StreetView House Numbers: Tensorflow Model

Oliver Morris

26 July 2017

Summary

This notebook details the method used to create a convolutional neural network (CNN) whose aim is to identify house numbers from photos. Those photos are made available in the SVHM dataset, which is from Google streetview.

The work is part of the capstone project for the Udacity 730 Tensorflow course. It follows the approach of the google research team in the paper Goodfellow, et al 2013.

96.4% Accuracy

The model achieves 96.4% accuracy on its test set, whereas the Google team achieved 97.8%. So a surprisingly high performance can be achieved with this 'home' setup of tensorflow on an 'entry level' gaming PC.

The hardware used was a Windows 10 PC 64bit with Core i5-7500 (3.4GHz) 16GB RAM, NVIDIA GeForce GTX1050Ti (4GB). This GPU retails for GBP 135.00. The model was trained in 20 epochs, over 8hrs, on this GPU.

Tensorflow within R, not Python

Unusually, this notebook has been completed in R, the original course is in Python. This is possible thanks to the Tensorflow package for R provided by R Studio. The syntax is broadly the same as it would be in python, the most notable difference being that "$" replaces "." and that R occassionally submits lists or tuples where python would present square brackets [] to tensorflow. Also, R is 1 based for indexing, whereas python/tensorflow are 0 based, which can be confusing when viewing them in a single code chunk

Source Data

An associated notebook, published on GitHub by this author, pre-processes the data in R. This notebook assumes that pre- procesing has been completed.

```r
library(readr)
setwd('~/CNN_data')

reconstitute_x <- function(x_table_filename){

  # load the file, assumes already in correct folder
  # NB convert grayscale integers to numeric (i.e float32),
  # which is required for Tensorflow 'x' values (ie examples)
  format_tab <- suppressMessages(read_delim(as.character(x_table_filename),
                                            col_types=cols(y=col_integer(),
                                                           x=col_integer(),
                                                           value=col_double(),
                                                           Sequence=col_integer()),
                                            delim=' '))
  # Reshape to images_rows=32, image_cols=32, image_value=1, image_qty=nrow
  format_arr <- array(format_tab$value,c(32,32,1,max(format_tab$Sequence)))

  # Tensorflow expects the image qty first, followed by x and y. So, reshape accordingly
  # If you attempt this shape first then x and y values get in terrible mess.
  return(aperm(format_arr, c(4,1,2,3)))

  # To confirm, we should have returned an array (aka Tensor)
  # where shape = (image_qty,32,32,1)
}

reconstitute_y <-function(y_table_filename){

  # load the file, assumes already in correct folder
  format_tab <- suppressMessages(read_delim(as.character(y_table_filename), delim=' '))

  # convert to array so Tensorflow can use the R object as a Tensor.
  # Also, drop 'sequence' and 'digit count' columns such that only the LabelsA-E are returned
  return(matrix(as.matrix(format_tab[,3:7]), ncol=5))

  # to confirm, we should have returned a matrix (aka Tensor)
  # where shape = (rows,5). Column1=LabelA, Column5=LabelE
}

x_validate <- reconstitute_x('imagetensor_validate.txt')
x_test     <- reconstitute_x('imagetensor_test.txt')
x_train    <- reconstitute_x('imagetensor_train.txt')

y_validate <- reconstitute_y('labeltensor_validate.txt')
y_test     <- reconstitute_y('labeltensor_test.txt')
y_train    <- reconstitute_y('labeltensor_train.txt')
```

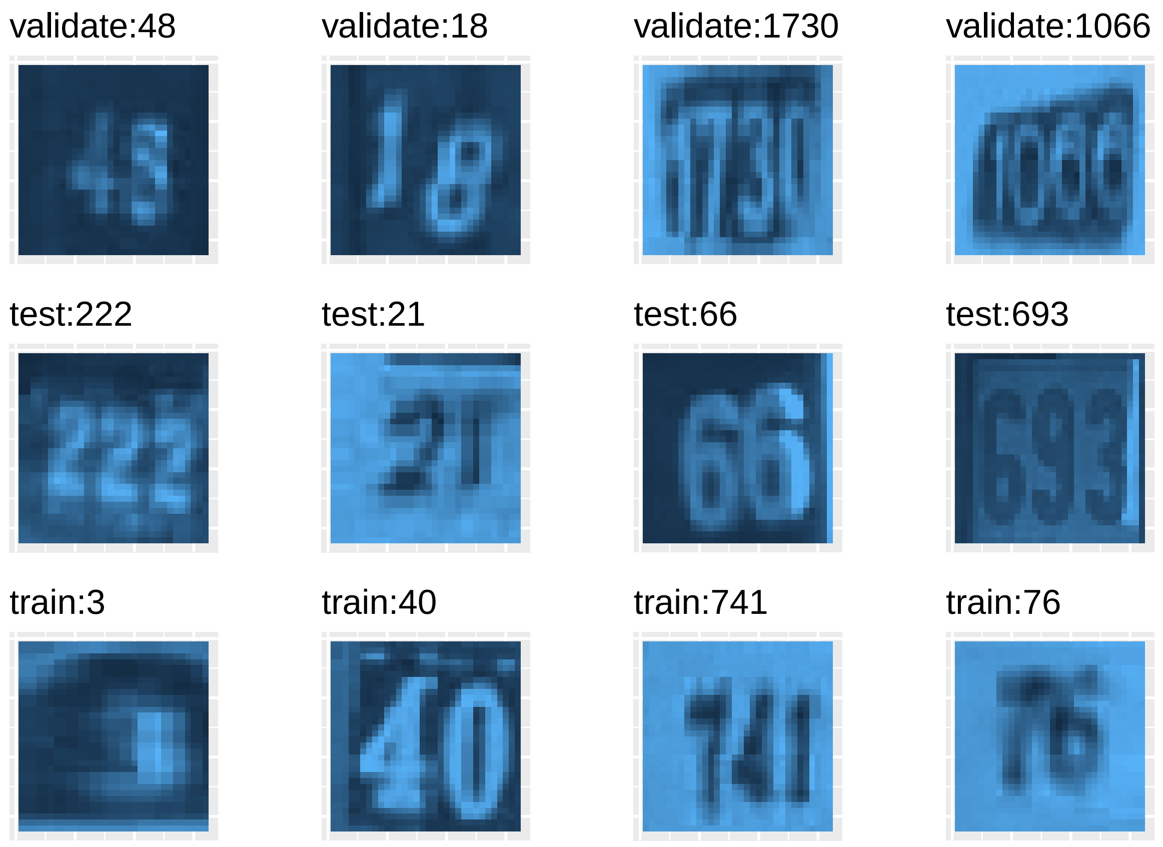
Let's take a look at the data, nine examples (three from each set) to ensure it makes sense:

```
library(reshape2)
library(ggplot2)
library(grid)
library(gtable)
library(gridExtra)
setwd('~/CNN_data')

# create function for displaying images
displayImage <- function(x_sampleset, y_sampleset, title, id_range){
  listOfPlots <- list()
  for(i in 1:length(id_range)){
    local({
      i<-i
      image <- melt(x_sampleset[id_range[i],,,1])
      image <- as.data.frame(image)
      colnames(image) <- c('y','x','value')
      suppressWarnings(
      theplot <- ggplot(image,aes(x,y)) +
                   geom_raster(aes(fill=value)) +
                   theme(axis.title.x=element_blank(),
                              axis.text.x =element_blank(),
                              axis.ticks.x=element_blank(),
                              axis.title.y=element_blank(),
                              axis.text.y =element_blank(),
                              axis.ticks.y=element_blank()
                              ) +
                        scale_y_continuous(trans=scales::reverse_trans()) +
                        coord_fixed() +
                        ggtitle(paste0(title,
                                       ':',
                                       paste(y_sampleset[id_range[i],
                                                         which(y_sampleset[id_range[i],]!=10)],
                                             collapse=''))) +
                        guides(fill=FALSE))
      listOfPlots[[i]] <<- theplot
      })
  }
  return(listOfPlots)
}

#let's randomly select 12 images to display; 4 from train, 4 from test and 4 from validate
IDforValidate <- sample(1:nrow(y_validate), 4, replace=FALSE)
IDforTest     <- sample(1:nrow(y_test), 4, replace=FALSE)
IDforTrain    <- sample(1:nrow(y_train), 4, replace=FALSE)

grobs1 = lapply(displayImage(x_validate, y_validate, 'validate', IDforValidate), ggplotGrob)
grobs2 = lapply(displayImage(x_test, y_test, 'test', IDforTest), ggplotGrob)
grobs3 = lapply(displayImage(x_train, y_train, 'train', IDforTrain), ggplotGrob)
grid.arrange(grobs=c(grobs1,grobs2,grobs3), ncol=4)
```



How to Approach TensorFlow Models

Tensorflow demands we define our model (aka graph) before executing anything. This means defining the Variables, Model, Optimisation Method, Evaluation method and Feeding method (for spooning the data into the model). So the majority of this notebook is dedictated to definitions before finally 'instantiating' the model (with 'tf.session') and then 'executing' the model (with 'tf.session.run').

Helper Functions for Variables

Tensorflow models use 'variables' and 'placeholders'. Variables are parameters that exist only within the model, such as weights or biases. Placeholders hold a place for parameters which must be fed into the model, such as training examples, x, labels, y, or dropout proportions.

Its helpful in defining a clean, easy to read model if some of the repetitive definition is handed over to 'Helper functions'. This notebook will create helpers for Variables, Layers and the Feeding process (more on that later). The below code chunk defines helper functions for Variables,i.e. weights and biases.

One should generally initialize weights with a small amount of noise for symmetry breaking, and to prevent 0 gradients. Since we're using ReLU activation, it is also good practice to initialize neurons with a slightly positive initial bias to avoid 'dead neurons'.

Xavier Initialisation

This model has 13 layers, 7 of which are stacked on top of each other, i.e. it has appreciable depth. To assist small signals pass deep into the layers and prevent large signals dominating later layers, this model uses the Xavier initialisation, which is based on the number of input and output neurons. There is a simple introduction to Xavier at Andy Jones blog.

```r
library(tensorflow)

initialise_weights_conv <- function(shape, name){
  initial <- tf$get_variable(name, shape, initializer=tf$contrib$layers$xavier_initializer_conv2d())
  return(initial)
}

initialise_weights_dense <- function(shape, name){
  initial <- tf$get_variable(name, shape, initializer=tf$contrib$layers$xavier_initializer())
  return(initial)
}

initialise_bias <- function(shape, name){
  initial <- tf$constant(0.0, shape=shape)
  return (tf$Variable(initial, name=name))
}
```

Helper Functions for Layers

There are two basic types of layer, convolutional and dense, as per other CNN's. However, this model will be a little unusual in that the output layer will be comprised of 5 dense layers laid in parallel. Each such layer's task will be to identify the digit (1-12) for each of the 5 possible digits in a sequence.

There will be 3 convolutional layers, then 2 dense layers, then the 5 parallel layers.

All convolutional layers will have a similar definition; - input tensor shape - Filter details: – filter quantity (more filters means more identification power) – filter spatial dims (eg 5px x 5px) – filter depth (eg 1 for grayscale) – stride/padding details (eg stride = 1px, padding = 0px) - relu activation - max pooling

All dense (aka fully connected) layers will also share similar definitions: - 2D weights [input quantity, output quantity]. – input qty is the input tensor volume, for example, 6px x 6px x 64channels deep. – output qty is the number of neurons on the layer - biases, which must be the same as the number of neurons on the layer

Since there are so many layers and so many similarities between those layers, its worthwhile creating some helper functions.

```r
# Helper function for creating convolutional layers

conv2d<-function(input_tensor,
                 filter_dim_x,
                 filter_dim_y,
                 filter_depth,
                 filter_qty,
                 maxpool,
                 tensorbrd_name){

  # Add layer name scopes for better graph visualization
  with(tf$name_scope(tensorbrd_name),{

    # Initialise weights and biases
    weights <- initialise_weights_conv(
                 shape= shape(filter_dim_x, filter_dim_y, filter_depth, filter_qty),
                 name = paste0(tensorbrd_name, '_weights'))
    biases  <- initialise_bias(
                  shape= shape(filter_qty),
                  name = paste0(tensorbrd_name,'_biases'))

    # Add histogram summaries for weights
    tf$summary$histogram(paste0(tensorbrd_name,'_weights'), weights)

    # Use tf.nn.conv2d to create the convolution, zero padding (aka 'SAME')
    strides <- c(1L, 1L, 1L, 1L)
    result <- tf$nn$conv2d(input_tensor, weights, strides, 'SAME')+biases

    # activate the result
    result_activated <- tf$nn$relu(result)

    # Add histogram summaries for activations
    tf$summary$histogram(paste0(tensorbrd_name,'_activations'), result_activated)

    if(maxpool==TRUE){
      # max pool it. Uses 2x2 pool, so for example, 32x32 image would be max pooled to 16x16
      result_activated <- tf$nn$max_pool(value   = result_activated,
                                         ksize   = c(1L, 2L, 2L, 1L), #kernel, ie snapshot, size
                                         strides = c(1L, 2L, 2L, 1L), #pixel strides per snapshot
                                         padding = 'SAME')
    }
    # Return the resulting layer
    return(result_activated)
    })
}
```

```
# Helper function for creating dense layers

dense1d <- function(input_tensor,   # The previous layer,
                    input_dim,      # Previous input size. Eg, prior layer with 8x8x64deep = 4096.
                    output_dim,     # The qty of neurons is the qty of outputs, = qty of biases.
                    tensorbrd_name){# The layer name

  # Add layer name scopes for better graph visualization
  with(tf$name_scope(tensorbrd_name),{

        # Create new weights and biases.
        weights <- initialise_weights_dense(shape= shape(input_dim, output_dim),
                                            name = paste0(tensorbrd_name,'_weights'))
        biases  <- initialise_bias(shape= shape(output_dim),
                                  name = paste0(tensorbrd_name,'_biases'))

        # Add histogram summaries for weights
        tf$summary$histogram(paste0(tensorbrd_name,'_weights'), weights)

        # Calculate the layer
        result = tf$matmul(input_tensor, weights)+biases

        #no activation here, just return the result
        return(result)
  })
}
```

Placeholders

As stated earlier, placeholders are the values which need to be fed into the model, unlike weights or biases, they are not generated within the model. Here we define the placeholders for x, y and dropout. Dropout is revisited in the next code chunk.

```
#Establish placeholders for inputs (x and y) and drop out proportion.

with(tf$name_scope('placeholder_input'),{
  # images are 32px x 32px x grayscale. Permit float, not just int
  x_placeholder <- tf$placeholder(tf$float32, shape=shape(NULL, 32L, 32L, 1L), name='x')

  # there will be 5 labels for each image, digitsA-E
  y_placeholder <- tf$placeholder(tf$int64, shape=shape(NULL, 5L), name='y')

})

with(tf$name_scope('placeholder_dropout'),{

  keepprob_placeholder_1 <- tf$placeholder(tf$float32)
  keepprob_placeholder_2 <- tf$placeholder(tf$float32)
  keepprob_placeholder_3 <- tf$placeholder(tf$float32)

  # present keep_prob in the summary
  tf$summary$scalar('keep_prob_1', keepprob_placeholder_1)
  tf$summary$scalar('keep_prob_2', keepprob_placeholder_2)
  tf$summary$scalar('keep_prob_3', keepprob_placeholder_3)
})
```

Tensor("placeholder_dropout/keep_prob_3:0", shape=(), dtype=string)

Model Definition

This is the where we define the layers of the model. As the tensors are swung through the graph, it reads like a square dance caller. A number of models were attempted, most with 3 to 7 convolutional layers, then 2 to 3 dense layers. Models with 6 convolutional layers achieve over 95% accuracy. The below model uses 7 convolutional layers and squeezes out the last percentage point of accuracy (from 95% to 96%). This arrangement of layers was first outlined in Python by Thomas Almenningen of Accenture. The layers are:

- Input -> Drop Out (keep 90%) ->
- Convolution -> ReLu ->
- Convolution -> ReLu -> MaxPool -> Drop Out (keep 70%) ->
- Convolution -> ReLu ->
- Convolution -> ReLu -> MaxPool -> Drop Out (keep 70%) ->
- Convolution -> ReLu ->
- Convolution -> ReLu ->
- Convolution -> ReLu -> MaxPool -> Drop Out (keep 70%) -> Reshape ->
- Dense -> ReLu -> Drop Out (keep 50%) ->
- Dense -> ReLu ->

- Parallel(Dense, Dense, Dense, Dense, Dense) -> Softmax

Maxpool & Drop Out

This model, as with others, uses maxpooling to ensure it remains within the GPU card's memory limits, just 4GB. It uses Drop Out at five locations (1x 90% retention, 3x 70% retention and 1x 50% retention) to reduce overtraining. The first dropout is applied immediately to the input image, which is counter intuitive, why obscur any of the input image? This is an approach used by the Google team in the original report and appears to counter over training very well.

```
# Ensure shape of x is batch_qty x 32px x 32px x 1(grayscale)
  x_image <- tf$reshape(x_placeholder, shape(-1L, 32L, 32L, 1L))

# inspect image on tensorboard, ensure as expected
  tf$summary$image('images', x_image, 3)
```

```
Tensor("images:0", shape=(), dtype=string)
```

```
# Apply dropout directly to image input
  x_image <- tf$nn$dropout(x_image, keepprob_placeholder_1, name='Input_DropOut')

## LAYER 1 Conv. kernel=5Wx5Hx1deep, filters=32. Input was 32x32, maxpool OFF, so output = 32x32x32
  layer1 <- conv2d(x_image, 5L, 5L, 1L, 32L, maxpool=FALSE, 'Layer1_Conv')
## LAYER 2 Conv. kernel=5Wx5Hx32deep, filters=32. Input was 32x32, maxpool 2x2, so output = 16x16x32
  layer2 <- conv2d(layer1, 5L, 5L, 32L, 32L, maxpool=TRUE, 'Layer2_Conv')
## Drop OUT!
  layer2 <- tf$nn$dropout(layer2, keepprob_placeholder_2, name='Layer2_DropOut')

## LAYER 3 Conv. kernel=5Wx5Hx64deep, filters=64. Input was 16x16, maxpool OFF, so output = 16x16x64
  layer3 <- conv2d(layer2, 5L, 5L, 32L, 64L, maxpool=FALSE, 'Layer3_Conv')
## LAYER 4 Conv. kernel=5Wx5Hx64deep, filters=64. Input was 16x16, maxpool 2x2, so output = 8x8x64
  layer4 <- conv2d(layer3, 5L, 5L, 64L, 64L, maxpool=TRUE, 'Layer4_Conv')
## Drop OUT!
  layer4 <- tf$nn$dropout(layer4, keepprob_placeholder_2, name='Layer4_DropOut')

## LAYER 5 Conv. kernel = 5Wx5Hx64deep, filters=128. Input was 8x8, maxpool OFF, so output = 8x8x128
  layer5 <- conv2d(layer4, 5L, 5L, 64L, 128L, maxpool=FALSE, 'Layer5_Conv')
## LAYER 6 Conv. kernel = 5Wx5Hx128deep, filters=128. Input was 8x8, maxpool OFF, so output = 8x8x128
  layer6 <- conv2d(layer5, 5L, 5L, 128L, 128L, maxpool=FALSE, 'Layer6_Conv')
## LAYER 7 Conv. kernel = 5Wx5Hx128deep, filters=128. Input was 8x8, maxpool 2x2, so output = 4x4x128
  layer7 <- conv2d(layer6, 5L, 5L, 128L, 128L, maxpool=TRUE, 'Layer7_Conv')
## Drop OUT!
  layer7 <- tf$nn$dropout(layer7, keepprob_placeholder_3, name='Layer7_DropOut')

## Reshape. The dense layer needs a 1d input, so we need to reshape
  layer7 <- tf$reshape(layer7, shape(-1L, 2048L)) #2048 = 4*4*128

## LAYER 9 dense. Prior layer was conv2d with 4x4x128deep = 2048. This layer will use 256 neurons
  layer8 <- dense1d(layer7, 2048L, 256L, 'Layer8_Dense')
  layer8 <- tf$nn$relu(layer8)
## Drop OUT!
  layer8 <- tf$nn$dropout(layer8, keepprob_placeholder_3, name='Layer8_DropOut')

## LAYER 9 dense. Prior layer was 256 neurons. This layer will also use 256 neurons
```

```
  layer9 <- dense1d(layer8, 256L, 256L, 'Layer9_Dense')
  layer9 <- tf$nn$relu(layer9)

## LAYER 10-14 dense, in parallel. Prior layer was 256 neurons, each parallel layer needs 11 outputs
  # These parallel layers allow one model to categorise each digit for up to 5 digits in an image.
  # Being probabilities of 1-11, softmax will be need to be applied, as it is by the NEXT code chunk
  logits_digitA <- dense1d(layer9, 256L, 11L, 'Layer10_DigitA')
  logits_digitB <- dense1d(layer9, 256L, 11L, 'Layer11_DigitB')
  logits_digitC <- dense1d(layer9, 256L, 11L, 'Layer12_DigitC')
  logits_digitD <- dense1d(layer9, 256L, 11L, 'Layer13_DigitD')
  logits_digitE <- dense1d(layer9, 256L, 11L, 'Layer14_DigitE')

## Get predictions (aka y_pred, yhat or logit)
  # create a array of the prediction arrays.
  # NB, each 'logits_digitX' is an array of probabilities for digit 0-9 or 'blank'
  y_hat_probs <- tf$stack(list(logits_digitA,
                               logits_digitB,
                               logits_digitC,
                               logits_digitD,
                               logits_digitE))

## To get the prediction from the array of probabilities we simply find the location of the
 # highest probability, that index is conveniently 0-10
  y_hat <- tf$transpose(tf$argmax(y_hat_probs, dimension=2L))
```

Cost Function

To find the loss for a batch of examples, we find the average loss for each digit (A-E) across that batch and then sum those averages. Using tf.nn.sparse_softmax_cross_entropy_with_logits means we don't have to apply OneHotEncoding to our label values.

```
with(tf$name_scope('loss'),{

  # Calculate the loss for each individual digit in the sequence
  # note, softmax is applied at this stage, by 'sparse_softmax_cross_entropy_with_logits'
  Avg_loss_digitAs <- tf$reduce_mean(tf$nn$sparse_softmax_cross_entropy_with_logits(
                          logits=logits_digitA, labels=y_placeholder[,0L]))
  Avg_loss_digitBs <- tf$reduce_mean(tf$nn$sparse_softmax_cross_entropy_with_logits(
                          logits=logits_digitB, labels=y_placeholder[,1L]))
  Avg_loss_digitCs <- tf$reduce_mean(tf$nn$sparse_softmax_cross_entropy_with_logits(
                          logits=logits_digitC, labels=y_placeholder[,2L]))
  Avg_loss_digitDs <- tf$reduce_mean(tf$nn$sparse_softmax_cross_entropy_with_logits(
                          logits=logits_digitD, labels=y_placeholder[,3L]))
  Avg_loss_digitEs <- tf$reduce_mean(tf$nn$sparse_softmax_cross_entropy_with_logits(
                          logits=logits_digitE, labels=y_placeholder[,4L]))

  # Calculate the total loss for all predictions
  loss <- Avg_loss_digitAs +
          Avg_loss_digitBs +
          Avg_loss_digitCs +
          Avg_loss_digitDs +
          Avg_loss_digitEs
  tf$summary$scalar('loss', loss) # present loss in the summary
})
```

Tensor("loss/loss:0", shape=(), dtype=string)

Optimization Method

This model will use the RMSprop optimiser in its gradient descent. A comparison of the optimisers can be found at: Sabastian Ruder. RMSProp is Geoff Hinton's proposed method, it requires a 'learning rate' which reduces, exponentially, over time. For example, it should at least halve over our 5 epochs (36,000 steps, see above).

```
with(tf$name_scope('optimiser'),{

  # Set up exponential decay of the learning rate
  global_step   <- tf$Variable(0L, trainable=FALSE)  # required to compute decaying learning rate
  learning_rate <- tf$train$exponential_decay(1e-3, global_step, 7500L, 5e-1, staircase=TRUE)
  tf$summary$scalar('learning_rate', learning_rate) # present learning rate in the summary

  # Set up the RMSProp optimiser
  optimiser <- tf$train$RMSPropOptimizer(learning_rate)$minimize(loss, global_step=global_step)
  #
    #tf$train$GradientDescentOptimizer(1e-2)$minimize(loss, global_step=global_step)
})
```

Define How to Evaluate the Predictions

In simple models we would use a line like this to find where the prediction equals the y label (i.e. a correct prediction)

`correct_prediction = tf.equal(tf.argmax(y_hat,1), tf.argmax(y_,1))`

such that:

`accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))`

But, this model is different, there more than one prediction to deal with:

```
with(tf$name_scope('accuracy'),{

  # Predicted qty of digits equals the true qty of digits?
  correct_prediction <- tf$reduce_min(tf$cast(tf$equal(y_hat, y_placeholder), tf$float32), 1L)

  # Calculate the mean
  accuracy <- tf$reduce_mean(correct_prediction) * 100.0

  # Present accuracy in summary
  tf$summary$scalar('accuracy', accuracy)
})
```

Tensor("accuracy/accuracy:0", shape=(), dtype=string)

Batches & Epochs

The limiting factor in my implementation of this model is memory. The GTX1050Ti has only 4Gb available. Therefore, a small batch size has been chosen:

- Training set = 230,400 images
- Batch size = 32 images
- Epoch size = 7,200 batches (230,400/32)
- No. Epochs = 20 (or as many as you can wait for)
- Total Steps = 144,000 (20 x 7,200)

```
batch_size <- 32L
epochs     <- 20L
```

Instantiate the Model

The above code sets out a definition of how the tensors should flow through the model (aka graph). To use this definition we need to open a 'session'. To borrow a term from OOP, the session creates an instance (aka instantiates) the graph and executes it over the batches and epochs which we have defined.

```
# Instantiate the graph in a session
session <- tf$Session()
KeepProbability_1  <- 0.9 #90%
KeepProbability_2  <- 0.7 #70%
KeepProbability_3  <- 0.5 #50%
```

Use Checkpoints

Variables are the weights and biases of the model, finding these variables through training takes a long time, so its worth saving the work done to date. So, let's create a 'saver object' for saving them to. Tensorflow has the concept of 'checkpoints' to assist this process.

The code first checks to see if there is already a checkpoint. First we try to restore the latest checkpoint. This will fail if you have changed the TensorFlow graph or a checkpoint does not exist.

Also, the above code refers to Tensorflow summaries, usueful for reviewing model progress. This chunk will bring those summary items together and log progress.

```
# Checkpoints
saver <- tf$train$Saver()
saver_dir <- 'checkpoints/streetimages_published/'
print('looking for previous checkpoints...')
```

[1] "looking for previous checkpoints…"

```
result <- 'NULL'

result <- tryCatch({
  last_chk_path <- tf$train$latest_checkpoint(checkpoint_dir=saver_dir)
  saver$restore(session, save_path=last_chk_path)
  message(paste0('Restored checkpoint from:', last_chk_path))

  }, error = function(e) {
  session$run(tf$global_variables_initializer())
  message('No checkpoint - initialized variables')
  }
)
```

```
## Restored checkpoint from:checkpoints/streetimages_published/-151200
```

```
# Summaries
log_dir <- 'logs/streetimages_published/'
summary_merged <- tf$summary$merge_all()

# Pass the graph to the writer to display it in TensorBoard
summary_writer_training   <- tf$summary$FileWriter(paste0(log_dir,'/train'), session$graph)
summary_writer_validation <- tf$summary$FileWriter(paste0(log_dir,'/validation'))
```

(For the sake of knitr, this code was executed earlier and a checkpoint created)

Define How to Feed the Model

Tensorflow optimises weights+biases over each spoonful (i.e batch) of images in the training set. Later it will chew on the test set for comparison. Feeding this monster all the training data will consume a lot of memory, as will evaluation on the test set. Therefore we need the following helper function to ration the feeding process.

```
# Map data onto the placeholders, this is the feed dictionary (feed_dict)
feed_dict <- function(stepcount=0L){

  # Calculate the offset using modulus
  offset <- (stepcount * batch_size) %% (nrow(y_train) - batch_size) + 1
  #in python you'd remove the +1, cos indexes are 0based
  to_limit <- min((offset+batch_size-1),nrow(y_train))

  # Get the next batch of data
    # NB, R has a habit of ignoring the depth of a sliced array if it is 1

    # eg Where dim(my_array) = [6500,32,32,1] then dim(my_array[slice,,,])=[slice,32,32].
    # See that? It dropped the 1.

    # Tensorflow expects tensors to have that depth dim, and it is standard for image tensors,
    # so we FORCE it using array()
  xs <- array(x_train[offset:to_limit,,,],c(nrow(x_train[offset:to_limit,,,]),32L,32L,1L))

  # This is not a problem for y tensors because they have no dim which is just 1.
  ys <- y_train[offset:to_limit,]

  return(dict(x_placeholder=xs,
              y_placeholder=ys,
              keepprob_placeholder_1=KeepProbability_1,
              keepprob_placeholder_2=KeepProbability_2,
              keepprob_placeholder_3=KeepProbability_3))
}

# Also, evaluate the model, via test and validation, in batches
evaluate_batch <- function(test, batch_size){

  # cumulative accuracy over ALL batches
  cumulative_accuracy <- 0.0

  # Get the number of images for test or validation
  if(test){
    n_images <- nrow(y_test)
    }else{
    n_images <- nrow(y_validate)
    }

  # Numer of batches needed to evaluate all images
  n_batches <- (n_images %/% batch_size) + 1L

  # for each batch, get the batch images
  for (i in 1:n_batches){

      offset <- (i-1) * batch_size + 1 #in python this would be i*n_batches cos loop start at 0

      if (test){
          # batch of test images
          to_limit <- min((offset+batch_size-1),nrow(y_test))
          xs <- array(x_test[offset:to_limit,,,],c(nrow(x_test[offset:to_limit,,,]),32L,32L,1L))
          ys <- y_test[offset:to_limit,]
      }else{
          # batch of validation images
          to_limit <- min((offset+batch_size-1),nrow(y_validate))
          xs <- array(x_validate[offset:to_limit,,,],c(nrow(x_validate[offset:to_limit,,,]),32,32,1))
          ys <- y_validate[offset:to_limit,]
      }
      cumulative_accuracy <- cumulative_accuracy +
                              session$run(accuracy, dict(x_placeholder=xs,
                                                          y_placeholder=ys,
                                                          keepprob_placeholder_1=1L,
                                                          keepprob_placeholder_2=1L,
                                                          keepprob_placeholder_3=1L
                                                          ))
                              # The 1 is for keepprob.
                              # If dropout was 0.5 then we'd get at most 50% accuracy in test.
  }
  # Return the average accuracy over all batches
  return(cumulative_accuracy / (0.0 + n_batches))
}
```

Define How to Execute the Training

Finally, we get to execute the model. A for loop sends the data, batch by batch to the session. The session then runs its optimisation process, which was defined above, over that batch of data. The key line of code is the session.run statement. It executes the session with a 'session.run' command that takes the following items, which have defined above.

- the summary data desired
- the global step
- the optimsier
- the feeding method - which in turn takes the inputs x and y

… and returns a list of

- the output summary
- the step the session is currently on

Just under this code, we instruct the 'summary_writer_training', defined above, to add the latest figures to its summary. Then, once per 200 batches, we print out the progress. At the end of the process, we report accuracy.

```r
executeGraph <- function(qty_batches, display_step){

  # Let's time the process, so we grab the start time.
  start_time <- proc.time()

  for (stepcount in 1:qty_batches){

      # Run the optimizer using this batch of training data. Returns a list (summary, i, _)
      # note use of list()
      # this is required because tf expects a tuple but this is R
      # so we can't simply pass [a,b,c], must use list()
      # since we use a list, we refer to its contents using [[]]
      summary_stepcount <- session$run(list(summary_merged, global_step, optimiser),
                                       feed_dict=feed_dict(stepcount))

      summary_writer_training$add_summary(summary_stepcount[[1L]], summary_stepcount[[2L]])

      # Print progress, once per display_step iteration and last
      if ((summary_stepcount[[2L]] %% display_step == 0L) || (stepcount == qty_batches - 1L)){

          batch_accuracy <- session$run(accuracy, feed_dict=feed_dict(stepcount))
          message(paste0('Accuracy at step ',summary_stepcount[[2L]],': ', batch_accuracy))
      }
  }
  # After completed, display elapsed time
  run_time <- proc.time() - start_time
  print(paste0('\nTime elapsed: ', run_time))

  # Display accuracy on test set
  test_accuracy <- evaluate_batch(test=TRUE, batch_size=64L)
  print(paste0('Test accuracy: ', test_accuracy))

  # Save all the variables (weights, biases) of the TensorFlow graph
  saver$save(session, saver_dir, global_step)
  print(paste0('Model saved in file: ', saver_dir))
}
```

Before we commit to running the entire training process, which may take many hours, let's try the model on half the data (half an epoch) to see if everything is performing reasonably.

```r
executeGraph(qty_batches=3600L, display_step=600L)
```

Great, it appears to be training with scope to improve if given more time, let's commit to 20 epochs…

```r
executeGraph(qty_batches=144000L, display_step=7200L)
```

Examine Performance on the Test Set

Having trained the model lets feed it with the test set and get the predictions for each test image. To get the y_hats we create a new 'feed_dict' pointing the x and y placeholders to the test set, then run the session (ie model) again.

```r
# Feed the test set to the model in batches (try feeding all at once causes out-of-memory).
testSetSize <- nrow(x_test)
n_batches   <- testSetSize %/% batch_size + ifelse((testSetSize %% batch_size) > 0,1,0)

# for each batch, get the batch y_hat, i.e. predictions, then return the complete set
y_hat_test<-matrix(NA,ncol=5,nrow=testSetSize)
for (i in 1:n_batches){

  offset <- (i-1) * batch_size + 1
  to_limit <- min((offset+batch_size-1),nrow(y_test))
  xs <- array(x_test[offset:to_limit,,,],c(nrow(x_test[offset:to_limit,,,]),32L,32L,1L))
  ys <- y_test[offset:to_limit,]

  # Execute model on the batch, switch off dropout as we are using the model, not training it.
  y_hat_test_batch <- session$run(y_hat, dict(x_placeholder=xs,
                                              y_placeholder=ys,
                                              keepprob_placeholder_1=1L,
                                              keepprob_placeholder_2=1L,
                                              keepprob_placeholder_3=1L))

  y_hat_test[offset:to_limit,]<-y_hat_test_batch
}
y_hat_test <- y_hat_test[1:testSetSize,]
# Convert predictions for individual digits to quantity of digits (1-5digits)
# ...here are the correct quantity of digits for each image...
y_test_qty     <- matrix(as.integer(rowSums(y_test[,]!=10)),ncol=1)
# ...and here are the predictions
y_hat_test_qty <- matrix(as.integer(rowSums(y_hat_test[,]!=10)),ncol=1)
```

There are two types of prediction; 1) the quantity of digits in an image and 2) the identity of each digit therein.

The first examination of results on the test data will be for 1) the quantity of digits, represented by variable 'y_hat_qty'. Here the model achieves 97.6% accuracy, very impressive for such a simple model.

```r
# apply factors so we can use the caret package for analysis
y_hat_test_qty_factor <- factor(y_hat_test_qty)
y_test_qty_factor     <- factor(y_test_qty)

# ensure predictions have all 5 levels (1 digit, 2 digits..5 digits)
levels(y_hat_test_qty_factor) <- c("1", "2", "3", "4", "5")

library(caret)
```

```
## Loading required package: lattice
```

```r
library(e1071)
confusionMatrix(y_hat_test_qty_factor, y_test_qty_factor, positive = NULL,
                dnn = c("Prediction", "Actual"))
```

```
## Confusion Matrix and Statistics
##
##           Actual
## Prediction    1    2    3    4    5
##          1  848   71    5    0    0
##          2   35 5043   75    1    0
##          3    1   77 5990   28    0
##          4    0    4   39  778    5
##          5    0    0    0    0    0
##
## Overall Statistics
##
##                Accuracy : 0.9738
##                  95% CI : (0.9709, 0.9764)
##     No Information Rate : 0.4699
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.9572
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: 1 Class: 2 Class: 3 Class: 4  Class: 5
## Sensitivity           0.95928   0.9707   0.9805  0.96406 0.0000000
## Specificity           0.99373   0.9858   0.9846  0.99606 1.0000000
## Pos Pred Value        0.91775   0.9785   0.9826  0.94189       NaN
## Neg Pred Value        0.99702   0.9806   0.9828  0.99762 0.9996154
## Prevalence            0.06800   0.3996   0.4699  0.06208 0.0003846
## Detection Rate        0.06523   0.3879   0.4608  0.05985 0.0000000
## Detection Prevalence  0.07108   0.3965   0.4689  0.06354 0.0000000
## Balanced Accuracy     0.97650   0.9783   0.9826  0.98006 0.5000000
```

Examples of Incorrect Quantity of Digits

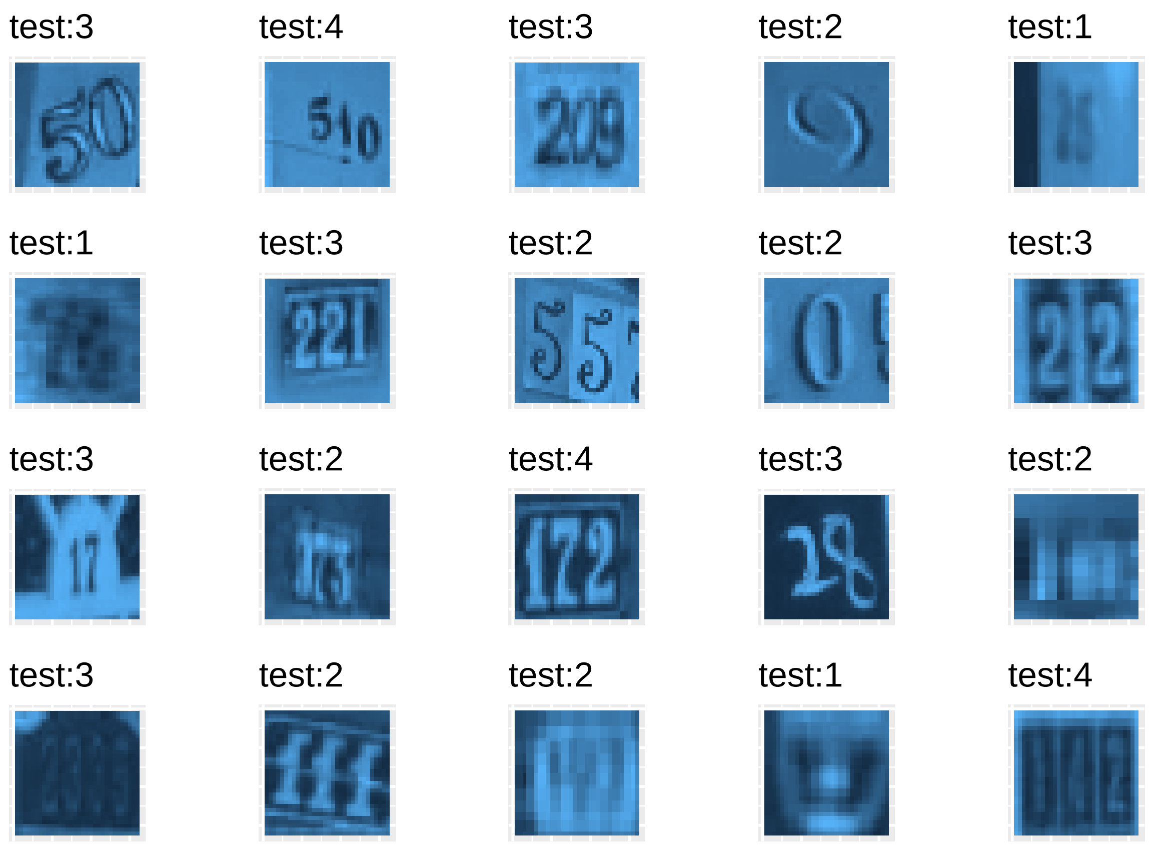
Let's have a look at some of the images where the model field to identify the correct quantity of digits within the image. It is difficult to see a pattern in these failures by inspection alone.

```r
# Select the images where the quantity of digits was incorrectly predicted
PredictionOutcomes_qty <- y_test_qty==y_hat_test_qty
x_test_qty_wrong <- x_test[!PredictionOutcomes_qty,,,]
x_test_qty_wrong <- array(x_test_qty_wrong,c(nrow(x_test_qty_wrong),32,32,1))
y_hat_test_qty_wrong <- matrix(y_hat_test_qty[!PredictionOutcomes_qty],ncol=1)

# Randomly select 20 such images for display.
IDforTest1 <- sample(1:nrow(y_hat_test_qty_wrong), 20, replace=FALSE)

grobs_correct <- lapply(displayImage(x_test_qty_wrong,
                                     y_hat_test_qty_wrong,
                                     'test',
                                     IDforTest1),
                        ggplotGrob)

grid.arrange(grobs=grobs_correct, ncol=5)
```



Performance per Digit

So how well does the model recognise each individual digit? Its overall accuracy is 96.47%, Google achieved 97.84% so this is a very encouraging result given the off-the-rack hardware and obvious limitations of a 32px x 32px training set (constrained by memory limitations) with only a handful of layers. Afterall, this is not Google's 'inception' model.

Results show the least accurately identified digit is 8, which is most commonly confused for a 6. Other common confusions were 5 for 3, and 7 for 1. These errors are common even in humans, so the results are very assuring.

```
y_hat_test_factor <- factor(y_hat_test)
y_test_factor     <- factor(y_test)

confusionMatrix(y_hat_test_factor, y_test_factor, positive = NULL,
                dnn = c("Prediction", "Actual"))
```

```
## Confusion Matrix and Statistics
##
##            Actual
## Prediction     0     1     2     3     4     5     6     7     8     9
##          0   2646    51     4    17    18     6    34     6    31    44
##          1     17  5308    33    25    48     3    15    59    15     9
##          2      3    52  4501    28    39    10     3    41    17    27
##          3      3    13    48  3567     8    87    13    20    45    32
##          4      7    71    18    11  2930     4    17     4     6     7
##          5      6    12     9    68     5  3042    38     2    16    13
##          6     18    15     5     5    11    67  2401     1    76     3
##          7      1    95    19    10     8     1     0  2503     0    10
##          8     18    13    22    53     9    13    53     2  1934    40
##          9     25     8    26    23    13    12     4     6    19  1889
##          10    22    33    23    14    18    20    15    19    11    16
##           Actual
## Prediction    10
##          0    23
##          1    21
##          2    18
##          3    19
##          4    15
##          5    20
##          6     6
##          7    19
##          8    10
##          9    10
##          10 31985
##
## Overall Statistics
##
##                Accuracy : 0.9647
##                  95% CI : (0.9633, 0.9661)
##     No Information Rate : 0.4946
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.9514
##  Mcnemar's Test P-Value : 1.461e-10
##
## Statistics by Class:
##
##                      Class: 0 Class: 1 Class: 2 Class: 3 Class: 4 Class: 5
## Sensitivity           0.95662  0.93599  0.95603  0.93353  0.94303  0.93170
## Specificity           0.99624  0.99587  0.99605  0.99529  0.99741  0.99694
## Pos Pred Value        0.91875  0.95588  0.94978  0.92529  0.94822  0.94150
## Neg Pred Value        0.99807  0.99389  0.99656  0.99585  0.99714  0.99639
## Prevalence            0.04255  0.08725  0.07243  0.05878  0.04780  0.05023
## Detection Rate        0.04071  0.08166  0.06925  0.05488  0.04508  0.04680
## Detection Prevalence  0.04431  0.08543  0.07291  0.05931  0.04754  0.04971
## Balanced Accuracy     0.97643  0.96593  0.97604  0.96441  0.97022  0.96432
##                      Class: 6 Class: 7 Class: 8 Class: 9 Class: 10
## Sensitivity           0.92595  0.93992  0.89124  0.90383    0.9950
## Specificity           0.99668  0.99739  0.99629  0.99768    0.9942
## Pos Pred Value        0.92063  0.93886  0.89248  0.92826    0.9941
## Neg Pred Value        0.99692  0.99743  0.99624  0.99681    0.9951
## Prevalence            0.03989  0.04097  0.03338  0.03215    0.4946
## Detection Rate        0.03694  0.03851  0.02975  0.02906    0.4921
## Detection Prevalence  0.04012  0.04102  0.03334  0.03131    0.4950
## Balanced Accuracy     0.96132  0.96865  0.94377  0.95075    0.9946
```

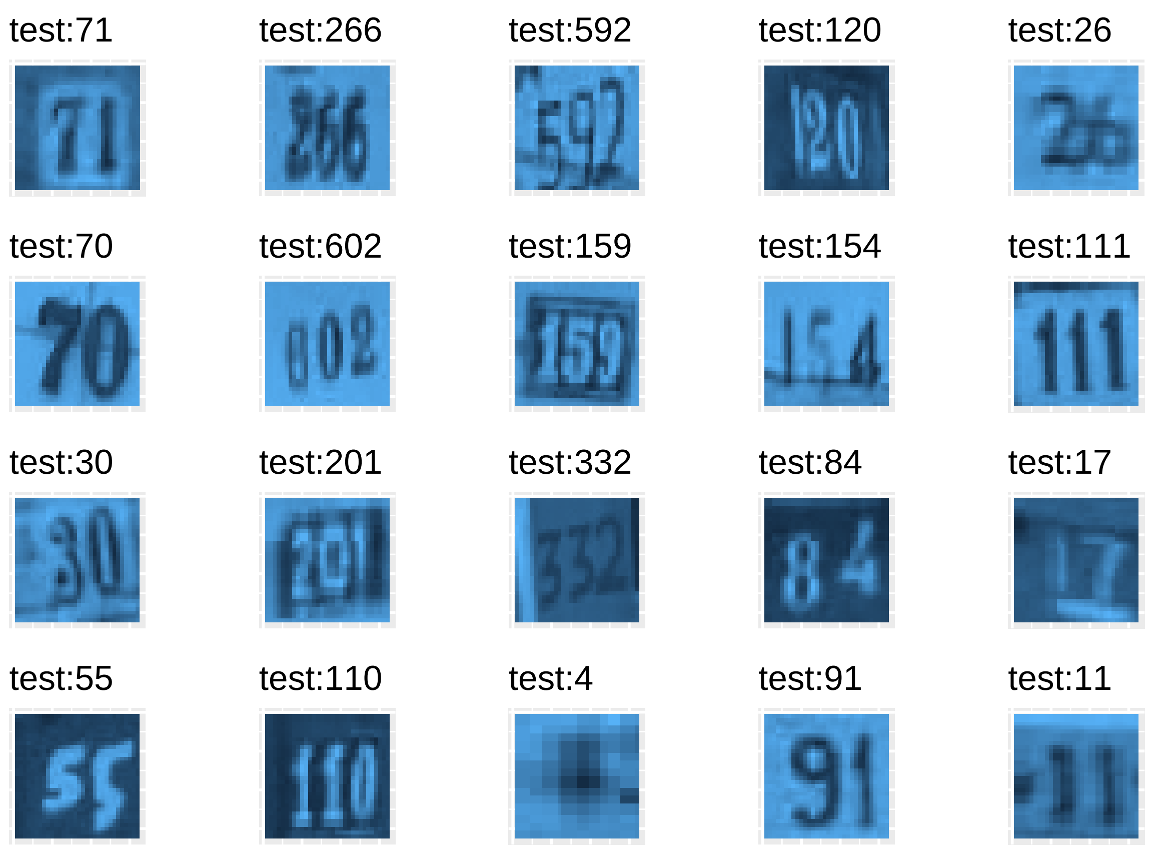
Correctly Classified Examples

```r
# Select the images where all digits were correctly predicted
PredictionOutcomes <- y_test==y_hat_test
x_test_correct <- x_test[rowSums(PredictionOutcomes)==5,,,]
x_test_correct <- array(x_test_correct,c(nrow(x_test_correct),32,32,1))
y_hat_test_correct <- y_hat_test[rowSums(PredictionOutcomes)==5,]

# Randomly select 20 such images for display
IDforTest2 <- sample(1:nrow(y_hat_test_correct), 20, replace=FALSE)

grobs_correct <- lapply(displayImage(x_test_correct,
                                     y_hat_test_correct,
                                     'test',
                                     IDforTest2),
                        ggplotGrob)

grid.arrange(grobs=grobs_correct, ncol=5)
```



Incorrectly Classified Examples

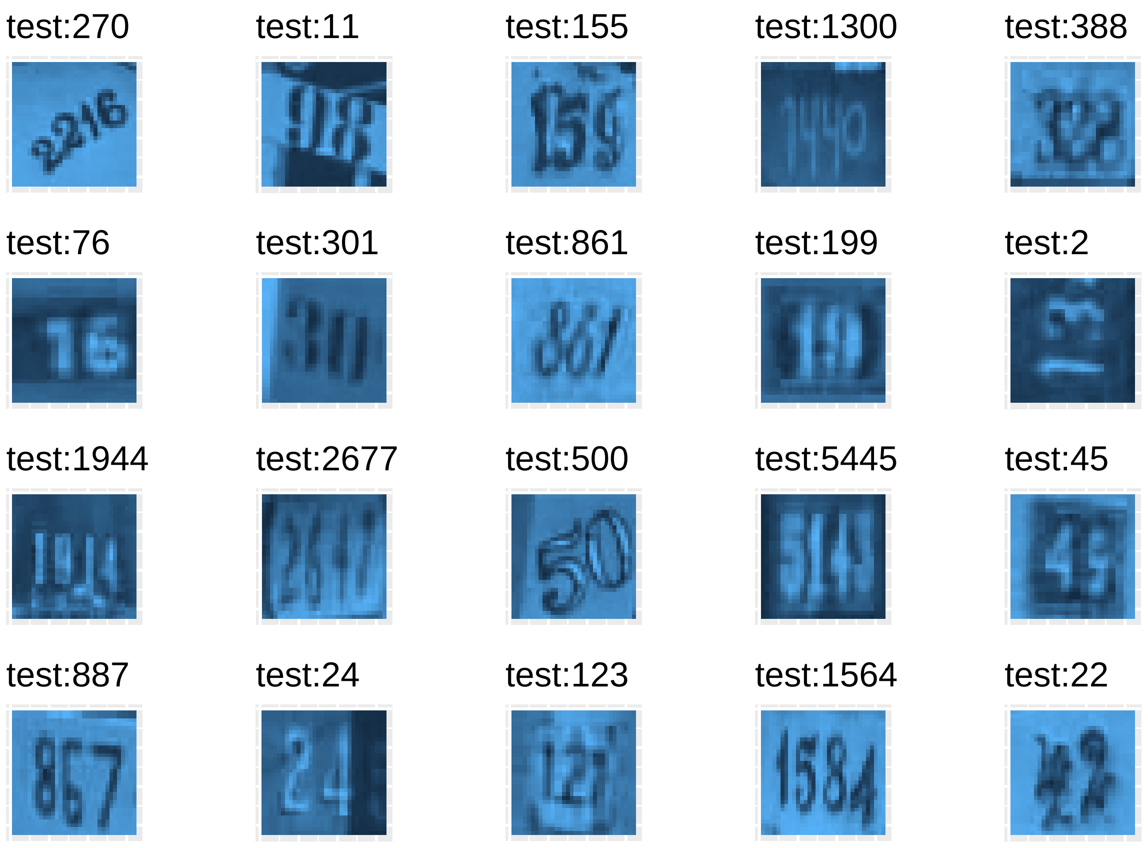
The below examples illustrate how difficult some of the examples are, even a human would struggle with them. Common errors are 7s for 1s, 8s for 3s, as is also common with humans.

```r
# Select the images where all digits were correctly predicted
x_test_wrong <- x_test[rowSums(PredictionOutcomes)!=5,,,]
x_test_wrong <- array(x_test_wrong,c(nrow(x_test_wrong),32,32,1))
y_hat_test_wrong <- y_hat_test[rowSums(PredictionOutcomes)!=5,]

# Randomly select 20 such images for display
IDforTest3 <- sample(1:nrow(y_hat_test_wrong), 20, replace=FALSE)

grobs_wrong <- lapply(displayImage(x_test_wrong,
                                   y_hat_test_wrong,
                                   'test',
                                   IDforTest3),
                      ggplotGrob)

grid.arrange(grobs=grobs_wrong, ncol=5)
```



Examine TensorBoard

For more details on the model, view the tensorboard which results from the log files...

```r
tensorboard(log_dir = "~/CNN_data/logs/streetimages_published/train")
```

```
Started TensorBoard at http://127.0.0.1:3014
```