Introduction to Neural Networks

Open Assessment, 2018/2019

Pedro Henrique Machado Wigderowitz

Special acknowledgment to Dr. Miguel Patrício, without whom it would not have been possible to complete this report to the appropriate standard.

0. Preface

The implementation was entirely carried out in Python with Keras under Windows 10 (64-bit). See appendix (section 5) for details about dependencies, and information about obtaining source code and other files referenced throughout the report. All measurements shown in this report were rounded to 4 decimal places.

1. Discussion of data and architectures

Our data, the Breast Cancer Coimbra Dataset (BCCD), first introduced in [1], is comprised of 9 columns (ignoring the class label) and 116 rows. Hence, its dimension (or shape) is 116 by 9. Each column presents the measurement of a different component, all of which are obtainable through routine blood analysis [1]. Abnormalities in the measurement of these components (hitherto referred to as variables) have been found to be related to obesity-associated breast cancer [2]. Within our data, we see that a tenth variable has been artificially inserted, indicating whether a sample comes from someone healthy, or someone diagnosed with breast cancer. Healthy controls are comprised of 52 entries (44.83%), while cancer patients are comprised of 64 entries (55.17%).

After analysing the data, it is trivial to see that it poses the task of binary classification. To tackle this task, there is a variety of classes of neural network (NN) architectures which may be suitable.

Multilayer perceptron (MLP): MLPs are one of the first and simplest forms of an NN [3]. In literature, they have found great success, depending on the specific architecture as well as on available data [4]. One advantage of MLPs is that they can theoretically solve any binary classification problem [5], though finding such a solution may not be practically feasible. Furthermore, there is some difficulty in finding weights capable of solving the task at hand, though that can be mitigated through backpropagation [4], [6], [7].

Radial basis function network (RBFN): RBFNs are currently not a very popular research topic, and most breakthroughs are not particularly recent [8]. The most notable difficulty found within this architecture class is that performance heavily depends on appropriate selection of centres [9], which is not a trivial task and still an open research problem [10]. When centres are chosen carefully, however, task performance may turn out to be superior than that achieved by using alternative architectures [9], [10]. In addition, RBFNs allow for the transformation of data into higher dimensional spaces, which is desirable as it may aid in separability of classes [9].

Convolutional neural network (CNN): CNNs have found massive success recently due to their unparalleled performance in multiple domains, such as image recognition [11], [12] and natural language processing [13]. Furthermore, CNNs are known to use relatively less pre-processing, which is considered a major advantage [11]. Given their massive success, it is hard to find disadvantages about CNNs, but it is worth noting that they introduce more hyperparameters [11], which can make for a more arduous tuning phase,

potentially adding difficulty in the obtention of good results. In addition, training CNNs can be very computationally expensive depending on the input size [11].

Deep learning (DL): DL is an incredibly trendy area of research at the moment. The distinction between shallow and deep learning networks is not black and white, and there have been networks in literature considered to be deep while presenting as few as 3 hidden layers [14]. It is also worth noting that the concept of deep learning is not limited to any class of architecture [3]. More importantly, recent investigation has evidenced the immense capability of deep learning over both large and small datasets [15]–[17]. While the exact reason behind the efficiency of deep models is still not completely understood, some hypothesise that they can approximate more useful functions [15], [18].

Unfortunately for us, there is no straightforward way to select an architecture best suited to our task; this is an old problem [19] and still an open research topic. Hence, we must make a decision based on available information. Within [1], the figures obtained from the attempts at classification by logistic regression (LR) indicate that the data is not linearly separable. Furthermore, since the kernel of choice for the best performing support vector machine classifiers is RBF (as seen in the attached R script from [1]), building an RBFN to classify the data sounds like a sane choice. However, Keras does not have native support for RBFNs. This fact, alongside the difficulty behind choosing centres, turns our attention to other alternatives. Seeing the success of backpropagated NNs in literature [4] as well as the aforementioned DL, we opt for a deep MLP architecture with backpropagation (DBPNN). While a CNN could be a better choice than an MLP-based architecture, we choose to avoid it because of lack of literature stating its success for the type of data we have at hand. In any case, since our architecture selection will be an iterative process [5], we could always attempt to tackle the task with a CNN in case we achieve mediocre results with an MLP.

2. Creation and application of a neural network

Before the pre-processing of our data, we must first select features. Fortunately for us, all relevant analysis has already been conducted in [1], so we maintain only the *Glucose*, *Resistin*, *Age*, and *BMI* variables.

Before feeding our data as the input to an NN, we must first pre-process it. Since there are a variety of pre-processing methods we could use [20] but no specific mention of pre-processing procedures in [1], we investigate the support vector machine (SVM) sections within the R code from [1], and the R language documentation of the *svm* function [21]. We find that the data is automatically scaled to zero mean and unit variance, typical of pre-processing for SVM classification [22]. The same effect can be achieved through the *scale* function from the *scikit-learn* library [23] (maintaining optional arguments as their defaults). For the purpose of scaling our data, we first separate the *Classification* label from it by splitting it into *x* and *y* arrays of equal length, where entries in *x* maintain all other variables, and entries in *y* maintain just the *Classification* label of the corresponding *x* entry. Then, we can pass *x* as the argument of the *scale* function to obtain our scaled data. See *load_data.py* under the appendix for more details. Given that scaling might not be the ideal type of pre-processing for NNs, we provide performance metrics for unscaled and normalised data for comparison later in this section.

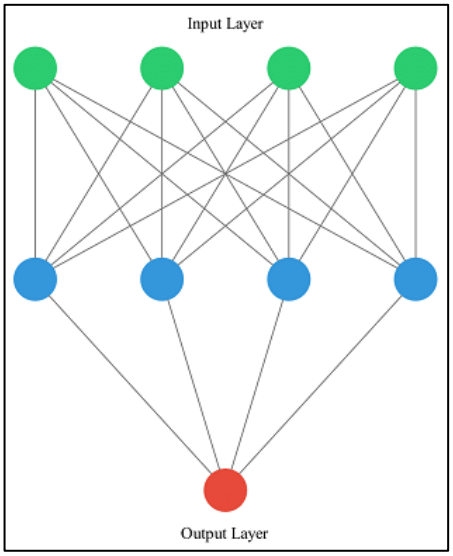

Figure 1: visual architecture of the first prototype of our DBPNN architecture.

To build our DBPNN architecture, we search for the best model in an iterative fashion [5]. As a starting point, we set up our input and output layers, as well as one simple hidden layer: using the Keras sequential model, we add a *Dense* layer with 4 units (arbitrarily defined), input dimension of 4 (since this is how many variables we have), data type of *float32*, and *relu* activation (since it learns faster in networks with many layers [15], which is what we are aiming for).

This adds 2 layers to our model: our 4-dimensional input layer, and a hidden layer with 4 units. Our final layer needs to output a 1-dimensional value which will be the class prediction probability, so we add a sigmoid-activated *Dense* layer with a single unit [5]. The completed first version of the architecture of the model is highlighted in figure 1. We then compile the model with the *adam* optimizer due to its effectiveness [24], and binary cross-entropy loss due to its preferability in binary classification tasks [25]. When fitting our data to the model, we set the amount of epochs to 10 as an initial guess, and the batch size to 10 as larger batches may degrade the quality of the model [26]. Concerning backpropagation, it is turned on by default in Keras [27]. Any other unmentioned hyperparameters are set to Keras' defaults.

The performance of our first prototype is found to be better than expected. The 95% confidence interval (CI) of the area under the receiver operating characteristic curve (ROCAUC) was found to be [0.9177, 0.9651] (details about how this is calculated are discussed under section 3). Unfortunately, after plotting (see *plotting.py*) our loss over the epochs, we find our model to be noticeably overfit; see figure 2.

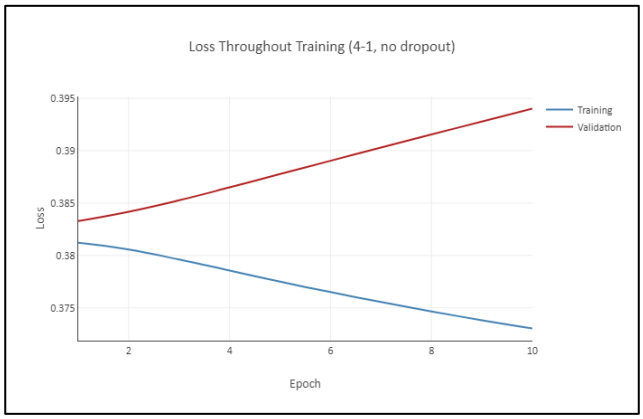

Figure 2: loss as a function of epoch calculated over the first prototype of our DBPNN architecture (see figure 1). Blue is training, red is validation.

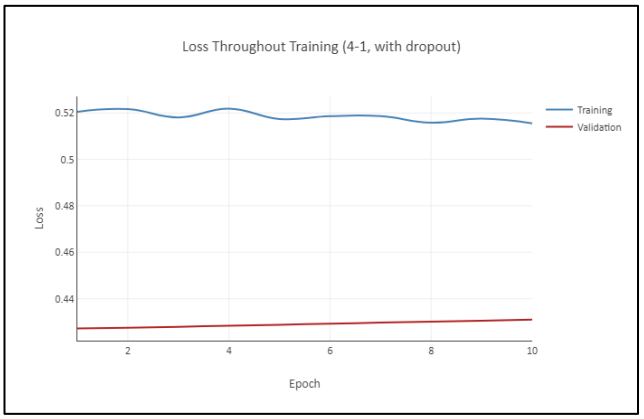

Figure 3: loss as a function of epoch calculated over the first prototype of our DBPNN architecture (see figure 1) with an added dropout layer. Blue is training, red is validation.

We can semi-effortlessly correct our overfitting by regularising through the addition of a dropout layer [11] between the hidden layers and the output layer. The dropout rate of 0.5 is elected as it provides the most regularisation [28]. Inconveniently, before adding dropout, our validation loss at the last epoch is 0.3940, while 0.4310 otherwise. The new CI of the ROCAUC is thus predictably lower: [0.8554, 0.9083]. However, it is prudent to maintain our new regularised architecture, because once our network is deepened, it will tend to overfit even further [11].

Next, we double the number of units in our first and only hidden layer, obtaining a CI of [0.8982, 0.9438]. By doubling our units once again, the new CI is [0.9159, 0.9589]. Since little improvement is found by increasing from 8 to 16 units in comparison to 4 to 8, we may start deepening our model.

To make our model deeper, we repeatedly add more 16-unit *Dense* layers between the first hidden layer and the dropout layer, setting only the *activation* hyperparameter to *relu* and leaving all others defaulted. We do this until our increase in performance starts to flatten, which happens around 6 hidden layers, as evidenced by figure 4. We opt to select the architecture of the model with 5 hidden layers as our final, since it boasts very high performance. A discussion on the results obtained from this model takes place under section 3. The code used throughout this

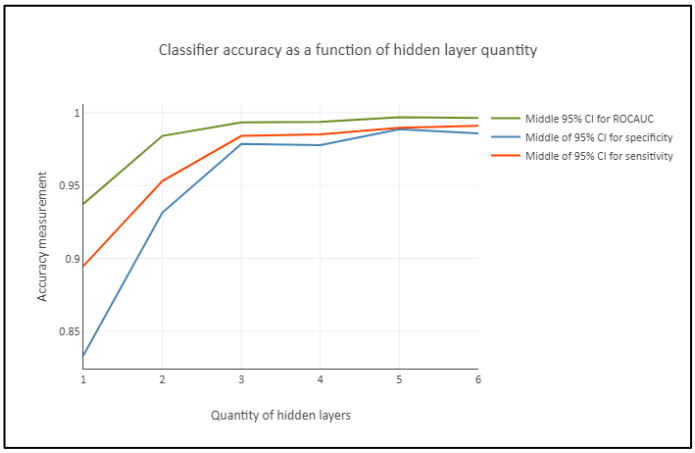

Figure 4: classifier performance as a function of hidden layer amount. For ease of visualisation, the middle value of confidence intervals is used instead of the interval itself. Green is ROCAUC, blue is specificity, and orange is sensitivity.

prototyping phase can be found under *mlp.py*. The visual representation of the final choice of architecture is too large to be inserted inline and can be found under *16(5)-1.pdf*.

Before moving onto the next section, we re-train our selected architecture with raw unscaled data and normalised data (see figure 5). It is clear that scaling the data yields better results in comparison with normalising or not doing any pre-processing (see section 3). Finding literature that explains this phenomenon is difficult, but Sarle [29] suggests that standardising tends to improve the numerical condition of the optimisation problem, thus making the training process better behaved.

	95% CI	
	Raw	**Normalised**
ROCAUC	[0.9432, 0.9817]	[0.9100, 0.9555]
Sensitivity	[0.8246, 0.8870]	[0.8689, 0.9288]
Specificity	[0.8594, 0.9228]	[0.7393, 0.8156]

Figure 5: 95% CI of the ROCAUC, sensitivity, and specificity values of our selected architecture when trained with raw unscaled data, and normalised data.

3. Results and evaluation

Within the paper by Patrício et al. [1], we see that an SVM was trained on some data, the task being binary classification. Performance was evaluated through Monte Carlo Cross-Validation (MCCV) [30] with stratification, achieving the lower bound of 0.87 and upper bound of 0.91 for the 95% CI of the ROCAUC. These methods are a very robust and well thought out; they seem to be very appropriate when it comes to measuring performance of a classifier trained on top of a small dataset. Therefore, it is desirable to replicate them so that we can evaluate the performance of our own NNs. This would also allow us to perform a fair comparison between our NN performance figures against those from [1]. In the interest of guaranteeing beyond reasonable doubt that our replication of the methods is correct, we must reimplement one of the classifiers from [1] and obtain similar figures. On this occasion, we choose to reimplement the 4-variable SVM.

Given that the implementation of [1] was carried out in the R programming language, we must search for Python packages with functions that either facilitate or fully implement the methods above. If no relevant packages are found, we must either interpret the R script from [1] and reimplement relevant functions, or perform an implementation according to some reliable scientific source.

Fortunately, the *scikit-learn* package does offer a ROCAUC scoring function [31], as well as a function for calculating sensitivity and specificity [32]. As for CI, while there is no Python package at the time of writing that offers a plug-and-play function, we can trivially reproduce the same functionality from the R script with the help of the *statistics*, *math*, and *scipy* Python packages (see *confidence_interval.py*). Lastly, concerning MCCV, it is fairly easy to implement the right algorithm by following the visual description from figure 1 in [1]. Finally, in order to check whether all code has been written correctly, we build an SVM and draw predictions upon the data just as described in [1]. If similar results are obtained, it is safe to say that our MCCV and CI implementations are behaving correctly.

As discussed under section 2, we must scale our data with *scale* from *scikit-learn* [23]. Under section 1, we note that RBF is the chosen SVM kernel in [1], so we reproduce that in our code too. We come to a final version of code, which we name *svm.py* (see appendix).

Performance figures obtained after the usual 500 iterations of MCCV can be seen in figure 6. ROCAUC values clearly match those in [1], while sensitivity and specificity do not, the reason being we do not use the specificity-sensitivity pair that maximises Youden's index to calculate the CI, but rather just the values taken from the *classification_report* function [32]. We are not particularly interested in changing our code to cater for the maximisation of Youden's index since we already know that our MCCV, ROCAUC, and CI implementations are working as they should seeing as the SVM ROCAUC CI figure from [1] was accurately reproduced. In any case, we

	95% CI
ROCAUC	[0.8625, 0.9023]
Sensitivity	[0.8083, 0.8585]
Specificity	[0.7586, 0.8139]

Figure 6: SVM figures from Python reimplementation of code from [1].

maintain the calculation of sensitivity and specificity as is, as their values still provide us with information about performance (see figure 4).

	95% CI
ROCAUC	[0.9846, 1.0091]
Sensitivity	[0.9715, 1.0076]
Specificity	[0.9693, 1.0078]

Figure 7: figures from our final architecture selected in section 2.

Figure 7 finally reveals the performance of our selected model from section 2. It is worth noting that while the accuracy of a classifier can obviously not exceed 100%, CIs may present an upper bound value of more than 1.0 due to how they are calculated [1] (see *confidence_interval.py*). We leave this unchanged in order to maintain consistency with [1]. In any case, we can see that our ROCAUC values from our final architecture considerably outclass those from [1]. We can also safely say that sensitivity and specificity are superior even though they are not being calculated in the same way, since our estimations from figure 6 are pessimistic compared to Patrício's [1]. In recent literature, we can also see attempts at the same BCCD binary classification task [33], [34], and our model vastly outperforms those too. Still, seeing as our model does have the potential of misprediction, it cannot be proposed as an alternative to digital mammography, just like in [1]. However, due to its high sensitivity, it could serve as a preliminary low-cost test: during a routine blood test, a prediction could be drawn from the obtained measurements, and in the case of a positive output (cancer diagnosis), the patient could be referred to take a mammography for further investigation, such that the predictions don't have to be taken as absolute truths.

If a model with around 99.9% accuracy could be built, it could then be feasible to suggest that it may serve as a valid alternative to digital mammography, which is obviously something desirable. This could perhaps be achieved through a deep CNN, or most likely if more data was available. It would be more interesting though if our data did not limit us to detecting obesity-related breast cancer, so that we could build a more general model.

4. Further application

This once again poses the task of binary classification to us, but this time, with images as inputs. As discussed under section 1, CNNs are unparalleled when it comes to the task of image classification [11], [12], and deep learning methods can be made powerful as well as general when trained with both small and large datasets [15]–[17], so a good choice of architecture class for this problem would be a deep CNN (DCNN) [35].

Given that we are to assume that there are a large number of mammograms of both classes, we do not need to worry about unbalanced data or robust cross-validation methods [20], as the holdout method tends to suffice for larger data. However, as seen in [35], it is a good idea to apply horizontal and vertical flipping so that the training dataset can be oversampled [20] and thus augmented. We also see in [35] that instead of splitting our data to generate a test set, a better approach might be to use all of it for training/validation, then using an entirely separate dataset for testing, since there are multiple publicly available datasets with mammogram data.

Prior to training, our data would have to be pre-processed. Recent research in image classification with CNNs shows that zero component analysis is an effective pre-processing technique for images [36], so this technique could be employed. Another point is that since the resolution of images from the training data might be relatively high, it is worthwhile downscale them to facilitate training [35], although if hardware is plentiful, it is better to avoid this, since preserving the original image quality may lead to higher performance.

We may want to attempt to select the best DCNN architecture by trial and error [5], but a better approach would be to select a high-performing architecture from literature [37] and alter it according to our needs, slowly iterating towards an ideal model. This is certainly the most complicated part of the task, because the structure of the architecture will determine the ability of the model to detect anomalies in mammograms,

and in addition, the model would have to be able to detect whether these anomalies are benign, malign, or normal. This can only be accomplished with very specific purpose-made architectures. Ribli [35] suggests that regional proposal networks (RPNs) are key for effective lesion finding in this type of task, and goes on to propose the selection of the Faster R-CNN architecture [38].

In terms of evaluation, [35] proposes the usage of ROCAUC as well as a FROC curve [39], which is a great visualisation tool as it shows sensitivity as a function of the number of false positive marks put on an image.

Due to the unchanging nature that benign and malignant lesions are be hard to discern in imagery, [35] presents us with a cancer detection model that overcomes many hurdles, but still fails to deliver very high performance of approximately 99%. Therefore, for the task of building a mammogram classifier, we must exercise care in how we generate our outputs. Since it is particularly important to generate accurate predictions when it comes to cancer diagnosis, it is important not to take classifier outputs as absolute truths, due to the possibility of misclassification (as discussed under section 3). A prudent approach to this would be to have the classifier generate 2 possible classes: "healthy" (0), and "requires further analysis" (1). Data would be classified as 1 if the model outputs a probability of at least 10% for a malignant lesion, and 0 otherwise. Then, patients whose mammograms were classified as 1 could be sent for further analysis by having their imagery seen by a health professional, and those with mammograms classified as 0 could be considered healthy.

Hopefully, after enough advancements in NN research, image classifiers will eventually have virtually perfect performance, greatly reducing human error, which is particularly important for medical diagnoses.

5. Appendix

5.1. Dependencies

Within this section, we list all dependencies as well as their versions, for more ease of reproducibility. The Python version throughout the implementation was 3.6.8.

Python package	Version
Keras	2.2.4
tensorflow-gpu	1.12.0
scikit-learn	0.20.3
scipy	1.2.1
numpy	1.16.2
plotly	3.7.1
ann_visualizer	2.5
graphviz	0.10.1
tqdm (optional)	4.31.1

Figure 8: Python dependencies.

Software	Version
NVIDIA GPU drivers	418.81
CUDA Toolkit	9.0
cuDNN SDK	7.4.2
Visual Studio	2017
Graphviz	2.38

Figure 9: external dependencies.

All Keras models were trained using a Gigabyte GeForce GTX 1050 Ti OC Low Profile NVIDIA GPU (compute capability of 6.1). On no occasion did training take longer than 5 minutes.

5.1 Referenced files

All files referenced throughout this report, including Python source code, can be found under the *appendix* folder, located under the root folder of this report's file.

6. References

[1] M. Patrício *et al.*, "Using Resistin, glucose, age and BMI to predict the presence of breast cancer," *BMC Cancer*, vol. 18, no. 1, pp. 1–8, 2018.

[2] J. Crisóstomo *et al.*, “Hyperresistinemia and metabolic dysregulation: a risky crosstalk in obese breast cancer,” *Endocrine*, vol. 53, no. 2, pp. 433–442, Aug. 2016.

[3] J. Schmidhuber, “Deep Learning in neural networks: An overview,” 2015.

[4] P. Jeatrakul and K. W. Wong, “Comparing the performance of different neural networks for binary classification problems,” in *Symposium on Natural Language Processing*, 2009, pp. 111–115.

[5] S. O’Keefe, “Solving more complex problems,” *INNS*. University of York, 2018.

[6] S. O’Keefe, “Learning with an MLP,” *INNS*. University of York, 2018.

[7] A. Schmidt, “A Modular Neural Network Architecture with Additional Generalization Abilities for High Dimensional Input Vectors,” Manchester Metropolitan University, 2000.

[8] “radial basis function network - Google Scholar.” [Online]. Available: https://scholar.google.com/scholar?q=radial+basis+function+network. [Accessed: 16-Apr-2019].

[9] S. O’Keefe, “Radial Basis Function Networks,” *INNS*. University of York, 2018.

[10] D. Gong, C. Wei, L. Wang, L. Feng, and L. Wang, “Adaptive methods for center choosing of radial basis function interpolation: A review,” *Lect. Notes Comput. Sci.*, vol. 6377 LNCS, no. M4D, pp. 573–580, 2010.

[11] S. O’Keefe, “Convolutional Networks,” *INNS*. University of York, 2018.

[12] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet Classification with Deep Convolutional Neural Networks,” in *Neural Information Processing Systems*, 2012, pp. 1097–1105.

[13] Y. Kim, “Convolutional Neural Networks for Sentence Classification,” in *Empirical Methods in Natural Language Processing*, 2014, pp. 1746–1751.

[14] G. E. Hinton, S. Osindero, and Y.-W. Teh, “A Fast Learning Algorithm for Deep Belief Nets,” *Neural Comput.*, vol. 18, no. 7, pp. 1527–1554, 2006.

[15] S. O’Keefe, “Deep Neural Networks,” *INNS*. University of York, 2018.

[16] I. Goodfellow, J. Bengio, and A. Courville, *Deep Learning*, Web. MIT Press, 2016.

[17] M. Olson, A. Wyner, and R. Berk, “Modern Neural Networks Generalize on Small Data Sets,” in *Neural Information Processing Systems*, 2018, pp. 3623–3632.

[18] H. W. Lin, M. Tegmark, and D. Rolnick, “Why Does Deep and Cheap Learning Work So Well?,” *J. Stat. Phys.*, vol. 168, no. 6, pp. 1223–1247, Sep. 2017.

[19] T. G. Dietterich, “Approximate Statistical Tests for Comparing Supervised Classification Learning Algorithms,” *Neural Comput.*, vol. 10, no. 7, pp. 1895–1924, 1998.

[20] S. O’Keefe, “Handling data,” *INNS*. University of York, 2018.

[21] “svm function | R Documentation.” [Online]. Available: https://www.rdocumentation.org/packages/e1071/versions/1.7-1/topics/svm. [Accessed: 13-Apr-2019].

[22] C. W. Hsu, C. C. Chang, and C. J. Lin, “A Practical Guide to Support Vector Classification,” 2003.

[23] “sklearn.preprocessing.scale — scikit-learn 0.20.3 documentation.” [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.scale.html. [Accessed: 13-Apr-2019].

[24] D. P. Kingma and J. Ba, “Adam: A Method for Stochastic Optimization,” in *International Conference for Learning Representations*, 2014.

[25] J. D. McCaffrey, “Why You Should Use Cross-Entropy Error Instead Of Classification Error Or Mean Squared Error For Neural Network Classifier Training,” 2013. [Online]. Available: https://jamesmccaffrey.wordpress.com/2013/11/05/why-you-should-use-cross-entropy-error-instead-of-classification-error-or-mean-squared-error-for-neural-network-classifier-training/. [Accessed: 17-Apr-2019].

[26] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang, “On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima,” in *International Conference on Learning Representations*, 2017.

[27] “Backpropagation in Neural Networks: Process, Example & Code - :” [Online]. Available: https://missinglink.ai/guides/neural-network-concepts/backpropagation-neural-networks-process-examples-code-minus-math/. [Accessed: 16-Apr-2019].

[28] P. Baldi and P. J. Sadowski, “Understanding Dropout,” in *Neural Information Processing Systems*, 2013, pp. 2814–2822.

[29] W. S. Sarle, “III-Conditioning in Neural Networks,” Cary, NC, 1999.

[30] W. Dubitzky, M. Granzow, and D. Berrar, *Fundamentals of data mining in genomics and proteomics*. Springer Science & Business Media, 2007.

[31] “sklearn.metrics.roc_auc_score — scikit-learn 0.20.3 documentation.” [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.roc_auc_score.html. [Accessed: 13-Apr-2019].

[32] “sklearn.metrics.classification_report — scikit-learn 0.20.3 documentation.” [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.classification_report.html. [Accessed: 17-Apr-2019].

[33] M. Fatih Aslan, Y. Celik, K. Sabanci, and A. Durdu, “Breast Cancer Diagnosis by Different Machine Learning Methods Using Blood Analysis Data,” *Intell. Syst. Appl. Eng.*, vol. 6, no. 4, pp. 289–293, 2018.

[34] Y. Li and Z. Chen, “Performance Evaluation of Machine Learning Methods for Breast Cancer Prediction,” *Appl. Comput. Math.*, vol. 7, no. 4, pp. 212–216, 2018.

[35] D. Ribli, A. Horváth, Z. Unger, P. Pollner, and I. Csabai, “Detecting and classifying lesions in mammograms with Deep Learning,” *Sci. Rep.*, vol. 8, no. 1, p. 4165, Dec. 2018.

[36] K. K. Pal and K. S. Sudeep, “Preprocessing for image classification by convolutional neural networks,” in *Recent Trends in Electronics, Information & Communication Technology*, 2016, pp. 1778–1781.

[37] W. Rawat and Z. Wang, “Deep Convolutional Neural Networks for Image Classification: A Comprehensive Review,” *Neural Comput.*, vol. 29, no. 9, pp. 2352–2449, Sep. 2017.

[38] S. Ren, K. He, R. Girshick, and J. Sun, “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks,” Jun. 2015.

[39] P. C. Bunch, J. F. Hamilton, G. K. Sanderson, and A. H. Simmons, “A Free Response Approach To The Measurement And Characterization Of Radiographic Observer Performance,” in *Application of Optical Instrumentation in Medicine*, 1977, vol. 0127, pp. 124–135.