Module 11: Recommended Exercises

TMA4268 Statistical Learning V2024

Daesoo Lee, Kenneth Aase, Stefanie Muff, Sara Martino
Department of Mathematical Sciences, NTNU

April 18, 2024

Keras Installation

First install kears by running:

```
install.packages("keras")
library(keras)
install_keras()
```

The library is installed on a virtual environment named "r-tensorflow".

NB! by installing Keras, Tensorflow is also installed.

After installation and session restart, we first need to activate the virtual environement by running:

```
library(reticulate)
use_virtualenv("r-tensorflow")
```

Test your installation by running:

```
library(tensorflow)
tf$constant("Hello TensorFlow!")
```

In addition, you need to install the necessary libraries on the virtual environment like

```
install.packages("NeuralNetTools")
install.packages("caret")
install.packages("dplyr")
install.packages("ggplot2")
install.packages("TSrepr")
install.packages("ramify")
```

Problem 1

a)

Write down the equation that describes and input is related to output in this network, using general activation functions ϕ_o, ϕ_h and ϕ_{h^*} and bias nodes in all layers. What would you call such a network?

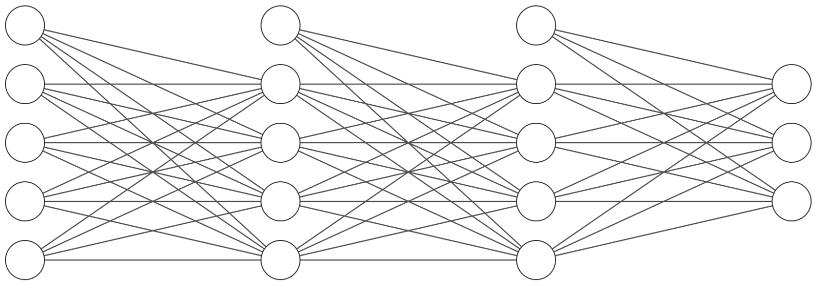

Figure 1: Image created here http://alexlenail.me/NN-SVG/index.html

b)

The following image is the illustration of an artificial neural network at Wikipedia.

- What can you say about this network architecture
- What do you think it can be used for (regression/classification)?

c)

What are the similarities and differences between a feedforward neural network with one hidden layer with `linear` activation and `sigmoid` output (one output) and logistic regression?

d)

In a feedforward neural network you may have $10'000$ weights to estimate but only 1000 observations. How is this possible?

Problem 2

a)

Which network architecture and activation functions does this formula correspond to?

$$\hat{y}_1(\mathbf{x}) = \beta_{01} + \sum_{m=1}^{5} \beta_{m1} \cdot \max(\alpha_{0m} + \sum_{j=1}^{10} \alpha_{jm} x_j, 0)$$

How many parameters are estimated in this network?

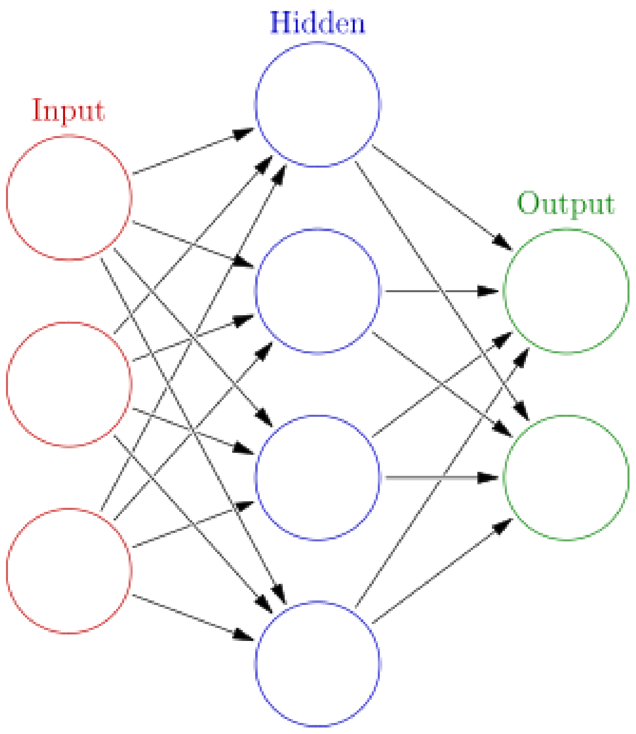

Figure 2: Image taken from https://commons.wikimedia.org/wiki/File:Colored_neural_network.svg

b)

Which network architecture and activation functions does this formula give?

$$\hat{y}_1(\mathbf{x}) = (1 + \exp(-\beta_{01} - \sum_{m=1}^{5} \beta_{m1} \max(\gamma_{0m} + \sum_{l=1}^{10} \gamma_{lm} \max(\sum_{j=1}^{4} \alpha_{jl} x_j, 0), 0))^{-1}$$

How many parameters are estimated in this network?

c)

In a regression setting: Consider

- A sum of non-linear functions of each covariate in Module 7.
- A sum of many non-linear functions of sums of covariates in feedforward neural networks (one hidden layer, non-linear activation in hidden layer) in Module 11.

Explain how these two ways of thinking differ? Pros and cons?

Problem 3: Regression with Feedforward Neural Network (FNN)

The following problem involves training a feedforward neural network on the Boston Housing Prices dataset. The Boston Housing Prices dataset is a collection of housing prices data from the Boston area, containing 506 samples with 13 numerical features each. The features include factors such as crime rate, average number of rooms per dwelling, property tax rate, and more. The goal is to predict the median value of owner-occupied homes in $1000s.

In this example, we will design our feedforward neural network (FNN) architecture using a series of fully connected (dense) layers. The model will take the 13 input features and learn to map them to a single output representing the predicted median housing price. To accomplish this, the network will be trained using an appropriate loss function, such as mean squared error, and an optimization algorithm like stochastic gradient descent or Adam.

```r
# load
boston_housing <- dataset_boston_housing()
x_train <- boston_housing$train$x
y_train <- boston_housing$train$y
x_test <- boston_housing$test$x
y_test <- boston_housing$test$y

# preprocess
mean <- apply(x_train, 2, mean)
std <- apply(x_train, 2, sd)
x_train <- scale(x_train, center = mean, scale = std)
x_test <- scale(x_test, center = mean, scale = std)
```

1. Load and preprocess data

a) Fill in the missing parts in the following steps and run the model.

```r
model_r <- keras_model_sequential() %>%
  layer_dense(units = 64, activation = "relu", input_shape = ...) %>%  # fill in the length of the inpu
  layer_dense(units = 32, activation = "relu") %>%
  layer_dense(units = ...)  # fill in the `units` (i.e., output size) of the last layer.

summary(model_r)
```

2. Define the model What should the output size be to be compatible with y?

```r
model_r %>% compile(
  loss = "...",  # fill in the loss function.
  optimizer = optimizer_adam(learning_rate = 0.001),  # adam is the most common optimizer for its robus
  metrics = c("mean_absolute_error")
)
```

3. Compile For the loss function, choose one among 'binary_crossentropy', 'categorical_crossentropy' and 'mean_squared_error'.

```r
history <- model_r %>% fit(
  x_train, y_train,
  epochs = 100,
  batch_size = 32,
  validation_data = list(x_test, y_test)
)
```

4. Train the model

```r
scores <- model_r %>% evaluate(x_test, y_test, verbose = 0)

cat("Test loss (MSE):", scores[[1]], "\n",
    "Test mean absolute error (MAE):", scores[[2]], "\n")
```

5. Test

```r
plot(history)
```

Plot training history

```r
predictions <- model_r %>% predict(x_test)
plot_df <- data.frame(Predicted = predictions, Actual = y_test)
ggplot(plot_df, aes(x = Actual, y = Predicted)) +
  geom_point() +
  geom_abline(slope = 1, intercept = 0, color = "red", linetype = "dashed") +
  theme_bw() +
  xlab("Actual Values") +
  ylab("Predicted Values") +
  ggtitle("Predicted vs. Actual Values (Feedforward NN)") +
  xlim(0, 55) +
  ylim(0, 55)
```

Additional plot: confusion matrix

b) Fit a linear regression model and compare its performance (i.e., MSE, MAE) to that of the feedforward network.

```r
# Fit a linear regression model
linear_model <- lm(..., data = as.data.frame(cbind(x_train, y_train)))  # fill in the expression
```

```r
# Make predictions on the test set
predictions <- predict(linear_model, as.data.frame(x_test))

# Calculate the mean squared error and mean absolute error
mse <- ...  # fill in (write an expression to compute MSE)
mae <- ...  # fill in (write an expression to compute MAE)

cat("=== [Feedforward Neural Network] === \n", "Test loss (MSE):", scores[[1]],
    "\n",
    "Test mean absolute error (MAE):", scores[[2]],
    "\n",
    "==================================== \n\n",
    "=== [Linear Regression] === \n",
    "Test loss (MSE):", mse, "\n",
    "Test mean absolute error (MAE):", mae, "\n",
    "===========================\n\n")
```

```r
plot_df <- data.frame(Predicted = predictions, Actual = y_test)
ggplot(plot_df, aes(x = Actual, y = Predicted)) +
  geom_point() +
  geom_abline(slope = 1, intercept = 0, color = "red", linetype = "dashed") +
  theme_bw() +
  xlab("Actual Values") +
  ylab("Predicted Values") +
  ggtitle("Predicted vs. Actual Values (Linear Regression)") +
  xlim(0, 55) +
  ylim(0, 55)
```

Comparison to a Linear Regression Model

c) Please share your thoughts on the comparative performance of the two models based on their results.

Problem 4: Convolutional Neural Network (CNN)

Problem 4.1: Image Classification with CNN

The following problem involves training a Convolutional Neural Network (CNN) model on an image dataset, called CIFAR-10. The CIFAR-10 dataset is a collection of 60,000 32x32 color images in 10 classes, with 6,000 images per class. The dataset is split into 50,000 training images and 10,000 testing images. The 10 classes include airplanes, automobiles, birds, cats, deer, dogs, frogs, horses, ships, and trucks. The examples and description of CIFAR-10 can be found here.

In this example, we have designed our CNN model architecture following established models such as AlexNet [1], VGG [2], and ResNet [3]. Our design incorporates a common pattern where the channel dimension size (i.e., filters) increases while the input shape to each layer decreases across the layers (see the figure below). It is also worth noting that a convolutional layer is sometimes followed by a non-linear pooling layer which reduces the spatial dimension size. One most common pooling layer is the max pooling layer (see the figure below). Our example model follows the same protocol while keeping the model size small.

[1] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." Communications of the ACM 60.6 (2017): 84-90. [2] Simonyan, Karen, and

Andrew Zisserman. “Very deep convolutional networks for large-scale image recognition.” arXiv preprint arXiv:1409.1556 (2014). [3] He, Kaiming, et al. “Deep residual learning for image recognition.” Proceedings of the IEEE conference on computer vision and pattern recognition. 2016.

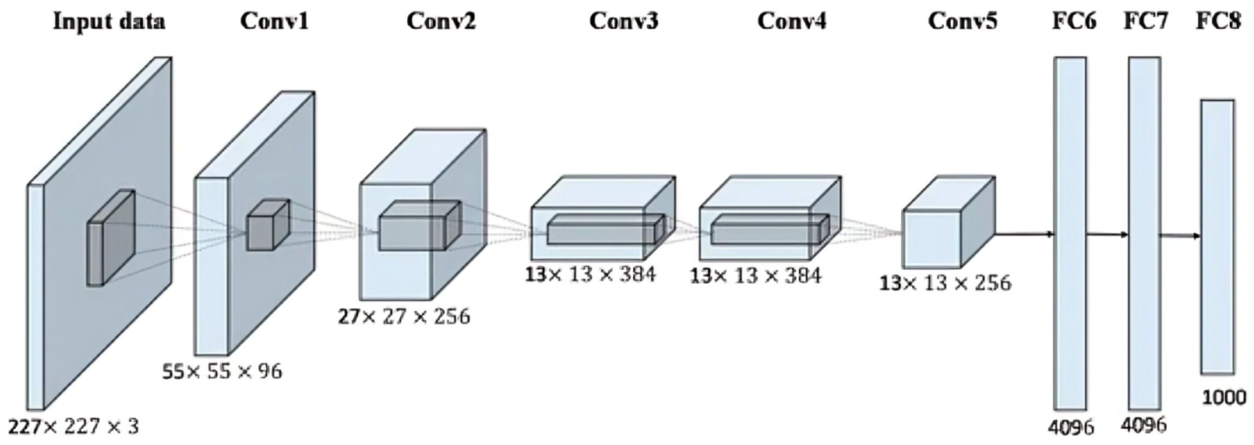

Figure 3: Fig. Architecture of AlexNet. The input data has a dimension of (height x width x #channels. #channels is 3 for RGB colors. The common CNN architecture design is the increase in #channels and decrease in height-width size across the layers.)

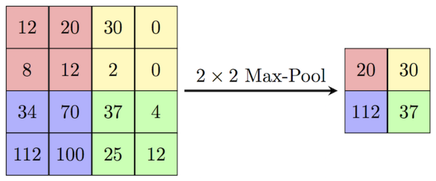

Figure 4: Fig. Max Pooling Layer: The 2x2 max pooling layer decreases the spatial dimensions by a factor of 2 in both height and width, applying the maximum operation within each local region.)

```r
cifar10 <- dataset_cifar10()
x_train <- cifar10$train$x / 255
y_train <- to_categorical(cifar10$train$y, num_classes = 10)
x_test <- cifar10$test$x / 255
y_test <- to_categorical(cifar10$test$y, num_classes = 10)
```

1. Load and preprocess data

a) Fill in the missing parts in the following steps and run the model.

```r
model_c <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = c(3, 3), activation = "relu", input_shape = c(32, 32, 3)) %
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 64, kernel_size = c(3, 3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_flatten() %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dense(units = 10, activation = "softmax")

summary(model_c)
```

2. Define the model

```
model_c %>% compile(
  loss = "...",
  optimizer = optimizer_adam(learning_rate = 0.001),  # adam is the most common optimizer for its robus
  metrics = c("accuracy")
)
```

3. Compile

For the loss function, choose one among 'binary_crossentropy', 'categorical_crossentropy' and 'mean_squared_error'.

```
history <- model_c %>% fit(
  x_train, y_train,
  epochs = 20,
  batch_size = 64,
  validation_data = list(x_test, y_test)
)
```

4. Train the model

```
scores <- model_c %>% evaluate(x_test, y_test, verbose = 0)

cat("Test loss:", scores[[1]], "\n",
    "Test accuracy:", scores[[2]], "\n")
```

5. Test

```
plot(history)
```

Plot training history

```r
library(caret)
predictions <- model_c %>% predict(x_test)%>% k_argmax()
y_true <- cifar10$test$y
confusion_matrix <- confusionMatrix(factor(as.vector(predictions)), factor(y_true))
print(confusion_matrix$table)
```

Additional plot: confusion matrix

b) Compute the misclassification error given the confusion matrix.

Problem 4.2: Improving the test accuarcy with data augmentation techniques

Data augmentation techniques are used to artificially increase the size of the training dataset by applying various transformations to the original images. This helps to improve the robustness and generalization ability of CNN models. Data augmentation can be viewed as a regularization technique for better generalization.

Here are some commonly used data augmentation techniques for image data (the visual examples can be found here):

- Rotation: Rotating the image by a certain degree to generate new samples.
- Flip: Flipping the image horizontally or vertically to generate new samples.
- Zooming: Zooming into or out of the image by a certain factor to generate new samples.
- Translation: Shifting the image horizontally or vertically by a certain distance to generate new samples.
- Shearing: Tilting the image in a particular direction by a certain angle to generate new samples.
- Noise addition: Adding random noise to the image to generate new samples.
- Color jittering: Modifying the brightness, contrast, or hue of the image to generate new samples.
- Cropping: Cropping a portion of the image to generate new samples.

By applying these transformations to the original images, we can generate new samples that are different from the original ones but still share the same class label. This helps the model to learn to recognize the important features of the images, regardless of their position or orientation, and make more accurate predictions on unseen data.

In this example, we apply several simple data augmentations such as random rotations, width and height shifts, and horizontal flips.

```r
# 1) Load and preprocess data
cifar10 <- dataset_cifar10()
x_train <- cifar10$train$x / 255
y_train <- to_categorical(cifar10$train$y, num_classes = 10)
x_test <- cifar10$test$x / 255
y_test <- to_categorical(cifar10$test$y, num_classes = 10)

# 2) Define the model
model_ca <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = c(3, 3), activation = "relu", input_shape = c(32, 32, 3)) %
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 64, kernel_size = c(3, 3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_flatten() %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dense(units = 10, activation = "softmax")

# 3) Compile
model_ca %>% compile(
  loss = "categorical_crossentropy",
  optimizer = optimizer_adam(learning_rate = 0.001),
  metrics = c("accuracy")
)

# 4) Data augmentation
datagen <- image_data_generator(
  rotation_range = 10,
  width_shift_range = 0.1,
  height_shift_range = 0.1,
  horizontal_flip = TRUE
)

# Compute the data generator internal statistics
datagen %>% fit_image_data_generator(x_train)

# 5) Train the model with data augmentation
batch_size = 64
train_generator <- flow_images_from_data(x = x_train, y = y_train, generator = datagen, batch_size = ba

history <- model_ca %>% fit_generator(
  generator = train_generator,
  steps_per_epoch = as.integer(nrow(x_train) / batch_size),
  epochs = 20,
  validation_data = list(x_test, y_test)
)
```

```r
# Test
scores <- model_ca %>% evaluate(x_test, y_test, verbose = 0)

cat("Test loss:", scores[[1]], "\n",
    "Test accuracy:", scores[[2]], "\n")
```

```r
# Plot training history
plot(history)
```

```r
# Additional plot: confusion matrix

predictions <- model_ca %>% predict(x_test)%>% k_argmax()
y_true <- cifar10$test$y
confusion_matrix <- confusionMatrix(factor(as.vector(predictions)), factor(y_true))
print(confusion_matrix$table)
```

a) What do you think of the effects of data augmentation given the obtained results?

Problem 5: Univariate Time Series Classification with CNN

The following problem involves training a 1-dimensional Convolutional Neural Network (1D-CNN) model on a univariate time series dataset called Wafer, from the UCR Time Series Classification Archive. The Wafer dataset, formatted by R. Olszewski as part of his thesis at Carnegie Mellon University in 2001, contains inline process control measurements from various sensors during the processing of silicon wafers for semiconductor fabrication. Each sample within the dataset includes measurements from a single sensor during the processing of one wafer by one tool. The data is categorized into two classes: normal and abnormal. The Wafer dataset is a collection of 7164 time series samples, each with a length of 152. The dataset is divided into 1000 training samples and 6164 testing samples with the two different classes.

Convolutional Neural Networks (CNNs) have been proven effective not only at capturing spatial patterns in image data but also at detecting temporal patterns in time series data [4]. In this example, we will design our 1D-CNN model architecture to learn patterns and features from the univariate time series data to perform a time series classification. The model will consist of several 1D convolutional layers, followed by pooling layers to reduce dimensionality and extract relevant features and fully connected (dense) layers for classification.

[4] Wang, Zhiguang, Weizhong Yan, and Tim Oates. "Time series classification from scratch with deep neural networks: A strong baseline." 2017 International joint conference on neural networks (IJCNN). IEEE, 2017.

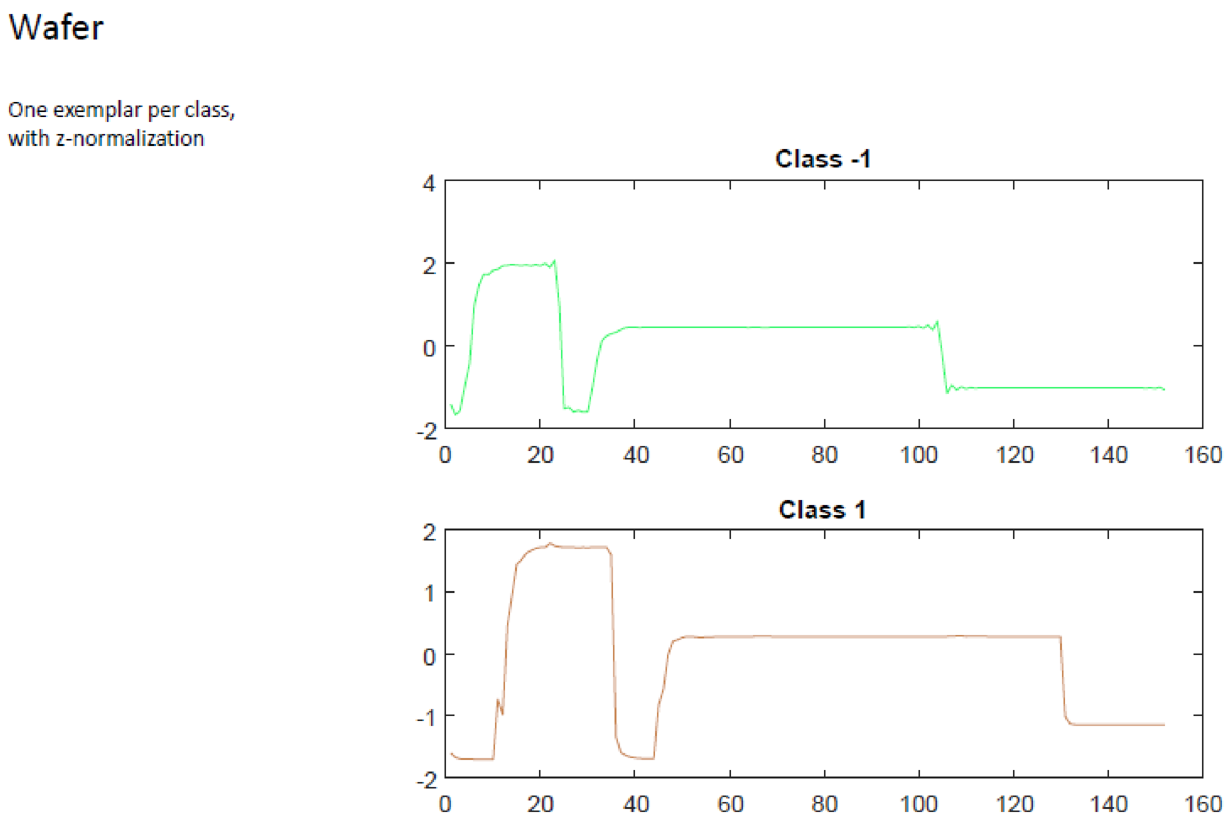

Figure 5: Fig. Examples of the samples from the Wafer dataset with respect to different classes.)

```r
# load the Wafer dataset
train <- read.delim("dataset/Wafer/Wafer_TRAIN.tsv", header = FALSE, sep = "\t")
test <- read.delim("dataset/Wafer/Wafer_TEST.tsv", header = FALSE, sep = "\t")

# the first column in `train` and `test` contains label info.
# therefore we separate them into `x` and `y`.
x_train <- train[,2:dim(train)[2]]
```

```
y_train <- clip(train[,1], 0, 1) #- 1
y_train <- to_categorical(y_train)
x_test <- test[,2:dim(test)[2]]
y_test <- clip(test[,1], 0, 1) #- 1
y_test <- to_categorical(y_test)

# create a channel dimension so that `x` has dimension of (batch, channel, length)
x_train <- array(as.matrix(x_train), dim = c(nrow(x_train), ncol(x_train), 1))
x_test <- array(as.matrix(x_test), dim = c(nrow(x_test), ncol(x_test), 1))

# preprocess
# The provided dataset has already been preprocessed, therefore no need for it.
```

1. Load and preprocess data

a) Fill in the missing parts in the following steps and run the model.

```
# Define the model
model_1dc <- keras_model_sequential() %>%
    layer_conv_1d(filters = .., kernel_size = .., activation = "..", input_shape = c(dim(x_train)[2], 1
    layer_max_pooling_1d(pool_size = 2) %>%
    layer_conv_1d(filters = .., kernel_size = .., activation = "..") %>%
    layer_max_pooling_1d(pool_size = 2) %>%
    layer_conv_1d(filters = .., kernel_size = .., activation = "..") %>%
    layer_max_pooling_1d(pool_size = 2) %>%
    layer_flatten() %>%
    layer_dense(units = 2, activation = "softmax")

summary(model_1dc)
```

2. Define the model We're building a CNN model that has filter sizes of {16, 32, 64} and kernel sizes of {8, 5, 3} with the relu activation function for all convolutional layers. Fill in the blanks accordingly.

```
model_1dc %>% compile(
  loss = "categorical_crossentropy",
  optimizer = optimizer_adam(learning_rate = 0.001),
  metrics = c("accuracy")
)
```

3. Compile

```
history <- model_1dc %>% fit(
  x_train, y_train,
  epochs = 100,
```

```r
  batch_size = 64,
  validation_data = list(x_test, y_test)
)
```

```r
# Test
scores <- model_1dc %>% evaluate(x_test, y_test, verbose = 0)

cat("Test loss:", scores[[1]], "\n",
    "Test accuracy:", scores[[2]], "\n")
```

```r
# Plot training history
plot(history)
```

4. Train the model

b) Fit a logistic regression model and compare its performance (i.e., accuracy) to that of the CNN model.

```r
# dataset
x_train <- train[,2:dim(train)[2]]
y_train <- clip(train[,1], 0, 1)
x_test <- test[,2:dim(test)[2]]
y_test <- clip(test[,1], 0, 1)

# Fit a linear regression model
logit_reg <- glm(..., data = as.data.frame(cbind(x_train, y_train)), family = "...")  # fill in

# Make predictions on the test set
predictions <- predict(logit_reg, newdata = x_test, type = "response")
predictions <- as.integer(predictions > 0.5)  # cutoff = 0.5
predictions <- as.factor(predictions)

result <- confusionMatrix(predictions, as.factor(y_test))

cat("=== [1D CNN] === \n", "Test accuracy:", scores[[2]],
    "\n",
    "============================ \n\n",
    "=== [Logistic Regression] === \n",
    "Test accuracy:", result$overall["Accuracy"], "\n",
    "==============================\n\n")
```

Comparison to a Logistic Regression Model