Classifying Campaign Images with a Convolutional Neural Network: Initial Report

Ryan Heslin

March 1, 2022

```
library(torch)
library(torchvision)
library(luz)

source(here::here("R", "utilities.R"))
source(here::here("R", "classifier_utilities.R"))
source(here::here("R", "candidate_image_dataset.R"))
```

Much of this is copied from here At present, this script uses the pretrained RESNET18 classifier to classify images as either containing or not containing Donald Trump. It implements this network with the `torch` package, an R interface to the `torch` machine learning library also employed by the popular PyTorch framework. I chose `torch` because it does not introduce a Python dependency and is easier to debug than `keras`, an alternative, since it uses less abstraction. This script takes about 10 minutes to run on my machine.

```
set.seed(1)
data_dir <- here::here("data/classifier/trump_image")
```

Preliminaries

First, training, testing, and validation datasets need to be created. This is trivial with the provided `image_folder_dataset` class, but the off-the-shelf solution does not account for the class imbalance. So I subclassed it myself to implement weighted sampling and randomized transformations on sampling, standard methods of generating extra training data. When this dataset is indexed, an image is selected by a weighted random sample, then randomly transformed according to the transformation parameters. This means a single observation can effectively generate multiple training examples, and the weights ensure that the relatively few Trump images available are frequently selected in training. Much of the code in the relevant file is copied from here.

```
sample_weights <- c(
  no_trump = 1,
  trump = 20
```

```
)
transform_params <- list(angle = 20, flip = .7, rescale = .9)
train_ds <- candidate_image_dataset(
  img_dir = file.path(data_dir, "train"), sample_weights = sample_weights, transform_params = transform
)
valid_ds <- candidate_image_dataset(
  img_dir = file.path(data_dir, "valid"),
  sample_weights = sample_weights, transform_params = transform_params
)
test_ds <- candidate_image_dataset(
  img_dir = file.path(data_dir, "test"),
  sample_weights = sample_weights, transform_params = transform_params
)
```

Right now I use the standard **dataloader** class, which creates an object capable of retrieving observations in batches. A batch is the number of observations processed before weights are updated. Several are processed each epoch (a complete cycle through each training observation).

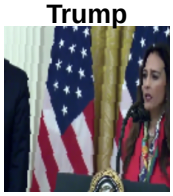
```
batch_size <- 32

train_dl <- dataloader(train_ds, batch_size = batch_size, shuffle = TRUE)
valid_dl <- dataloader(valid_ds, batch_size = batch_size)
test_dl <- dataloader(test_ds, batch_size = batch_size)
```

These are the images contained in one batch. Note the rotations, flips, and resizings.

```
batch <- train_dl$.iter()$.next()
inspect_image_batch(batch)
```

Trump



No Trump

Trump

No Trump

Trump

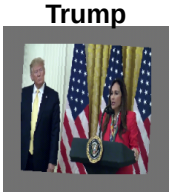
No Trump

No Trump

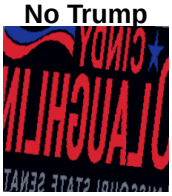
Trump

No Trump

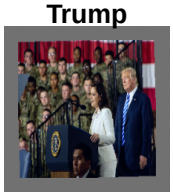
Trump



No Trump

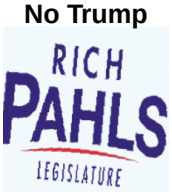


Trump



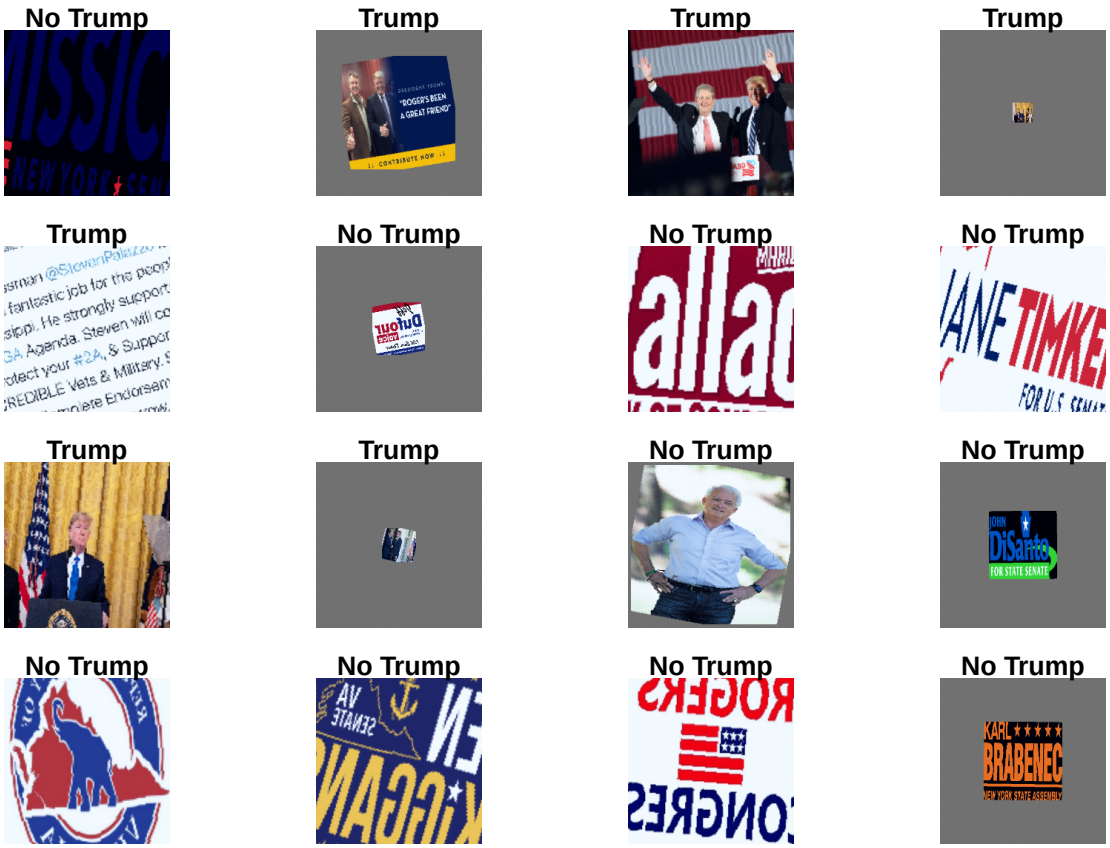
No Trump

No Trump



No Trump

No Trump



RESNET18 has a complicated architecture. A possible next step is implementing a simpler network from scratch. There are millions of parameters: training them all would be expensive

```
model <- model_resnet18(pretrained = TRUE)
model
```

```
An ‘nn_module‘ containing 11,689,512 parameters.

-- Modules ---------------------------------------
* conv1: <nn_conv2d> #9,408 parameters
* bn1: <nn_batch_norm2d> #128 parameters
* relu: <nn_relu> #0 parameters
* maxpool: <nn_max_pool2d> #0 parameters
* layer1: <nn_sequential> #147,968 parameters
* layer2: <nn_sequential> #525,568 parameters
* layer3: <nn_sequential> #2,099,712 parameters
* layer4: <nn_sequential> #8,393,728 parameters
* avgpool: <nn_adaptive_avg_pool2d> #0 parameters
* fc: <nn_linear> #513,000 parameters
```

This code modifies all model parameters to be ignored in gradient computations, freezing the pretrained

weights. This saves a great deal of computing power. The one essential change is reshaping the output layer to yield two values, for the two classes.

```
model$parameters %>% purrr::walk(function(param) param$requires_grad_(FALSE))
model$fc <- nn_linear(in_features = model$fc$in_features, out_features = length(CLASS_NAMES))

model <- model$to(device = DEVICE)
```

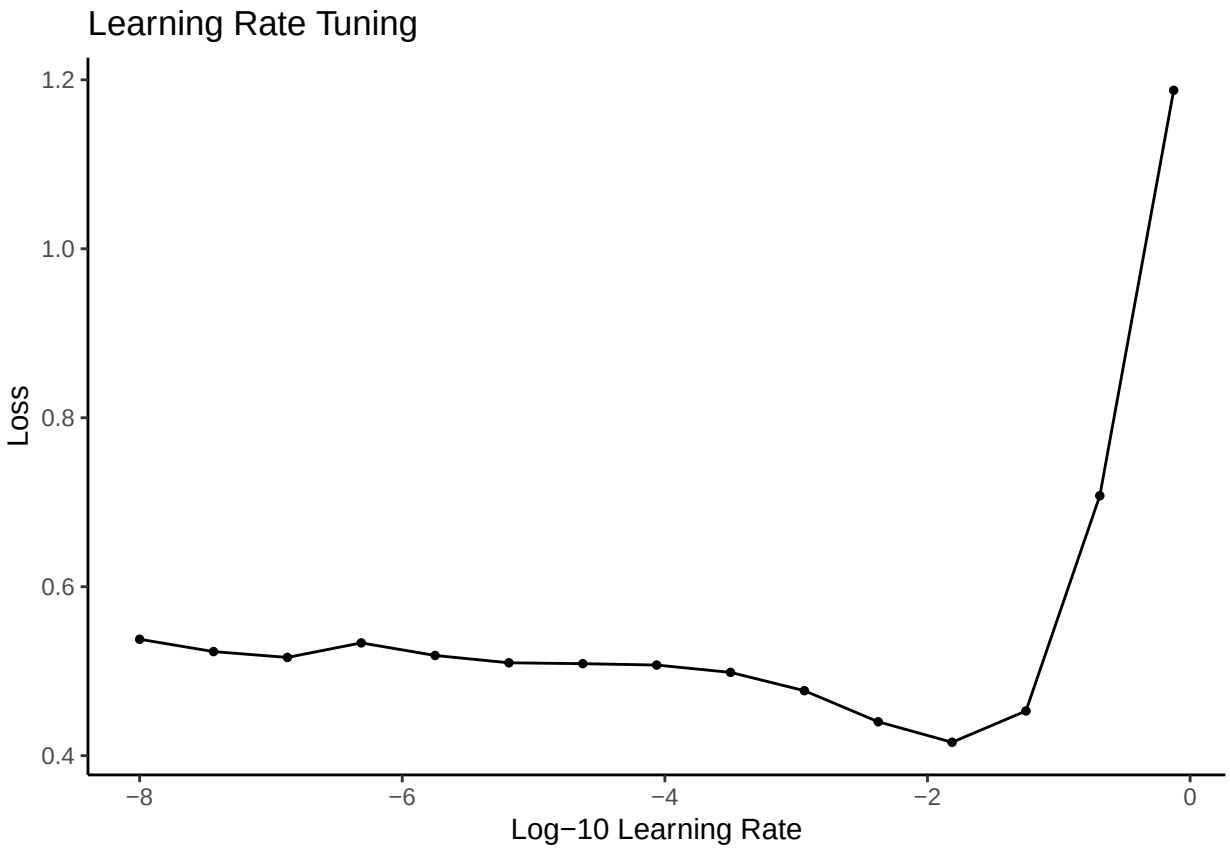
I set a standard loss function and apply some weights to account for the class imbalance. This will increase the computed loss for the Trump class.

```
loss_weights <- torch_tensor(1 / prop.table(table(torch::as_array(train_ds$targets)))) * c(.5, 1)
criterion <- nn_cross_entropy_loss(weight = loss_weights) # add weight rescaling tensor here (inverse p
optimizer <- optim_sgd(model$parameters, lr = 0.1, momentum = 0.9)
```

Learning rate controls how fast the model updates weights. I use a function copied from the source cited above. The optimal rate seems to vary between model runs, perhaps due to the class imbalance. Each of the 10 epochs should take 491 batches to process all 491 examples.

```
lrs <- find_lr(train_dl, optimizer) %>%
  suppressMessages()

ggplot2::ggplot(lrs, aes(log_lrs, losses)) +
  ggplot2::geom_point(size = 1) +
  ggplot2::geom_line() +
  ggplot2::theme_classic() +
  ggplot2::labs(x = "Log-10 Learning Rate", y = "Loss", title = "Learning Rate Tuning")
```




```
# Find loss-minimizing learning rate ()
lr <- with(lrs, 10^log_lrs[which.min(losses)])
num_epochs <- 10
scheduler <- optimizer %>%
  lr_one_cycle(
    max_lr = .1, epochs = num_epochs, steps_per_epoch = train_dl$.length()
  )
```

Training and Assessment

This code trains the model and applies it to the test set. It naturally takes the longest to run. Since the `for` loops for each data partition are nearly identical, I wrote a function to substitute appropriate values into a generic expression. That would have been much harder in Python. As I write this, training loss is much higher than validation loss for most epochs, which is unusual for a model (like this one) without dropout layers. My best guess would be confusion between images of people and of text, or perhaps instability due to the class imbalance.

```
train_loop <- process_batches(train_dl, loss <- train_batch(b), train_losses <- c(train_losses, loss))
valid_loop <- process_batches(valid_dl, loss <- valid_batch(b), valid_losses <- c(valid_losses, loss))
test_loop <- process_batches(
  test_dl, out <- test_batch(b),
  results <- rbind(results, out[["results"]]),
```

```
  total <- total + out[["total"]],
  correct <- correct + out[["correct"]],
  test_losses <- c(test_losses, out[["loss"]])
)
test_losses <- double()
for (epoch in seq(num_epochs)) {
  train_losses <- valid_losses <- double()
  model$train()

  eval(train_loop)

  model$eval()

  eval(valid_loop)

  cat(
    sprintf(
      "\nLoss at epoch %d: training: %3f, validation: %3f\n",
      epoch, mean(train_losses), mean(valid_losses)
    )
  )
}
```

```
Loss at epoch 1: training: 24.580418, validation: 33.328242

Loss at epoch 2: training: 34.521406, validation: 35.562344

Loss at epoch 3: training: 21.254458, validation: 21.081666

Loss at epoch 4: training: 15.601388, validation: 16.194335

Loss at epoch 5: training: 12.552497, validation: 12.497918

Loss at epoch 6: training: 8.596450, validation: 6.232103

Loss at epoch 7: training: 4.882191, validation: 2.832907

Loss at epoch 8: training: 2.057236, validation: 0.350613

Loss at epoch 9: training: 1.077795, validation: 0.329729

Loss at epoch 10: training: 1.339115, validation: 0.193994
```

```
total <- correct <- 0
results <- data.frame(actual = integer(), log_probs = double())
# saveRDS(model, here::here("data/classifier/cnn_trained.Rds"))
```

```
eval(test_loop)
```

Accuracy overall is quite good, but again the class imbalance makes that a misleading measure (a "model" that just predicted "not Trump" every time would do well).

```
test_losses
```

```
[1] 0.17354432 0.04513443 0.46865103 0.04289013
```

```
correct / total
```

```
[1] 0.8846154
```

The confusion matrix.

```
table(factor(CLASS_NAMES[results$actual], levels = CLASS_NAMES),
  factor(CLASS_NAMES[results$pred_class], levels = CLASS_NAMES),
  dnn = c("truth", "predicted")
)
```

```
          predicted
truth      no_trump trump
  no_trump       38    12
  trump           0    54
```

I establish a metric to track area under the ROC curve. This is roughly the proportion of accurate classifications, usually defined as all fitted probabilities as prediction thresholds. I specify some thresholds myself. I just revised an incorrect means of computing this quantity, so I'm not completely confident in it

```
roc_spec <- luz_metric_binary_auroc(thresholds = seq(.1, .9, .1))
test_roc_auc <- roc_spec$new()
test_roc_auc$update(torch_tensor(results$prob), torch_tensor(results$actual)$subtract(1))
```

```
torch_tensor
 50
[ CPUFloatType{1} ]
```

```
test_roc_auc$compute()
```

```
[1] 0.78
```

Unused code for fitting the model with `luz`, a higher-level `torch` interface.

```
# fitted <- net %>%
# setup(
# loss = criterion,
# optimizer = optimizer_adam,
# metrics = list(
# luz_metric_binary_accuracy,
# luz_metric_binary_auroc
# )
# ) %>%
# set_hprams(num_class = 10) %>%
# set_opt_hparams(lr = lr) %>%
# fit(train_dl, epochs = 10, valid_data = valid_dl)
```

At present, I see the following areas to improve:

1. Separating images of text logos from images of candidates. RESNET18, and CNNs generally, are not suited for text recognition, so these training examples are not useful to the model. Separating the two types of images by hand would be tedious but doable, but there may be a programmatic way of doing so
2. Customizing the neural network I currently am using RESNET18 almost unmodified. I could try including more weights in gradient computations, or designing my own network with a different (probably simpler) architecture
3. Hyperparameter customization. From my limited knowledge, tuning learning rate, batch size, and other hyperparameters is largely guesswork, but necessary to produce useful models.
4. Experimenting with preprocessing. Cropping, flipping, etc., could be used to generate more training examples. ‘torchvision‘ has functions for many different transformations.