STAT 471: Homework 5

Ashley Clarke

Due: December 8, 2021 at 11:59pm

Contents

Instructions

Setup

Pull the latest version of this assignment from Github and set your working directory to `stat-471-fall-2021/homework/homework-5`. Consult the getting started guide if you need to brush up on `R` or `Git`.

Collaboration

The collaboration policy is as stated on the Syllabus:

> "Students are permitted to work together on homework assignments, but solutions must be written up and submitted individually. Students must disclose any sources of assistance they received; furthermore, they are prohibited from verbatim copying from any source and from consulting solutions to problems that may be available online and/or from past iterations of the course."

In accordance with this policy,

Please list anyone you discussed this homework with: -Zach Bradlow and Sarah Hu

Please list what external references you consulted (e.g. articles, books, or websites): - https://towardsdatascience.com/understanding-and-calculating-the-number-of-parameters-in-convolution-neural-networks-cnns-fc88790d530d

Writeup

Use this document as a starting point for your writeup, adding your solutions after "**Solution**". Add your R code using code chunks and add your text answers using **bold text**. Consult the preparing reports guide for guidance on compilation, creation of figures and tables, and presentation quality.

Programming

The `tidyverse` paradigm for data wrangling, manipulation, and visualization is strongly encouraged, but points will not be deducted for using base `R`.

We'll need to use the following R packages:

```
library(keras)          # to train neural networks
library(kableExtra)     # to print tables
library(cowplot)        # to print side-by-side plots
library(tidyverse)      # tidyverse
```

We'll also need the deep learning helper functions written for STAT 471:

```
source("../../functions/deep_learning_helpers.R")
```

Grading

The point value for each problem sub-part is indicated. Additionally, the presentation quality of the solution for each problem (as exemplified by the guidelines in Section 3 of the preparing reports guide will be evaluated on a per-problem basis (e.g. in this homework, there are three problems).

Submission

Compile your writeup to PDF and submit to Gradescope.

Fashion MNIST Data

In this homework, we will analyze the Fashion MNIST data, which is like MNIST but with clothing items rather than handwritten digits. There are ten classes, as listed in Table 1.

Table 1: The ten classes in the Fashion MNIST data.

Index	Name
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

The code provided below loads the data, and prepares it for modeling with `keras`.

```r
# load the data
fashion_mnist <- dataset_fashion_mnist()
```

```
## Loaded Tensorflow version 2.4.1
```

```r
# extract information about the images
num_classes = nrow(class_names)                    # number of image classes
num_train_images = dim(fashion_mnist$train$x)[1]   # number of training images
num_test_images = dim(fashion_mnist$test$x)[1]     # number of test images
img_rows <- dim(fashion_mnist$train$x)[2]          # rows per image
img_cols <- dim(fashion_mnist$train$x)[3]          # columns per image
num_pixels = img_rows*img_cols                     # pixels per image
max_intensity = 255                                # max pixel intensity

# normalize and reshape the images
x_train <- array_reshape(fashion_mnist$train$x/max_intensity,
                         c(num_train_images, img_rows, img_cols, 1))
x_test <- array_reshape(fashion_mnist$test$x/max_intensity,
                        c(num_test_images, img_rows, img_cols, 1))

# extract the responses from the training and test data
g_train <- fashion_mnist$train$y
g_test <- fashion_mnist$test$y

# recode response labels using "one-hot" representation
y_train <- to_categorical(g_train, num_classes)
y_test <- to_categorical(g_test, num_classes)
```

1 Data exploration

i. How many observations in each class are there in the training data? (Kable output optional.)

```r
observations <- y_train %>%
  colSums(na.rm = FALSE, dims = 1)

tibble("class" = class_names$name, observations) %>%
  kable(format = "latex", row.names = NA,
        booktabs = TRUE, digits = 2, caption = "Observations in Each Class") %>%
  kable_styling(position = "center")
```

ii. Plot the first six training images in a 2×3 grid, each image titled with its class name from the second column of Table 1.

```r
p1 = plot_grayscale(x_train[1,,,], g_train[1], class_names)
p2 = plot_grayscale(x_train[2,,,], g_train[2], class_names)
p3 = plot_grayscale(x_train[3,,,], g_train[3], class_names)
p4 = plot_grayscale(x_train[4,,,], g_train[4], class_names)
p5 = plot_grayscale(x_train[5,,,], g_train[5], class_names)
p6 = plot_grayscale(x_train[6,,,], g_train[6], class_names)

plot_grid(p1, p2, p3, p4, p5, p6, nrow = 2)
```

Table 2: Observations in Each Class

class	observations
T-shirt/top	6000
Trouser	6000
Pullover	6000
Dress	6000
Coat	6000
Sandal	6000
Shirt	6000
Sneaker	6000
Bag	6000
Ankle boot	6000

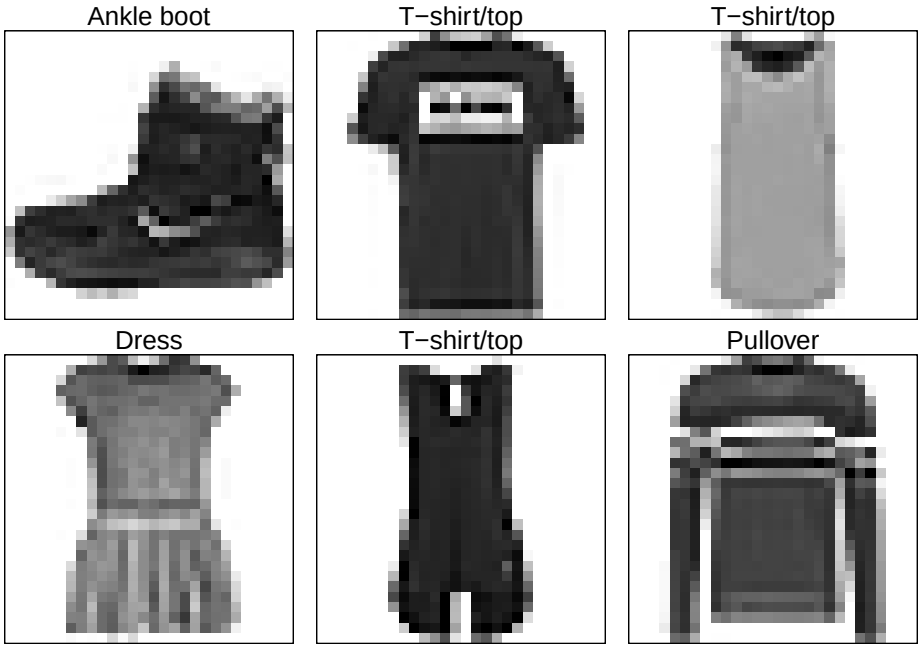

Figure 1: First six training images labeled by their class name

iii. Comment on the extent to which you (a human) would have been able to successfully classify the observations plotted in part ii. Would you have had any trouble? If so, with which observations? **Though the images are a bit blurry, I believe I would be able to classify most of the observations successfully. Howver, I would have had trouble determining if image 3 and 5 were dresses or t-shirts/tops**

iv. What is the human performance on this classification task? You can find it at the Fashion MNIST webpage linked above by searching for "human performance." **Human performance on this task is equal to 83.5% accuracy**

2 Model training

2.1 Multi-class logistic regression

i. Define a `keras_model_sequential` object called `model_lr` for multi-class logistic regression, and compile it using the `categorical_crossentropy` loss, the `adam` optimizer, and the `accuracy` metric.

```
model_lr <- keras_model_sequential() %>%
  layer_flatten(input_shape =                    # flatten during model-building
                  c(img_rows, img_cols, 1)) %>%
  layer_dense(units = num_classes,               # number of outputs
              activation = "softmax")
#compile the model
model_lr %>%
  compile(loss = "categorical_crossentropy",     # which loss to use
          optimizer = optimizer_adam(),          # how to optimize the loss
          metrics = c("accuracy"))               # how to evaluate the fit
```

ii. Print the summary of the model. How many total parameters are there? How does this number follow from the architecture of this simple neural network?

```
summary(model_lr)
```

```
## Model: "sequential"
## _________________________________________________________________________
## Layer (type)                 Output Shape              Param #
## =========================================================================
## flatten (Flatten)            (None, 784)               0
## _________________________________________________________________________
## dense (Dense)                (None, 10)                7850
## =========================================================================
## Total params: 7,850
## Trainable params: 7,850
## Non-trainable params: 0
## _________________________________________________________________________
```

Along with intercepts (referred to as biases in the deep-learning community) this network has 7850 parameters (referred to as weights). The input layer has p = 784 units and the output layer has 10 units. 784 is equal to number of image rows times the number of image columns. The output layer produces 10 responses instead of one. Thus, the first step is to compute ten different linear models similar to our single model. The matrix stores 10 x 784 elements for the output layer. Also, There are 785 rather than 784 because we must account for the intercept or bias term. Therefore, there are 7850 parameters

iii. Train the model for 10 epochs, using a batch size of 128, and a validation split of 20%. Save the model to `model_lr.h5` and its history to `model_lr_hist.RDS`, and then set this code chunk to `eval = FALSE` to avoid recomputation. How many total stochastic gradient steps were taken while training this model, and how did you arrive at this number? Based on the output printed during training, roughly how many milliseconds did each stochastic gradient step take?

```
history = model_lr %>%
  fit(x_train,                  # supply training features
      y_train,                  # supply training responses
      epochs = 10,               # an epoch is a gradient step
      batch_size = 128,         # we will learn about batches in Lecture 2
      validation_split = 0.2)   # use 20% of the training data for validation
```

```
# save model
save_model_hdf5(model_lr, "model_lr.h5")
# save history
saveRDS(model_lr$history$history, "model_lr_hist.RDS")
```

DO NOT KNOW According to the output, each stochastic gradient step took roughly 2 milliseconds

iv. Load the model and its history from the files saved in part iii. Create a plot of the training history. Based on the shape of the validation loss curve, has any overfitting occurred during the first 10 epochs?

```
# load model
model_lr = load_model_hdf5("model_lr.h5")
# load history
model_lr_hist = readRDS("model_lr_hist.RDS")
# plot training history
plot_model_history(model_lr_hist)
```

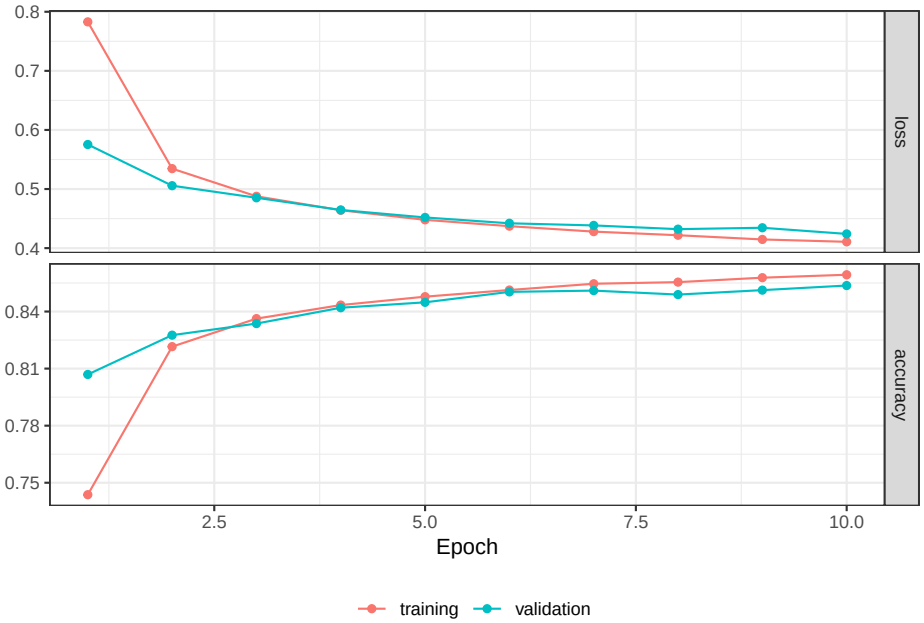

Figure 2: Validation loss curve based on training hisotry

No, overfitting does not appear to occur during first 10 epochs because training and validation loss and accuracy appear very similar

2.2 Fully connected neural network

i. Define a `keras_model_sequential` object called `model_nn` for a fully connected neural network with three hidden layers with 256, 128, and 64 units, `relu` activations, and dropout proportions 0.4, 0.3, and 0.2, respectively. Compile it using the `categorical_crossentropy` loss, the `rmsprop` optimizer, and the `accuracy` metric.

```
model_nn = keras_model_sequential() %>%
  layer_flatten(input_shape = c(img_rows, img_cols, 1)) %>%
  layer_dense(units = 256, activation = "relu") %>%
```

```
  layer_dropout(rate = 0.4) %>%
  layer_dense(units = 128, activation = "relu") %>%
  layer_dropout(rate = 0.3) %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dropout(rate = 0.2) %>%
  layer_dense(units = 10, activation = "softmax") %>%
  compile(loss = "categorical_crossentropy",
          optimizer = optimizer_rmsprop(),
          metrics = c("accuracy")
          )
```

ii. Print the summary of the model. How many total parameters are there? How many parameters correspond to the connections between the second and third hidden layers? How does this number follow from the architecture of the neural network?

```
summary(model_nn)
```

```
## Model: "sequential_1"
## _________________________________________________________________________
## Layer (type)                 Output Shape              Param #
## =========================================================================
## flatten_1 (Flatten)          (None, 784)               0
## _________________________________________________________________________
## dense_4 (Dense)              (None, 256)               200960
## _________________________________________________________________________
## dropout_2 (Dropout)          (None, 256)               0
## _________________________________________________________________________
## dense_3 (Dense)              (None, 128)               32896
## _________________________________________________________________________
## dropout_1 (Dropout)          (None, 128)               0
## _________________________________________________________________________
## dense_2 (Dense)              (None, 64)                8256
## _________________________________________________________________________
## dropout (Dropout)            (None, 64)                0
## _________________________________________________________________________
## dense_1 (Dense)              (None, 10)                650
## =========================================================================
## Total params: 242,762
## Trainable params: 242,762
## Non-trainable params: 0
## _________________________________________________________________________
```

There are 242,762 total parameters The input layer has p = 784 units, the three hidden layers K1 = 256, K2 = 128 units, and K3 = 56 respectively, and the output layer 10 units. 784 is equal to number of image rows time the number of image columns. Along with intercepts (referred to as biases in the deep-learning community) this network has 242,762 parameters (referred to as weights). The first layer has 785 x 256 elements. There are 785 rather than 784 because we must account for the intercept or bias term. The second layer has 257 x 128 elements becuase the first layer feeds into the second layer via the matrix weights. Lastly, the third layer has 129 x 64 elements. The output layer produces 10 responses instead of one. Thus, the first step is to compute ten different linear models similar to our single model. The matrix stores 10 x 65 elements for the output layer. Added together, all of the layers produce 200,960 + 32,896 + 8256 + 650 parameters, which is equal to 242,762

iii. Train the model using 15 epochs, a batch size of 128, and a validation split of 0.2. Save the model to

`model_nn.h5` and its history to `model_nn_hist.RDS`, and then set this code chunk to `eval = FALSE` to avoid recomputation. Based on the output printed during training, roughly how many milliseconds did each stochastic gradient step take?

```
history = model_nn %>%
  fit(x_train,                    # supply training features
      y_train,                    # supply training responses
      epochs = 15,                # an epoch is a gradient step
      batch_size = 128,           # we will learn about batches in Lecture 2
      validation_split = 0.2)     # use 20% of the training data for validation

# save model
save_model_hdf5(model_nn, "model_nn.h5")
# save history
saveRDS(model_nn$history$history, "model_nn_hist.RD")
```

According to the output, each stochastic gradient step took roughly 2 milliseconds

iv. Load the model and its history from the files saved in part iii. Create a plot of the training history.

```
# load model
model_nn = load_model_hdf5("model_nn.h5")
# load history
model_nn_hist = readRDS("model_nn_hist.RD")
# plot training history
plot_model_history(model_nn_hist)
```

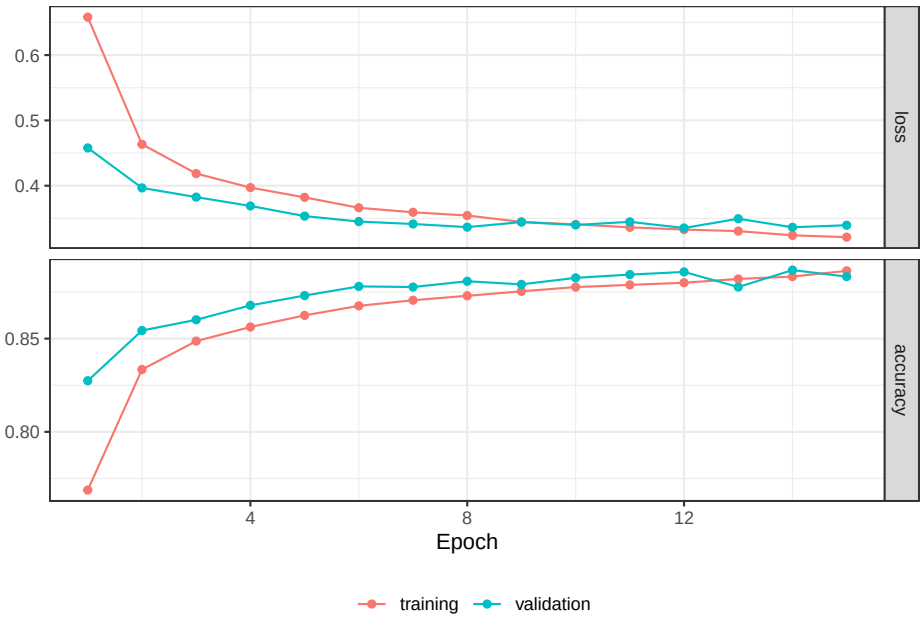

Figure 3: Validation loss curve based on training history for fully connected neural network

2.3 Convolutional neural network

i. Define a `keras_model_sequential` object called `model_cnn` for a convolutional neural network with a convolutional layer with 32 3×3 filters, followed by a convolutional layer with 64 3×3 filters, followed by a max-pooling step with 2×2 pool size with 25% dropout, followed by a fully-connected layer

with 128 units and 50% dropout, followed by a softmax output layer. All layers except the output layer should have `relu` activations. Compile the model using the `categorical_crossentropy` loss, the `adadelta` optimizer, and the `accuracy` metric.

```
model_cnn <- keras_model_sequential() %>%
  layer_conv_2d(filters = 64, kernel_size = c(3,3), activation = 'relu',
                input_shape = c(img_rows, img_cols, 1)) %>%
  layer_conv_2d(filters = 64, kernel_size = c(3,3), activation = 'relu') %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_dropout(rate = 0.25) %>%
  layer_flatten() %>%
  layer_dense(units = 128, activation = 'relu') %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = num_classes, activation = 'softmax')

model_cnn %>% compile(loss = "categorical_crossentropy",
                      optimizer = optimizer_adadelta(),
                      metrics = c("accuracy")
                      )
```

ii. Print the summary of the model. How many total parameters are there? How many parameters correspond to the connections between the first and second convolutional layers? How does this number follow from the architecture of the neural network?

```
summary(model_cnn)
```

```
## Model: "sequential_2"
## _________________________________________________________________________
## Layer (type)                 Output Shape              Param #
## =========================================================================
## conv2d_1 (Conv2D)            (None, 26, 26, 64)        640
## _________________________________________________________________________
## conv2d (Conv2D)              (None, 24, 24, 64)        36928
## _________________________________________________________________________
## max_pooling2d (MaxPooling2D) (None, 12, 12, 64)        0
## _________________________________________________________________________
## dropout_4 (Dropout)          (None, 12, 12, 64)        0
## _________________________________________________________________________
## flatten_2 (Flatten)          (None, 9216)              0
## _________________________________________________________________________
## dense_6 (Dense)              (None, 128)               1179776
## _________________________________________________________________________
## dropout_3 (Dropout)          (None, 128)               0
## _________________________________________________________________________
## dense_5 (Dense)              (None, 10)                1290
## =========================================================================
## Total params: 1,218,634
## Trainable params: 1,218,634
## Non-trainable params: 0
## _________________________________________________________________________
```

There are 1,218,634 paramenters. Input layer has nothing to learn, at it's core, what it does is just provide the input image's shape. Thus, number of parameters = 0. Parameters in the CONV layer is: ((shape of width of filter x shape of height filter x number of filters in the previous layer+1) x number of filters) = (((3x3x3)+1) x 64) = 640. Paramenters in the next CONV layer are equal to ((3x3x64)+1)x64 = 36928. For the pooling layer, there are no

learnable parameters because all it does is calculate a specific number. Therefore, there is no backprop learning involved. The number of parameters = 0. Parameters in the fully connected layer is caluclated by ((current layer c x previous layer p)+1 x c) = (128 x 9216) + 1 x 128 = 1179776. The final layer is calculated by ((current layer c x previous layer p)+1 x c). Since this is the last layer, (10 x 128) + 1 x 10 = 1290. When all of the parameters are summed together, the total is equal to 1,218,634

iii. Train the model using 8 epochs, a batch size of 128, and a validation split of 0.2. Save the model to `model_cnn.h5` and its history to `model_cnn_hist.RDS`, and then set this code chunk to `eval = FALSE` to avoid recomputation. Based on the output printed during training, roughly how many milliseconds did each stochastic gradient step take?

```
model_cnn %>%
  fit(x_train,                  # supply training features
      y_train,                  # supply training responses
      epochs = 8,               # 8 epoch cycles
      batch_size = 128,         # 128 batch size
      validation_split = 0.2)   # use 20% of the training data for validation

# save model
save_model_hdf5(model_cnn, "model_cnn.h5")
# save history
saveRDS(model_cnn$history$history, "model_cnn_hist.RDS")
```

Each stochastic gradient steep took roughly 155 ms

iv. Load the model and its history from the files saved in part iii. Create a plot of the training history.

```
# load model
model_cnn = load_model_hdf5("model_cnn.h5")
# load history
model_cnn_hist = readRDS("model_cnn_hist.RDS")
# plot training history
plot_model_history(model_cnn_hist)
```

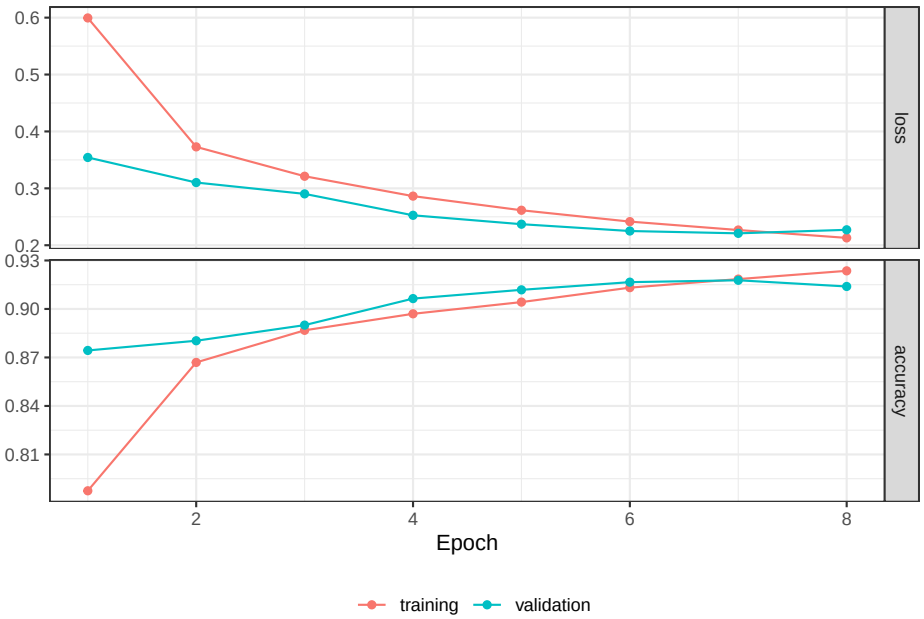

Figure 4: Validation loss curve based on training history for onvolutional neural network

3 Evaluation

i. Evaluate the test accuracy for each of the three trained neural network models. Output this information in a table, along with the number of layers, number of parameters, and milliseconds per stochastic gradient descent step. Also include a row in the table for human performance. Compare and contrast the three neural networks and human performance based on this table.

```
# LR: get predicted classes
lr_predictions = model_lr %>% predict(x_test) %>% k_argmax() %>% as.integer()
# NN: get predicted classes
nn_predictions = model_nn %>% predict(x_test) %>% k_argmax() %>% as.integer()
# CNN: get predicted classes
cnn_predictions = model_cnn %>% predict(x_test) %>% k_argmax() %>% as.integer()

# LR: test accuracy
lr_accuracy = mean(lr_predictions == g_test)
# : test accuracy
nn_accuracy = mean(nn_predictions   == g_test)
# CNN: test accuracy
cnn_accuracy = mean(cnn_predictions == g_test)

tibble(lr_accuracy, nn_accuracy, cnn_accuracy) %>%
  kable(format = "latex", row.names = NA,
                        booktabs = TRUE,
                        digits = 3,
                        col.names = c("Multi-class Logistic Regression",
                                      "Fully Connected Neural Net",
                                      "Convolutional Neural Net"),
                        caption = "Test Accuracy Across Three Trained Neural Network Models") %>%
  kable_styling(position = "center") %>%
```

```
  kable_styling(latex_options = "HOLD_position")
```

Table 3: Test Accuracy Across Three Trained Neural Network Models

Multi-class Logistic Regression	Fully Connected Neural Net	Convolutional Neural Net
0.842	0.875	0.912

ii. Plot confusion matrices for each of the three methods. For each method, what class gets misclassified most frequently? What is most frequent wrong label for this class?

```
#LR confusion matrix
lr_confusion_matrix = plot_confusion_matrix(predicted_responses = lr_predictions,
                      actual_response = g_test)
lr_confusion_matrix = lr_confusion_matrix +
  ggtitle("Multi-class Logistic Regression") +
  theme(plot.title = element_text(face = "bold", size = 8))

#NN confusion matrix
nn_confusion_matrix = plot_confusion_matrix(predicted_responses = nn_predictions,
                      actual_response = g_test)
nn_confusion_matrix = nn_confusion_matrix +
  ggtitle("Fully Connected Neural Net")  +
  theme(plot.title = element_text(face = "bold", size = 8))

#CNN confusion matrix
cnn_confusion_matrix = plot_confusion_matrix(predicted_responses = cnn_predictions,
                      actual_response = g_test)
cnn_confusion_matrix = cnn_confusion_matrix +
  ggtitle("Convolutional Neural Net")  +
  theme(plot.title = element_text(face = "bold", size = 9))

plot_grid(lr_confusion_matrix, nn_confusion_matrix, cnn_confusion_matrix, nrow = 1)
```

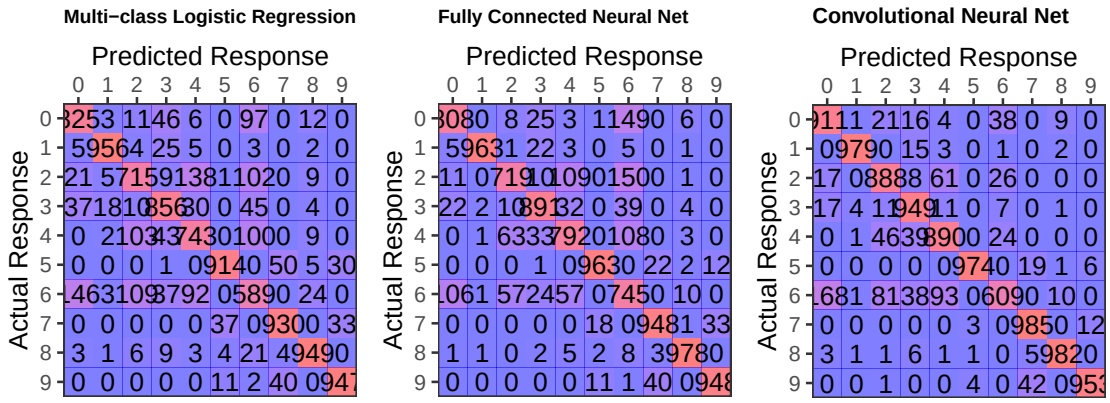

Figure 5: Confusion Matrices Across Three Trained Neural Network Models: Multi-class Logistic Regression, Fully Connected Neural Net, and Convolutional Neural Net
(#fig:confusion matrix)

iii. Consider CNN's most frequently misclassified class. What are the three most common incorrect classifications for this class? Extract one image representing each of these three type of misclassifications, and plot these side by side (titled with their predicted labels). Would you have gotten these right?

```
top_misclass =  cnn_confusion_matrix$data %>%
  filter(!(predicted_response == actual_response))  %>%
  group_by(actual_response) %>%
  summarise(num_misclass = sum(n)) %>%
  arrange(desc(num_misclass)) %>%
  head(1) %>% select(actual_response) %>% pull() %>% as.integer()

misclass =  cnn_confusion_matrix$data %>%
  filter(!(predicted_response == actual_response), actual_response == top_misclass )  %>%
  arrange(desc(n)) %>% select(predicted_response) %>% head(3) %>% pull() %>% as.integer()

#Plot highest misclass erorr
misclass_1 = which(cnn_predictions == misclass[1] & g_test == top_misclass)
idx_1 = misclass_1[1]
p1 <-plot_grayscale(x_test[idx_1,,,], misclass[1], class_names)
#Plot second highest misclass erorr
misclass_2 = which(cnn_predictions == misclass[2] & g_test == top_misclass)
idx_2 = misclass_2[1]
p2 <-plot_grayscale(x_test[idx_2,,,], misclass[2], class_names)
#Plot third highest misclass erorr
misclass_3 = which(cnn_predictions == misclass[3] & g_test == top_misclass)
```

```
idx_3 = misclass_3[1]
p3 <-plot_grayscale(x_test[idx_3,,,], misclass[3], class_names)

plot_grid(p1, p2, p3, nrow = 1)
```

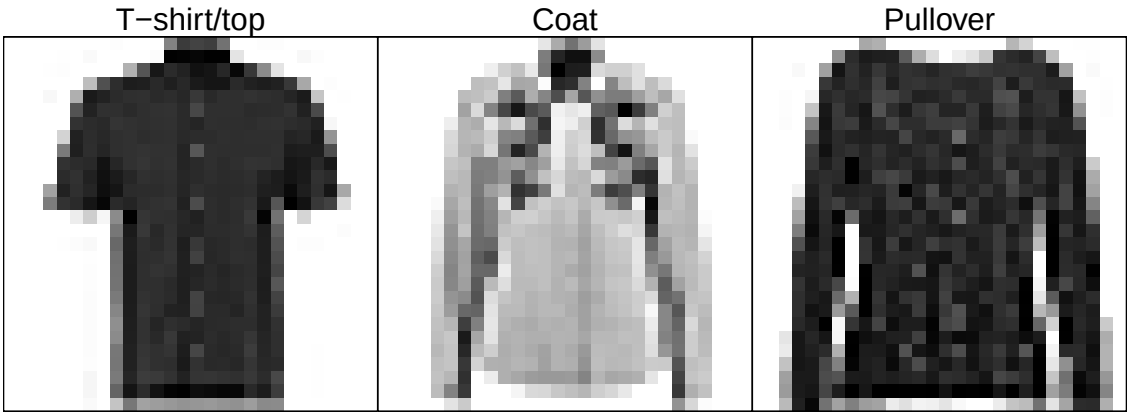

Figure 6: Three most common incorrect classifications where the image was incorrectly classifed when it should have been classifed as a shirt

CNN's current most misclassifed class is class 6, which correspodns to the "shirt" class. Most commonly the model predicted that shirts (6) are actually "T-shirt/top"(0), which makes sense because these items look very similar even to the human eye. The three most common misclassifications are predicting "T-shirt/top"(0) when it is actually a "Shirt" (6), predicting "Coat" (4), when the image is a shirt, and predicting "Pullover" (2) when the image is a "Shirt" (6). I see how all of the images could be classifed as what they are predicted to be. I also would have classifed image 1 as a top and image 2 as a coat. However, I would have classifed image 3 correctly.