Template for the final exam 2022-11-30

Machine Learning ST5401, 7.5 credits

0008-BUB

Loading packages

```
cat("\014")
```

```
rm(list = ls())
graphics.off()
suppressMessages(library(tree))
suppressMessages(library(caret))
suppressMessages(library(glmnet))
# Set your path below. In Windows, \\ is required instead of \.
# Example "C:\\Users\\SUA32538\\Downloads"
setwd("C:/Users/aljo0159/Downloads")
set.seed(1234) # set the seed for reproducibility
```

Problem 1

a) /0p

b)

The decaying learning rate is necessary for the stochastic gradient descent algorithm to converge, as opposed to a constant learning rate. With a constant learning rate, the algorithm would not converge but rather go back and forth. The reason is that in stochastic gradient descent, we are using an estimate of the gradient. Because we are using an estimate, we do not know if the estimated gradient is actually zero in the function which we want to minimize. The gradient is stochastic, i.e. random, and the idea is that it on average leads us in the right direction, and with a decaying learning rate we approach the optimum.

/2p

c)

Unsupervised Gaussian mixture models are used when data has no labels. This implies problems with initializing the EM-algorithm. In a semi-supervised setting, one would use the information of the labels at hand to initialize the EM-algorithm. In an unsupervised setting, the starting values will be set arbitrarily instead of using the labelled data. The label-switching problem comes from the fact that all labels/classes have the same likelihood and points can therefore change between labels/classes (i.e. switch labels).

In this setting, the result of the algorithm is based on the similarity of the points and gives which points/observations are most likely to come from the same class-conditional distribution, and depends on how we initialize the algorithm.

/2p

d)

I'll start by just explaining the bias-variance tradeoff, which can be understood as:

Suppose we obtain a sample of a dataset and use it to construct a very simple model, which accounts for some of the features present in the data. An example could be a dataset with measurements on grades on this exam, study time, previous grades in university, points on Högskoleprovet etc. Let's say we train a simple model and predict grades on this exam based on new data on points on Högskoleprovet. Then, we obtain a new sample and train another model (to the different sample). Because the model only predicts grades based on points on Högskoleprovet, we do not expect the models to differ too much (as we have random samples). Because the two models are simple and only account for one of the features in the data and because we assume that the

samples are representative of the whole data, the models will have low variance. However, the bias will be high as the models are simple (more factors should be considered when predicting the grade).

If we increase the model complexity by taking into account all the features in our data, and do the same thing again, where we fit multiple models to different samples of the data, the models will instead have high variance. This is because the complex models are accounting for all the features in the sample, and adjusting the model very much to the sample itself, and not the data generating process. On the other side, the models will have low bias because they fit the data well.

The goal is to find a model in the “sweet spot” between a simple model and a complex model.

Bagging is based on bootstrapping and is a technique used to reduce the variance while not significantly increasing the bias. It works by fitting several complex models from bootstrapped samples of the data, and then computes the model as the average of these models. In doing so, we have accounted for the features present in our data (by fitting a complex model which allows us to account for patterns in the data). When doing random forests, we do this - we fit several tree models from bootstrapped samples of our data.

/2p

e)

If we fit an additive linear regression model, while the true data generating process is that we have an interaction between x_1 and x_2, we are underfitting the training data. This is because we are not accounting for the true patterns in our data. The additive linear regression model cannot capture the interaction between x_1 and x_2, and thus, we are underfitting. A Gaussian process regression can mitigate this, as Gaussian process regression determines a function f which is used to describe the data as $y_i = f(x_i) + \varepsilon$ and can handle interactions.

/2p

8

Problem 2

```
load("Japan_temperatures_training.RData")
load("Japan_temperatures_test.RData")
```

a)

To use L^2-regularization is R, I set $\alpha = 0$ in the cv.glmnet() function. To find the optimal λ, I use the lambda.min.

```
# A function which creates a matrix with polynomials for x
poly_matrix <- function(x, order) {
    x_mat <- cbind(1, x)
    if (order==1){return(x_mat)}
    for (k in 2:order) {
        x_mat <- cbind(x_mat, x^k)
    }
    return(x_mat)
}

# A sequence for k, the order of polynomials to try
order_seq <- seq(1, 8, 1)
RMSE_train <- NA
RMSE_test <- NA
lambda_opt <- NA

for (i in 1:length(order_seq)) {
  # Splitting the data into x and y
  x_train <- poly_matrix(Japan_temperatures_training$time, i)
  y_train <- Japan_temperatures_training$temperature

  x_test <- poly_matrix(Japan_temperatures_test$time, i)
  y_test <- Japan_temperatures_test$temperature

  # Ridge regression, which is optained by alpha = 0.
  # Cross-validation with 10 folds.
  cv_fit <- cv.glmnet(x_train, y_train, alpha = 0, nfolds = 10,
                      lambda = seq(0, 1, length.out = 100))
  # Saving the optimal lambda in a vector
  lambda_opt[i] <- cv_fit$lambda.min

  # Making predictions and calculating RMSE on the test data
  y_pred <- predict(cv_fit, newx = x_test, s = lambda_opt[i])
  RMSE_test[i] <- sqrt(sum((y_test - y_pred)^2) / length(y_test))
}

plot(1:8, RMSE_test, type = "l", xlab = "Polynomial order", ylab = "RMSE test")
```

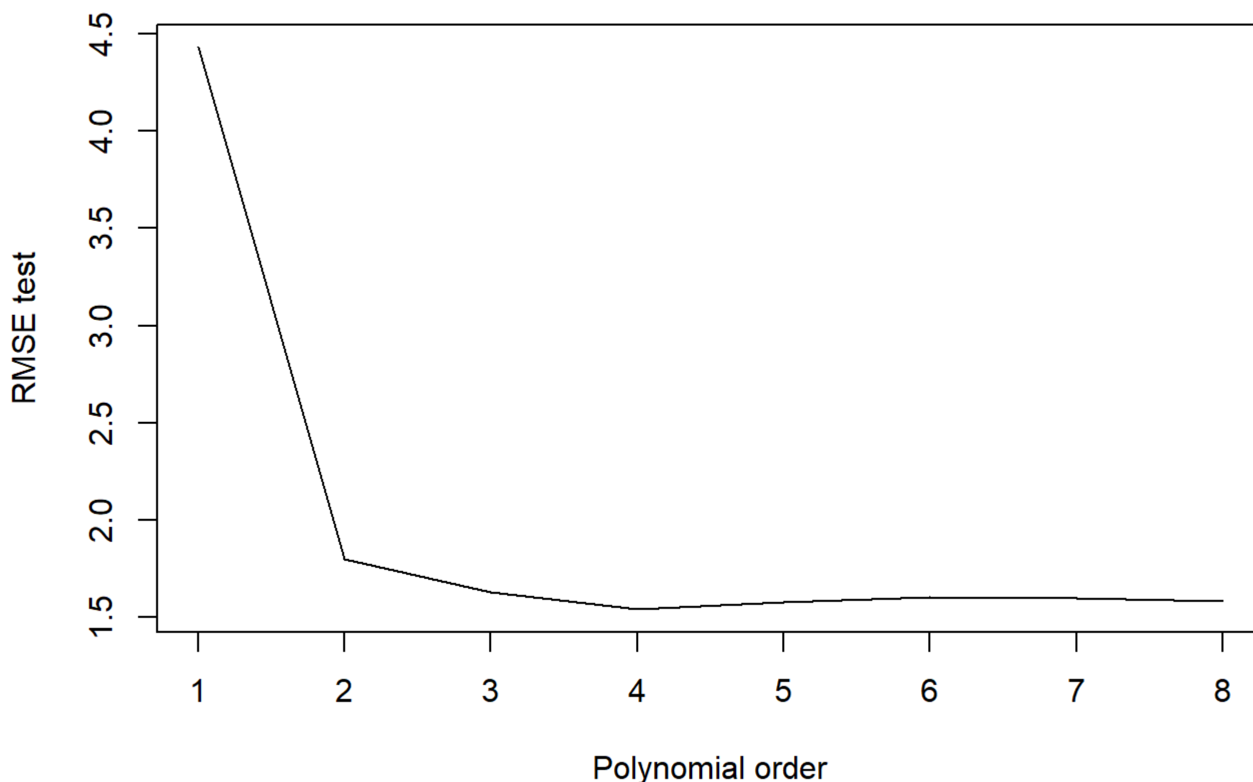

Figure 1. RMSE test for different polynomial orders

```
optimal_k <- which.min(RMSE_test)
lambda_opt_full <- lambda_opt[optimal_k]
```

```
## The polynomial that minimises the root mean squared error on the test data is order 4 whic
h has RMSE 1.540888
```

```
## The optimal value for lambda is 0
```

As can be seen from the output above and in Figure 1, the polynomial of order 4 is the one that minimizes the root mean squared error on the test data. The optimal lambda was 0, which means that no penalty is used.

/4p

b)

Ridge regression uses L^2 regularization to minimize the L^2-penalized residual sum of squares, i.e.

$$\underset{\beta}{argmin}(\mathbf{y}-\mathbf{X}\boldsymbol{\beta})^T(\mathbf{y}-\mathbf{X}\boldsymbol{\beta})+\lambda||\boldsymbol{\beta}||_2^2$$

The penalizing term increases the bias to reduce the variance and by doing so, increases the generalization capability. More specifically, λ is a tuning parameter which controls the strength of the penalty. If $\lambda = 0$, then the penalty has no effect (as the last term is just zero). In such a situation, ridge regression reduces to ordinary

/2p

least squares. With increased lambda, the impact of the shrinkage penalty grows and the ridge coefficients will thus decrease. This means that increasing lambda decreases the flexibility of the model. As decreased flexibility of the model means that the smoothness of the model increases, the bias will increase. In short, large values of lambda means higher bias and lower variance.

c)

```
# Fitting the model with k = 4
x_train <- poly_matrix(Japan_temperatures_training$time, optimal_k)
y_train <- Japan_temperatures_training$temperature

x_test <- poly_matrix(Japan_temperatures_test$time, optimal_k)
y_test <- Japan_temperatures_test$temperature

poly_fit <- glmnet(x_train, y_train, alpha = 0,
                   lambda = lambda_opt_full, intercept = FALSE)
```

```
## The estimated polynomial coefficients are 0 150.3233 -292.9831 262.8598 -102.4327
```

$$\hat{\beta}_0 = 0$$

$$\hat{\beta}_1 = 150.3233$$

$$\hat{\beta}_2 = -292.9831$$

$$\hat{\beta}_3 = 262.8598$$

$$\hat{\beta}_4 = -102.4327$$

/1p

(-1 wrong coefficient).

d)

```
x_grid <- seq(0, 1, length = 1000)
x_mat_grid <- poly_matrix(x_grid, 4)
beta_hat <- poly_fit$beta[, 1]
y_fit <- x_mat_grid[, 2:5] %*% beta_hat[2:5]

plot(Japan_temperatures_training$time,
     Japan_temperatures_training$temperature,
     pch = 16,
     cex = 0.5,
     xlab = "Time",
     ylab = "Temperature",
     col = "black",
     ylim = c(0, 30))
points(Japan_temperatures_test$time,
       Japan_temperatures_test$temperature,
       pch = 16,
       cex = 0.5,
       col = "blue")
lines(x_grid, y_fit, col = "red", lwd = 2)
legend(x = "bottomright",
       inset = .05,
       legend = c("Training data", "Test data", "Fit"),
       lty = c(NA, NA, 1),
       lwd = c(2, 2, 2),
       pch = c(16, 16, NA),
       col = c("black", "blue", "red"))
```

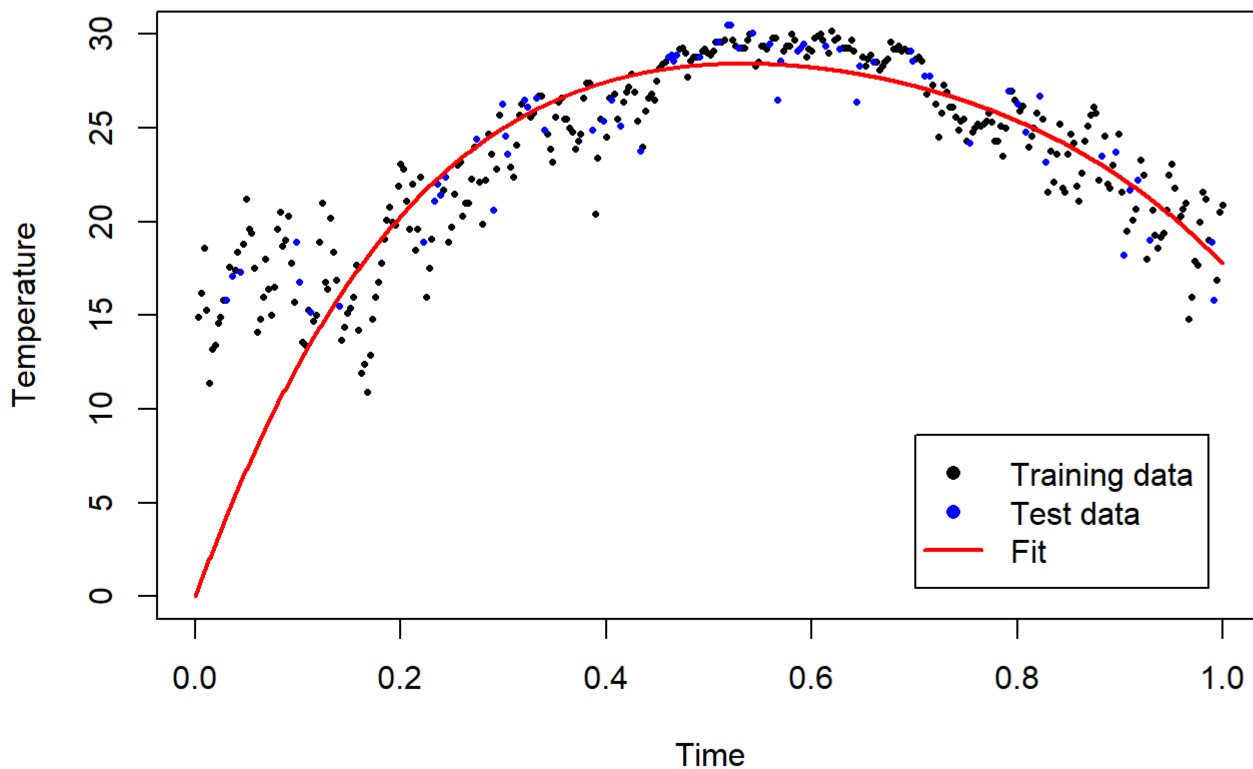

Figure 2. Fitted polynomial from (c).

From a visual inspection of the Figure 2, the fit okay, but not perfect. As we have observations for temperatures below 10, I will change the y-axis a little.

```
plot(Japan_temperatures_training$time,
     Japan_temperatures_training$temperature,
     pch = 16,
     cex = 0.5,
     xlab = "Time",
     ylab = "Temperature",
     col = "black")
points(Japan_temperatures_test$time,
       Japan_temperatures_test$temperature,
       pch = 16,
       cex = 0.5,
       col = "blue")
lines(x_grid, y_fit, col = "red", lwd = 2)
legend(x = "bottomright",
       inset = .05,
       legend = c("Training data", "Test data", "Fit"),
       lty = c(NA, NA, 1),
       lwd = c(2, 2, 2),
       pch = c(16, 16, NA),
       col = c("black", "blue", "red"))
```

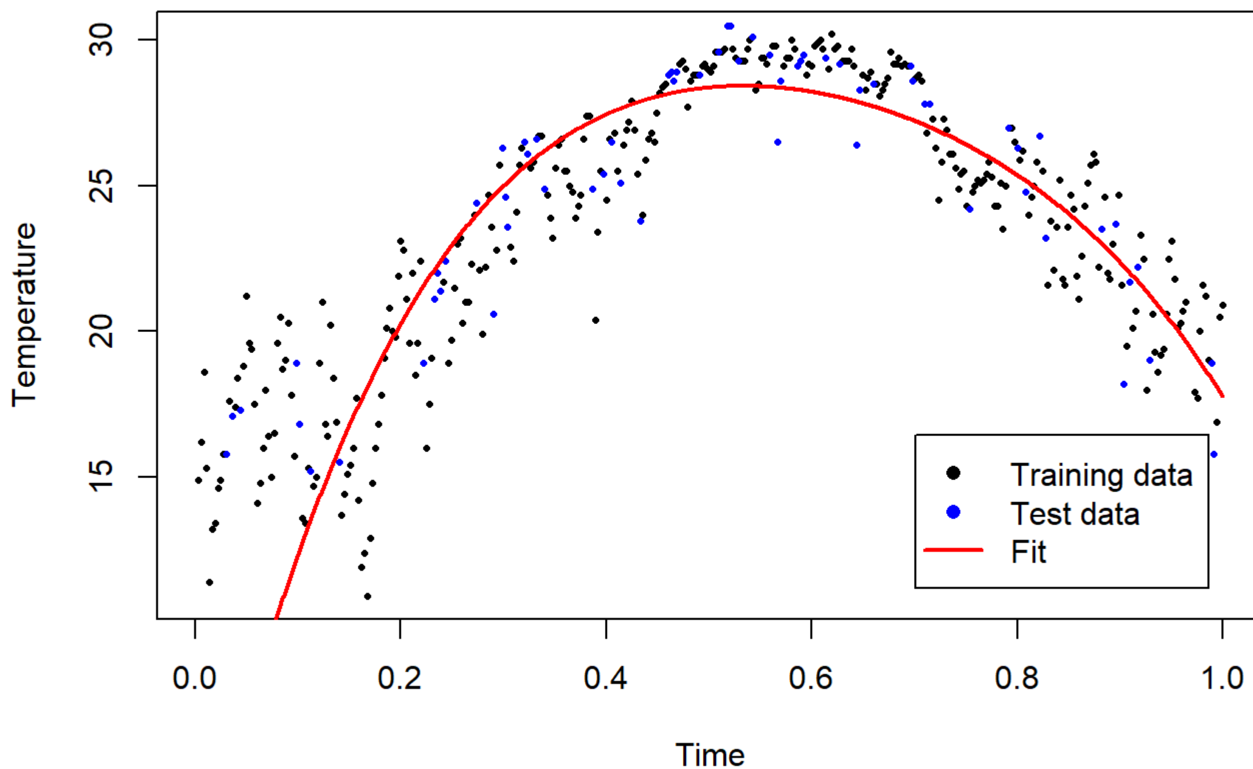

Figure 3. Fitted polynomial from (c).

Now, looking at Figure 3, I think one could argue that there is room for improvement. The model has been trained on the black dots. However, it looks like the prediction for temperature around time 0.6 should be higher, and that we have too steep of a fit for times between 0 and 0.4. I would say that we are underfitting, and to solve the issue, an alternative could be to use splines.

Problem 3

The code block.

```
load("Spam_training.RData")
load("Spam_test.RData")
```

a)

```
# Splitting the data and scaling the covariates
x_train <- scale(as.matrix(Spam_training[, 2:16]))
training <- as.data.frame(cbind(spam = Spam_training$spam, x_train))
training$spam <-ifelse(training$spam == 1, 0, 1) # changing from levels 1 and 2 to 0 and 1
training$spam <- factor(training$spam, levels = c(0, 1), labels = c("ham", "spam"))

x_test <- scale(as.matrix(Spam_test[, 2:16]))
test <- as.data.frame(cbind(spam = Spam_test$spam, x_test))
test$spam <-ifelse(test$spam == 1, 0, 1) # changing from levels 1 and 2 to 0 and 1
test$spam <- factor(test$spam, levels = c(0, 1), labels = c("ham", "spam"))

glm_fit <- glm(spam ~ ., data = training, family = binomial(link = "logit"))
```

```
## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
```

```
y_probs <- predict(glm_fit, newdata = test, type = "response")

threshold <- 0.5 # Predict Sold if yProbs>threshold
y_preds <- as.factor(y_probs > threshold)
levels(y_preds) <- c("ham", "spam")

# As the classifier is used to detect spam, spam is the positive outcome
comp_mat_glm <- confusionMatrix(y_preds, test$spam, positive = "spam")
comp_mat_glm$table
```

```
##           Reference
## Prediction ham spam
##       ham  898   79
##       spam  47  577
```

```
tp <- comp_mat_glm$table[2, 2]
fp <- comp_mat_glm$table[2, 1]
tn <- comp_mat_glm$table[1, 1]
fn <- comp_mat_glm$table[1, 2]

precision_glm <- tp / (tp + fp)
recall_glm <- tp / (tp + fn)
accuracy_glm <- (tn + tp) / sum(comp_mat_glm$table)
```

```
## The precision is 0.9246795 and the recall is 0.8795732
```

```
## The accuracy is 0.9212992
```

The precision is computed as $\frac{TP}{TP+FP}$ and thus refers to the proportion of true positives out of all predicted positives. That is, out of all emails which have been predicted to be spam, how many are actually spam. If the precision is too low in this context, this would mean that a lot of emails that are not spam end up in the spam-

inbox (false positives).

The recall is computed as $\frac{TP}{TP+FN}$ and thus refers to the proportion of true positives that are correctly predicted as positives. In this context, it answers to how many spam emails are correctly predicted to be spam emails. Again, we want recall to be as high as possible, as we want spam emails to end up in the spam filter. If the recall is low, then a lot of spam emails would end up in the regular mail (false negatives).

/5p

b)

```
class_tree <- tree(spam ~ ., data = training)
y_probs <- predict(class_tree, newdata = test)

threshold <- 0.5 # Predict Sold if yProbs>threshold
y_preds <- as.factor(y_probs[, 2] > threshold)
levels(y_preds) <- c("ham", "spam")

# As the classifier is used to detect spam, spam is the positive outcome
comp_mat_tree <- confusionMatrix(y_preds, test$spam, positive = "spam")
comp_mat_tree$table
```

```
##           Reference
## Prediction ham spam
##       ham  920  139
##       spam  25  517
```

```
tp <- comp_mat_tree$table[2, 2]
fp <- comp_mat_tree$table[2, 1]
tn <- comp_mat_tree$table[1, 1]
fn <- comp_mat_tree$table[1, 2]

precision_tree <- tp / (tp + fp)
recall_tree <- tp / (tp + fn)
accuracy_tree <- (tn + tp) / sum(comp_mat_tree$table)
```

```
## The precision is 0.9538745 and the recall is 0.7881098
```

```
## The accuracy is 0.897564
```

```
## Difference in precision: 0.02919505
```

```
## Difference in recall: -0.09146341
```

```
## Difference in accuracy: -0.02373517
```

As seen by the output above, the precision is higher for the classification tree by approximately 0.03 whereas

the recall for the classification tree is lower by 0.09, compared to the logistic regression. This means that the a larger proportion of spam emails are correctly predicted to be spam. That is, the false positives have been reduced. As the recall has decreased, this means that out of all the predicted spam, less emails are actually spam. Thus, more spam emails have ended up in the regular mail (false negatives have increased).

The accuracy has also decreased for the classification tree.

In terms of the problem at hand, one could argue that it is important that emails which are not spam do not end up in the spam-filter. Personally, I would think that the general opinion is that the occasional spam-email ending up in the regular email-inbox is more tolerable than normal emails ending up in the spam-inbox. Therefore, I would prefer the classification tree.

c)

The implication of the complaint is that the customer thinks that the rate of false negatives is too high (contrary to my previous discussion). To resolve the issue, the threshold needs to be decreased. This would reduce the amount of emails that are spam, but are not being classified as spam (the false negatives).

In doing so, we increase the number of false positives. This means that more emails that are not spam will be classified as spam. The drawbacks are thus that the customer will have to keep an eye on their spam-inbox, to make sure that no important emails get stuck there!

Problem 4

a)

The input $\mathbf{x}$ is a vector of dimension $p \cdot 1$. As we have $p = 15$ features, the dimensions are $15 \cdot 1$. That is, $\mathbf{x} = [x_1 \dots x_{15}]^T$. The output, $\Pr(y = 1|\mathbf{x})$ is just a number between 0 and 1, as it is the probability of spam, given the features.

b)

The first equation, $\mathbf{q}^{(1)} = h\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right)$, where $\mathbf{q}^{(1)}$ is a vector of dimension $12 \cdot 1$, $\mathbf{W}^{(1)}$ is a matrix of dimension $12 \cdot 15$, $\mathbf{x}$ is a vector of dimension $15 \cdot 1$ and $\mathbf{b}^{(1)}$ is a vector of dimension $12 \cdot 1$.

The parameters in this equation come from $\mathbf{W}^{(1)}$ and $\mathbf{b}^{(1)}$. By their dimensions, it follows that the number of parameters for the first equation is $15 \cdot 12 + 12 = 192$, as there are 15 input variables and 12 hidden units, and each unit has an associated bias.

The second equation, $\mathbf{q}^{(2)} = h\left(\mathbf{W}^{(2)}\mathbf{q}^{(1)} + \mathbf{b}^{(2)}\right)$, where $\mathbf{q}^{(2)}$ is a vector of dimension $6 \cdot 1$, $\mathbf{W}^{(2)}$ is a matrix of dimension $6 \cdot 12$, $\mathbf{q}^{(1)}$ is a vector of dimension $12 \cdot 1$ and $\mathbf{b}^{(2)}$ is a vector of dimension $6 \cdot 1$.

Again, the parameters come from $\mathbf{W}^{(2)}$ and $\mathbf{b}^{(2)}$. It follows that the number of parameters for $\mathbf{q}^{(2)}$ is $12 \cdot 6 + 6 = 78$.

The third equation, $z = \mathbf{W}^{(3)}\mathbf{q}^{(2)} + b^{(3)}$ where z is a scalar, $\mathbf{W}^{(3)}$ is a vector of dimension $1 \cdot 6$, $\mathbf{q}^{(2)}$ is a vector of dimension $6 \cdot 1$ and $b^{(3)}$ is a scalar.

Here, the parameters come from $\mathbf{W}^{(3)}$ and $b^{(3)}$, and it follows from their dimensions that the number of parameters for z is $6 \cdot 1 + 1 = 7$.

The total number of parameters in this two layer dense neural network is $192 + 78 + 7 = 277$.

/2p

c)

$$\mathbf{q}^{(1)} \text{ dimensions} : 12 \cdot 1$$

$$\mathbf{q}^{(2)} \text{ dimensions} : 6 \cdot 1$$

$$\mathbf{b}^{(1)} \text{ dimensions} : 12 \cdot 1$$

$$\mathbf{b}^{(2)} \text{ dimensions} : 6 \cdot 1$$

$$\mathrm{b}^{(3)} \text{ dimensions} : 1 \cdot 1$$

$$\mathbf{W}^{(1)} \text{ dimensions} : 12 \cdot 15$$

$$\mathbf{W}^{(2)} \text{ dimensions} : 6 \cdot 12$$

$$\mathbf{W}^{(3)} \text{ dimensions} : 1 \cdot 6$$

$$\mathrm{z} \text{ dimensions} : 1 \cdot 1$$

/3p

d)

A large number of hidden units in a layer increases the flexibility of a model. When increasing the flexibility of a model too much, we risk overfitting, which means that the bias is low but variance is high. Such problems can be observed when fitting a model, if the validation error begins to get worse. There are of course ways to deal with this, such as limiting the number of epochs.

/2p

e)

Yes, this changes the dimensions of z from a scalar to a vector. $\mathbf{z} = \mathbf{W}^{(3)}\mathbf{q}^{(2)} + \mathbf{b}^{(3)}$, where $\mathbf{z}$ is a vector of dimension $2 \cdot 1$, $\mathbf{W}^{(3)}$ is a vector of dimension $2 \cdot 6$, $\mathbf{q}^{(2)}$ is a vector of dimension $6 \cdot 1$ and $\mathbf{b}^{(3)}$ is a vector of dimension $2 \cdot 1$.

Thus, the number of parameters for $\mathbf{z}$ if using the softmax function is $6 \cdot 2 + 2 = 14$.

The total number of parameters in the network would be $192 + 78 + 14 = 284$.

/2p

/10

Total: 37

What a great exam, congratulations!