Binary Classification of Chest X-rays for Prediction of Pneumonia

Gabriela Copetti

1. Introduction

Binary image classification models were built and trained in PyTorch using the "Chest X-Rays Classification Dataset", available in Hugging Face [1]. Simple models with different sets of hyperparameters were built to investigate how these affect model training and performance. Good predictions, with accuracy ~93%, were obtained using neural network models with simple architectures.

2. Methodology

All computational steps were performed in Google Collaboratory notebooks, making use of the T4 GPU. Code was adapted from Daniel Bourke's PyTorch tutorial [2].

2.1. The dataset

The dataset is divided into train, validation and testing subsets, with 4077, 1165 and 582 images, respectively. A sample of the dataset is shown in Fig. 1 and 2. The image is in PIL format, in mode RGB, and has a size of 640x640. Label 0 indicates that the X-ray image is "normal", i.e. negative for pneumonia. Label 1 indicates a patient positive for pneumonia [1].

```
{'image_file_path': '/storage/hf-datasets-cache/all/datasets/60340657865253-config-parquet-and-info-keremberke-chest-xray-cla-
9d66ea8b/downloads/extracted/8202f7dd6f1edf5e674abe75990eb233fbbca4408e132a3acd5268bd99708e15/NORMAL/IM-0003-
0001_jpeg.rf.3fffcf9c33575f8f928b017484f99a64.jpg',
 'image': <PIL.JpegImagePlugin.JpegImageFile image mode=RGB size=640x640>,
 'labels': 0}
```

Figure 1 - Sample entry in the dataset.

Figure 2 – Image from sample entry.

Exploratory analysis showed no missing or duplicated values. The dataset is **imbalanced**, with most images having label 1 (pneumonia) (Fig. 3), which could lead to a bias towards classifying images as 1.

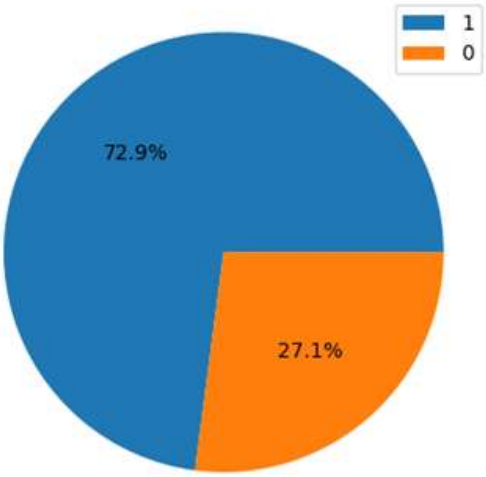

Figure 3 – Percentages of 0 (NORMAL) and 1 (PNEUMONIA) classes in training set.

2.2. Data preparation

The following transformations were applied to the dataset prior to training:

1. Image and labels were transformed into *torch.Tensor* objects, with dtype *torch.float*, using the *set_format()* function [3]. Also, image path was discarded.

The transformed images had shape *torch.Size([3, 640, 640])*, which indicates colour channels = 3, height = 640 and width = 640. Pixel values were in the 0-255 range.

2. Since images are grayscale, *rgb_to_grayscale* transformation from *torchvision* was used to convert channel number from 3 to 1 [4]. The dataset *map()* function, with *batch_size* = 32, was used to prevent memory overload [3].

3. The tensors were divided by 255 for pixel values to be in 0-1 range, also using the *map()* function.

Sample images were plotted to check if transformations did not alter the images visually.

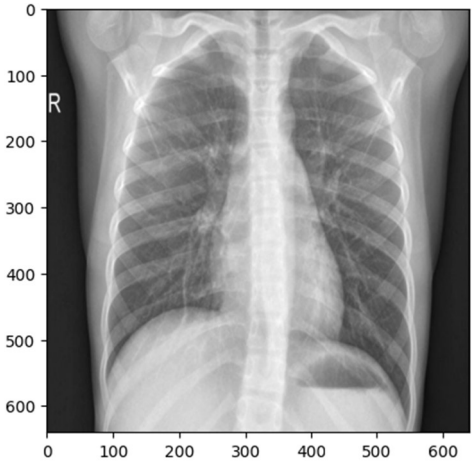

Figure 4 - Image after transformation.

For faster data retrieval and to reduce model overfitting, the datasets were loaded into PyTorch DataLoader. The DataLoader allows the creation of minibatches of data, which can be reshuffled during training [5]. The datasets were split into batches of 32 samples (Fig. 5).

```
#Setting up DataLoader

# Setup the batch size hyperparameter
BATCH_SIZE = 32

# Turn datasets into iterables (batches)
train_dataloader = DataLoader((train_dataset), # dataset to turn into iterable
    batch_size=BATCH_SIZE, # how many samples per batch?
    shuffle=True # shuffle data every epoch?
)

valid_dataloader = DataLoader((valid_dataset),
    batch_size=BATCH_SIZE,
    shuffle=False # don't necessarily have to shuffle the validation data
)

test_dataloader = DataLoader((test_dataset),
    batch_size=BATCH_SIZE,
    shuffle=False # don't necessarily have to shuffle the testing data
)

print(f"Dataloaders: {train_dataloader, valid_dataloader, test_dataloader}")
print(f"Length of train dataloader: {len(train_dataloader)} batches of {BATCH_SIZE}")
print(f"Length of validation dataloader: {len(valid_dataloader)} batches of {BATCH_SIZE}")
print(f"Length of test dataloader: {len(test_dataloader)} batches of {BATCH_SIZE}")
```

```
Dataloaders: (<torch.utils.data.dataloader.DataLoader object at 0x7dbdd7bcc5b0>, <torch.ut
Length of train dataloader: 128 batches of 32
Length of validation dataloader: 37 batches of 32
Length of test dataloader: 19 batches of 32
```

Figure 5 - Creating minibatches of data using dataloader.

2.3. The experiment

The X-ray images were classified using machine learning algorithms called artificial neural networks or ANN [6]. These neural networks are composed of different layers, each containing multiple nodes (referred here as hidden units). The networks are density connected, i.e., each node is connected to every node in the next layer. The input data is processed by these nodes, which attempt to find relationships between its features. This is achieved by minimizing a loss function, which describes how far off the algorithm performance is from achieving its goal. Minimization is performed by the optimizer, which will try to find which sets of parameters, called weights, lead to the smallest loss.

Three types of ANN models were built:

1. **Linear models:** These are linear neural networks, i.e., networks that able to find linear relationships between features. The models were set with *input_shape* = 409600, which corresponds to the size of the image input after being flatten into a single vector (1 x 640 x 640). In Fig 6 is an example of model structure in PyTorch.

```
XRAYS_linear2(
  (layer_stack): Sequential(
    (0): Flatten(start_dim=1, end_dim=-1)
    (1): Linear(in_features=409600, out_features=10, bias=True)
    (2): Linear(in_features=10, out_features=1, bias=True)
  )
)
```

Figure 6 - Linear model with 2 linear layers and 10 hidden units.

   Different models were built varying the number of layers and the number of hidden units (nodes). To allow convergence, a learning rate of 10^{-4} was used. The learning rate controls how big is the step the model will take during the gradient descent to minimize the loss function [7].

2. **Linear + ReLU models:** These are multilayer perceptrons that use the rectifier activation function (ReLU) to introduce non-linearity after each linear layer [2, 8]. MLPs can find more complex relationships between the data [6]. In Fig 7 is an example of the structure in PyTorch. Once again, models were built with different number of layers and nodes. Learning rate was kept as 10^{-4}.

```
XRAYS_linear2ReLU(
  (layer_stack): Sequential(
    (0): Flatten(start_dim=1, end_dim=-1)
    (1): Linear(in_features=409600, out_features=10, bias=True)
    (2): ReLU()
    (3): Linear(in_features=10, out_features=1, bias=True)
    (4): ReLU()
  )
)
```

Figure 7 - ReLU activation function is used after each layer to introduce non-linearity.

3. **Convolutional Neural Networks:** A CNN is a special type of neural network that uses filters or kernels to recognise patterns in images. It consists of a convolutional layer that generates a feature map using the filters, a pooling layer to downsample the feature map and reduce overfitting, and a fully connected layer [9, 6]. CNN models were built with varying number of filters, which the number of hidden units. The size of filters in the convolution layers used were 3x3 and 5x5. The filter size in the pooling layer was kept as 2x2. Max pooling was used, which takes the highest value covered by the filter. Fig 8 exemplifies the model structure in PyTorch. To allow convergence, learning rate of 10^{-2} was used.

```
XRAYS_CNN(
  (block_1): Sequential(
    (0): Conv2d(1, 10, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): Conv2d(10, 10, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU()
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (block_2): Sequential(
    (0): Conv2d(10, 10, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): Conv2d(10, 10, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU()
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (classifier): Sequential(
    (0): Flatten(start_dim=1, end_dim=-1)
    (1): Linear(in_features=256000, out_features=1, bias=True)
  )
)
```

Figure 8 - CNN model with 10 hidden units (filters) of size 3x3 in the convolution layer.

Models were selected to vary other parameters further:

i. **The optimizer:**
The main optimizer used was the SGD, which implements stochastic gradient descent [10]. A few models were trained with the Adam algorithm [11] for comparison.

ii. **The weight parameter the loss function:**
The BCEWithLogitLoss() function was used to calculate loss. It combines a Sigmoid layer with Binary Cross Entropy loss function. The *pos_weight* parameter is designed to adjust the imbalance between negative and positive samples [12, 13]. Since the 0/1 proportion in the training set is of 0.37, this value is set as the weight.

iii. **The number of epochs:** After the selection of the other parameters, the number of epochs, i.e. the number of times the data is going through the learning algorithm, was increased. This gives more opportunities for the model to learn and minimize the loss function, though it can lead to overfitting [14].

Table 1 summarises the characteristics of models trained using batch size 32, available on the notebook named "Chest_XRays_Training_and_Evaluation_BatchSize32.ipynb".

Table 1 – Parameters of models created with batch size 32.

Model	Class	Linear layers	Hidden units	Kernel size in Cov2D	Class weight	Optimizer	Learning rate	Epochs
0	Linear	2	10		no	SGD	10^{-4}	10
1	Linear	3	10		no	SGD	10^{-4}	10
2	Linear	4	10		no	SGD	10^{-4}	10
3	Linear	2	5		no	SGD	10^{-4}	10
4	Linear	2	20		no	SGD	10^{-4}	10
5	Linear+ReLU	2	10		no	SGD	10^{-4}	10
6	Linear+ReLU	3	10		no	SGD	10^{-4}	10
7	Linear+ReLU	4	10		no	SGD	10^{-4}	10
8	Linear+ReLU	2	5		no	SGD	10^{-4}	10
9	Linear+ReLU	2	20		no	SGD	10^{-4}	10
10	CNN	1	5	3x3	no	SGD	10^{-2}	10
11	CNN	1	10	3x3	no	SGD	10^{-2}	10
12	CNN	1	20	3x3	no	SGD	10^{-2}	10
13	CNN	1	10	5x5	no	SGD	10^{-2}	10
14	Linear	2	10		yes	SGD	10^{-4}	10
15	Linear+ReLU	2	10		yes	SGD	10^{-4}	10
16	CNN	1	10	3x3	yes	SGD	10^{-2}	10
17	Linear	2	10		yes	Adam	10^{-4}	10
18	Linear+ReLU	2	10		yes	Adam	10^{-4}	10
19	CNN	1	10	3x3	yes	Adam	10^{-4}	10
20	Linear	2	10		no	SGD	10^{-4}	50
21	Linear+ReLU	2	10		no	SGD	10^{-4}	50
22	CNN	1	10	3x3	no	SGD	10^{-2}	50
23	Linear	2	10		yes	SGD	10^{-4}	50
24	Linear+ReLU	2	10		yes	SGD	10^{-4}	50
25	CNN	1	10	3x3	yes	SGD	10^{-2}	50

Finally, to explore the influence of **batch size** on stability, models were trained for 60 epochs in separate notebook ("Chest_XRays_Training_and_Evaluation_BatchSize64.ipynb") using the batch sizes of 64, instead of 32, in the train, validation and test dataloaders (Table 2). Batch size sets the number of samples that are used to calculate the gradient and compute weights before the model's internal parameters are updated. Small batch sizes require less memory but can lead to noisy gradients and instability, while large batch sizes lead to faster convergence and more stability but might not generalise so well [14].

Table 2 – Parameters of models created with batch size 64.

Model	Class	Linear layers	Hidden units	Kernel size in Cov2D	Class weight	Optimizer	Learning rate	Epochs
20b	Linear	2	10		no	SGD	10^{-4}	60
21b	Linear+ReLU	2	10		no	SGD	10^{-4}	60
22b	CNN	1	10	3x3	no	SGD	10^{-2}	60
23b	Linear	2	10		yes	SGD	10^{-4}	60
24b	Linear+ReLU	2	10		yes	SGD	10^{-4}	60
25b	CNN	1	10	3x3	yes	SGD	10^{-2}	60

2.4. Evaluation metrics

The selection of parameters during the experiment was based on both the accuracy during validation step and training stability. At the end of the experiment, models were selected to be evaluated using the test data subset.

The accuracy computes the number of times a model made a correct prediction. One must be careful when analysing accuracy with an unbalanced dataset [15]. Since about 70% of labels are equal to "1", if the model were to always predict "1", it would still give us around 70% accuracy.

Other two useful metrics are: precision and recall. The precision considers the number of false positives, while the recall the number of false negatives. They are given by

$$\text{Precision} = \frac{\text{T.P.}}{\text{T.P. +F.P.}} \text{ , Recall} = \frac{\text{T. P.}}{\text{T.P. +F.N.}},$$

in which T.P. is the number of true positives, F.P the number of false positives and F.N. the number of false negatives. Confusion matrices were used to visualize the number of false positives and false negatives. The F1 score assesses class-wise performance, by combining precision and recall, so that

$$\text{F1 Score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision+Recall}} \text{ [16].}$$

The F1 score ranges from 0 to 1. The closer it is to 1, the better the model is as classifier.

3. Results

3.1. Selecting number of layers and hidden units

Linear models

Fig. 9 shows the loss and accuracy curves for linear models with 10 hidden units (nodes) and number of layers varying from 2 to 4. One can see that the speed at which the loss is minimized decreases with number of layers, as well as the training accuracy. Fixing the number of layers as 2, the number of nodes were also varied (Fig. 10). Changing the number of nodes had less an impact on accuracy than changing the number of layers. Still, the model with 10 nodes in each layer minimized the loss slightly faster.

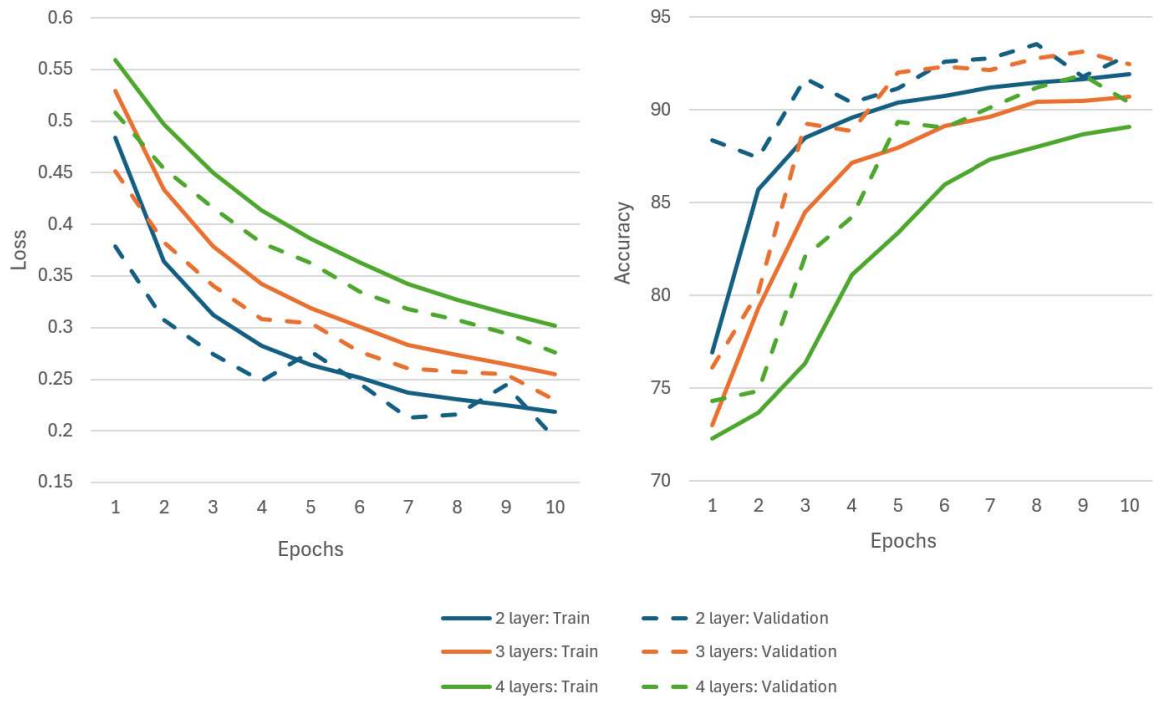

Figure 9 - Loss and accuracy curves for models 0, 1 and 2, with respectively 2, 3 and 4 linear layers. The three models have 10 hidden units (nodes). Optimizer used was SGD and no weight used in the loss function.

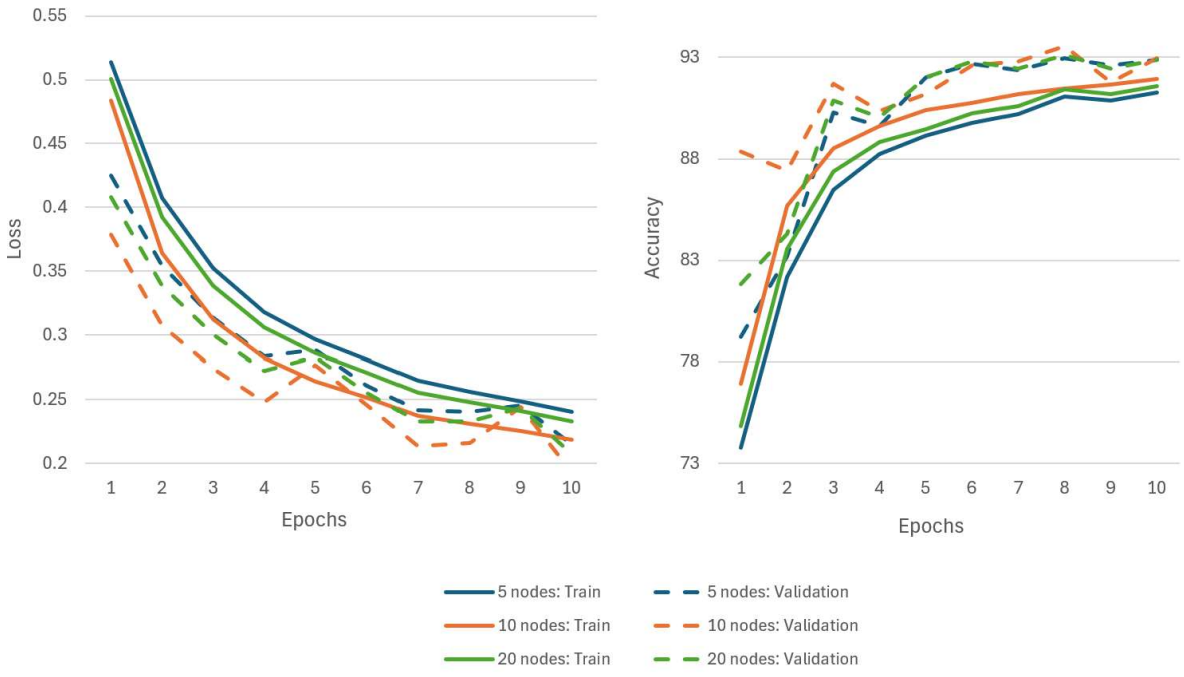

Figure 10 - Loss and accuracy curves for linear models 3, 0 and 4, with respectively 5, 10 and 20 hidden units (nodes). The three models have 2 linear layers. Optimizer used was SGD and no weight used in the loss function.

Linear + ReLU models

When adding the ReLU function, the model converged with 2 layers. However, it did not converge with neither 3 nor 4 layers, even when trying to adjust the learning rate with values from 10^{-1} to 10^{-5}. Keeping only two layers, the number of nodes were then varied. The best results were obtained with 10 nodes, as can be seen in Fig. 11.

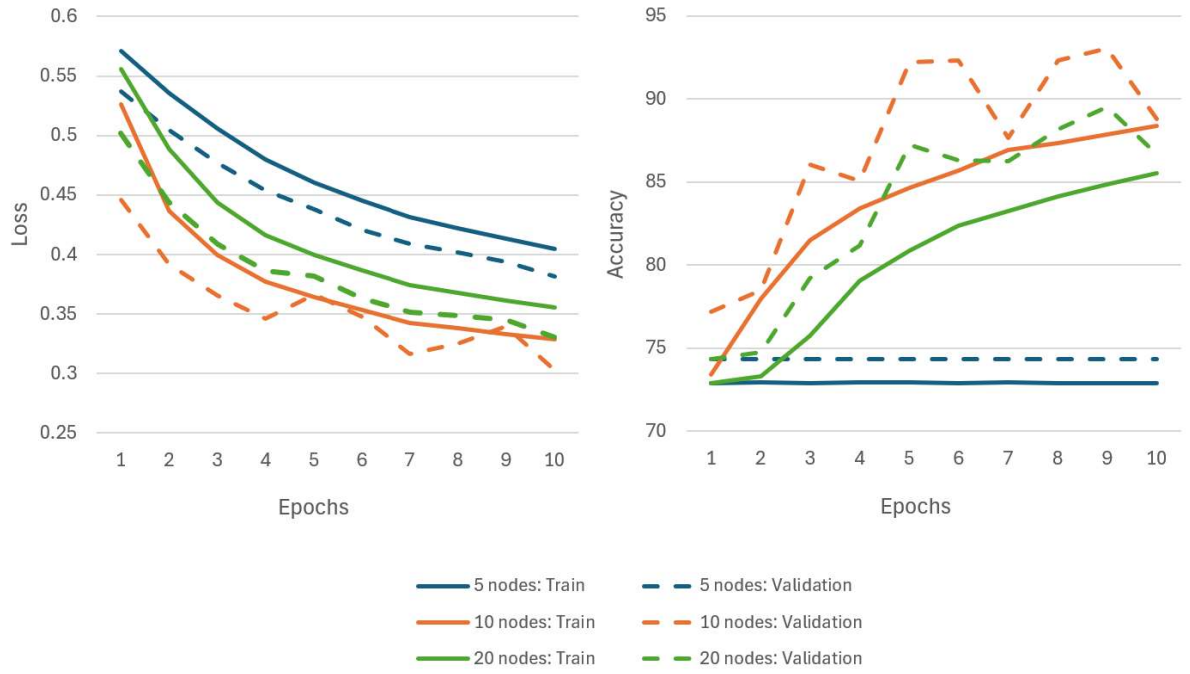

Figure 11 - Loss and accuracy curves for Linear + ReLU models number 5, 8 and 9, with respectively 5, 10 and 20 hidden units (nodes). The three models have 2 linear layers. Optimizer used was SGD and no weight used in the loss function.

CNN models

CNN models with kernel size 3x3 in the convolution layer were built with 5, 10, and 20 hidden units (filters). The loss and accuracy curves have visibly more oscillation than the ones of the previous models (Fig 12). Models with 5 and 10 filters outperform the one with 20 and behave quite similarly. Therefore, it was chosen to fix the number of filters as 10 for the next models.

A CNN model with kernel size 5x5 in the convolution layer and 10 filters was built for comparison. This did not improve stability or performance, as seen on Fig 13.

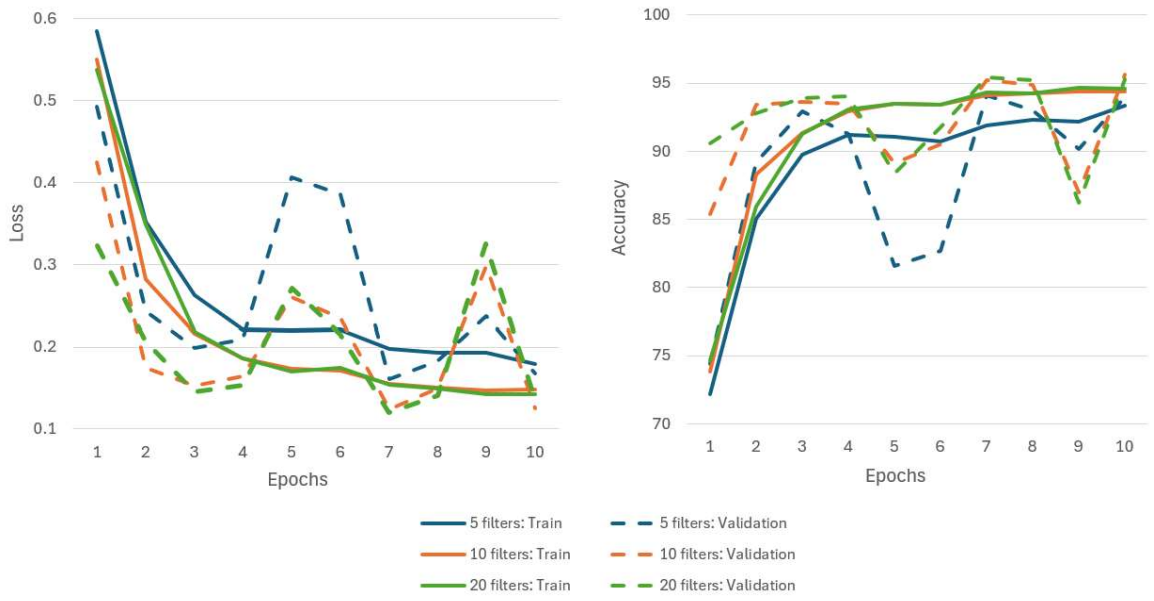

Figure 12 - Loss and accuracy curves of CNN models 10, 11 and 12, with respectively 5, 10 and 20 hidden units (filters). The three models have kernel size 3x3 in the convolution layer. Optimizer used was SGD and no weight used in the loss function.

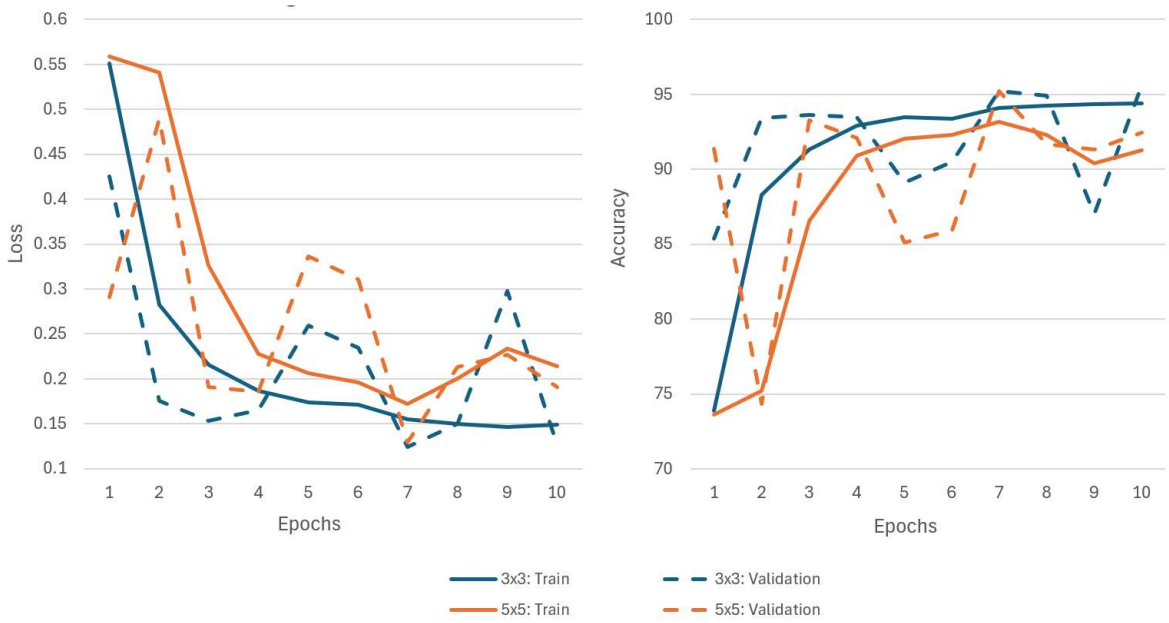

Figure 13 - Loss and accuracy curves for CNN models 11 and 13, with respectively 3x3 and 5x5 kernel size in the convolution layers. Both models have 10 hidden units (filters). Optimizer used was SGD and no weight used in the loss function.

3.2. Weighted loss function and optimizer

Models were trained with the class weight in the loss function to attempt to reduce any bias caused by class imbalance. This seemed to have different effects depending on the class of the model. For the Linear model, class weight resulted in the accuracy curve starting to plateau at

a smaller accuracy value (Fig. 14). For the Linear + ReLU model, curves behave similarly with or without weight, with a slight increase in accuracy when adding the weight parameter (Fig. 15). For both Linear and CNN models, weighing the loss function resulted in more oscillations in validation accuracy (Fig. 16). More epochs are needed for further assessment.

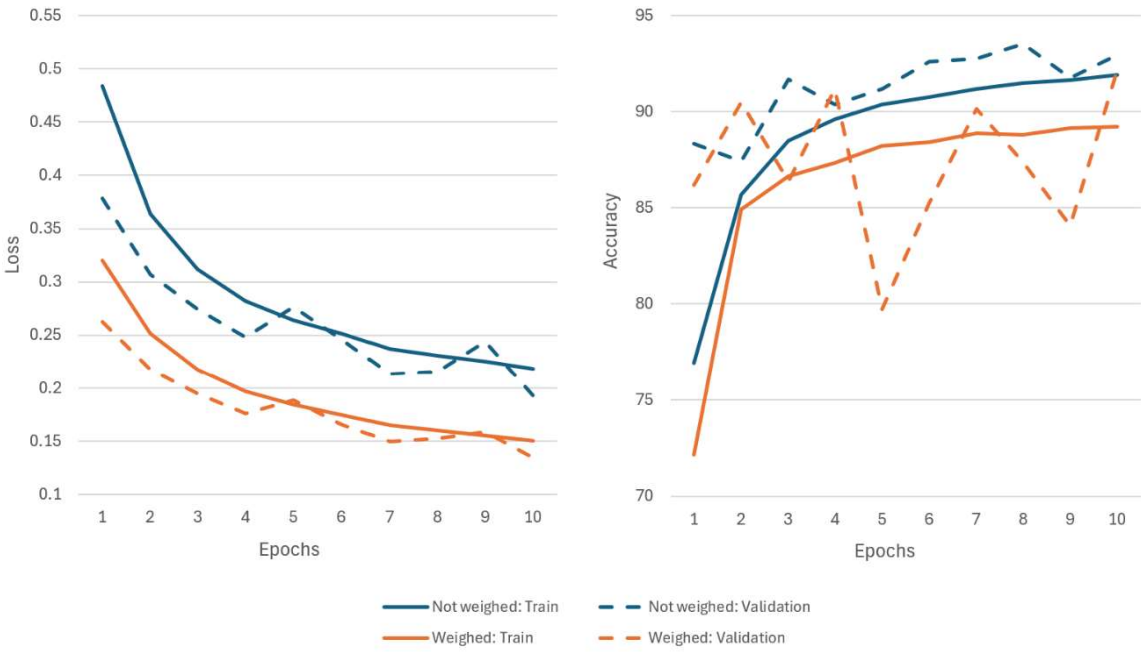

Figure 14 - Loss and accuracy curves for Linear models with and without weight in loss function (models 0 and 14, respectively). Both models have two linear layers and 10 hidden units (nodes). SGD optimizer was used.

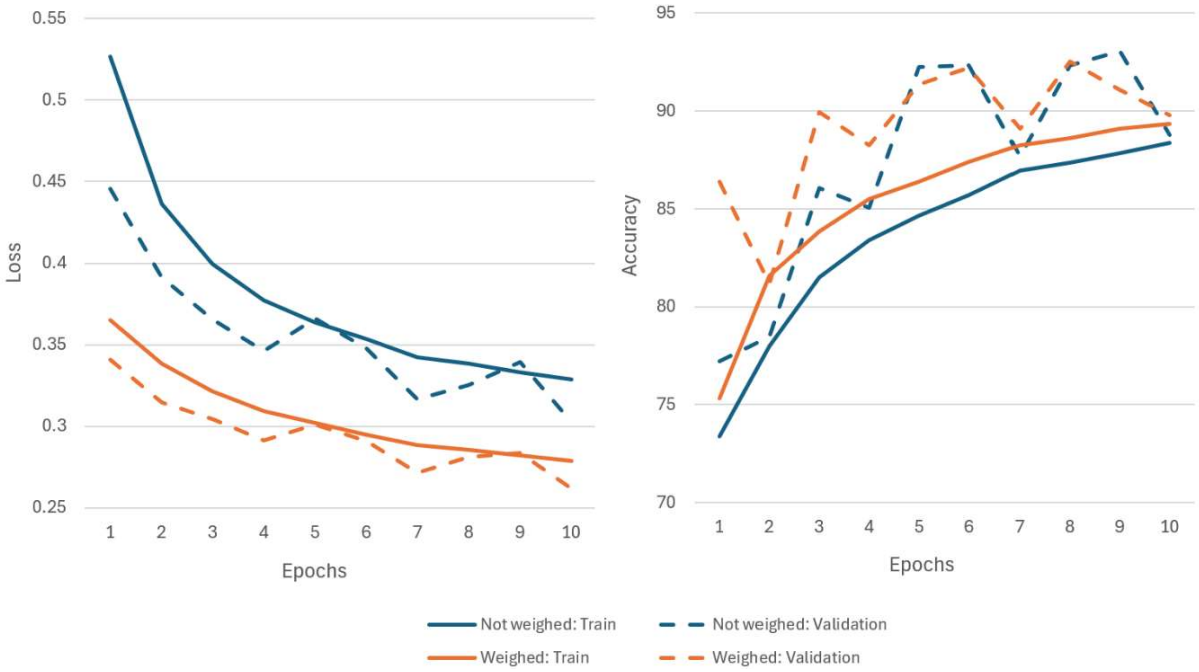

Figure 15 - Loss and accuracy curves for Linear + ReLU models with and without weight in loss function (models 5 and 15, respectively). Both models have two linear layers and 10 hidden units (nodes). SGD optimizer was used.

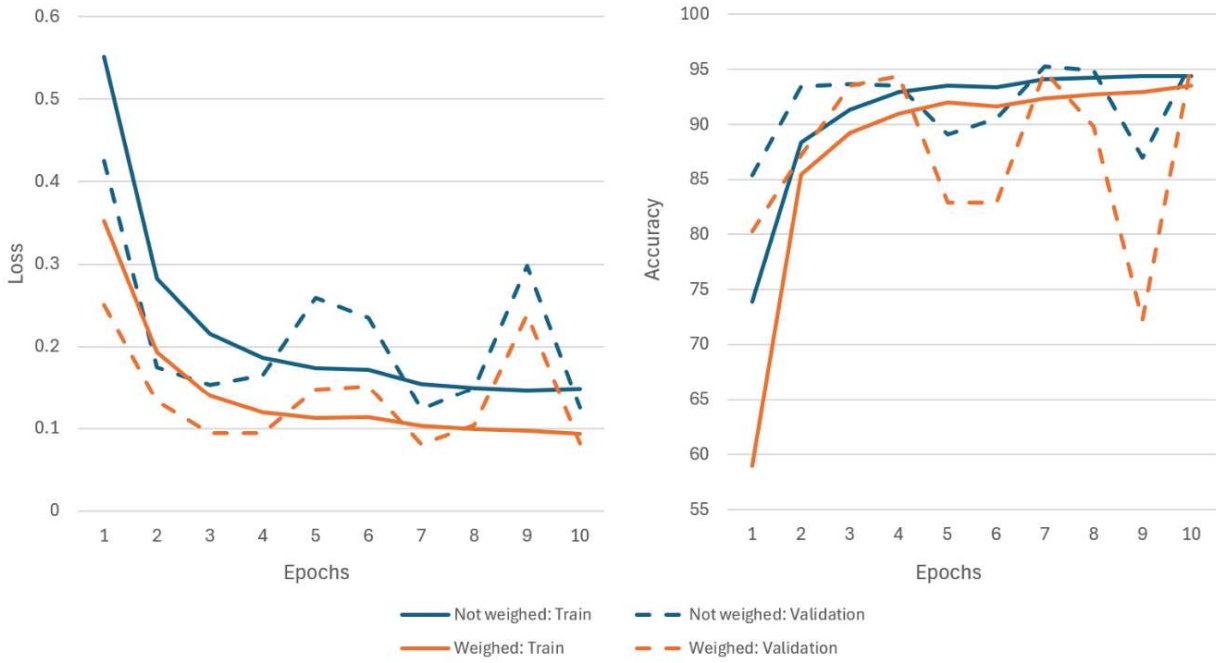

Figure 16 - Loss and accuracy curves for CNN models with and without weight in loss function (models 11 and 19, respectively). Kernel size in convolution layer is 3x3 and the number of hidden units (filters) is 10. SGD optimizer was used.

Attempting to improve optimization when using class weight, the optimizer was changed from SGD to Adam. Using the same set of hyperparameters selected in the previous steps, the Adam algorithm did not converge for neither the Linear + ReLU nor CNN. Even for the Linear model, it was very unstable (Fig 17). Generally, a fine-tuned Adam would outperform SGD [17]. Tuning the hyperparameter for Adam was not included in the present work but could be an interesting project continuation.

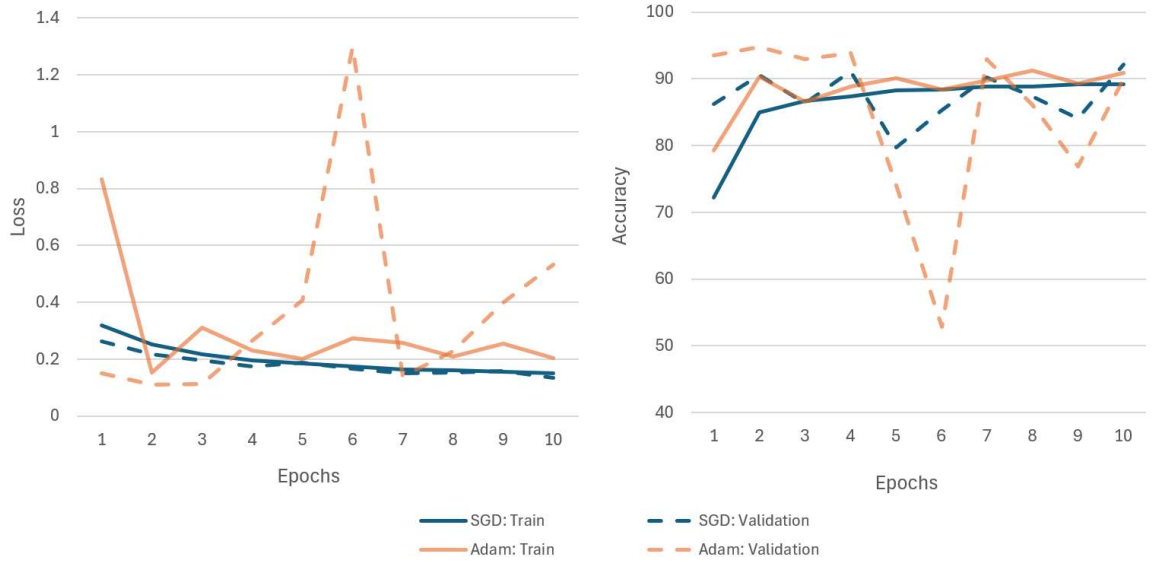

Figure 17 – Loss and accuracy curves for Linear models 0 (SGD optimizer) and 19 (Adam optimizer). Both models have 2 layers and 10 hidden units (nodes). No weights were used in the loss function.

3.3. Increasing number of epochs

After selecting the number of layers (2), number of hidden units (10) and kernel size (3x3), the number of epochs was increased from 10 to 50. The loss and accuracy curves of models using both unweighted and weighted loss functions are shown in Fig 18.

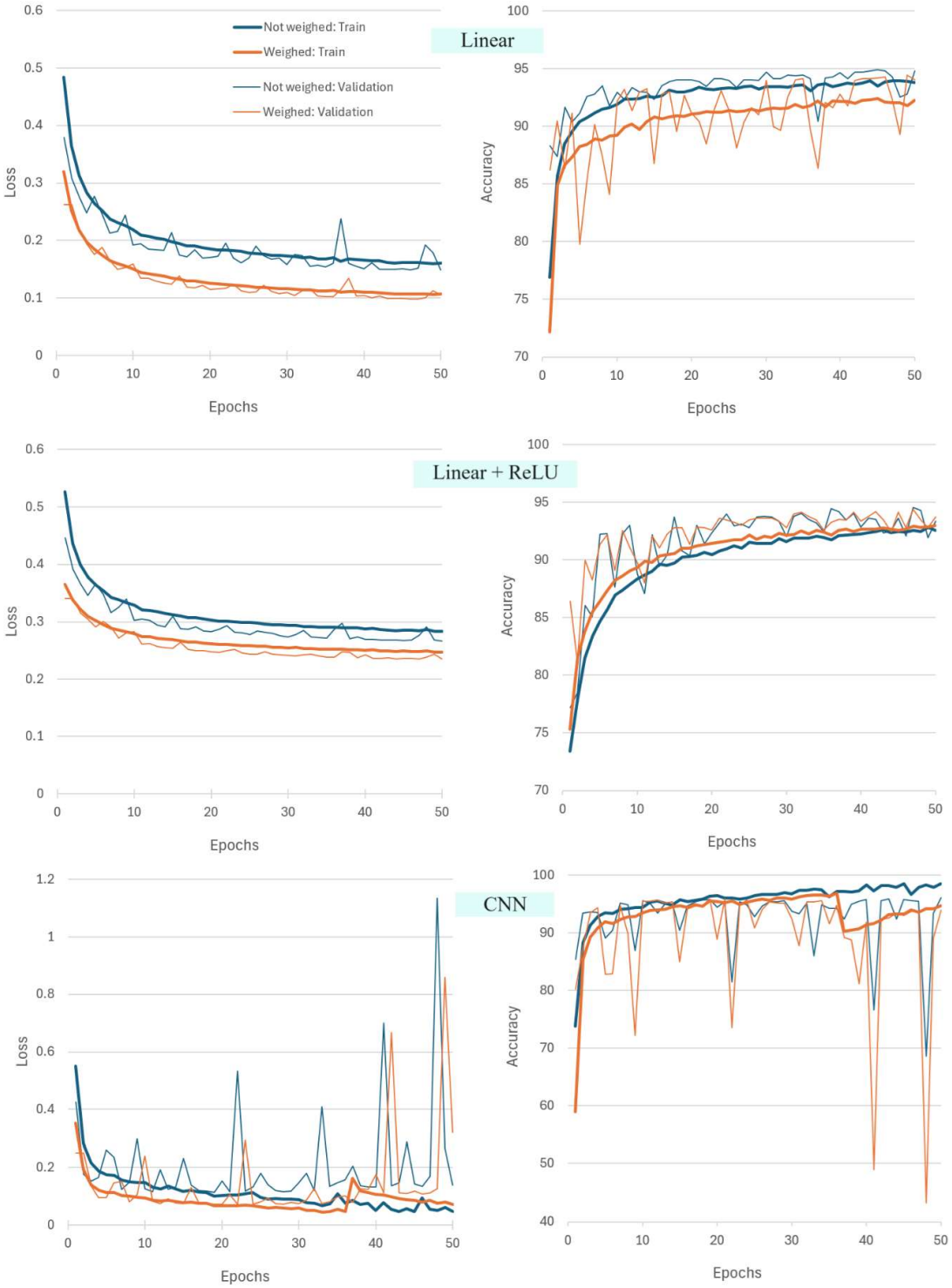

Figure 18 – Loss and accuracy curves for Linear (2 linear layers), Linear + ReLU (2 linear layers), and CNN (3x3 kernel in convolution layer) models, when increasing the number of epochs to 50. Curves using weighted and not weighted loss functions are shown. 10 hidden units (nodes/filters) and the SGD optimizer were used.

Once again, adding a class weight to the loss function did not significantly impact the train/validation accuracy curves of the Linear + ReLU model. However, in the case of the Linear models, using weighted loss function led to a lower validation accuracy and more oscillations. By its turn, both CNN models were unstable and showing signs of overfitting, as the train loss continues to decrease while the baseline of the validation loss plateaus.

These six models were evaluated on unseen data using the test data subset. In Figure 19, one of the confusion matrices produced in this evaluation step is shown as an example. Model 20 (Linear, no weight) was able to predict 400 true positives and 142 true negatives, while predicting 29 false positives and 11 false negatives.

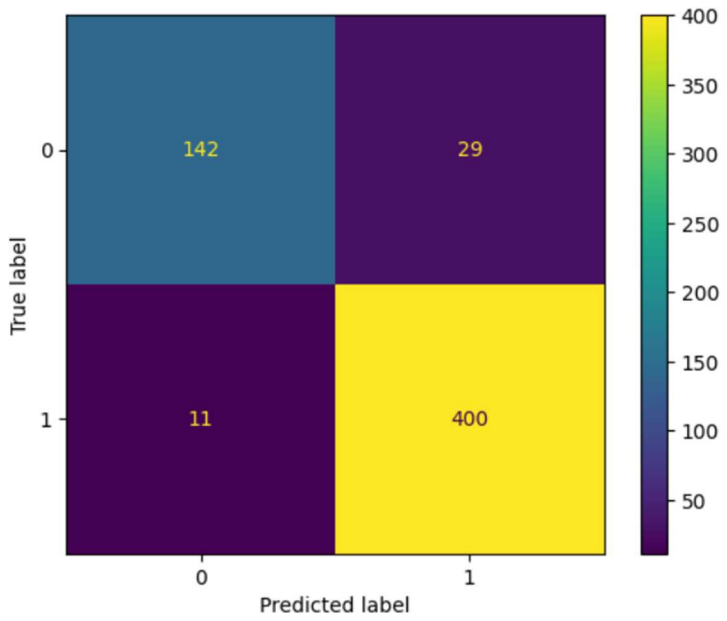

Figure 19 - Confusion matrix of model 20's predictions when applied to the test dataset. Model 20 is a linear model, with two layers and 10 hidden units (nodes). No weight was applied to the loss function and SGD optimizer was used.

Figure 20 shows the models performance in the different evaluation metrics. Model 20 (Linear, no weight), Model 21 (Linear + ReLU, no weight) and Model 22 (Linear + ReLU, weight) had the best performance, with a F1 score of 0.95 and accuracy around 93%.

	Class	Weighed?	Accuracy	Precision	Recall	F1 Score
model_20	Linear	no	93.4	0.93	0.97	0.95
model_21	Linear+ReLU	no	93.4	0.92	0.98	0.95
model_24	Linear+ReLU	yes	93.1	0.93	0.97	0.95
model_22	CNN	no	91.3	0.94	0.95	0.95
model_23	Linear	yes	91.1	0.96	0.92	0.94
model_25	CNN	yes	90.2	0.92	0.95	0.93

Figure 20 - Accuracy, precision, recall and F1 scores obtained for Linear, Linear + ReLU and CNN models, with/without a weight being applied to the loss function, when predicting new data in the test dataset. Models trained for 50 epochs.

When comparing models that were trained with and without a weighted loss function (Fig. 21), it was observed that making use of the *pos_weight* parameter in the BCEWithLogitLoss() function lead to both a decrease not only in accuracy, but also in the F1 score.

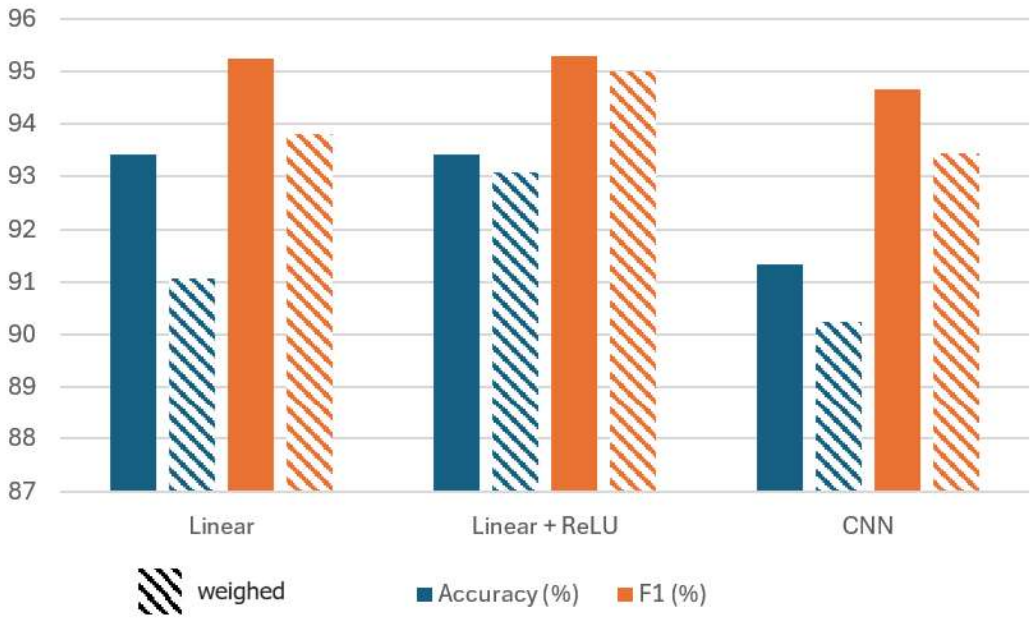

Figure 21- Accuracy and F1 scores for Linear, Linear + ReLU and CNN models (models 20 to 25). Solid colours indicate loss function with no class weights; hashed indicates weighted loss function. Models were trained for 50 epochs.

3.4. The batch size in dataloaders

To assess the impact of batch size on stability, batch size in the dataloaders were increased from 32 to 64. In Fig 22, the loss and accuracy curves of Linear, Linear + ReLU and CNN models are shown. CNN model presented a small improvement in stability after increasing batch size. Though the fluctuations in the loss and accuracy curves remain, they are of smaller intensity. The curves still indicate overfitting in the CNN model, so early stopping might be needed.

The increase in batch size did not lead to an increase in performance. In fact, when evaluated with the test set, the Linear models had the worst accuracy among all models. However, the metrics of the Linear + ReLU models remained virtually unchanged, with accuracy still above 93% and F1 score of 0.95 (Fig 23). Applying class weight in the loss function again did not result in any increase in accuracy or F1 score (Fig 24).

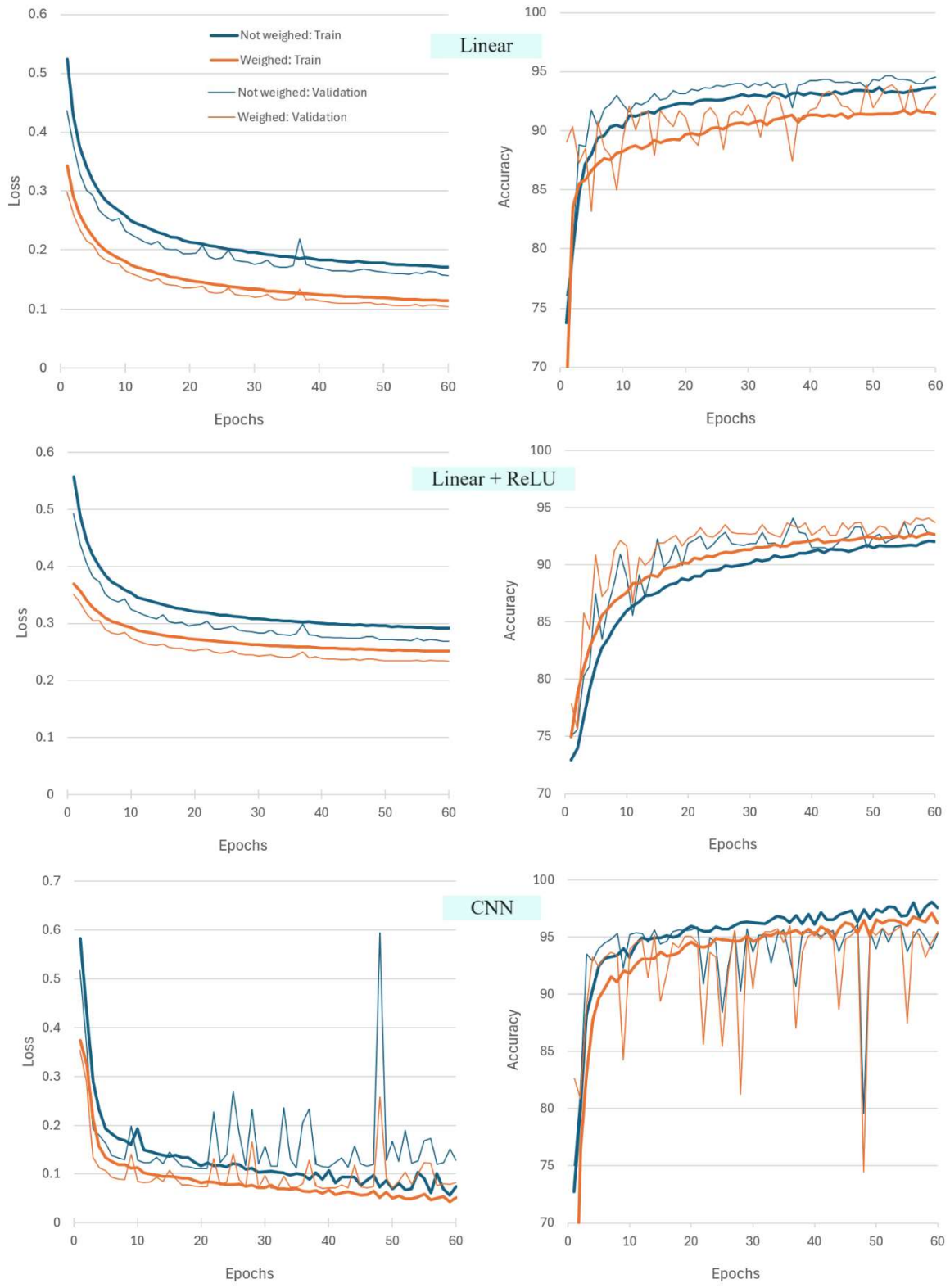

Figure 22 - Loss and accuracy curves of Linear, Linear + ReLU and CNN models when using batch size of 64 and training for 60 epochs. 10 hidden units and the SGD optimizer were used. Models with and without class weight in the loss function are shown.

	Class	Weighed?	Accuracy	Precision	Recall	F1 Score
model_24b	Linear+ReLU	yes	93.3	0.94	0.96	0.95
model_21b	Linear+ReLU	no	93.1	0.91	0.99	0.95
model_22b	CNN	no	92.4	0.94	0.97	0.95
model_25b	CNN	yes	92.2	0.95	0.96	0.95
model_20b	Linear	no	91.8	0.93	0.97	0.95
model_23b	Linear	yes	90.2	0.96	0.91	0.93

Figure 23 - Accuracy, precision, recall and F1 scores obtained for the Linear, Linear + ReLU and CNN models when using batch size 64 in dataloaders and trained for 60 epochs.

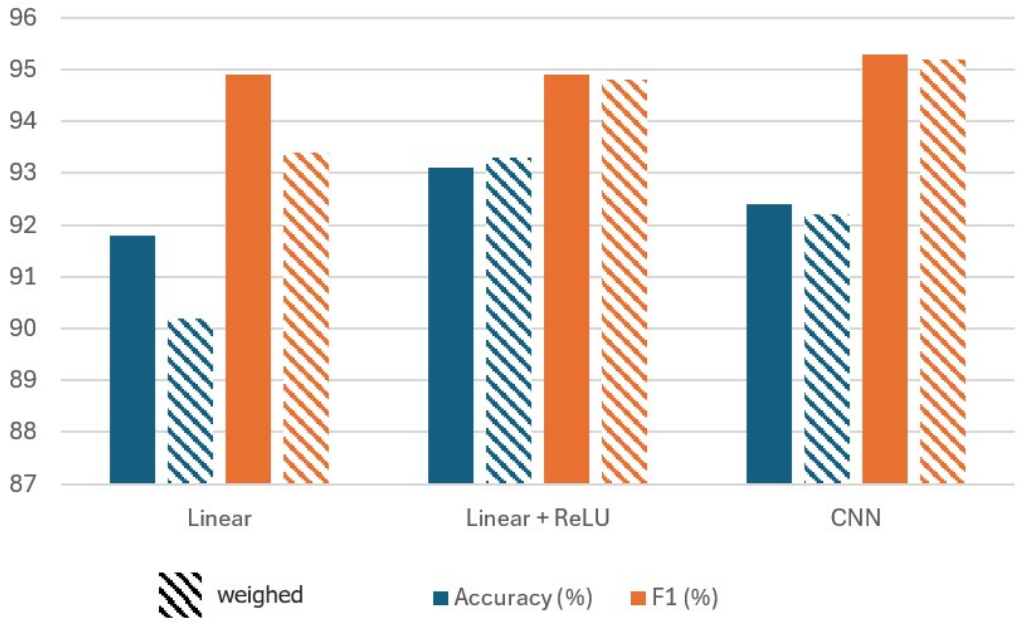

Figure 24 - Accuracy and F1 scores for Linear, Linear + ReLU and CNN models (20b to 25b), using batch size of 64 and 60 epochs. Solid colours indicate loss function with no weights; hashed indicates weighted loss function.

3.5. Training duration

It is also important to note that throughout this work, the CNN models took much longer to train that the Linear and Linear + ReLU models. To exemplify, Fig 25 shows training times for the three model classes trained for 60 epochs, with batch size 64. High computational cost is one of the main drawbacks of CNN models [6].

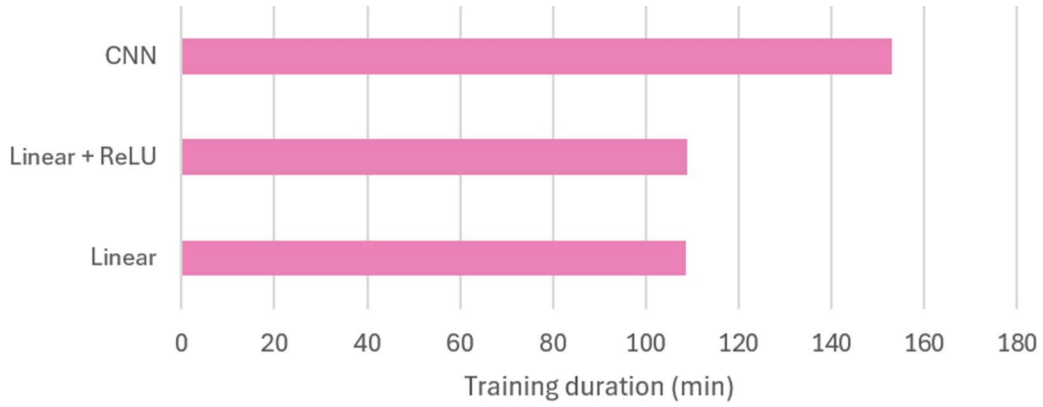

Figure 25 - Training duration in minutes for models 20b (Linear), 21b (Linear + ReLU) and 22b (CNN), trained for 60 epochs, with batch size 64.

4. Conclusion

A simple and shallow multilayer perceptron, using two linear layers and ReLU as activation function, was able to give good predictions of pneumonia using chest X-ray images (accuracy ~93%, F1 score 0.95). This class of model showed to be the most consistent, presenting similar metrics independently of applying weight to the loss function or changing batch size. Within this simple experiment, it outperformed CNN models, which have a more sophisticated architecture. Further tuning of hyperparameters is needed to make use of the CNN's full potential [6]. Data augmentation, i.e. generation of artificial data, could help to reduce overfitting and class imbalance [18]. Nonetheless, all models, even simple linear neural networks, were able to give descent predictions with accuracy above 90%. This demonstrates the power that even the less complex neural networks have as image classifiers.

5. References

[1] K. Berke, “Chest-xray-classification,” *HuggingFace.co*, Feb. 22, 2023. Available: https://huggingface.co/datasets/keremberke/chest-xray-classification. [Accessed: Aug. 24, 2024].

[2] D. Bourke, "Learn PyTorch for Deep Learning: Zero to Mastery," *LearnPyTorch.io*. Available: https://www.learnpytorch.io/. [Accessed: Aug. 24, 2024].

[3] Hugging Face, "Datasets process documentation," *HuggingFace.co*. Available: https://huggingface.co/docs/datasets/process. [Accessed: Aug. 24, 2024].

[4] PyTorch, "Convert RGB image to grayscale," *PyTorch.org*. Available: https://pytorch.org/vision/main/generated/torchvision.transforms.functional.rgb_to_grayscale.html. [Accessed: Aug. 24, 2024].

[5] PyTorch, "Loading data in PyTorch," *PyTorch.org*. Available: https://pytorch.org/tutorials/beginner/basics/data_tutorial.html. [Accessed: Aug. 24, 2024].

[6] F. Thiele, A. J. Windebank, and A. M. Siddiqui, “Motivation for using data-driven algorithms in research: A review of machine learning solutions for image analysis of micrographs in neuroscience,” *Journal of Neuropathology & Experimental Neurology*, vol. 82, no. 7. Oxford University Press (OUP), pp. 595–610, May 27, 2023. Doi: 10.1093/jnen/nlad040.

[7] V. Bhattbhatt, "Learning rate and its strategies in neural network training," *Medium*, Feb. 9, 2021. Available: https://medium.com/thedeephub/learning-rate-and-its-strategies-in-neural-network-training-270a91ea0e5c. [Accessed: Aug. 24, 2024].

[8] PyTorch, "torch.nn.ReLU," *PyTorch.org*. Available: https://pytorch.org/docs/stable/generated/torch.nn.ReLU.html. [Accessed: Aug. 24, 2024].

[9] R. Qayyum, "Introduction to pooling layers in CNN," *Towards AI*, Jun. 20, 2021. Available: https://towardsai.net/p/l/introduction-to-pooling-layers-in-cnn. [Accessed: Aug. 24, 2024].

[10] PyTorch, "torch.optim.SGD," *PyTorch.org*. Available: https://pytorch.org/docs/stable/generated/torch.optim.SGD.html. [Accessed: Aug. 24, 2024].

[11] PyTorch, "torch.optim.Adam," *PyTorch.org*. Available: https://pytorch.org/docs/stable/generated/torch.optim.Adam.html#torch.optim.Adam. [Accessed: Aug. 24, 2024].

[12] PyTorch, "torch.nn.BCEWithLogitsLoss," *PyTorch.org*. Available: https://pytorch.org/docs/stable/generated/torch.nn.BCEWithLogitsLoss.html. [Accessed: Aug. 24, 2024].

[13] T. Hangtao, "Use weighted loss function to solve imbalanced data classification problems," *Medium*, Jul. 19, 2019. [Online]. Available: https://medium.com/@zergtant/use-weighted-loss-function-to-solve-imbalanced-data-classification-problems-749237f38b75. [Accessed: Aug. 24, 2024].

[14] S. T. Anantha, "How to choose batch size and number of epochs when fitting a model," *GeeksforGeeks*, Apr. 17, 2020. Available: https://www.geeksforgeeks.org/how-to-choose-batch-size-and-number-of-epochs-when-fitting-a-model/. [Accessed: Aug. 24, 2024].

[15] R. Kundu, "F1 score guide," *V7 Labs*, Mar. 5, 2021. Available: https://www.v7labs.com/blog/f1-score-guide. [Accessed: Aug. 24, 2024].

[16] TorchMetrics, "F1 score," *TorchMetrics*. Available: https://torchmetrics.readthedocs.io/en/v0.8.2/classification/f1_score.html. [Accessed: Aug. 24, 2024].

[17] S. Park, "A 2021 guide to improving CNNs: Optimizers - Adam vs SGD," *Medium*, Oct. 10, 2021. Available: https://medium.com/geekculture/a-2021-guide-to-improving-cnns-optimizers-adam-vs-sgd-495848ac6008. [Accessed: Aug. 24, 2024].

[18] Amazon Web Services (AWS), "What is data augmentation?" [Online]. Available: https://aws.amazon.com/what-is/data-augmentation/. [Accessed: Aug. 24, 2024].