

Neuron Spike Detection, Classification and Optimisation – Coursework C

i. Introduction:

This report scrutinises the implementation and performance of two neuron spike classifiers. Neuron spikes were detected using an evaluated spike detection program and subsequently classified into one of 5 neuron classes. This process was first completed on a training dataset, where the noise was relatively minimal, and the exact position of each spike and corresponding class was known. Following the training, evaluation, and optimisation of each classifier and peak detection method, a submission dataset where noise was considerably higher due to patient movement underwent peak detection and classification. Resultant index and class vectors from the noisy submission dataset generated by the optimised classifier were stored in MATLAB vectors as per the training dataset. All code produced for this report is in python and may be found on [GitHub](#).

a. The Data

The data provided consisted of a training and submission datasets. Both contained data vectors with 1440000 samples at 25 kHz sample rate. Spike index and class vectors were provided for the training dataset but needed to be generated by the methods investigated in this report for the submission dataset.

b. Data Split

Classifiers require data to be separated into three sections: training, validation, and test. Training sets are used to fit the model to the training data, validation sets are then used to provide an evaluation of the model and tune hyper parameters looking to improve performance and avoid overfitting, finally the test set is used to provide an evaluation of the final model on unseen data preventing biasing. As submission.mat data was unlabelled, it could not be used for any of the three required datasets. Therefore training.mat was divided into 70% training, 15% validation, and 15% test sets for the Convolutional Neural Network (CNN), and 70% training 30% test for the K Nearest Neighbour (KNN) classifier. Note the validation section of data has been included in the test set for KNN as it is a ‘lazy’ machine learning model and does not require training, all of the data is instead kept and used at run time.

ii. Performance Metrics:

All classification methods in this paper, including peak detection, required performance evaluation. Classical accuracy can be very dangerous when used to assess the performance of classifiers. As the number of correct classifications out of the number of classifications made, on face value this seems a suitable performance metric, its shortfall lies in the ability to hide severe deficiencies in a classifier’s performance. For example, if a classifier must predict whether a patient has cancer or not, if 10 patients require classification and 9 patients have cancer, the model will likely predict all patients to have cancer. This will produce a high 90% accuracy as it was correct 9 of the 10 times, however it is hiding the fact that the model accurately predicts a patient not to have cancer 0% of the time. For this reason, accuracy should never be used as a performance metric where there is a majority data class. There is clear majority data class for peak detection; around three thousand peaks for almost 1.5 million data points. Whilst the class distribution of the training set had a much more even distribution, the same could not be assumed for the unknown class split of the submission data, as such better performance metrics that did not hide the deficiencies of classification models were required.

a. Confusion Matrix

The confusion matrix is the basis for many performance metrics that aim to highlight deficiencies in classification rather than obfuscate them. It is a cross table that records the number of occurrences of each possible classification type: true positive, true negative, false positive, and false negative, which are the building blocks of subsequent performance metrics [1].

- True positive: where an action is predicted to happen, and it happens
- True negative: where an action is predicted not to happen, and it does not happen
- False positive: where an action is predicted to happen, but it does not happen
- False negative: where an action is predicted not to happen, but it does happen

b. Precision

Precision in equation.1 [1] is the proportion of units correctly predicted to be positive out of all positive predictions.

$$Precision = \frac{true\ positive}{true\ positive + false\ positive} \quad (1)$$

Essentially precision defines how much the model can be trusted when predicting an occurrence to be positive.

c. Recall

Recall in equation.2 [1] is the proportion of units correctly predicted to be positive out of all positive units.

$$Recall = \frac{true\ positive}{true\ positive + false\ negative} \quad (2)$$

Essentially recall defines how much the model can be trusted to find all positive units in the data.

d. F1-Score

Precision and Recall are both very useful metrics in assessing the performance of classification, however the optimisation of one without the other can still hide deficiencies in the model. As the desired classification model in this report has both a high precision and recall, a performance metric capable of maximising precision and recall simultaneously was required. The F1-score in equation.3 conveys the balance between precision and recall.

$$F1\ score = 2 \cdot \frac{(precision \cdot recall)}{(precision + recall)} \quad (3)$$

The F1-score uses the concept of harmonic mean to produce a weighted average between the two performance metrics. Unlike conventional a mean, it favours a system with similar precision and recall values, otherwise weighting the lower performing metric higher such that deficiencies in the model are not missed [1].

From the literature reviewed, precision, recall and ultimately F1-score were the chosen performance metrics for evaluation of the classifiers in this report. The peak detection algorithm required a custom python confusion matrix generation, whereas the classifier confusion matrices were produced using the *sklearn.metrics* library.

iii. Peak Detection – Custom Convolutional Peak Detection:

a. Method

The first challenge addressed was the act of detecting and indexing neuron peaks. As a multiclass problem the activation of multiple peaks simultaneously is inevitable, peak detection therefore must be robust enough to detect both standalone peaks, and peaks in close proximity to one another despite potentially large noise in the data stream. A wavelet multi-level decomposition and reconstruction (WMLDR) filter was used to reduce noise. Typically, a Butterworth bandpass filter is used to perform this filtering, however literature

suggests wavelet filtering better preserves spike shape, whilst improving signal-to-noise ratio and cluster discrimination [2]. Preserving the shape of the peaks was vital, as filtering methods which corrupt the subtle morphology differences between classes would greatly reduce later classification performance. The Daubechies(4) wavelet filter to a depth of 6 was implemented using python's *PyWavelets* library. Depth of 6 was chosen as a compromise between higher levels distorting the waveform less with a higher SNR, and lower levels better increasing cluster discrimination [2]. After filtering the data, a custom convolutional peak detection method was implemented. A window was convolved over every point in the data set, and the current central point of the window compared to the mean of the window. Creating a moving mean updated by each window position allowed a peak threshold to be set, such that any datapoint above this threshold would be considered as a potential peak. Initially this method had issues with the false positive detection rate. To combat false summit detection as the window was convolved over the data, a smaller false summit window was implemented to check 10 points ahead and behind the current central datapoint. If the current datapoint was the largest in the false summit prevention window, and more than half of the points either side of peak position were above the peak threshold – it was deemed a peak. This method resulted in peak detection able to disseminate between peaks with maxima at least 10 datapoints or 0.0004 seconds apart.

b. Performance

Performance evaluation was carried out using the performance metrics from part ii yielding the following confusion matrix in table.1:

Table 1 – Convolutional peak detection confusion matrix

Classes	Predicted Positive	Predicted Negative	Total
Actual Positive	3100	243	3343
Actual Negative	243	1436414	1436657
Total	3343	1436657	1440000

The confusion matrix was then used to produce the three main performance metrics in table.2:

Table 2 – Convolutional peak detection performance metrics precision, recall, and F1-score, where performance 1 = best, 0 = worst

Precision	Recall	F1-score
0.927	0.927	0.927

Table.2 shows an equilibrium between precision and recall was reached, both obtaining almost 93% and the F1-score reflecting this. Of the 3343 know peaks in the training data, the peak detection found 3100 true positives, with the same number of false positives and false negatives. An almost 93% F1-score means we can trust the peak detection model to accurately detect peaks within the data stream, with the 7% deficiency constituting mostly of simultaneous neuron firings below the resolution of cluster discrimination. True positive detections were used as inputs to the classifiers.

iv. First Solution – Convolutional Neural Network Classifier:

a. Method

The first machine learning solution implemented was the Convolutional Neural Network (CNN). CNNs operate by convolving a filter window or 'kernel' over the entire input matrix to produce a feature map. The feature map is then passed through a pooling layer – maxpool being the most common – where the largest of the pool is retained to decrease the number of features whilst maintaining their salient features. Several layers of convolution and pooling occur to learn features at different levels of the input before the final dimensions are passed onto an MLP for classification. Python library *Keras* was used to create and implement the CNN classifier. Keras's heavy GPU optimisation meant it was more computationally efficient, allowing the focus of

this report to be the design architecture and hyperparameter selection rather than implementation efficiency. It was decided a 3 block CNN would be used as this may capture large, medium, and small features of the input neuron spike waveforms. Each block consisted of a convolution layer, followed by a max pool layer, and finally a dropout layer. Dropout layers were used to prevent overfitting of the model, by randomly dropping out a proportion of the nodes, making it much harder for the model to fit exactly to the training data. Note dropout layers are not used by Keras when the trained model is used to predict classes on test data. The literature suggests several design rules which must be followed when selecting number of filters, kernel size, and dropout for each convolutional block [3]:

1. The number of feature maps and hence filters should be at least 32 more than the previous convolutional block
2. Kernel size should be equal or smaller in size to the previous convolution block
3. Pooling kernel size should be fixed at 2 throughout, preventing premature reduction in feature map size
4. Dropout should be greater or equal to the previous convolution block, with the first dropout being 0.2 and maximum dropout 0.5

The initial architecture and parameters for the CNN are shown in figure.1, where it can be seen the design rules were followed for convolution, pool, and dropout layers 1, 2, and 3. Filter number begins at 32 and increases in size deeper into the network. Dropout also increased the deeper into the network, whilst kernel size was set as 3 throughout. Convolution 4 however breaks the previously mentioned ruleset. Convolution 4 had been specifically configured to implement channel-wide pooling by setting the kernel size to 1 and filter number to less than the previous convolution. Facilitating a decrease in depth from 512 to 256. The feature map is not the same but has captured the salient features performing dimension reduction and drastically reducing the computational intensity at deep layers. A fifth rule may therefore be defined as:

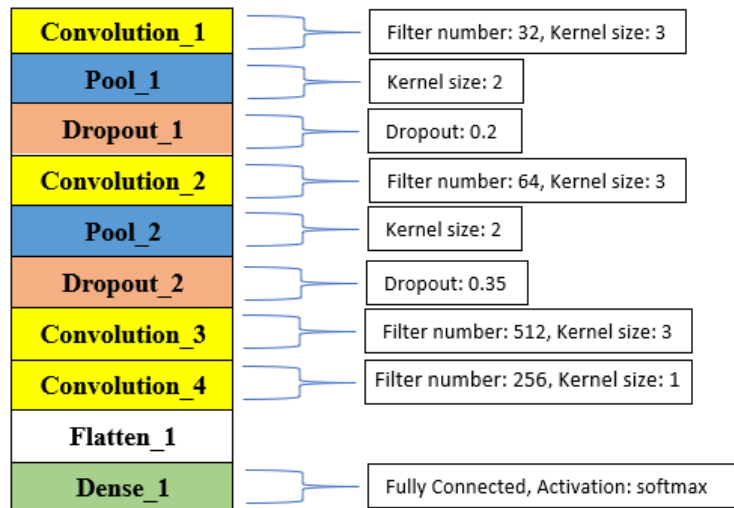


Figure 1 – Initial CNN architecture and selected hyperparameters

5. Convolution_4 must have constant kernel size 1, and filter number at least 32 less than the previous convolution layer

b. Performance

The initial network was trained using a batch size of 128 over 100 epochs using early stopping dependant on the difference between training and validation loss. Where loss was defined as the categorical cross entropy between the one-hot-encoded label and prediction. The model started to overfit after 42 epochs, so training was halted. Figure.2 shows the CNN loss history during training and validation (named 'test' in the figure). After the model completed training and evaluation, it's classification performance metrics shown in table.3 were calculated for each of the five classes in the test dataset.

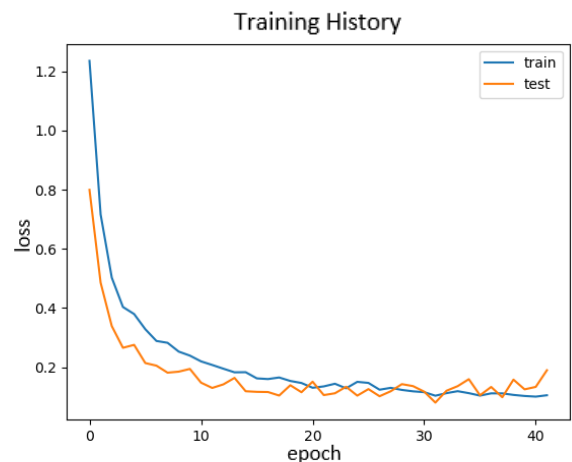


Figure 2 – CNN training and validation loss history with early stopping occurring after 42 epochs to prevent overfitting.

Table 3 – Performance metrics of trained CNN classifier for each of the 5 neuron spike classes

Class	Precision	Recall	F1-score
1	0.990	0.980	0.985
2	1.000	0.955	0.977
3	0.947	0.973	0.959
4	0.823	0.982	0.895
5	0.961	0.779	0.861

It is clearly seen that the classifier performed well with class 1, 2, and 3 neuron spikes, in comparison to class 4 and 5 spikes which were commonly misclassified. This would suggest the morphology of class 4 and 5 spikes to be very similar, and the hyperparameters not sufficiently optimised to confidently discern between the two. Plotting 10 of the correctly classified waveforms for class 4 and 5 in figure 3 and 4 respectively confirms their similarity, and increased classification difficulty.



Figure 3 – Training Dataset 10 correctly classified class 4 waveforms



Figure 4 – Training Dataset 10 correctly classified class 5 waveforms

v. Second Solution – K Nearest Neighbour Classifier:

a. Method

The second machine learning solution implemented was the K-Nearest Neighbour (KNN) classifier. Unlike the CNNs which are ‘eager learners’, KNN is a ‘lazy’ algorithm, not because it inherently performs worse, but because rather than training, the algorithm stores all the training data. Therefore, KNN is non-parametric as it has little to no assumptions made about the mapping function, until it is required to produce a prediction, at which point it uses the K-nearest training points to predict the output. This approach is starkly different to that of the CNN, and why it was chosen as the second method. As no training takes place before the prediction stage dimensionality drastically impacts the computational time required to place all training points on the problem state space, it was chosen to use principal component analysis (PCA) to first reduce the number of dimension. Both PCA and the KNN algorithm were implemented using *sklearn* python libraries. PCA works to re-structure the initial variables by way of linear combination, such that the maximum possible disseminating

information is placed in the first components, with less and less variance being explained by the later components. This allowed the highly dimensional waveform data to be converted to much fewer dimensions whilst retaining almost all variance. The first 6 components in figure.5 were found to capture 95% of the variance, and so were used for KNN greatly reducing the curse of dimensionality when placing points on the problem space.

Only K value and distance metrics can be changed for the algorithm. Smaller values of K provide better fit to the data but run the risk of poor outlier rejection. K must also be odd to stop an equal split of neighbours preventing classification. A value of 5 was decided suitably large to provide some resilience to outliers, but small enough to fit tightly to the class boundaries in the problem space. There are several distance metrics commonly used for KNN, of which the most popular are Euclidean and Manhattan distance. As the name suggest Manhattan distance calculates the distance between two points in an 'x,y' grid pattern, similar to the famous streets, this limits the metric to two natural dimensions and so is less suitable for the 6 PCA dimensions being fed into the classifier. Euclidean distance is easily applied to 'n' dimensions to generate the distance between two points, so selected for use.

b. Performance

Performance of the KNN classifier was evaluated using the same metrics as the CNN such that their performance could be compared. Precision, recall and subsequent F1-scores for each class are displayed in table.4 below.

Table 4 – performance metrics of KNN classifier using K=5 and Euclidean distance

Class	Precision	Recall	F1-score
1	0.981	0.972	0.976
2	0.962	0.978	0.970
3	0.969	0.981	0.975
4	0.979	0.969	0.974
5	0.978	0.973	0.975

The F1-scores were high and much more consistent between classes compared to the CNN, achieving at least 97% in all classes. Most notable difference in performance seen in class 4 and 5 classification, providing performance almost 10% above that of the CNN classifier. This was likely down to the robustness of KNN compared to the CNN, as a very simple network with only three variable parameters: number of PCA components, K, and distance metric, compared to the CNN with at least 10 hyperparameters needing optimisation. It is likely then that this disparity in performance is not due to KNN being an inherently better classifier, but that tolerance in the limited hyperparameters is much higher, and as such the chosen initial values perform better than the chosen initial values of the CNN.

vi. Optimisation Approach – Simulated Annealing of CNN classifier:

a. Method

Comparison of the two classifiers showed both methods could produce high F1-scores and trustworthy classifications. The CNN produced an average F1-score of 93.5% with poor performance on classes 4 and 5, whereas KNN produced an average F1-score of 97.4% with consistent performance across the classes.

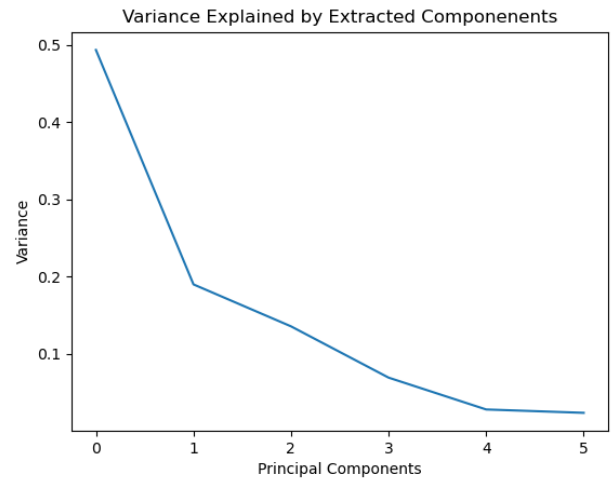


Figure 5 – Variance explained by the first 6 components after PCA for use with KNN classifier

Previously hypothesising that poor optimisation of the CNN hyper parameters was to blame for the inconsistent class performance, combined with worse performance than the KNN classifier, it was the perfect candidate for optimisation. Genetic algorithms (GA) and Simulated annealing (SA) are the two most widespread optimisation methods used in machine learning. GAs are population based with several or hundreds of solutions depending on population size, SA starts with a single solution and works to enhance it to produce one solution. As the time needed to train evaluate and test each CNN was nontrivial, simulated annealing was chosen as the optimisation method avoided the vast computational expense needed to train multiple CNNs per generation with GAs.

Simulated annealing is a probabilistic technique for approximating the global optimum and is a form of guided random search. It is not totally random as the algorithm produces a neighbouring solution and based on the neighbour's cost and the current temperature, the solution is either accepted or rejected. This means better moves are always accepted, but worse move are also sometimes accepted, with a probability decreasing in relation to temperature. This ensures more of the problem space can be explored to reduce the likelihood of getting stuck in local minima. SA required two parameters, and two application specific functions to be chosen, alpha was set to 0.80, decreasing the temperature by 80% each iteration, and the number of iterations at each temperature set to 5. The SA was supplied with the filter number, kernel size, and dropout rates from the non-optimised CNN.

The first specific function was the cost function, this was used to compute the cost or performance of a solution and compare it to the performance of a randomly selected neighbour so the better may be retained. The chosen cost function trained the CNN using the currently selected weights, and calculated the F1-score for each class, the RMS error between the calculated F1-scores and a demand matrix of all F1-scores equal to one was returned as solution cost. This cost function rewards high balanced F1-scores across all classes, the main issue previously faced by the non-optimised CNN.

The second specific function was the neighbour function which when passed the current solution returned a random neighbour solution. To achieve this a random CNN hyperparameter was selected, and then randomly increased or decreased in value to the closest neighbour. Complication arises due to the architectural design rules outlined in section *iv. a*. Allowing a single hyperparameter to be changed to a neighbouring position without ensuring design rules were still adhered to would exponentially open up the search space and allow non-valid designs to be explored. To prevent excess computational intensity and the possibility for non-valid architectures being produced, after the single random hyperparameter was changed, the remaining hyperparameters would be assessed and changed where necessary such that the neighbour as a whole met the required design rules.

Simulated annealing was carried out on the initial CNN hyperparameter values in figure.1 Accepted solution history and their costs in figure.6 clearly shows the nature of SA to accept both worse and better solutions during its guided search. Early stopping was implemented when a solution had a cost value less than 0.01 to avoid wasting excess computational resources and resulted in figure.7s optimised hyperparameters.

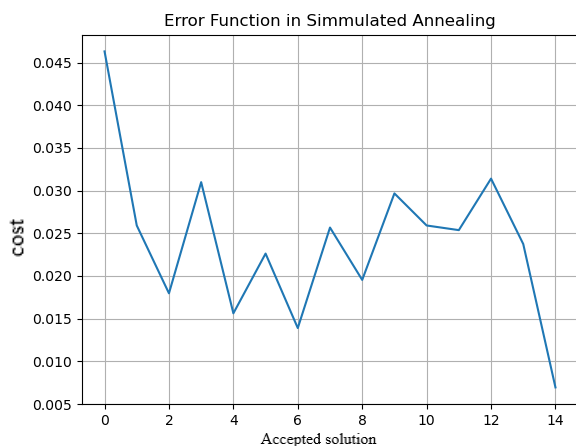


Figure 6 – Cost history of accepted solutions in SA history

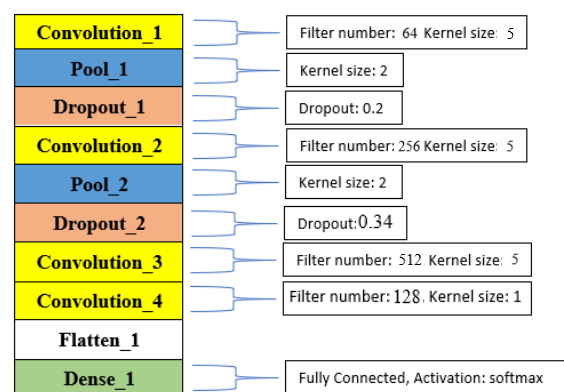


Figure 7 – Optimised CNN hyperparameters after SA

b. Performance

The optimised model's classification performance metrics shown in table.5 were calculated for each of the five classes in the test dataset.

Table 5 – Optimised CNN performance metrics

Class	Precision	Recall	F1-score
1	0.990	1.000	0.995
2	1.000	1.000	1.000
3	0.986	0.986	0.986
4	1.000	0.991	0.995
5	0.989	0.989	0.989

The optimised classifier produces exceptionally high levels of classification performance across all classes, improvements compared to the non-optimised hyperparameters are shown in figures 8, 9, 10, 11, and 12.

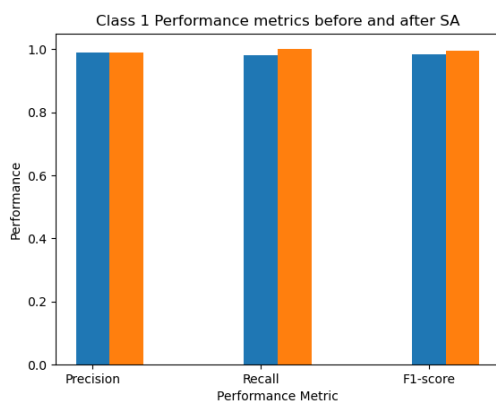


Figure 8 – Class 1 performance metric gain from SA

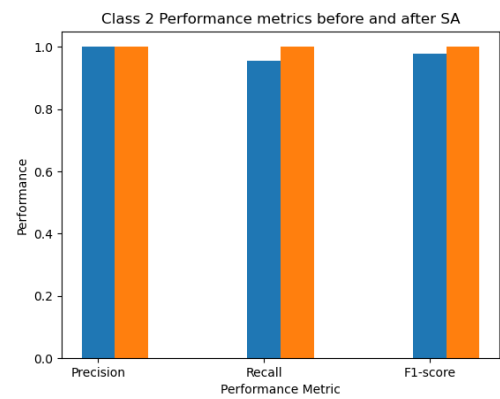


Figure 9 - Class 2 performance metric gain from SA

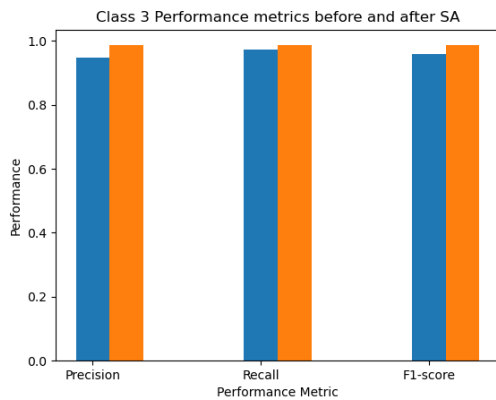


Figure 10 - Class 3 performance metric gain from SA

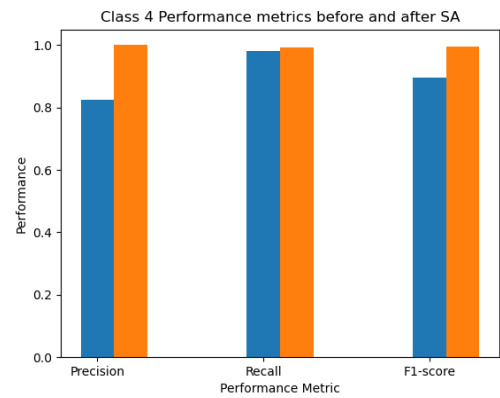



Figure 11 - Class 4 performance metric gain from SA

 = Non-optimised

 = Optimised

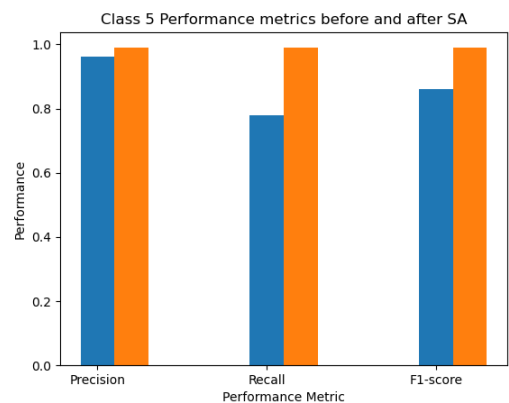


Figure 12 - Class 5 performance metric gain from SA

As hypothesised the largest performance gains from optimisation occurred in class 4 and 5 classification: experiencing over 10% improvement in F1-scores. The average non-optimised F1-score increased from 93.5% to 99.3% suggesting the optimisation was a success.

vii. Conclusion & Confidence Level:

a. Confidence Level

The convolutional peak detector and optimised CNN model were used to produce the final submission dataset neuron index and class vectors. As the index and class of the peaks within the submission dataset were unknown, performance metrics could not be calculated and instead a confidence level must be obtained. Visual inspection of the classified submission waveforms against known correctly classified training waveforms was used to gauge confidence of the submission data classifications. Ten randomly selected waveforms for each class were plotted in figures 13, 14, 15, 16, and 17.

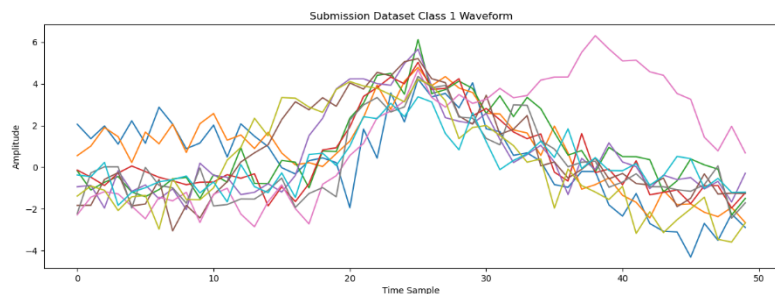


Figure 13 - Class 1 classified peaks - training (left), submission (right)

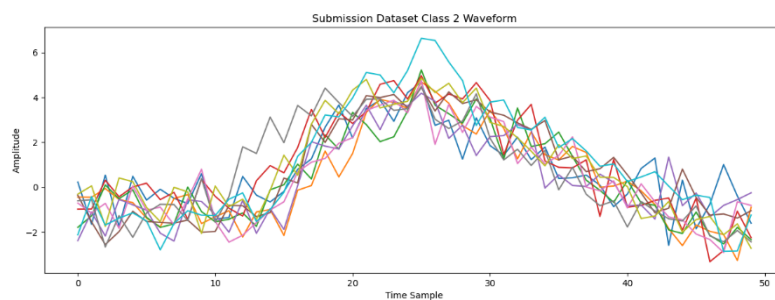


Figure 14 - Class 2 classified peaks - training (left), submission (right)

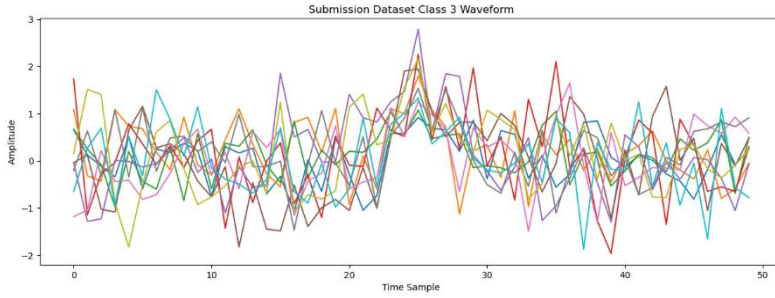


Figure 15 - Class 3 classified peaks - training (left), submission (right)

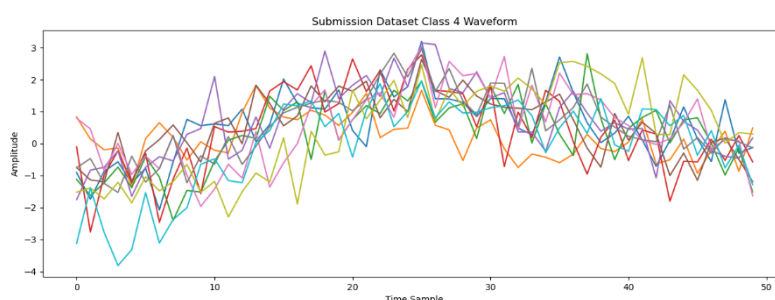


Figure 16 - Class 4 classified peaks - training (left), submission (right)

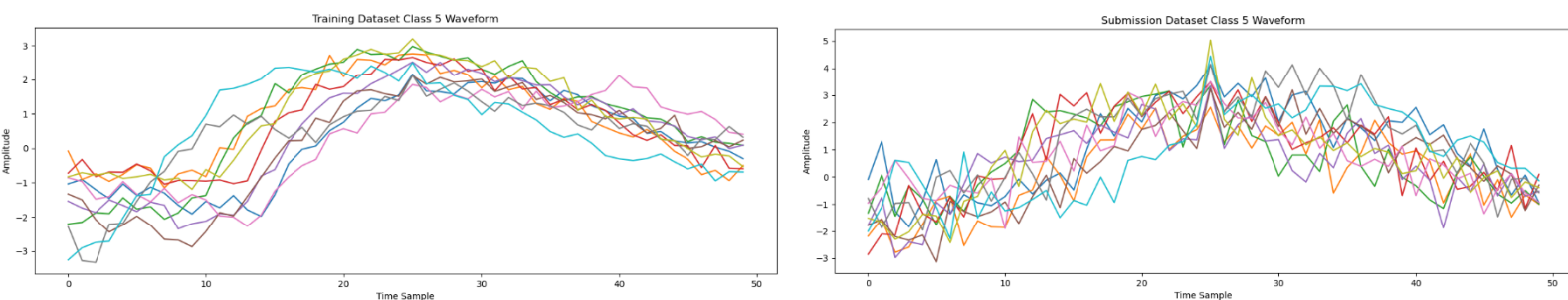


Figure 17 - Class 5 classified peaks - training (left), submission (right)

Visual comparison between 10 random training and submission waveforms for each class, showed it can be confidently assumed classification was successful for the submission dataset. Noise was clearly a larger issue on the submission dataset, but submission waveform morphology did represent those of corresponding training class waveforms in both amplitude and shape. Submission class 3 was most affected by noise, its low peak amplitude of 1.5 presented a challenging SNR for the submission dataset. However, the submission waveforms in figure.15 (right) did resemble the same morphology as the training waveforms (left), as noise in the submission peak reduced for the same width as the training peak, centred around the 25th time sample. Amplitude also rose to that of the known class waveform.

b. Conclusion

This report has clearly shown both CNN and KNN classification models are capable of producing high F1-scores and classifying neuron spikes to high precision. KNN was much more forgiving in hyperparameter selection, producing consistent 97% F1-scores for each class of neuron, whereas CNN initially performed worse with an average F1-score of 93.5% which varied greatly between classes. As an 'eager learner' it was hypothesised optimisation of the CNN hyperparameters would drastically improve the model's performance and make it more consistent between classes. Simulated annealing was proven to be a strong efficient means of optimisation, successfully optimising the CNNs hyperparameters to an average F1-score of 99.3%, with great consistency between classes. This optimised performance confirmed the hypothesis that neither network is inherently a better performer, but that CNNs do require more time and optimisation to reach the higher performances. Finally, the index and class vectors for the submission data were produced, noise did make waveforms harder to distinguish in the submission dataset, but a high confidence level was attained via visual inspection between known class waveforms and randomly selected submission class waveform morphologies.

References

- [1] M. Grandini, E. Bagli and G. Visani, "Metrics for Multi-Class Classification: an Overview," *A WHITE PAPER*, vol. 1, no. 1, pp. 1-17, 2020.
- [2] A. B. Wiltchko, G. J. Gage and J. D. Berke, "Wavelet Filtering before Spike Detection Preserves Waveform Shape and Enhances Single-Unit Discrimination," *J Neurosci Methods*, vol. 172, no. 1, pp. 34-40, 2008.
- [3] A. GÜLCÜ and Z. KUŞ, "Hyper-Parameter Selection in Convolutional," *IEEE Access*, vol. 8, no. 1, pp. 52528-52540, 2020.