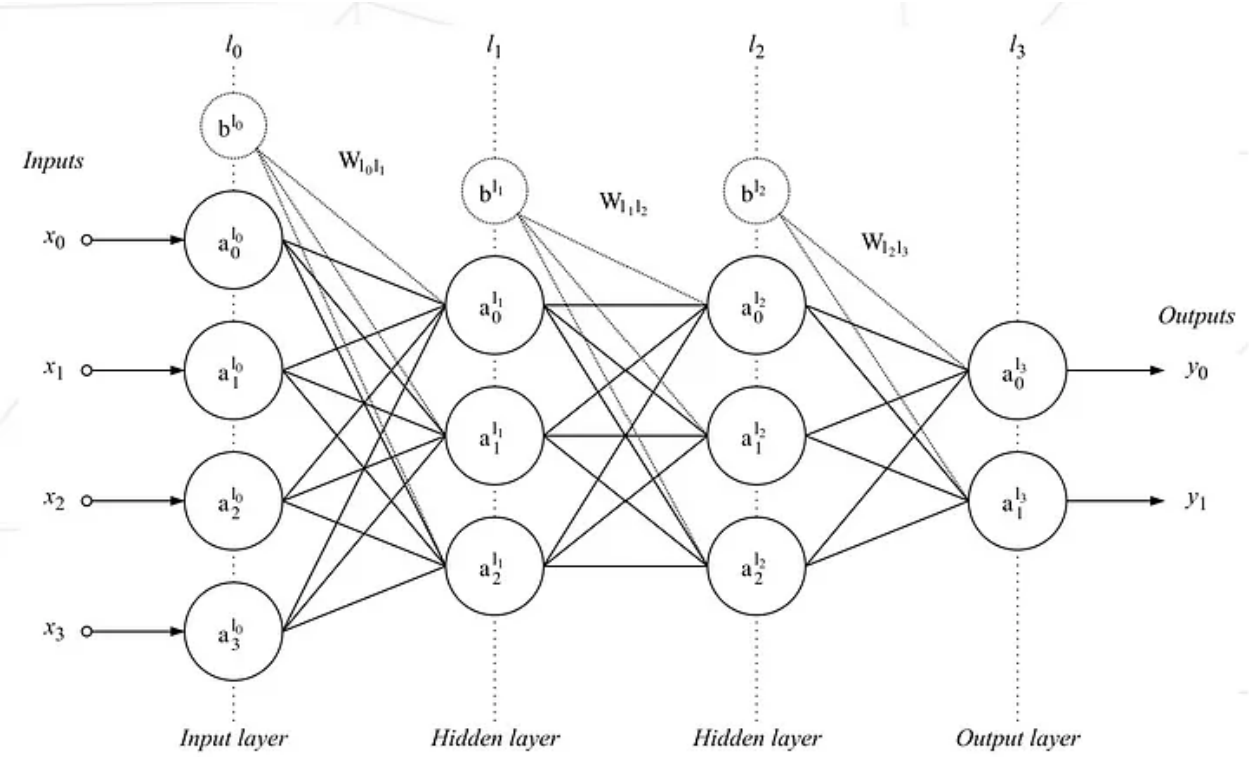
Learning AI through Pneumonia Image Classification

Tags



I - Introduction
II - Basics of AI
- A. Machine Learning
  - 1) Description of the Dataset
- B. Activation functions
  - **1) Types of Neural Networks Activation Functions**
  - 2) Forward and Backward Propagation
  - 3) O**ptimization techniques**

III - Models
- **A. Rosenblatt Perceptron**
  - 1) How does a Perceptron Learn?
  - 2) Classifying Images using the Perceptron

B. Multi-Layer Perceptron
C. Logistic regression
D. **Convolutional Neuronal Network**
1) Data-augmentation
2) Cross-Validation
IV - Models comparison
A. Metrics
1) Metrics for the perceptron
2) Metrics for the CNN and logistic regression
B. Perceptron
1) No Scheduler
2) Exponential Decay
3) StepDecay
C. Multi Layer Perceptron
D. Multi-class models
1) CNN
2) CNN - pseudo ResNet
3) Logistic regression
4) Overall comparison
V - Cloud
A. Azure Bucket
B. Cloud computing
C. The price
D. Deployment
VI - Conclusion

I - Introduction

Recently, we started witnessing a huge expansion of Artificial Intelligence (AI) and it's numerous applications. One of these is healthcare, where we can use a method known as computer vision to help professionals reach a more accurate diagnosis. In this article, we explore how machine learning and deep learning algorithms can be used to help with the diagnosis of pneumonia from chest X-Rays. Our goal with these models is to reach a point where one can outperform a professional with regard to his specific diagnosis.

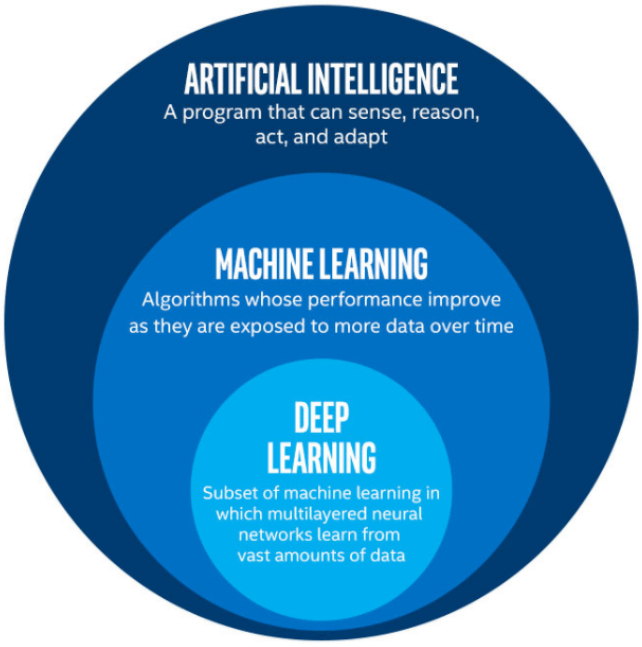
We will first explain the basics of machine learning and deep learning necessary to

understand the mechanisms that hide underneath what we call AI. Then, we will describe the several models we used during our research and how well they perform in comparison to each other. We will end by explaining how one can use cloud computing to scale and distribute a model.

II - Basics of AI

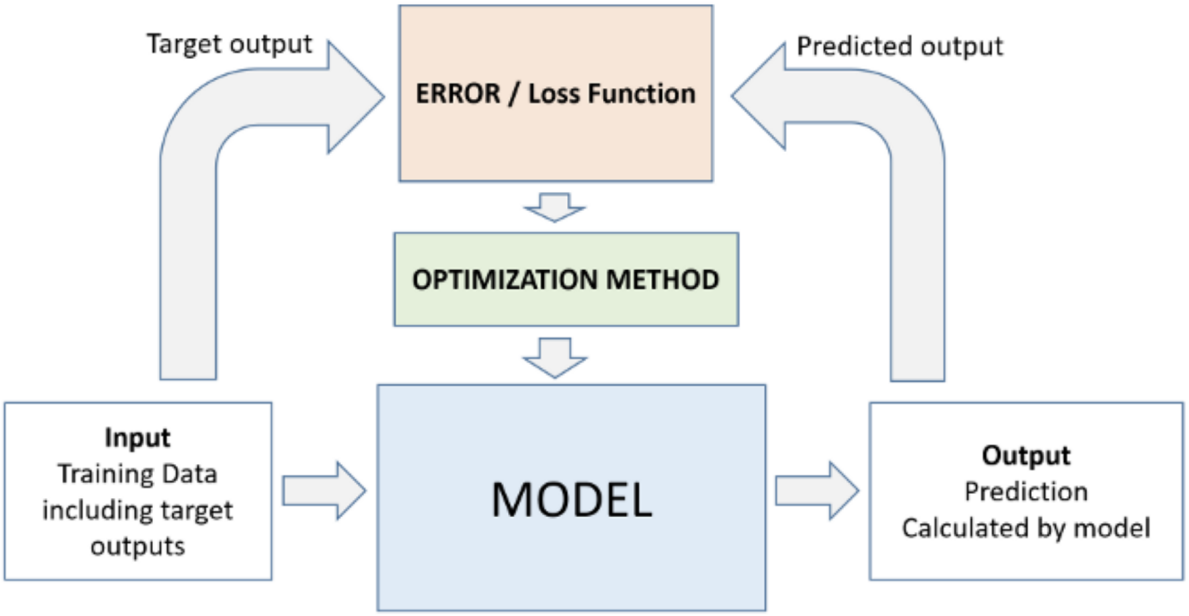
Artificial intelligence is a broad field of computer science dedicated to creating systems capable of performing tasks that typically require human intelligence. These tasks include reasoning, learning, problem-solving, perception, language understanding, and decision-making. AI systems leverage various techniques, such as machine learning, natural language processing, and robotics, to interpret complex data, recognize patterns, and adapt to new information. By mimicking cognitive functions, AI aims to automate and enhance a wide range of processes, from everyday tasks like virtual assistants and customer service chatbots to more complex applications in healthcare, finance, and autonomous driving, ultimately transforming industries and impacting our daily lives.

AI can be broken down into two other subfields, Machine Learning and Deep Learning.



A. Machine Learning

Machine learning is a concept based on learning from mistakes.



The machine learning process goes as follow:

1. We throw inputs in our model. This input is based on a dataset.
2. Our model, which is basically a function, outputs a prediction
3. The output is sent to a loss function to calculate how far is the output from the target output, by comparing both. The loss function then outputs a value, the higher it is and the worse our model is. The lower that value is, the better our model.
4. This value will modify our model through the optimization steps. By knowing how far the model is from correctly answer, the model can adapt its parameters (weight and biases) to make better predictions in the future.
5. We come back to step 1.

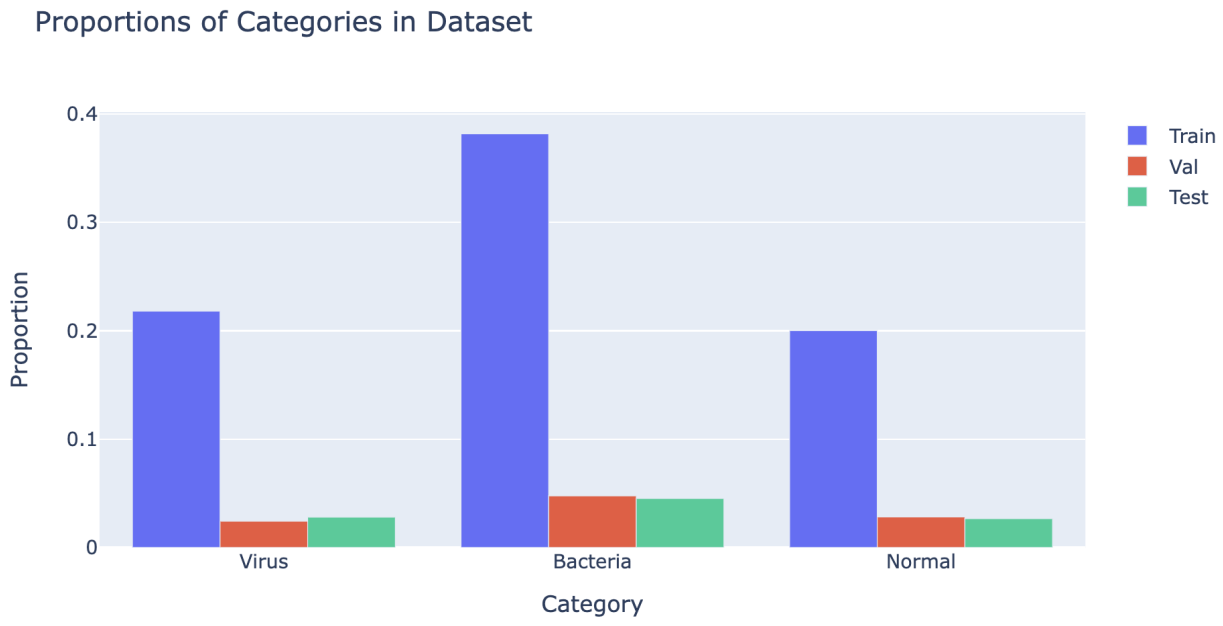
1) Description of the Dataset

The dataset which was used consists of X-ray images categorized into three distinct classes. These categories are essential for training, validating, and testing the machine learning models to detect pneumonia.

Categories

1. **No Pneumonia**: X-ray images that do not show any signs of pneumonia. These images represent healthy lung conditions.
2. **Viral Pneumonia**: X-ray images that show signs of pneumonia caused by a viral infection. This category helps the model learn the specific patterns associated with viral infections in the lungs.
3. **Bacterial Pneumonia**: X-ray images that show signs of pneumonia caused by a bacterial infection. This category helps the model differentiate between bacterial and viral pneumonia, as well as from healthy lungs.

Dataset Composition



- The dataset contains a substantial number of X-ray images for each category to ensure the model can learn effectively.
- **No Pneumonia**: These images serve as the baseline and help the model identify what a healthy lung looks like.
- **Viral Pneumonia**: These images teach the model to recognize the distinct patterns associated with viral pneumonia, which might differ from bacterial pneumonia.
- **Bacterial Pneumonia**: These images teach the model to recognize the distinct patterns associated with bacterial pneumonia, helping in making accurate distinctions.

Train size	Validation size	Test size
4684	586	585

Dataset Usage

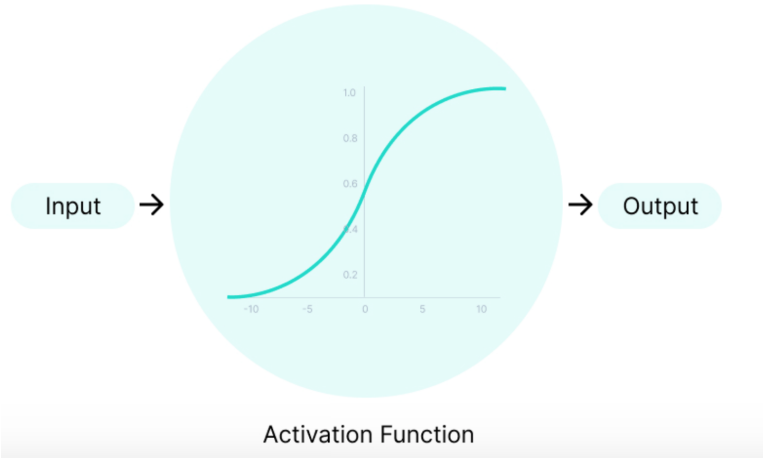
- **Training**: A significant portion of the dataset is used to train the machine learning models. This involves teaching the model to identify patterns and features that distinguish between the three categories.
- **Validation**: A separate portion of the dataset is used for validation. This helps in fine-tuning the model and adjusting hyperparameters to improve accuracy and generalization.
- **Testing**: The remaining portion of the dataset is used for testing the model's performance. This final evaluation helps in understanding how well the model performs on unseen data.

B. Activation functions

Activation Function helps the neural network to use important information while suppressing irrelevant data points

Why do Neural Networks Need an Activation Function ?

Well, the purpose of an activation function is to add non-linearity to the neural network.



Activation Function

Activation functions introduce an additional step at each layer during the forward propagation, but its computation is worth it. Here is why—

Let's suppose we have a neural network working *without* the activation functions.

In that case, every neuron will only be performing a linear transformation on the inputs using the weights and biases. It's because it doesn't matter how many hidden layers we attach in the neural network; all layers will behave in the same way because the composition of two linear functions is a linear function itself.

Although the neural network becomes simpler, learning any complex task is impossible, and our model would be just a linear regression model.

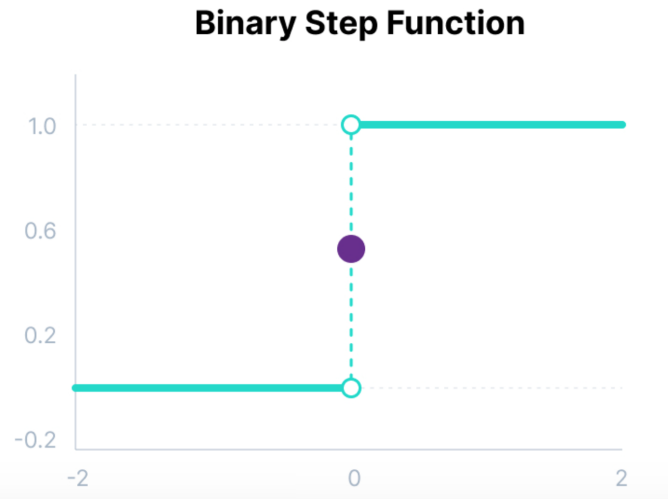
1) Types of Neural Networks Activation Functions

Now, as we've covered the essential concepts, let's go over the most popular neural networks activation functions.

Binary Step Function

Binary step function depends on a threshold value that decides whether a neuron should be activated or not.

The input fed to the activation function is compared to a certain threshold; if the input is greater than it, then the neuron is activated, else it is deactivated, meaning that its output is not passed on to the next hidden layer.



Mathematically it can be represented as:

$$f(x) = 1, x >= 0 = 0, x < 0$$

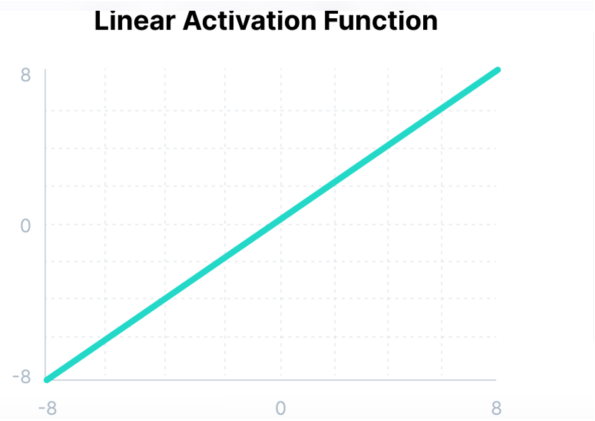
Here are some of the limitations of binary step function:

- It cannot provide multi-value outputs—for example, it cannot be used for multi-class classification problems.
- The gradient of the step function is zero, which causes a hindrance in the backpropagation process.

Linear Activation Function

The linear activation function, also known as "no activation," or "identity function" (multiplied x1.0), is where the activation is proportional to the input.

The function doesn't do anything to the weighted sum of the input, it simply spits out the value it was given.



$$Linear$$

$$f(x) = x$$

However, a linear activation function has two major problems :

- It's not possible to use backpropagation as the derivative of the function is a constant and has no relation to the input x.
- All layers of the neural network will collapse into one if a linear activation function is used. No matter the number of layers in the neural network, the last layer will still be a linear function of the first layer. So, essentially, a linear activation function turns the neural network into just one layer.

Non-Linear Activation Functions

The linear activation function shown above is simply a linear regression model.

Because of its limited power, this does not allow the model to create complex mappings between the network's inputs and outputs.

Non-linear activation functions solve the following limitations of linear activation functions:

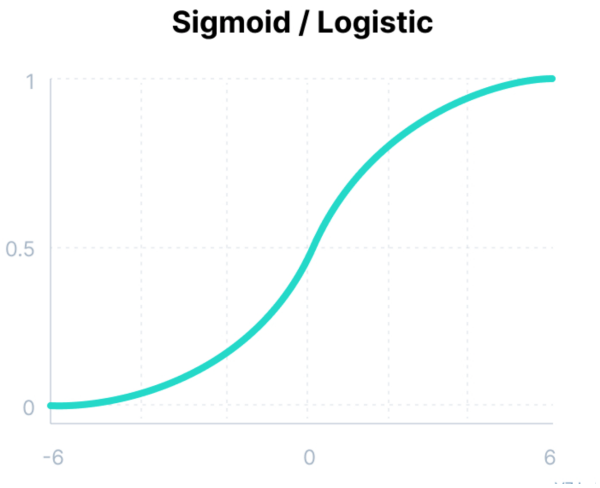
- They allow backpropagation because now the derivative function would be related to the input, and it's possible to go back and understand which weights in the input neurons can provide a better prediction.
- They allow the stacking of multiple layers of neurons as the output would now be a non-linear combination of input passed through multiple layers. Any

output can be represented as a functional computation in a neural network.

Now, let's have a look at ten different non-linear neural networks activation functions and their characteristics.

Sigmoid / Logistic Activation Function

$$z_j = b_{j0} + b_{j1}x_1 + b_{j2}x_2 + ... + b_{jn}x_n$$



$$Sigmoid\ /\ Logistic$$

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

This function takes any real value as input and outputs values in the range of 0 to 1.

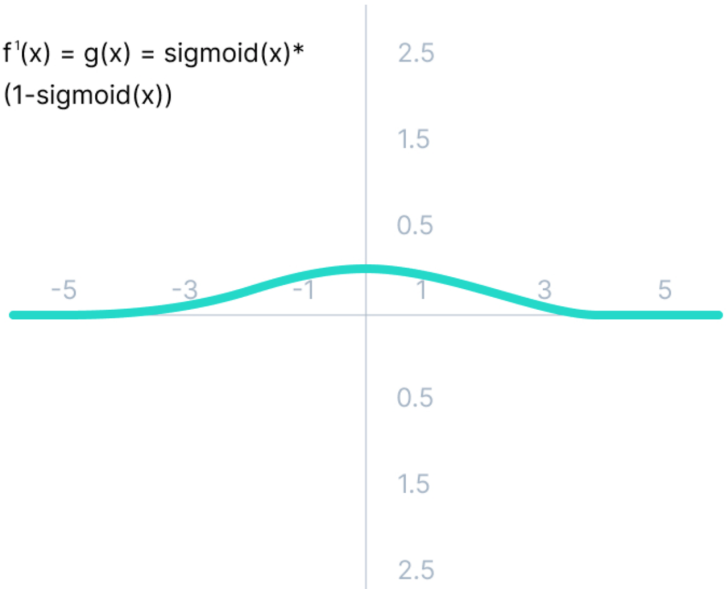
The larger the input (more positive), the closer the output value will be to 1.0, whereas the smaller the input (more negative), the closer the output will be to 0.0, as shown below.Mathematically it can be represented as:

Here's why sigmoid/logistic activation function is one of the most widely used functions:

- It is commonly used for models where we have to predict the probability as an output. Since probability of anything exists only between the range of 0 and 1, sigmoid is the right choice because of its range.
- The function is differentiable and provides a smooth gradient, i.e., preventing jumps in output values. This is represented by an S-shape of the sigmoid activation function.

The limitations of sigmoid function are discussed below:

- The derivative of the function is f'(x) = sigmoid(x)*(1-sigmoid(x)).



The derivative of the Sigmoid Activation Function

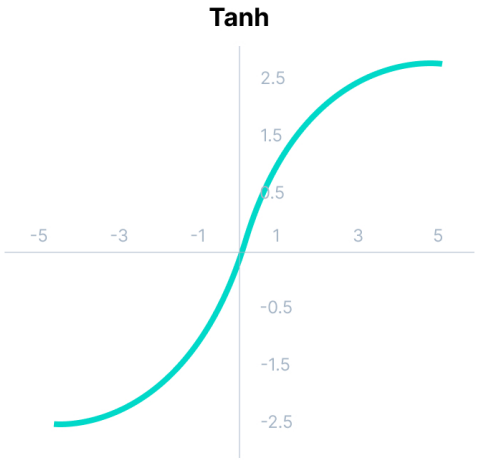
As we can see from the above Figure, the gradient values are only significant for range -3 to 3, and the graph gets much flatter in other regions.

It implies that for values greater than 3 or less than -3, the function will have very small gradients. As the gradient value approaches zero, the network ceases to learn and suffers from the *Vanishing gradient* problem.

- The output of the logistic function is not symmetric around zero. So the output of all the neurons will be of the same sign. This makes the training of the neural network more difficult and unstable.

Tanh Function (Hyperbolic Tangent)

Tanh function is very similar to the sigmoid/logistic activation function, and even has the same S-shape with the difference in output range of -1 to 1. In Tanh, the larger the input (more positive), the closer the output value will be to 1.0, whereas the smaller the input (more negative), the closer the output will be to -1.0.



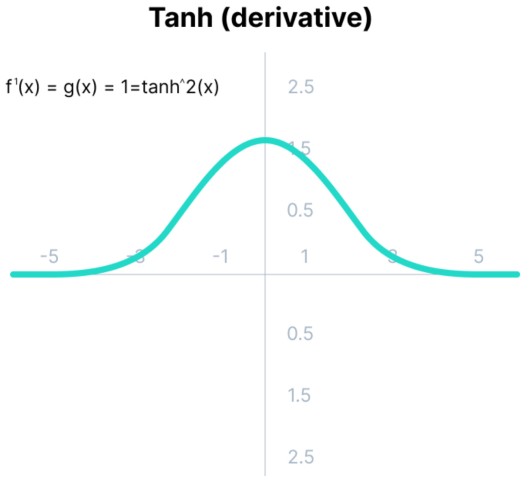
$$Tanh$$

$$f(x) = \frac{\left(e^{x} - e^{-x}\right)}{\left(e^{x} + e^{-x}\right)}$$

Advantages of using this activation function are:

- The output of the tanh activation function is Zero centered; hence we can easily map the output values as strongly negative, neutral, or strongly positive.
- Usually used in hidden layers of a neural network as its values lie between -1 to; therefore, the mean for the hidden layer comes out to be 0 or very close to it. It helps in centering the data and makes learning for the next layer much easier.

Have a look at the gradient of the tanh activation function to understand its limitations.



Gradient of the Tanh Activation Function

As you can see— it also faces the problem of vanishing gradients similar to the sigmoid activation function. Plus the gradient of the tanh function is much steeper as compared to the sigmoid function.

💡 **Note: Although both sigmoid and tanh face vanishing gradient issue, tanh is zero centered, and the gradients are not restricted to move in a certain direction. Therefore, in practice, tanh nonlinearity is always preferred to sigmoid nonlinearity.**

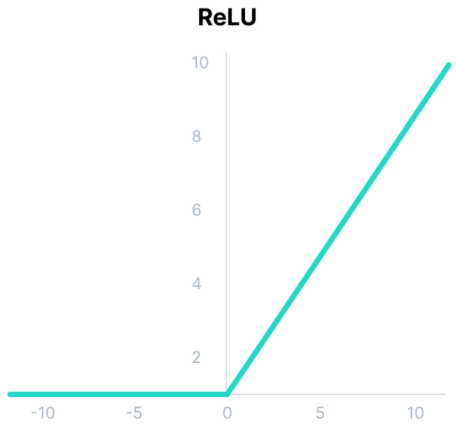
ReLU Function

ReLU stands for Rectified Linear Unit.

Although it gives an impression of a linear function, ReLU has a derivative function and allows for backpropagation while simultaneously making it computationally efficient.

The main catch here is that the ReLU function does not activate all the neurons at the same time.

The neurons will only be deactivated if the output of the linear transformation is less than 0.



$$ReLU$$

$$f(x) = max\ (0, x)$$

ReLU Activation Function

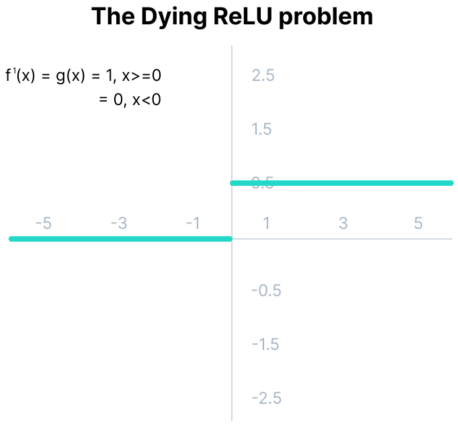
Mathematically it can be represented as:

The advantages of using ReLU as an activation function are as follows:

- Since only a certain number of neurons are activated, the ReLU function is far more computationally efficient when compared to the sigmoid and tanh functions.
- ReLU accelerates the convergence of gradient descent towards the global minimum of the loss function due to its linear, non-saturating property.

The limitations faced by this function are:

- The Dying ReLU problem, which I explained below.



The Dying ReLU problem

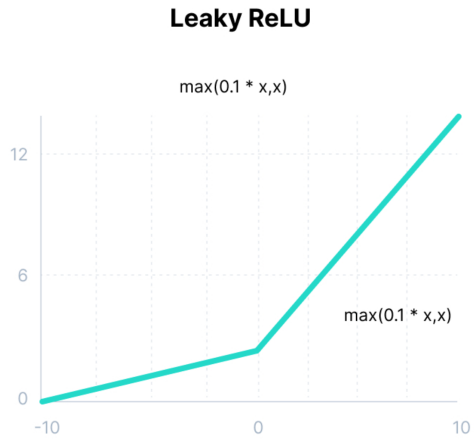
The negative side of the graph makes the gradient value zero. Due to this reason, during the backpropagation process, the weights and biases for some neurons are not updated. This can create dead neurons which never get activated.

- All the negative input values become zero immediately, which decreases the model's ability to fit or train from the data properly.

Note: For building the most reliable ML models, split your data into train, validation, and test sets.

Leaky ReLU Function

Leaky ReLU is an improved version of ReLU function to solve the Dying ReLU problem as it has a small positive slope in the negative area.



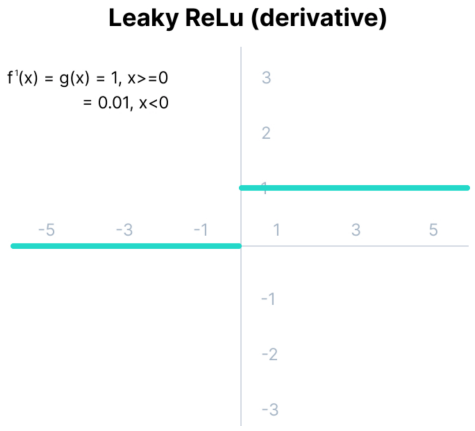
Leaky ReLU

Mathematically it can be represented as:

The advantages of Leaky ReLU are same as that of ReLU, in addition to the fact that it does enable backpropagation, even for negative input values.

By making this minor modification for negative input values, the gradient of the left side of the graph comes out to be a non-zero value. Therefore, we would no longer encounter dead neurons in that region.

Here is the derivative of the Leaky ReLU function.



The derivative of the Leaky ReLU function

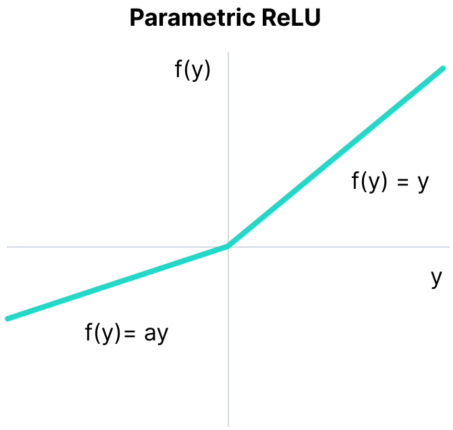
The limitations that this function faces include:

- The predictions may not be consistent for negative input values.
- The gradient for negative values is a small value that makes the learning of model parameters time-consuming.

Parametric ReLU Function

Parametric ReLU is another variant of ReLU that aims to solve the problem of gradient's becoming zero for the left half of the axis.

This function provides the slope of the negative part of the function as an argument *a*. By performing backpropagation, the most appropriate value of *a* is learnt.



$$Parametric\ ReLU$$

$$f(x) = max\ (ax, x)$$

Parametric ReLU

Mathematically it can be represented as:

Where *"a"* is the slope parameter for negative values.

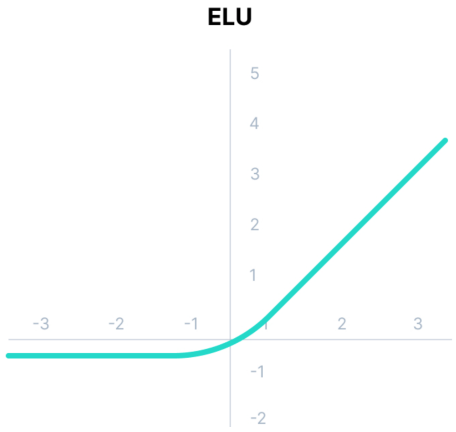
The parameterized ReLU function is used when the leaky ReLU function still fails at solving the problem of dead neurons, and the relevant information is not successfully passed to the next layer.

This function's limitation is that it may perform differently for different problems depending upon the value of slope parameter ***a*.**

Exponential Linear Units (ELUs) Function

Exponential Linear Unit, or ELU for short, is also a variant of ReLU that modifies the slope of the negative part of the function.

ELU uses a log curve to define the negativ values unlike the leaky ReLU and Parametric ReLU functions with a straight line.



$$ELU$$

$$\begin{cases} x & for\ x \geqslant 0 \\ \alpha\left(e^{x}-1\right) & for\ x < 0 \end{cases}$$

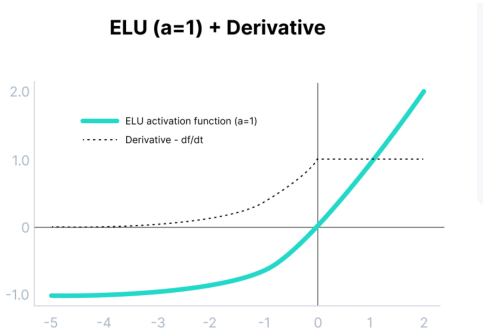
ELU Activation Function

ELU is a strong alternative for f ReLU because of the following advantages:

- ELU becomes smooth slowly until its output equal to -α whereas RELU sharply smoothes.
- Avoids dead ReLU problem by introducing log curve for negative values of input. It helps the network nudge weights and biases in the right direction.

The limitations of the ELU function are as follow:

- It increases the computational time because of the exponential operation included
- No learning of the ‘a’ value takes place
- Exploding gradient problem



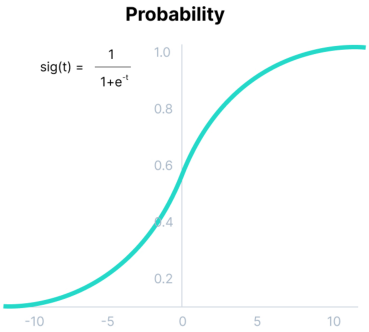
$$ELU\ derivative$$

$$f'(x) = \begin{cases} 1 & for\ x \geqslant 0 \\ f(x) + \alpha & for\ x < 0 \end{cases}$$

ELU Activation Function and its derivative

Softmax Function

Before exploring the ins and outs of the Softmax activation function, we should focus on its building block—the sigmoid/logistic activation function that works on calculating probability values.



Probability

The output of the sigmoid function was in the range of 0 to 1, which can be thought of as probability.

But—

This function faces certain problems.

Let’s suppose we have five output values of 0.8, 0.9, 0.7, 0.8, and 0.6, respectively. How can we move forward with it?

The answer is: We can’t.

The above values don’t make sense as the sum of all the classes/output probabilities should be equal to 1.

You see, the Softmax function is described as a combination of multiple sigmoids.

It calculates the relative probabilities. Similar to the sigmoid/logistic activation function, the SoftMax function returns the probability of each class.

It is most commonly used as an activation function for the last layer of the neural network in the case of multi-class classification.

Mathematically it can be represented as:

Softmax

$$\text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

V7 Labs

Softmax Function

Let’s go over a simple example together.

Assume that you have three classes, meaning that there would be three neurons in the output layer. Now, suppose that your output from the neurons is [1.8, 0.9, 0.68].

Applying the softmax function over these values to give a probabilistic view will result in the following outcome: [0.58, 0.23, 0.19].

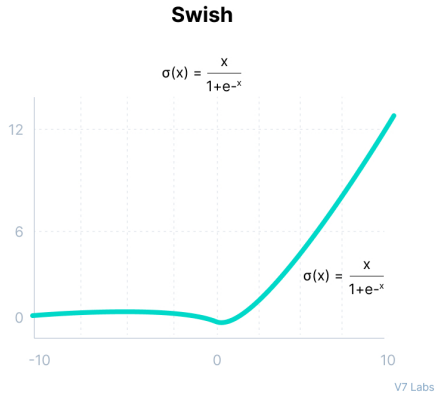
The function returns 1 for the largest probability index while it returns 0 for the other two array indexes. Here, giving full weight to index 0 and no weight to index 1 and index 2. So the output would be the class corresponding to the 1st neuron(index 0) out of three.

You can see now how softmax activation function make things easy for multi-class classification problems.

Swish

It is a self-gated activation function developed by researchers at Google.

Swish consistently matches or outperforms ReLU activation function on deep networks applied to various challenging domains such as image classification, machine translation etc.



$$Swish$$

$$f(x) = x*sigmoid(x)$$

Swish Activation Function

This function is bounded below but unbounded above i.e. *Y* approaches to a constant value as *X* approaches negative infinity but *Y* approaches to infinity as *X* approaches infinity.

Mathematically it can be represented as:

Here are a few advantages of the Swish activation function over ReLU:

- Swish is a smooth function that means that it does not abruptly change direction like ReLU does near x = 0. Rather, it smoothly bends from 0 towards values < 0 and then upwards again.
- Small negative values were zeroed out in ReLU activation function. However, those negative values may still be relevant for capturing patterns underlying the data. Large negative values are zeroed out for reasons of sparsity making it a win-win situation.
- The swish function being non-monotonous enhances the expression of input data and weight to be learnt.

Gaussian Error Linear Unit (GELU)

The Gaussian Error Linear Unit (GELU) activation function is compatible with BERT, ROBERTa, ALBERT, and other top NLP models. This activation function is motivated by combining properties from dropout, zoneout, and ReLUs.

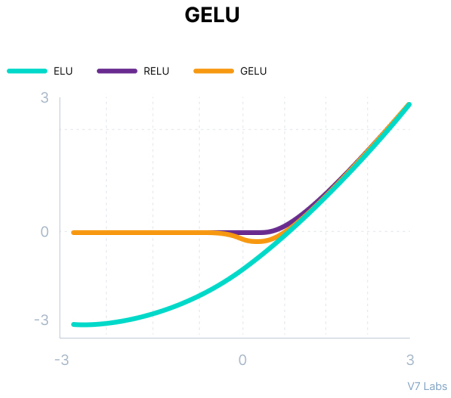
ReLU and dropout together yield a neuron's output. ReLU does it deterministically by multiplying the input by zero or one (depending upon the input value being positive or negative) and dropout stochastically multiplying by zero.

RNN regularizer called zoneout stochastically multiplies inputs by one.

We merge this functionality by multiplying the input by either zero or one which is stochastically determined and is dependent upon the input. We multiply the neuron input x by

m ~ Bernoulli(Φ(x)), where Φ(x) = P(X ≤x), X ~ N (0, 1) is the cumulative distribution function of the standard normal distribution.

This distribution is chosen since neuron inputs tend to follow a normal distribution, especially with Batch Normalization.



$$GELU$$

$$f(x) = xP(X \leq x) = x\Phi(x)$$
$$= 0.5x\left(1 + tanh\left[\sqrt{2/\pi}\left(x + 0.044715x^3\right)\right]\right)$$

Gaussian Error Linear Unit (GELU) Activation Function

Mathematically it can be represented as:

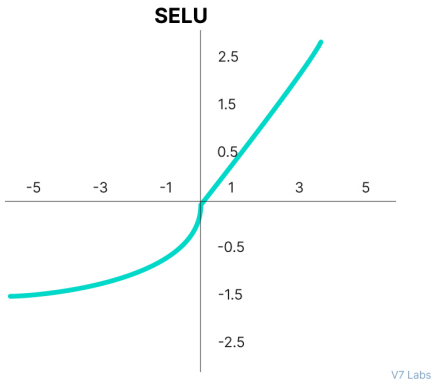
GELU nonlinearity is better than ReLU and ELU activations and finds performance improvements across all tasks in domains of computer vision, natural language processing, and speech recognition.

Scaled Exponential Linear Unit (SELU)

SELU was defined in self-normalizing networks and takes care of internal normalization which means each layer preserves the mean and variance from the previous layers. SELU enables this normalization by adjusting the mean and variance.

SELU has both positive and negative values to shift the mean, which was impossible for ReLU activation function as it cannot output negative values.

Gradients can be used to adjust the variance. The activation function needs a region with a gradient larger than one to increase it.



$$SELU$$

$$f(\alpha, x) = \lambda \begin{cases} \alpha(e^x - 1) & for\ x < 0 \\ x & for\ x \geqslant 0 \end{cases}$$

SELU Activation Function

SELU has values of alpha α and lambda λ predefined.

Here's the main advantage of SELU over ReLU:

- Internal normalization is faster than external normalization, which means the network converges faster.

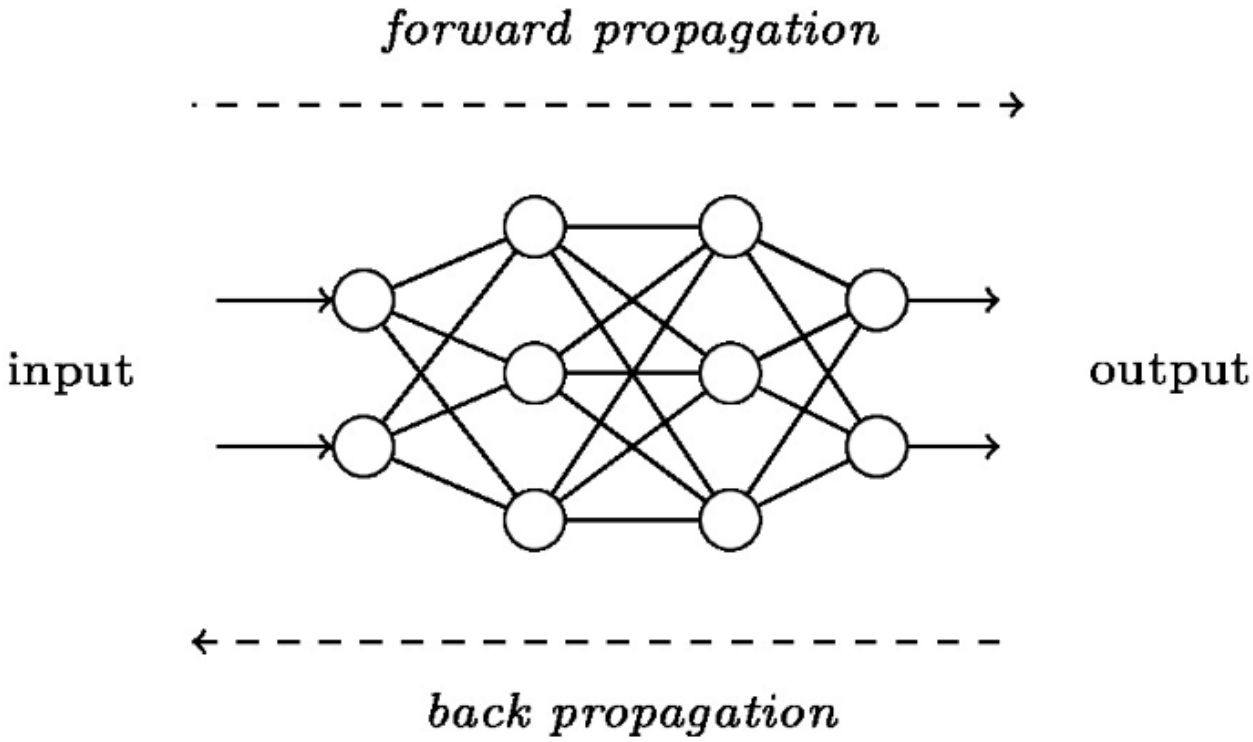
SELU is a relatively newer activation function and needs more papers on architectures such as CNNs and RNNs, where it is comparatively explored.

2) Forward and Backward Propagation

Forward propagation, often referred to as forward pass, is the process of computing the output of a neural network by moving from the input layer to the output layer, passing through all the hidden layers. It is a fundamental operation in training and evaluating neural networks.

During forward propagation, the input data is fed into the network, and each layer performs a series of transformations on the data. These transformations are typically linear operations followed by non-linear activations. For example, in a fully connected layer, the linear operation involves computing the dot product of the input vector and the weight matrix, then adding a bias vector. The result is then passed through an activation function (such as ReLU, sigmoid, or tanh), which introduces non-linearity to the model, enabling it to learn more complex patterns. This process is repeated layer by layer until the data reaches the output layer, where the final predictions are made. In a classification task, for instance, the output layer might use a softmax activation to produce a probability distribution over the classes.

Forward propagation is crucial not only for making predictions with a trained model but also for calculating the loss during training. The loss, which quantifies the difference between the predicted output and the true target, is then used in backpropagation to update the model's parameters. Thus, forward propagation is integral to both the inference and training phases of neural network operations.



Backpropagation, short for "backward propagation of errors," is a crucial algorithm for training neural networks. It involves propagating the error from the output layer back through the network to update the weights and biases, thereby minimizing the loss function.

After forward propagation, the network produces an output which is compared to the true target to compute the loss, using a chosen loss function (e.g., mean squared error for regression or cross-entropy for classification). Back-propagation then begins with this loss value. The goal is to determine how much each weight in the network contributed to the error. This is achieved by calculating the gradient of the loss function with respect to each weight, using the chain rule of calculus. The process starts at the output layer and moves backwards through each hidden layer to the input layer.

At each layer, the gradients of the weights and biases are computed. These gradients represent the partial derivatives of the loss function with respect to the weights and biases. By applying the chain rule, the gradient of the loss with respect to the weights in a given layer is the product of the gradient of the loss with respect to the output of that layer and the gradient of the output of that layer

with respect to the weights. This process is repeated layer by layer, ensuring that every weight in the network is adjusted in proportion to its contribution to the total error.

Once the gradients are calculated, the weights and biases are updated using an optimization algorithm, commonly stochastic gradient descent (SGD) or its variants (such as Adam). These updates are made by subtracting a fraction of the gradient (scaled by the learning rate) from the current weights, effectively moving the weights in the direction that reduces the loss.

Back-propagation is thus essential for learning in neural networks. It enables the network to iteratively reduce the error by fine-tuning the weights and biases, improving its performance on the given task.

3) Optimization techniques

When we train our models, we need to make sure of several things: we don't want it to overfit. We also don't want it to take too long to train, nor do we want it to plateau with a poor performance. This is where optimization techniques come in handy. By applying some of these techniques, we can speed-up our training, but we can also avoid overfitting and end-up with a model which performs very well.

Learning rate scheduler

A learning rate scheduler is a technique used in training neural networks to adjust the learning rate over time. The learning rate determines the size of the steps the optimization algorithm takes when updating the model's weights. A well-chosen learning rate can significantly improve the training process, helping the model converge faster and reach a better performance.

During training, the learning rate can impact how quickly and effectively the model learns:

- **Too high**: The model may oscillate around the minimum loss and fail to converge.
- **Too low**: The model may converge very slowly and get stuck in local minima.

- **Dynamic adjustment**: Adjusting the learning rate during training can help the model converge more quickly and avoid local minima.
-

Normalization

Normalization in artificial intelligence (AI), particularly in machine learning, is an optimization technique that involves rescaling the input features of a model so that they fall within a common range, typically between 0 and 1 or centered around a mean of 0 with a standard deviation of 1. This technique is crucial for ensuring faster and more stable convergence of learning algorithms, especially those utilizing gradient descent, such as neural networks.

Why is Normalization Important?

1. **Faster Convergence**: Learning algorithms like gradient descent converge faster when data is normalized. Non-normalized features can cause significant oscillations, slowing down the optimization process.
2. **Feature Scale**: When features have different scales, the weight updates can be disproportionate, destabilizing the model and slowing learning.
3. **Increased Accuracy**: Normalization can improve model accuracy by ensuring that each feature contributes equally to the learning process.

Common Normalization Techniques

There are several commonly used normalization techniques in machine learning:

1. **Min-Max Scaling**:
    - **Description**: This technique transforms the data so that it lies within a specific range, typically between 0 and 1. Each value is rescaled relative to the minimum and maximum values of the feature.
2. **Standardization (Z-score Normalization)**:
    - **Description**: This technique transforms data so that it has a mean of 0 and a standard deviation of 1. It is useful when features have different distributions but follow a normal distribution.
3. **Robust Scaler**:

- **Description**: This method uses the median and the interquartile range (IQR) to rescale features. It is less sensitive to outliers compared to standardization.

Impact of Normalization

- **Distance-based Algorithms**: Normalization is especially important for distance-based algorithms, such as k-NN (k-nearest neighbors) and clustering, where differences in scale can lead to biased results.
- **Neural Networks**: In neural networks, normalizing inputs can speed up training convergence and improve overall model performance.
- **Support Vector Machines (SVM)**: SVMs are also sensitive to feature scales, and normalization can enhance prediction accuracy.

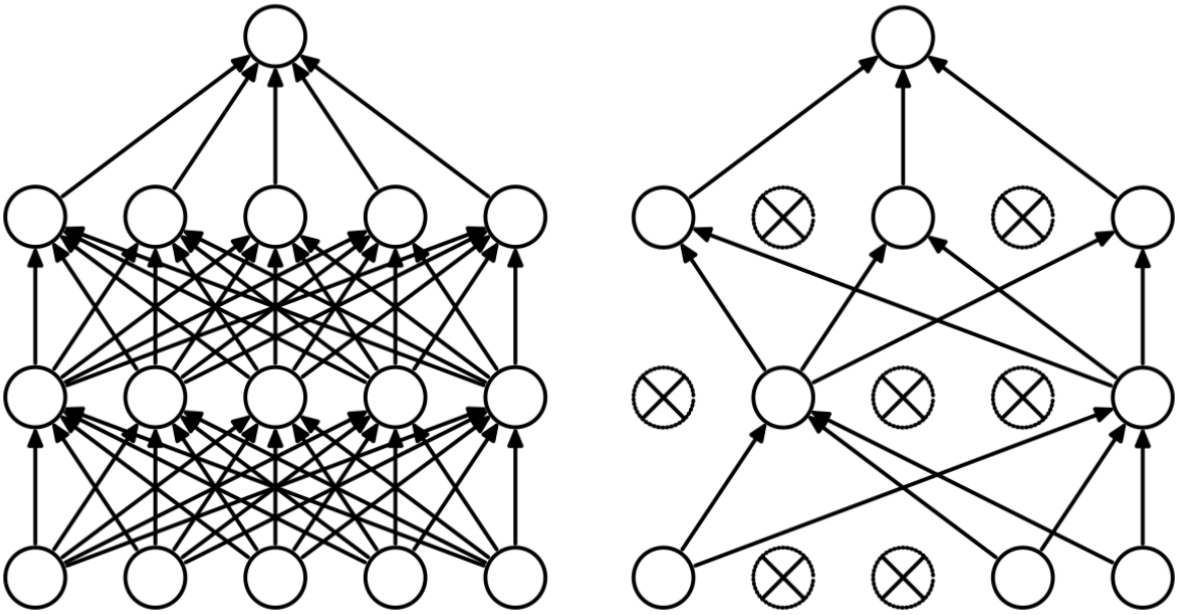
Dropout

Dropout is an optimization technique aimed at reducing the issue of overfitting. To prevent the model from relying too much on only a few of its neurons, we randomly deactivate neurons each time the model runs.

Before applying dropout to the model **After applying dropout to the model**

$$Leaky\ ReLU$$

$$f(x) = max\ (0.1x, x)$$



L2 Regularization and weight decay

L2 regularization is a technique which is used to prevent our model from over-fitting. It is achieved by adding a value to our loss, which is based on the sum of weights squared:

$$L2penalty = \lambda * \sum(w_i^2)$$

`λ`: the loss

`w`: the weights

By doing this, the model will get punished from having weights which are too big.

Weight decay is basically an implementation of L2 Regularization, but applied to neural networks.

Gradient clipping

Gradient clipping is also a technique used to prevent our model from over-fitting. It basically sets a maximum value for the gradients during the back-propagation.

III - Models

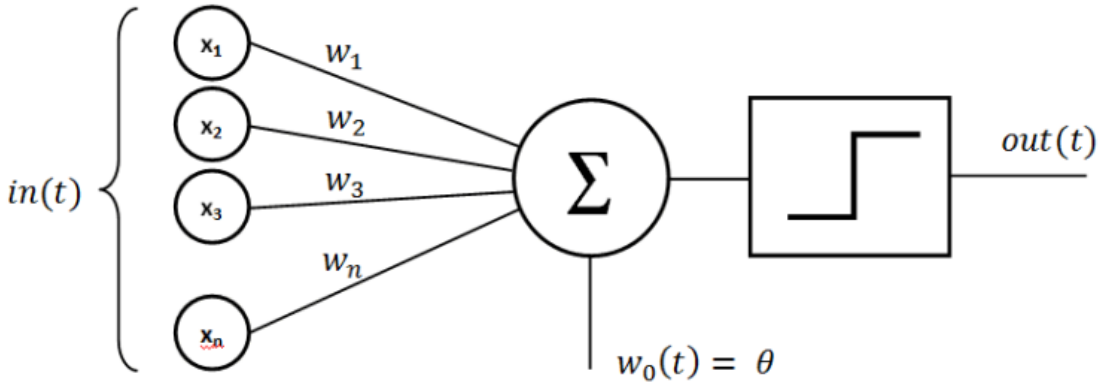
A. Rosenblatt Perceptron

In 1957, Frank Rosenblatt invented the Perceptron, the simplest form of neural network and one of the first models to demonstrate machine learning. This groundbreaking model laid the foundation for future advancements in the field, as it was one of the earliest systems capable of learning from data. A model that would later be applied to fields such as image processing and speech recognition, it showcased the potential of machine learning to tackle complex tasks by improving performance through experience. Based on a simple algorithm, the Perceptron could adapt to the input data in order to perform better over time. However, despite its innovative approach, the Perceptron had hard limitations. It could not solve non-linear problems, meaning it struggled with tasks where the relationship between input and output was not straightforward. This limitation was famously highlighted by Marvin Minsky and Seymour Papert in their 1969 book ”Perceptrons”, which demonstrated that the Perceptron could not handle the XOR problem, a simple example of a non-linear classification task.

1) How does a Perceptron Learn?

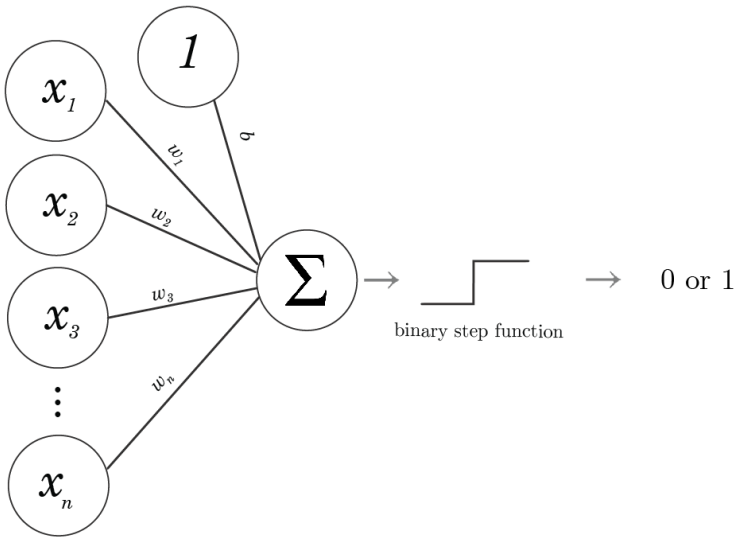
The Perceptron involves principles that are foundational to modern artificial intelligence. It comprises key components such as Inputs and Weights, an Activation Function, and Back-propagation. At its core, the Perceptron is a single neuron directly connected to the inputs. Each input is assigned a specific weight, which represents its importance in the prediction process. To make a prediction, the Perceptron calculates the weighted sum of all inputs, which is the sum of each input multiplied by its corresponding weight. Once the weighted sum is computed, the Perceptron employs an activation function to determine the output. In this simplified model, we use a threshold as the activation function. If the weighted sum exceeds the set threshold, the Perceptron produces a positive result; otherwise, it

returns a negative result. This binary classification process allows the Perceptron to distinguish between two classes based on the input data.



Perceptron

To improve its performance over time, the Perceptron uses a learning process called Back-propagation. During training, the Perceptron adjusts its weights based on the errors between the predicted and actual outcomes. By iteratively updating the weights to minimize these errors, it learns to make more accurate predictions. In some rare cases we may encounter a set of inputs that sum to 0, with such input, no matter the weights the result will always be 0. To avoid this issue we use a concept called bias. It is a complementary input that we add with a fixed value of 1, so that no matter the input, the result can always be different than 0.



Perceptron with bias

2) Classifying Images using the Perceptron

Using this model, we can classify images. However, due to the Boolean nature of the Perceptron's output, it can only differentiate between two classes, which is enough in our case. Before we proceed, we need to normalize our data. Each image will be resized to a fixed number of pixels—64×64 in this case—which provides a good balance between detail and low memory usage. Then, the color ranges will be converted to gray-scale values, leaving us with a single number per pixel. Each of these gray-scale values will serve as an input to the Perceptron.

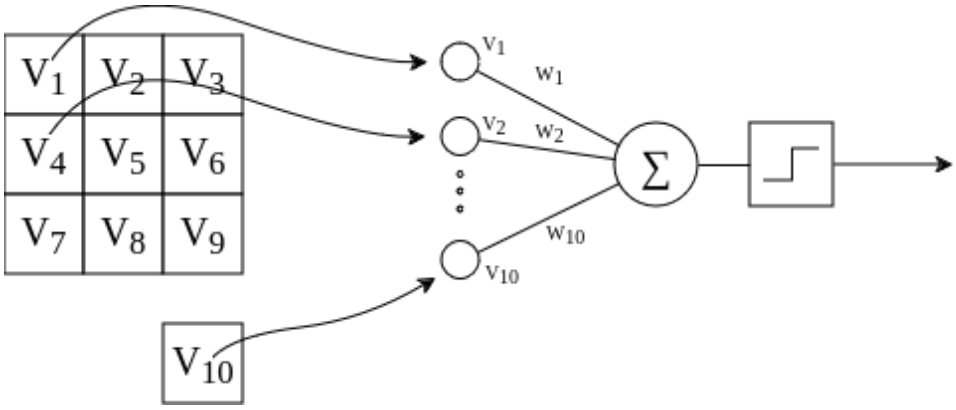
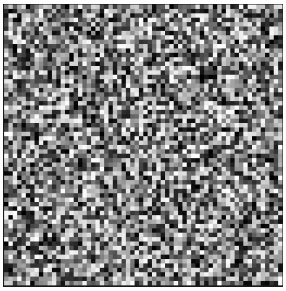
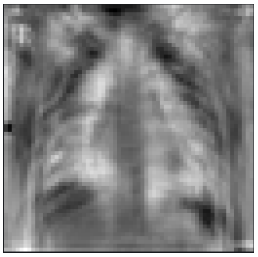


Image Classification Perceptron

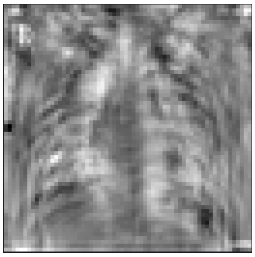
The final step before starting the training process is to define the hyper parameters, which are the human-controlled settings of the model. For a Perceptron, these include the learning rate, the activation function, the number of epochs, and the number of inputs. We will use a simple threshold for the activation function, and the number of inputs has already been established during data normalization. Therefore, we only need to define the learning rate and the number of epochs. The learning rate is crucial during the Perceptron's learning phase; after predicting a value and calculating the error, the error is multiplied by the learning rate and used to adjust the weights accordingly. The learning rate needs to be small enough to allow the back-propagation process to find precise values for the Perceptron's weights. Finally, the number of epochs define how many times the training process will go through the entire dataset. Perceptrons cannot be over-fit and are very quick to train due to its simple structure, so setting a high number of epochs should not disturb the training process. Due to the simple structure of the Perceptron, visualizing its weights is quite simple. By assigning one weight per pixel, we can generate an image that mirrors the properties of our normalized training data, providing a glimpse into the Perceptron's "brain." Additionally, when predicting an image, we can combine each pixel's value with the corresponding weight from the Perceptron and normalize the result. This allows us to see what the Perceptron focuses on during image prediction.



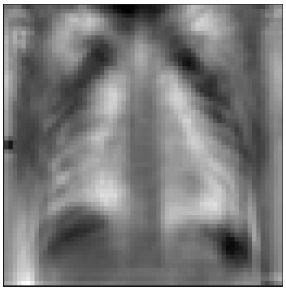
Weights Epoch 0



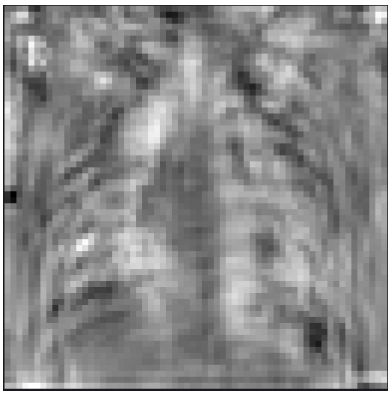
Weights Epoch 4



Weights Epoch 16



Weights Epoch 1

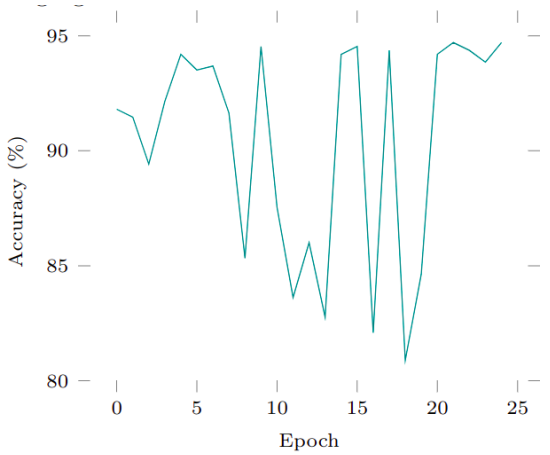


Weights After Training

Image Predicted

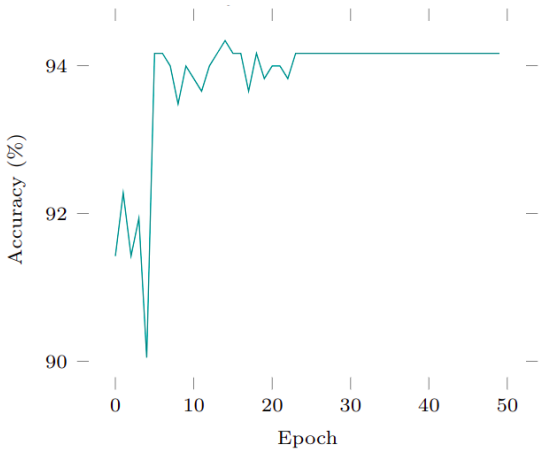
Weights + Image

When analyzing the accuracy trend over multiple epochs in training a Perceptron, a significant fluctuation often indicates that the model is struggling to find an optimal solution. This issue is frequently caused by an excessively high learning rate, which results in back-propagation overshooting while trying to adjust the weights based on the error. Essentially, the adjustments made during training are too large, causing the model to oscillate around the optimal point rather than converging to it.



There are several types of learning rate schedulers, but in this case, three specific types were tested: Step Decay, Exponential Decay, and Cosine Annealing. Step Decay reduces the learning rate by a factor at specified intervals. Exponential Decay continuously decreases the learning rate exponentially. Cosine Annealing, on the other hand, adjusts the learning rate following a cosine function. In practice, Step Decay has shown the most positive impact on accuracy among the tested methods. This approach allows the learning rate to be lowered in a controlled, step-wise manner, which seems to aid the model in converging more effectively by balancing between fast initial learning and stable final adjustments.

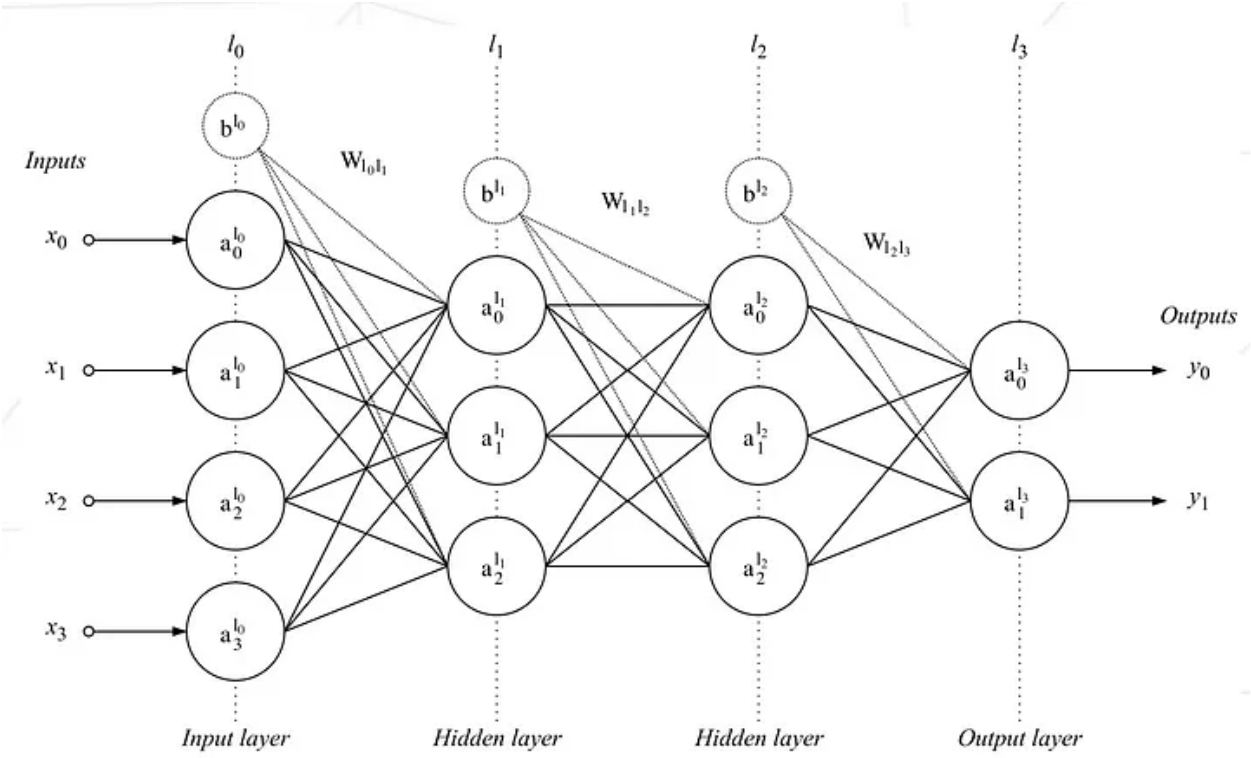
To mitigate this problem, implementing a learning rate scheduler can be beneficial. A learning rate scheduler dynamically adjusts the learning rate during training, typically decreasing it as the training progresses to allow for finer adjustments.



B. Multi-Layer Perceptron

An MLP (Multi-Layer Perceptron) is a class of feed-forward artificial neural networks. It consists of at least three layers of nodes: an input layer, one or more hidden layers, and an output layer. Each node (except for the input nodes) is a neuron that uses a nonlinear activation function. The input layer is responsible for receiving the input data and transferring it to the next layer. The hidden layers transforms this input into something that the output layer can use, and then the output layer produces the final output.

MLPs also introduces two key features, one is the Back-propagation which traverses the neural netowrok backwards after a prediction and adjusts the weights. And the other is non-linearity, by using non-linear activation functions such as ReLU (Rectified Linear Unit), sigmoid, or tanh.



The XOR Problem

The XOR (exclusive OR) problem is a classic problem in the field of neural networks and machine learning. The XOR function outputs true or 1 only when the inputs differ (one is true, the other is false). The truth table for XOR is:

Input 1	Input 2	XOR Output
0	0	0
0	1	1
1	0	1
1	1	0

As mentioned previously, a single-layer perceptron can only solve linearly separable problems. The XOR function is not linearly separable, meaning no single line can separate the true outputs from the false outputs on a two-dimensional plane. An MLP can solve the XOR problem by introducing one or more hidden layers, which allow the network to create a non-linear decision boundary.

C. Logistic regression

Logistic regression is a machine learning technique used to make predictions. It's often used for binary classification, where the outcome is one of two possibilities. However, it can also be used for multiclass classification, where the outcome can be one of multiple possibilities. In our case, we want to predict if a patient has no pneumonia, bacterial pneumonia, or viral pneumonia.

Why "Logistic"?

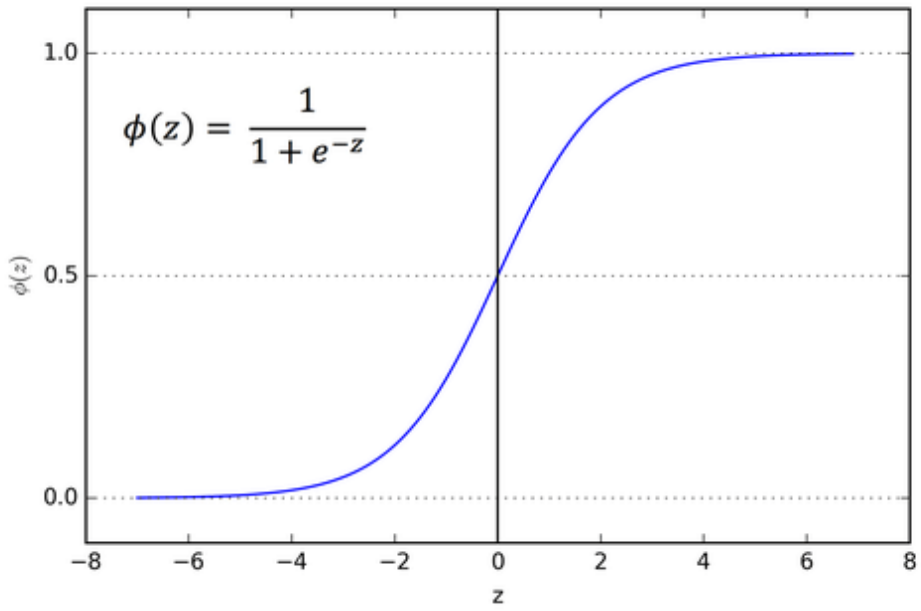
The term "logistic" comes from the logistic function, also known as the sigmoid function. This function helps transform any input value into a probability between 0 and 1.

The Sigmoid Function

For binary classification, the sigmoid function is defined as:

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

Here's a graph of the sigmoid function:



Where:

- $\sigma(z)$ is the output of the sigmoid function (a probability between 0 and 1).
- e is the mathematical constant equal to ~2.718.
- z is the input to the function, often a linear combination of the independent variables (features).

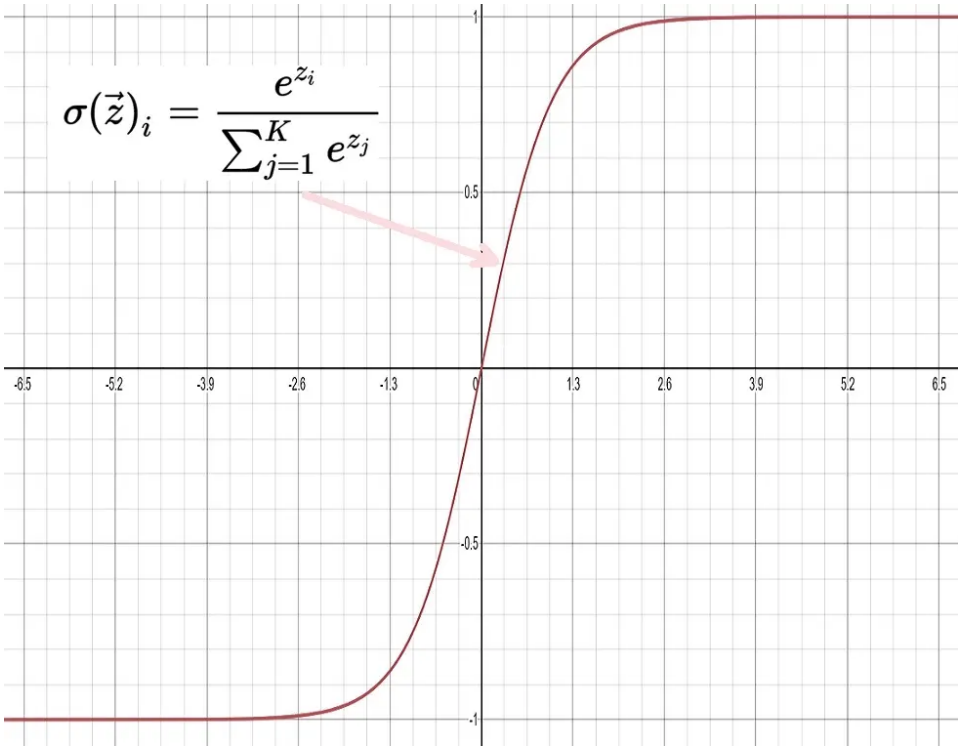
Multiclass Logistic Regression

When we have more than two possible outcomes, we use softmax regression instead of the sigmoid function. This is how multiclass logistic regression works.

The softmax function is defined as:

$$P(y = j|x) = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

Here's a visual representation of how softmax works:



Where:

- $P(y = j|x)$ is the probability that the input x belongs to class j.
- z_j is the score calculated for class j.
- K is the total number of classes.
- e is the mathematical constant equal to ~2.718.

How Multiclass Logistic Regression Works

1. **Independent Variables (Features):** These are the input data. For predicting pneumonia type, features could include symptoms, lab results, patient age, etc.
2. **Linear Combination:** For each class (no pneumonia, bacterial pneumonia, viral pneumonia), calculate a linear combination of the features:

   Where:

   - z_j is the linear combination for class j.
   - $b_{j0}, b_{j1}, ..., b_{jn}$ are the coefficients for class j.
   - $x_1, x_2, ..., x_n$ are the features.
3. **Apply the Softmax Function:** Compute the probabilities for each class using the softmax function:

$$P(y = j|x) = \frac{e^{z_j}}{\sum_{k=1}^{3} e^{z_k}}$$

   This gives the probability that the patient belongs to each class (no pneumonia, bacterial pneumonia, viral pneumonia).

Interpretation of the Output

The softmax function outputs probabilities that add up to 1. For example:

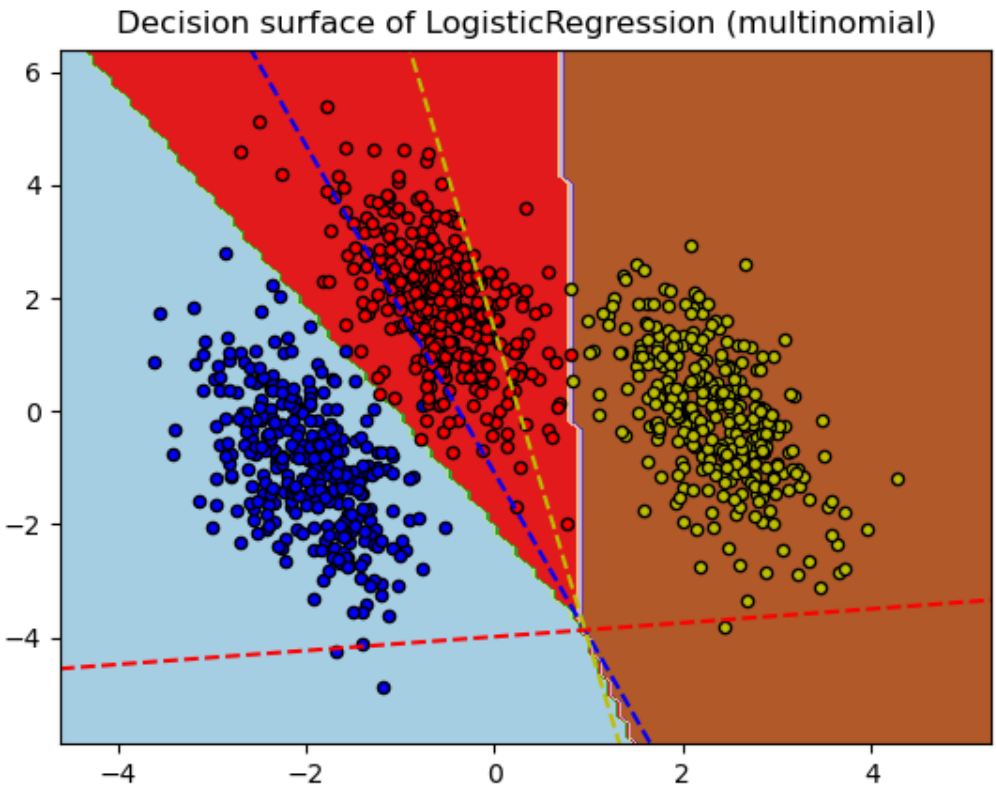
- $P(\text{no pneumonia}|x) = 0.7$ (70% chance of no pneumonia)
- $P(\text{bacterial pneumonia}|x) = 0.2$ (20% chance of bacterial pneumonia)

- $P(\text{viral pneumonia}|x) = 0.1$ (10% chance of viral pneumonia)

- $P(\text{bacterial pneumonia}|x) = 0.2$
- $P(\text{viral pneumonia}|x) = 0.1$

In this example, the model predicts that the patient is most likely to have no pneumonia.

Here's a simplified visual representation of these steps:



Training the Model

Training a multinomial logistic regression model involves finding the coefficients $b_{j0}, b_{j1}, ..., b_{jn}$ that maximize the likelihood of the observed data. This is typically

done using a method called maximum likelihood estimation (MLE).

Logistic Regressions For Pneumonia Detection

In our case, we will apply logistic regression to predict if a patient has no pneumonia, viral pneumonia or bacterial pneumonia using image data.

The dataset consists of images labeled as “NORMAL”, “VIRUS” or “BACTERIA”. We will train our model with default and tuned hyperparameters.

Default configurations :

- **Optimizer** : Grid Search with Cross-Validation
- **Hyperparameters** : Regularization (C), Penalty (l1,l2), Max Iterations, Solver
- **Test Size** : 20%

Data Preparation :

- **Convert** images to grayscale.
- **Normalize** image to a fixed size (256×256)
- **Flaten** the images to 1D array
- **Labelize** the images as 0 (NORMAL), 1 (VIRUS), 2 (BACTERIA)

> 🎯 The goal is to use logistic regression to classify pneumonia to classify pneumonia types from image data.

D. Convolutional Neuronal Network

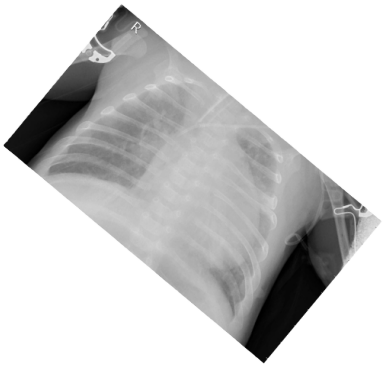
> We are going to lists some techniques which were used during our trainings.

1) Data-augmentation

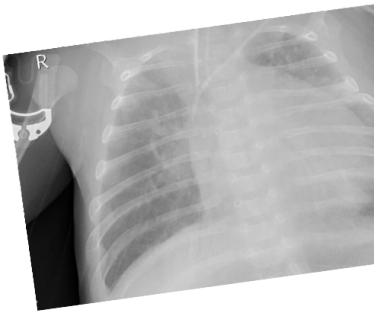
Data augmentation is a technique used to **artificially** increase the size of a training dataset by creating modified versions of existing data. This is useful in training Convolutional Neural Networks (CNNs) for image classification tasks. Data augmentation helps improve the generalization capability of the model and reduces overfitting.

Common Data Augmentation Techniques With Examples

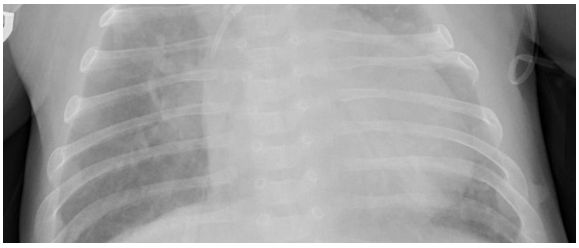
Rotation: The images are rotated by a random angle between -40 to +40 degrees. This helps the model learn to recognize objects regardless of their orientation.



Shearing: The images are tilted at random angles, making them look like they're viewed from different perspectives.



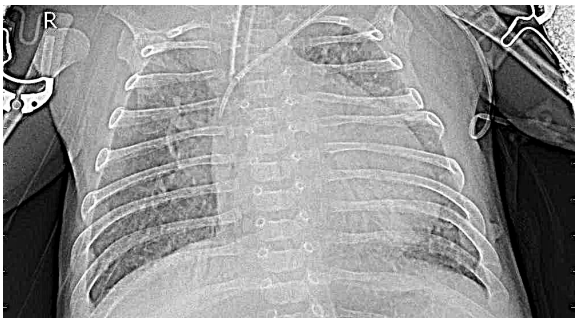
Zooming: The images are randomly zoomed in or out by up to 20%. This helps the model recognize objects at different scales.



Horizontal Flipping: The images are flipped horizontally. This helps the model learn to recognize objects in both left-to-right and right-to-left orientations.

Brightness Adjustment: The brightness of the images is randomly adjusted. This helps the model recognize objects under varying lighting conditions.

Noise Addition: Random noise is added to the images. This helps the model become more robust to noisy inputs.



2) Cross-Validation

Cross-Validation is a technique used to evaluate the performance of a model by partitioning the data into multiple subsets. Its goal is to avoid splitting our dataset into a train - validation - test, since we will only end up with a train - test dataset. This way, the train dataset can be bigger and we get more training resources.

What is Cross-Validation?

Cross-validation is a statistical method used to evaluate the performance of a ML model. it partitionate the data into subsets, training the model on some subsets (training set), and evaluates it on the remaining subsets (validation set).

K-Fold Cross-Validation

Process :

- **Split** : The dataset is split into k equal folds.
- **Training and validation** : The model is trained k times, each time using a different fold as the validation set and the left k-1 folds as the training set.

Example :

For 5-fold cross-validation :

- Divide the dataset into 5 folds.
- Train the model 5 times, each time using a different fold as the validation set and the remaining 4 folds as the training set.
- Average the performance metrics (accuracy, loss, F1 score...) from all 5 runs.

Limits :

Upon doing a K-Fold Cross-Validation, the issue is that we might end-up with a model which learns patterns out of the validation split. Due to this, the results obtained through validation upon evaluating our model can be drastically different from what we get with the test dataset.

IV - Models comparison

To evaluate the effectiveness of a given model, we utilize a variety of metrics, including Accuracy and F1 score among others. In this chapter, we will explore the specific metrics we have chosen to measure, providing detailed explanations of each metric's significance and functionality. This will enable us to draw meaningful comparisons between our different models. We will go into the reasoning behind our selection of these metrics, and how they contribute to a comprehensive assessment of model performance.

By the end of this chapter, you will have a clear understanding of the tools we use to gauge the quality of our models and the reasons for their importance in our analysis.

A. Metrics

Different model have different means to be evaluated, depending on if it’s trained to classify 2 classes or 3 classes. That means that the metrics won’t be the same for these models based on the number of classes they try to predict.

1) Metrics for the perceptron

We are using 3 metrics here to evaluate our model:

$$FPR = FP/(FP + TN)$$

$$FNR = FN/(FN + TP)$$

$$Accuracy = (TP + TN)/(TP + TN + FP + FN)$$

- `TP (True Negative)`: The number of actual negative instances that are correctly classified as negative.
- `TP (True Positives)`: The number of actual positive instances that are correctly classified as positive.
- `FP (False Positives)`: The number of actual negative instances that are misclassified as positive.
- `FN (False Negatives)`: The number of actual positive instances that are misclassified as negative.

False Positive Rate (FPR) measures the proportion of negative instances that are misclassified as positive.

False Negative Rate (FNR) measures the proportion of positive instances that are misclassified as negative.

The accuracy, which is basically the amount of right answers over the dataset size. It measures the precision of our model, how accurate it is at predicting the right answer. It ranges from 0 to 1, 0 being a mistake on each prediction and 1 being a perfect prediction each time.

2) Metrics for the CNN and logistic regression

We are using 3 metrics to evaluate our model:

$$Accuracy = (TP + TN)/(TP + TN + FP + FN)$$

The accuracy, which is basically the amount of right answers over the dataset size. It measures the precision of our model, how accurate it is at predicting the right

answer. It ranges from 0 to 1, 0 being a mistake on each prediction and 1 being a perfect prediction each time.

The loss, which is calculated with the cross-entropy function. Of course, we apply a Softmax before calculating the loss. It ranges from plus infinite to 0, the higher it is and the worse our model performs. We use this metric because the accuracy isn't always accurate. It's meaningful in the human language, but in the machine language having this metric is more meaningful and more precise to see how it performs.

$$F1 = 2 * (precision * recall) / (precision + recall)$$

The F1 score where:

- Precision = TP / (TP + FP)
  - a.k.a the accuracy
- Recall = TP / (TP + FN)
  - evaluates the accuracy our model predicts "true".
- `TP (True Positives)`: The number of actual positive instances that are correctly classified as positive.
- `FP (False Positives)`: The number of actual negative instances that are misclassified as positive.
- `FN (False Negatives)`: The number of actual positive instances that are misclassified as negative.

This metric ranges from 0 to 1, 0 being the worst score possible and 1 being a perfect score. It's a very useful metric when our model is imbalanced and has one class which is over-represented compared to the other, which is our current case (we have way more bacteria than the others).

An F1 score of 0.8 is already great, and 0.9 is very strong. We want to avoid anything below that if possible.

We normally use the F1 score for binary classifications. In our case, we are using it for more than 2 classes, which makes it harder to calculate. We need to calculate

the F1 score for each class, then make a macro-average F1 score to get the right one. Everything is done under the hood by PyTorch and Sci-kit learn.

B. Perceptron

This model has shown impressive results despite its simplicity, exceeding 94% accuracy in a binary classification scenario. But not only it did have a great accuracy, it also had a very low false negative rate (FNR) at only about 2%. This model was tested with a few different learning rate schedulers.

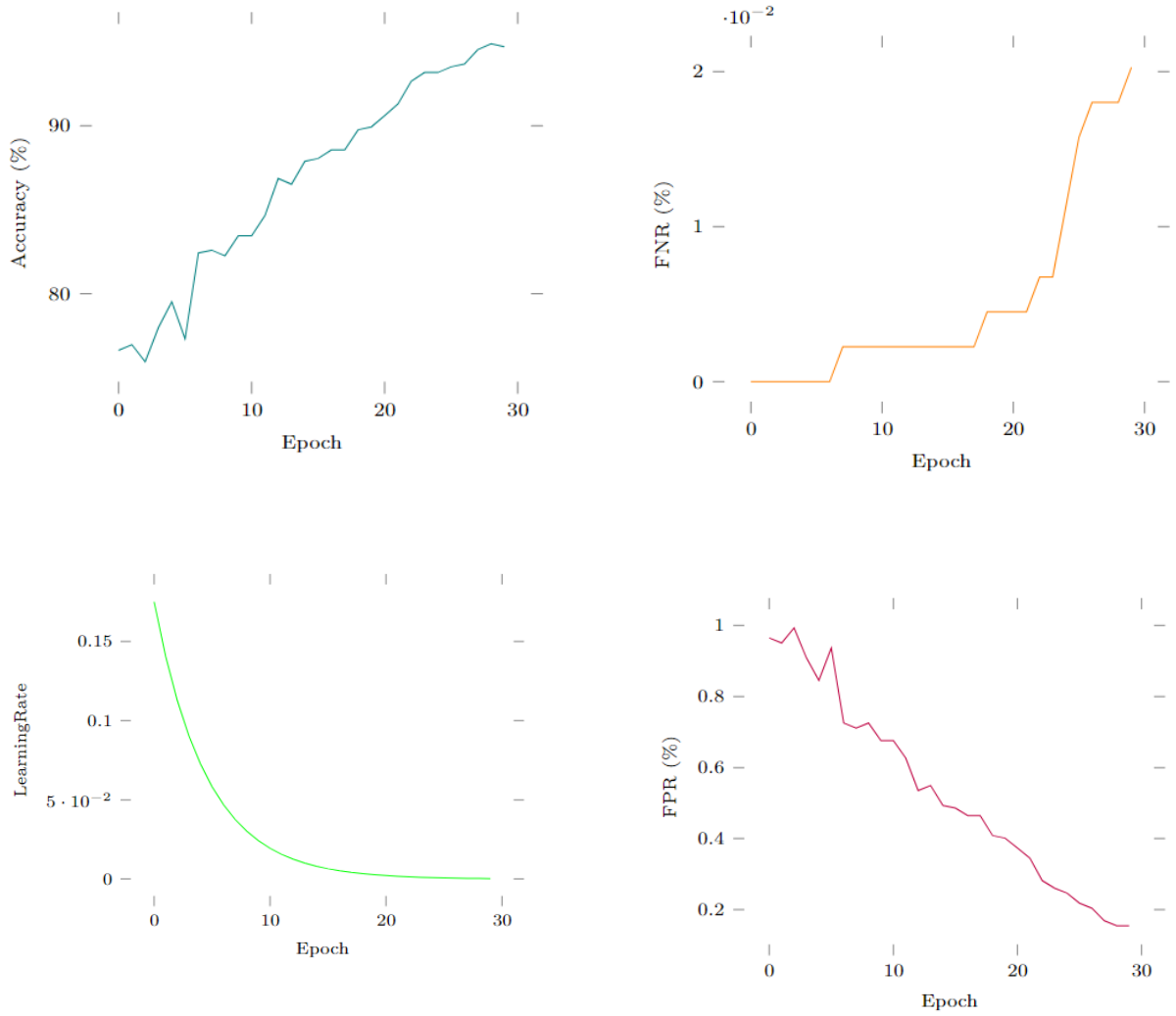
1) No Scheduler

This method consists of giving a fixed value for the learning rate. However it was not able to show any interesting results that would be worth comparing here.

2) Exponential Decay

$$lr * exp(-dr * e)$$

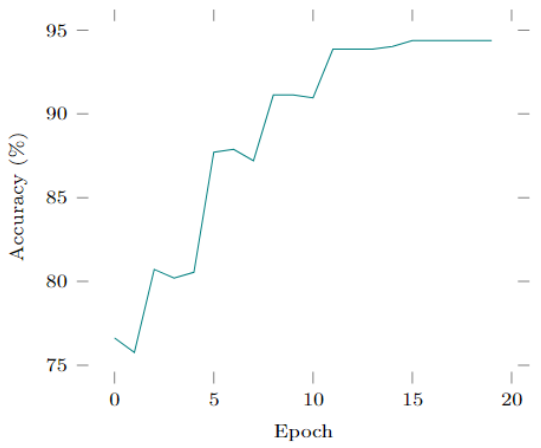
Where `lr` is the current **learning rate**, `dr` is the **decay rate** parameter and `e` is the **epoch**. This scheduler showed a consistent ability to push the accuracy over 94% with a **decay rate** of 0.220, a base **learning rate** of 0.175 and and a total of 30 **epochs**.

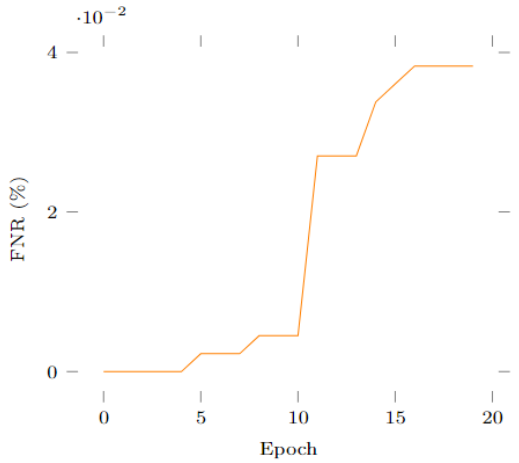


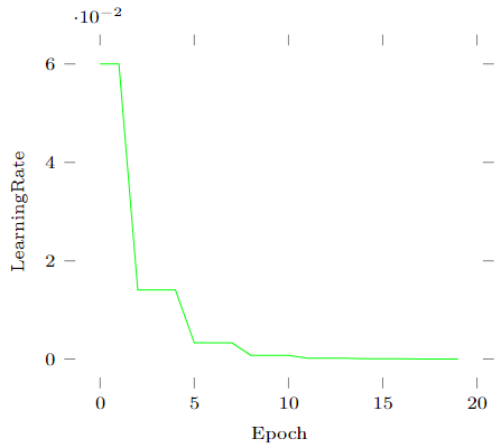
3) StepDecay

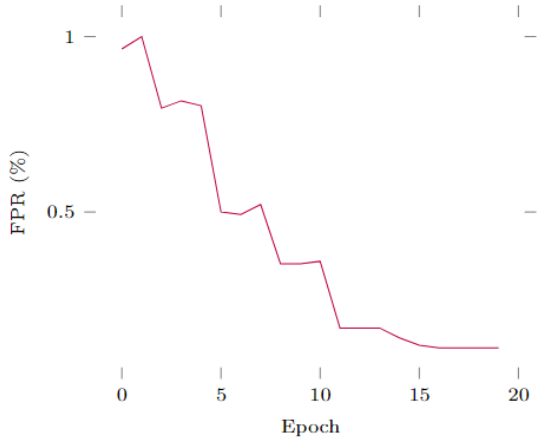
$$lr * pow(df, (1 + e)/ss)$$

Where `lr` is the current **learning rate**, `df` is the **decay factor** parameter, `e` is the **epoch** and `ss` is the **step size**. This scheduler showed a consistent ability to push the accuracy over 94% with a **decay factor** of 0.234, a base **learning rate** of 0.060, a **step size** of 3 and and a total of 20 **epochs**.









C. Multi Layer Perceptron

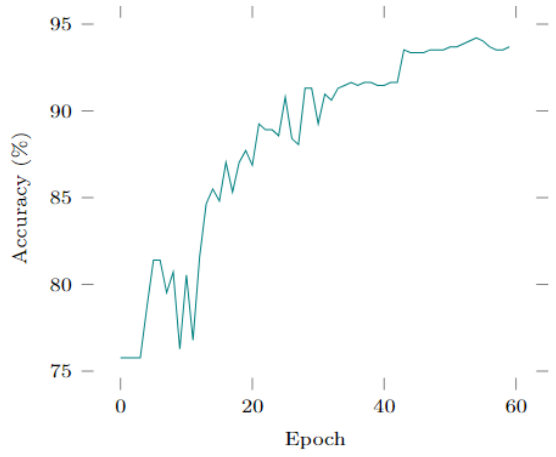
Juste like the Perceptron, MLP has also shown quite impressive results. We have been trying a few combinations of architectures and learning rate schedulers and here are the ones that came out on top.

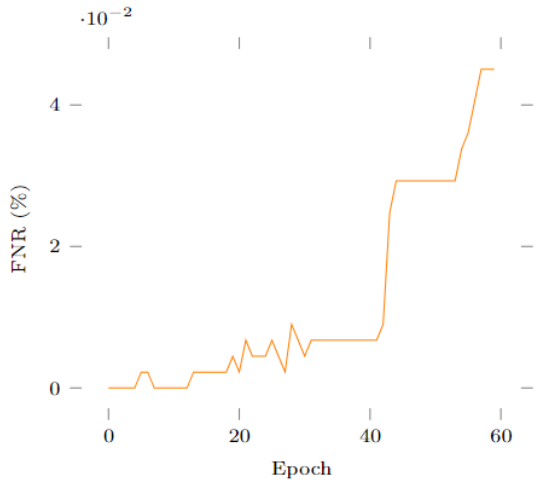
⚠ Since these models have been handcrafted from scratch, we did not test models with more than 1 hidden layer as this would take too much time developping a full neural network.

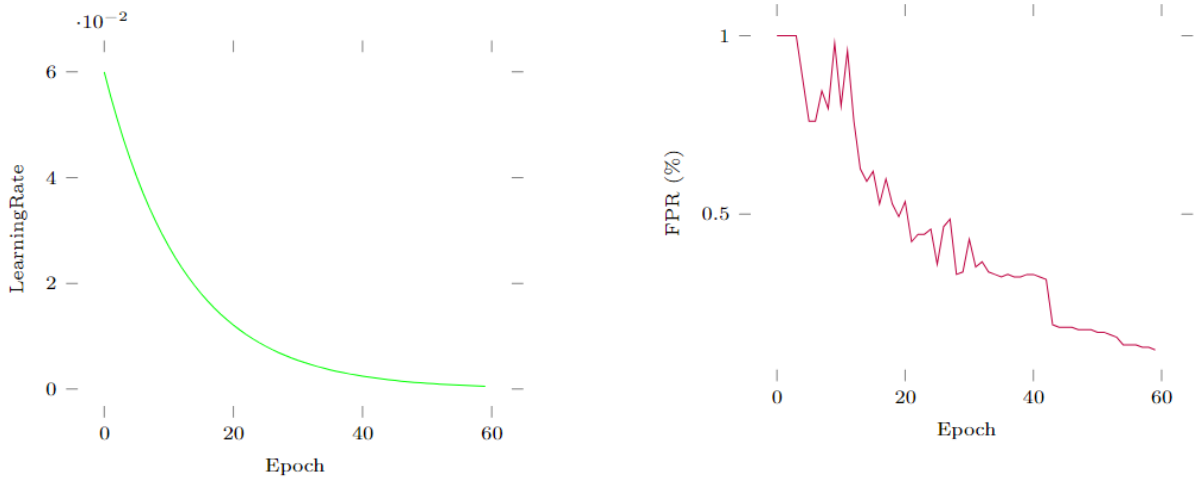
The first one had the following configuration:

- LR Scheduler:
  - Type: Exponential Decay
  - Learning rate: 0.060
  - Decay Rate: 0.075
- Epochs: 60
- Hidden Layers:
  - Size: 4

Despite the fact that it took more time to train than a simple Perceptron, this MLP had approximately the same results, almost reaching 95% accuracy.







In the second one, we went for a more complex architecture, with 8 neurons instead of 4. With these changes, the optimal parameteres were the following:

- LR Scheduler:
  - Type: Exponential Decay
  - Learning Rate: 0.025
  - Decay Rate: 0.075
- Epochs: 60
- Hidden Layers:
  - Size: 8

D. Multi-class models

1) CNN

I Convolutional Neural Network (CNN) Algorithm

A Convolutional Neural Network (CNN) is a deep learning algorithm commonly used for image recognition and classification tasks. Here's a step-by-step explanation of how a CNN works:

Input Layer:

- The input layer receives the raw pixel values of an image. For instance, a 28×28 grayscale image has 784 pixels, and each pixel is an input feature.

Convolutional Layer:

- This layer applies a set of filters (kernels) to the input image. Each filter slides (convolves) over the input image and performs element-wise multiplication, producing a feature map. Filters help detect different features such as edges, textures, or specific patterns within the image.
- The convolution operation reduces the spatial dimensions (width and height) of the input.

ReLU Activation Function:

- After convolution, an activation function called ReLU (Rectified Linear Unit) is applied to introduce non-linearity into the model. This function replaces all negative pixel values in the feature map with zero, keeping positive values unchanged.
- ReLU helps the network learn complex patterns and relationships in the data.

Pooling Layer:

- A pooling layer (typically Max Pooling) is used to reduce the spatial dimensions of the feature maps further. It does this by taking the maximum value from a set of neighboring pixels (e.g., a 2×2 block).
- Pooling helps reduce the computational load and the risk of overfitting by summarizing the presence of features in sub-regions of the image.

Flattening:

- After several convolutional and pooling layers, the 2D feature maps are flattened into a 1D vector. This step prepares the data for the fully connected layers.

Fully Connected (Dense) Layers:

- These layers are traditional neural network layers where each neuron is connected to every neuron in the previous layer. Fully connected layers help combine the features extracted by the convolutional layers to make final predictions.

- Typically, there are one or more dense layers, followed by an output layer

Output Layer:

- The output layer produces the final predictions. For example, in a classification task with 10 classes, the output layer will have 10 neurons, each representing the probability of the input image belonging to one of the classes.
- An activation function such as Softmax is often used in the output layer to convert raw scores into probabilities.

Simple CNN Model Architecture

Here's a simple architecture of a CNN model for image classification:

1. **Input Layer**: Takes in the raw image pixel values.
2. **Convolutional Layer**: Applies a set of filters to the input image to create feature maps.
3. **ReLU Activation**: Applies the ReLU activation function to introduce non-linearity.
4. **Pooling Layer**: Uses max pooling to reduce the spatial dimensions of the feature maps.
5. **Convolutional Layer**: Adds another set of filters to further extract features from the image.
6. **ReLU Activation**: Again, applies the ReLU activation function.
7. **Pooling Layer**: Uses max pooling to reduce the dimensions of the feature maps.
8. **Flattening**: Flattens the 2D feature maps into a 1D vector.
9. **Fully Connected Layer**: A dense layer to combine features and make predictions.
10. **Output Layer**: A dense layer with the number of neurons equal to the number of classes, using the Softmax activation function for classification.

Test differently activate function

Epoch	Loss Sigmoid	Loss ReLU	Loss Tanh	Loss Softmax
1	0.693	0.687	0.690	0.695
2	0.692	0.684	0.689	0.694
3	0.691	0.682	0.688	0.693
...	...	...	...	...
18	0.650	0.600	0.620	0.675

Overall Performance:

- The loss decreases for all activation functions over the epochs, indicating an improvement in the model's fit.

Interpretation:

- **ReLU**: Generally the best activation function for simple neural network models, as it helps overcome the vanishing gradient problem and speeds up learning.
- **Tanh**: Also performs well, often used for hidden layers as it is zero-centered, unlike Sigmoid.
- **Sigmoid**: May be less effective for hidden layers due to the vanishing gradient problem.
- **Softmax**: Primarily used for the output layer in multi-class classification problems and may not be ideal as an activation function for hidden layers.

Okay, here's how to test different activation functions and visualize the results using a simple model with 18 epochs, without going into code details.

Steps to follow

1. **Select Activation Functions** :
    - Sigmoid
    - ReLU
    - Tanh

- Softmax

2. **Select Activation Functions** :
    - Sigmoid
    - ReLU
    - Tanh
    - Softmax
3. **Create a Neural Network Model**:
    - Use a simple neural network model with three layers: an input layer, a hidden layer, and an output layer.
    - Keep the same architecture for all models to compare activation functions fairly.
4. **Train Each Model**:
    - Use the same training data for each model.
    - Train each model for 18 epochs.
    - Record loss after each epoch for each activation function.
5. **Create Results Table** :
    - Record the loss for each activation function at each epoch in a table.

Epoch	Loss Sigmoid	Loss ReLU	Loss Tanh	Loss Softmax
1	0.693	0.687	0.690	0.695
2	0.692	0.684	0.689	0.694
3	0.691	0.682	0.688	0.693
...	...	...	...	...
18	0.650	0.600	0.620	0.675

2) CNN - pseudo ResNet

When it comes to a CNN models, there are more than one way to tweak it. Every hyper-parameter has its own importance, and to so its impact on the model we can

change it for several values and see the results in the metrics. Of course, while doing so, we keep the other hyper-parameters constant to avoid any bias.

Here are the default configurations:

- Optimiser: Adam
- Learning Rate: 1e-3
- Epochs: 15
- Gradient clipping: 1
- Image size: 75*75
- Batch size: 50
- Data Normalisation:

```
NORMALIZED_MEAN = [0.5, 0.5, 0.5]
NORMALIZED_STD = [0.5, 0.5, 0.5]
```

- Data augmentation
  - Random horizontal flipping
  - Random rotation ± 10 degrees
  - Brightness = 0.1
  - Contrast = 0.1
  - Saturation = 0.1

The reason behind the modification of brightness, contrast and saturation is to highlight details through these modifications.

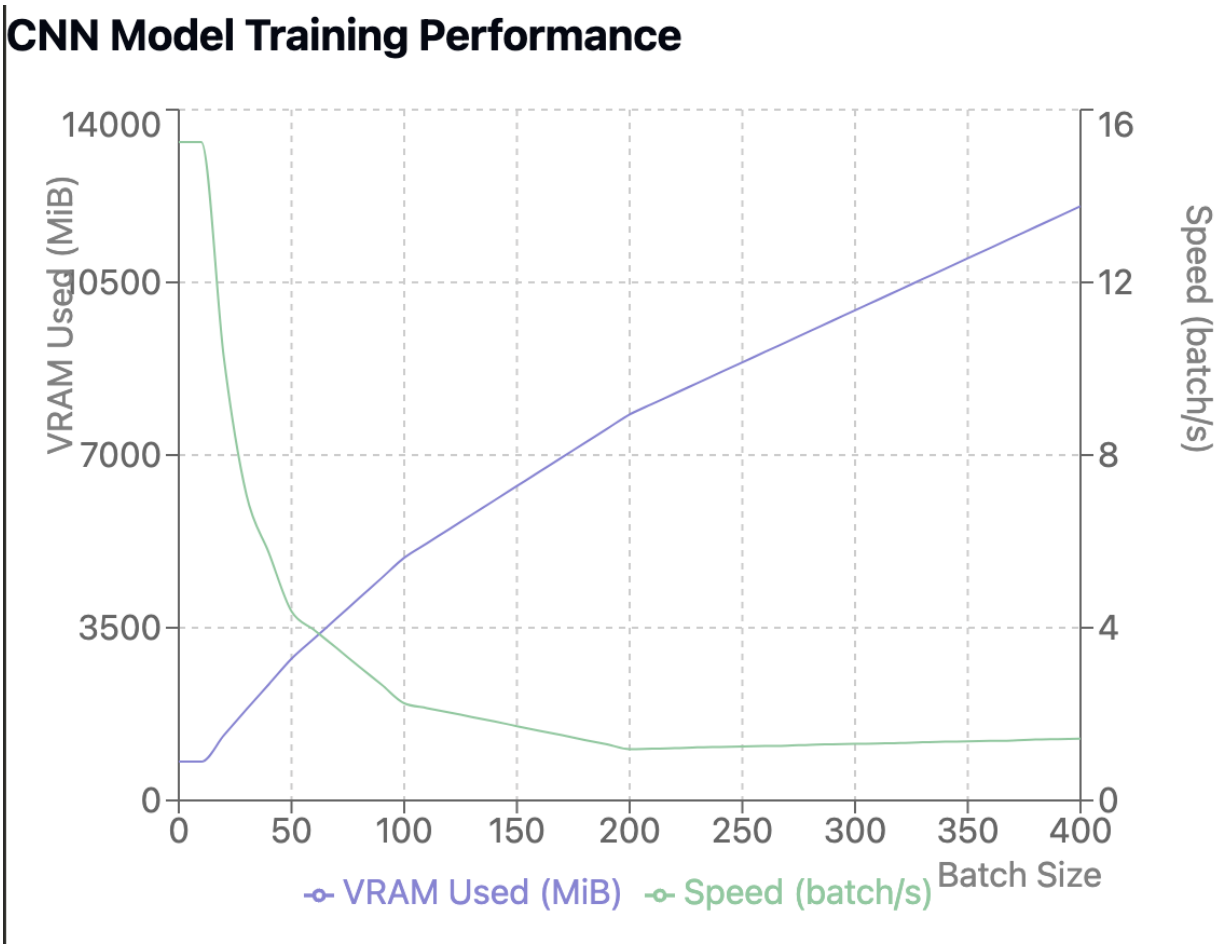
Batch sizes

🎯 The goal is to see which batch size works the best for speed and efficiency. We want our model to still be accurate with a certain amount of batch while being able to train quickly.

For this specific training set, we have a fit-one-cycle learning rate scheduler set. It's not present in the other training sets if not stated.

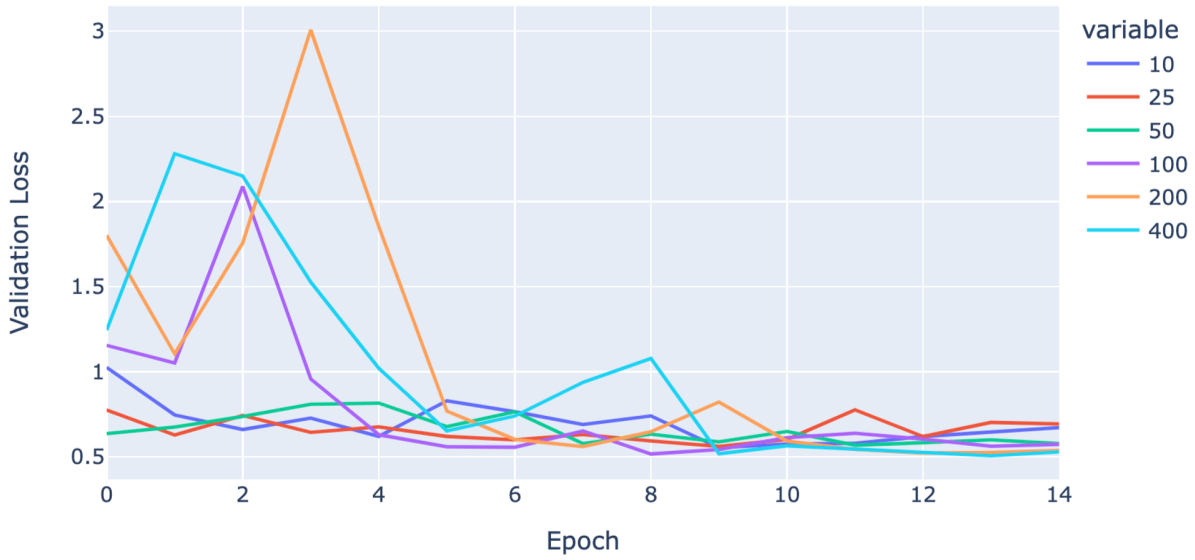
Here is a table summarising all the batch sizes used with its graph to visualise the data:

Batch Size	VRAM used	Speed
400	12043MiB	1.43 ± 0.1 batch/s
200	7825MiB	1.19 ± 0.1 batch/s
100	4919MiB	2.25 ± 0.1 batch/s
50	2871MiB	4.38 ± 0.1 batch/s
25	1587MiB	7.75 ± 0.12 batch/s
10	785MiB	15.25 ± 0.25 batch/s



Double line chart of the VRAM and speed in batch per second relative to the batch size.

As we can see, the speed plateau at 200 of batch size while the VRAM consumption still rises up, which means that it’s useless to go any higher than this.



The evolution of loss on the validation dataset relative to the batch size and the number of epochs

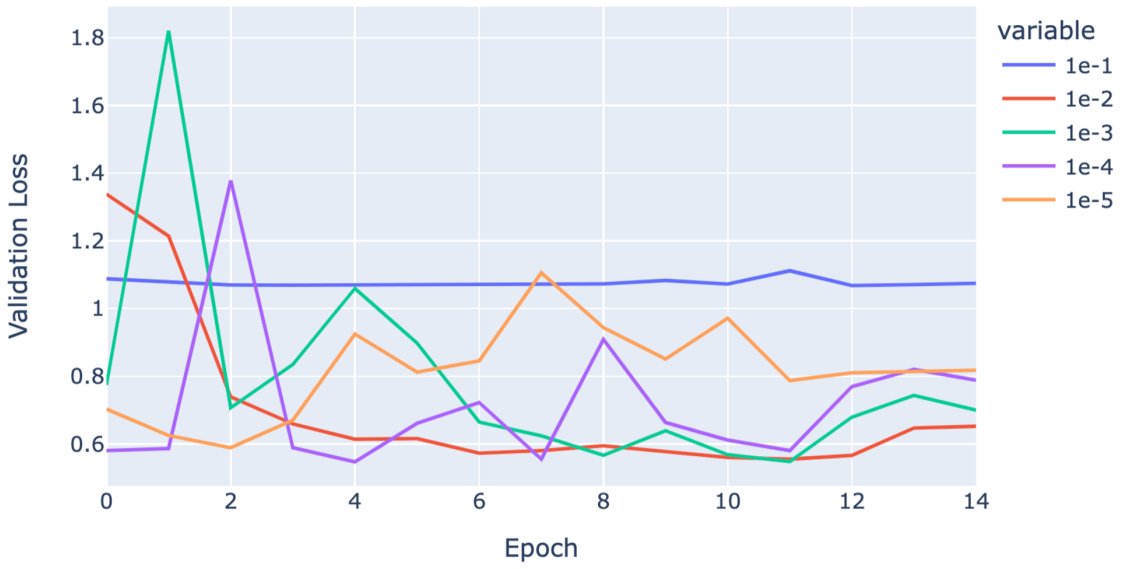
This graph displays the evolution of the model's loss relative to the number of epochs. Each line represent a different batch size. What we can infer from this graph is that the higher the batch size, the more epochs it takes to converge quickly toward a low loss. We need to look at what happens before 6 epochs to see this. At 400, 200 and 100 epochs, the model struggles to quickly converge. On the other hand, below 50 epochs, every result seem to be relatively the same. Batch 10 yields extremely similar results to batch 50.

We can conclude that batch 50 is the ideal batch size from this graph.

Learning rate

🎯 The goal here is to see which learning rate yields the best results in terms of loss on the validation dataset. Here are the learning rate values which are being compared:

1e-5, 1e-4, 1e-3, 1e-2, 1e-1



The evolution of loss on the validation dataset relative to the learning rate and the number of epochs

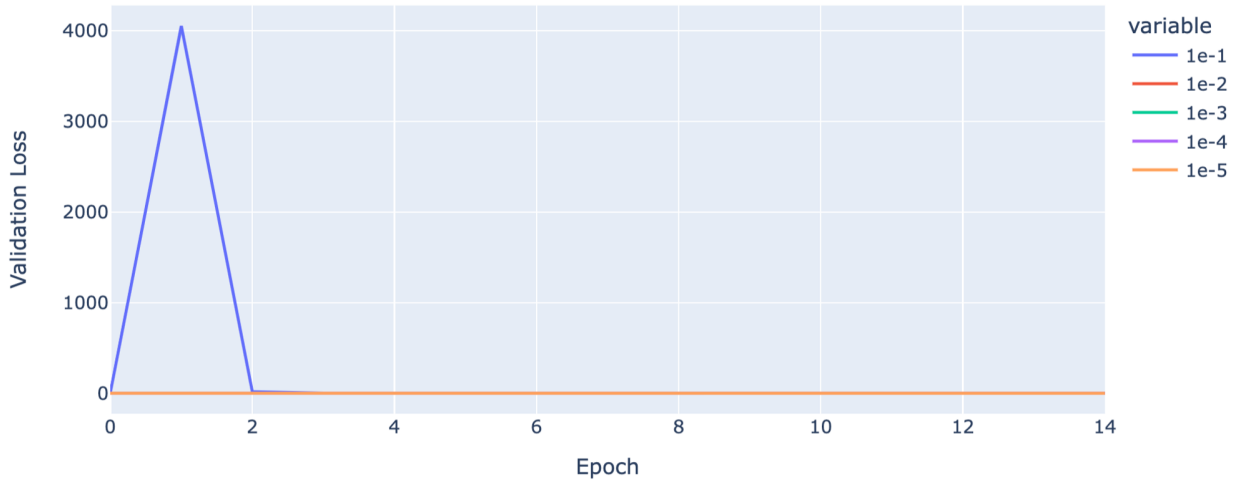
We can infer from this graph that the learning 1e-1 is too large, so the model is not learning at all. On the other hand, 1e-2 is where is starts to get interesting since the loss seem to start converging toward lower values.

We can note that 1e-5 and 1e-4 are the ones which converge the fastest. 1e-5 reaches its minimum loss at only 2 epochs while 1e-4 reaches its minimum at 4 epochs. 1e-2 is the learning rate which is the most consistant in decreasing, but it might just be a fluke so further testing are needed.

Learning rate schedulers

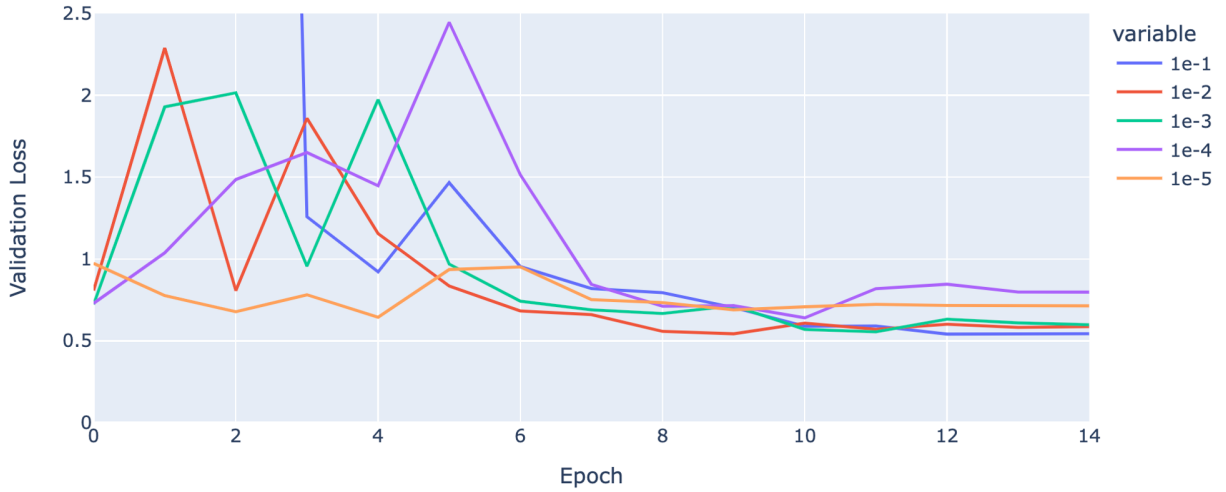
🎯 The goal here is to determine which learning rate scheduler is the most appropriate and yields the best result in terms of loss on the validation dataset. We are going to use the fit-one-cycle with the following learning rates:

1e-5, 1e-4, 1e-3, 1e-2, 1e-1



The evolution of loss on the validation dataset relative to the learning rate with a scheduler and the number of epochs

As seen on this graph, 1e-1 reaches a loss which is so high that it crushes the other lines, so we are going to readjust the y-axis.



The evolution of loss on the validation dataset relative to the learning rate with a scheduler and the number of epochs

In general, the 6 first epochs are quite noisy but the model seem to converge for every cases after 8 epochs to similar values. While 1e-1 reaches absurd number for the loss, it still manages to converge to better values than the other learning rates.

It reaching the lowest value of loss might be a fluke, since it's not significantly different enough from 1e-2 and 1e-3.

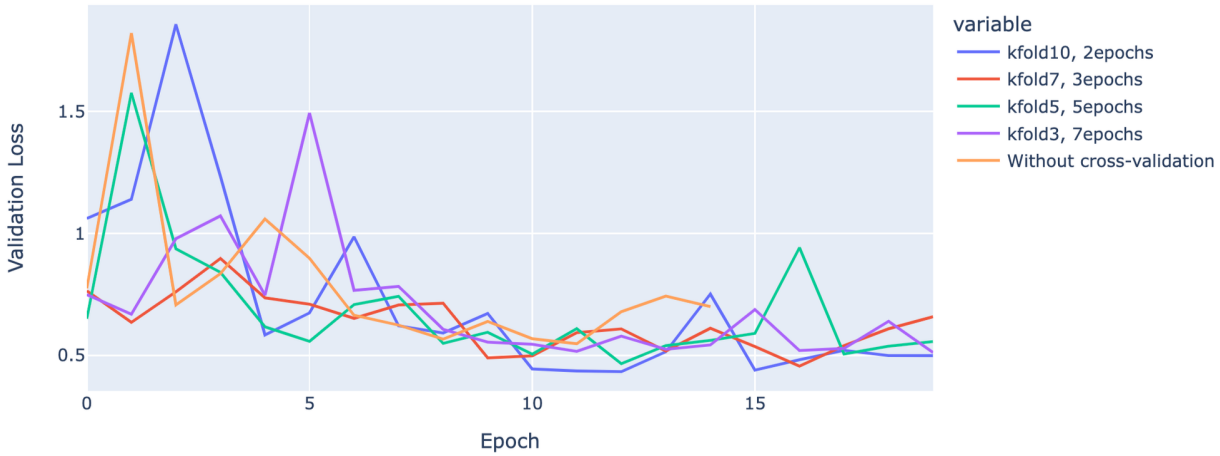
If we were to compare this to without scheduler, we seem to reach lower values of loss in average. 1e-3, 1e-2 and 1e-1 seem to be the learning rates which yield the best results after 14 epochs, better than the minimums obtained without learning rate scheduler.

To note, the first 6 epochs are quite noisy because the learning rate increases and decreases a lot during these epochs, which impacts the a lot the loss at that section of the training.

Cross-validation

🎯 The goal here is to see if cross-validation have a positive impact on the training and result for our model. We tried to remain at 20-21 total epochs throughout all the k-folds for this comparison. Here is a list of what is going to be compared:

- 10 k-folds and 2 epochs
- 7 k-folds and 3 epochs
- 5 k-folds and 5 epochs
- 3 k-folds and 7 epochs
- Without cross-validation



The evolution of loss on the validation dataset relative to the number of k-folds and the number of epochs

The validation dataset for the cross validation is quite different from what we get in without cross-validation, so the results should be tested on the test dataset to ensure that no overfitting occurred.

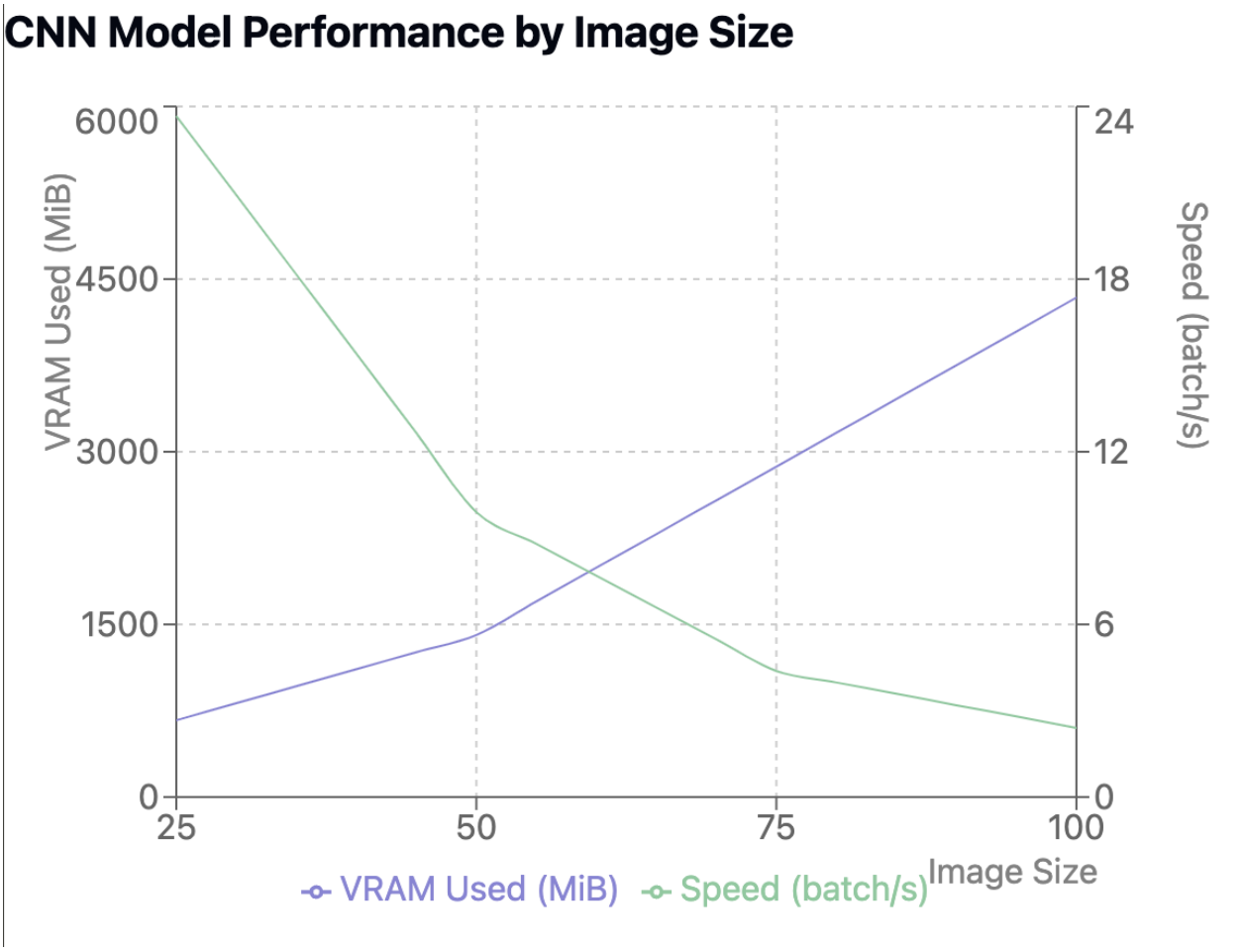
Other than that, 10 k-folds and 2 epochs seem to yield more consistant results. That's only on the surface though, since the validation dataset cycles through the whole dataset so many time, the model gets the time to learn every images from the dataset, so it learns the images from the validation dataset which biases the result. Knowing this, we can't really tell anything from these results.

Image size

🎯 The goal is to see the impact of the image size on our model. We want to know if our model gets more accurate with higher image quality. We will also see the resources consumption per image size.

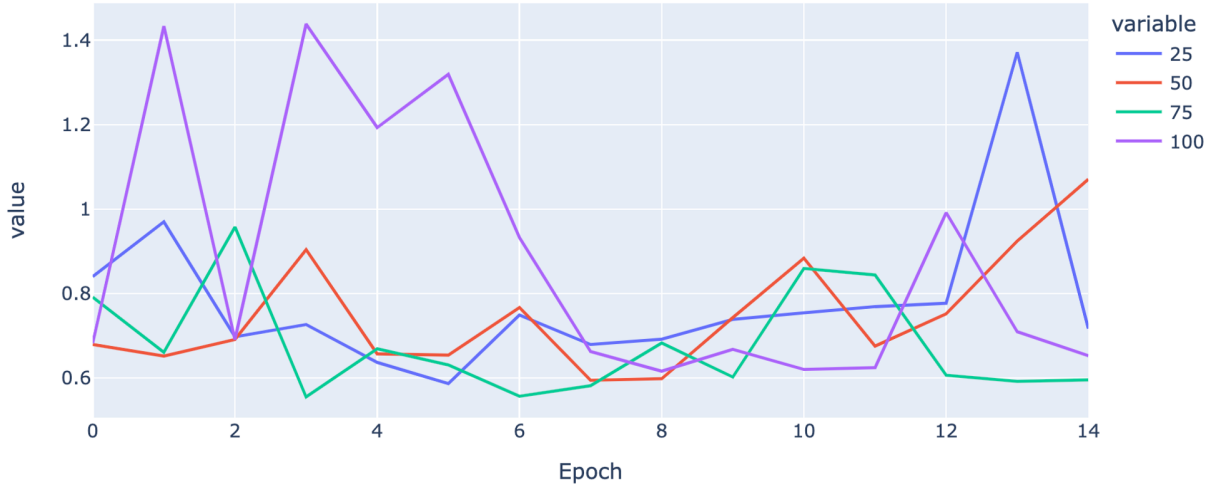
Here is a table summarising the configurations used:

Image size	VRAM used	Speed
100	4339MiB	2.4 ± 0.1 batch/s
75	2871MiB	4.38 ± 0.1 batch/s
50	1407MiB	9.9 ± 0.2 batch/s
25	667MiB	23.67 ± 0.2 batch/s



Double line chart of the VRAM and speed in batch per second relative to the image size

We can infer from this graph that the amount of VRAM used seem to linearly increase as the image size increases while the speed exponentially decreases.



The evolution of loss on the validation dataset relative to the image size and the number of epochs

With an image size of 100, the model seem to struggle for the first 7 epochs. This might be due to images having too much details.

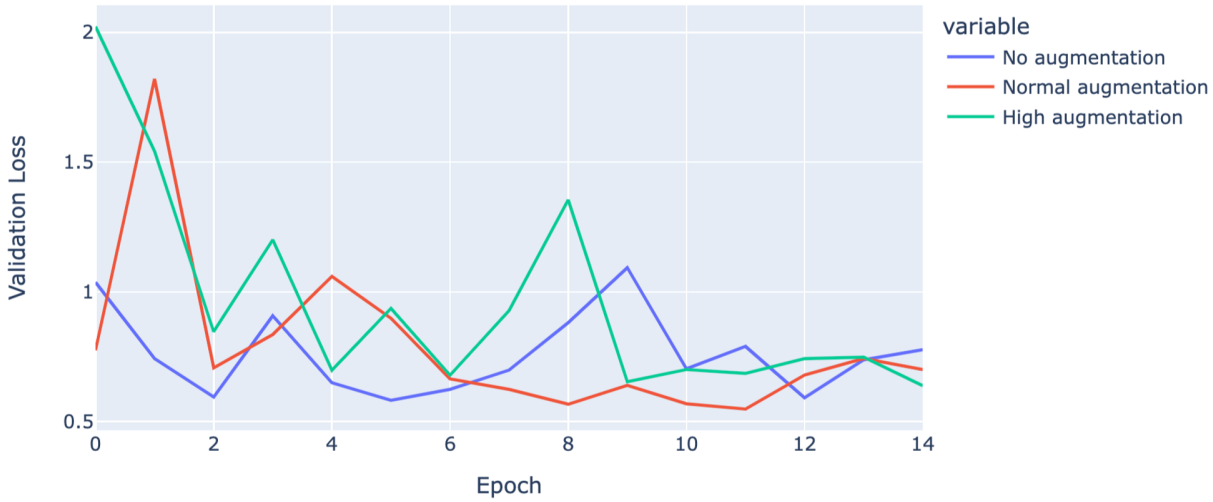
Every image size seem to converge toward 0.5-0.6, 75 image size being the one which does it the fastest.

We can infer from this that 75 is good enough as an image size. Going any higher will only lengthen the training and won't yield better results. Going lower might decrease the performances.

Data augmentation

🎯 The goal is to see if data augmentation allows to avoid over-fitting or not. We have 3 configurations for the data-augmentation:

- Without
- Normal:
  - Random horizontal flip
  - Random rotation ± 10 degrees
  - brightness, contrast and saturation + 0.1
- High:
  - Random horizontal flip
  - Random rotation ± 100 degrees
  - brightness, contrast and saturation + 0.5



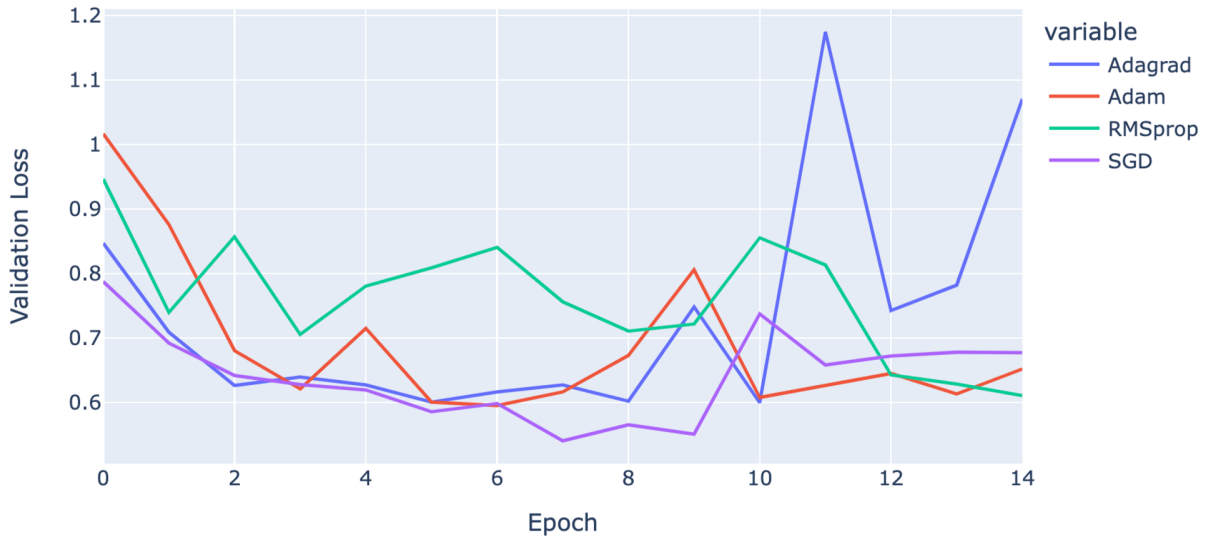
The evolution of loss on the validation dataset relative to the intensity of data augmentation and the number of epochs

We can infer from this graph that no augmentation seem to work the best under these configuration since its way less volatile throughout the training. But in the end, the three of them yield not significant different results, so other data augmentations should be tested.

Optimisers

🎯 The goal is to see which optimiser yields the best result to train our model. Most models seem to use Adam's optimiser, so we will compare it to 3 other optimisers. Here is the list of what's going to be compared:

- SGD (Stochastic Gradient Descent)
- Adam (Adaptive Moment Estimation)
- RMSprop (Root Mean Square Propagation)
- Adagrad (Adaptive Gradient Algorithm)



The evolution of loss on the validation dataset relative to the optimiser and the number of epochs

We can infer from this graph that Adagrad seem to be the most prone to overfit. Adam seem to be performing better than RMSprop overall and SGD appears to be the best optimiser, since it can reach very low loss before starting to overfit.

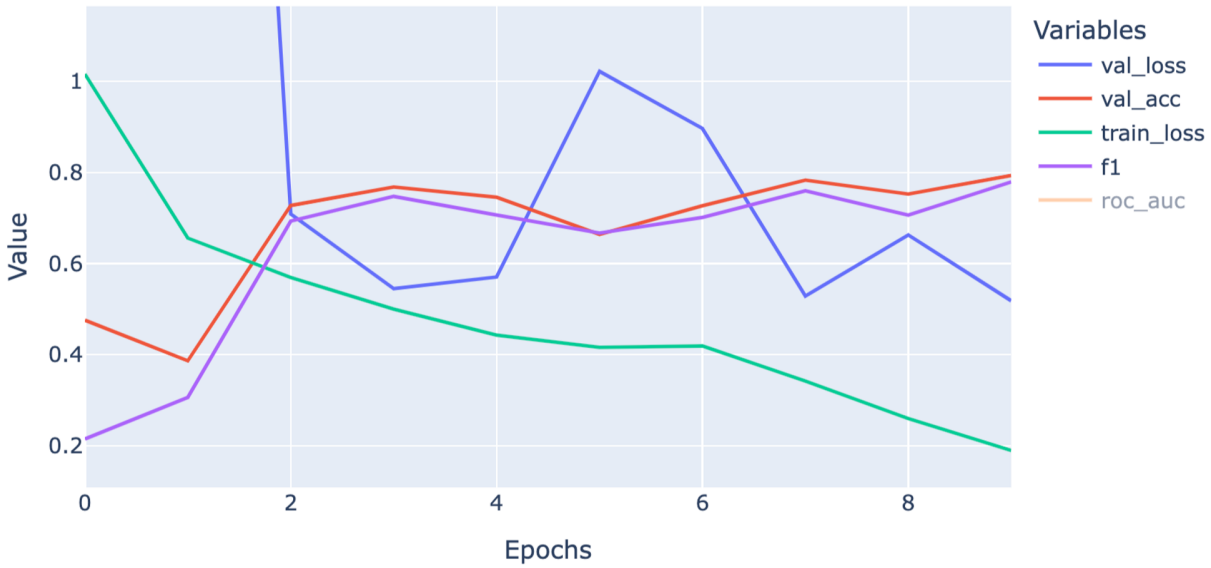
Do note that there is a small error in the Y-axis, it does not go from 0 to 0.6 on the first step. It's from 0.5 to 0.6, so the lowest we've reached is 0.55 loss with SGD and not 0.3.

Final model configuration

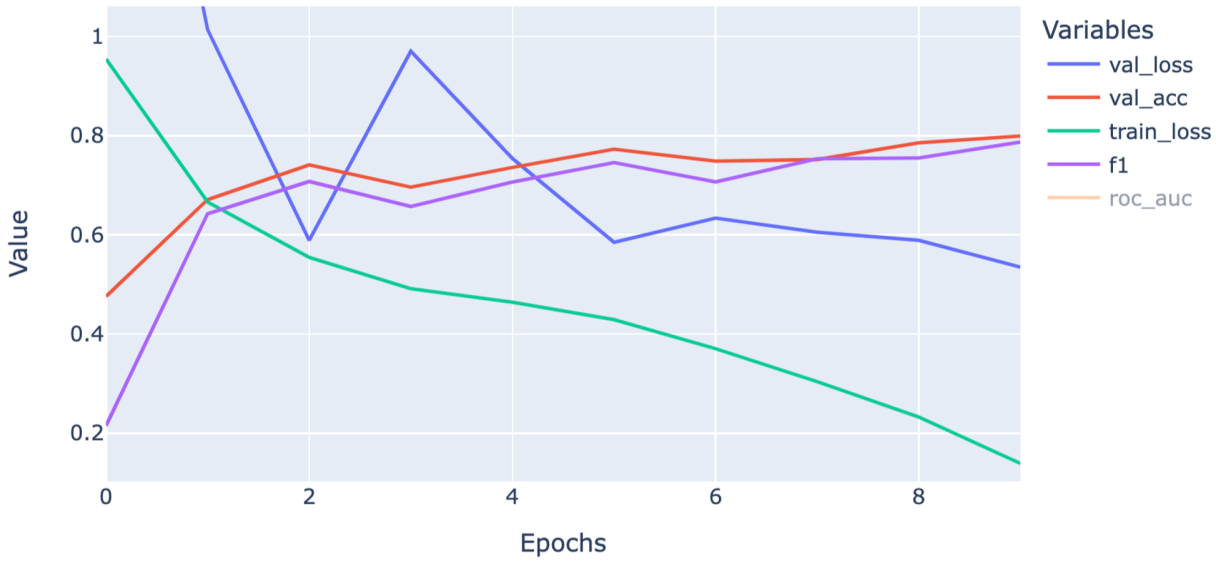
🎯 Knowing all of these, we can now try to arrange better hyper-parameters for our model. Here are all the conclusions from these training sets to choose the hyper-parameters:

- The optimiser with the best results was the SGD optimiser.
- We will choose an image size of 75 since it's not too slow to train while yielding the best results.
- The learning rate will be adjusted to 1e-2.
- We will use the fit-one-cycle learning rate scheduler.
- We don't need to go higher than 10 epochs to get the best model since we use a learning rate scheduler.
- The batch size will be set to 200, since every batch size generally converge toward the the same, it's just a little more noisy for the first 6 epochs.
- We will first try with and then without cross-validation.
- 3 trainings will be done on the same set of hyper-parameters to keep the best one.

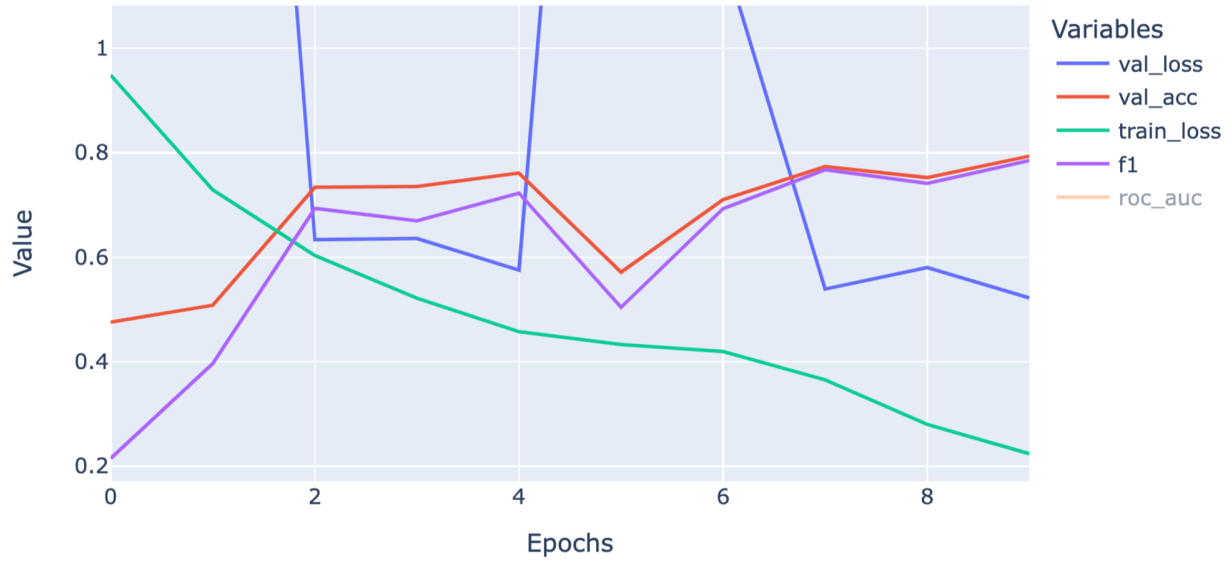
Up until now, we were only taking a look at the loss since plotting the accuracy and F1 score would encumber the graphs and render it unreadable.



Training 1



Training 2



Training 3

In general, the 3 of these trainings reach nearly the same results, but the 2nd one is the one which yields the best ones with these scores:

Train loss	Validation loss	Validation Accuracy	Validation F1 Score
0.1391	0.5349	0.7992	0.7870

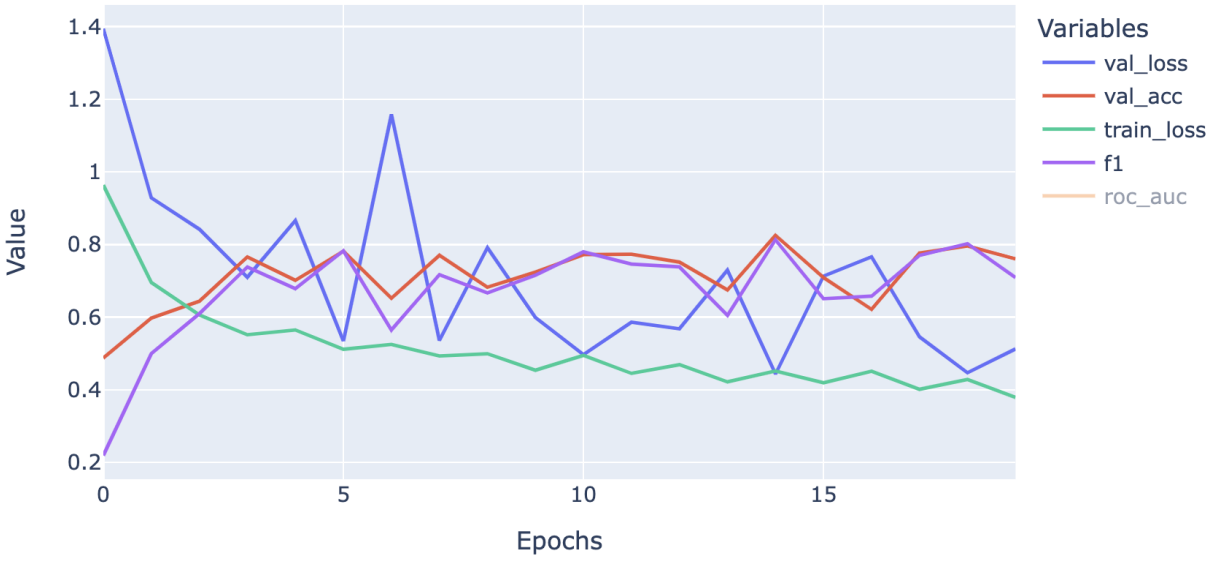
Let’s make some inferences on the Training 2.

We can see that the F1 score closely follows the accuracy, to reach a value close to 0.80 for both. In general, a diminution of the validation loss leads to a lower accuracy and lower F1 score.

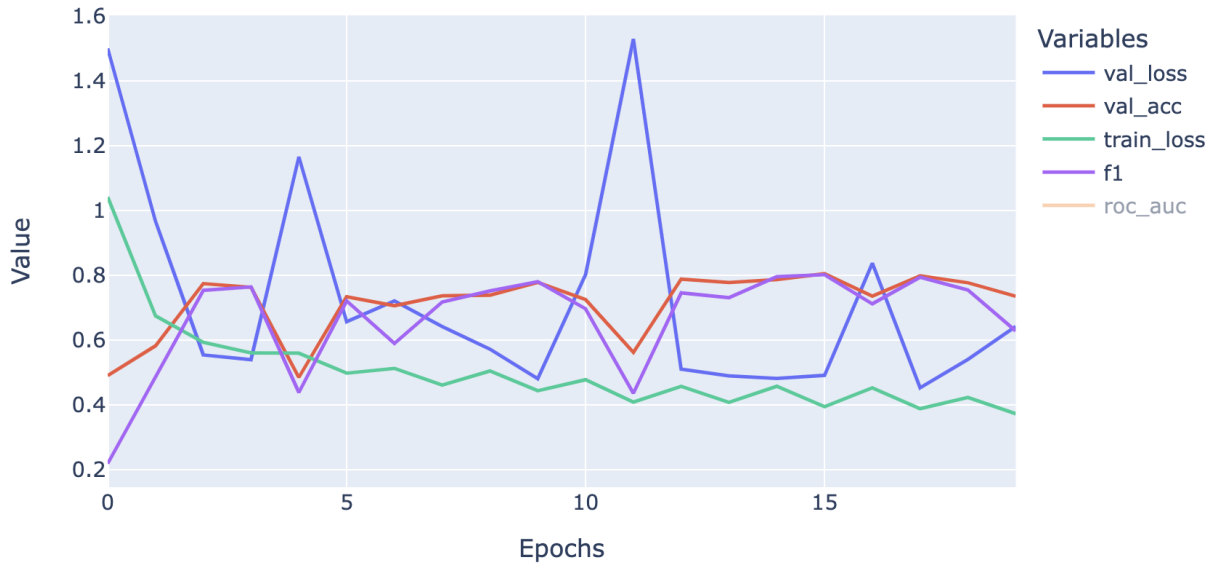
After 2 epochs, the train loss seem to keep going down while the validation loss starts to plateau and very slowly decreasing. Keep in mind that the model still has yet to overfit, since the validation loss is not increasing. This plateau can be explained by several things. Since we use techniques to prevent overfitting such as weight decay and dropouts, this might prevent the increase of the validation loss.

🎯 We will now train the model with a cross-validation. Here are the hyper-parameters:

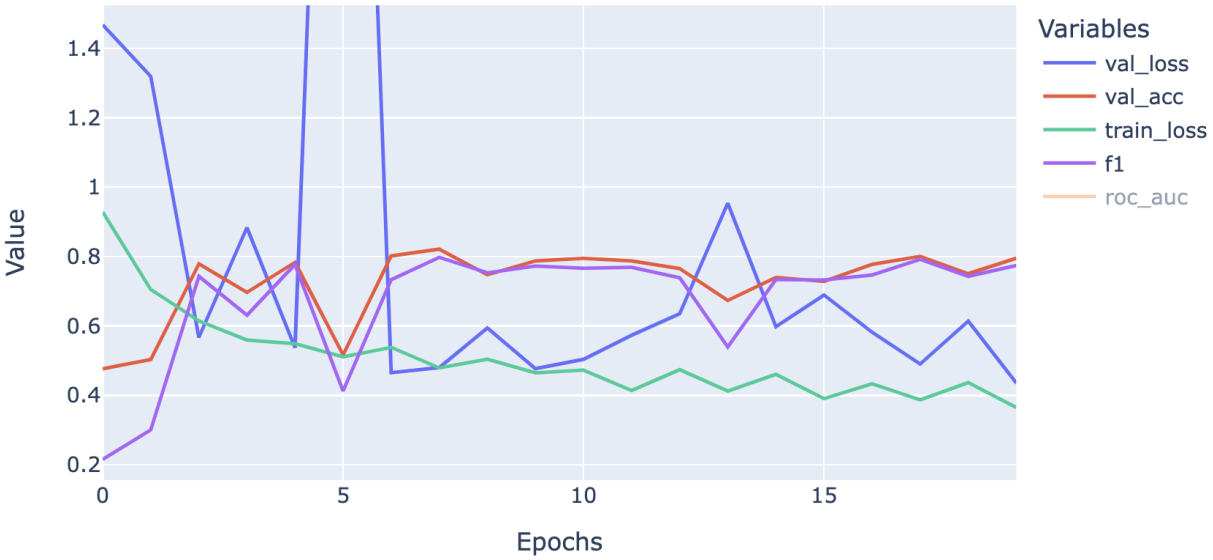
- We won't be using any scheduler here since it does not make sense to use one in a cross-valdation setting.
- We are going to be training over 10 k-folds and 2 epochs
- Other than that, the other hyper-parameters will remain the same.



Training 1



Training 2



Training 3

We now have 3 models, and the 3rd one is the one which performs the best with these results:

Train loss	Validation loss	Validation Accuracy	Validation F1 Score
0.37	0.44	0.8	0.77

Getting these results was very difficult, since the model kept overfitting after a total of 18 epochs for other number of k-fold and epochs. Let's make some inferences out of the Training 3:

- The final loss is quite low, in fact lower than what we got without cross-validation. The accuracy and F1 scores are a bit lower though.
- The validation loss and train loss are quite close compared to without cross-validation.

We will now test our pseudo-ResNet models on the test dataset to see its performances:

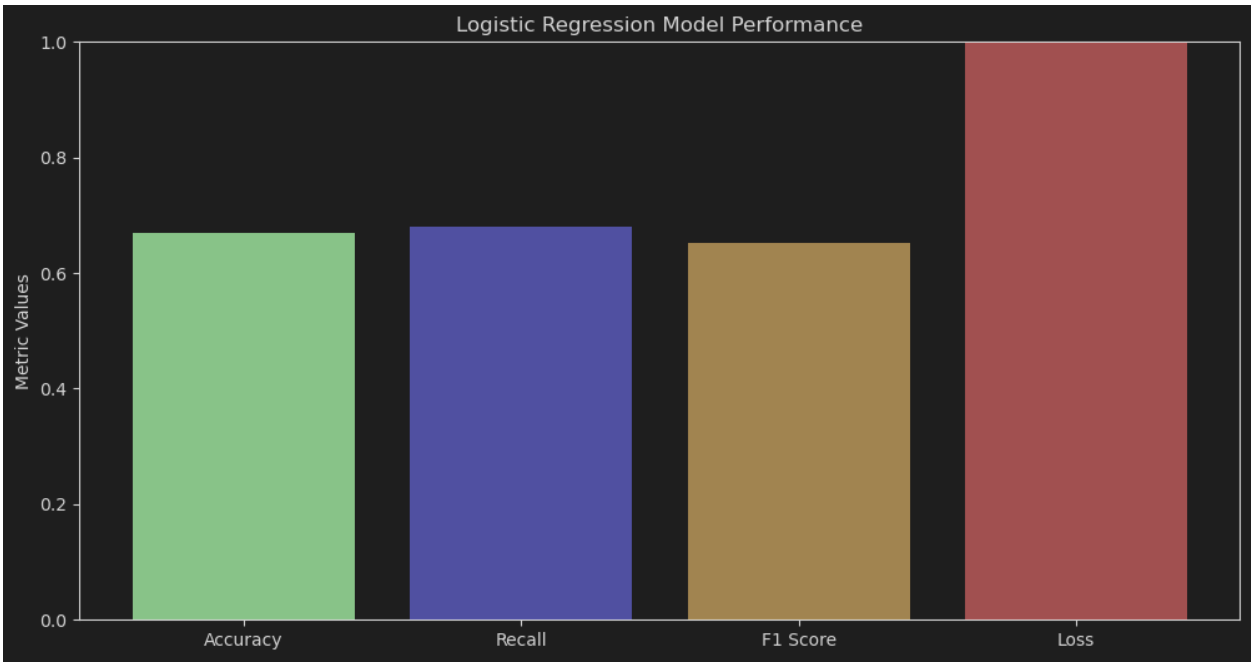
Model	Loss	Accuracy	F1 Score
Pseudo-ResNet	0.48	0.82	0.81
Pseudo-ResNet with Cross-Validation	0.49	0.79	0.76

In general, the cross-validation method performed a little less well than the without on the Test dataset.

3) Logistic regression

Logistic Regression Configuration :

- Data Preprocessing : Grayscale conversion, resizing to 256×256, flattening.
- **Hyperparameter Tuning** :
  - **C** : Regularization strength
  - Penalty : Regularization type (l1 or l2)
  - Max Iterations : 1000 ⇒ 8000



Advantages :

- Simpler and faster to train
- Hyper params tuning with GridSearchCV

Disadvantages :

- Lower performance metrics

Conclusion :

The logistic regression model can be a great alternative in a resource constrained environment.

4) Overall comparison

We will now test every models on 3 classes on the test dataset to see its performances:

Model	Loss	Accuracy	F1 Score	Recall
Logistic regression	11.27	0.67	0.65	0.68
Simple CNN	0.52	0.73	0,80	
Pseudo-ResNet with Cross-Validation	0.49	0.79	0.76	
Pseudo-ResNet	0.48	0.82	0.81	

As seen in these results, the most performant model is the pseud-ResNet in every metrics. While it is performant, it is also the model which is the most resource intensive to train, and is also highly more complexe than the others.

The logistic regression through Scikit-learn didn't perform well due to the dataset, most likely due to the small size of our dataset.

The simple CNN did perform pretty well, and could have probably be further fine-tuned to yield better results.

V - Cloud

To train our pseudo-ResNet model, we chose to use cloud computing. The dataset is stored in an Azure bucket, which serves the VMs when needed. We will dive into more details in how we trained our pseudo-ResNet models in this chapter.

A. Azure Bucket

As said earlier, we used Azure to store our dataset. It's separated in 3 containers (i.e. train, validation and test). We are using hot storage, since we need to

frequently access the datasets as we destroy and rebuild our VMs.

The bucket is also used by everyone to train every models on our project with the same datasets.

B. Cloud computing

We used GCP (Google Cloud Platform) to train our pseudo-ResNet model. The reason behind this is that we can have access to GPUs to train our model. Here are the specs of our VMs:

- 2 vCPU, 13GB of memory
- GPU: 1x TESLA T4
- 50GB of persistant memory
- Hourly cost of 0.4 USD

Of course, we had to destroy and recreate our VMs on several occasions due to GPU availability issues. There are several ways to combat this:

- one way of doing so would be to make an image out of the VM to then recreate other VMs based on that image. This came in handy when we had forgotten to push our code to Github, or had important save states of our model on the VM.
- another solution have been to create an Ansible and Teraform setup to automate the build and destruction of our VMs. This way of doing is great since it allows for more flexibility, and can be very useful for future projects.

So we used Ansible to automate the setup of a PyTorch environment, ready for development. We had to setup Nvidia, Python, Jupyter Notebook, Cuda and its dependencies on the VMs through Ansible playbooks. We now have a tool which is very flexible and can easily create any environment we want to develop in the cloud with PyTorch.

Teraform was used to automatically destroy and create VMs on GCP. It would automatically set the configurations stated above for our VMs, choose the right regions and create the VM.

The workflow was basically to look for an available region with Teraform, create an VM and automate its setup with Ansible. All of this can be done in under 15 minutes.

The choice of GPU was a Tesla T4. It's the most cost efficient GPU for a budget setup in the scenario of training AIs. We also tried using 2 Tesla T4 in a single setup, which did increase the training efficiency by about 1.75 times for a 1.75 time the price increase:

Number of TESLA T4	Speed	Hourly cost (USD)
1	1.3 ± 0.1 batch/s	0.4
2	2.25 ± 0.1 batch/s	0.7

We kept using 1 Tesla T4 though, since we didn't only train our models but also did some exploration on it.

The disk size is set to 50GB for several reasons:

- The system itself (Debian 12) takes about 5GB.
- Nvidia takes about 30GB by itself.
- All the dependencies and environment take roughly 3GB.
- The dataset weights about 2GB.
- Our ResNet models produces save files in .pth which weight about 300-500MB each, so if we want a little bit of margin, we need a little more storage.

C. The price

When it comes to price, we used about 150 USD on this project. Thankfully, it was free credit given by GCP for the most part which also reflects the time spent on everything in this project:

- 30% of this price went into trying to automate everything with Ansible and Teraform

- 25% was used for exploration
- 25% was used for training
- 10% was used for deployment
- 10% was used for building an architecture

Of course, our time could have been used more efficiently, but the credits were about to expire so we came in with the intention of using as much credit as possible from GCP.

D. Deployment

In the end, the model was not fully deployed since we couldn't get our inferences out of it. We had a small module which was made into a tarball and then sent into an AWS bucket, to then be used in a serverless environment to make inferences through AWS SageMaker. We chose a serverless environment since we didn't mind the cold start, as we wouldn't make inferences very frequently.

VI - Conclusion

This project was the occasion to learn how to create the first model for some people, while for other it allowed to stretch their knowledge about machine learning and deep learning to build something which is more akin to what would be done in a professional environment. This allowed to highlight several means to produce a similar result, with different scores of course. Creating and training an image recognition model in this case isn't that difficult, and doesn't require that much resources. That being said, the dataset was quite limiting: its size was way too small. Having roughly 5500 images to work with is not enough to produce strong models with deep learning. This small number led to easily overfitting CNN models, even with all of the optimization techniques we used.

We could have further explored with other models, such as a Densenet, which could have possibly led to better results on this dataset, since it's more appropriate than a ResNet. Other optimization techniques weren't used as well such as early

stopping. The deployment could have been finished. We could have made a report entirely through LaTeX.

All in all, while we could improve this project in some regard, we still produced satisfactory results.