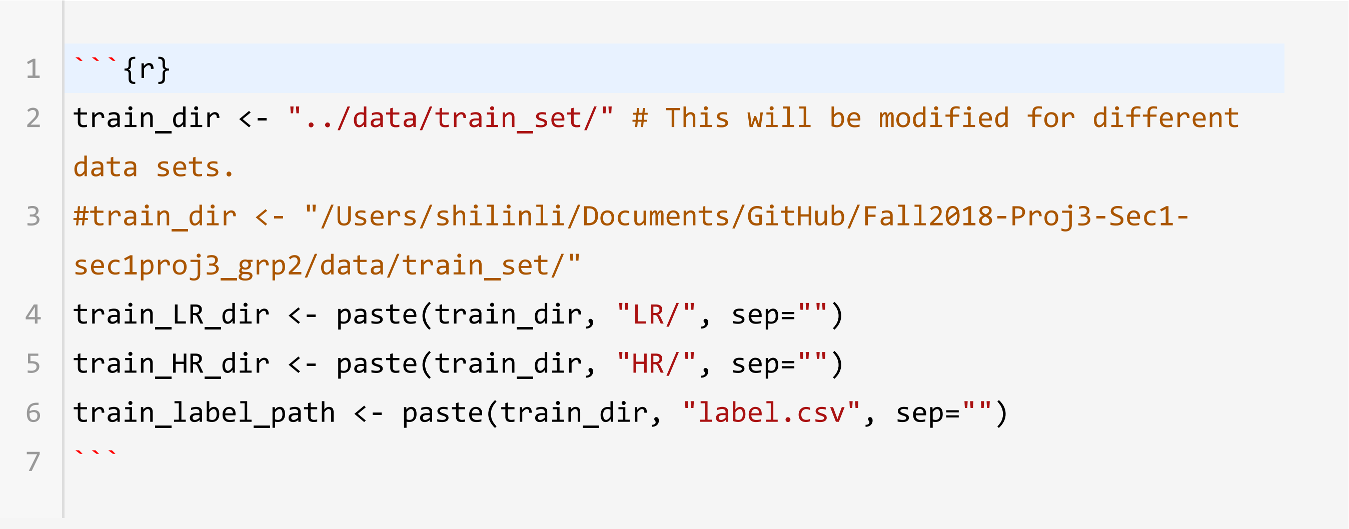
### Step 0: specify directories.

Set the working directory to the image folder. Specify the training and the testing set. For data without an independent test/validation set, you need to create your own testing data by random subsampling. In order to obain reproducible results, set.seed() whenever randomization is used.



~~~
```{r wkdir, eval=FALSE}
set.seed(2018)
setwd("~/Documents/GitHub/Fall2018-Proj3-Sec1-grp2")
# here replace it with your own path or manually set it in RStudio to where this rmd file is located.
# use relative path for reproducibility
```

~~~

Provide directories for training images. Low-resolution (LR) image set and High-resolution (HR) image set will be in different subfolders.



~~~
```{r}
train_dir <- "../data/train_set/" # This will be modified for different data sets.
#train_dir <- "/Users/shilinli/Documents/GitHub/Fall2018-Proj3-Sec1-sec1proj3_grp2/data/train_set/"
train_LR_dir <- paste(train_dir, "LR/", sep="")
train_HR_dir <- paste(train_dir, "HR/", sep="")
train_label_path <- paste(train_dir, "label.csv", sep="")
```
~~~

### Step 1: set up controls for evaluation experiments.

In this chunk, we have a set of controls for the evaluation experiments.

+ (T/F) cross-validation on the training set

+ (number) K, the number of CV folds

+ (T/F) process features for training set

+ (T/F) run evaluation on an independent test set

+ (T/F) process features for test set

~~~
```{r exp_setup}
run.cv=TRUE # run cross-validation on the training set
K <- 5 # number of CV folds
run.feature.train=TRUE # process features for training set
run.test=TRUE # run evaluation on an independent test set
run.feature.test=TRUE # process features for test set
```
~~~

Using cross-validation or independent test set evaluation, we compare the performance of models with different specifications. In this example, we use GBM with different `depth`. In the following chunk, we list, in a vector, setups (in this case, `depth`) corresponding to models that we will compare. In your project, you might compare very different classifiers. You can assign them numerical IDs and labels specific to your project.

~~~
```{r model_setup}
model_values <- seq(3, 11, 2)
model_labels = paste("GBM with depth =", model_values)
```
~~~

### Step 2: import training images class labels.

We provide extra information of image label: car (0), flower (1), market (2). These labels are not necessary for your model.

~~~
```{r train_label}
extra_label <- read.csv(train_label_path, colClasses=c("NULL", NA, NA))
```
~~~

### Step 3: construct features and responses

`feature.R` should be the wrapper for all your feature engineering functions and options. The function `feature( )` should have options that correspond to different scenarios for your project and produces an R object that contains features and responses that are required by all the models you are going to evaluate later.

+ `feature.R`

  + Input: a path for low-resolution images.

  + Input: a path for high-resolution images.

  + Output: an RData file that contains extracted features and corresponding responses

~~~
```{r feature}
source("../lib/feature.R")

tm_feature_train <- NA
if(run.feature.train){
  tm_feature_train <- system.time(dat_train <- feature(train_LR_dir, train_HR_dir))
  feat_train <- dat_train$feature
  label_train <- dat_train$label
}

#save(dat_train, file="./output/feature_train.RData")
```

~~~

### Step 4: Train a classification model with training images

Call the train model and test model from library.

`train.R` and `test.R` should be wrappers for all your model training steps and your classification/prediction steps.

+ `train.R`

  + Input: a path that points to the training set features and responses.

  + Output: an RData file that contains trained classifiers in the forms of R objects: models/settings/links to external trained configurations.

+ `test.R`

+ Input: a path that points to the test set features.

+ Input: an R object that contains a trained classifier.

+ Output: an R object of response predictions on the test set. If there are multiple classifiers under evaluation, there should be multiple sets of label predictions.

~~~
```{r loadlib}
source("../lib/train.R")
source("../lib/test.R")
```
~~~

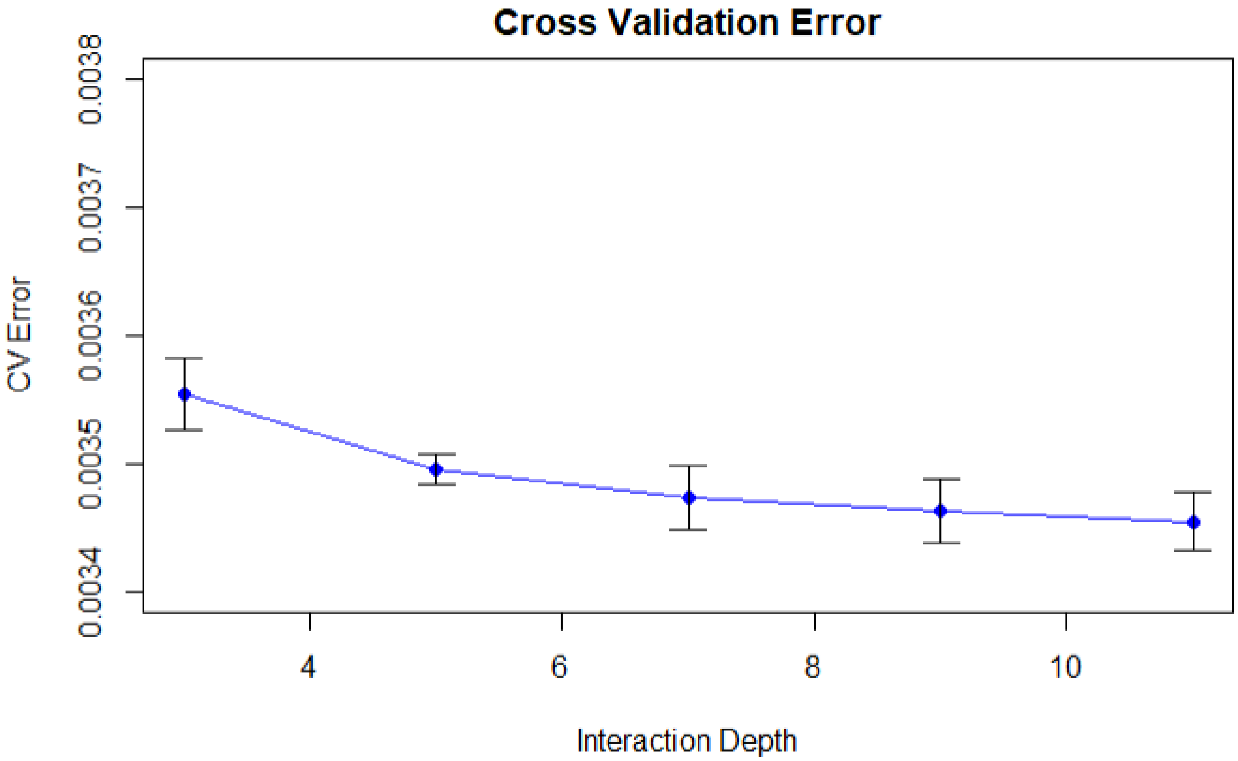
#### Model selection with cross-validation

* Do model selection by choosing among different values of training model parameters, that is, the interaction depth for GBM in this example.

~~~
```{r runcv, message=FALSE, warning=FALSE}
source("../lib/cross_validation.R")

if(run.cv){
  err_cv <- array(dim=c(length(model_values), 2))
  for(k in 1:length(model_values)){
    cat("k=", k, "\n")
    err_cv[k,] <- cv.function(feat_train, label_train, model_values[k], K)
  }
  save(err_cv, file="../output/err_cv.RData")
}
```
~~~

~~~
```{r runcv, message=FALSE, warning=FALSE}
~~~

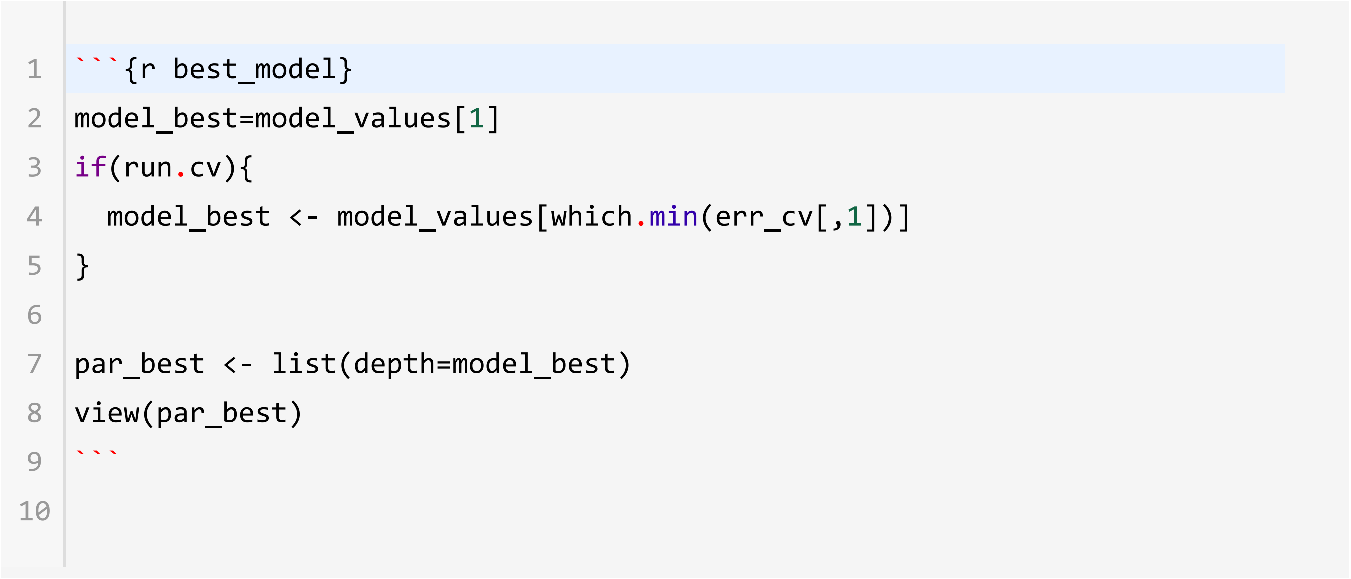
```
source("../lib/cross_validation.R")

if(run.cv){
  err_cv <- array(dim=c(length(model_values), 2))
  for(k in 1:length(model_values)){
    cat("k=", k, "\n")
    err_cv[k,] <- cv.function(feat_train, label_train, model_values[k], K)
  }
  save(err_cv, file="../output/err_cv.RData")
}
```
```

Visualize cross-validation results.

````
```{r cv_vis}
if(run.cv){
  load("../output/err_cv.RData")
  plot(model_values, err_cv[,1], xlab="Interaction Depth", ylab="CV Error",
       main="Cross Validation Error", type="n", ylim=c(0, 0.25))
  points(model_values, err_cv[,1], col="blue", pch=16)
  lines(model_values, err_cv[,1], col="blue")
  arrows(model_values, err_cv[,1]-err_cv[,2], model_values, err_cv[,1]+err_cv[,2],
         length=0.1, angle=90, code=3)
}
```
````
````



There two choices for the optimal depth. one is  5 and the other is 11. Since the point at 5 has the biggest gradient changing while the point at 11

reaches the minumum MSE. At this project, we prefer depth = 11.

* Choose the "best"" parameter value



~~~
```{r best_model}
model_best=model_values[1]
if(run.cv){
  model_best <- model_values[which.min(err_cv[,1])]
}

par_best <- list(depth=model_best)
view(par_best)
```
~~~

par_best = 11

* Train the model with the entire training set using the selected model (model parameter) via cross-validation.

~~~
```{r final_train}
load("../lib/feature_train.RData")

# Baseline GBM
#tm_train=NA
#tm_train <- system.time(fit_train_gbm <- train(dat_train$feature, dat_train$label,run_gbm = T,run_xgb = F,run_lr=F, run_rf = F))

# Improved Model XGB
tm_train=NA
#tm_train <- system.time(fit_train <- train(feat_train, label_train, par_best))
tm_train <- system.time(fit_train_xgb <- train(dat_train$feature, dat_train$label,run_gbm = F,run_xgb = T,run_lr=F, run_rf = F))

#save(fit_train_gbm, file="../output/fit_train_gbm.RData")
#save(fit_train_xgb, file="../output/fit_train_xgb.RData")
```
~~~

### Step 5: Super-resolution for test images

Feed the final training model with the completely holdout testing data.

+ `superResolution.R`

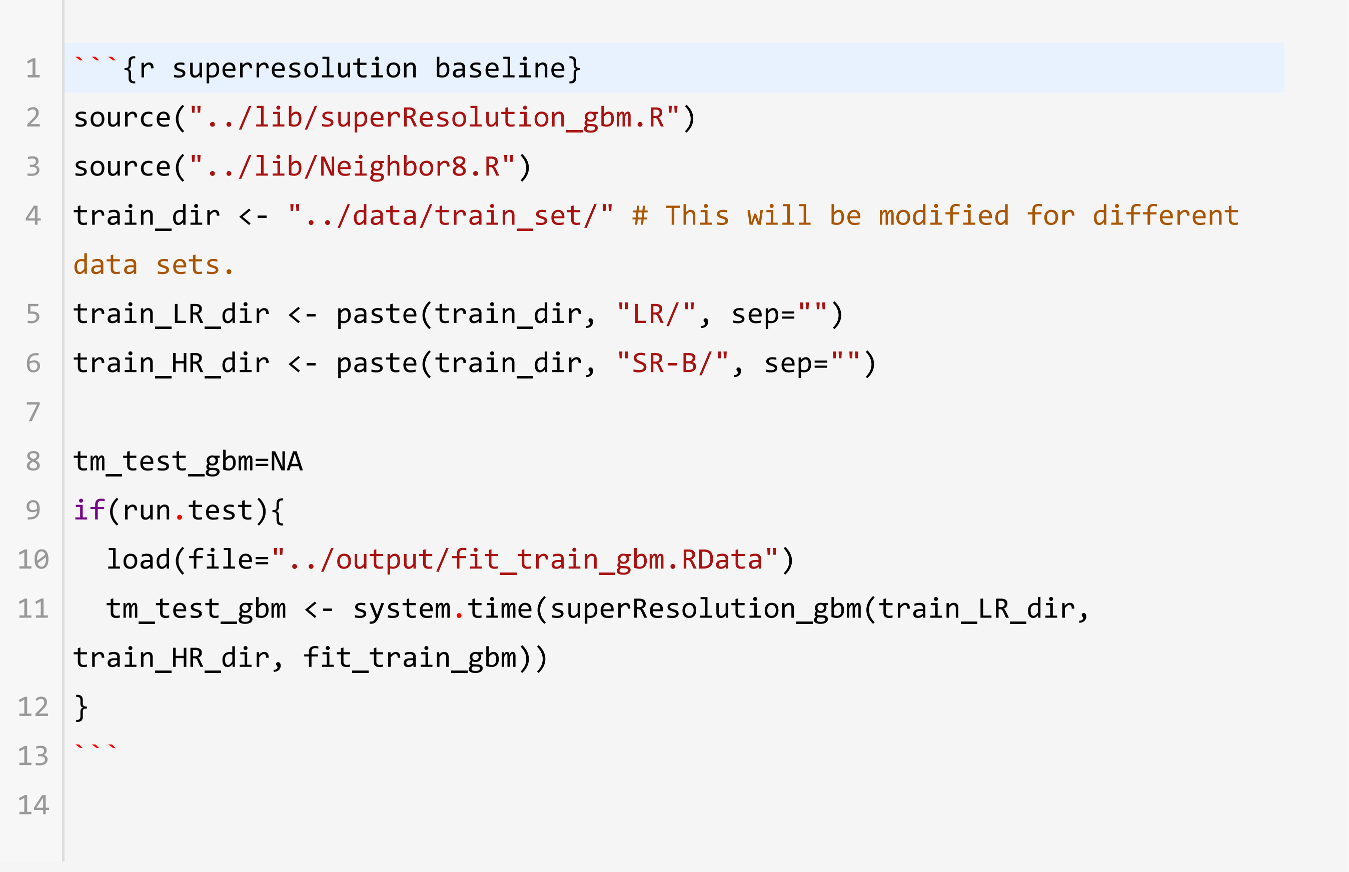
  + Input: a path that points to the folder of low-resolution test images.

  + Input: a path that points to the folder (empty) of high-resolution test images.

  + Input: an R object that contains tuned predictors.

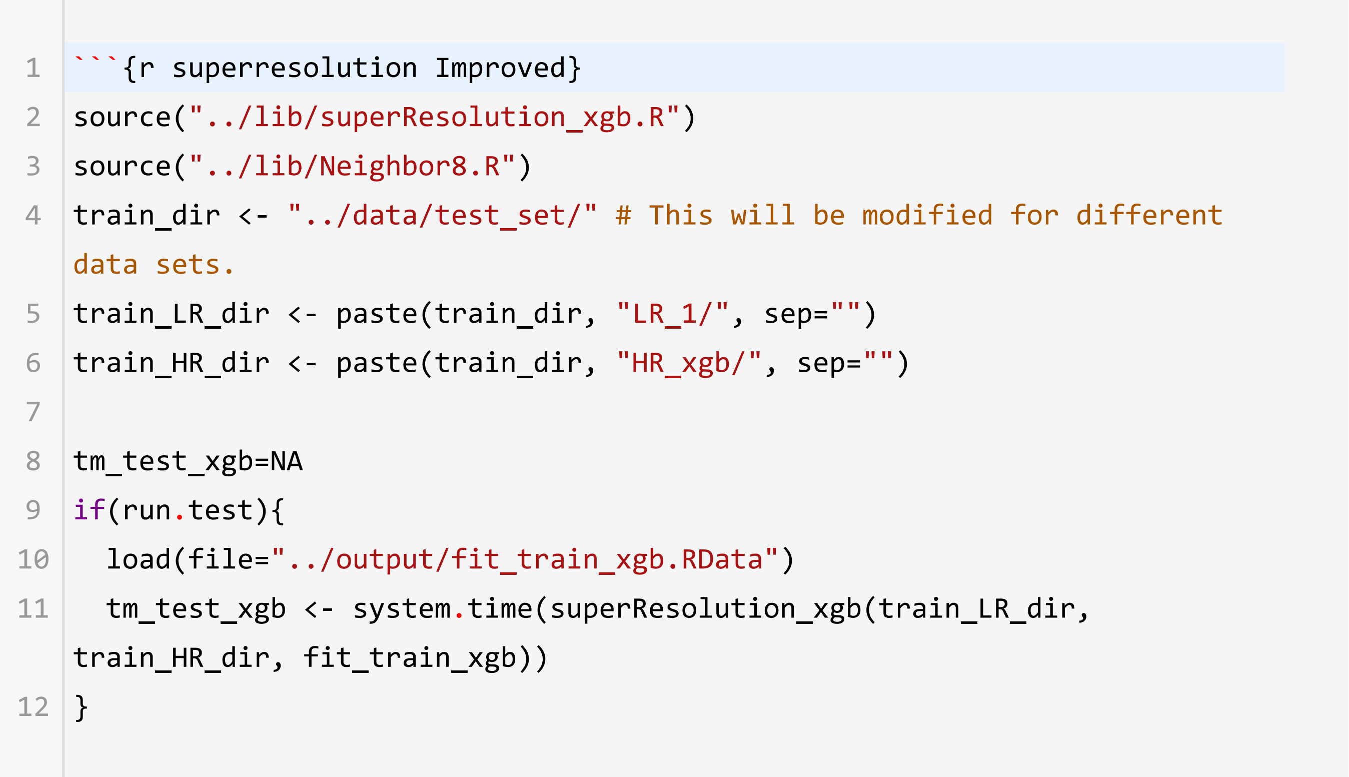
  + Output: construct high-resolution versions for each low-resolution test image.

GBM Baseline



~~~
```{r superresolution baseline}
source("../lib/superResolution_gbm.R")
source("../lib/Neighbor8.R")
train_dir <- "../data/train_set/" # This will be modified for different data sets.
train_LR_dir <- paste(train_dir, "LR/", sep="")
train_HR_dir <- paste(train_dir, "SR-B/", sep="")

tm_test_gbm=NA
if(run.test){
  load(file="../output/fit_train_gbm.RData")
  tm_test_gbm <- system.time(superResolution_gbm(train_LR_dir, train_HR_dir, fit_train_gbm))
}
```

~~~

XGB Improved



~~~
```{r superresolution Improved}
source("../lib/superResolution_xgb.R")
source("../lib/Neighbor8.R")
train_dir <- "../data/test_set/" # This will be modified for different data sets.
train_LR_dir <- paste(train_dir, "LR_1/", sep="")
train_HR_dir <- paste(train_dir, "HR_xgb/", sep="")

tm_test_xgb=NA
if(run.test){
  load(file="../output/fit_train_xgb.RData")
  tm_test_xgb <- system.time(superResolution_xgb(train_LR_dir, train_HR_dir, fit_train_xgb))
}
~~~

~~~
```
~~~

### Summarize Running Time

Prediction performance matters, so does the running times for constructing features and for training the model, especially when the computation resource is limited.

~~~
```{r running_time}
 tm_train[1]
 tm_test[1]
 tm_test_xgb[1]
 tm_test_xgb[1]
```
~~~

| | time | model type |
|---|---|---|
| tm_train | > 6h | training time of baseline |
| tm_test | 53 min | testing time of baseline |
| tm_train_xgb | 244.42s | training time of xgb |
| tm_test_xgb | 12min | testing time of xgb |

We also tryed  deep learning model called DCSCN, a tensorflow implementation of "Fast and Accurate Image Super Resolution by Deep CNN with Skip Connection and Network in Network", a deep learning based Single-Image Super-Resolution (SISR) model.But  this model actually cost a lot of time to generate the pics, so we gave up at the final run.

Reference link: https://github.com/jiny2001/dcscn-super-resolution

```
"""
Paper: "Fast and Accurate Image Super Resolution by Deep CNN with Skip
```

```
Connection and Network in Network"
Ver: 2.0

DCSCN model implementation (Transposed-CNN / Pixel Shuffler version)
See Detail: https://github.com/jiny2001/dcscn-super-resolution/

Please note this model is updated version of the paper.
If you want to check original source code and results of the paper,
please see https://github.com/jiny2001/dcscn-super-resolution/tree/ver1.
"""

import logging
import math
import os
import time

import numpy as np
import tensorflow as tf

from helper import loader, tf_graph, utilty as util

BICUBIC_METHOD_STRING = "bicubic"


class SuperResolution(tf_graph.TensorflowGraph):
    def __init__(self, flags, model_name=""):

        super().__init__(flags)

        # Model Parameters
        self.scale = flags.scale
        self.layers = flags.layers
        self.filters = flags.filters
        self.min_filters = min(flags.filters, flags.min_filters)
        self.filters_decay_gamma = flags.filters_decay_gamma
        self.use_nin = flags.use_nin
        self.nin_filters = flags.nin_filters
        self.nin_filters2 = flags.nin_filters2
        self.reconstruct_layers = max(flags.reconstruct_layers, 1)
```

```python
        self.reconstruct_filters = flags.reconstruct_filters
        self.resampling_method = BICUBIC_METHOD_STRING
        self.pixel_shuffler = flags.pixel_shuffler
        self.pixel_shuffler_filters = flags.pixel_shuffler_filters
        self.self_ensemble = flags.self_ensemble

        # Training Parameters
        self.l2_decay = flags.l2_decay
        self.optimizer = flags.optimizer
        self.beta1 = flags.beta1
        self.beta2 = flags.beta2
        self.epsilon = flags.epsilon
        self.momentum = flags.momentum
        self.batch_num = flags.batch_num
        self.batch_image_size = flags.batch_image_size
        if flags.stride_size == 0:
            self.stride_size = flags.batch_image_size // 2
        else:
            self.stride_size = flags.stride_size
        self.clipping_norm = flags.clipping_norm
        self.use_l1_loss = flags.use_l1_loss

        # Learning Rate Control for Training
        self.initial_lr = flags.initial_lr
        self.lr_decay = flags.lr_decay
        self.lr_decay_epoch = flags.lr_decay_epoch

        # Dataset or Others
        self.training_images = int(math.ceil(flags.training_images / flags.batch_num) * flags.batch_num)
        self.train = None
        self.test = None

        # Image Processing Parameters
        self.max_value = flags.max_value
        self.channels = flags.channels
        self.output_channels = 1
        self.psnr_calc_border_size = flags.psnr_calc_border_size
        if self.psnr_calc_border_size < 0:
            self.psnr_calc_border_size = self.scale

        # Environment (all directory name should not contain tailing '/' )
        self.batch_dir = flags.batch_dir

        # initialize variables
        self.name = self.get_model_name(model_name)
        self.total_epochs = 0
        lr = self.initial_lr
        while lr > flags.end_lr:
            self.total_epochs += self.lr_decay_epoch
            lr *= self.lr_decay

        # initialize environment
        util.make_dir(self.checkpoint_dir)
        util.make_dir(flags.graph_dir)
        util.make_dir(self.tf_log_dir)
        if flags.initialize_tf_log:
            util.clean_dir(self.tf_log_dir)
        util.set_logging(flags.log_filename, stream_log_level=logging.INFO, file_log_level=logging.INFO,
                         tf_log_level=tf.logging.WARN)
        logging.info("\nDCSCN v2-------------------------------------")
        logging.info("%s [%s]" % (util.get_now_date(), self.name))

        self.init_train_step()

    def get_model_name(self, model_name, name_postfix=""):
        if model_name is "":
            name = "dcscn_L%d_F%d" % (self.layers, self.filters)
            if self.min_filters != 0:
                name += "to%d" % self.min_filters
            if self.filters_decay_gamma != 1.5:
                name += "_G%2.2f" % self.filters_decay_gamma
            if self.cnn_size != 3:
                name += "_C%d" % self.cnn_size
            if self.scale != 2:
                name += "_Sc%d" % self.scale
```

```python
            if self.use_nin:
                name += "_NIN"
                if self.nin_filters != 0:
                    name += "_A%d" % self.nin_filters
                if self.nin_filters2 != self.nin_filters // 2:
                    name += "_B%d" % self.nin_filters2
            if self.pixel_shuffler:
                name += "_PS"
            if self.max_value != 255.0:
                name += "_M%2.1f" % self.max_value
            if self.activator != "prelu":
                name += "_%s" % self.activator
            if self.batch_norm:
                name += "_BN"
            if self.reconstruct_layers >= 1:
                name += "_R%d" % self.reconstruct_layers
                if self.reconstruct_filters != 1:
                    name += "F%d" % self.reconstruct_filters
            if name_postfix is not "":
                name += "_" + name_postfix
        else:
            name = "dcscn_%s" % model_name

        return name

    def load_dynamic_datasets(self, data_dir, batch_image_size):
        """ loads datasets
        Opens image directory as a datasets. Images will be loaded when build_input_batch() is called.
        """

        self.train = loader.DynamicDataSets(self.scale, batch_image_size, channels=self.channels,
                                            resampling_method=self.resampling_method)
        self.train.set_data_dir(data_dir)

    def load_datasets(self, data_dir, batch_dir, batch_image_size, stride_size=0):
        """ build input patch images and loads as a datasets
        Opens image directory as a datasets.
        Each images are splitted into patch images and converted to input image. Since loading
        (especially from PNG/JPG) and building input-LR images needs much computation in the
        training phase, building pre-processed images makes training much faster. However, images
        are limited by divided grids.
        """

        batch_dir += "/scale%d" % self.scale

        self.train = loader.BatchDataSets(self.scale, batch_dir, batch_image_size, stride_size, channels=self.channels,
                                          resampling_method=self.resampling_method)

        if not self.train.is_batch_exist():
            self.train.build_batch(data_dir)
        else:
            self.train.load_batch_counts()
        self.train.load_all_batch_images()

    def init_epoch_index(self):

        self.batch_input = self.batch_num * [None]
        self.batch_input_bicubic = self.batch_num * [None]
        self.batch_true = self.batch_num * [None]

        self.training_psnr_sum = 0
        self.training_loss_sum = 0
        self.training_step = 0
        self.train.init_batch_index()

    def build_input_batch(self):

        for i in range(self.batch_num):
            self.batch_input[i], self.batch_input_bicubic[i],
self.batch_true[i] = self.train.load_batch_image(
                self.max_value)

    def build_graph(self):

        self.x = tf.placeholder(tf.float32, shape=[None, None, None, self.channels], name="x")
        self.y = tf.placeholder(tf.float32, shape=[None, None, None, self.output_channels], name="y")
        self.x2 = tf.placeholder(tf.float32, shape=[None, None, None, self.output_channels], name="x2")
        self.dropout = tf.placeholder(tf.float32, shape=[], name="dropout_keep_rate")
        self.is_training = tf.placeholder(tf.bool, name="is_training")

        # building feature extraction layers

        output_feature_num = self.filters
        total_output_feature_num = 0
        input_feature_num = self.channels
        input_tensor = self.x

        if self.save_weights:
            with tf.name_scope("X"):
                util.add_summaries("output", self.name, self.x, save_stddev=True, save_mean=True)

        for i in range(self.layers):
            if self.min_filters != 0 and i > 0:
                x1 = i / float(self.layers - 1)
                y1 = pow(x1, 1.0 / self.filters_decay_gamma)
                output_feature_num = int((self.filters - self.min_filters) * (1 - y1) + self.min_filters)

            self.build_conv("CNN%d" % (i + 1), input_tensor, self.cnn_size, input_feature_num,
                            output_feature_num, use_bias=True, activator=self.activator,
                            use_batch_norm=self.batch_norm,
dropout_rate=self.dropout_rate)
            input_feature_num = output_feature_num
            input_tensor = self.H[-1]
            total_output_feature_num += output_feature_num

        with tf.variable_scope("Concat"):
            self.H_concat = tf.concat(self.H, 3, name="H_concat")
        self.features += " Total: (%d)" % total_output_feature_num

        # building reconstruction layers ---

        if self.use_nin:
            self.build_conv("A1", self.H_concat, 1, total_output_feature_num, self.nin_filters,
                            dropout_rate=self.dropout_rate, use_bias=True, activator=self.activator)
            self.receptive_fields -= (self.cnn_size - 1)

            self.build_conv("B1", self.H_concat, 1, total_output_feature_num, self.nin_filters2,
                            dropout_rate=self.dropout_rate, use_bias=True, activator=self.activator)

            self.build_conv("B2", self.H[-1], 3, self.nin_filters2, self.nin_filters2,
                            dropout_rate=self.dropout_rate, use_bias=True, activator=self.activator)

            self.H.append(tf.concat([self.H[-1], self.H[-3]], 3, name="Concat2"))
            input_channels = self.nin_filters + self.nin_filters2
        else:
            self.H.append(self.H_concat)
            input_channels = total_output_feature_num

        # building upsampling layer
        if self.pixel_shuffler:
            if self.pixel_shuffler_filters != 0:
                output_channels = self.pixel_shuffler_filters
```

```python
            else:
                output_channels = input_channels
            if self.scale == 4:
                self.build_pixel_shuffler_layer("Up-PS", self.H[-1], 2, input_channels, input_channels)
                self.build_pixel_shuffler_layer("Up-PS2", self.H[-1], 2, input_channels, output_channels)
            else:
                self.build_pixel_shuffler_layer("Up-PS", self.H[-1], self.scale, input_channels, output_channels)
            input_channels = output_channels
        else:
            self.build_transposed_conv("Up-TCNN", self.H[-1], self.scale, input_channels)

        for i in range(self.reconstruct_layers - 1):
            self.build_conv("R-CNN%d" % (i + 1), self.H[-1], self.cnn_size, input_channels, self.reconstruct_filters,
                            dropout_rate=self.dropout_rate, use_bias=True, activator=self.activator)
            input_channels = self.reconstruct_filters

        self.build_conv("R-CNN%d" % self.reconstruct_layers, self.H[-1], self.cnn_size, input_channels,
                        self.output_channels)

        self.y_ = tf.add(self.H[-1], self.x2, name="output")

        if self.save_weights:
            with tf.name_scope("Y_"):
                util.add_summaries("output", self.name, self.y_, save_stddev=True, save_mean=True)

        logging.info("Feature:%s Complexity:%s Receptive Fields:%d" % (
            self.features, "{:,}".format(self.complexity), self.receptive_fields))

    def build_optimizer(self):
        """
        Build loss function. We use 6+scale as a border and we don't calculate MSE on the border.
        """

        self.lr_input = tf.placeholder(tf.float32, shape=[], name="LearningRate")

        diff = tf.subtract(self.y_, self.y, "diff")

        if self.use_l1_loss:
            self.mse = tf.reduce_mean(tf.square(diff, name="diff_square"), name="mse")
            self.image_loss = tf.reduce_mean(tf.abs(diff, name="diff_abs"), name="image_loss")
        else:
            self.mse = tf.reduce_mean(tf.square(diff, name="diff_square"), name="mse")
            self.image_loss = tf.identity(self.mse, name="image_loss")

        if self.l2_decay > 0:
            l2_norm_losses = [tf.nn.l2_loss(w) for w in self.Weights]
            l2_norm_loss = self.l2_decay * tf.add_n(l2_norm_losses)
            if self.enable_log:
                tf.summary.scalar("L2WeightDecayLoss/" + self.name, l2_norm_loss)

            self.loss = self.image_loss + l2_norm_loss
        else:
            self.loss = self.image_loss

        if self.enable_log:
            tf.summary.scalar("Loss/" + self.name, self.loss)

        if self.batch_norm:
            update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)
            with tf.control_dependencies(update_ops):
                self.training_optimizer = self.add_optimizer_op(self.loss, self.lr_input)
        else:
            self.training_optimizer = self.add_optimizer_op(self.loss, self.lr_input)

        util.print_num_of_total_parameters(output_detail=True)

    def get_psnr_tensor(self, mse):

        with tf.variable_scope('get_PSNR'):
            value = tf.constant(self.max_value, dtype=mse.dtype) / tf.sqrt(mse)
            numerator = tf.log(value)
            denominator = tf.log(tf.constant(10, dtype=mse.dtype))
            return tf.constant(20, dtype=mse.dtype) * numerator / denominator

    def add_optimizer_op(self, loss, lr_input):

        if self.optimizer == "gd":
            optimizer = tf.train.GradientDescentOptimizer(lr_input)
        elif self.optimizer == "adadelta":
            optimizer = tf.train.AdadeltaOptimizer(lr_input)
        elif self.optimizer == "adagrad":
            optimizer = tf.train.AdagradOptimizer(lr_input)
        elif self.optimizer == "adam":
            optimizer = tf.train.AdamOptimizer(lr_input, beta1=self.beta1, beta2=self.beta2, epsilon=self.epsilon)
        elif self.optimizer == "momentum":
            optimizer = tf.train.MomentumOptimizer(lr_input, self.momentum)
        elif self.optimizer == "rmsprop":
            optimizer = tf.train.RMSPropOptimizer(lr_input, momentum=self.momentum)
        else:
            print("Optimizer arg should be one of [gd, adadelta, adagrad, adam, momentum, rmsprop].")
            return None

        if self.clipping_norm > 0 or self.save_weights:
            trainables = tf.trainable_variables()
            grads = tf.gradients(loss, trainables)

            if self.save_weights:
                for i in range(len(grads)):
                    util.add_summaries("", self.name, grads[i], header_name=grads[i].name + "/", save_stddev=True,
                                       save_mean=True)

        if self.clipping_norm > 0:
            clipped_grads, _ = tf.clip_by_global_norm(grads, clip_norm=self.clipping_norm)
            grad_var_pairs = zip(clipped_grads, trainables)
            training_optimizer = optimizer.apply_gradients(grad_var_pairs)
        else:
            training_optimizer = optimizer.minimize(loss)

        return training_optimizer

    def train_batch(self):

        feed_dict = {self.x: self.batch_input, self.x2: self.batch_input_bicubic, self.y: self.batch_true,
                     self.lr_input: self.lr, self.dropout: self.dropout_rate, self.is_training: 1}

        _, image_loss, mse = self.sess.run([self.training_optimizer, self.image_loss, self.mse], feed_dict=feed_dict)
        self.training_loss_sum += image_loss
        self.training_psnr_sum += util.get_psnr(mse, max_value=self.max_value)

        self.training_step += 1
        self.step += 1

    def log_to_tensorboard(self, test_filename, psnr, save_meta_data=True):

        if self.enable_log is False:
            return

        # todo
        save_meta_data = False

        org_image = util.set_image_alignment(util.load_image(test_filename, print_console=False), self.scale)

        if len(org_image.shape) >= 3 and org_image.shape[2] == 3 and self.channels == 1:
            org_image = util.convert_rgb_to_y(org_image)

        input_image = util.resize_image_by_pil(org_image, 1.0 / self.scale, resampling_method=self.resampling_method)
        bicubic_image = util.resize_image_by_pil(input_image, self.scale, resampling_method=self.resampling_method)

        if self.max_value != 255.0:
            input_image = np.multiply(input_image, self.max_value / 255.0)  # type: np.ndarray
            bicubic_image = np.multiply(bicubic_image, self.max_value / 255.0)  # type: np.ndarray
            org_image = np.multiply(org_image, self.max_value / 255.0)  # type: np.ndarray

        feed_dict = {self.x: input_image.reshape([1, input_image.shape[0], input_image.shape[1], input_image.shape[2]]),
                     self.x2: bicubic_image.reshape(
                         [1, bicubic_image.shape[0], bicubic_image.shape[1], bicubic_image.shape[2]]),
                     self.y: org_image.reshape([1, org_image.shape[0], org_image.shape[1], org_image.shape[2]]),
                     self.dropout: 1.0,
                     self.is_training: 0}

        if save_meta_data:
            # profiler = tf.profiler.Profile(self.sess.graph)

            run_metadata = tf.RunMetadata()
            run_options = tf.RunOptions(trace_level=tf.RunOptions.FULL_TRACE)
            summary_str, _ = self.sess.run([self.summary_op, self.loss], feed_dict=feed_dict, options=run_options,
                                           run_metadata=run_metadata)
            self.test_writer.add_run_metadata(run_metadata, "step%d" % self.epochs_completed)

            filename = self.checkpoint_dir + "/" + self.name + "_metadata.txt"
            with open(filename, "w") as out:
                out.write(str(run_metadata))

            # filename = self.checkpoint_dir + "/" + self.name + "_memory.txt"
            # tf.profiler.write_op_log(
            #   tf.get_default_graph(),
            #   log_dir=self.checkpoint_dir,
            #   #op_log=op_log,
            #   run_meta=run_metadata)

            tf.contrib.tfprof.model_analyzer.print_model_analysis(
                tf.get_default_graph(), run_meta=run_metadata,
                tfprof_options=tf.contrib.tfprof.model_analyzer.PRINT_ALL_TIMING_MEMORY)

        else:
            summary_str, _ = self.sess.run([self.summary_op, self.loss], feed_dict=feed_dict)

        self.train_writer.add_summary(summary_str, self.epochs_completed)
        if not self.use_l1_loss:
            if self.training_step != 0:
                util.log_scalar_value(self.train_writer, 'PSNR', self.training_psnr_sum / self.training_step,
                                      self.epochs_completed)
        util.log_scalar_value(self.train_writer, 'LR', self.lr, self.epochs_completed)
        self.train_writer.flush()

        util.log_scalar_value(self.test_writer, 'PSNR', psnr, self.epochs_completed)
        self.test_writer.flush()

    def update_epoch_and_lr(self):

        self.epochs_completed_in_stage += 1

        if self.epochs_completed_in_stage >= self.lr_decay_epoch:

            # set new learning rate
            self.lr *= self.lr_decay
            self.epochs_completed_in_stage = 0
            return True
        else:
            return False

    def print_status(self, psnr, ssim, log=False):

        if self.step == 0:
            logging.info("Initial PSNR:%f SSIM:%f" % (psnr, ssim))
        else:
            processing_time = (time.time() - self.start_time) / self.step
            if self.use_l1_loss:
                line_a = "%s Step:%s PSNR:%f SSIM:%f (Training Loss:%0.3f)" % (
                    util.get_now_date(), "{:,}".format(self.step), psnr, ssim,
                    self.training_loss_sum / self.training_step)
            else:
                line_a = "%s Step:%s PSNR:%f SSIM:%f (Training PSNR:%0.3f)" % (
                    util.get_now_date(), "{:,}".format(self.step), psnr, ssim,
                    self.training_psnr_sum / self.training_step)
            estimated = processing_time * (self.total_epochs - self.epochs_completed) * (
                self.training_images // self.batch_num)
            h = estimated // (60 * 60)
            estimated -= h * 60 * 60
            m = estimated // 60
            s = estimated - m * 60
            line_b = "Epoch:%d LR:%f (%2.3fsec/step) Estimated:%d:%d:%d" % (
                self.epochs_completed, self.lr, processing_time, h, m, s)
            if log:
                logging.info(line_a)
                logging.info(line_b)
            else:
                print(line_a)
                print(line_b)

    def print_weight_variables(self):

        for bias in self.Biases:
            util.print_filter_biases(bias)

        for weight in self.Weights:
            util.print_filter_weights(weight)

    def evaluate(self, test_filenames):

        total_psnr = total_ssim = 0
        if len(test_filenames) == 0:
            return 0, 0

        for filename in test_filenames:
            psnr, ssim = self.do_for_evaluate(filename, print_console=False)
            total_psnr += psnr
            total_ssim += ssim

        return total_psnr / len(test_filenames), total_ssim / len(test_filenames)

    def do(self, input_image, bicubic_input_image=None):

        h, w = input_image.shape[:2]
        ch = input_image.shape[2] if len(input_image.shape) > 2 else 1

        if bicubic_input_image is None:
            bicubic_input_image = util.resize_image_by_pil(input_image, self.scale,
                                                           resampling_method=self.resampling_method)
        if self.max_value != 255.0:
            input_image = np.multiply(input_image, self.max_value / 255.0)  # type: np.ndarray
            bicubic_input_image = np.multiply(bicubic_input_image, self.max_value / 255.0)  # type: np.ndarray

        if self.self_ensemble > 1:
            output = np.zeros([self.scale * h, self.scale * w, 1])

            for i in range(self.self_ensemble):
                image = util.flip(input_image, i)
                bicubic_image = util.flip(bicubic_input_image, i)
                y = self.sess.run(self.y_, feed_dict={self.x: image.reshape(1, image.shape[0], image.shape[1], ch),
                                                      self.x2: bicubic_image.reshape(1, self.scale * image.shape[0],
                                                                                     self.scale * image.shape[1],
                                                                                     ch),
                                                      self.dropout: 1.0, self.is_training: 0})
                restored = util.flip(y[0], i, invert=True)
                output += restored

            output /= self.self_ensemble
        else:
            y = self.sess.run(self.y_, feed_dict={self.x: input_image.reshape(1, h, w, ch),
                                                  self.x2: bicubic_input_image.reshape(1, self.scale * h,
                                                                                       self.scale * w, ch),
                                                  self.dropout: 1.0, self.is_training: 0})
            output = y[0]

        if self.max_value != 255.0:
            hr_image = np.multiply(output, 255.0 / self.max_value)
        else:
            hr_image = output

        return hr_image

    def do_for_file(self, file_path, output_folder="output"):

        org_image = util.load_image(file_path)

        filename, extension = os.path.splitext(os.path.basename(file_path))
        output_folder += "/" + self.name + "/"
        util.save_image(output_folder + filename + extension, org_image)

        if len(org_image.shape) >= 3 and org_image.shape[2] == 3 and self.channels == 1:
            input_y_image = util.convert_rgb_to_y(org_image)
            scaled_image = util.resize_image_by_pil(input_y_image, self.scale, resampling_method=self.resampling_method)
            util.save_image(output_folder + filename + "_bicubic_y" + extension, scaled_image)
            output_y_image = self.do(input_y_image)
            util.save_image(output_folder + filename + "_result_y" + extension, output_y_image)

            scaled_ycbcr_image = util.convert_rgb_to_ycbcr(
                util.resize_image_by_pil(org_image, self.scale, self.resampling_method))
            image = util.convert_y_and_cbcr_to_rgb(output_y_image, scaled_ycbcr_image[:, :, 1:3])
        else:
            scaled_image = util.resize_image_by_pil(org_image, self.scale, resampling_method=self.resampling_method)
            util.save_image(output_folder + filename + "_bicubic_y" + extension, scaled_image)
            image = self.do(org_image)

        util.save_image(output_folder + filename + "_result" + extension, image)

    def do_for_evaluate_with_output(self, file_path, output_directory, print_console=False):

        filename, extension = os.path.splitext(file_path)
        output_directory += "/" + self.name + "/"
        util.make_dir(output_directory)

        true_image = util.set_image_alignment(util.load_image(file_path, print_console=False), self.scale)

        if true_image.shape[2] == 3 and self.channels == 1:

            # for color images
            input_y_image = loader.build_input_image(true_image, channels=self.channels, scale=self.scale,
                                                     alignment=self.scale, convert_ycbcr=True)
            input_bicubic_y_image = util.resize_image_by_pil(input_y_image, self.scale,
                                                             resampling_method=self.resampling_method)

            true_ycbcr_image = util.convert_rgb_to_ycbcr(true_image)

            output_y_image = self.do(input_y_image,
```

```python
                                                input_bicubic_y_image)
            psnr, ssim = util.compute_psnr_and_ssim(true_ycbcr_image[:, :, 0:1], output_y_image,
                                                     border_size=self.psnr_calc_border_size)
            loss_image = util.get_loss_image(true_ycbcr_image[:, :, 0:1], output_y_image,
                                             border_size=self.psnr_calc_border_size)

            output_color_image = util.convert_y_and_cbcr_to_rgb(output_y_image, true_ycbcr_image[:, :, 1:3])

            util.save_image(output_directory + file_path, true_image)
            util.save_image(output_directory + filename + "_input" + extension, input_y_image)
            util.save_image(output_directory + filename + "_input_bicubic" + extension, input_bicubic_y_image)
            util.save_image(output_directory + filename + "_true_y" + extension, true_ycbcr_image[:, :, 0:1])
            util.save_image(output_directory + filename + "_result" + extension, output_y_image)
            util.save_image(output_directory + filename + "_result_c" + extension, output_color_image)
            util.save_image(output_directory + filename + "_loss" + extension, loss_image)

        elif true_image.shape[2] == 1 and self.channels == 1:

            # for monochrome images
            input_image = loader.build_input_image(true_image, channels=self.channels, scale=self.scale,
                                                   alignment=self.scale)
            input_bicubic_y_image = util.resize_image_by_pil(input_image, self.scale,
                                                             resampling_method=self.resampling_method)
            output_image = self.do(input_image, input_bicubic_y_image)
            psnr, ssim = util.compute_psnr_and_ssim(true_image, output_image, border_size=self.psnr_calc_border_size)
            util.save_image(output_directory + file_path, true_image)
            util.save_image(output_directory + filename + "_result" + extension, output_image)
        else:
            return None, None

        if print_console:
            print("[%s] PSNR:%f, SSIM:%f" % (filename, psnr, ssim))

        return psnr, ssim

    def do_for_evaluate(self, file_path, print_console=False):

        true_image = util.set_image_alignment(util.load_image(file_path, print_console=False), self.scale)

        if true_image.shape[2] == 3 and self.channels == 1:

            # for color images
            input_y_image = loader.build_input_image(true_image, channels=self.channels, scale=self.scale,
alignment=self.scale, convert_ycbcr=True)
            true_y_image = util.convert_rgb_to_y(true_image)
            input_bicubic_y_image = util.resize_image_by_pil(input_y_image, self.scale,
resampling_method=self.resampling_method)
            output_y_image = self.do(input_y_image, input_bicubic_y_image)
            psnr, ssim = util.compute_psnr_and_ssim(true_y_image, output_y_image,
 border_size=self.psnr_calc_border_size)

        elif true_image.shape[2] == 1 and self.channels == 1:

```

```python
            # for monochrome images
            input_image = loader.build_input_image(true_image, channels=self.channels, scale=self.scale,
                                                   alignment=self.scale)
            input_bicubic_y_image = util.resize_image_by_pil(input_image, self.scale,
                                                             resampling_method=self.resampling_method)
            output_image = self.do(input_image, input_bicubic_y_image)
            psnr, ssim = util.compute_psnr_and_ssim(true_image, output_image, border_size=self.psnr_calc_border_size)
        else:
            return None, None

        if print_console:
            print("[%s] PSNR:%f, SSIM:%f" % (file_path, psnr, ssim))

        return psnr, ssim

    def evaluate_bicubic(self, file_path, print_console=False):

        true_image = util.set_image_alignment(util.load_image(file_path, print_console=False), self.scale)

        if true_image.shape[2] == 3 and self.channels == 1:
            input_image = loader.build_input_image(true_image, channels=self.channels, scale=self.scale,
                                                   alignment=self.scale, convert_ycbcr=True)
            true_image = util.convert_rgb_to_y(true_image)
        elif true_image.shape[2] == 1 and self.channels == 1:
            input_image = loader.build_input_image(true_image, channels=self.channels, scale=self.scale,
                                                   alignment=self.scale)
        else:
            return None, None

        input_bicubic_image = util.resize_image_by_pil(input_image, self.scale, resampling_method=self.resampling_method)
        psnr, ssim = util.compute_psnr_and_ssim(true_image, input_bicubic_image, border_size=self.psnr_calc_border_size)

        if print_console:
            print("PSNR:%f, SSIM:%f" % (psnr, ssim))

        return psnr, ssim

    def init_train_step(self):
        self.lr = self.initial_lr
        self.epochs_completed = 0
        self.epochs_completed_in_stage = 0
        self.min_validation_mse = -1
        self.min_validation_epoch = -1
        self.step = 0

        self.start_time = time.time()

    def end_train_step(self):
        self.total_time = time.time() - self.start_time

    def print_steps_completed(self, output_to_logging=False):

        if self.step == 0:
            return

        processing_time = self.total_time / self.step
        h = self.total_time // (60 * 60)
        m = (self.total_time - h * 60 * 60) // 60
        s = (self.total_time - h * 60 * 60 - m * 60)

        status = "Finished at Total Epoch:%d Steps:%s Time:%02d:%02d:%02d (%2.3fsec/step) %d x %d x %d patches" % (
            self.epochs_completed, "{:,}".format(self.step), h, m, s, processing_time,
            self.batch_image_size, self.batch_image_size, self.training_images)

        if output_to_logging:
```

```python
            logging.info(status)
        else:
            print(status)

    def log_model_analysis(self):
        run_metadata = tf.RunMetadata()
        run_options = tf.RunOptions(trace_level=tf.RunOptions.FULL_TRACE)

        _, loss = self.sess.run([self.optimizer, self.loss], feed_dict={self.x: self.batch_input,
                                self.x2: self.batch_input_bicubic,
                                self.y: self.batch_true,
                                self.lr_input: self.lr,
                                self.dropout: self.dropout_rate},
                                options=run_options, run_metadata=run_metadata)

        # tf.contrib.tfprof.model_analyzer.print_model_analysis(
        #   tf.get_default_graph(),
        #   run_meta=run_metadata,
        #   tfprof_options=tf.contrib.tfprof.model_analyzer.PRINT_ALL_TIMING_MEMORY)
        self.first_training = False

```