Problem Set 3: Deep Learning & Unsupervised Learning

author 1 (matriculation number) author 2 (matriculation number)
author 3 (matriculation number)

2021-12-06 till 2022-01-16

- You may answer this problem set in groups of up to three students. Please state your names and matriculation numbers in the `YAML` header before submission (under the `author` field).
- Please hand in your answers via GitHub until 2022-01-16, 23:59 pm. When you have pushed your final results to GitHub, create a GitHub issue with the title "Hand-in" and mark Marius (GitHub username: *puchalla*) and Simon (GitHub username: *simonschoe*) as assignees or reference us in the issue description. Please ensure that all code chunks can be executed without error before pushing your last commit!
- You are supposed to Git commit your code changes regularly! In particular, each group member is required to contribute commits to the group's final submission to indicate equal participation in the group assignment. In case we observe a highly imbalanced distribution of contributions (indicated by the commits), individual seminar participants may be penalized.
- Make sure to answer **all** questions in this notebook! You may use plain text, R code and LaTeX equations to do so. You must always provide at least one sentence per answer and include the code chunks you used in order to reach a solution. Please comment your code in a transparent manner.
- For several exercises throughout the assignment you may find hints and code snippets that should support you in solving the exercises. You may rely on these to answer the questions, but you don't have to. Any answer that solves the exercise is acceptable.
- **Important:** In this assignment you will have to work with dedicated deep learning software packages (in particular, `keras` and `tensorflow` for `R`). These packages require a Python installation on your computer as the deep learning code will be send from `R` to `Python` in the background. Commonly, setting up Python (incl. `keras` and `tensorflow` for Python) and making it work in unison with `R` can be quite a struggle. Therefore, we urge you to follow the official installation guideline that accompanies the ISLR textbook.

Before starting, leverage the `pacman` package using the code chunk below. This package is a convenient helper for installing and loading packages at the start of your project. It checks whether a package has already been installed and loads the package if available. Otherwise it installs the package autonomously. Use `pacman::p_load()` to install and load the following packages:

- `tidyverse` (meta-package to load `dplyr`, `ggplot2` and co.)
- `ISLR` (contains the data set for *Task 2*)
- `rsample` (functions for data partitioning and resampling)
- `broom` (functions for converting `parsnip` objects into tidy tibbles)
- `yarstick` (performance metrics for classification and regression)
- `patchwork` (syntax for combining separate ggplots into the same graphic)
- `tensorflow` (low-level deep learning API written in C++, CUDA, Python)
- `keras` (high-level deep learning API that sits on top of `tensorflow` and enables more user-friendly implementations)
- `dendextend` (enhanced capabilities for hierarchical clustering and dendrogram plotting)

```
# check if pacman is installed (install if evaluates to FALSE)
if (!require("pacman")) install.packages("pacman")
# load (or install if pacman cannot find an existing installation) the relevant packages
pacman::p_load(
  tidyverse, ISLR, rsample, broom, yardstick,
  tensorflow, keras, dendextend
)
```

In case you need any further help with the above mentioned packages and included functions, use the `help()` function build into RStudio (e.g., by running `help(rsample)`).

Task 1: Feed-Forward Neural Networks I

Task 1.1

The proclaimed benefits of neural network-based machine learning (aka *deep learning*) are manifold. In the literature, neural networks are characterized, amongst others, as **simple**, **scalable**, **versatile**, and **reusable**. Briefly explain these features in your own words by reflecting on the following questions:

1. What makes neural networks *simple* relative to traditional machine learning methods? Which problem does deep learning solve for you that machine learning models could not?
2. How can neural networks be *scaled*? What does it mean to *scale* a neural network?
3. Why would you consider neural networks as *versatile* tools?
4. How are neural networks *reusable*? How does the mechanism of *reusing* a neural network work?

Hint: To answer these questions, you will have to perform your own research and discuss within your group. Presumably, you won't find direct answers to these questions in the lecture slides/textbook. Also, you might find it helpful to revisit these questions after you have finished the programming tasks which follow.

Task 1.2

Even though they are very powerful, neural networks are not the best tool for every task (cf. Chapter 10.6 in the ISLR textbook). Consider the three following tasks and argue whether you would use neural nets to address them:

1. You want to identify the influence of a football club's budget on its success in the German Bundesliga, measured by how many points it won in the last season.
2. Given the text of 20,000 book reviews, your goal is to identify whether they are positive or negative.
3. You are supposed to build a model that explains the credit score of customers, using a vast data set with multiple potential variables.

Task 1.3

Now we try to predict housing prices in Boston with deep learning. The Boston housing data set may seem familiar to you as it is used for several exercises in the ISLR textbook. It contains the housing prices in the mid 1970s. It consists of data points about the suburb at the time, such as the crime rate or the local property tax rate. Additional info can be found in the dataset documentation (`?dataset_boston_housing`). Let's inspect the dataset structure:

```
houses <- keras::dataset_boston_housing()
```

```
> Loaded Tensorflow version 2.7.0
```

```
str(houses)
```

```
> List of 2
>  $ train:List of 2
>   ..$ x: num [1:404, 1:13] 1.2325 0.0218 4.8982 0.0396 3.6931 ...
>   ..$ y: num [1:404(1d)] 15.2 42.3 50 21.1 17.7 18.5 11.3 15.6 15.6 14.4 ...
>  $ test :List of 2
>   ..$ x: num [1:102, 1:13] 18.0846 0.1233 0.055 1.2735 0.0715 ...
>   ..$ y: num [1:102(1d)] 7.2 18.8 19 27 22.2 24.5 31.2 22.9 20.5 23.2 ...
```

`houses` is a list object which contains our training and testing data. Next, we assign each element of `houses` to an individual variable using `keras` multi-assignment operator `%<-%`.

```
c(c(x_train, y_train), c(x_test, y_test)) %<-% houses
```

Looking at the structure of `houses` above, how many observations are present in each dataset? What is the ratio that was used to perform the validation split and divide the data into training and test set? Which values are stored in `y_train` respectively `y_test`?

Task 1.4

Describe what the below-written lines of code do and briefly explain the intention behind them. Unfortunately, our ISLR textbook is largely silent about the importance of the presented step, hence we urge you to consult your preferred search engine to answer this question (potential starting points: Source 1, ch. 4.3, Source 2, Source 3).

```
mean_vec <-
  purrr::map_dbl(.x = as_tibble(x_train), .f = mean)

sd_vec <-
  purrr::map_dbl(.x = as_tibble(x_train), .f = sd)

x_train  <- scale(x_train, center = mean_vec, scale = sd_vec)
x_test   <- scale(x_test, center = mean_vec, scale = sd_vec)
```

Task 1.5

Currently, several deep learning frameworks[1] exist that allow for easy training and prototyping of neural network models. Three of the most prominent deep learning frameworks are PyTorch, Tensorflow, and Keras. In fact, Keras has been recently integrated into Tensorflow and the two are now considered the main entry points for `R` users to enter the realm of deep learning.

We start by implementing a shallow feed-forward neural network model. As our dataset is relatively small, we use only two hidden layers with 64 units each. The number of inputs (`input_shape`) to the first hidden layer is determined by the dimensionality of the predictor set while the number of output nodes (`units` in the output layer) is determined by the task at hand, i.e. the neural net outputs one continuous values for regression problems but one probability per class for classification problems. Complete the following code chunk in order to initialize the model.

```
nn_mod <- keras::keras_model_sequential() %>%
  # first hidden layer
  layer_dense(units = ___, activation = "relu", input_shape = ____) %>%
  # second hidden layer
  layer_dense(units = ___, activation = "relu") %>%
  # output layer
  layer_dense(units = ___)

nn_mod
```

[1] You can think of a deep learning framework as a software package that allows you to easily write, train and evaluate neural network models. Importantly, it comes with high-level functions that abstract a lot of the underlying (mathematical) complexity away (similar to what `tidymodels` tries to achieve as a machine learning framework in `R`). Note that in this assignment we can only cover the basic use of a deep learning framework, such as `keras`, and point to the respective textbooks for an in-depth survey of these frameworks (e.g., Deep Learning with R, Deep Learning with Python, or Deep Learning with PyTorch).

We prepare our model for training by specifying how gradient descent should be performed. In essence, `compile` communicates your training details to the underlying `python` process that will run the training procedure.

```
nn_mod %>%
  keras::compile(
    optimizer = "rmsprop",
    loss = "mse",
    metrics = list("mae")
  )
```

In this example, we learn the optimal model coefficients (i.e. weights and biases) by minimizing the models mean squared error (`loss = "mse"`). Gradient descent is implemented using the *RMSprop* algorithm (see this blog post by Sebastian Ruder for alternative gradient descent algorithms). Roughly speaking, you can think of *RMSprop* as an advanced version of the plain vanille back-propagation algorithm which executes the following main steps:

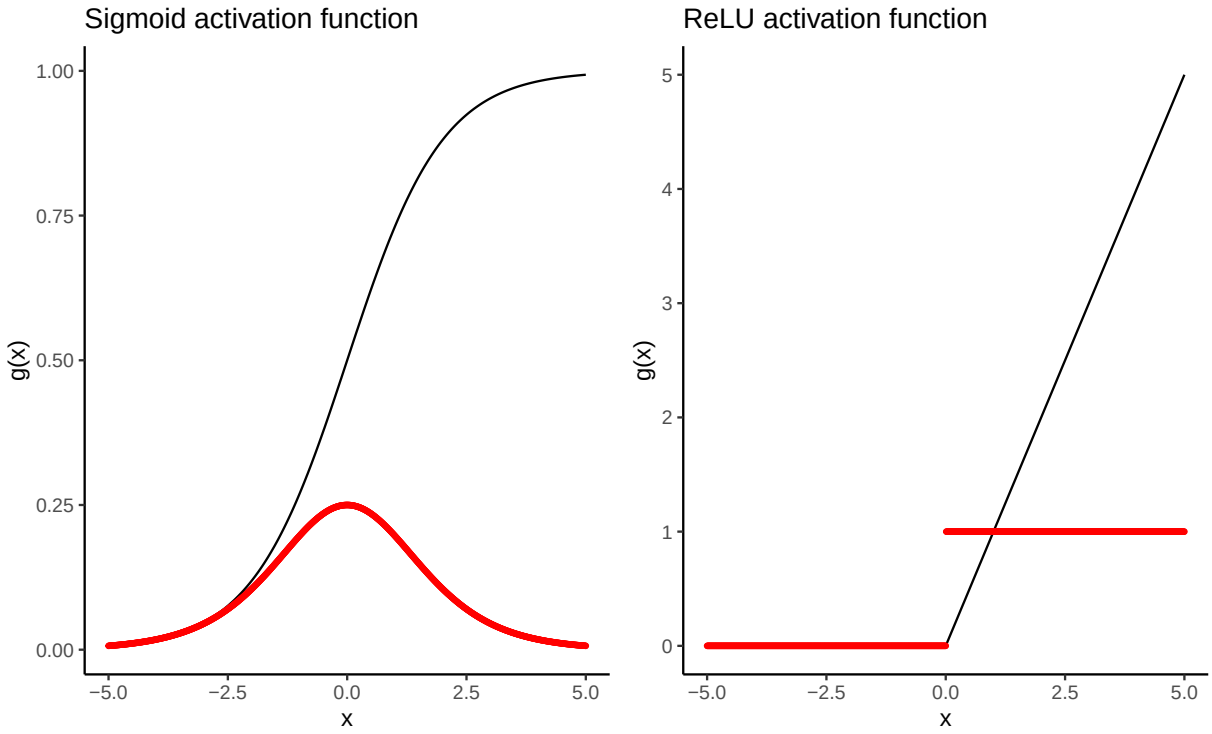
1. Calculate the output of each node in a neural network based on the current model weights (*forward pass*).
2. Calculate the partial derivative of the loss (here we minimize the `mse`) with respect to each model weight (i.e. the *gradient*) to identify the direction in which weights must be updated.
3. Update the model weights according to the gradients and the specified learning rate (*backward pass*).
4. Repeat step 1-3.

Task 1.6

Below, you find a code snippet that illustrates the *Sigmoid* as well as *ReLU* activation function. It displays the activations `g(x)` for a range of values for `x` and depicts the derivative of the activation function in red. Briefly describe the purpose of a non-linear activation function (i.e. what happens if a neural network is constructed without one)? In addition, discuss the pros and cons of both activation functions.

```
sigmoid <- function(x) 1 / (1 + exp(-x))
relu    <- function(x) pmax(0, x)

activations <-
  tibble(x = seq(-5, 5, 0.01), sigm = sigmoid(x), relu = relu(x)) %>%
  dplyr::mutate(
    deriv_sigm = sigm * (1-sigm),
    deriv_relu = dplyr::if_else(relu == 0, 0, 1)
  )

p_sigmoid <-
  activations %>%
  ggplot2::ggplot() +
  geom_line(aes(x = x, y = sigm)) +
  geom_point(aes(x = x, y = deriv_sigm), color = "red", size = .75) +
  theme_classic() +
  labs(y = "g(x)", title = "Sigmoid activation function")

p_relu <-
  activations %>%
  ggplot2::ggplot() +
  geom_line(aes(x = x, y = relu)) +
```

```
  geom_point(aes(x = x, y = deriv_relu), color = "red", size = .75) +
  theme_classic() +
  labs(y = "g(x)", title = "ReLU activation function")

library(patchwork)

p_sigmoid | p_relu
```



Hint: This task may require you to perform some additional research to answers this question convincingly.

Task 1.7

Finally, let's train the neural network model (running the code below may take some time depending on your hardware). Similar to the `tidymodels` framework, `keras` exposes a `fit` function to execute the model training routine. In this example, we wrap our training procedure in a 4-fold cross-validation loop to obtain a more robust estimate of model performance. By setting `verbose` to `2` in the call to `fit`, `keras` prints additional training statistics and spawns a tensorboard in your RStudio *Viewer* pane to help you track the model error during training.

```
set.seed(2021)

init_nn <- function() {

  nn <- keras::keras_model_sequential() %>%
    # first hidden layer
    layer_dense(units = ___, activation = "relu", input_shape = ____) %>%
    # second hidden layer
```

```
    layer_dense(units = ___, activation = "relu") %>%
    # output layer
    layer_dense(units = ___)

  nn %>%
    keras::compile(
      optimizer = "rmsprop",
      loss = "mse",
      metrics = c("mae")
    )

}

folds <- rsample::vfold_cv(x_train, v = 4)
log <- NULL

for (k in seq_along(folds$splits)) {

  cat("--- Processing", folds %>% purrr::pluck("id", k), "---\n\n")

  # extract ids of training samples per fold
  ids <- folds %>% purrr::pluck("splits", k) %>% .$in_id

  # index into overall training data to obtain CV train and validation set
  x_val <- x_train[-ids,]
  y_val <- y_train[-ids]
  x_train_cv <- x_train[-ids,]
  y_train_cv <- y_train[-ids]

  # initialize new model
  nn_mod <- init_nn()
  nn_mod %>%
    keras::compile(
      optimizer = "rmsprop",
      loss = "mse",
      metrics = c("mae")
    )

  # specify hyperparameters
  epochs  <- 1500
  bs      <- 32

  # fit the neural network and track training time
  ptm <- Sys.time()

  train_results <- nn_mod %>%
    keras::fit(
      x = x_train_cv, y = y_train_cv, validation_data = list(x_val, y_val),
      shuffle = T, epochs = epochs, batch_size = bs, verbose = 0
    )

  cat("Processing time:", Sys.time() - ptm, "\n\n")
```

```
  # record the cross-validation errors
  mae <- train_results$metrics$val_mae
  mse <- train_results$metrics$val_loss
  log <- rbind(log, mae, mse)

}
```

What is the meaning of `epochs` in the call to `keras::fit`? What happens when you increase or decrease the value? Currently, you are running *mini-batch gradient descent* (`batch_size = 32`). Re-run the training loop using *stochastic gradient descent (SGD)* and *batch gradient descent* by adjusting `bs` accordingly. How does this design choice influence your cross-validation errors over the course of the training run? What are your final (average) cross-validation errors (MSE and MAE, as stored in `log`) for each of the three specifications? How many training steps (i.e. iterations) per epoch do you observe when implementing SGD, mini-batch GD, and batch GD?

Note: When using mini-batch gradient descent you generally use batch sizes of 2^n*. In particular, with very large datasets, you would increase* `bs` *up to a point where the batch still fits into your computer's memory. Further, to gain some intuition on why data points are shuffled before each step (i.e.* `shuffle = T`*) cf. LeCun et al. (1998), ch. 4.2. Finally, note that when calling* `fit` *on a* `keras` *model, it resumes the training process (in lieu of fitting a new model from scratch). To prevent continuous learning after each iteration of our CV loop, we initialize a new model and train that one from scratch (otherwise our CV error would further decrease after each fold).*

Task 1.8

In a last step, you are supposed to replicate the tensorboards that appear if `verbose = 2` from the previous task yourself. First, re-run the code from *Task 1.7* using mini-batch GD (with a batch size of 32) and set `epochs = 1500`. Second, use the code snippet below to collect your cross-validation metrics from the `log` object. Third, plot the cross-validation MSE and MAE using `ggplot2::ggplot()` and `facet_wrap()`. According to your two plots, would you argue that the network already starts to overfit on your data? If yes, why and when approximately? If no, how could you change the training routine to identify overffiting?

```
errors <-
  tibble(
    epoch = c(1:epochs, 1:epochs),
    metric = c(rep("mae", epochs), rep("mse", epochs)),
    value = c(
      apply(log[rownames(log) == "mae", ], 2, mean),
      apply(log[rownames(log) == "mse", ], 2, mean)
    )
  )

errors
```

Task 2: Feed-Forward Neural Networks II

In this task, you will work with the `College` dataset from the `ISLR` package (run `help(College)` to access the data dictionary) to predict `Outstate`. `Outstate` records the out-of-state tuition fees for 777 U.S. colleges (in the U.S., students who attend a public college in a state outside of their residency typically pay more than in-state students). As always, we start with a brief look at the data and subsequently split it into a training and test set.

```
data("College", package = "ISLR")

glimpse(College)
```

```
> Rows: 777
> Columns: 18
> $ Private     <fct> Yes, Yes, Yes, Yes, Yes, Yes, Yes, Yes, Yes, Yes, Yes, Yes~
> $ Apps        <dbl> 1660, 2186, 1428, 417, 193, 587, 353, 1899, 1038, 582, 173~
> $ Accept      <dbl> 1232, 1924, 1097, 349, 146, 479, 340, 1720, 839, 498, 1425~
> $ Enroll      <dbl> 721, 512, 336, 137, 55, 158, 103, 489, 227, 172, 472, 484,~
> $ Top10perc   <dbl> 23, 16, 22, 60, 16, 38, 17, 37, 30, 21, 37, 44, 38, 44, 23~
> $ Top25perc   <dbl> 52, 29, 50, 89, 44, 62, 45, 68, 63, 44, 75, 77, 64, 73, 46~
> $ F.Undergrad <dbl> 2885, 2683, 1036, 510, 249, 678, 416, 1594, 973, 799, 1830~
> $ P.Undergrad <dbl> 537, 1227, 99, 63, 869, 41, 230, 32, 306, 78, 110, 44, 638~
> $ Outstate    <dbl> 7440, 12280, 11250, 12960, 7560, 13500, 13290, 13868, 1559~
> $ Room.Board  <dbl> 3300, 6450, 3750, 5450, 4120, 3335, 5720, 4826, 4400, 3380~
> $ Books       <dbl> 450, 750, 400, 450, 800, 500, 500, 450, 300, 660, 500, 400~
> $ Personal    <dbl> 2200, 1500, 1165, 875, 1500, 675, 1500, 850, 500, 1800, 60~
> $ PhD         <dbl> 70, 29, 53, 92, 76, 67, 90, 89, 79, 40, 82, 73, 60, 79, 36~
> $ Terminal    <dbl> 78, 30, 66, 97, 72, 73, 93, 100, 84, 41, 88, 91, 84, 87, 6~
> $ S.F.Ratio   <dbl> 18.1, 12.2, 12.9, 7.7, 11.9, 9.4, 11.5, 13.7, 11.3, 11.5, ~
> $ perc.alumni <dbl> 12, 16, 30, 37, 2, 11, 26, 37, 23, 15, 31, 41, 21, 32, 26,~
> $ Expend      <dbl> 7041, 10527, 8735, 19016, 10922, 9727, 8861, 11487, 11644,~
> $ Grad.Rate   <dbl> 60, 56, 54, 59, 15, 55, 63, 73, 80, 52, 73, 76, 74, 68, 55~
```

```
set.seed(2021)

college_split <- College %>%
  dplyr::mutate(across(Private, as.numeric)) %>%
  rsample::initial_split(prop = 0.9, strata = Outstate, breaks = 5)
```

Task 2.1

As discussed in *Task 1*, preprocessing input signals is important before fitting a neural network. Apply feature-wise normalization to the predictors (but not to the response!). You may either recycle code from *Task 1.4* or use the `recipes` package and `step_normalize` to perform feature normalization.

Task 2.2

Write down the equation (similar to p. 404 and p. 408 of the ISLR textbook) which describes a feed-forward network that predicts `Outstate` with 2 hidden layers using ReLU activation functions and 64 units each. Please describe all symbols used in your equation (in particular indexes). When working on classification problems, we would usually employ the softmax as output layer activation (the so-called *softmax layer*) to

squish model signals into the [0; 1]-interval to resemble probabilities. What activation function do we choose for the output layer in neural networks for regression?

Hint: You may write down the equation in LaTeX. Alternatively, if you rather trust you handwriting, you could also take a screenshot/photo and embed into this notebook using markdown syntax `![](<path-to-file>)` *or* `knitr::include_graphics()`.

Task 2.3

Next, train the neural network that you specified in *Task 2.2* using `keras`. Please consider the following points in your implementation:

1. **Replicability:** Use `set.seed(2021)` to ensure that your results are replicable (since neural network weights are initialized randomly, this will secure that initial weights do not change between runs). *Hint: Fitting* `keras` *models can take some time. If you want to prevent your entire code to be executed each time you try to* `knit` *your* `.Rmd` *document, you may cache your results using* `cache=TRUE` *in the chunk header.*
2. **Training Setup:** Train your network for 300 epochs using RMSprop as gradient descent algorithm and a mini-batch size of 8. Optimize for the MSE by setting `loss = "mse"` in the `compile` function. In addition, let `keras` also compute MAE statistics along the way (by setting `metrics = c("mae")` in the compiler).
3. **Training Budget:** Use the normalized training data from `college_split` for model training (i.e. 90% of the overall data). Before doing so, split this training data once more using the validation split approach into 80% actual training data and 20% validation data (i.e. the actual training amounts to $0.9 * 0.8 = 0.72$ of the overall data). *Hint: the* `fit` *function exposes a* `validation_split` *argument (*`help("fit.keras.engine.training.Model")`*) which automatically performs a validation split internally. Alternatively, you can make this split explicit using the* `rsample` *package and use the* `validation_data` *argument, similar to how we did it in Task 1.7.*
4. **Train Results:** Plot the training and validation loss (i.e. MSE) as a function of the number of epochs. In addition, what is the total number of training steps (e.g., if your dataset is of size 320 and your mini-batch size is 32, then the training loop runs 10 training steps per epoch)?
5. **Generalization Performance:** Finally, report the MAE and $RMSE$ on the out-of-sample test set (i.e. the remaining 10% data points contained in `college_split`).

Task 2.4

If you inspect the `keras` model object that you've created in *Task 2.3*, you will realize that the number of network parameters exceeds the number of training samples by an order of magnitude (i.e. $p >> n$). Due to the tremendous amount of different model parameters that need to be estimated, neural networks are highly prone to overfitting on the training data. To alleviate this issue, the ML community devised various regularization techniques, some of which you have learned about in the lecture:

1. ℓ_1/ℓ_2-regularization, aka weight decay
2. Layer-wise variable dropout
3. Early stopping
4. Data augmentation (i.e. noise induction)

Choose one of the first two approaches and implement it in your feed-forward network from *Task 2.3* (all else being equal). If you select *weight decay*, optimize over the following decay rates: `seq(0.1, 0.4, 0.1)`. If you select *variable dropout* for your implementation instead, optimize over the following penalties: `10^seq(-5, -1, 1)`. What hyperparameter value yields the best test set $RMSE$? Importantly, does it improve the performance of your network relative to *Task 2.3*? Finally, choose one of the latter two regularization

techniques (i.e. *early stopping* or *data augmentation*) and briefly explain the main underlying idea and how regularization is achieved.

Note: Please again use `set.seed(2021)` *to make results comparable. Further, you might have to consult the official* `keras` *documentation to get an idea of how regularization and dropout can be implemented. In order to optimize over the given hyperparameter ranges, you may find it helpful to implement a* `for`*-loop and log your results after each iteration.*

[Optional]: You might even try out your own network architecture to predict out-of-state-tuition (no worries, this won't be graded!).

Optional: Pretrained Neural Networks

This task is entirely optional, i.e., you will not have to answer any questions at all. Hopefully, though, you will get a grasp of what deep neural networks are capable of. In this task, we will not train a neural network ourselves but instead download a *pretrained model* from the internet. These days, there are various *model hubs* (with huggingface being likely the most prominent) which host a wide variety of pretrained models for all kinds of machine learning tasks (e.g., document and image classification, text generation, or even sound detection). In general, these models are so large and have been trained for so many epochs that you could not easily replicate them on your local machine. The idea is rather to download these models, including all the pretrained model weights (we refer to this technique as *weight freezing*), and to use them for your own *downstream* task.

In the following, we employ one of these models, namely the popular **ResNet50** model. *ResNet50* is a very deep convolutional neural network (CNN), involving 50 layers, and a spiritual successor to the 2012 *AlexNet* model which is widely considered as the pivotal invention that led to the deep learning revolution as we have witnessed it in the recent years. In case you feel confident enough to test the waters, you might even check out the seminal *AlexNet* and *ResNet* paper.

The *ResNet50* model that we employ has been pretrained on the famous *Imagenet* dataset. The *Imagenet* project is one of the greatest human efforts to construct high-quality training data for computer vision. By now, roughly 14 million images have been manually classified into more than 20,000 classes. We will start by downloading and importing the trained model using `keras` (print the `resnet_mod` by uncommenting the second code line to inspect the model architecture):

```
resnet_mod <- keras::application_resnet50(weights = "imagenet")
# resnet_mod
```

Second, we download some dummy images from the ISLR website in order try out the performance of the pretrained model on unseen data:

```
url <- "https://www.statlearning.com/s/book_images.zip"
file_name <- basename(url)

# download and unzip in your assignment folder
if (!file.exists(file_name)) {
  download.file(url, basename(url))
  unzip(file_name)
}
```

Third, we need to import the downloaded images and store them in a format that can be processed by the network. In particular, we will store each image as a three-dimensional array respectively *tensor* (in the deep learning jargon, we usually refer to higher-dimensional arrays as tensors). The *ResNet50* model expects the tensors to be of shape $(224, 224, 3)$, with the first two dimensions reflecting the image's height and width (in pixels) and the third reflecting the three color channels involved in an image (i.e. RGB). The values of this tensor will represent the image's pixel values, ranging from 0 to 255. Note that if the image height and width differ from this specification, `keras` will rescale (i.e. interpolate) the image for you (via `image_load`).

```
dir_name    <- tools::file_path_sans_ext(file_name)
img_names   <- list.files(dir_name)

# construct tensor template
x <- array(dim = c(length(img_names) , 224, 224, 3))

# import images into tensor
```

```
for (i in 1:length(img_names)) {

  img_path  <- paste(dir_name, img_names[i], sep = "/")
  image     <- image_load(img_path, target_size = c(224, 224))
  x[i,,,]   <- image_to_array(image)

}

x <- keras::imagenet_preprocess_input(x)

dim(x)
```

```
> [1]   6 224 224   3
```

Note that our tensor `x` exhibits a fourth dimension (the first dimension here) which reflects our batch size. Finally, we can deploy the *ResNet50* model to issue predictions. Let's see what it says and how confident it is in each case:

```
class_preds <- resnet_mod %>%
  predict(x) %>%
  keras::imagenet_decode_predictions(top = 3)

names(class_preds) <- img_names

print(class_preds)
```

```
> $flamingo.jpg
>   class_name class_description       score
> 1  n02007558          flamingo 0.926349938
> 2  n02006656         spoonbill 0.071699433
> 3  n02002556       white_stork 0.001228211
>
> $hawk.jpg
>   class_name class_description     score
> 1  n03388043          fountain 0.2788653
> 2  n03532672              hook 0.1785543
> 3  n03804744              nail 0.1080727
>
> $hawk_cropped.jpeg
>   class_name class_description      score
> 1  n01608432              kite 0.72270924
> 2  n01622779    great_grey_owl 0.08182573
> 3  n01532829       house_finch 0.04218878
>
> $huey.jpg
>   class_name           class_description      score
> 1  n02097474             Tibetan_terrier 0.50929672
> 2  n02098413                       Lhasa 0.42209941
> 3  n02098105 soft-coated_wheaten_terrier 0.01695856
>
> $kitty.jpg
>   class_name    class_description      score
```

```
> 1  n02105641 Old_English_sheepdog 0.83265990
> 2  n02086240             Shih-Tzu 0.04513895
> 3  n03223299              doormat 0.03299776
>
> $weaver.jpg
>   class_name class_description      score
> 1  n01843065           jacamar 0.49795419
> 2  n01818515             macaw 0.22193287
> 3  n02494079   squirrel_monkey 0.04287856
```

You can even test the model on your own images or download some free pictures from sources like Unsplash to evaluate the model in a real-world context. Can you trick it into issuing false predictions?

Task 3: Unsupervised Learning (Clustering)

In this task, you will perform hierarchical as well as k-means clustering on the `USArrests` data.

```
data(USArrests, package = "datasets")

USArrests %>%
  tibble::as_tibble()
```

```
> # A tibble: 50 x 4
>    Murder Assault UrbanPop  Rape
>     <dbl>   <int>    <int> <dbl>
>  1   13.2     236       58  21.2
>  2   10       263       48  44.5
>  3    8.1     294       80  31
>  4    8.8     190       50  19.5
>  5    9       276       91  40.6
>  6    7.9     204       78  38.7
>  7    3.3     110       77  11.1
>  8    5.9     238       72  15.8
>  9   15.4     335       80  31.9
> 10   17.4     211       60  25.8
> # ... with 40 more rows
```

Data dictionary:

- `Murder`: murder arrests (per 100,000)
- `Assault`: assault arrests (per 100,000)
- `UrbanPop`: percent urban population
- `Rape`: rape arrests (per 100,000)

Task 3.1

Cluster the U.S. states using hierarchical clustering with complete linkage and Euclidean distance as dissimilarity measure. Plot the result and use `color_branches()` from the `dendextend` package to colorize the `k = 5` most dissimilar clusters. Which variable is plotted on the y-axis of the dendrogram plot? Use `set.seed(2021)`.

Task 3.2

Suppose you are satisfied with the five cluster solution generated by the algorithm. What is the cutoff value to receive the five clusters? How many states are included in each cluster? Which states does the smallest cluster contain? Which states are the two most dissimilar ones and what is their Euclidean distance?

Task 3.3

If you carefully inspect the data dictionary stated at the beginning of this task, you will realize that `Murder`, `Rape`, `Assault` are on a different scale than `UrbanPop` (100,000 vs. percent * 100). Redo *Task 3.1* and hierarchically cluster the states using complete linkage and Euclidean distance after scaling the predictors to a zero-mean and a standard deviation of one (*z-normalization*). What is the new cutoff value to receive the five clusters and why has it changed? How many states are there per cluster? Which states does the second largest cluster contain? Use `set.seed(2021)`.

Task 3.4

Lastly, you are supposed to implement k-means clustering to validate if `k = 5` is indeed the optimal number of clusters for the normalized data. Employ the `stats::kmeans()` function as well as the `purrr` package to perform k-means clustering and iterate over a sequence of 15 values for `k` (from 1 to 15). Eventually, the goal is to produce an *elbow plot* that allows you to identify the optimal number of clusters by evaluating the decrease in the total within-cluster sum of squares (`tot.withinss`). You may use the following code snippet and replace the various placeholders (`___`) accordingly:

```
set.seed(2021)

k <- ___

kmeans_res <- purrr::map_df(
  .x = k,
  .f = ~ stats::kmeans(___, .x, iter.max = 1000, nstart = .x) %>%
    broom::glance() %>%
    dplyr::mutate(k = .x)
  )

kmeans_res %>%
  ggplot2::ggplot(aes(x = k, y = ___)) +
    geom_point(size = 3, color = "blue4") +
    geom_line(alpha = .25, lty = "dashed") +
    scale_x_continuous(breaks = c(1:15)) +
    labs(title = "k-means elbow plot",
         y = "Total within-cluster sum of squares") +
    theme_classic()
```

According to the *elbow criterion*, what would you argue is the optimal number of clusters `k` and why? Refit the k-means model using the optimal `k`. What are the sizes of the `k` clusters? Which states share a cluster with West Virginia?

Task 3.5

Which of the following statements are true, which are false? Explain your answer in case you select "false".

1. In principal component analysis (PCA), the second principal component is the linear combination of the predictors that has the largest variance of all the linear combinations that are uncorrelated with the first principal component.
2. It makes no difference for the results of PCA if the variables are standardized beforehand or not.
3. The results of k-means clustering are robust to the random initialization of cluster centers.
4. PCA is most helpful when all predictors are uncorrelated.