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]. Models with varied degrees of architectural complexity and different sets of hyperparameters were built to investigate how these affect model training and performance. Good predictions (accuracy ~93%, F1 score ~95%), were obtained using relatively simple neural network models.

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.

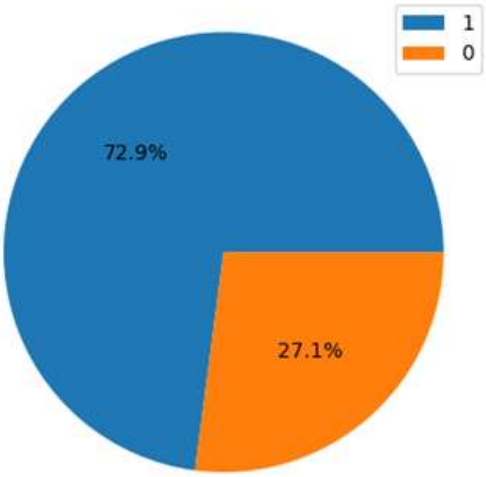

Figure 3 – Percentages of 0 (NORMAL) and 1 (PNEUMONIA) classes in training set.

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.

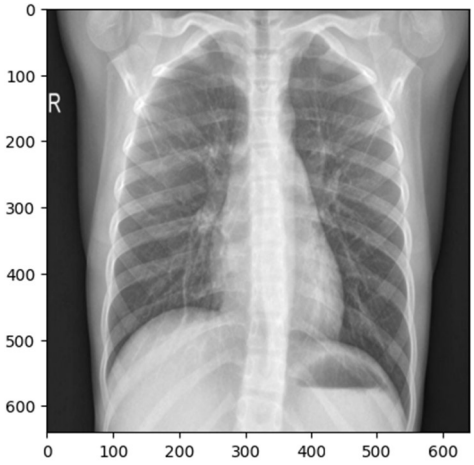

Figure 4 - Image after transformation.

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 sets were split into batches of 32 or 64 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 data loaders.

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 nodes in one layer are connected to the nodes 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 1 hidden layer 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. **MLP (Linear + ReLU) models:** These are multilayer perceptrons that use the rectifier activation function (ReLU) to introduce non-linearity [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 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 convolutional layers that generate a feature map using the filters, a pooling layer to downsample the feature map and reduce overfitting, and a fully connected layer [9, 6]. Model architecture of TinyVGG from CNN Explainer was used (Fig 8) [10]. There are numerous hyperparameters that can be optimized in CNN models [6]. In the present work, models were built with different the number of filters (hidden units) and the size of filters (kernel size). The filter size in the pooling layer was kept as 2x2. Max pooling was used, which takes the highest value covered by the filter. For 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 2 convolutional blocks, containing 10 hidden units (filters) of size 3x3.

Models were selected to vary other parameters further:

i. **The optimizer:** The main optimizer used was SGD, implementing stochastic gradient descent [11]. For comparison, a few models were trained with the Adam algorithm [12]

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 [13, 14]. Since the 0/1 proportion in the training set is of 0.37, this value is set as the class weight in some models.

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 [15]. Overfitting happens when the model learns the training data so well, that it has difficulty adapting to new, unseen data [16] .

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

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 [15].

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

Model	Class	Hidden layers	Hidden units	Kernel size in Cov2D	Class weight	Optimizer	Learning rate	Epochs	Weight decay
20b	Linear	1	10		no	SGD	10^{-4}	60	0
21b	Linear+ReLU	1	10		no	SGD	10^{-4}	60	0
22b	CNN	5	10	3x3	no	SGD	10^{-2}	60	0
23b	Linear	1	10		yes	SGD	10^{-4}	60	0
24b	Linear+ReLU	1	10		yes	SGD	10^{-4}	60	0
25b	CNN	5	10	3x3	yes	SGD	10^{-2}	60	0
26b	CNN	5	10	3x3	No	SGD	10^{-2}	60	0.1
27b	Linear	1	10		No	SGD	10^{-4}	60	0.1
28b	Linear+ReLU	1	10		no	SGD	10^{-4}	60	0.1

There are several strategies to tackle overfitting, including early stopping, data augmentation, and regularization techniques [16]. In the present work, L2 regularization was used to reduce overfitting of the CNN model. This technique penalizes the sum of the squares of the weights, forcing them to be small and reducing the influence of less significant features [17]. It was implemented using the *weight_decay* parameter on the SGD optimizer [11]. Reasonable values range between 0 and 0.1 [18]. Higher it is, less overfitting, but the model could be less powerful.

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 [19]. 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

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

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

$$\text{F1 Score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision+Recall}} \text{ [20].}$$

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 train/validation loss and accuracy curves for linear models with 10 hidden units (nodes) and number of hidden layers varying from 1 to 3. 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 hidden layers as 1, 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.

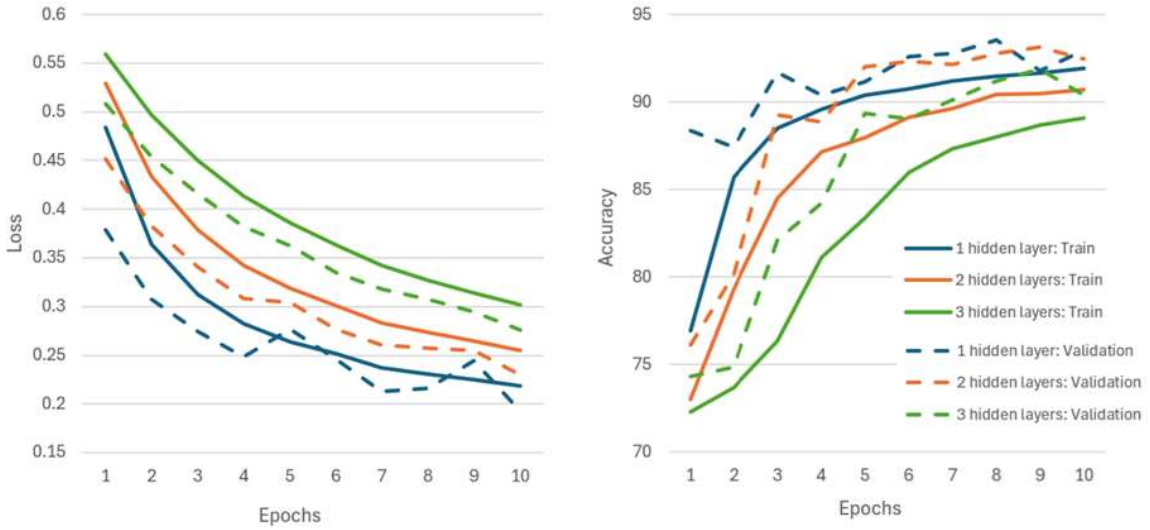

Figure 9 - Loss and accuracy curves for models 0, 1 and 2, with respectively 1, 2 and 3 hidden layers. The three models have 10 hidden units (nodes). SGD optimizer and pos_weight of 1 (default) were used.

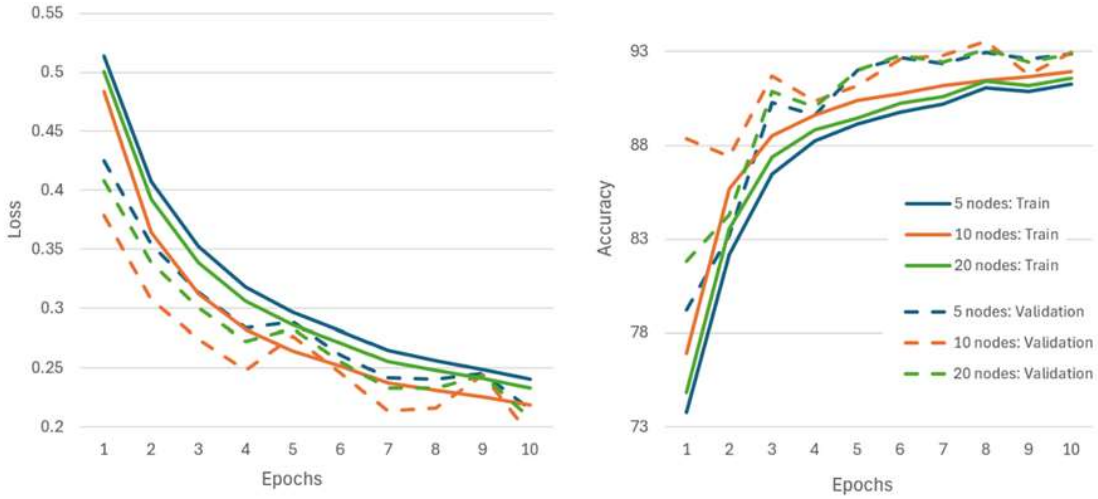

Figure 10 - Loss and accuracy curves for linear models 3, 0 and 4, with respectively 5, 10 and 20 hidden units (nodes). The three models have 2 linear layers. SGD optimizer and pos_weight of 1 (default) were used.

MLP (Linear + ReLU) models: When adding the ReLU activation function, the algorithm only converged with 1 hidden layer. Converged was not obtained with 2 or 3 hidden layers even when trying to adjust the learning rate with values from 10^{-1} to 10^{-5}. Maintaining a single hidden layer, the number of nodes were then varied. Faster optimization and higher accuracy were obtained with 10 nodes, as can be seen in Fig. 11.

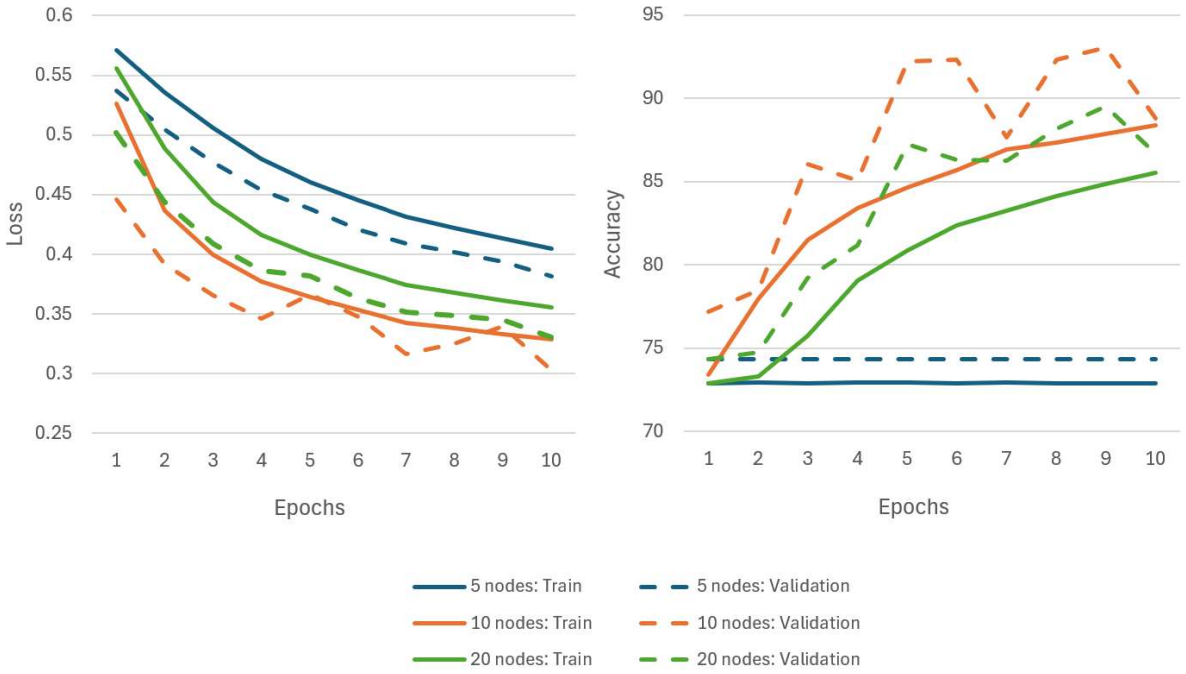

Figure 11 - Loss and accuracy curves for MLP models number 5, 8 and 9, with respectively 5, 10 and 20 hidden units (nodes). The three models have 1 hidden layer. SGD optimizer and pos_weight of 1 (default) were used.

CNN models: CNN models with kernel size 3x3 in the convolution layers were built with 5, 10, and 20 hidden units (filters). The validation loss curves have more oscillation than the ones of the previous models (Fig 12). Models with 5 and 10 filters have similar loss and accuracy curves, outperforming the model with 20 filters. It was chosen to fix the number of filters as 10. A CNN model with kernel size 5x5 in the convolution layer and 10 filters was built for comparison. The increase in kernel size led to less smooth loss curves and lower accuracy, as seen on Fig 13.

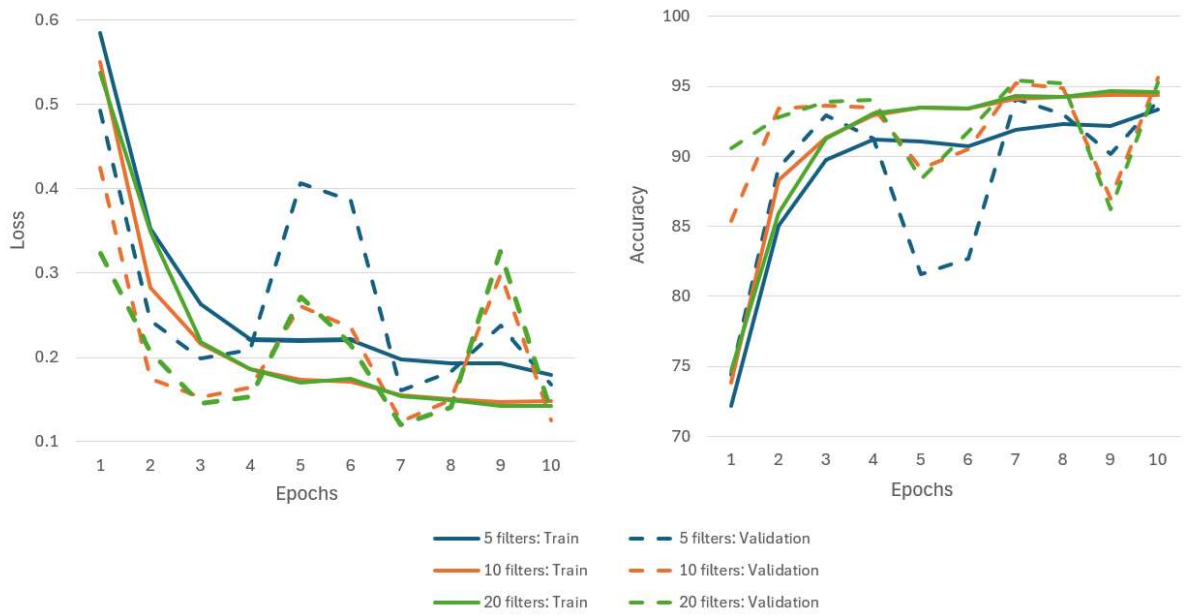

Figure 12 - Loss and accuracy curves of CNN models 10, 11 and 12, with respectively 5, 10 and 20 filters. The three models have kernel size 3x3 in the convolution layer. Optimizer was SGD and no class weights were used.

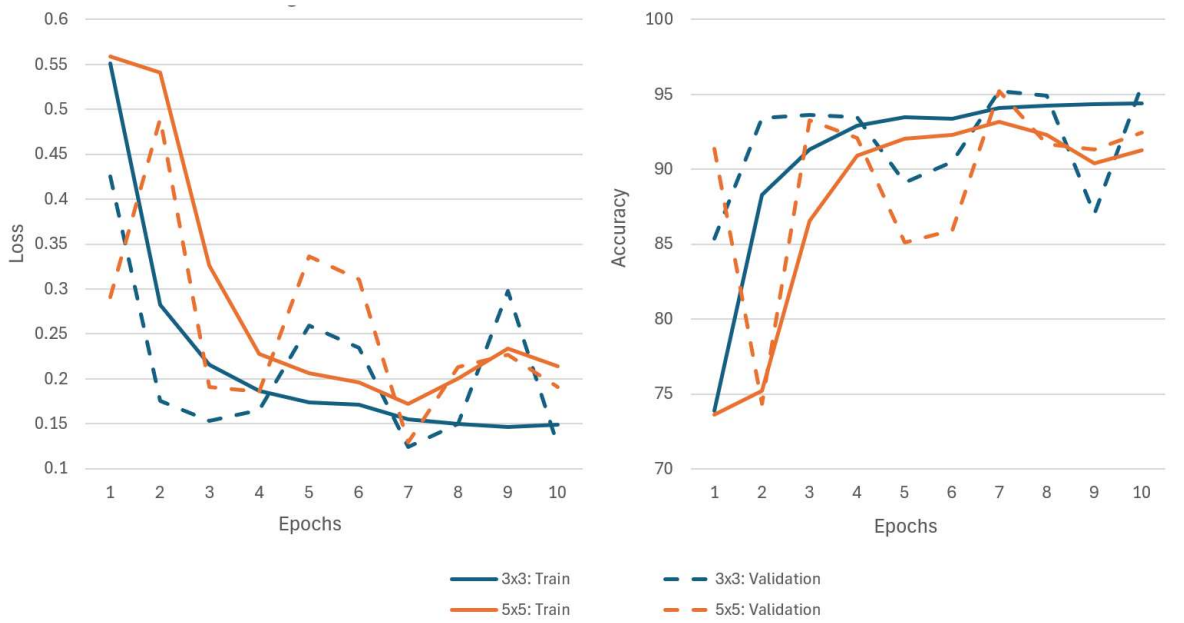

Figure 13 - Loss and accuracy curves for CNN models 11 and 13, with respectively 3x3 and 5x5 kernel size in the convolution layers. Both models have 10 filters. SGD optimizer and pos_weight of 1 (default) were used.

3.2. Weighted loss function and optimizer

In attempt to reduce any bias caused by class imbalance, the *pos_weight* parameter in the loss function was changed from 1 (default value) to 0.37 in some models. The loss and accuracy curves for the linear, MLP and CNN models are shown in Fig 14-16. For the linear model, the

change in class weight resulted in accuracy curve plateauing at a lower value. For both linear and CNN models, adjusting class weight resulted in more oscillations in validation accuracy. However, more epochs are needed for further assessment.

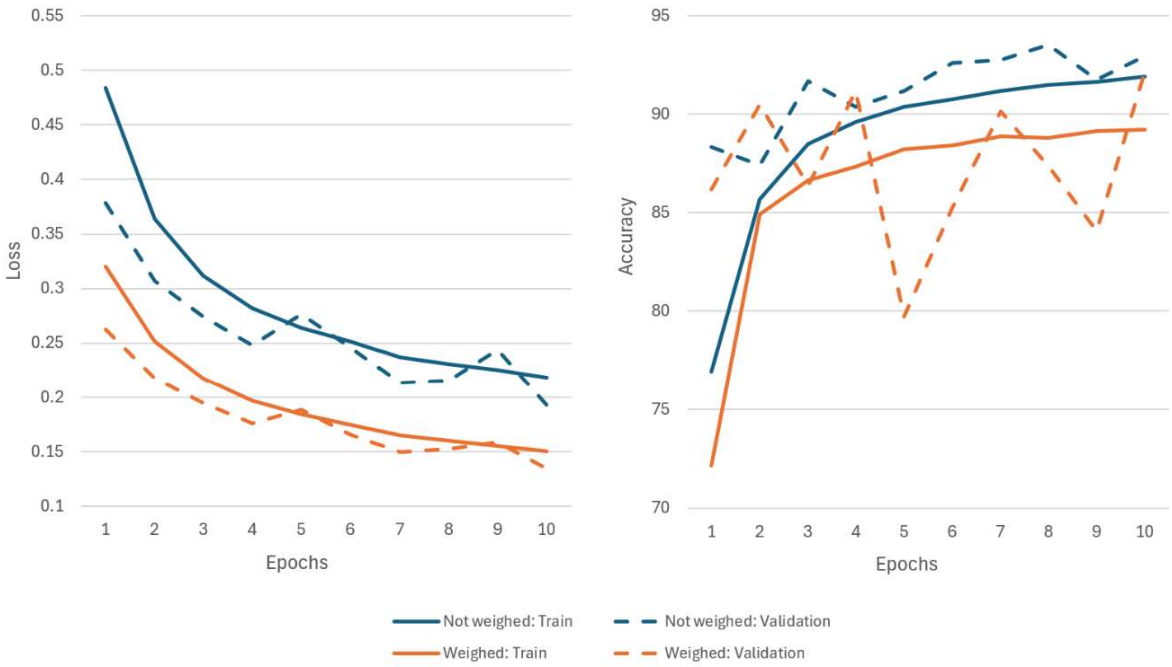

Figure 14 - Loss and accuracy curves for linear models 0 and 14 (1 h.l., 10 nodes), with pos_weight = 1 and pos_weight = 0.37 in the loss function, respectively. SGD optimizer was used.

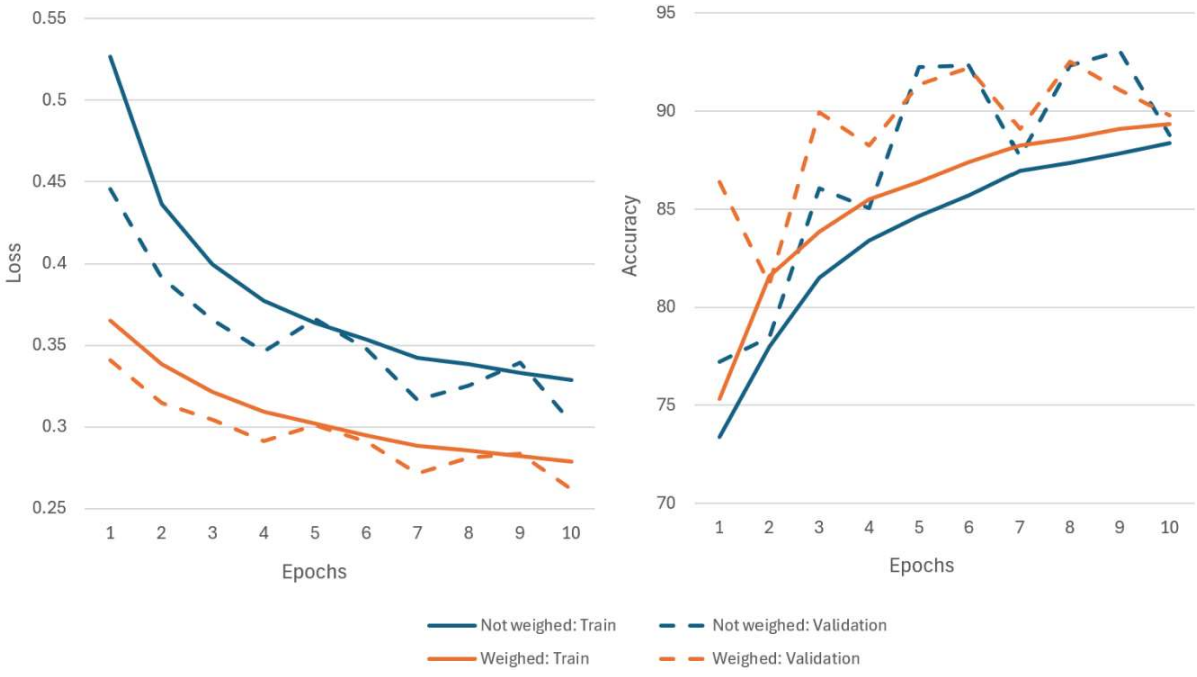

Figure 15 - Loss and accuracy curves for MLP models 5 and 15 (1 h.l., 10 nodes), with pos_weight = 1 and pos_weight = 0.37 in the loss function, respectively. SGD optimizer was used.

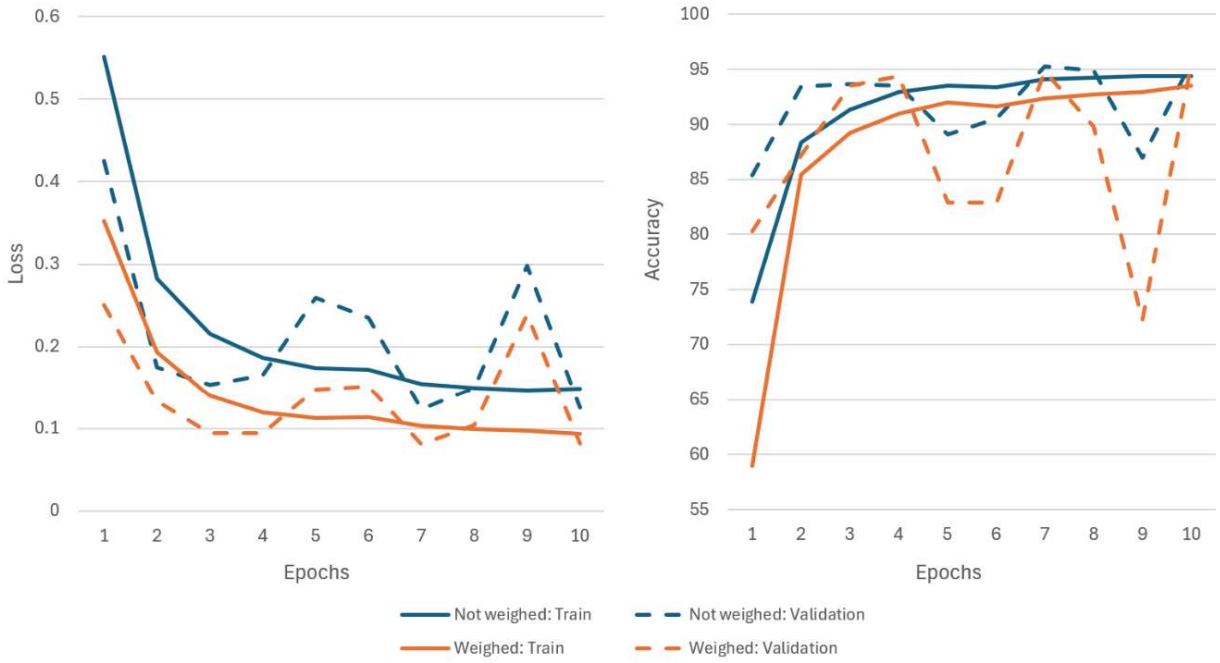

Figure 16 - Loss and accuracy curves for CNN models 11 and 19, with pos_weight = 1 and pos_weight = 0.37, respectively. Kernel size in convolution layer is 3x3 and the number of filters is 10. SGD optimizer was used.

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 MLP nor CNN models. Even for the linear model, it was very unstable (Fig 17). Generally, a fine-tuned Adam would outperform SGD [21]. Tuning the hyperparameters for Adam was not included in the present work but would make an interesting project continuation.

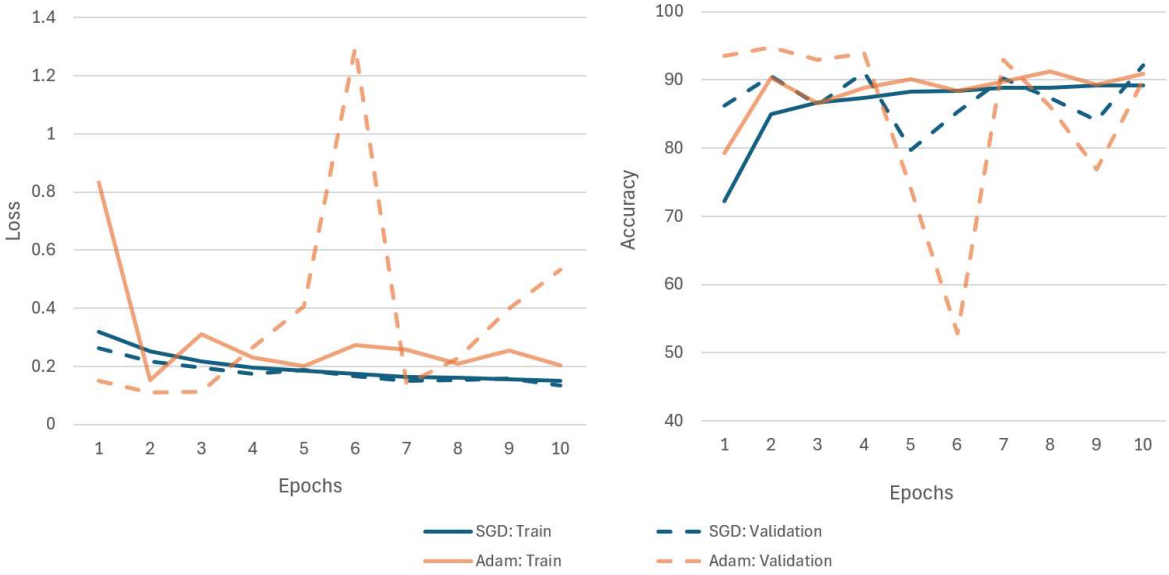

Figure 17 – Loss and accuracy curves for linear models 0 (SGD) and 19 (Adam). Both models have 1 hidden layer, 10 nodes, and pos_weight of 1 in the loss function.

3.3. Increasing number of epochs

After selecting the hyperparameter for different model classes:

- Linear – 1 hidden layer, 10 hidden units,
- MLP (Linear + ReLU) – 1 hidden layer, 10 hidden units,
- CNN – 10 hidden units, 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. Adding a class weight (*pos_weight* = 0.37) to the loss function did not change significantly the train/validation accuracy curves of the MLP model. However, in the case of the linear model, using weighted loss function led to a lower validation accuracy and more oscillations. CNN models were also more unstable and showing signs of overfitting, as the train loss continues to decrease while the baseline of the validation loss plateaus [16]. Model complexity could be too high for the nature of the problem.

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

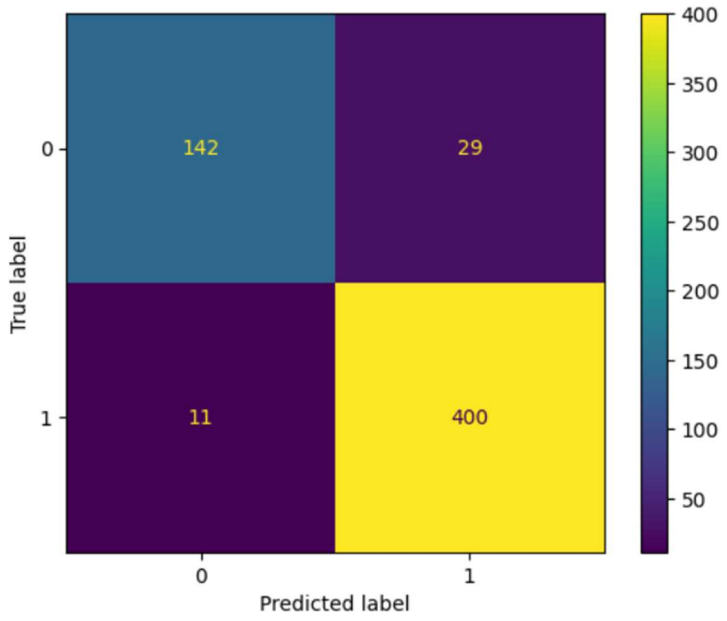

Figure 18 - Confusion matrix of model 20's predictions when applied to the test dataset. Model 20 is a linear model, with two layers and 10 hidden units (nodes). SGD optimizer and pos_weight of 1 (default) were used.

The models' performance in different evaluation metrics, sorted by the F1 score, is shown in Fig. 20. Model 21 (MLP, no class weight) and model 20 (linear, no class weight) had the best performance, with a F1 score above 95% and accuracy of 93.4%. It was observed that using a weighted loss function led to a decrease both in accuracy and the F1 score of the three classes of model (Fig. 21).

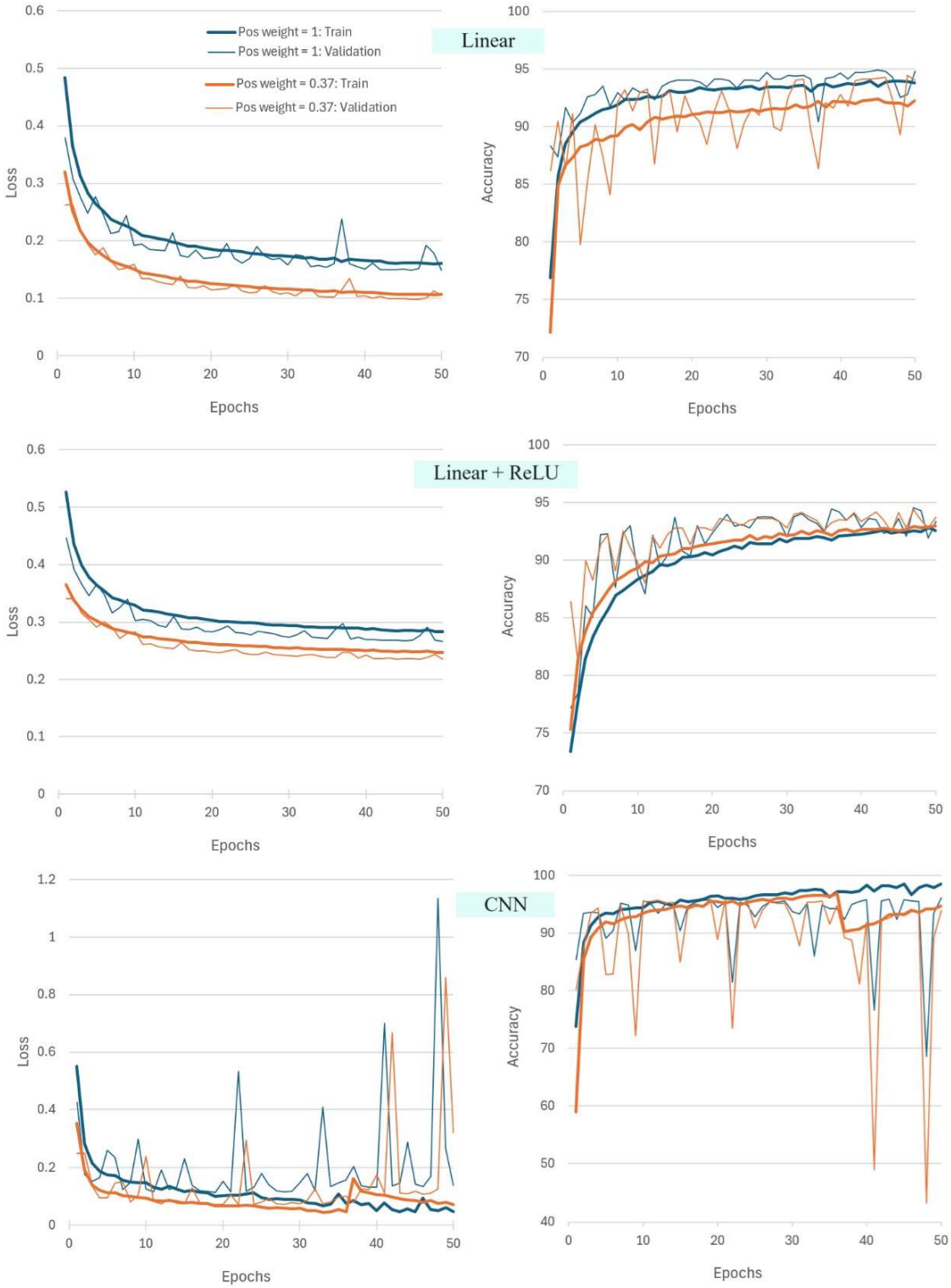

Figure 19 – Loss and accuracy curves for linear (1 h.l.), MLP (1 h.l.), and CNN (3x3 kernel in conv. l.) models. Blue curves represent models with pos_weight = 1 (default) and orange curves models with pos_weight = 0.37. SGD optimizer and 10 hidden units (nodes/filters) were used.

	Class	Pos weight	Precision(%)	Recall(%)	F1 Score(%)	Accuracy(%)
model_21	Linear+ReLU	1.00	92.4	98.3	95.3	93.4
model_20	Linear	1.00	93.2	97.3	95.2	93.4
model_24	Linear+ReLU	0.37	93.2	96.8	95.0	93.1
model_22	CNN	1.00	94.4	94.9	94.7	91.3
model_23	Linear	0.37	95.7	92.0	93.8	91.1
model_25	CNN	0.37	91.6	95.4	93.4	90.2

Figure 20 - Accuracy, precision, recall and F1 scores obtained for linear, MLP and CNN models, when predicting new data in the test dataset. Models were trained for 50 epochs.

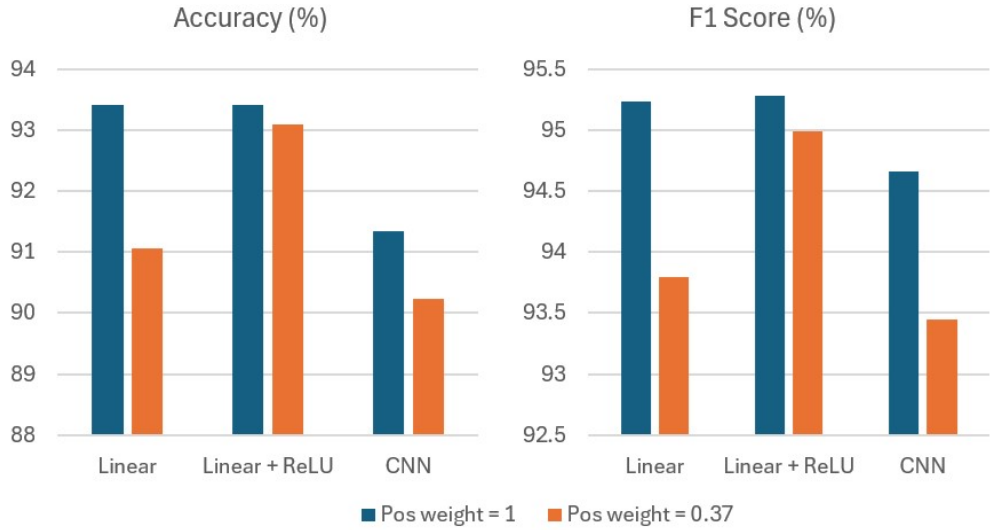

Figure 21- Accuracy and F1 scores for linear, MLP and CNN models, with pos_weight of 1 (default) and 0.37 (considering class imbalance). Models were trained for 50 epochs.

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, MLP and CNN models are shown. After increasing batch size, the fluctuations in the loss and accuracy curves of the CNN model decreased in intensity. However, the curves still indicate overfitting, with the validation loss plateauing while the train loss continues to decrease.

Models' performance when making predictions with the test set is shown in Fig 23. Most models have F1 score around 95%, with the MLP having highest accuracy (~93%). Applying class weight in the loss function again did not result in any increase in accuracy or F1 score (Fig 24).

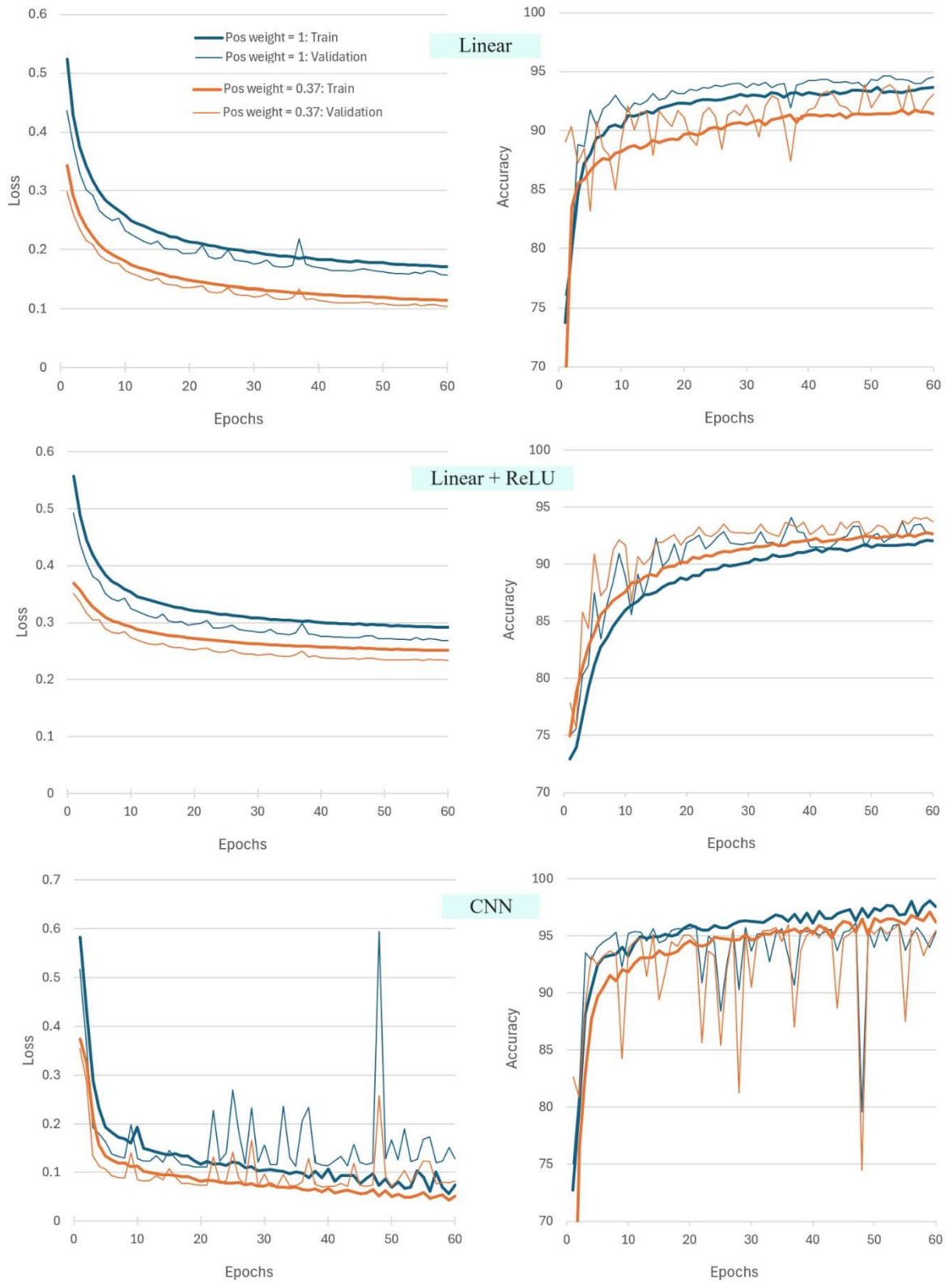

Figure 22 - Loss and accuracy curves of linear, MLP and CNN models when using batch size of 64. Blue curves represent models with pos_weight = 1 (default) and orange curves models with pos_weight = 0.37. SGD optimizer and 10 hidden units (nodes/filters) were used.

	Class	Pos weight	Precision(%)	Recall(%)	F1 Score(%)	Accuracy(%)
model_22b	CNN	1.00	93.9	96.8	95.3	92.4
model_25b	CNN	0.37	94.7	95.6	95.2	92.2
model_20b	Linear	1.00	93.2	96.6	94.9	91.8
model_21b	Linear+ReLU	1.00	91.4	98.5	94.8	93.1
model_24b	Linear+ReLU	0.37	93.6	96.1	94.8	93.3
model_23b	Linear	0.37	95.9	91.0	93.4	90.2

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

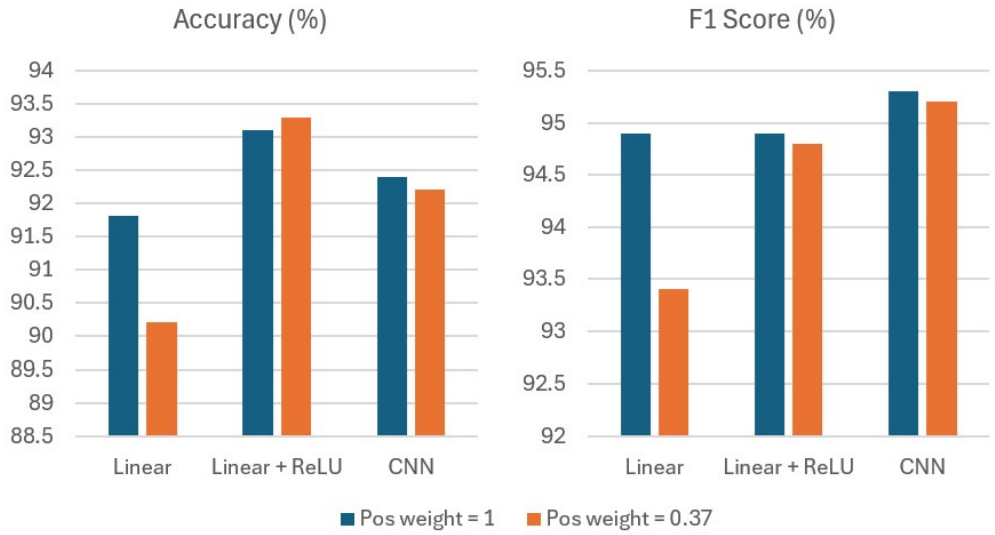

Figure 24 - Accuracy and F1 scores for linear, MLP and CNN models trained and evaluated with batch size 64, with pos_weight of 1 (default) and 0.37 (considering class imbalance). Models were trained for 60 epochs.

3.5.L2 regularization

The three classes of models were submitted to L2 regularization in the "Chest_XRays_Training_and_Evaluation_BatchSize64" notebook. The evaluation scores of these models are shown in Fig. 25.

The loss and accuracy curves of CNN model with and without L2 regularization are shown in Fig. 26. Without regularization, the validation loss remains constant while the train loss decreases. Likewise, the validation accuracy stops increasing, while train accuracy continues. When applying regularization, the validation and train curves have now the same shape, and the curves' oscillations decrease. This indicates that overfitting has been successfully reduced. The regularization technique led to improved metrics in both the CNN and linear models (Fig.27).

	Class	Pos weight	Precision(%)	Recall(%)	F1 Score(%)	Accuracy(%)
model_26b	CNN	1	94.7	96.6	95.7	92.9
model_27b	Linear	1	93.2	96.8	95.0	93.4
model_28b	Linear+ReLU	1	91.4	98.5	94.8	93.1

Figure 25 - Accuracy, precision, recall and F1 scores obtained for the linear, MLP (Linear + ReLU) and CNN models submitted to L2 regularization. Batch size of 64 was used in data loaders. Models trained for 60 epochs.

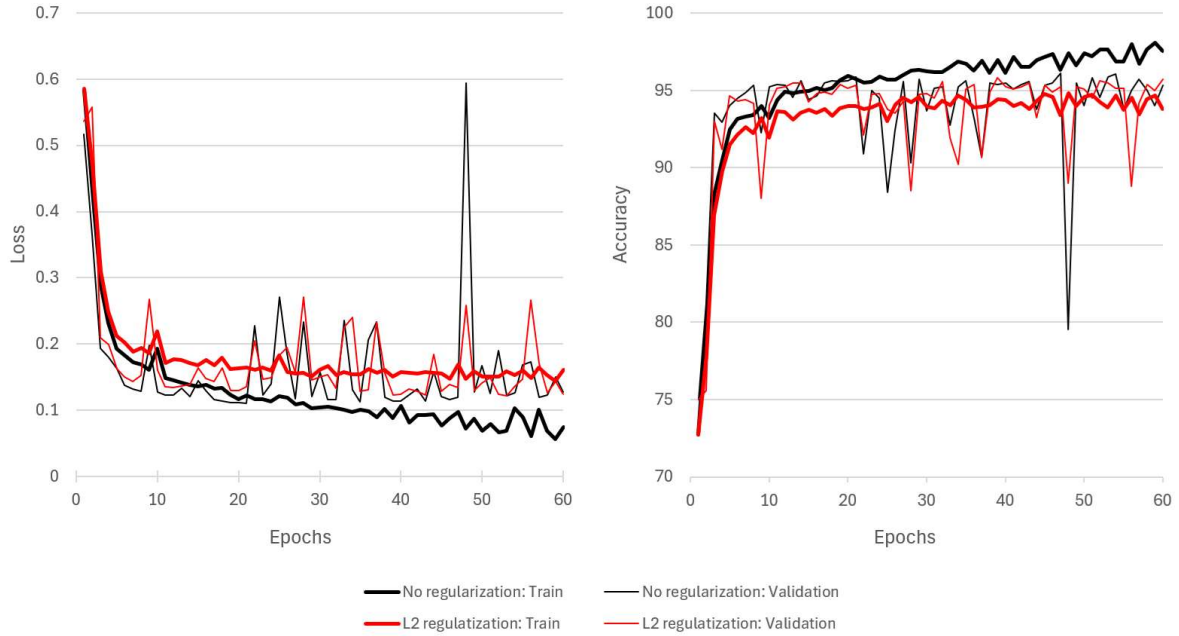

Figure 26 – Loss and accuracy curves for CNN models with (26b) and without L2 regularization (22b). Batch size of 64 was used in the data loaders. No class weight applied.

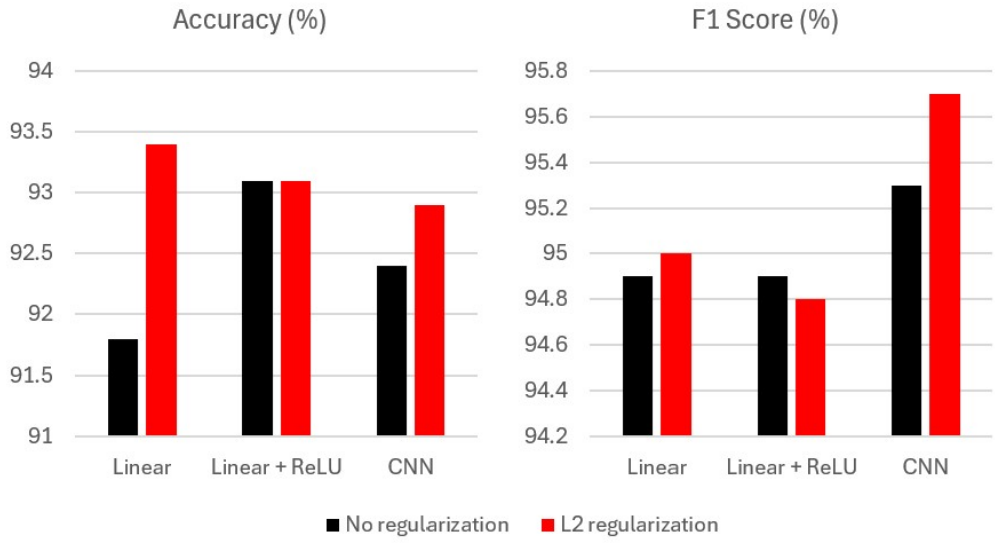

Figure 27 – Accuracy and F1 scores for Linear, MLP and CNN models with and without L2 regularization. Models were trained for 60 epochs, using batch size 64.

3.6. Training duration

It is also important to note that throughout this work, the CNN models took much longer to train that the linear and MLP models. To exemplify, Fig 25 shows training times for the three model classes trained for 60 epochs, with batch size 64 and L2 regularization. High computational cost is one of the main drawbacks of CNN models [6].

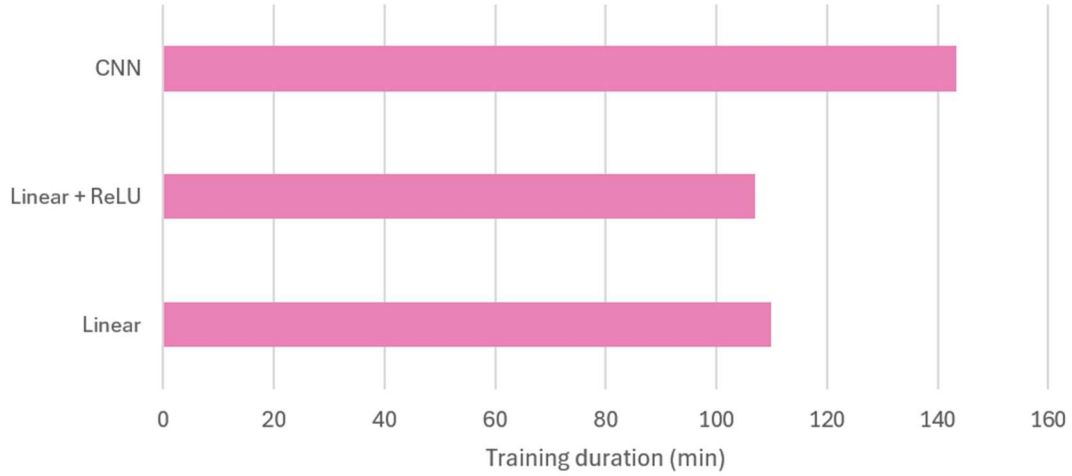

Figure 28 - Training duration in minutes for Linear (27b), Linear + ReLU (28b) and CNN (26b) models.

4. Conclusion

A simple linear model, containing a single hidden layer with 10 nodes, was able to give good predictions of pneumonia using chest X-ray images (accuracy ~93%, F1 score ~ 95%). Increase in model architectural complexity did not result in significant increase in performance. Multilayer perceptron models, using ReLU as activation function, were the most consistent, presenting similar metrics independently of batch size or class weights. F1 score was slightly higher for the convolutional neural network once L2 regularization was used to prevent overfitting. However, this increase (<+1%) does not compensate for significantly higher computational cost (>+20%). Further tuning of the great number hyperparameters [6], as well as data augmentation techniques [22], might be needed to make use of the CNN's full potential. Nonetheless, this work demonstrates that even simple neural networks can be powerful image classifiers depending on the task at hand.

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] Wang, J., Turko, R., Shaikh, O., Park, H., Das, N., Hohman, F., Kahng, M. and Chau, P. *CNN Explainer*. [online] poloclub.github.io. Available at: https://poloclub.github.io/cnn-explainer/. [Accessed: Set. 04, 2024].

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

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

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

[14] 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].

[15] 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].

[16] Google for Developers, "Overfitting," *ML Concepts (2024).* Available at: https://developers.google.com/machine-learning/crash-course/overfitting/overfitting. [Accessed Sep. 4, 2024].

[17] A. Bharti, "What Are the Possible Approaches to Fixing Overfitting on a CNN?," *GeeksforGeeks,* Feb. 19, 2024. Available at: https://www.geeksforgeeks.org/what-are-the-possible-approaches-to-fixing-overfitting-on-a-cnn/. [Accessed Sep. 4, 2024].

[18] J. Brownlee, "How to Use Weight Decay to Reduce Overfitting of Neural Network in Keras," *Machine Learning Mastery*, Aug. 25, 2020. Available at: https://machinelearningmastery.com/how-to-reduce-overfitting-in-deep-learning-with-weight-regularization/. [Accessed Sep. 4, 2024].

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

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

[21] 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].

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