Code Annotation

by Amy Hendrickson, TEXnology Inc, October, 2018

Example of Annotated Code. (Many more examples follow)

Listing 0.1 Hold-out Validation

```
                                                            Shuffling the data is
                                                            usually appropriate.
indices <- sample(1:nrow(data), size = 0.80 * nrow(data))  <-
evaluation_data <- data[-indices, ]              <- Defines the validation set

training_data <- data[indices, ]                <- Defines the training set

model <- get_model()                                     Trains a model on the
model %>% train(training_data)                           training data, and evaluates
validation_score <- model %>% evaluate(validation_data)  it on the validation data

model <- get_model()                             Once you've tuned your
model %>% train(data)                            hyperperameters, it's common to
test_score <- model %>% evaluate(test_data)      train your final model from scratch
                                                 on all non-test data available.
```

How to enter annotation. Numbers are defined and then used in the `\begin{code}...\end{code}` *environment:*

```
\def\1{\arrowtext/3/3.5/Shuffling the data is\\ usually appropriate./}
\def\2{\arrowtext/2/0/Defines the validation set/}
\def\3{\arrowtext/2.5/0/Defines the training set/}
\def\4{\noarrowRighttext/3/Trains a model on the\\ %
training data, and evaluates\\ it on the validation data/}
\def\5{\TwolinesRightDown/3/1/4/Once you've tuned %
your\\ hyperperameters, it's common to \\ train your final model from %
scratch\\ on all non-test data available./}


\codetitle{Hold-out Validation}

\begin{code}
indices <- sample(1:nrow(data), size = 0.80 * nrow(data))      \1
evaluation_data <- data[-indices, ]          \2

training_data <- data[indices, ]          \3

model <- get_model()
model \%>\% train(training_data)
validation_score <- model \%>\% evaluate(validation_data)\4

model <- get_model()
model \%>\% train(data)
test_score <- model \%>\% evaluate(test_data) \5
\end{code}
```

Notes on Entering Annotation Commands

Note 1: Because these characters are treated in LaTeX as "special use characters" we need to precede them with a backslash: `\%`, `\{`, `\}`, `\_`

In the `\begin{code}...\end{code}` environment if there is a command where you want the backslash to appear, it must be preceded with `\string`:
ie, `\string\codeexample`.

Note 2: To move annotations to the left, or down, use a negative number. To move annotations right or up, use a positive number.

Note 3: All of our annotation commands use `/` to separate arguments, so if you want to use `/` in the text, please use `$\slash$`, as you see in the ***Acrobat p. 149*** example below.

`\\` can be used to start new lines; ie,

```
\arrowtext/1.5/-2/x and y are 2D \\ tensors (matrices)./
```

Note 4: To make it easy to enter annotation commands in the body of a code sample, please make a definition for each annotation, using a number for the name of the definition.

Use this form:

```
\def\1{<command>/ / / ...}
```

You can then put `\1` in your code to get the results of your definition in the exact place where you want it to appear.

If you need more than 9 annotations for a particular piece of code, you should continue numbering with a10, b10 etc. For example:

```
\def\a10{<command>...},
\def\b10{<command>...},
\def\c10{<command>...}
```

Annotation Command List

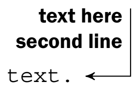
(Each unit = 9 pt)

I. All the things you can do with arrows

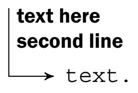
```
\arrowtext/(left or right)/(up or down)/(text)/
```

The `\arrowtext` first argument is distance from the arrow, either left or right; the second argument is height or depth of line going from the end of the arrow. The third argument is for the text which will be placed at the top of a line ending upwards, or at the bottom of a line ending downwards.

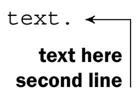
```
text. \arrowtext/2/3/text here\\ second line/ =
```



```
\arrowtext/-2/3/text here\\ second line/text. =
```

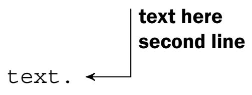


```
text.\arrowtext/2/-3/text here\\ second line/ =
```

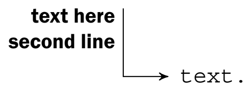


Add `\outside` to the code examples above and the text at the end of the arrow will go to the outside of the page:

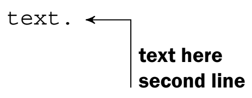
```
text. \outside\arrowtext/2/3/text here\\ second line/ =
```



```
\outside\arrowtext/-2/3/text here\\ second line/text. =
```



```
text.\outside\arrowtext/2/-3/text here\\ second line/ =
```

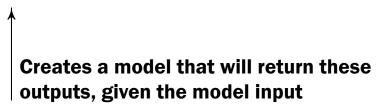


Arrow pointing up:

```
\arrowtext/(line to left or right)/(height or depth of arrow)/text/
```

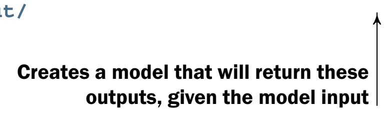
To make a single line pointing up, use '`0`' for first argument. Since we only want one vertical line we set the horizontal line to zero units:

```
\arrowtext/0/-4/Creates a model that will return these\\
 outputs, given the model input/
```


Creates a model that will return these
outputs, given the model input

Adding `\outside` to make the text go to the left:

```
\outside\arrowtext/0/-4/Creates a model that will return these\\
 outputs, given the model input/
```


Creates a model that will return these
outputs, given the model input

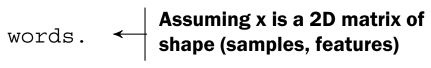
Arrow with perpendicular line:

```
\lineheight=(height of perpendicular line)\vertline
\arrowtext/(line to left or right)/(height or depth of arrow)/text/
```

Adding the command `\vertline` will make the horizontal line bump into a vertical line. The height of the line is determined with `\lineheight=<number of units>`. You can make the arrow point to the right by supplying a negative first argument, as you see below. The second argument to `\arrowtext` must be '0'.

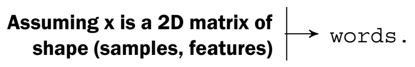
Point arrow to the left:

```
words. \lineheight=2\vertline\arrowtext/1.5/0/Assuming x is a
2D matrix of\\ shape (samples, features)/
```



Point arrow to the right with negative first argument:

```
\lineheight=2\vertline\arrowtext/-1.5/0/Assuming x is a 2D
matrix of \\ shape (samples, features)/
```



Arrow with only a horizontal straight line:

```
\arrowtext/(left or right)/(up or down, must be '0')/(text)/}
```

To make a single horizontal line, use '`0`' for second argument. Since we only want one horizontal line we set the vertical line to zero units:

`text. \arrowtext/2.5/0/Installs the Keras R  package/` =

text. ← **Installs the Keras R package**

II. No Arrows

```
\noarrowRighttext/(height of vertical line)/(text to right)/
\noarrowLefttext/(height of vertical line)/(text to left)/
```

`words.\noarrowRighttext/2/top line\\ bottom line/` =

words. | **top line**
bottom line

`\noarrowLefttext/2/top line\\ bottom line/ words.` =

top line
bottom line | words.

noarrowLinetext

```
\lineheight=<number of units>
\noarrowLinetext/(left or right)/(up or down)/(text).
```

`\noarrowLinetext` works similar to `\arrowtext`: in the first argument a negative number means move the line towards the left; a positive number means move towards the right. For the second argument, a negative number means move down, a positive number means move up. `\lineheight` means the height of the starting vertical line.

```
words. \lineheight=2
\noarrowLinetext/2/6/Text\\ more text\\ still more./
```

Text
more text
still more.

words.

And with negative arguments:

```
\lineheight=2
\noarrowLinetext/-2/-6/Text\\ more text\\ still more./words.
```

words.

Text
more text
still more.

Preceding `\noarrowLinetext` with `\outside` will shift the text to the outside of the page, as we've seen with `\arrowtext`.

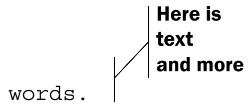
III. Two vertical lines with diagonal line joining the center point of each

Outside of two lines raised:

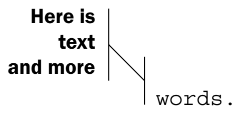
```
\TwolinesRightUp/(height of left line)/
(raise outer line by this much)/(height of outer line)/text/}}
```

```
\TwolinesLeftUp/(height of left line)/
 (raise outer line by this much)/(height of outer line)/text/
```

`words.\TwolinesRightUp/2/1/3/Here is\\ text\\ and more/` =



`\TwolinesLeftUp/3/1/2/Here is\\ text\\ and more/words.` =

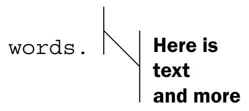


Outside of two lines lowered:

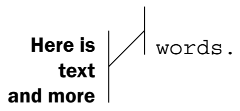
```
\TwolinesRightDown/(height of left line)/
(lower outer line by this much)/(height of inner line)/(text)/
```

```
\TwolinesLeftDown/(height of left line)/
 (lower outer line by this much)/(height of inner line)/(text)/
```

`words.\TwolinesRightDown/2/1/3/Here is\\ text\\ and more/` =



`\TwolinesLeftDown/3/1/2/Here is\\ text\\ and more/words.` =



Examples from *Deep Learning with R.*

Acrobat p. 58

```
naive_relu <- function(x) {      ← x is a 2D tensor
  for (i in nrow(x))               (R matrix).
    for (j in ncol(x))
      x[i, j] <- max(x[i, j], 0)
  x
}
```

```
\def\1{\outside\arrowtext/1.5/-2/x is a 2D tensor\\ (R matrix)./}
\begin{code}
naive_relu <- function(x) \{ \1
  for (i in nrow(x))
    for (j in ncol(x))
      x[i, j] <- max(x[i, j], 0)
  x
\}
\end{code}
```

Acrobat p. 60

In this example we use `\vskip9pt` to lower the text, to make the top line of text line up with arrow:

```
naive_matrix_vector_dot <- function(x, y) {  ←—— x is a 2D tensor (matrix).
                                                  y is a 1D tensor(vector).
  z <- rep(0, nrow(x))
  for (i in 1:nrow(x))
    for (j in 1:ncol(x))
      z[[i]] <- z[[i]] + x[[i, j]] * y[[j]]
  z
}
```

```
\def\1{\arrowtext/3/0/\vskip9pt
x is a 2D tensor (matrix).\\ y is a 1D tensor(vector)./}
\begin{code}
naive_matrix_vector_dot <- function(x, y) \{ \1

  z <- rep(0, nrow(x))
  for (i in 1:nrow(x))
    for (j in 1:ncol(x))
      z[[i]] <- z[[i]] + x[[i, j]] * y[[j]]
  z
\}
\end{code}
```

Acrobat p. 79

```
install.packages("keras")  ← Installs the Keras R package

library(keras)            | Installs the core Keras
install_keras()           | library and TensorFlow
```

```
\def\1{\arrowtext/2.5/0/Installs the Keras R  package/}
\def\2{\noarrowRighttext/2/Installs the core Keras\\ library
and TensorFlow/}

\begin{code}
install.packages(”keras”)\1

library(keras)
install_keras()    \2
\end{code}
```

Acrobat p. 115

```
x <- scle(x)  ←| Assuming x is a 2D matrix of
               | shape (samples, features)
```

```
\def\1{\lineheight=2\vertline\arrowtext/1.5/0/Assuming x is a 2D matrix of
\\ shape (samples, features)/
}
\begin{code}
x <- scle(x) \1
\end{code}
```

Acrobat p. 149

Notice that `$\slash$` instead of `/` must be used in the third argument or it would be interpreted as the end of the argument.

```
train_datagen <- image_data_generator(rescale = 1/255)       | Rescales all
validation_datagen <- image_data_generator(rescale = 1/255)  | images by 1/255
```

```
\def\1{\noarrowRighttext/2/Rescales all\\ images by 1$\slash$255/}

\begin{code}
train_datagen <- image_data_generator(rescale = 1/255)
validation_datagen <- image_data_generator(rescale = 1/255)\1
\end{code}
```

Acrobat p. 103

```
num_epochs <- 500 all_mae_histories <- NULL
for (i in 1:k) {
  cat("processing fold #", i, "\n")

  val_indices <- which(folds == i, arr.ind = TRUE)
  val_data <- train_data[val_indices,]
  val_targets <- train_targets[val_indices]

  partial_train_data <- train_data[-val_indices,]
  partial_train_targets <- train_targets[-val_indices]

model <- build_model()

  history <- model %>% fit(
    partial_train_data, partial_train_targets,
    validation_data = list(val_data, val_targets),
    epochs = num_epochs, batch_size = 1, verbose = 0
  )
  mae_history <- history$metrics$val_mean_absolute_error
  all_mae_histories <- rbind(all_mae_histories, mae_history)
}
```

Prepares the validation data: data from partition #k

Prepares the training data: data from all other partitions

Builds the Keras model (already compiled)

Trains the model (in silent mode, verbose=0)

```
\def\1{\outside\arrowtext/2/3/Prepares the\\ validation data: data\\ from
partition \#k/}
\def\2{\outside\arrowtext/2/3/Prepares the training\\
data: data from all\\ other partitions/}
\def\3{\outside\arrowtext/2/-3/Trains the model (in\\ silent mode,\\
verbose=0)/}
\def\4{\outside\arrowtext/-2/-5.5/Builds\\ the Keras\\ model\\(already\\
compiled)/}

\begin{code}
num_epochs <- 500 %
all_mae_histories <- NULL
for (i in 1:k) \{
  cat(”processing fold \#”, i, ”\string\n”)

  val_indices <- which(folds == i, arr.ind = TRUE)   \1
  val_data <- train_data[val_indices,]
  val_targets <- train_targets[val_indices]

  partial_train_data <- train_data[-val_indices,]   \2
  partial_train_targets <- train_targets[-val_indices]

\4model <- build_model()

  history <- model \%>\% fit(                         \3
    partial_train_data, partial_train_targets,
    validation_data = list(val_data, val_targets),
    epochs = num_epochs, batch_size = 1, verbose = 0
  )
  mae_history <- history$metrics$val_mean_absolute_error
  all_mae_histories <- rbind(all_mae_histories, mae_history)
\}
\end{code}
```

Acrobat p. 152

```
augmentation_generator <- flow_images_from_data(
  img_array,
  generator = datagen,
  batch_size = 1
)

op <- par(mfrow = c(2, 2), pty = "s", mar = c(1, 0, 1, 0))
for (i in 1:4) {
  batch <- generator_next(augmentation_generator)
  plot(as.raster(batch[1,,,]))
}
par(op)
```

Generates batches of randomly transformed images. Loops indefinitely, so you need to break the loop at some point.

Plots the images

```
\def\1{\TwolinesRightDown/4.5/2/4/Generates batches of
randomly\\  transformed images. Loops\\
indefinitely, so you need to\\ break the loop at some point./}

\def\2{\noarrowRighttext/7/Plots the images\vskip2.8\baselineskip/}

\begin{code}
augmentation_generator <- flow_images_from_data(
  img_array,
  generator = datagen,
  batch_size = 1                                  \1
)

op <- par(mfrow = c(2, 2), pty = ”s”, mar = c(1, 0, 1, 0))
for (i in 1:4) \{
  batch <- generator_next(augmentation_generator)
  plot(as.raster(batch[1,,,]))
\}
par(op)                                                     \2
\end{code}
```

Acrobat p. 171

Showing up arrow. Notice that you need to use `\outside` to make text go to the right.

```
layer_outputs <- lapply(model$layers[1:8], function(layer) layer$output)
activation_model <- keras_model(inputs = model$input,outputs=layer_outputs)
```

Extracts the outputs of the top eight layers

Creates a model that will return these outputs, given the model input

```
\def\1{\arrowtext/-2.5/-6.5/Extracts the outputs of\\ the top eight
layers/}

\def\2{\outside\arrowtext/0/-4/Creates a model that will return these\\ outputs,
given the model input/}
\begin{code}
\1layer_outputs <- lapply(model$layers[1:8], function(layer) layer$output)
activation_model <- keras_model(inputs = model$input,outputs=layer_outputs)
                                                                \2
\end{code}
```

Acrobat p. 178

```
iterate <- k_function(list(model$input), list(loss, grads))
```

Returns the loss and grads given the input picture

```
\def\1{\lineheight=3\vertline\arrowtext/3/0/Returns the loss\\
and grads given\\ the input picture/}
\begin{code}
iterate <- k_function(list(model$input), list(loss, grads)) \1
\end{code}
```

Acrobat p. 184

Listing 0.2 Setting up the Grad-CAM algorithm

”African elephant” entry in the prediction vector

Output feature map of the block5_conv3 layer, the last convolutional layer in YGG 16

Gradient of the “African elephant” class with regard to the output feature map of block5_conv3

```
african_elephant_output <- model$output[, 387]

last_conv_layer <- model %>% get_layer("block5_conv3")

grads<-k_gradients(african_elephant_output,last_conv_layer$output)[[1]]

pooled_grads <- k_mean(grads, axis = c(1, 2, 3))

iterate <- k_function(list(model$input),
                      list(pooled_grads, last_conv_layer$output[1,,,]))

c(pooled_grads_value, conv_layer_output_value) %<-% iterate(list(img))

for (i in 1:512)
  conv_layer_output_value[,,i] <-
    conv_layer_output_value[,,i] * pooled_grads_value[[i]]

heatmap <- apply(conv_layer_output_value, c(1,2), mean)
```

The channel-wise mean of the resulting feature-map is the heatmap of the class activation.

Lets you access the values of the quantities you just defined: pooled_grads and the output feature-map of block5_conv3, given a sample image

Multiplies each channel in the feature-map array by “how important this channel is” with regard to the “elephant” class

Vector of shape (512) where each entry is the mean intensity of the gradient over a specific feature-map channel

Values of these two quantities, given the sample image of two elephants

(LaTeX Code on following page)

```
\codetitle{Setting up the Grad-CAM algorithm}
\vskip30pt

\def\1{\arrowtext/-1/4/''African elephant'' entry in\\ the
prediction vector/}%

\def\2{\arrowtext/1/7/Output feature map of the\\ block5\_conv3
layer, the last\\ convolutional layer in YGG 16/}

\def\3{\arrowtext/3/9/Gradient of the\\
"African elephant"\\
class with regard to\\
the output feature\\
map of block5\_conv3/}

\def\4{\arrowtext/-2/-27/Vector of shape (512) where each entry\\ is %
the mean intensity of the gradient\\ over a specific feature-map %
channel/}

\def\5{\arrowtext/-1/-20.5/Lets you access the values of the %
quantities\\ you just defined: pooled\_grads and the\\ output %
feature-map of block5\_conv3, given\\ a sample image/}

\def\6{\arrowtext/4/-21.7/Values of these two quantities, given\\ the %
sample image of two elephants/}

\def\7{\arrowtext/12/-16/%
Multiplies each channel in the\\ feature-map array by ``how\\ %
important this channel is'' with\\ regard to the ``elephant'' class/}

\def\8{\arrowtext/3/-5.5/The channel-wise mean of the\\ resulting %
feature-map is the\\ heatmap of the class activation./}

\begin{code}
\1african_elephant_output <- model$output[, 387]

last_conv_layer <- model \%>\% get_layer("block5_conv3")\2

grads<-k_gradients(african_elephant_output,last_conv_layer$output)[[1]]~\3


\4pooled_grads <- k_mean(grads, axis = c(1, 2, 3))

\5iterate <- k_function(list(model$input),
                      list(pooled_grads, last_conv_layer$output[1,,,]))

c(pooled_grads_value, conv_layer_output_value) \%<-\% iterate(list(img))\6

for (i in 1:512) {                                  %
\7
  conv_layer_output_value[,,i] <-
    conv_layer_output_value[,,i] * pooled_grads_value[[i]]
}

heatmap <- apply(conv_layer_output_value, c(1,2), mean)  %
\8

\end{code}
```

Acrobat p. 255

Building the left branch of the model: inputs are variable-length sequences of vectors of size 128.

Building the right branch of the model: when you call an existing layer instance, you reuse its weights.

```
left_input <- layer_input(shape = list(NULL, 128))
left_output <- left_input %>% lstm()

right_input <- layer_input(shape = list(NULL, 128))
right_output <- right_input %>% lstm()

merged <- layer_concatenate(list(left_output, right_output))

predictions <- merged %>%
  layer_dense(units = 1, activation = "sigmoid")

model <- keras_model(list(left_input, right_input), predictions)
model %>% fit(
  list(left_data, right_data), targets)
)
```

Builds the classifier on top

Instantiating and training the model: when you train such a model, the weights of the LSTM layer are updated based on both inputs.

```
\def\1{\lineheight=2\noarrowLinetext/-2/6/Building the left branch of
the\\  model: inputs are
variable-length\\ sequences of vectors of size 128./}

\def\2{\outside\lineheight=2\noarrowLinetext/2/7/Building the right\\
branch of the model:\\ when you call an\\ existing layer instance,\\
you reuse its weights./}

\def\3{\noarrowRighttext/2/Builds the classifier on top\vskip5pt/}

\def\4{\lineheight=4\noarrowLinetext/-2/-7/Instantiating and training the
model: when you\\ train such a model, the weights of the LSTM\\
layer are updated based on both inputs./}

\begin{code}
left_input <- layer_input(shape = list(NULL, 128))
\1left_output <- left_input \%>\% lstm()

right_input <- layer_input(shape = list(NULL, 128))
right_output <- right_input \%>\% lstm()               \2

merged <- layer_concatenate(list(left_output, right_output))

predictions <- merged \%>\%
  layer_dense(units = 1, activation = ”sigmoid”)   \3

model <- keras_model(list(left_input, right_input), predictions)
model \%>\% fit(
  list(left_data, right_data), targets)
\4)
\end{code}
```

Acrobat p. 256

```
merged_features <- layer_concatenate(
  list(left_features, right_features)
}
```

The merged features contain information from the right visual feed and theleft visual feed.

```
\def\1{\noarrowRighttext/3.2/The merged features contain\\
information from the right visual\\ feed and theleft visual feed.\vskip1pt/}

\begin{code}
merged_features <- layer_concatenate(
  list(left_features, right_features)
\}                                    \1
\end{code}
```

Acrobat p. 258

Example of how to center text vertically, using `\vskip14pt`. The size of the skip is up to you to choose, perhaps after a bit of experimentation.

```
model %>% fit(
  x, y,
  epochs = 10,
  batch_size = 32,
  callbacks = callbacks_list,
  validation_data = list(x_val, y_val)
)
```

Because the callback will monitor the validation loss, you need to pass validation_data to the call to fit.

```
\def\1{\noarrowRighttext/8.5/Because the callback will\\
monitor the validation loss, you\\ need to pass validation\_data
to the  call to fit.\vskip24pt/
}
\begin{code}
model \%>\% fit(
  x, y,
  epochs = 10,
  batch_size = 32,
  callbacks = callbacks_list,
  validation_data = list(x_val, y_val)
)                                       \1
\end{code}
```

Acrobat p. 286

```
gradient_ascent <- function(x, iterations, step, max_loss = NULL) {
  for (i in 1:iterations) {
    c(loss_value, grad_values) %<-% eval_loss_and_grads(x)
    if (!is.null(max_loss) && loss_value > max_loss)
      break
    cat("...Loss value at", i, ":", loss_value, "\n")
    x <- x + (step * grad_values)
  }
  x
}
```

This function runs gradient ascent for a number of iterations.

```
\def\1{\noarrowLefttext/12.5/This function runs\\
gradient ascent for a\\ number of iterations./
}

\begin{code}
gradient_ascent <- function(x, iterations, step, max_loss = NULL) \{
  for (i in 1:iterations) \{
    c(loss_value, grad_values) \%<-\% eval_loss_and_grads(x)
    if (!is.null(max_loss) && loss_value > max_loss)
      break
    cat(”...Loss value at”, i, ”:”, loss_value, ”\string\n”)
    x <- x + (step * grad_values)
  \}
  x
\}                                                                       \1
\end{code}
```

Acrobat p. 293

```
deprocess_image <- function(x) {
  x <- x[1,,,]
  x[,,1] <- x[,,1] + 103.939
  x[,,2] <- x[,,2] + 116.779
  x[,,3] <- x[,,3] + 123.68
  x <- x[,,c(3,2,1)]
  x[x > 255] <- 255
  x[x < 0] <- 0
  x[] <- as.integer(x)/255
  x
}
```

Zero-centers by removing the mean pixel value from ImageNet. This reverses a transformation done by Imagenet_preprocess_Input.

Converts images from ‘BGR’ to ‘RGB’. This is also part of the reversal of Imagenet_preprocess_Input.

```
\def\1{\TwolinesRightUp/3.5/0/4/Zero-centers by removing the
mean\\ pixel value from ImageNet. This\\ reverses a transformation\\
done by Imagenet\_preprocess\_Input./}

\def\2{\outside\arrowtext/1.5/-4.2/Converts images from ‘BGR’ to ‘RGB’.\\
This is also part of the reversal of\\
Imagenet\_preprocess\_Input./}
\begin{code}
deprocess_image <- function(x) \{
  x <- x[1,,,]
  x[,,1] <- x[,,1] + 103.939
  x[,,2] <- x[,,2] + 116.779
  x[,,3] <- x[,,3] + 123.68    \1
  x <- x[,,c(3,2,1)]         \2
  x[x > 255] <- 255
  x[x < 0] <- 0
  x[] <- as.integer(x)/255
  x
\}
\end{code}
```

Acrobat p. 295

Listing 0.3 Defining the final loss that you'll minimize

Named list that maps layer names to activation tensors

Layer used for content loss

```
outputs_dict <- lapply(model$layers, `[[`, "output")
names(outputs_dict) <- lapply(model$layers, `[[`, "name")

content_layer <- "block5_conv2"
style_layers = c("block1_conv1", "block2_conv1",
                 "block3_conv1", "block4_conv1",
                 "block5_conv1")
```

Layers used for style loss

```
total_variation_weight <- 1e-4
style_weight <- 1.0
content_weight <- 0.025
```

Weights in the weighted average of the loss components

You'll define the loss by adding all components to this scalar variable.

```
loss <- k_variable(0.0)
layer_features <- outputs_dict[[content_layer]]
target_image_features <- layer_features[1,,,]
combination_features <- layer_features[3,,,]
loss <- loss + content_weight * content_loss(target_image_features,
                                             combination_features)
```

Adds the content loss.

```
for (layer_name in style_layers) {
  layer_features <- outputs_dict[[layer_name]]
  style_reference_features <- layer_features[2,,,]
  combination_features <- layer_features[3,,,]
  sl <- style_loss(style_reference_features, combination_features)
  loss <- loss + ((style_weight / length(style_layers)) * sl)
}
loss <- loss +
loss (total_variation_weight * total_variation_loss(combination_image))
```

Adds a style loss component for each target layer

Adds the total variation loss

```
\codetitle{Defining thefinal loss that you'll minimize}
\def\1{\arrowtext/-1/4/Named list that maps layer\\ names to
activation tensors/}
\def\2{\arrowtext/7/7.25/Layer used for content loss/}
\def\3{\noarrowRighttext/3.5/Layers used for style loss\vskip12pt/}
\def\4{\noarrowRighttext/3/Weights in the\\ weighted average
of\\ the loss components/}
\def\5{\outside\arrowtext/11/3/You'll define the loss by\\ adding
all components\\ to this scalar variable./}
\def\6{\noarrowLefttext/5.75/Adds the content loss.\vskip32pt/}
\def\7{\outside\arrowtext/8/-2/Adds a style loss component\\
for each target layer/}
\def\8{\TwolinesLeftUp/4/0/2/Adds the\\ total\\ variation\\loss/}
```

LaTeX code continued on next page.

```
\begin{code}
\vskip36pt
\1outputs_dict <- lapply(model$layers, `[[`, ”output”)
names(outputs_dict) <- lapply(model$layers, `[[`, ”name”)

content_layer <- ”block5_conv2”                \2
style_layers = c(”block1_conv1”, ”block2_conv1”,
                 ”block3_conv1”, ”block4_conv1”,
                 ”block5_conv1”)                    \3

total_variation_weight <- 1e-4
style_weight <- 1.0
content_weight <- 0.025         \4

loss <- k_variable(0.0)             \5
layer_features <- outputs_dict[[content_layer]]
target_image_features <- layer_features[1,,,]
combination_features <- layer_features[3,,,]
loss <- loss + content_weight * content_loss(target_image_features,
                                             combination_features) %
    \6

for (layer_name in style_layers) \{    \7
  layer_features <- outputs_dict[[layer_name]]
  style_reference_features <- layer_features[2,,,]
  combination_features <- layer_features[3,,,]
  sl <- style_loss(style_reference_features, combination_features)
  loss <- loss + ((style_weight / length(style_layers)) * sl)
\}
loss <- loss +
\8loss (total_variation_weight * total_variation_loss(combination_image))
\end{code}
```