MNIST Written Character Classification

St.No. 17005465

9th December 2018

1. Introduction

Awkward authors’ note

In the interest of honesty I should note that pretty much every tutorial on neural networks I’ve found used the MNIST dataset as the example, it seems pretty ubiquitous. Primarily I’ve used an introduction to Keras for R at https://blog.rstudio.com/2017/09/05/keras-for-r/ (https://blog.rstudio.com/2017/09/05/keras-for-r/), and the book Deep Learning in Python by F. Chollet. I’ve done my best to forget what I learned of specific parameters in building these NN’s, and in fairness the guides did not use PCA. Also yes, Keras for R is just calling Python in the background and yes, that’s just embarassing for R. But ONS insists… I've hidden the setup code that loads the data and libraries but my full code (original R markdown) is available on my github (https://github.com/OzwaldCavendish/classifying_MNIST).

Machine Learning and Supervised Classification

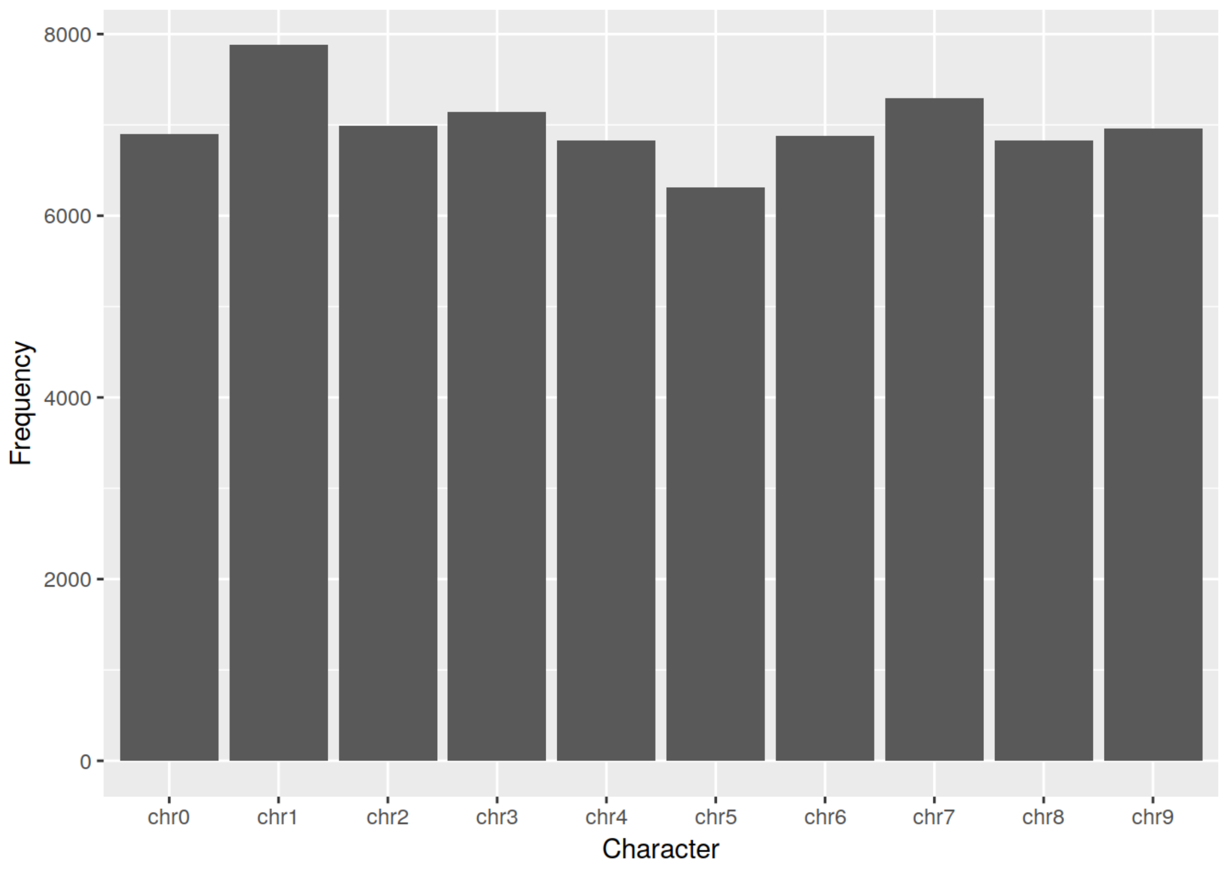
In machine learning, a supervised classification problem entails fitting an algorithm to map some input features X to a desirable output label Y, ie; training a machine to automatically classify a sample into some category based on available data about it (“Is this customer fraudulent? Is this a picture of a face?”).

Labelled data is used to train the ML algorithm, split into training and test data. In training the algorithm we run the risk of over-fitting, a complex model can become over-reliant on aspects of the data that are just quirks of the particular set. The result is reduced performance when the trained model is used to classify new data that it’s not been exposed to during training. The solution to this is to hold out a “test” set of labelled data, and use it to evaluate the model’s performance on unseen data.

In this example we have a training dataset of 60,000 samples and a test set of 10,000 samples. We will train different kNN models and neural networks on the first 60,000 and evaluate performance of the different models on the last 10,000. This will be done for kNN by first reducing the dimensionality of the data using PCA, and then selecting the best performing hyperparameter k (number of nearest neighbors to consider) through cross-validated training performance before evaluating a final model on the test set.

As seen in the figure below the classes 0-9 for MNIST are roughly balanced - in this case simple “Accuracy” is a suitable metric for model performance. If the dataset were unbalanced, “Precision”, “Recall” and their average “F1-Score” would be good choices because they can penalise the model for poor performance on important (for humans) minority classes.

```
# Plot class membership of the training and test data
all_ys <- c(as.character(trainset$y), as.character(testset$y))

# Create a table of counts by membership
all_ys_df <- data.frame( table( all_ys ) )

colnames(all_ys_df) <- c("Character", "Frequency")

# Visualise the numbers within each category
pl <- ggplot(dat=all_ys_df, aes(x=Character, y=Frequency)) +
      geom_bar(stat="identity")
pl
```



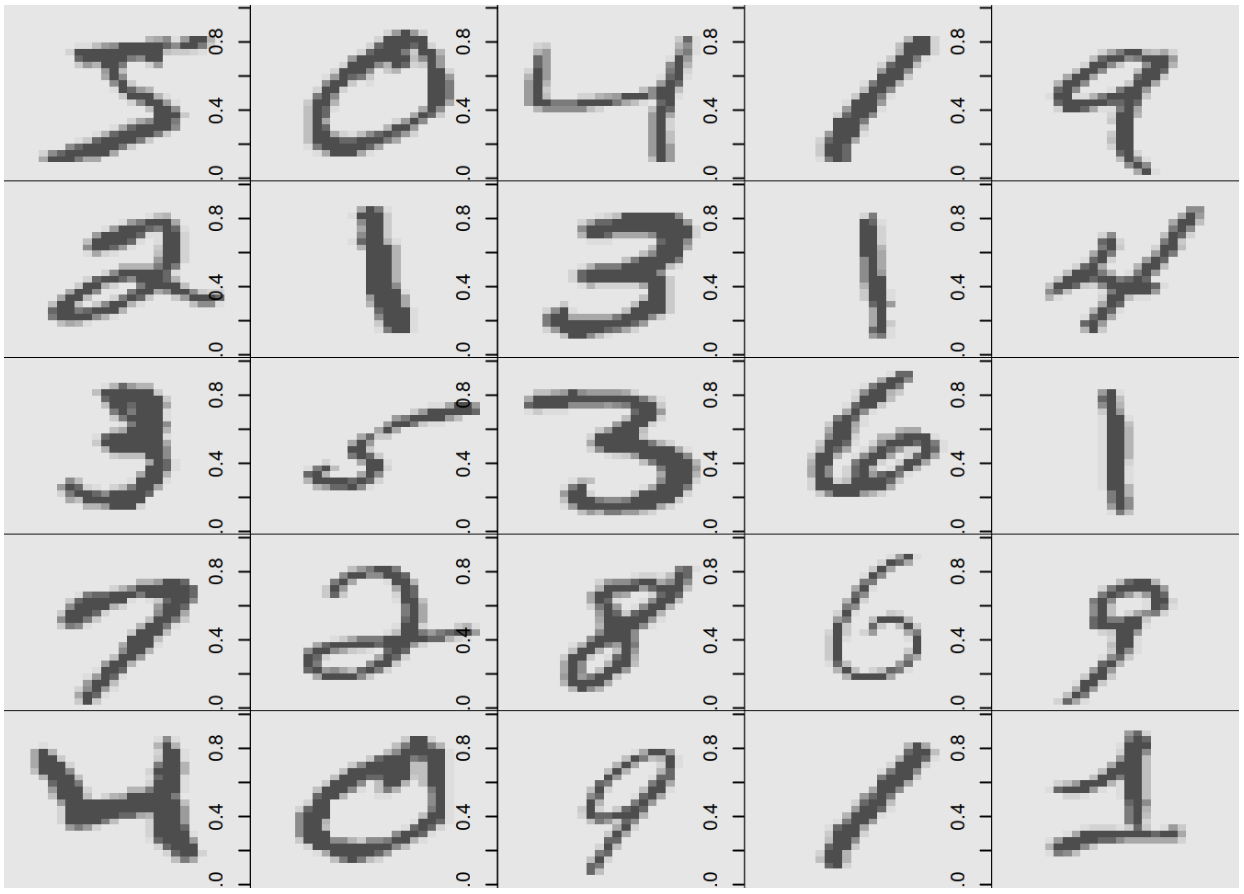
The MNIST data

The MNIST character dataset is an industry standard image classification problem. An important aspect of the data is that the images have been centred and scaled to the characters, reduced to black and white images and the resolution set uniform. This removes the need for us to engage in complex image normalisation steps before attempting classification, or more complex models like

Convolutional Neural Networks that can automate the discovery of the lowest-level features and relevant areas of an image. ML algorithms applied to this dataset are then left with higher level logic, finding significant edges and pixels to distinguish between characters.

The images are 28 * 28 matrices of pixel intensity, the first 25 images are displayed below. The values (which range between 0-255 to represent pixel intensity) are scaled to lie between 0 and 1 - this scaling will help the neural network model to converge and is in general a good idea. More sophisticated methods of scaling values exist but offer no advantage here.

```
# ------------------------------------------- #
#      The first few training images
# ------------------------------------------- #



to.read = file("./data/train-images.idx3-ubyte", "rb")

# Read the header of the training data
header = readBin(to.read, 'integer', n=4, endian="big")

# Retrieve and plot 25 of the images
# (it's a sequential read)

par(mfrow=c(5, 5), mar=c(0, 0, 0, 0))
for(i in 1:25) {
  m = matrix(readBin(to.read, 'integer', size=1, n=28*28, endian="big", sign
ed=F),28,28)

  # Invert the image values for plotting
  m <- 255 - m

  # visualise
  image(m[,28:1], col=grey.colors(12), zlim=c(0.0, 255.0))
}
```



```
# Close the file
close(to.read)
```

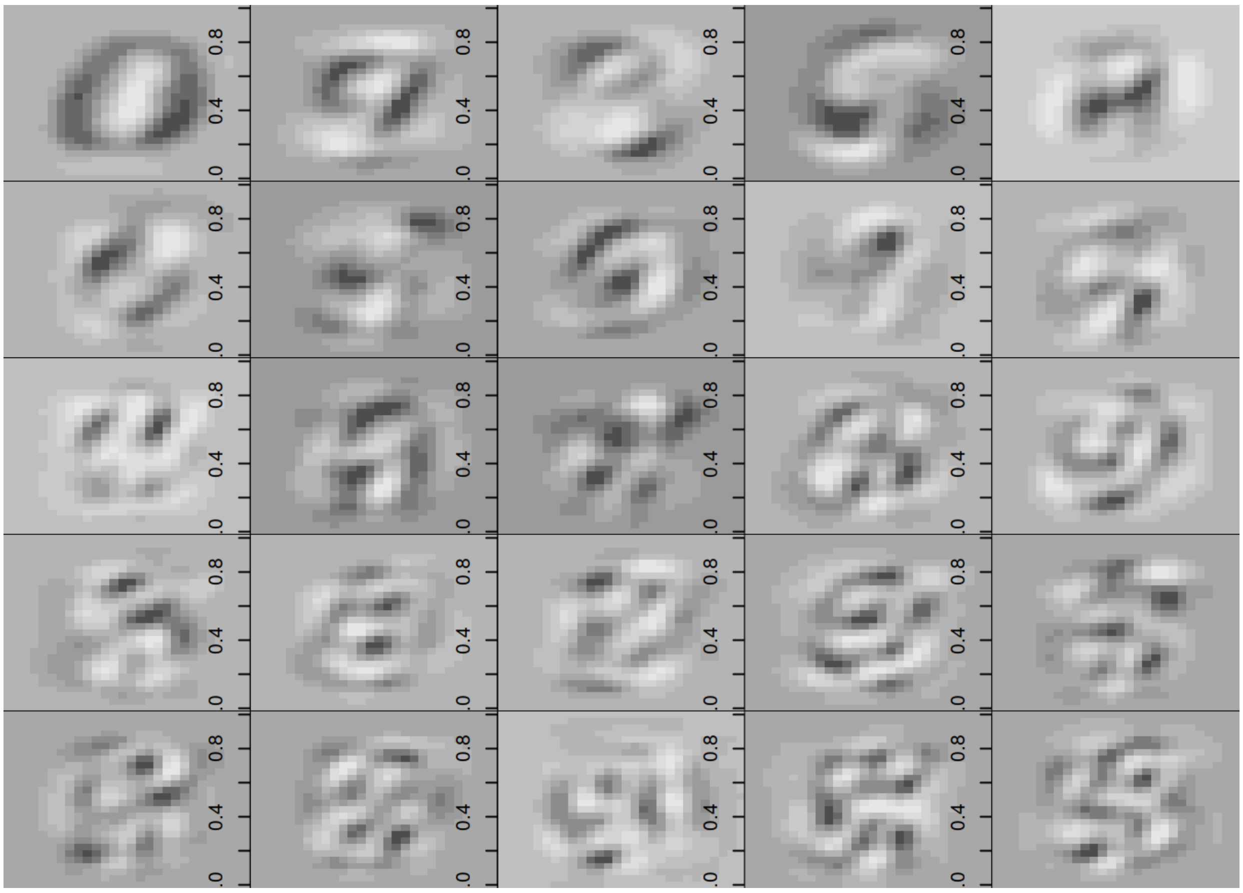
2. Dimensionality Reduction

Principal Component Analysis (PCA)

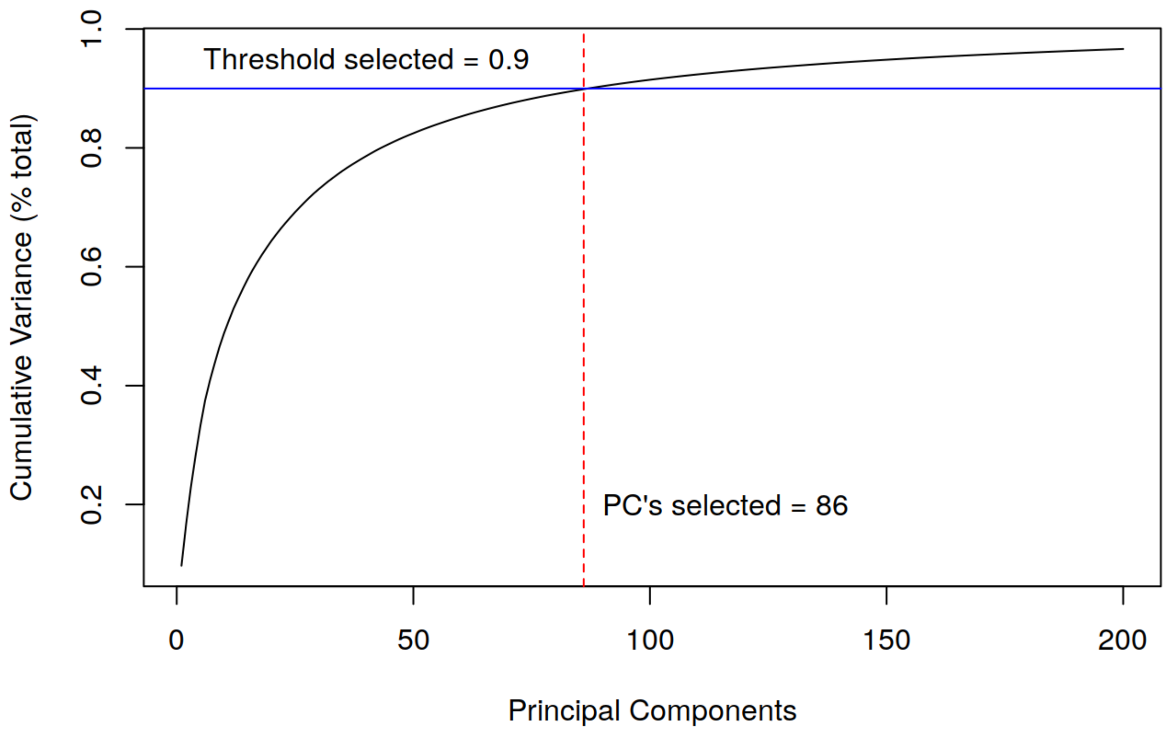
Principal Component Analysis (PCA) works by calculating the axis in the n-dimensional data space along which variance is greatest, and then sequentially finding perpendicular axes with the next greatest variance. The algorithm can be used for dimensionality reduction, by dropping all but the first few axes embodying the highest percentage of variance within the dataset. This retains axes along which data is well separated while dropping “noisy” axes lacking clear information (separation of categories). The visualised rotations, the linear mappings from the images to each principal component, show for example features like empty space at the centre (first tile) and curves that look suspiciously like the number nine (second tile).

Common practice is to keep those that embody 80 % of the variance in the data. Based on a visual inspection of the cumulative variance with dimensions below, 90 % is chosen instead. This looks to be a good balance between capturing variance and the diminishing returns with more dimensions. This variance is in the first 86 Principal components, a significant improvement upon the original dimensionality of 784 individual features (pixels). The choice of number of principal components is another hyperparameter that one might choose to optimise.

```r
# First; pca!
train_pca <- prcomp(trainset[,2:785], center=TRUE)

# For fun, visualise the PCA rotations
par(mfrow=c(5, 5), mar=c(0, 0, 0, 0))
for(i in 1:25) {

  # Get a principal component's mapping and rescale to pixel values
  pca_comp <- rescale(train_pca$rotation[,i], c(0,255))
  m = matrix(pca_comp,28,28)

  # Invert the image values for plotting
  m <- 255 - m

  # visualise
  image(m[,28:1], col=grey.colors(12), zlim=c(0.0, 255.0))
}
```



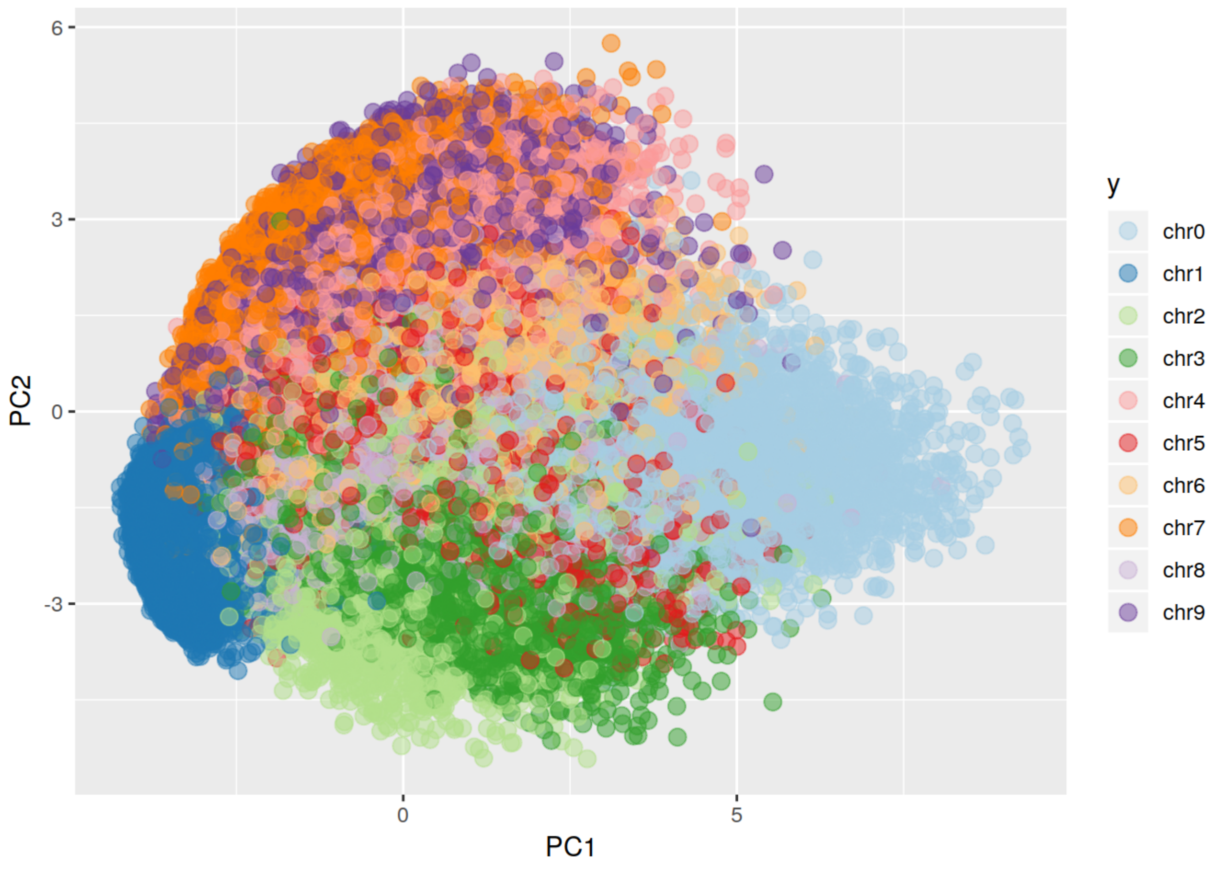
```
# How many of the PCA axes are useful?  Assume threshold of 80 % of cumulati
ve variance
select_dimensionality <- function(sdevs, threshold=0.8) {

  # Function assumes it receives a descending-ordered list of the standard d
eviations of a PCA object
  pca_variance <- sdevs ** 2.0  # Variance is, of course, SD to power 2
  cumulative_variance <- cumsum(pca_variance)

  num_PCs = sum( cumulative_variance < threshold * sum(pca_variance) )

  plot(cumulative_variance[1:200] / sum(pca_variance) ~ seq(1:200), type="l"
, xlab="Principal Components", ylab= "Cumulative Variance (% total)")
  abline(v=num_PCs, col="red", lty=2)
  abline(h=threshold, col="blue", lty=1)
  text(x=num_PCs+30, y=0.2, labels = paste("PC's selected =", as.character(n
um_PCs)))
  text(x=40, y=threshold+0.05, labels=paste("Threshold selected =", as.chara
cter(threshold)))

  return(num_PCs)
}

num_PCs <- select_dimensionality(summary(train_pca)$sdev, threshold=0.9)
```



```
# Extract the data to a new DF for later use in ML algorithms
# Drop unwanted PC's
trainset.pca <- data.frame(train_pca$x[,1:num_PCs])
trainset.pca$y <- trainset$y

# Finally, apply the PCA transformations to the test data
testset.pca <- data.frame( predict(train_pca, testset[,2:785])[,1:num_PCs] )
testset.pca$y <- testset$y
colnames(testset.pca) <- sub("x.", "PC", colnames(testset.pca))
```

A secondary advantage of PCA is that it becomes possible to visualise the associations between high-dimensional data by plotting the first few Principal components, containing the greatest variance:

```
# Visualise first two principal components nicely
pl <- ggplot(dat=trainset.pca, aes(x=PC1, y=PC2, group=y, color=y)) +
      geom_point(size=3, alpha=0.5) +
      scale_color_brewer(palette="Paired")
pl
```



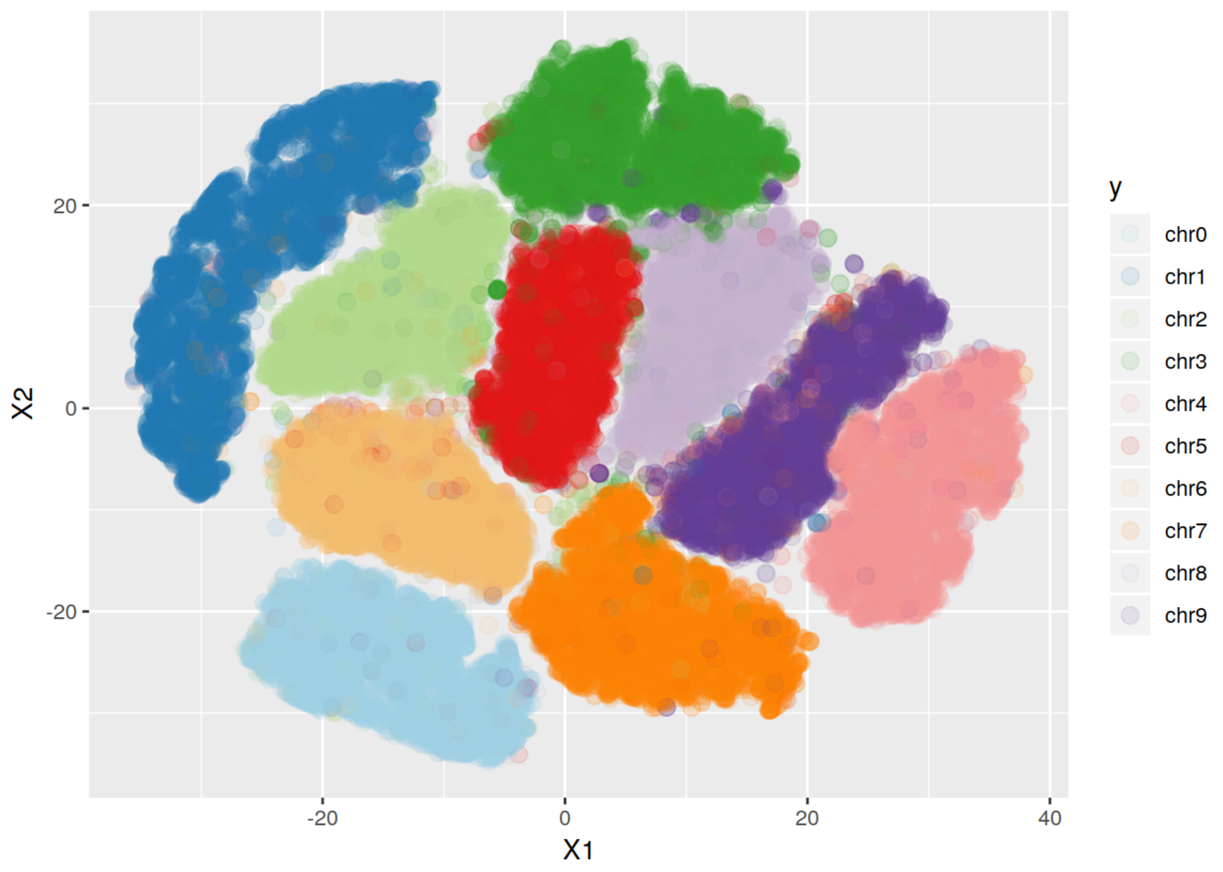
t-SNE

For merely visualising high-dimensional data, there is also t-SNE (t-Distributed Stochastic Neighbor Embedding). This is more powerfully able to visualise the separability of data by dynamically mapping the distance between points in the high-dimensional space to a low-dimensional space in a flexible, stochastic way that is itself considered an ML algorithm. In this case the algorithm is able to find mappings that clearly separate the majority of the data within the two-dimensional plot. This indicates that plenty of information is present within the dataset to power classification.

The downside of the method is that because it does not create a specific, algorithmic mapping between the high-dimensional space and the low, it cannot be applied to new data or test data. It can only be applied to a dataset as a whole and and is not reproducible, so it can't be easily used to create inputs for ML algorithms.

```
# Set random seed for reproducibility (stochastic process)
set.seed(7)

# Second, t-SNE!  Much preferable.
# Can reduce to arbitrary number of dimensions, default is 2
train_tsne <- Rtsne(as.matrix(trainset[,2:785]), num_threads=8)  # Extra thr
eads not used because openMP on Windows sucks
```

```
trainset.tsne <- data.frame(train_tsne$Y)
trainset.tsne$y <- trainset$y

# The t-SNE algorithm directly reduced the number of dimensions to 2
pl <- ggplot(dat=trainset.tsne, aes(x=X1, y=X2, group=y, color=y)) +
  geom_point(size=3, alpha=0.1) +
  scale_color_brewer(palette="Paired")
pl
```



3. Classification with k-Nearest Neighbors (k-NN)

k-NN applied to this dataset benefits greatly from scaling and Principal Component Analysis, both of which have already been applied. What remains is to determine the best value for the hyperparameter k. For this purpose I'm using the "caret" ML library, which has lots of useful options and functions for performing grid searches over hyperparameters, before reporting and comparing results on multiple metrics. Here I compare a raft of different potential values for k in a k-NN model.

Once the model's ideal hyperparameters are identified the model is automatically retrained on all of the available training data, in preparation for application to test data. At this stage the models are evaluated through 5-fold cross-validation rather than creating an additional hold-out validation set, given the low-ish number of training samples and the reasonably fast execution of the algorithm.

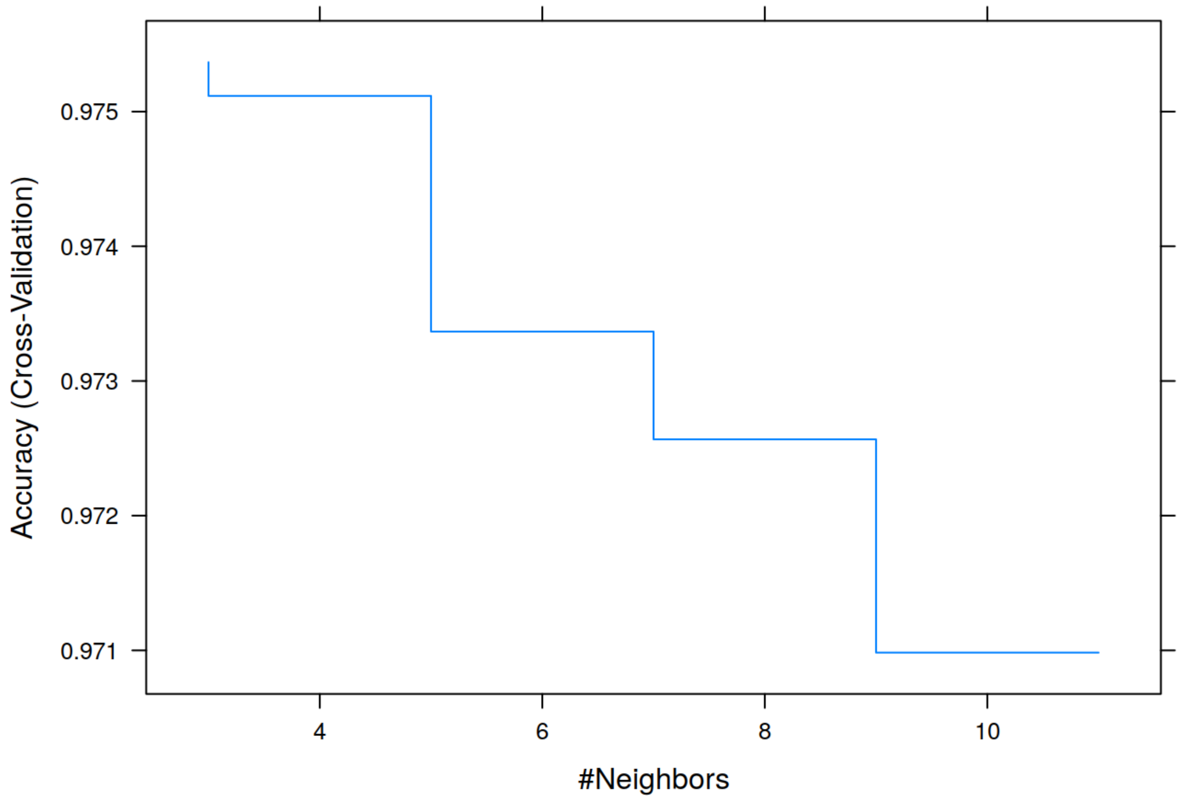
The highest predictive accuracy of 0.9753 is obtained with k=3 nearest neighbors. As a baseline, with 10 ~ balanced classes we could expect a random process to achieve an accuracy of 0.1. As an aside, the logLoss and AUC are poor guides to performance here because they are calculated from the probabilities of each label for each sample rather than the final classifications, and so for any given sample can appear to improve as a higher k provides better overall probabilities, whilst the final decision/predicted class is in fact wrong.

```
# Set up the training algorithm (allows for specifying performance measures
 and cross-validation folds, etc)
# Specified class probabilities, a more comprehensive summary function and 5
-fold cross-validation
control <- trainControl(method="cv", number=5, classProbs=TRUE, summaryFunct
ion=multiClassSummary)

# The metric the algorithm will use to decide the best-performer
metric <- "Accuracy"

# Create the parameterspace to be explored (only one for this algorithm)
param_grid = data.frame(k = c(3, 5, 7, 9, 11))
```

Fitting the model can take a while.

```
# Fit the kNN models
set.seed(7)
fit.knn <- train(y~., data=trainset.pca, method="knn", metric=metric, trCont
rol=control, tuneGrid=expand.grid(param_grid))
save(fit.knn, file="knn_model.Rdata")
```

```
# Plot performance of all knn variants
plot(fit.knn, print.thres=0.5, type="S")
```



```
# Lets look at the model results on some key measures of accuracy
fit.knn$results[,1:7]
```

	k <dbl>	logLoss <dbl>	AUC <dbl>	prAUC <dbl>	Accuracy <dbl>	Kappa <dbl>	Mean_F1 <dbl>
1	3	0.3649283	0.9946004	0.05791487	0.9753667	0.9726196	0.9751798
2	5	0.2789634	0.9961368	0.09265941	0.9751167	0.9723416	0.9749598
3	7	0.2408438	0.9968465	0.12176355	0.9733667	0.9703963	0.9732042
4	9	0.2185707	0.9972738	0.14768607	0.9725667	0.9695069	0.9724265
5	11	0.2007899	0.9976238	0.17121879	0.9709834	0.9677468	0.9708355

5 rows

4. A Further Model - Neural Networks

For the fun of it I'm going to implement a simple, fully connected feedforward neural network. Normally for image analysis problems a more complex Convolutional Neural Network (CNN) is better able to fit to the low-level features useful for classification of messy image data. In this case, as stated earlier, the images are already centred, grey-scaled and reduced to standardised 28*28 pixel resolution. In combination with PCA to find optimal low-level features a standard MLP should perform quite well on this task.

First, one-hot encode the targets and convert the input to a numeric matrix. Keras comes with the MNIST dataset pre-formatted but I want to use the data I've already "PCA'd"" and used for kNN, for consistency.

```
# Convert the (now character) classes into one-hot-encoded categorical targe
t vars
trainset_y_cat = to_categorical( as.integer( sub("chr", "", trainset.pca$y)
 ) )
testset_y_cat = to_categorical( as.integer( sub("chr", "", testset.pca$y) )
 )

# Convert the input data into a pure number matrix format
trainset_x_mat = as.matrix( trainset.pca[, -ncol(trainset.pca)])
testset_x_mat = as.matrix( testset.pca[, -ncol(testset.pca)])
```

```
# Tell it to use python 3 to run this stuff
use_python("usr/bin/python3")


# Model encapsulated in function to facilitate creating multiple variants
build_model <- function(hidden_layers=2, n_neurons=c(64, 32), drop_rate=0.2,
 input_shape) {

  # Import sequential model class
  model <- keras_model_sequential()

  # Create all the hidden layers
  for(i in seq(1:hidden_layers)) {

    if(i == 1) {
      model %>% layer_dense(units=n_neurons[i], activation="relu", input_sha
pe=input_shape)
    } else {
      model %>% layer_dense(units=n_neurons[i], activation="relu")
    }

    # Add dropout to every hidden layer
    model %>% layer_dropout(rate=drop_rate)
  }

  # Create the final output layer
  model %>% layer_dense(units=10, activation="softmax")

  # Set up loss function, optimiser and metric
  model %>% compile(
    loss="categorical_crossentropy",
    optimizer=optimizer_rmsprop(),
    metrics=c("accuracy")
  )

  return(model)
}
```

Refining neural network hyperparameters “by hand”

A range of neural networks with different hyperparameters are were tested iteratively, manually refining the number of neurons and the dropout rate for regularisation. A more effective method in future, especially given the rapidity of training (< 1 minute per model) would be to set up a grid search over the number of neurons in the 1st and 2nd layers, and over the dropout rate.

The second model tested gives the best validation accuracy at the end of training despite showing signs of slight overfitting, with training accuracies consistently ~ 0.5 % higher than validation accuracies in the final few epochs (see plots). This model is retrained with all available training data at the end. In comparison, in the plots the first neural network with more neurons in the second layer shows worse overfitting and overall validation performance. Increasing the dropout rate to combat this (model 4) lowers the validation accuracy, in later runs for this model the training accuracy is significantly and consistently lower than validation accuracy.

```
# Fit a variety of NN's

# Start with something oversized
model1 <- build_model(n_neurons = c(256, 64), input_shape=c( dim(trainset.pc
a)[2] - 1 ))
summary(model1)
```

```
## ________________________________________________________________________
__
## Layer (type)                     Output Shape                  Param #

## ========================================================================
==
## dense_1 (Dense)                  (None, 256)                   22272

## ________________________________________________________________________
__
## dropout_1 (Dropout)              (None, 256)                   0

## ________________________________________________________________________
__
## dense_2 (Dense)                  (None, 64)                    16448

## ________________________________________________________________________
__
## dropout_2 (Dropout)              (None, 64)                    0

## ________________________________________________________________________
__
## dense_3 (Dense)                  (None, 10)                    650

## ========================================================================
==
## Total params: 39,370
## Trainable params: 39,370
## Non-trainable params: 0
## ________________________________________________________________________
__
```

```
set.seed(7)
history1 <- model1 %>% fit(trainset_x_mat, trainset_y_cat, epochs = 30, batc
h_size = 64, validation_split = 0.2)

# Shrink second hidden layer, reduce overfitting?
model2 <- build_model(n_neurons = c(256, 32), input_shape=c( dim(trainset.pc
a)[2] - 1 ))
summary(model2)
```

```
## ________________________________________________________________________
__
## Layer (type)                     Output Shape                  Param #

## ========================================================================
==
## dense_4 (Dense)                  (None, 256)                   22272

## ________________________________________________________________________
__
## dropout_3 (Dropout)              (None, 256)                   0

## ________________________________________________________________________
__
## dense_5 (Dense)                  (None, 32)                    8224

## ________________________________________________________________________
__
## dropout_4 (Dropout)              (None, 32)                    0

## ________________________________________________________________________
__
## dense_6 (Dense)                  (None, 10)                    330

## ========================================================================
==
## Total params: 30,826
## Trainable params: 30,826
## Non-trainable params: 0
## ________________________________________________________________________
__
```

```
set.seed(7)
history2 <- model2 %>% fit(trainset_x_mat, trainset_y_cat, epochs = 30, batc
h_size = 64, validation_split = 0.2)

# Shrink it a bit further, to reduce comp time and overfitting
model3 <- build_model(n_neurons = c(128, 32), input_shape=c( dim(trainset.pc
a)[2] - 1 ))
summary(model3)
```

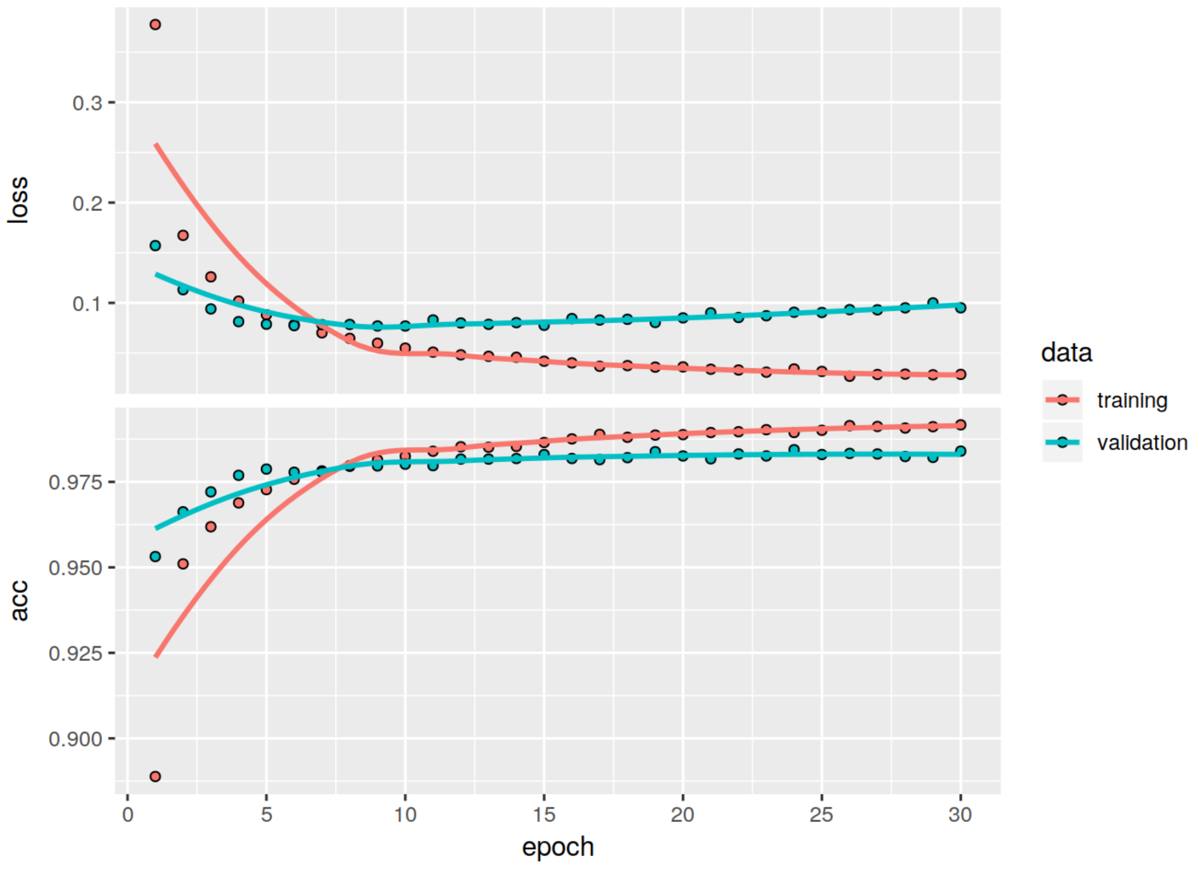
```
## ________________________________________________________________________
__
## Layer (type)                     Output Shape                  Param #

## ========================================================================
==
## dense_7 (Dense)                  (None, 128)                   11136

## ________________________________________________________________________
__
## dropout_5 (Dropout)              (None, 128)                   0

## ________________________________________________________________________
__
## dense_8 (Dense)                  (None, 32)                    4128

## ________________________________________________________________________
__
## dropout_6 (Dropout)              (None, 32)                    0

## ________________________________________________________________________
__
## dense_9 (Dense)                  (None, 10)                    330

## ========================================================================
==
## Total params: 15,594
## Trainable params: 15,594
## Non-trainable params: 0
## ________________________________________________________________________
__
```

```
set.seed(7)
history3 <- model3 %>% fit(trainset_x_mat, trainset_y_cat, epochs = 30, batc
h_size = 64, validation_split = 0.2)

# Try a higher dropout rate
model4 <- build_model(n_neurons = c(256, 64), drop_rate=0.4, input_shape=c(
 dim(trainset.pca)[2] - 1 ))
summary(model4)
```

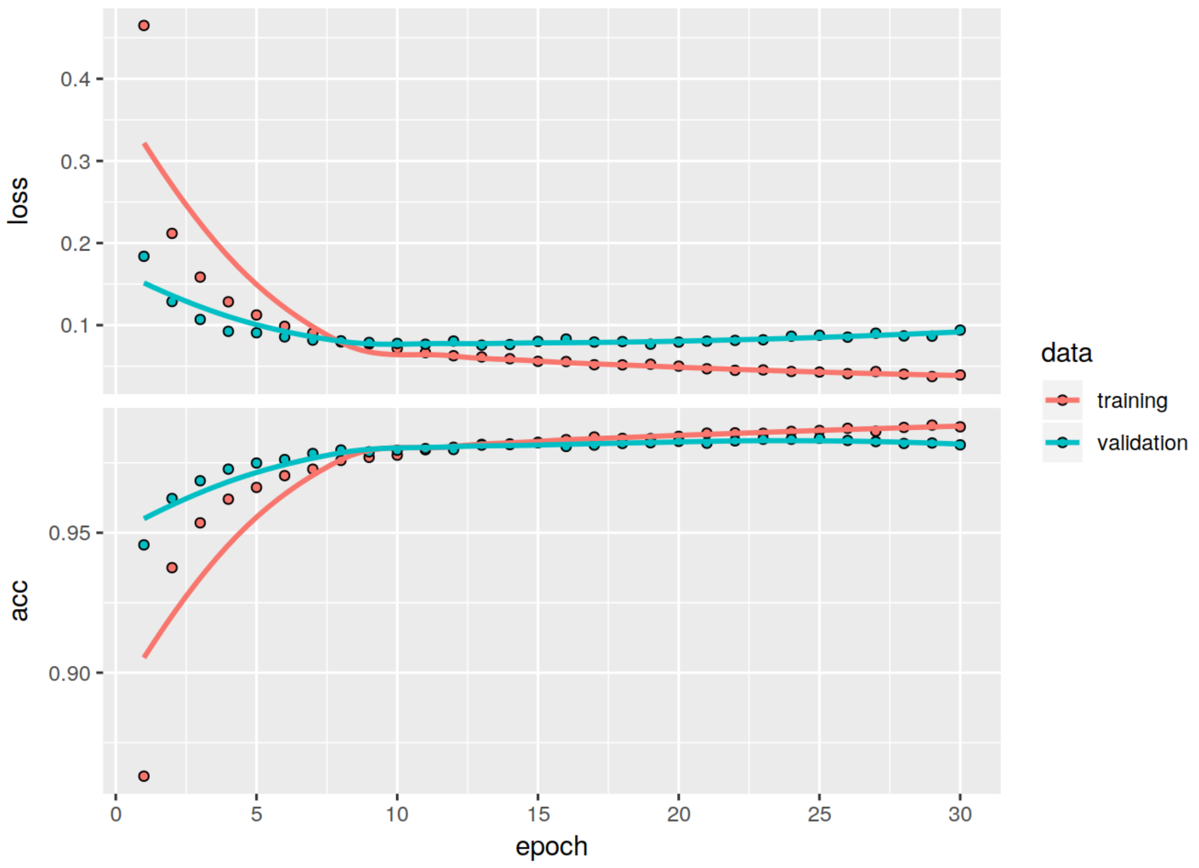
```
## ________________________________________________________________________
__
## Layer (type)                     Output Shape                  Param #

## ========================================================================
==
## dense_10 (Dense)                 (None, 256)                   22272

## ________________________________________________________________________
__
## dropout_7 (Dropout)              (None, 256)                   0

## ________________________________________________________________________
__
## dense_11 (Dense)                 (None, 64)                    16448

## ________________________________________________________________________
__
## dropout_8 (Dropout)              (None, 64)                    0

## ________________________________________________________________________
__
## dense_12 (Dense)                 (None, 10)                    650

## ========================================================================
==
## Total params: 39,370
## Trainable params: 39,370
## Non-trainable params: 0
## ________________________________________________________________________
__
```

```
set.seed(7)
history4 <- model4 %>% fit(trainset_x_mat, trainset_y_cat, epochs = 30, batc
h_size = 64, validation_split = 0.2)
```

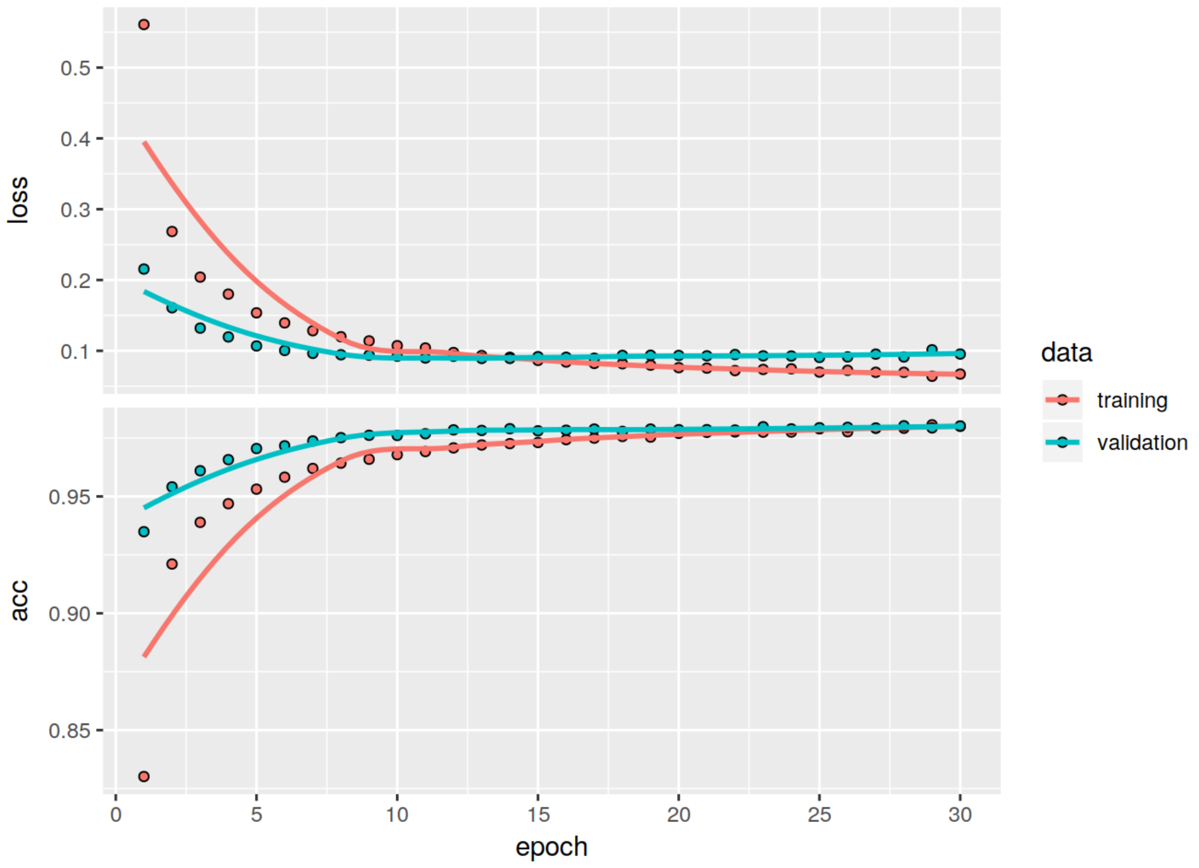
```
# Plot the model's histories to compare
plot(history1)
```



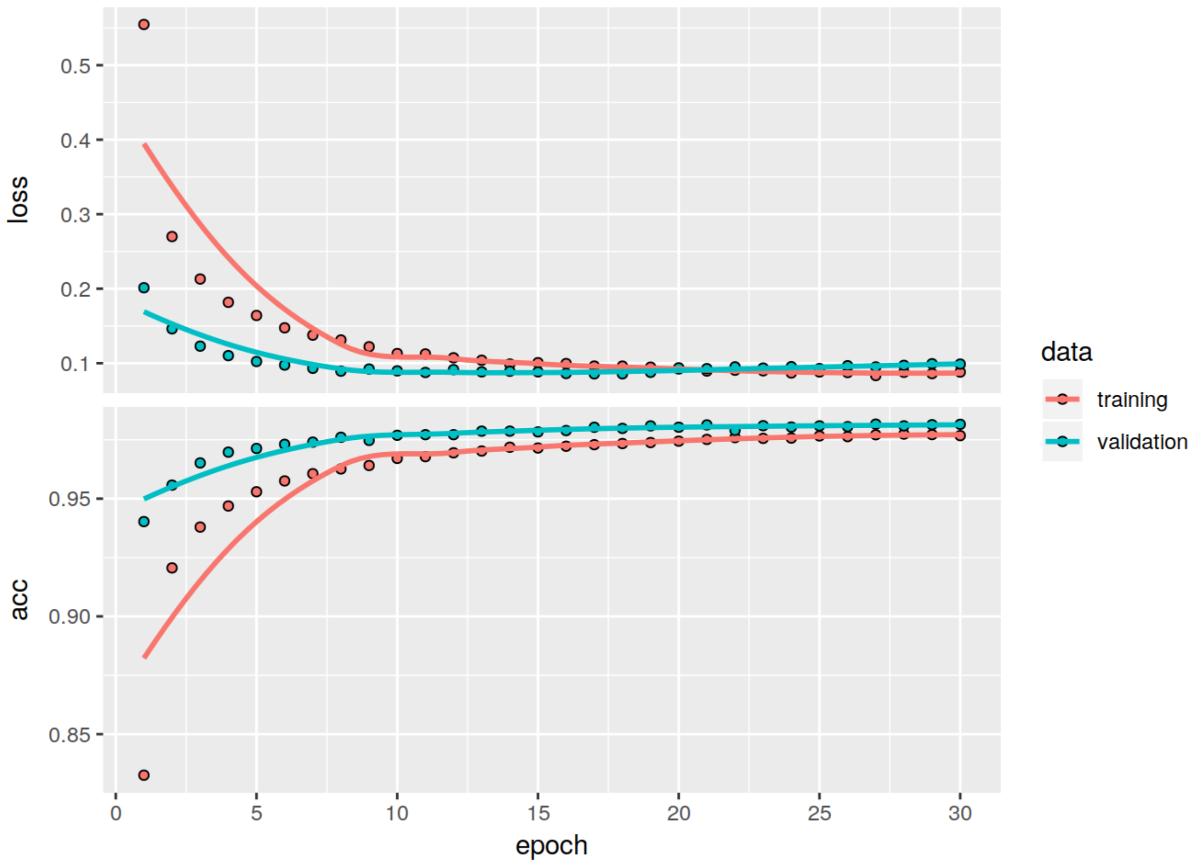
```
plot(history2)
```



```
plot(history3)
```



```
plot(history4)
```



```
# Retrain best-fit NN with all data
model2 <- build_model(n_neurons = c(256, 32), input_shape=c( dim(trainset.pc
a)[2] - 1 ))
set.seed(7)
history2 <- model2 %>% fit(trainset_x_mat, trainset_y_cat, epochs = 30, batc
h_size = 64)
```

5. Evaluate Models on Test Set

The two candidate models, k-NN (k=3) and the best-performing neural network (two hidden layers of 256 and 32 neurons, dropout rate of 0.2) were chosen based upon their cross-validated accuracies. the kNN algorithm manages as accuracy of 0.9737, while the neural network improves on this with 0.9823, a relative performance gain of 32 %. An important property beside accuracy is whether the model performs equally well across all classes. The final confusion matrix indicates that this is the case, meaning the model would not have a bias problem if used “in anger”. An added practical advantage of the neural network is that it is much, much faster to train and apply than the kNN algorithm.

```
# Evaluate the model on the best-performing NN on the test data
model2 %>% evaluate(testset_x_mat, testset_y_cat)
```

```
## $loss
## [1] 0.09936656
##
## $acc
## [1] 0.9823
```

```
# Evaluate the knn model on the test data
# This looks slightly different due to a lack of neat inbuilt evaluation fun
ctions like what Keras has
# fit.knn <- load("knn_model.Rdata")

knn_pred <- predict(fit.knn, testset.pca[,1:86])

Accuracy(y_pred=knn_pred, y_true=testset.pca$y)
```

```
## [1] 0.9737
```

```
# Create a table of class vs prediction for the neural network

nn_pred <- model2 %>% predict_classes(testset_x_mat)
nn_true <- as.integer( sub("chr", "", testset.pca$y) )

table(Actual=nn_true, Predicted=nn_pred)
```

```
##       Predicted
## Actual    0    1    2    3    4    5    6    7    8    9
##      0  970    0    1    1    0    1    1    0    3    3
##      1    0 1127    2    1    1    0    2    1    1    0
##      2    5    0 1014    3    1    0    2    5    2    0
##      3    1    0    3  997    0    3    0    3    2    1
##      4    1    0    1    0  966    0    5    2    1    6
##      5    2    0    0    8    0  872    5    2    3    0
##      6    2    2    1    0    3    5  942    0    3    0
##      7    1    8    9    1    0    0    0 1006    1    2
##      8    5    1    1    4    4    1    3    4  947    4
##      9    2    2    0    3   11    4    0    3    2  982
```

6. Conclusions & Recommendations

The neural network outperforms kNN, but only just. The biggest advantage to the neural network is that it scales well to larger datasets in terms of computational time. On this large-ish dataset and mid-range laptop the neural network can be trained in a much shorter time (minutes) than it takes to cross-validate a kNN model for given k (30 minutes - hour). Comparing the best performance here (accuracy of NN = 0.9827 on test data) to examples found on the MNIST database (http://yann.lecun.com/exdb/mnist/ (http://yann.lecun.com/exdb/mnist/)), it fares well in comparison. kNN has been used to achieve much better results than the PCA + k=3 implementation through the use of non-linear deformations to transform the feature space, getting test error rate of near 0.5 %. Convolutional neural networks, and neural networks in general with more neurons and more sophisticated loss functions, consistently achieve < 1 % test error rates.

If this simple fully connected neural network were further developed a good option might be to try other types of regularisation besides dropout. This is speculative, but as with Convolutional Neural Networks (CNN’s) since the neural network here is modelling a spatial image problem it’s possible that dropping features from the first hidden layer especially makes little sense. If those neurons are learning simple low-level features/correlations that are useful across all classifications then dropping them would rob the next layer of universal, essential features. L1 and L2 regularisation may limit overfitting without this issue.

If an entirely new model were to be developed, then dropping this approach with PCA and switching to a CNN would definitely work better. This is an image classification problem, and CNN’s are specifically structured to tackle such problems. Data augmentation (adding permutations of the same images to the dataset) would be possible, and would deal much more effectively with overfitting than dropout or L1/L2 regularisation.