Question 1

a)

```
#create column for particle type based on Y1, Y2, Y3
dat$Particle <- ifelse(dat$Y1 == 1, "alpha", ifelse(dat$Y2 == 1, "beta", "rho"))

ggplot(data = dat, aes(x = X1, y = X2, color = Particle)) +
  geom_point(shape = 16, size = 3) +
  scale_color_manual(values = c("alpha" = "red", "beta" = "lightskyblue", "rho" = "green"),labels = c("
  labs(x = "X1", y = "X2", title = "") +
  theme_minimal() +  coord_fixed()  +
  theme(legend.position = "right",axis.line = element_line(color = "black", linewidth = 1))
```

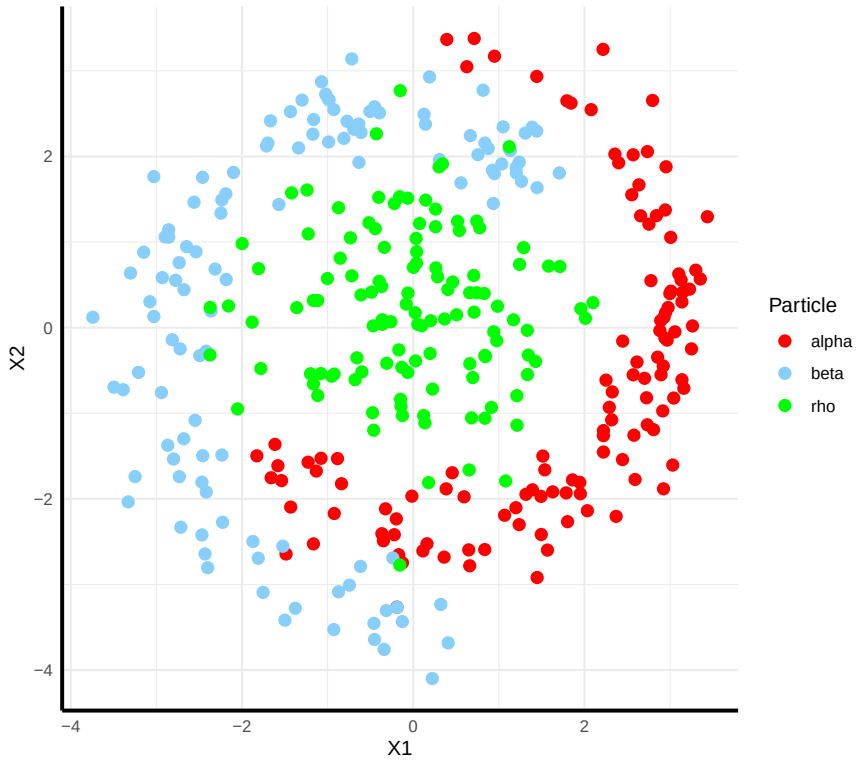

Figure 1: Scatter Plot of coordinates by Particle Type

The rho particles (green) form a circular region around the origin. The alpha particles (red) form a concave shape along the right side of the rho and beta particles. The beta particles (blue) also form a concave shape along the left side of the rho and alpha particles. There are signs of overlap with all three particles

A neural network is an appropriate model class for this problem as the particles above are not linearly separable and neural networks are good at learning non-linear decision boundaries and capturing complex patterns in the data.

b)

```
soft_max <- function(Z) {
  exp_Z <- exp(Z)
  denom <- colSums(exp_Z)
  probs <- exp_Z / matrix(denom, nrow = nrow(Z), ncol = ncol(Z), byrow = TRUE)
  return(probs)
}
```

c)

```
cross_ent <- function(y, y_hat) {
  if (y[1] == 1) {
    return(-log(y_hat[1]))
  } else if (y[2] == 1) {
    return(-log(y_hat[2]))
  } else if (y[3] == 1) {
    return(-log(y_hat[3]))
  }
}
```

Since y_i is one-hot encoded, the terms where $y_{ij} = 0$ contribute 0 to C_i. Therefore, it is computationally more efficient to evaluate only the terms corresponding to $y_{i\cdot} = 1$ as computing terms where $y_{i\cdot} = 0$ is computationally a waste of time and resources.

d)

```
g <- function(Yhat, Y) {
  #Yhat: 3 x N matrix of predicted probabilities
  #Y: 3 x N matrix of one-hot encoded responses
  log_Yhat <- -log(Yhat)
  C_i <- colSums(Y * log_Yhat)
  return(mean(C_i))
}
```

e)

For an (m, m)-AFnetwork with dim(x)=p , dim(y) = q ,

- Augmented features: original p inputs are transformed into p augmented features via a separate hidden layer of size p.
- $2p$ input nodes = (Original p nodes + Augmented features p).
- m nodes in the first hidden layer.
- m nodes in the second hidden layer.

- q output nodes.
- Augmentation Layer: p^2 Weights + p Bias terms = $p^2 + p$ parameters.
- Input to First Hidden Layer: $2pm$ Weights+ m Bias terms = $2pm + m$ parameters.
- First to Second Hidden Layer: m^2 Weights + m Bias Terms = $m^2 + m$ parameters.
- Second Hidden to Output Layer: mq weights + q biases = $mq + q$ parameters.
- Total parameters = $p(p + 1 + 2m) + m(m + 2 + q) + q$.

f)

```
# activation function for the augmented and hidden layers
act_fn <- function(Z){
  tanh(Z)
}

# X   - input matrix (N x p)
# Y     - output matrix(N x q)
# theta - A parameter vector (includes all of the parameters)
# m    - Number of nodes on hidden layers
# v     - regularisation parameter
AF_neural_net <- function(X, Y, theta, m, v){

   N = dim(X)[1]
   p = dim(X)[2]
   q = dim(Y)[2]

   # Populate weight-matrix and bias vectors:
   index = 1:(p*p)
   W1    = matrix(theta[index], p, p) #augmented layer
   index = max(index)+1:(2*p*m)
   W2    = matrix(theta[index], 2*p, m) #hidden layer 1
   index = max(index)+1:(m*m)
   W3    = matrix(theta[index], m, m) # hidden layer 2
   index = max(index)+1:(m*q)
   W4    = matrix(theta[index], m, q) # output layer

   index = max(index) + 1:(p)
   b1    = matrix(theta[index],p , 1) # augmented layer
   index = max(index)+1:(m)
   b2    = matrix(theta[index],m,1)   # hidden layer 1
   index = max(index)+1:(m)
   b3    = matrix(theta[index],m,1)   # hidden layer 2
   index = max(index)+1:(q)
   b4    = matrix(theta[index],q,1)   # output layer

   # activation functions
   ones_t = matrix(1,1,N)
   A0 = t(X)
   A1 = act_fn(t(W1) %*% A0 +b1 %*% ones_t) # pxN
   A_aug = rbind(A0, A1)                            # 2pxN
```

```
  A2 = act_fn(t(W2) %*% A_aug +b2 %*% ones_t) # mxN
  A3 = act_fn(t(W3) %*% A2 +b3 %*% ones_t) # mxN
  A4 = soft_max(t(W4) %*% A3 + b4 %*% ones_t)      # qnN

  # errors
  Yhat = t(A4)
  E1 = g(t(Yhat), t(Y)) #  cross entropy
  E2 = E1 + v/N * (sum(W1^2) + sum(W2^2) + sum(W3^2) + sum(W4^2))

  return(list(out = Yhat, E1 = E1, E2 = E2))
}
```

g)

```
set.seed(2025)

X <-  as.matrix(dat[,1:3])
Y <-  as.matrix(dat[,4:6])
N <-  dim(X)[1]

# splitting the data into 80-20 ratio for training and validation
set <- sample(1:N, round(0.8 * N), replace = FALSE)
X_train <- X[set,]
Y_train <- Y[set,]
X_val <- X[-set,]
Y_val <- Y[-set,]

p <- dim(X)[2]
q <- dim(Y)[2]
m <- 4
npars <- p*(p + 1 + 2*m) + m*(m + 2 + q) + q

nv <- 20
v_seq <-  exp(seq(-6,2,length.out = nv))
valid_err <- rep(NA, nv)

# validation
for(i in 1:nv){
  v <- v_seq[i]

  # penalised obj fn
  obj_pen = function(pars)
  {
    res_mod = AF_neural_net(X_train, Y_train, pars, m, v)
    return(res_mod$E2)
  }

  theta_rand <- runif(npars,-1,1)
  res_opt = nlm(obj_pen, theta_rand, iterlim = 2000)

  res_fit = AF_neural_net(X_val, Y_val, res_opt$estimate, m, 0)
```

```r
  valid_err[i] <- res_fit$E1
}

# selecting optimal v
v_opt <- v_seq[which.min(valid_err)]

plot_data <- data.frame(v = v_seq, ValidationError = valid_err)
ggplot(plot_data, aes(x = v, y = ValidationError)) +
  geom_line(color = "blue") + geom_point(color = "blue") +
  geom_vline(xintercept = v_opt, linetype = "dotted", color = "red", size = 1) +
  labs(x = "v", y = "Validation Cross-Entropy Error", title = "") +
  xlim(0, 2.5) + theme_minimal() + coord_fixed()
```

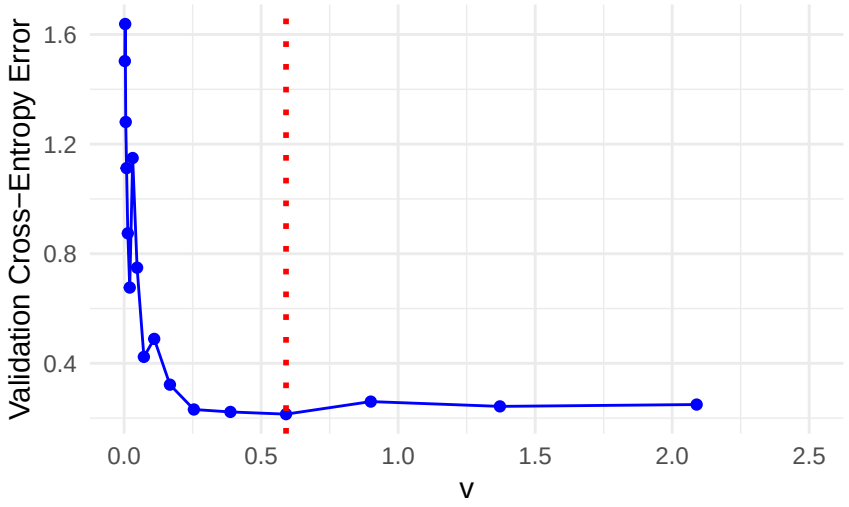

Figure 2: Validation Error vs Regularisation Parameter(v) of the neural network

We can see from the red dotted line, that the minimum validation cross entropy error occurs at the regularization level of 0.591. Therefore we will choose this value as our regularization level.

h)

```r
# Fit model with optimal v
obj <- function(pars) {
```

```
  res <- AF_neural_net(X_train, Y_train, pars, m = 4, v = v_opt)
  return(res$E2)
}
theta_rand <- runif(npars, -1, 1)
res_opt <- nlm(obj, theta_rand, iterlim = 2000)

#  response curves for X1 (fixing X2 = 0)
xx <- seq(-4, 4, length.out = 100)
Y0 <- matrix(0, 1, 3)
Yhat_A_X1 <- matrix(NA, 100, 3)  # Detector X3 = 0
Yhat_B_X1 <- matrix(NA, 100, 3)  # Detector X3 = 1

for (i in 1:100) {
  X_A <- matrix(c(xx[i], 0, 0), 1, 3)
  res_A <- AF_neural_net(X_A, Y0, res_opt$estimate, m = 4, v = v_opt)
  Yhat_A_X1[i, ] <- res_A$out
  X_B <- matrix(c(xx[i], 0, 1), 1, 3)
  res_B <- AF_neural_net(X_B, Y0, res_opt$estimate, m = 4, v = v_opt)
  Yhat_B_X1[i, ] <- res_B$out
}

#  response curves for X2 (fixing X1 = 0)
Yhat_A_X2 <- matrix(NA, 100, 3)  # Detector X3 = 0
Yhat_B_X2 <- matrix(NA, 100, 3)  # Detector X3 = 1

for (i in 1:100) {
  X_A <- matrix(c(0, xx[i], 0), 1, 3)
  res_A <- AF_neural_net(X_A, Y0, res_opt$estimate, m = 4, v = v_opt)
  Yhat_A_X2[i, ] <- res_A$out
  X_B <- matrix(c(0, xx[i], 1), 1, 3)
  res_B <- AF_neural_net(X_B, Y0, res_opt$estimate, m = 4, v = v_opt)
  Yhat_B_X2[i, ] <- res_B$out
}

# (X1 varying, X3 = 0)
plot_data_A_X1 <- data.frame(X1 = xx, Alpha = Yhat_A_X1[,1], Beta = Yhat_A_X1[,2], Rho = Yhat_A_X1[,3])
plot_data_long_A_X1 <- pivot_longer(plot_data_A_X1, cols = c("Alpha", "Beta", "Rho"), names_to = "Class

# (X1 varying, X3 = 1)
plot_data_B_X1 <- data.frame(X1 = xx, Alpha = Yhat_B_X1[,1], Beta = Yhat_B_X1[,2], Rho = Yhat_B_X1[,3])
plot_data_long_B_X1 <- pivot_longer(plot_data_B_X1, cols = c("Alpha", "Beta", "Rho"), names_to = "Class

# (X2 varying, X3 = 0)
plot_data_A_X2 <- data.frame(X2 = xx, Alpha = Yhat_A_X2[,1], Beta = Yhat_A_X2[,2], Rho = Yhat_A_X2[,3])
plot_data_long_A_X2 <- pivot_longer(plot_data_A_X2, cols = c("Alpha", "Beta", "Rho"), names_to = "Class

# (X2 varying, X3 = 1)
plot_data_B_X2 <- data.frame(X2 = xx, Alpha = Yhat_B_X2[,1], Beta = Yhat_B_X2[,2], Rho = Yhat_B_X2[,3])
plot_data_long_B_X2 <- pivot_longer(plot_data_B_X2, cols = c("Alpha", "Beta", "Rho"), names_to = "Class

# response curves for X1 (X3 = 0)
p1 <- ggplot(plot_data_long_A_X1, aes(x = X1, y = Probability, color = Class)) +
  geom_line(linewidth = 1) +
```

```r
  scale_color_manual(values = c("Alpha" = "red", "Beta" = "lightskyblue", "Rho" = "green")) +
  labs(x = "X1", y = "Class Probability", title = "X3 = 0") +
  theme_minimal() +
  theme(legend.position = "top", legend.title = element_blank(),plot.title = element_text(hjust = 0.5))

# Plot response curves for X1 (X3 = 1)
p2 <- ggplot(plot_data_long_B_X1, aes(x = X1, y = Probability, color = Class)) +
  geom_line(linewidth = 1) +
  scale_color_manual(values = c("Alpha" = "red", "Beta" = "lightskyblue", "Rho" = "green")) +
  labs(x = "X1", y = "Class Probability", title = "X3 = 1") +
  theme_minimal() +
  theme(legend.position = "top", legend.title = element_blank(),plot.title = element_text(hjust = 0.5))

# Plot response curves for X2 (X3 = 0)
p3 <- ggplot(plot_data_long_A_X2, aes(x = X2, y = Probability, color = Class)) +
  geom_line(linewidth = 1) +
  scale_color_manual(values = c("Alpha" = "red", "Beta" = "lightskyblue", "Rho" = "green")) +
  labs(x = "X2", y = "Class Probability", title = "X3 = 0") +
  theme_minimal() +
  theme(legend.position = "top", legend.title = element_blank(),plot.title = element_text(hjust = 0.5))

# Plot response curves for X2 (X3 = 1)
p4 <- ggplot(plot_data_long_B_X2, aes(x = X2, y = Probability, color = Class)) +
  geom_line(linewidth = 1) +
  scale_color_manual(values = c("Alpha" = "red", "Beta" = "lightskyblue", "Rho" = "green")) +
  labs(x = "X2", y = "Class Probability", title = "X3 = 1") +
  theme_minimal() +
  theme(legend.position = "top", legend.title = element_blank(),plot.title = element_text(hjust = 0.5))

# Display all four plots
grid.arrange(p1, p2, p3, p4, ncol = 2)
```

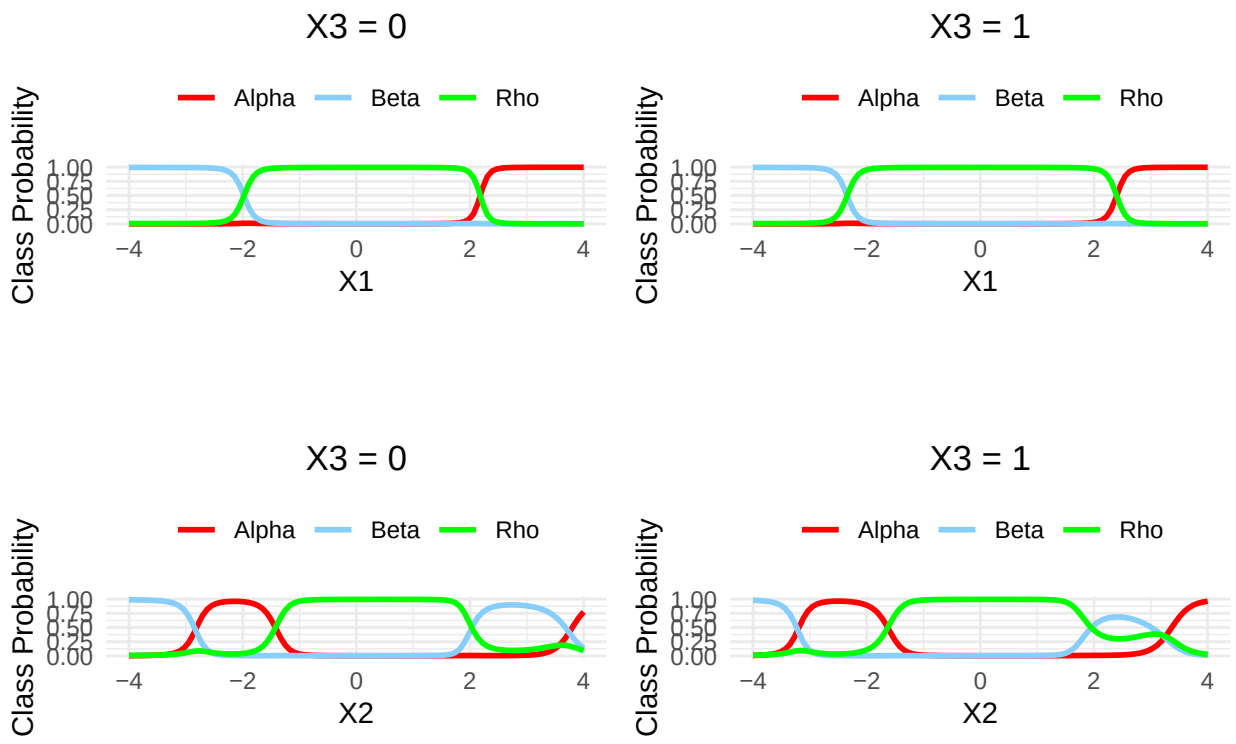

Figure 3: Response Curves for Particle Classification by Detector Type

i)

AFnetworks introduce a dedicated hidden layer that learns nonlinear patterns of the input features (e.g. rho particles in circular regions and alpha and beta particles in concave shapes) before they are fed into the main network. These transformations act like automatic, learned features that might capture complex patterns the original inputs could not express on their own leading to better generalization and performance of the model.

Additionally, as AFnetworks explicitly separate the original inputs from augmented inputs. This allows us to understand what features matter and how they matter. Hence, helps us understand how the network is reshaping the data before making predictions. For example if we needed to take a closer look at the results it produces, we could look at the value produced by the activation function of tanh in A1 for different observations.

The AFnetwork's, augmentation layer performs feature engineering which reduces the need for large hidden layers to correctly classify observations. For example in our AFnetwork, we only needed 4 nodes in our hidden layers instead of needing more nodes/layers which could have become computationally expensive without our augmented layer.