Apartment hunting in an expensive city is leading me to curses and exclamations. Following are some outstanding examples of insanely priced apartments in Portland, OR, run through Google Deep Dream in hopes of my understanding why people pay so much for a small box. These listings will be gone in no time (I’m sure) so including some captions for posterity.

Let’s start with this one. Indeed, it appears $1899 for 1 bedroom grants access to this clubhouse haunted by some floating apparition:



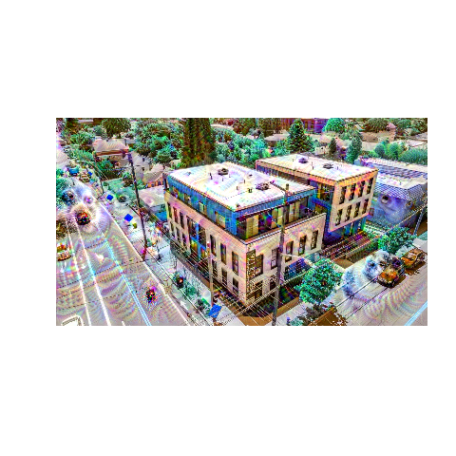
Deep Dream InceptionV3 algorithm here is trained on ImageNet, then makes changes that increase confidence in the predicted category. Looped several times with the num\_octave hyperparameter, it starts to look a good bit trippy and helps give some intuition what a neural network “sees” as prototypical examples of a predicted class. Apparently there is no “view of apartment” class as it keeps seeing ghastly animals. Perhaps it is no coincidence even before running InceptionV3 this clubhouse already looks like it could work in The Shining:



***$1850 / 1br – 697ft2 – BRAND NEW! Enjoy Luxury Uban [Urban?] Living at The Franklin Flats!***

*“NEW ON THE MARKET!*

*“The Franklin Flats is the newest addition to this desirable part of town! Built with the urban adventurer in mind, our small community offers luxury appeal with a neighborhood feel. Boasting a walkability score of 86 out of 100, you can’t beat the location! [unless an 87+?] Our close proximity to Mt. Tabor, food carts, [because you won’t have anything left over for restaurants] shopping and eateries gives you the classic Northwest experience you crave. Email or call to schedule a personal tour today!”*



Apparently the Attack on Titan seals make this a desirable part of town.

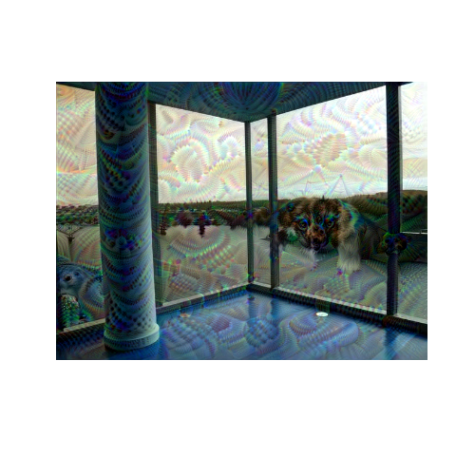
Perhaps those seals are why walkability doesn’t crack 90. If you survive the seals on a midday stroll, there are titan polar bears who can walk through walls:



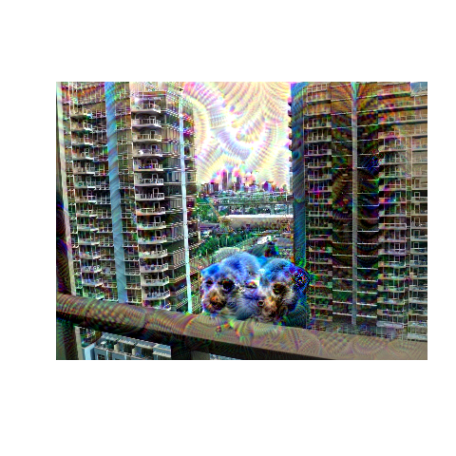
***$4250 / 2br – 1900ft2 – Condo on the Willamette***

*“Breathtaking views of the city and the Willamette River, located in the elegant Atwater. This condo has two bedrooms, living room, dining room, gourmet kitchen, gas fireplace, small office, two balconies, utility room and underground parking. Includes concierge desk, card-accessed security.”*

Something tells me this view of the Willamette River would be complete if a cocker spaniel is staring at me…



But this view is what you really pay for: look at all the suckers in the two identical buildings who massively overpaid for – how the heck did those get up here?!



***$3900 / 2br – 1004ft2 – Portland’s most well-appointed two bedroom apartments now available***

*“Portland’s premier rental community is now pre-leasing. Find everything you desire in one place: The finest dining, distinguished boutiques, and most-beloved haunts. Experience the perfect merger of luxury and livability with our timeless brick exterior, stunning marble lobby, tiled bathrooms, tall ceilings, ample light, extensive storage, concierge services, and even a dog washing station.*

*“Proudly situated in Portland’s [S]Nob Hill/23rd Ave neighborhood, 21 Astor boasts a “97” walk score and a “96” bike score. [Beat that Franklin Flats!] Life is great in the epicenter of Portland’s charm. Pick up your locally-grown produce at City Market. Grab happy hour with the gang at Seratto. Sweat it out at Corepower Yoga.*

*“Our greater than 1:1 parking ratio assures a space for your car in our garage. Imagine never having to find parking on NW 23rd again.”*

I work on NW 21st, that’s almost the cost of parking alone. People walk by on Oregon ‘pot tours’ and this may be how they see the building as well:



Whoa! Is that a pig in the sky? Far out!

At least their new $3.88/sqft/month kitchens aren’t yet haunted – oh, god! Are those giant psychedelic worms!



**Code:**

Started with keras code based on [Google DeepDream](https://en.wikipedia.org/wiki/DeepDream). Added a little looping to try different parameters over these 7 images:

Keras\_Code

library(keras)

library(tensorflow)

base\_image\_path = get\_file('paris.jpg', 'https://i.imgur.com/aGBdQyK.jpg')

result\_prefix = 'sky\_dream'

# These are the names of the layers

# for which we try to maximize activation,

# as well as their weight in the final loss

# we try to maximize.

# You can tweak these setting to obtain new visual effects.

layer\_settings = list(

'mixed4' = 1.0,

'mixed5' = 1.5,

'mixed6' = 2.0,

'mixed7' = 2.5

)

# Playing with these hyperparameters will also allow you to achieve new effects

step = 0.01 # Gradient ascent step size

num\_octave = 3 # Number of scales at which to run gradient ascent

octave\_scale = 1.4 # Size ratio between scales

iterations = 20 # Number of ascent steps per scale

max\_loss = 15.

# This is our base image:

plot(magick::image\_read(base\_image\_path))

# Let's set up some image preprocessing/deprocessing utilities:

preprocess\_image <- function(image\_path) {

# Util function to open, resize and format pictures

# into appropriate arrays.

img = tf$keras$preprocessing$image$load\_img(image\_path)

img = tf$keras$preprocessing$image$img\_to\_array(img)

img = tf$expand\_dims(img, axis=0L)

img = inception\_v3\_preprocess\_input(img)

img

}

deprocess\_image <- function(x) {

x = array\_reshape(x, dim = c(dim(img)[[2]], dim(img)[[3]], 3))

# Undo inception v3 preprocession

x = x / 2.

x = x + 0.5

x = x \* 255.

# Convert to uint8 and clip to the valid range [0, 255]

x = tf$clip\_by\_value(x, 0L, 255L) %>% tf$cast(dtype = 'uint8')

x

}

save\_img <- function(img, fname) {

img <- deprocess\_image(img)

image\_array\_save(img, fname)

}

# Build an InceptionV3 model loaded with pre-trained ImageNet weights

model <- application\_inception\_v3(weights = "imagenet",

include\_top = FALSE)

# Get the symbolic outputs of each "key" layer (we gave them unique names).

outputs\_dict = list()

for (layer\_name in names(layer\_settings)) {

coeff <- layer\_settings[[layer\_name]]

# Retrieves the layer's output

activation <- get\_layer(model, layer\_name)$output

outputs\_dict[[layer\_name]] <- activation

}

# Set up a model that returns the activation values for every target layer

# (as a named list)

feature\_extractor = keras\_model(inputs = model$inputs,

outputs = outputs\_dict)

compute\_loss <- function(input\_image) {

features = feature\_extractor(input\_image)

names(features) = names(layer\_settings)

loss = tf$zeros(shape=list())

for (names in names(layer\_settings)) {

coeff = layer\_settings[[names]]

activation = features[[names]]

# We avoid border artifacts by only involving non-border pixels in the loss.

scaling = tf$reduce\_prod(tf$cast(tf$shape(activation), 'float32'))

loss = loss + coeff \* tf$reduce\_sum(tf$square(activation)) / scaling

}

loss

}

# Set up the gradient ascent loop for one octave

gradient\_ascent\_step <- function(img, learning\_rate) {

with(tf$GradientTape() %as% tape, {

tape$watch(img)

loss = compute\_loss(img)

})

# Compute gradients.

grads = tape$gradient(loss, img)

# Normalize gradients.

grads = grads / tf$maximum(tf$reduce\_mean(tf$abs(grads)), 1e-6)

img = img + learning\_rate \* grads

list(loss, img)

}

gradient\_ascent\_loop <- function(img, iterations, learning\_rate, max\_loss = NULL) {

for (i in 1:iterations) {

c(loss, img) %<-% gradient\_ascent\_step(img, learning\_rate)

if (!is.null(max\_loss) && as.array(loss) > max\_loss)

break

cat("...Loss value at step", i, ":", as.array(loss), "\n")

}

img

}

# Run the training loop, iterating over different octaves

original\_img = preprocess\_image(base\_image\_path)

# Prepares a list of shape tuples defining the different scales at which to run gradient ascent

original\_shape <- dim(original\_img)[2:3]

successive\_shapes <- list(original\_shape)

for (i in 1:num\_octave) {

shape <- as.integer(original\_shape / (octave\_scale ^ i))

successive\_shapes[[length(successive\_shapes) + 1]] <- shape

}

# Reverses the list of shapes so they're in increasing order

successive\_shapes <- rev(successive\_shapes[1:3])

# Resizes the array of the image to the smallest scale

shrunk\_original\_img <- tf$image$resize(original\_img, successive\_shapes[[1]])

img = tf$identity(original\_img) # Make a copy

for (i in 1:length(successive\_shapes)) {

shape = successive\_shapes[[i]]

cat("Processing octave", i, "with shape", shape, "\n")

# Scales up the dream image

img <- tf$image$resize(img, shape)

# Runs gradient ascent, altering the dream

img <- gradient\_ascent\_loop(img,

iterations = iterations,

learning\_rate = step,

max\_loss = max\_loss)

# Scales up the smaller version of the original image: it will be pixellated

upscaled\_shrunk\_original\_img <-

tf$image$resize(shrunk\_original\_img, shape)

# Computes the high-quality version of the original image at this size

same\_size\_original <-

tf$image$resize(original\_img, shape)

# The difference between the two is the detail that was lost when scaling up

lost\_detail <-

same\_size\_original - upscaled\_shrunk\_original\_img

# Reinjects lost detail into the dream

img <- img + lost\_detail

shrunk\_original\_img <-

tf$image$resize(original\_img, shape)

tf$keras$preprocessing$image$save\_img(paste(result\_prefix,'.png',sep = ''), deprocess\_image(img$numpy()))

}

# Plot result

plot(magick::image\_read(paste(result\_prefix,'.png',sep = '')))

library(keras)

library(tensorflow)

library(purrr)

# Function Definitions ----------------------------------------------------

preprocess\_image <- function(image\_path){

image\_load(image\_path) %>%

image\_to\_array() %>%

array\_reshape(dim = c(1, dim(.))) %>%

inception\_v3\_preprocess\_input()

}

deprocess\_image <- function(x){

x <- x[1,,,]

# Remove zero-center by mean pixel

x <- x/2.

x <- x + 0.5

x <- x \* 255

# 'BGR'->'RGB'

x <- x[,,c(3,2,1)]

# Clip to interval 0, 255

x[x > 255] <- 255

x[x < 0] <- 0

x[] <- as.integer(x)/255

x

}

# Parameters --------------------------------------------------------

## list of images to process --

list\_images <- list.files('images/deep dream apartments/orig/', full.names = TRUE)

## list of settings to try --

list\_settings <- list(

settings = list(

features = list(

mixed2 = 0.2,

mixed3 = 0.5,

mixed4 = 2.,

mixed5 = 1.5),

hyperparams = list(

# Playing with these hyperparameters will also allow you to achieve new effects

step = 0.01, # Gradient ascent step size

num\_octave = 6, # Number of scales at which to run gradient ascent

octave\_scale = 1.4, # Size ratio between scales

iterations = 20, # Number of ascent steps per scale

max\_loss = 10

)

),

settings = list(

features = list(

mixed2 = 0.5,

mixed3 = 0.2,

mixed4 = 1.1,

mixed5 = 1.5),

hyperparams = list(

step = 0.01,

num\_octave = 9,

octave\_scale = 1.1,

iterations = 20,

max\_loss = 5

)

),

settings = list(

features = list(

mixed2 = 0.02,

mixed3 = 0.05,

mixed4 = 0.01,

mixed5 = 0.05),

hyperparams = list(

step = 0.01,

num\_octave = 11,

octave\_scale = 1.1,

iterations = 20,

max\_loss = 20

)

),

settings = list(

features = list(

mixed2 = 0.2,

mixed3 = 0.5,

mixed4 = 2.,

mixed5 = 1.5),

hyperparams = list(

step = 0.01,

num\_octave = 8,

octave\_scale = 1.4,

iterations = 20,

max\_loss = 10

)

),

settings = list(

features = list(

mixed2 = 0.2,

mixed3 = 0.1,

mixed4 = 0.4,

mixed5 = 0.3),

hyperparams = list(

step = 0.01,

num\_octave = 8,

octave\_scale = 1.4,

iterations = 20,

max\_loss = 10

)

),

settings = list(

features = list(

mixed2 = 1.2,

mixed3 = 1.5,

mixed4 = 3.,

mixed5 = 2.5),

hyperparams = list(

step = 0.01,

num\_octave = 8,

octave\_scale = 1.4,

iterations = 20,

max\_loss = 25

)

),

settings = list(

features = list(

mixed2 = 0.2,

mixed3 = 2.5,

mixed4 = 2.,

mixed5 = 3.5),

hyperparams = list(

step = 0.05,

num\_octave = 6,

octave\_scale = 1.4,

iterations = 20,

max\_loss = 13

)

),

settings = list(

features = list(

mixed2 = 0.2,

mixed3 = 2.5,

mixed4 = 2.,

mixed5 = 3.5),

hyperparams = list(

step = 0.05,

num\_octave = 8,

octave\_scale = 1.4,

iterations = 20,

max\_loss = 13

)

)

)

# Model Definition --------------------------------------------------------

k\_set\_learning\_phase(0)

# Build the InceptionV3 network with our placeholder.

# The model will be loaded with pre-trained ImageNet weights.

model <- application\_inception\_v3(weights = "imagenet", include\_top = FALSE)

# This will contain our generated image

dream <- model$input

# Get the symbolic outputs of each "key" layer (we gave them unique names).

layer\_dict <- model$layers

names(layer\_dict) <- map\_chr(layer\_dict ,~.x$name)

## just loop on images - bottleneck is training anyway --

for(image\_path in list\_images){

image <- preprocess\_image(image\_path)

#res <- predict(model,image)

#reduce(dim(res)[-1], .f = `\*`) %>%

# as.numeric() %>%

#array\_reshape(res, c(2, 51200)) %>%

# imagenet\_decode\_predictions()

## get model prediction:

#x <- image\_to\_array(image)

#x <- array\_reshape(image, c(1, dim(image)))

#x <- imagenet\_preprocess\_input(image)

#preds = predict(model, x)

#imagenet\_decode\_predictions(preds, top = 3)

#predict\_classes(model, image)

# need to try again later

setting\_counter <- 0

## then nested loop on settings --

for(settings in list\_settings){

setting\_counter <- setting\_counter + 1

print(paste0('starting ', image\_path, ', setting # ', setting\_counter))

# reload image each time

image <- preprocess\_image(image\_path)

# Define the loss

loss <- k\_variable(0.0)

for(layer\_name in names(settings$features)){

# Add the L2 norm of the features of a layer to the loss

coeff <- settings$features[[layer\_name]]

x <- layer\_dict[[layer\_name]]$output

scaling <- k\_prod(k\_cast(k\_shape(x), 'float32'))

# Avoid border artifacts by only involving non-border pixels in the loss

loss <- loss + coeff\*k\_sum(k\_square(x)) / scaling

}

# Compute the gradients of the dream wrt the loss

grads <- k\_gradients(loss, dream)[[1]]

# Normalize gradients.

grads <- grads / k\_maximum(k\_mean(k\_abs(grads)), k\_epsilon())

# Set up function to retrieve the value

# of the loss and gradients given an input image.

fetch\_loss\_and\_grads <- k\_function(list(dream), list(loss,grads))

# this function will crash the R session if too many octaves on too small an image

eval\_loss\_and\_grads <- function(image){

outs <- fetch\_loss\_and\_grads(list(image))

list(

loss\_value = outs[[1]],

grad\_values = outs[[2]]

)

}

gradient\_ascent <- function(x, iterations, step, max\_loss = NULL) {

for (i in 1:iterations) {

out <- tryCatch(eval\_loss\_and\_grads(x), error = function(e) NA) # need to add this for negative gradients

if(is.na(out$loss\_value)){

print(paste0('NA loss\_value on setting # ', setting\_counter))

break

} else if (!is.null(max\_loss) & out$loss\_value > max\_loss) {

break

}

print(paste("Loss value at", i, ':', out$loss\_value))

x <- x + step \* out$grad\_values

}

x

}

original\_shape <- dim(image)[-c(1, 4)]

successive\_shapes <- list(original\_shape)

for (i in 1:settings$hyperparams$num\_octave) {

successive\_shapes[[i+1]] <- as.integer(original\_shape/settings$hyperparams$octave\_scale^i)

}

successive\_shapes <- rev(successive\_shapes)

original\_image <- image

shrunk\_original\_img <- image\_array\_resize(

image, successive\_shapes[[1]][1], successive\_shapes[[1]][2]

)

shpnum <- 0 # for debugging

for (shp in successive\_shapes) {

shpnum <- shpnum + 1 # for debugging

image <- image\_array\_resize(image, shp[1], shp[2])

print(paste0('running octave ', shpnum))# for debugging

image <- gradient\_ascent(image, settings$hyperparams$iterations, settings$hyperparams$`step`,

settings$hyperparams$max\_loss)

print(paste0('finished octave ', shpnum))# for debugging

upscaled\_shrunk\_original\_img <- image\_array\_resize(shrunk\_original\_img, shp[1], shp[2])

same\_size\_original <- image\_array\_resize(original\_image, shp[1], shp[2])

lost\_detail <- same\_size\_original - upscaled\_shrunk\_original\_img

image <- image + lost\_detail

shrunk\_original\_img <- image\_array\_resize(original\_image, shp[1], shp[2])

}

image\_path %>%

gsub('/orig/', '/dream/', .) %>%

gsub('.jpg', paste0('\_dream', setting\_counter, '.png'), .) %>%

png(filename = .)

plot(as.raster(deprocess\_image(image)))

dev.off()

print(paste0('finished ', image\_path, ', setting # ', setting\_counter))

}

}