ADS2 Practical/problem set 29: example solution and notes
Supervised learning: MNIST digit classification

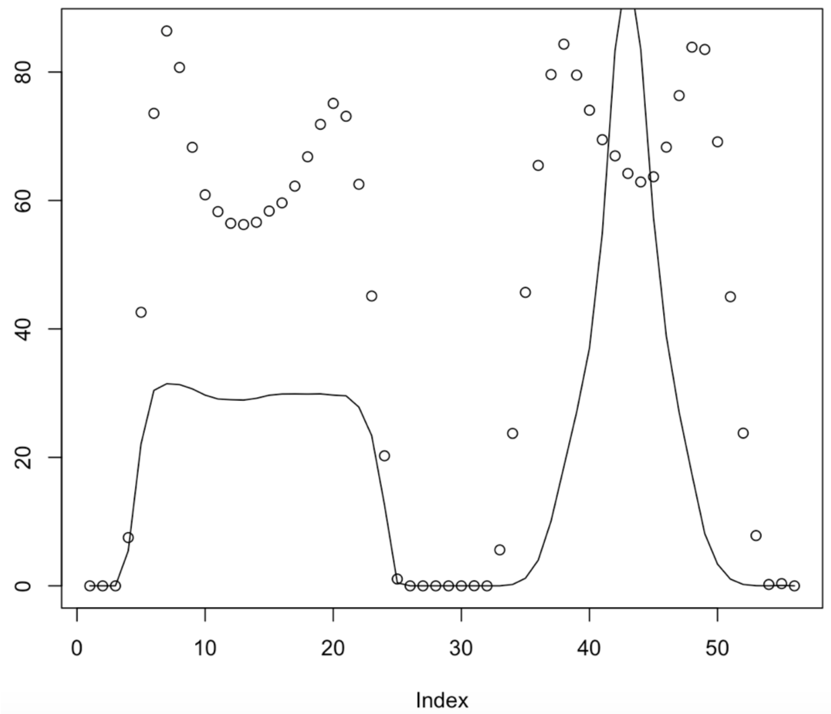
Gedi Lukšys

Training a supervised learning model for digit classification

Now hopefully you have a `features` dataframe with 1000 rows/observations and with the first column (`features$label`) representing class label (0..9) and other columns (e.g. 56 of them, if you choose the features suggested last week) representing features.

Note: computing features may take a while (like minutes or even hours), but once you have them, everything else should be pretty quick! (that's one reason why we use them)

To ensure that your features make sense, first plot their statistics for different labels to see that they seem correct as intended. For example, if you use mean values per row and then per column as your features, their mean values for digit 0 (circles) and digit 1 (lines) should look as follows:



Once you are confident about your features, we can proceed with training a supervised learning model. As we discussed quite a bit about neural networks in the lecture, I would suggest using a neural network, but you are free to use anything else you feel comfortable with. I will provide instructions how to do it with neural networks, but some of them apply to other methods as well. For them, use `library(nnet)`.

First of all, we should normalize our data. This can be achieved for example by dividing all feature values by 255 to ensure that they fall between 0 and 1 (in this way, normalisation will also not be dependent on statistics of the training data, so can also be equally applied to the testing data). The solution below is provided using this normalisation. However, other normalisations, e.g. to a wider range or even no normalisation at all may lead to an even better classifier performance because neural networks often have a fine balance between how high node activations can be and the resulting performance (making them higher can improve performance, but making them too high may lead to instability and crashing).

Once feature values are normalized, we should split our data into training and validation sets, e.g. with 700 examples for training and 300 for validation. It's best to select examples randomly, e.g. with `rows <- sample(1:1000, 700)` (after initialising the random number generator), but then use the same selection throughout your training.

We also have testing data, but normally we don't touch it until we finished adjusting our model parameters (like number of hidden layer neurons in a neural network).

Now we should extract training data and labels. For example, with the format described above it could be done as follows:

```
train_labels <- features[rows, 1]
valid_labels <- features[-rows, 1]
train_data <- features[rows, -1]
valid_data <- features[-rows, -1]

# normalizing the data
train_data = train_data/255
valid_data = valid_data/255
```

Now, as we have 10 classes (for digits from 0 to 9), we need to generate an appropriate output representation for a neural network (and some other methods too) with a N x 10 matrix where the correct class is denoted as 1 and other classes are denoted as 0 (and N is the number of examples, so for example 700 for training data).

This can be done using the following function:

```
train_labels_matrix = class.ind(train_labels)
```

So, for example, the following train_labels

```
> head(train_labels)
[1] 5 0 1 7 7 9
```

will produce the following train_labels_matrix

```
> head(train_labels_matrix)
     0 1 2 3 4 5 6 7 8 9
[1,] 0 0 0 0 0 1 0 0 0 0
[2,] 1 0 0 0 0 0 0 0 0 0
[3,] 0 1 0 0 0 0 0 0 0 0
[4,] 0 0 0 0 0 0 0 1 0 0
[5,] 0 0 0 0 0 0 0 1 0 0
[6,] 0 0 0 0 0 0 0 0 0 1
```

Now we are ready to train our neural network.

We can do this using the following function

```
nn = nnet(train_data, train_labels_matrix, size = 4, softmax = TRUE)
```

It will train a network with 4 hidden units (in one layer), performing 100 iterations by default. You will get a result like the following,

```
# weights:  278
initial  value 1676.371429
iter  10 value 1397.234493
iter  20 value 1086.076845
iter  30 value 866.645778
iter  40 value 697.143178
iter  50 value 615.241827
iter  60 value 568.235269
iter  70 value 538.730898
iter  80 value 521.986354
iter  90 value 507.359435
iter 100 value 488.741360
final  value 488.741360
stopped after 100 iterations
```

It indicates the number of weights (parameters) and development of errors with training.

To use your train model for classifying data, use the predict command. For example,

```
pred_train = predict(nn, train_data, type="class")
pred_valid = predict(nn, valid_data, type="class")
```

will compute predicted classes for training and validation data.

You can calculate your classification accuracy by using

```
mean(pred_train == train_labels)
mean(pred_valid == valid_labels)
```

for training and validation data.

What is the chance level? How does your result compare to it?

As there are 10 digits with roughly similar proportions, the chance level is 0.1. Our result (below) is much better than that, so there is no doubt that our model learns robustly.

Now experiment training your model with a different number of parameters (e.g. 1 to 12 hidden layer neurons). How does classification accuracy for training and validation data change with that? Does it look similar to the bias-variance dilemma plot from the lecture?

Let's use the following code for training neural networks and storing their performance:

```
# initialising vectors for storing training and validation performance
trainerrs = 1:12
validerrs = 1:12
# training the networks for each number of hidden layer neurons
for (i in 1:12)
{ nn = nnet(train_data, train_labels_matrix, size = i, softmax = TRUE)
  pred_train = predict(nn, train_data, type="class")
  pred_valid = predict(nn, valid_data, type="class")
  trainerrs[i] = mean(pred_train == train_labels)
  validerrs[i] = mean(pred_valid == valid_labels)
}
```

The following code is for plotting the results:

```
plot(trainerrs, xlab='Number of hidden layer neurons', ylab='Classification
performance')
lines(validerrs)
```

```
legend ("bottomright", c("training set", "validation set"), pch="o-")
title("Performance of a default neural network with 100 iterations for digit
classification")
```

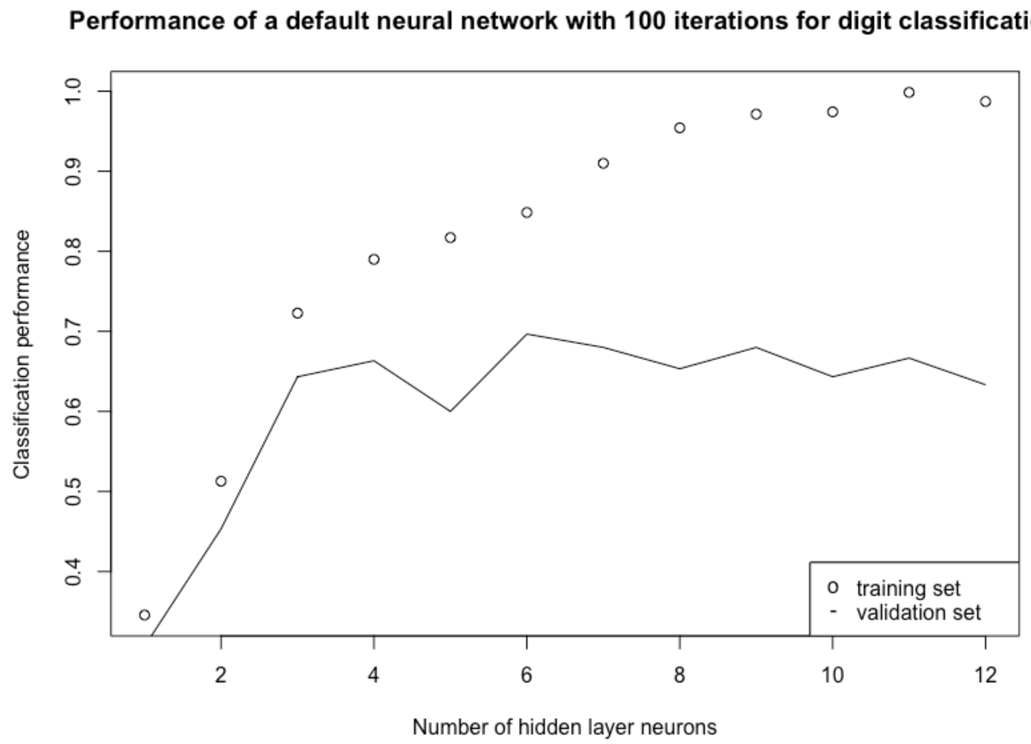
If we look at the development of errors with training below (shown in columns to save space), it's evident that in none of the cases the network has fully converged, as errors keep decreasing notably between the 90th and 100th iterations:

```
# weights:  77
initial  value 1649.792485
iter  10 value 1474.068921
iter  20 value 1300.815290
iter  30 value 1187.673351
iter  40 value 1158.573207
iter  50 value 1152.925674
iter  60 value 1148.102741
iter  70 value 1145.509243
iter  80 value 1142.018383
iter  90 value 1140.114879
iter 100 value 1138.605288
final  value 1138.605288
stopped after 100 iterations
# weights:  144
initial  value 1647.283363
iter  10 value 1396.899959
iter  20 value 1269.823262
iter  30 value 1147.802809
iter  40 value 1073.352538
iter  50 value 1011.069721
iter  60 value 977.153245
iter  70 value 953.519217
iter  80 value 928.588924
iter  90 value 913.080987
iter 100 value 905.386247
final  value 905.386247
stopped after 100 iterations
# weights:  211
initial  value 1708.712794
iter  10 value 1394.142562
iter  20 value 1037.966486
iter  30 value 836.221631
iter  40 value 757.378511
iter  50 value 698.879989
iter  60 value 655.614074
iter  70 value 624.521032
iter  80 value 587.805039
iter  90 value 563.461539
iter 100 value 548.362327
final  value 548.362327
stopped after 100 iterations
# weights:  278
initial  value 1683.494218
iter  10 value 1389.921089
iter  20 value 1086.699061
iter  30 value 856.171839
iter  40 value 743.950019
iter  50 value 657.251907
iter  60 value 583.818564
iter  70 value 548.646966
iter  80 value 519.092658
iter  90 value 478.699067
iter 100 value 423.181382
final  value 423.181382
stopped after 100 iterations
# weights:  345
initial  value 1739.655141
iter  10 value 1320.002394
iter  20 value 1049.552868
iter  30 value 693.583895
iter  40 value 592.646593
iter  50 value 532.903448
iter  60 value 487.063217
iter  70 value 457.084182
iter  80 value 437.019487
iter  90 value 414.851565
iter 100 value 394.036284
final  value 394.036284
stopped after 100 iterations
# weights:  412
initial  value 1759.815439
iter  10 value 1212.769319
iter  20 value 808.668871
iter  30 value 575.857094
iter  40 value 464.171224
iter  50 value 425.114448
iter  60 value 401.806418
iter  70 value 382.109539
iter  80 value 366.957372
iter  90 value 352.093062
iter 100 value 338.243008
final  value 338.243008
stopped after 100 iterations
# weights:  479
initial  value 1628.133518
iter  10 value 1085.919635
iter  20 value 705.366361
iter  30 value 503.992049
iter  40 value 374.647288
iter  50 value 304.770077
iter  60 value 261.114462
iter  70 value 234.404961
iter  80 value 214.248918
iter  90 value 196.002838
iter 100 value 182.148215
final  value 182.148215
stopped after 100 iterations
# weights:  546
initial  value 1825.873882
iter  10 value 1440.116198
iter  20 value 1028.914530
iter  30 value 729.945918
iter  40 value 466.581682
iter  50 value 345.761056
iter  60 value 256.079033
iter  70 value 201.224916
iter  80 value 168.148155
iter  90 value 141.981752
iter 100 value 118.479353
final  value 118.479353
stopped after 100 iterations
# weights:  613
initial  value 1854.220303
iter  10 value 1354.023754
iter  20 value 955.741552
iter  30 value 633.534659
iter  40 value 401.584682
iter  50 value 301.739073
iter  60 value 222.520094
iter  70 value 168.191052
iter  80 value 130.432828
iter  90 value 97.390547
iter 100 value 72.814772
final  value 72.814772
stopped after 100 iterations
# weights:  680
initial  value 1859.765477
iter  10 value 1057.232075
iter  20 value 632.414205
iter  30 value 391.685132
iter  40 value 285.388443
iter  50 value 225.656018
iter  60 value 181.944440
iter  70 value 138.520143
iter  80 value 103.068871
iter  90 value 75.928254
iter 100 value 59.708848
final  value 59.708848
stopped after 100 iterations
# weights:  747
initial  value 1764.812084
iter  10 value 1131.117669
iter  20 value 681.422073
iter  30 value 429.807572
iter  40 value 276.828113
iter  50 value 164.676662
iter  60 value 82.627822
iter  70 value 39.538367
iter  80 value 19.110405
iter  90 value 8.296250
iter 100 value 4.953954
final  value 4.953954
stopped after 100 iterations
# weights:  814
initial  value 1968.096150
iter  10 value 1307.014693
iter  20 value 697.501254
iter  30 value 436.214044
iter  40 value 274.425377
iter  50 value 179.802728
iter  60 value 101.302886
iter  70 value 64.002276
iter  80 value 46.454683
iter  90 value 35.073141
```

`iter 100 value 24.609327          final  value 24.609327          stopped after 100 iterations`

Note that the number of weights is `#HiddenLayerNeurons * (1 + 56 + 10) + 10`. Check lecture slides on neural networks and think why that is the case. ☺

The bias-variance dilemma plot looks as follows:



It resembles theoretical plot on the bias side and on the variance side the training set performance nears perfect as expected. However, the validation errors don't go down noticeably, which might reflect the fact that due to some reason, e.g. insufficient convergence, there is no noticeable overfitting that impairs the performance. However, notably after 3 or 4 hidden layer neurons the performance ceases to go up, suggesting that adding more flexibility to the neural network doesn't help improve its performance (and takes longer/uses more memory). Classification performance in the validation set is between 60 and 70%, which is considerably below top classifiers (that result in 90-95%), but it's reasonable with few simple features and only a small part of the data used.

You can also experiment with other parameters, e.g. number of iterations.

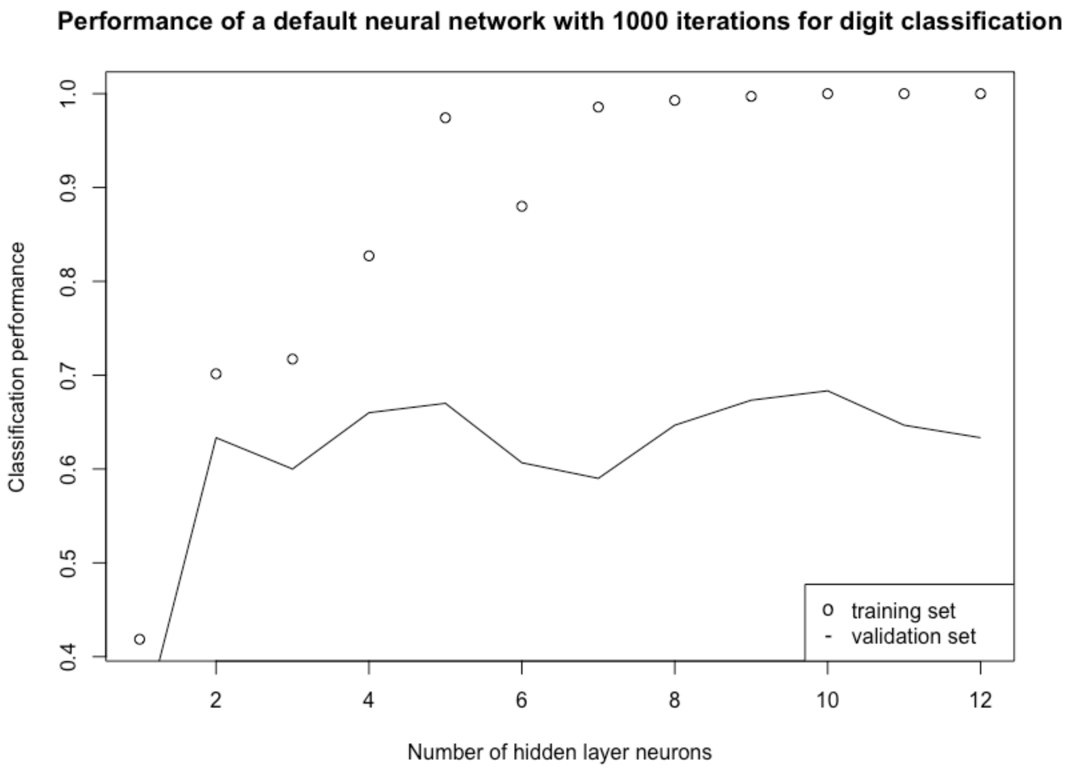
```
nn = nnet(train_data, train_labels_matrix, size = 7, maxit = 1000, softmax = TRUE)
```

will train a neural network with 7 hidden layer neurons and 1000 iterations.

(How) does the convergence of the errors change?

(How) does training and validation classification accuracy change?

With 1000 iterations the network converges nearly always (the output is not plotted here because it would take too much space – try yourself). Classification performance improves for the training set with fewer hidden layer neurons compared to 100 iterations; however, there is essentially no change in the validation set performance: only more noisiness, as the network is more likely to get stuck in local optima, whether they are good or not. Note that noisiness is a general feature of neural networks, as its weight initialization is random, but its degree can be shaped by various parameter settings such as the number of epochs.



If you computed features in several different ways, try to train models for all of them (or at least those you think are good). Discuss how they compare to each other.

Finally, take your best performing model and apply it to the testing data (e.g. again its first 1000 examples). Remember to calculate their features and normalize them in the same way. How does classification performance there compare to training and validation performance?

We choose the number of hidden layer neurons that leads to the best validation performance as our final model. In this case it's a model with 100 iterations and 6 hidden layer neurons, leading to 70% accuracy in the validation set.

We use the following line to train it again:

```
nn = nnet(train_data, train_labels_matrix, size = 6, softmax = TRUE)
```

Now we need to prepare features for the test data. For this use the same code as in part A, except reading from the different file:

```
mnist_raw <- read_csv("mnist_test.csv", col_names = FALSE)
```

Once the features data frame is ready, we convert it to data and labels, and normalize it:

```
test_labels <- features[, 1]
test_data <- features[, -1]
test_data <- test_data/255
```

Finally, we use our trained model to predict classes for new digits and evaluate its performance

```
pred_test <- predict(nn, test_data, type="class")
mean(pred_test == test_labels)
```

which gives us

```
0.626
```

As you can see, the performance is a bit worse than that in the validation set.

Generally, separation into validation and test sets is especially needed when extensive cross validation is performed without sufficient averaging/smoothing where the best validation performance may be better than the one in the test set (so called “meta-overfitting”). It can also be useful in detecting differences in distributions between training+validation and testing sets, which can lead to poorer performance in the test set. Note that in real life, a test set is normally not available, so people usually split their data into 3 parts: training data (usually the biggest), validation data and testing data.

You are also welcome to train your model with the full data set, not just 1000 examples, if your computer is fast enough for that. This may improve your classification performance.

If you tried any other features, played with other parameters or used other models/training methods and/or got substantially different results, please write about it on the discussion board! By no means 60-70% is good performance, but nor was that the goal in this exercise.