CITY UNIVERSITY
LONDON

Programming and Mathematics for Artificial Intelligence

Alessandro Abati & Julian Jimenez Nimmo
student IDs: 230040125 & 230066319
group: PG 01

January 5, 2024

GitHub repository

Contents

1 Task 1

1.1 ReLU and Sigmoid Activation Functions

In neural networks, activation functions are crucial for introducing non-linearity, allowing the model to learn complex patterns. A widely used activation function is the sigmoid function defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{1}$$

It maps every input value to a value between 0 and 1, making it useful for models where we need to predict probabilities, like in binary classification problems. However, in the regions of the input space where the sigmoid function saturates (either close to 0 or 1), the gradient becomes very small, leading to a vanishing gradient problem, and possibly slowing down the learning process. Moreover, the sigmoid function is harder to compute than the more computationally efficient ReLU activation function. The ReLU function is defined as:

$$ReLU(x) = max(0, x) \tag{2}$$

which outputs the input directly if positive, else it will output zero. Since ReLU does not saturate, the gradient is constant for all positive inputs resolving the vanishing gradient problem during backpropagation. However, ReLU has its challenges, notably the "dying ReLU" problem [1] arises when neurons consistently output zero for all inputs, essentially becoming inactive and possibly impacting the overall performance of the model. Despite this drawback, we decided to use ReLU activation function for the hidden layers of our model.

1.2 Softmax Activation Function

The softmax function is crucial in neural networks for multi-class classification, transforming a vector of real values into a probability distribution. It is defined as:

$$\sigma(\bar{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \text{ for } i = i, ..., K \text{ and } \bar{z} = (z_1, ..., z_K) \in \mathbb{R}^K \tag{3}$$

ensuring that the output values range between 0 and 1 and sum up to 1. A key challenge in implementing softmax is numerical instability due to large exponentials, which is commonly addressed by subtracting the maximum input value from each input before exponentiation, a technique known as the softmax "shift". This function is often paired with the categorical crossentropy loss function, providing stability and efficiency in gradient-based optimization. This pairing is beneficial because it interprets the softmax outputs as probabilities and simplifies the backpropagation process by having a straightforward form of the gradient, making it an efficient and effective choice for classification tasks in neural networks.

1.3 Dropout and Inverted Dropout

Dropout consists of, through a probability distribution, deactivating neurons and their respective connections during training by nullifying their weights. This technique helps reduce overfitting, forcing the network to learn from fewer parameters to produce more robust parameters. This ensures information is distributed across neurons in the network. The dropout mask can be implemented as follows, in our case, using a binomial distribution:

```
self.mask = np.random.binomial(1, 1-self.probability, size=input_data.shape) /(1-
    self.probability) # This last part is for Inverted Dropout
return input_data * self.mask
```

Dropout must be present and unchanged during each training epoch for both forward and backward pass. The initial dropout implementation required all neuron activations to be scaled down by the dropout probability during testing. A solution to isolate dropout from the prediction phase is Inverted Dropout, which additionally only scales the activations during training.

Parameter	Value
Input Layer	30 neurons
Hidden Layer 1	128 neurons, ReLU
Hidden Layer 2	32 neurons, ReLU
Output Layer	2 neurons, Sigmoid
Learning Rate	0.1
Optimizer	Momentum
Momentum Coefficient	0.9
Epochs	2500
Batch Size	284

Table 1: Model without Dropout Summary

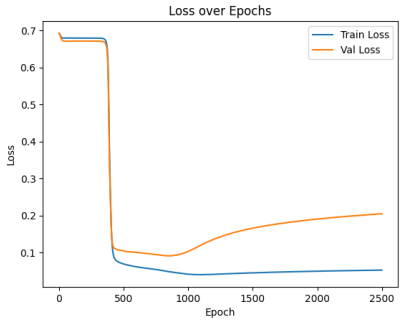

Figure 1: Training (in blue) and Validation (in orange) Loss over 2500 Epochs for model 1

Parameter	Value
Input Layer	30 neurons
Hidden Layer 1	128 neurons, ReLU
Dropout 1	25%
Hidden Layer 2	32 neurons, ReLU
Dropout 2	25%
Output Layer	2 neurons, Sigmoid
Learning Rate	0.1
Optimizer	Momentum
Momentum Coefficient	0.9
Epochs	2500
Batch Size	284

Table 2: Model with Dropout Summary

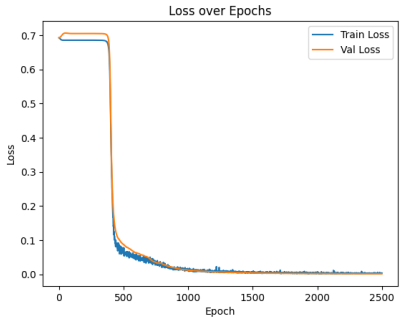

Figure 2: Training (in blue) and Validation (in orange) Accuracy over 2500 Epochs for model 2

To empirically validate the efficacy of the Dropout technique as a regulariser, we designed a comparative study using the breast cancer dataset [2]. The plots in Figure 1 and Figure 2 are obtained from the training, under the same condition, of a model without dropout (Table 1) and a model with dropout layers (Table 2) respectively. They show that the model without dropout quickly learned the training data, but its performance on the validation set began to worsen after 1000 epochs, indicating overfitting; on the other hand, the model with dropout maintained low validation loss, indicating better generalization and successful mitigation of overfitting.

1.4 Optimizers

Optimizers are crucial in neural network training, guiding how the model parameters adjust to minimize loss.

Gradient Descent (GD) is a very well known optimizer that follows the principle of iteratively

moving the weights in the direction of the steepest descent of the loss function:

$$\theta^{t+1} = \theta^t - \eta \cdot \nabla_\theta J(\theta) \tag{4}$$

where θ represents the parameters, η is the learning rate, and $\nabla_\theta J(\theta)$ is the gradient of the loss function J with respect to θ. While it is computationally straightforward, GD optimizer is vulnerable to local minima and oscillations; therefore, variants like Stochastic Gradient Descent (SGD) and Mini-batch Gradient Descent are usually preferred.

Momentum-based Optimizers incorporate the concept of momentum from physics to accelerate the gradient descent process:

$$\begin{cases} v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta) \\ \theta^{t+1} = \theta^t - v_t \end{cases} \tag{5}$$

where v_t is the velocity at time t, γ is the momentum coefficient, and η is the learning rate. This modification accelerates the convergence, especially for complex loss functions, and reduces oscillations while providing the ability to "escape" from local minima.

1.5 Fully-connected Neural Network

The core components of our implementation include a **Layer** class for defining fully connected layers, specific layer classes for the activation functions (**Sigmoid**, **ReLU**, **Softmax**) and **Dropout**, and a **NeuralNetwork** wrapper class for integrating the layers into the model. Our architecture is capable of handling different numbers of hidden layers, units per layer, and activation functions as well as using dropout, L1 and L2 regularisers. For training, we utilised an update rule that can use standard Gradient Descent, with all its variants, and Momentum-based optimization as defined in 1.4. Key training parameters are number of epochs, learning rate, momentum for Momentum optimizer, batch size and validation split set at 20% by default. Our model's performance was evaluated by calculating the accuracy on the test set, which comprised 20% of the total data. Accuracy, defined as the ratio of correctly predicted observations to the total observations, serves as a fundamental measure for evaluating how well the model correctly classifies the data. This measure is particularly relevant in scenarios where class distribution is balanced, as observed in the dataset employed for this study [3].

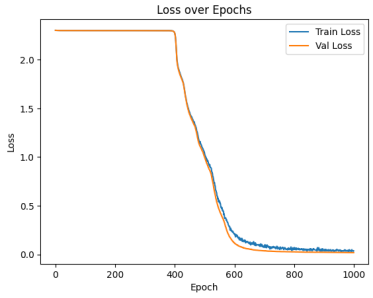

Figure 3: Training (in blue) and Validation (in orange) Loss over 1000 Epochs

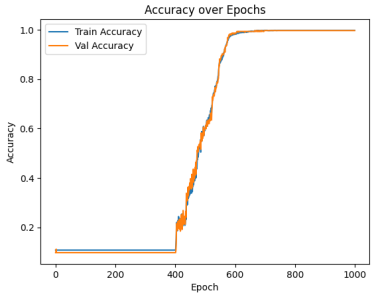

Figure 4: Training (in blue) and Validation (in orange) Accuracy over 1000 Epochs

Parameter	Value
Input Layer	64 neurons
Hidden Layer 1	32 neurons, ReLU
Dropout 1	25%
Hidden Layer 2	16 neurons, ReLU
Output Layer	10 neurons, Softmax
Learning Rate	0.001
Optimizer	Momentum
Momentum Coefficient	0.9
Epochs	1000
Batch Size	16

Table 3: Model Summary

The plots of training and validation loss (Figure 3), along with accuracy plots (Figure 4), results from the architecture in Table 3. They provide a visual representation of the model's learning

trajectory over the epochs, showing both training and validation curves reach the plateau, which reflects an effective learning and generalization supported by an accuracy of 95% on the test set. Moreover, the convergence of training and validation loss suggests that the model is not overfitting, as validation loss does not diverge from the training loss. Notably, we standardize each feature of the dataset before training by removing the mean and dividing by the standard deviation.

1.6 Discussion of results

A crucial component to achieve good performance across all experiments was a suitable and stable weights initialisation. A normal distribution with a mean of 0 isn't enough for fast and stable convergence (Figure 5). Using Xavier initialisation [4] significantly speeds up training and produces more accurate predictions (Figure 4).

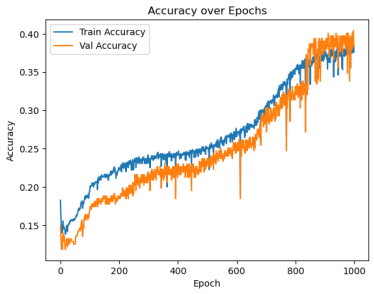

Figure 5: Accuracy plot using Normal distribution weight initialisation

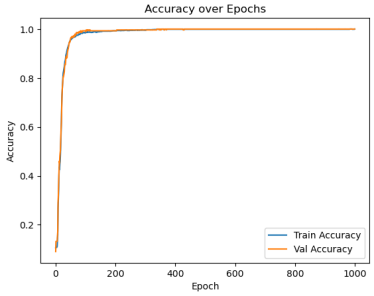

Figure 6: Performance increase by removing Hidden layer 2

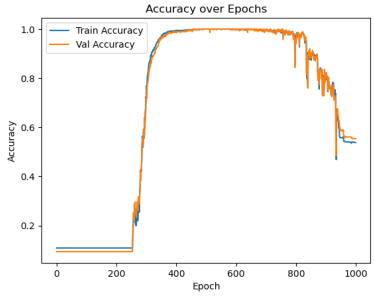

Figure 7: Accuracy plot using L2 regularizer in second hidden layer

Given the simplicity of the dataset, a shallower network provided better performance. Creating a network without the second hidden layer improved the performance in this task, achieving 97% accuracy (Figure 6). Using L1 and L2 regularizers did not improve the performance, rather significantly damaging all learning. Just with a small L2 coefficient of 0.01, it creates a rapid descent of performance once it has achieved good performance (Figure 7).

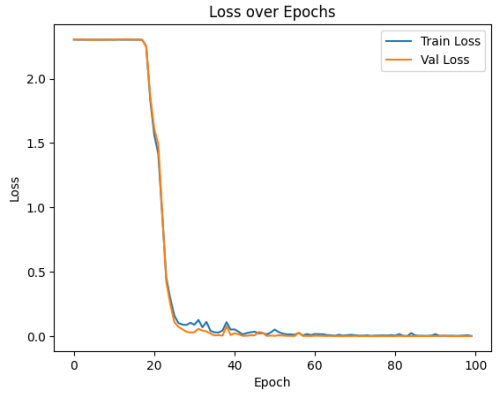

Figure 8: Training (in blue) and Validation (in orange) Loss over 100 epochs using Momentum Based Mini-batch Gradient Descent

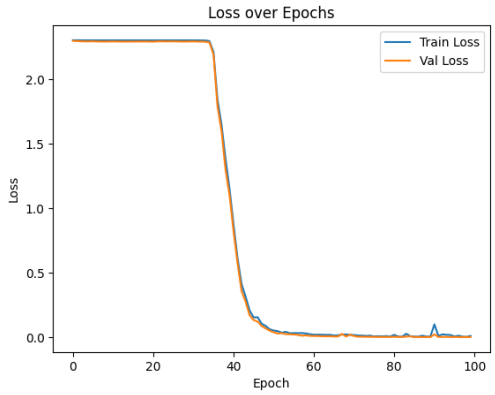

Figure 9: Training (in blue) and Validation (in orange) Loss over 100 epochs using Mini-batch Gradient Descent

In the loss plots (Figure 9 and Figure 8), both the mini-batch Gradient Descent optimizer and its momentum-based variant illustrate rapid decrease in loss, indicating similar levels of learning despite the momentum-based optimizer showing a faster convergence (40 epoch against 60 epochs). While momentum accelerated convergence is generally advantageous in more complex scenario, its impact on the final performance of the model may be less pronounced in simpler dataset like the one used in our experiment.

Our findings are confirmed and supported by the Random Search experiment run over 100 epochs detailed in the Appendix Table 5, indicating that the model can reach 98% accuracy with the architecture in Appendix Table 6. The next step would be increase the hyperparameter space and experiment with more sophisticated hyperparameter optimizers like Bayesian based optimizers [5] or TPE [6].

2 Task 2

2.1 Dataset

The topic of research we wanted to explore is tumor classification, specifically detecting different types of tumors or their absence from brain MRI scans. This is an important line of research, as when benign or malignant tumors grow in the brain, they can cause the pressure inside your skull to increase and damage the brain. Early detection of tumors can help save patient's lives. The multiclass Brain Tumor MRI Dataset from Kaggle [7] was used, containing 7023 images and 4 classes: **glioma - meningioma - no tumor** and **pituitary**. The test set contains 1311 images and the training set contains 5712 images.

2.2 Base Model

As a base model, we decided to use a Convolutional Neural Network (CNN) using **PyTorch** given its effectiveness in past studies [8]. A CNN architecture consists of a set of convolutional, pooling and fully connected layers. The convolutional layers apply a series of filters to the data, extracting features at different levels. It is based on the mathematical principle of convolution, described below:

$$o_{i,j} = \sum_{m=1}^{M} \sum_{n=1}^{N} x_{i-m,j-n} w_{m,n} \tag{6}$$

where $o_{i,j}$, $x_{i-m,j-n}$ and $w_{m,n}$ represent the output of the convolution layer at pixel (i, j), the input at pixel $(i - m, j - n)$, and the filter/kernel at pixel (m, n) respectively. Padding may be required to preserve the image size. The pooling layers reduce the dimensions of the data while preserving most of its features. The fully connected layers are placed after feature extraction, using the extracted features in neural network tasks, such as classification or regression. PyTorch provides an accessible framework for building a CNN through **nn.module**, with the **__init__()** method initialising the structure of the CNN, while the **forward()** method describes how the data is passed through each layer.

2.3 Improvements

Alongside creating a validation set and facilitating batch training through PyTorch's **Dataset** and **Dataloader** data primitives [9], data augmentations and preprocessing were introduced. Data transformations were incorporated (random horizontal flips and random rotations) using PyTorch

Transforms modules [10], improving the models generalisation and robustness. Data preprocessing through image resizing and data normalisation using the mean and variance of the dataset was implemented, providing numerical stability and faster convergence. A backbone in a CNN consists of the feature extracting section of the architecture. Large pre-trained architectures trained on large datasets, with extensive layers and weights, can be utilised as a backbone. This backbone can be combined with an untrained fully connected layer for a specific task, in our case, classification. This is an example of transfer learning. The weights of the backbone can be frozen, for faster training, at the slight cost of performance. VGG-16 [11] is an example of a largely used backbone, consisting of a 16-layer CNN, with demonstrated high performance in medical imaging [12]. PyTorch provides access to pre-trained models, including VGG-16.

2.4 Hyperparameter optimization

There are a large number of hyperparameters in a CNN which can be adjusted: number of layers and their respective types, number of filters per layer, kernel size of a filter, padding size, activation functions between layers, and all hyperparameters present in a neural network. A list of the best hyperparameter values found through human cross-validation can be found in Table 4.

Parameter	**Value**
Optional Backbone	VGG-16
Conv Layer 1	16 filters, kernel_size = 3
Pooling	kernel_size = 3, ReLU
Conv Layer 2	16 filters, kernel_size = 3
Pooling	kernel_size = 3, ReLU
Hidden Layer	191844 neurons, ReLU
Output Layer	1024 neurons, Softmax
Learning Rate	0.001
Optimizer	Adam
Loss	Cross Entropy
Epochs	20
Batch Size	16

Table 4: Model Summary

2.5 Results

For fair comparison between with and without using a backbone, the same data processing and fully connected classifier was used. They were trained up to 20 epochs, using the same learning rate and optimizer, and trained on a Cross Entropy loss. We evaluated the model's performance based on accuracy, precision, recall and confusion matrices, as well as the model's training and validation losses, as noted in Appendix Figures 12, 13. Results are very promising, achieving over 95% accuracy in both models. In this task, higher recall rather than precision is preferred given the sensitive nature of the task. Both of these metrics perform similarly, showcasing the robustness of the approach. Both models converge rapidly, with the backbone version taking 24 minutes compared to the ordinary CNN (16 minutes). The backbone demonstrates more instability in its loss plots (Appendix Figures 10,11), although still with improved performance. This may be due to using an incorrect hyperparameter, such as a too rapid learning rate. Improvements can include improved architecture, perhaps through Neural Architecture Search solutions [13]. More robust hyperparameters can increase the stability and performance of the models, alongside additional data transformations for robustness. An interesting angle can be additional backbone testing, including retraining the pre-trained backbone weights to better fit the data. The loss plots also indicate early stopping is a necessary step to avoid the currently occurring overfitting.

Appendix

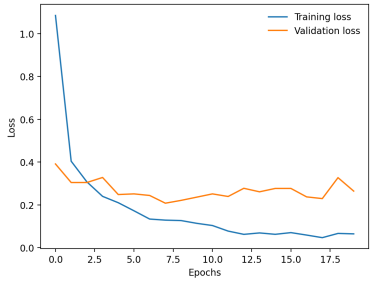

Figure 10: Training (blue) and Validation (orange) loss of CNN model over 20 Epochs

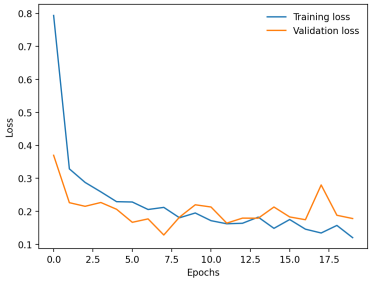

Figure 11: Training (blue) and Validation (orange) loss of CNN model over 20 Epochs

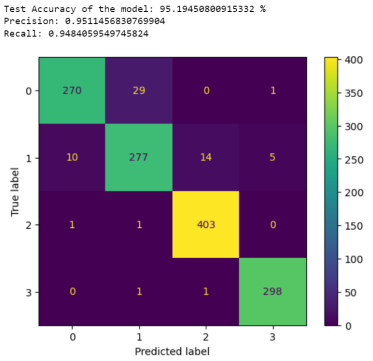

Figure 12: Evaluation metrics for CNN without backbone

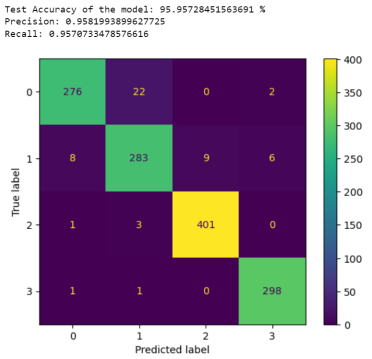

Figure 13: Evaluation metrics for CNN (VGG-16 backbone)

Hyperparameter	**Values**
Layer 1 Nodes	32, 64, 128
Layer 1 L1 Regularization	0.0, 0.01
Layer 1 L2 Regularization	0.0, 0.01
Layer 2 Nodes	16, 32, 64
Layer 2 L1 Regularization	0.0, 0.01
Layer 2 L2 Regularization	0.0, 0.01
Learning Rate	0.01, 0.1, 0.5
Epochs	100, 500, 1000
Optimizer	GD, Momentum
Momentum	0.5, 0.9
Batch Size	16, 32, 64

Table 5: Hyperparameter Space for Random Search

Hyperparameter	**Best Value**
Layer 1 Nodes	128
Layer 1 L1 Regularization	0.0
Layer 1 L2 Regularization	0.0
Layer 2 Nodes	32
Layer 2 L1 Regularization	0.0
Layer 2 L2 Regularization	0.0
Learning Rate	0.1
Epochs	500
Optimizer	Momentum
Momentum	0.5
Batch Size	16

Table 6: Best Hyperparameters Found by Random Search

References

[1] L. Lu, Y. Shin, Y. Su, and G. Karniadakis, "Dying relu and initialization: Theory and numerical examples," *Communications in Computational Physics*, vol. 28, pp. 1671–1706, 11 2020.

[2] Scikit-learn, "sklearn.datasets.load_breast_cancer." `https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_breast_cancer.html`, 2011. Accessed: 2024-01-02.

[3] Scikit-learn, "sklearn.datasets.load_digits." `https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_digits.html`, 2011. Accessed: 2024-01-02.

[4] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics* (Y. W. Teh and M. Titterington, eds.), vol. 9 of *Proceedings of Machine Learning Research*, (Chia Laguna Resort, Sardinia, Italy), pp. 249–256, PMLR, 13–15 May 2010.

[5] J. Wu, X.-Y. Chen, H. Zhang, L.-D. Xiong, H. Lei, and S.-H. Deng, "Hyperparameter optimization for machine learning models based on bayesian optimization," *Journal of Electronic Science and Technology*, vol. 17, no. 1, pp. 26–40, 2019.

[6] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," in *Advances in Neural Information Processing Systems* (J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K. Weinberger, eds.), vol. 24, Curran Associates, Inc., 2011.

[7] M. Nickparvar, "Brain tumor mri dataset," 2021.

[8] D. Sarvamangala and R. V. Kulkarni, "Convolutional neural networks in medical image understanding: a survey," *Evolutionary intelligence*, vol. 15, no. 1, pp. 1–22, 2022.

[9] PyTorch, "Data loading and processing tutorial." `https://pytorch.org/tutorials/beginner/data_loading_tutorial.html`, 2021. Accessed: 2024-01-02.

[10] PyTorch, "Torchvision transforms." `https://pytorch.org/vision/stable/transforms.html`, 2023. Accessed on 2024-01-02.

[11] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[12] T. Kaur and T. K. Gandhi, "Automated brain image classification based on vgg-16 and transfer learning," in *2019 International Conference on Information Technology (ICIT)*, pp. 94–98, IEEE, 2019.

[13] C. White, M. Safari, R. Sukthanker, B. Ru, T. Elsken, A. Zela, D. Dey, and F. Hutter, "Neural architecture search: Insights from 1000 papers," *arXiv preprint arXiv:2301.08727*, 2023.