DIPARTIMENTO
DI INFORMATICA
SAPIENZA
UNIVERSITÀ DI ROMA

Analyzing Tradeoffs between Activation Functions and Optimizers in First Gen. Pokemon Classification

Ramil Leonard Vincent (1984766) and Sannino Siria (2001580)

La Sapienza University of Rome, Department of Computer Science
https://github.com/u-siri-ous/analyzing-tradeoffs

Abstract. The program is designed to analyze pictures of Pokémon TCG 1st gen cards. It aims to analyze the tradeoff between combinations of activation functions and optimizers, emphasizing the importance of well-performing variants in a pure, CNN-based image classification task.
The code is released publicly under the AGPL-3.0 License on GitHub.

Keywords: Pokémon · Neural Networks · Convolutional Neural Network (CNN) · Adam Optimizer · ReLU

1 Introduction & Methods

1.1 Convolutional Neural Network (CNN)

A Convolutional Neural Network (CNN) is a type of deep neural network designed to process and analyze visual data. CNNs have proven highly effective in tasks such as image recognition, object detection, and image classification [5]. Key components of a CNN are:

- **Convolutional Layers**
  The core building blocks of a CNN are convolutional layers. These layers apply convolution operations to the input data using filters or kernels. Convolution helps the network learn features like edges, textures, and patterns in a spatially hierarchical manner.
- **Pooling (Subsampling) Layers**
  Pooling layers are used to reduce the spatial dimensions of the input data by down-sampling. Common pooling operations include max pooling, which retains the maximum value in a region, and average pooling, which computes the average value.
- **Fully Connected Layers**
  After several convolutional and pooling layers, fully connected layers are often added to the network. These layers connect every neuron to every neuron in the previous and subsequent layers, forming a dense layer. Fully connected layers are typically used for decision-making and classification.
- **Flattening**
  Before the fully connected layers, the output from the previous layers is flattened into a one-dimensional vector. This is necessary because fully connected layers require a one-dimensional input.

1.2 Activation Functions

An activation function is a mathematical operation applied to the output of a neuron (or a node) in a neural network. It introduces non-linearity to the network, enabling it to learn complex patterns and relationships in the data. In this project, we focus on the ReLU activation function with a piecewise-linear variant, PReLU, and a non-linear variant: GeLU.

ReLU The Rectified Linear Unit is one of the most popular activation functions because it's relatively simply defined:

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

This serves the specific purpose of introducing a gentle non-linearity into the model, while still preserving piecewise linearity.

PReLU PReLU, or Parametrized Linear Unit, is a variant of the classic ReLU, which features a small slope for negative values instead of being flat. The function receives a list of values that can either be positive or negative. All the negative values will be multiplied by the learned alpha value so they become close to zero. [7]

$$PReLU(x) = \begin{cases} \alpha x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

where α is a **learned array** with the same shape as x.

GELU GELU, or Gaussian Error Linear unit, is a non-linear variant of ReLU introduced in 2016 by Dan Hendrycks and Kevin Gimpel in the homonym paper [1]. It's defined as

$$GELU(x) = x\Phi(x)$$

where $\Phi(X)$ is the standard Gaussian cumulative distribution function.

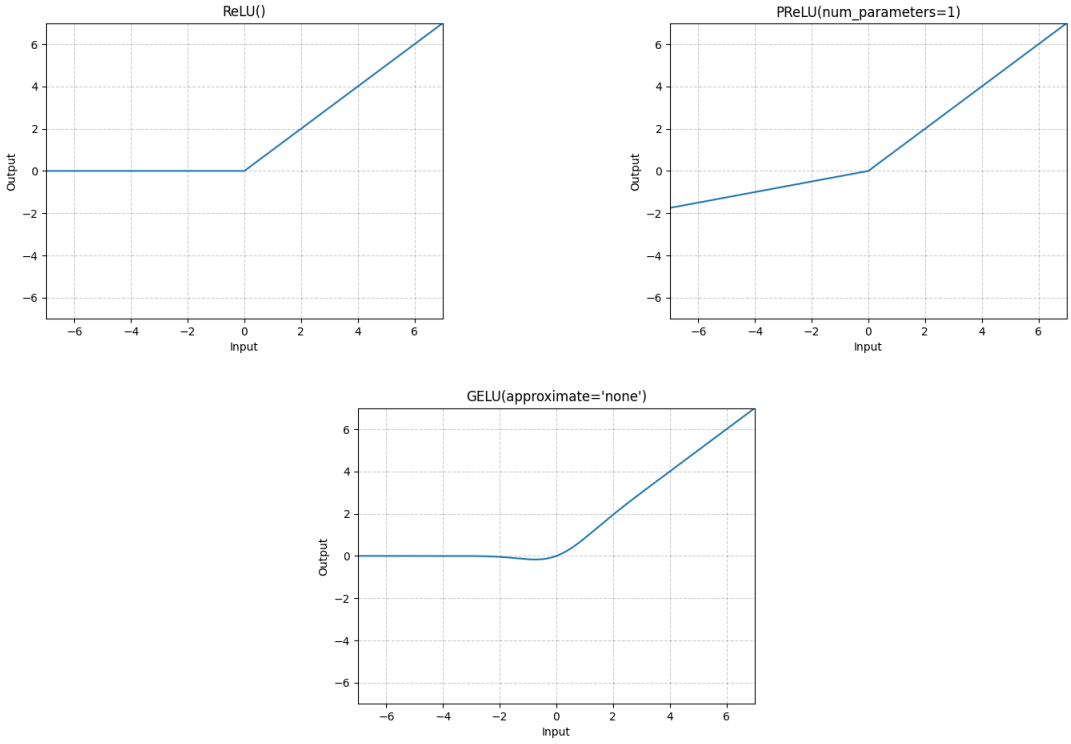


Fig. 1: Plots of the used ReLU variants: ReLU, PReLU and GELU

Variants of the classic ReLU help with the "dying ReLU problem" [4], in which neurons stop learning and become inactive during training, causing metrics to plummet.

1.3 Optimizers

An optimizer is an algorithm used to adjust the parameters of the model during the training process. The primary goal of optimization is to minimize the error or loss function, allowing the neural network to make accurate predictions on new, unseen data. The optimizer is responsible for updating the model parameters based on the difference between these predictions and the actual target values. In this project, we focus on the Adam optimizer with two different variants: AdamW and NAdam.

Adam Optimizer The Adam optimizer, short for Adaptive Moment Estimation, is an optimization algorithm widely used in training neural networks. It was introduced by D. P. Kingma and J. Ba in their 2014 paper "Adam: A Method for Stochastic Optimization." [2]

– **Key Idea**
The key idea behind Adam is to maintain two moving averages for each parameter: the first moment (mean) of the gradients and the second moment (uncentered variance) of the gradients. These moving averages are then used to adjust the learning rates for each parameter adaptively.
– **Algorithm Steps**
  • Given Parameters:
    ∗ Learning rate α
    ∗ Exponential decay rates for the moment estimates β_1 for the first moment and β_2 for the second moment
    ∗ Small constant to prevent division by zero ϵ
  • Algorithm Steps:
    1. Initialize parameters and moving averages.
    2. Compute the gradient of the loss with respect to the parameters.
    3. Update the moving averages of the first and second moments.
    4. Compute the bias-corrected estimates of the first and second moments.
    5. Update the parameters using the corrected estimates and the learning rate.
– **Mathematical Formulas**
  • First Moment Estimate
$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t \tag{1}$$
  • Second Moment Estimate
$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2 \tag{2}$$
  • Bias-Corrected First Moment Estimate
$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{3}$$
  • Bias-Corrected Second Moment Estimate
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{4}$$
  • Parameter Update
$$\theta_t = \theta_{t-1} - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{5}$$

NAdam NAdam, short for Nesterov-accelerated Adaptive Moment Estimation, is an extension of the Adam (Adaptive Moment Estimation) optimizer [6]. NAdam combines the ideas of Nesterov accelerated gradient (NAG) and Adam.

– **Nesterov Accelerated Gradient**
Nesterov accelerated gradient is a method that enables faster convergence by adjusting the momentum term in the gradient descent algorithm.
– **Adam vs NAdam**
The key difference between Adam and NAdam lies in the way the momentum term is calculated:
  • In Adam, the momentum term is calculated using exponentially decaying averages of past gradients and squared gradients
  • In NAdam, the Nesterov Momentum is used.
– **Nesterov Momentum**
In the context of gradient descent optimization, the Nesterov Momentum update formula for updating the parameters w at each iteration t can be expressed as follows:

$$\begin{aligned} v_t &= \mu v_{t-1} - \eta \nabla J\left(w_{t-1} - \mu v_{t-1}\right) \\ w_t &= w_{t-1} + v_t \end{aligned} \tag{6}$$

where:
  • η be the learning rate
  • μ be the momentum parameter
  • $\nabla J\left(w_t\right)$ be the gradient of the loss function with respect to the parameters evaluated at the current position w_t
  • v_t be the momentum term at iteration t

AdamW AdamW is an extension of the Adam optimizer that includes a weight decay term to address potential overfitting in neural network training. The AdamW optimizer was introduced by I. Loshchilov and F. Hutter in their 2017 paper "Fixing Weight Decay Regularization in Adam." [3]

- **Key Features: Weight Decay**
  AdamW includes a weight decay term, which is added directly to the weight update step during training. The weight decay term penalizes large weights, helping to prevent overfitting by encouraging the model to use simpler, more regularized weight values.
- **Mathematical Formulas**
  The update step in AdamW is modified compared to standard Adam. The parameter update formula is as follows:

$$\theta_t = \theta_{t-1} - \alpha \cdot \frac{\hat{m}t}{\sqrt{\hat{v}_t} + \epsilon} - \alpha \cdot \text{WeightDecay} \cdot \theta t - 1 \tag{7}$$

  where:
  - θ_t is the parameter at time step t
  - α is the learning rate
  - $\hat{m}_t$ and $\hat{v}_t$ are the bias-corrected estimates of the first and second moments of the gradients, respectively.
  - ϵ is a small constant to prevent division by zero.
  - $WeightDecay$ is the weight decay hyperparameter.

1.4 Evaluation Metrics

Metrics help capture a business goal into the quantitative target and also Evaluation Metrics help organize machine learning team effort towards that target. Evaluation Metrics are also useful to quantify the gap between:

- **Desired performance and baseline (estimate effort initially)**
  You can use it to see the gap between the desired performance and the baseline; a baseline is generally the first attempt that you come up with some kind of a simple model that gives you a sense of the difficulty of the overall project that you're starting with.
- **Desired performance and current performance**
  You can also measure the gap between the desired performance and the current performance which kind of gives you a sense of how much more progress is left to be made.
- **Measure progress over time**
  It's also useful to keep track of how our performance is improving over time. It gives you a sense of how much progress you have made towards the end goal.

So Evaluation metrics are useful to measure how well your model is doing in terms of your desired performance level, in essence, it's useful to quantify the gap.

Accuracy Accuracy is basically what is the fraction of all the examples that we got right, no matter what class they were that we predicted.

$$\text{Accuracy}(y, \hat{y}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} 1\left(\hat{y}_i = y_i\right) \tag{8}$$

Precision Precision means among the examples that we predicted to be positive what fraction of them were positive.

$$\text{Precision} = \frac{\text{TruePositive}}{\text{TruePositive} + \text{FalsePositive}} \tag{9}$$

Recall (Sensitivity) What recall measures is if we were to use this classifier in deployment what fraction of all the actual positives are we going to recover?

$$\text{Recall} = \frac{\text{TruePositive}}{\text{TruePositive} + \text{FalseNegative}} \tag{10}$$

The term sensitivity comes from epidemiology and in essence how sensitive this test is for detecting a certain condition.

F1 score F1-score is the harmonic mean of precision and recall:

$$\begin{aligned} \frac{1}{\text{FScore}} &= \frac{1}{\text{Precision}} + \frac{1}{\text{Recall}} \\ \text{FScore} &= \frac{\text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}} \end{aligned} \tag{11}$$

2 Overfitting and Underfitting

2.1 Underfitting

Underfitting typically occurs when the model is too simple to capture the complexities of the data or when the training process is not sufficient to enable the model to learn the underlying patterns. To address underfitting, we can increase the size and diversity of the training dataset to provide the model with more examples to learn from the data augmentation technique.

2.2 Ovefitting

Overfitting is when a machine learning model learns to perform extremely well on the training data but fails to generalize its predictions accurately to unseen data or test data.

Overfitting occurs when a model learns not only the underlying patterns in the training data but also the noise or random fluctuations present in the data. To address overfitting in deep learning, regularization techniques such as early stopping and data augmentation are used.

2.3 Early Stopping

Early Stopping involves monitoring the performance of a model on a separate validation dataset during training and stopping the training process once the performance on the validation dataset starts to degrade. Here's how early stopping works:

1. **Training Phase**
   During the training phase, the model is trained on the training dataset using an optimization algorithm.
2. **Validation Phase**
   After each epoch, the model's performance is evaluated on a separate validation dataset that the model hasn't seen during training.
3. **Monitoring Performance**
   The performance metric (e.g., accuracy, loss) on the validation dataset is monitored. If the performance metric starts to degrade or fails to improve for a certain number of epochs, it indicates that the model is beginning to overfit the training data.
4. **Stopping Criteria**
   Once the performance on the validation dataset fails to improve or begins to degrade for a specified number of consecutive epochs (known as the patience parameter), the training process is stopped early

2.4 Reducing Learning Rate on Plateau

We used ReduceLROnPlateau, which is a scheduling technique that decreases the learning rate when the specified metric stops improving for longer than the patience number allows. Thus, the learning rate is kept the same as long as it improves the metric quantity, but the learning rate is reduced when the results run into stagnation, after a defined amount of epochs. This is applied before EarlyStopping to avoid abrupt interruption. The learning rate gets modified with

$$New\_LR = factor * Old\_LR$$

with **factor** being a real number.

2.5 Data Augmentation

Data augmentation is a technique used in deep learning to artificially increase the size and diversity of a training dataset by applying various transformations to the existing data samples. The goal of data augmentation is to improve the generalization and robustness of machine learning models by exposing them to a wider variety of data during training [8].

Techniques Data augmentation techniques commonly include:

- **Rotation:** Rotating images by a certain degree (e.g., 90 degrees, 180 degrees) to simulate different viewpoints.
- **Scaling:** Resizing images to different dimensions, either larger or smaller, to simulate different zoom levels.
- **Shearing:** Applying a shearing transformation to skew the images along one of the axes.
- **Zooming:** Zooming in or out of images to simulate different levels of magnification.
- **Flipping:** Mirroring images horizontally or vertically to simulate reflections.

3 Loss Function

A loss function is a measure that quantifies the difference between the predicted values of a model and the actual ground truth labels in the training dataset.

3.1 Goal

The primary goal during training in deep learning is to minimize the value of the loss function. By iteratively updating the model's parameters using optimization, the model aims to reduce the discrepancy between its predictions and the ground truth labels, ultimately improving its performance on unseen data.

3.2 Categorical Cross-Entropy

Categorical cross-entropy loss is a commonly used loss function in deep learning, particularly for multi-class classification tasks. It measures the discrepancy between the true probability distribution of class labels and the predicted probability distribution produced by the model.

How categorical cross-entropy loss works:

1. Input: In a multi-class classification task, the model predicts the probability distribution over all possible classes for each input sample. The predicted probabilities are often obtained by applying the softmax activation function to the output layer of the neural network.
2. True Labels: The true labels for each input sample are represented as one-hot encoded vectors, where only the index corresponding to the true class label is set to 1, and all other indices are set to 0.
3. Calculation: Categorical cross-entropy loss is calculated by computing the cross-entropy between the true probability distribution (represented by the one-hot encoded vectors) and the predicted probability distribution outputted by the model.

Formula Mathematically, the categorical cross-entropy loss for a single training example can be expressed as:

$$L(y, \hat{y}) = -\sum_{i=1}^{N} y_i \cdot \log(\hat{y}_i) \tag{12}$$

where:

- y represents the true probability distribution (one-hot encoded vector) for the class labels.
- $\hat{y}$ represents the predicted probability distribution produced by the model.
- N is the number of classes.
- y_i and $\hat{y}_i$ are the true and predicted probabilities for class i, respectively.

4 Dataset

The dataset 7000 Labeled Pokémons by Lance Zhang was taken from Kaggle and adapted to the scope of the project, as some 1st Pokémons were missing. In particular, there are around 25 to 50 images for each Pokemon, all with the Pokemon in the centre. Most (if not all) images have relatively high quality (correct labels, centred). To train the models, we split the dataset into 8:2 for train and evaluation. So we have:

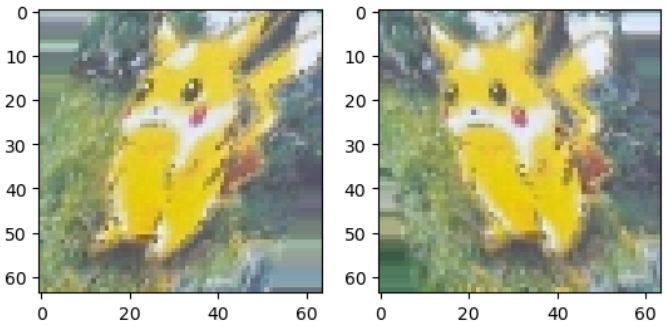
$$\begin{aligned} \text{TrainingSet} &\approx 0.8 \times 0.8 \times \underbrace{151}_{\text{Number of Pokémon}} \times [25, 50] \approx [2416, 4832] \\ \text{ValidationSet} &\approx 0.8 \times 0.2 \times \underbrace{151}_{\text{Number of Pokémon}} \times [25, 50] \approx [604, 1208] \end{aligned} \tag{13}$$

4.1 Data Augmentation

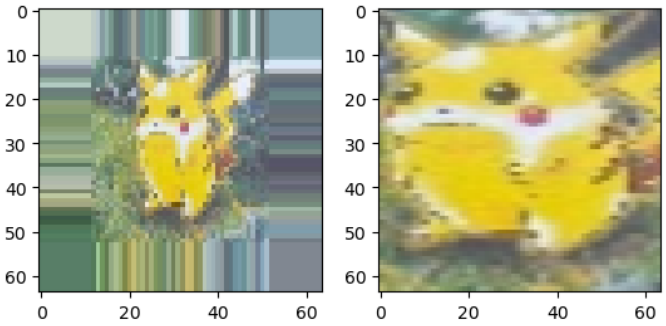
Since the dataset is small, we applied a series of transformations to the input image to create a variation thereof, which can help to increase the diversity of the data and prevent overfitting:

– **Rescaling Factor**

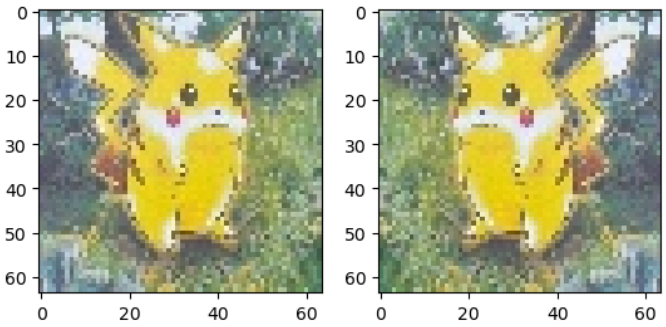
– **Shear Angle**



– **Zooming**



– **Randomly flip inputs horizontally**



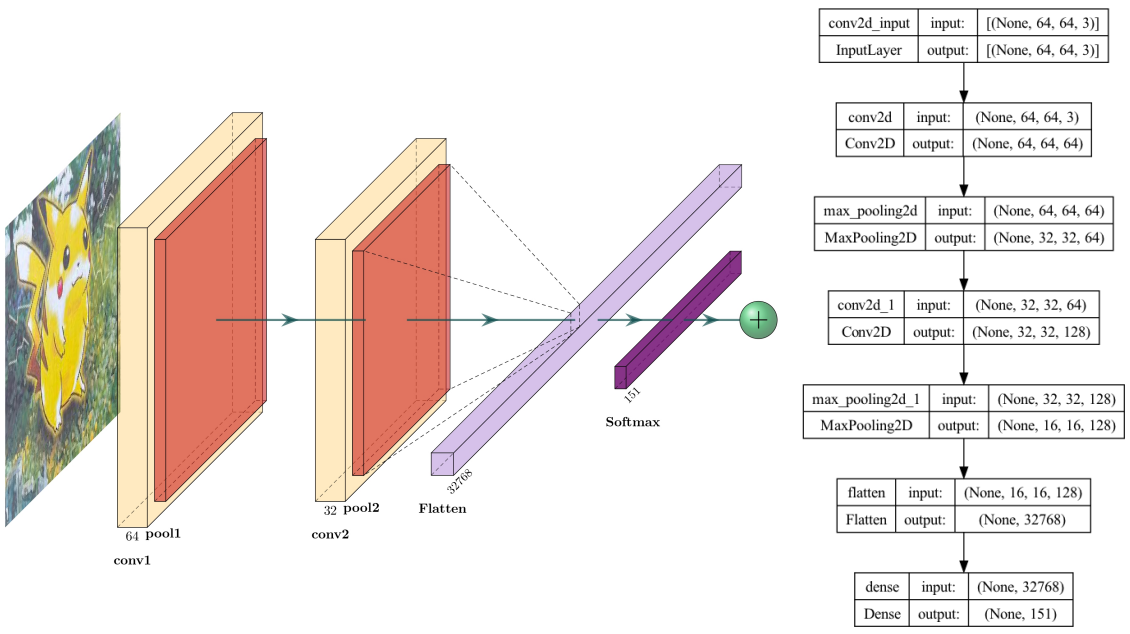
5 Models

The nine models were implemented using TensorFlow, Keras and Kaggle as a training facility with a Nvidia GPU100 as the accelerator. They share a general architecture form to focus on the metrics better. Early stopping was implemented in the training stage to prevent data fitting problems. If the current epoch's loss function is greater than the previous epoch's, training is stopped and the model is saved, generating the .h5 file.

```
# ----- Defining general model architecture -----

def general_model(image_size, num_classes):
    clear_session()
    reset_uids()

    classifier = Sequential()
    classifier.add(Conv2D(64, (5, 5), input_shape=image_size,
                   padding='same'))

    classifier.add('your_chosen_act_fcn')
    classifier.add(MaxPooling2D(pool_size = (2, 2)))
    classifier.add(Conv2D(128, (3, 3),
                   padding='same'))

    classifier.add('your_chosen_act_fcn')
    classifier.add(MaxPooling2D(pool_size = (2, 2)))
    classifier.add(Flatten())
    classifier.add(Dense(num_classes, activation = 'softmax'))

    classifier.compile(optimizer = 'your_chosen_optimizer',
                       loss='categorical_crossentropy',
                       metrics=['acc', f1_m, precision_m, recall_m])

    return classifier
```

As the snippet above mentions, each model had this structure with different combinations of activation functions and optimizers, specified in the subsections:

- 35 epochs
- 0.002 starting learning rate, reduced by the scheduler to a minimum of 0.001 if needed, with factor = 0.8
- 32 batch size
- Two convolutional layers:
  - 64 filters, (5, 5) stride
  - 128 filters (3, 3) stride
- Two 2 x 2 max pooling layers
- A final Dense Softmax layer to classify pictures



Categorical Cross-Entropy The choice of a loss function depends on the nature of the machine learning task being addressed. Different tasks typically require different loss functions tailored to the specific characteristics of the task. In this project, we used the Categorical Cross-Entropy Loss because it is particularly useful when the output of the model is a probability distribution over multiple classes.

A complete and cohesive summary with metric graphs can be found on the pages below or on the GitHub repository linked on the 1st page. The GitHub repository also has a run made **without** the ReduceLROnPlateau scheduler and is not included in this paper.

5.1 Table Summary & Considerations

ReLU-based models

Opt Type	Train Loss	Train Acc	Train Prec	Train Rec	Train F1
Adam	0.1982	0.9432	0.9566	0.9327	0.9443
Nadam	0.2727	0.9227	0.9402	0.9091	0.9242
AdamW	0.1493	0.9608	0.9670	0.9559	0.9613

(14)

Opt Type	Valid Loss	Valid Acc	Valid Prec	Valid Rec	Valid F1
Adam	0.1615	0.9605	0.9731	0.9523	0.9624
Nadam	0.2169	0.9413	0.9640	0.9243	0.9433
AdamW	0.1695	0.9626	0.9654	0.9609	0.9631

(15)

PReLU-based models

Opt Type	Train Loss	Train Acc	Train Prec	Train Rec	Train F1
Adam	0.1194	0.9687	0.9701	0.9662	0.9682
Nadam	0.1593	0.9564	0.9612	0.9539	0.9575
AdamW	0.0981	0.9727	0.9752	0.9699	0.9725

(16)

Opt Type	Valid Loss	Valid Acc	Valid Prec	Valid Rec	Valid F1
Adam	0.0892	0.9784	0.9785	0.9729	0.9756
Nadam	0.1488	0.9646	0.9702	0.9630	0.9665
AdamW	0.1283	0.9717	0.9748	0.9683	0.9715

(17)

GeLU-based models

Opt Type	Train Loss	Train Acc	Train Prec	Train Rec	Train F1
Adam	0.2009	0.9618	0.9645	0.9609	0.9627
Nadam	0.2608	0.9547	0.9564	0.9538	0.9551
AdamW	0.2787	0.9531	0.9545	0.9520	0.9532

(18)

Opt Type	Valid Loss	Valid Acc	Valid Prec	Valid Rec	Valid F1
Adam	0.2279	0.9609	0.9645	0.9605	0.9625
Nadam	0.3361	0.9484	0.9489	0.9453	0.9471
AdamW	0.3716	0.9434	0.9460	0.9441	0.9450

(19)

We found that the weight decay term of AdamW affects the PReLU by fixing the otherwise variable α parameter, essentially making it behave more like a standard ReLU if fixed to 0, or a LeakyReLU for every other value.

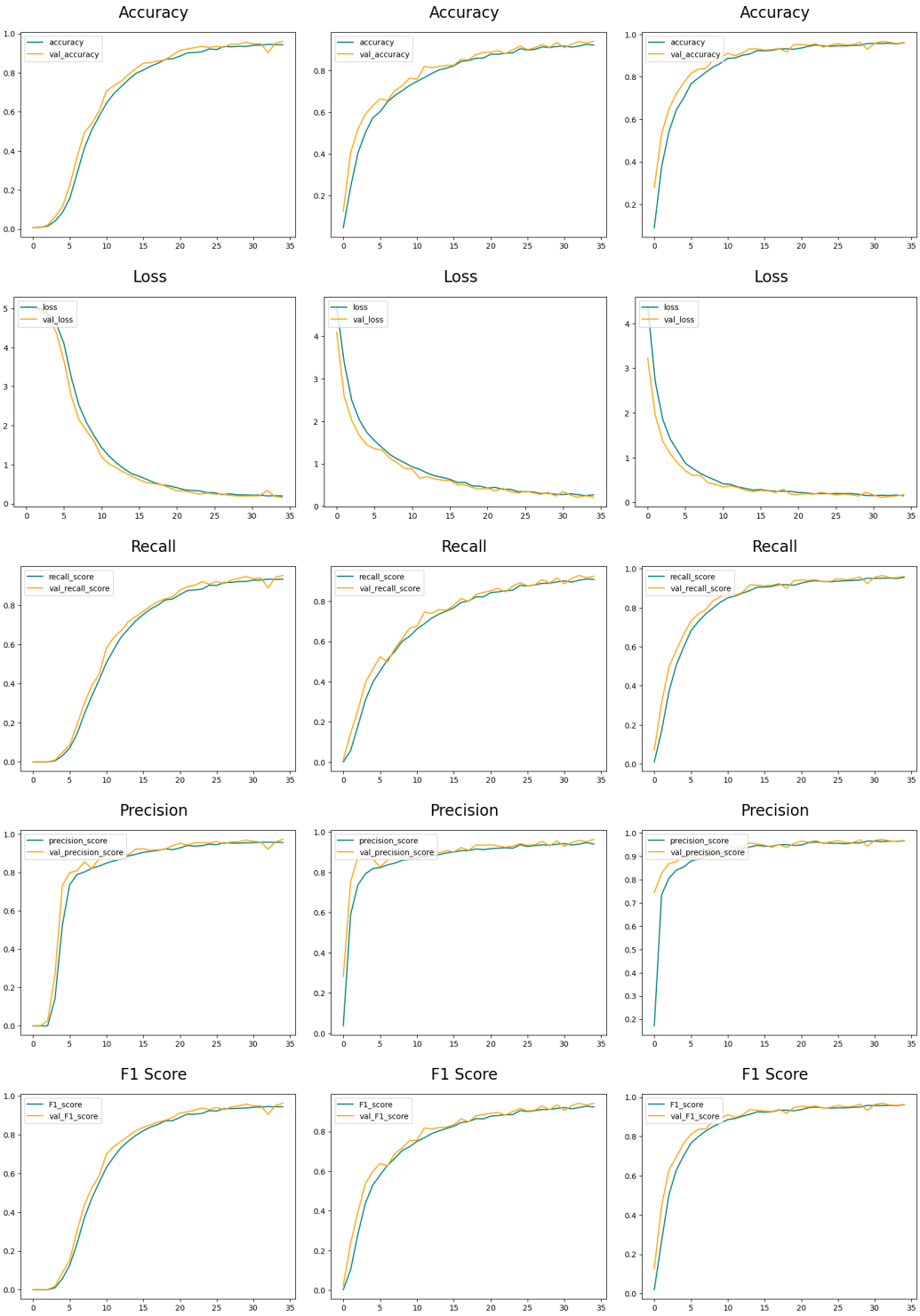
With our dataset and under the conditions specified above, GELU performed rather poorly on this task and early stopped at epoch 32 when coupled with Adam.

The NAdam optimizer performed better than Adam and AdamW across the board. This is because the Nesterov Momentum is proven to achieve convergence faster, ultimately smoothing any abrupt changes in metrics. This is noticeable with GELU more than other models.

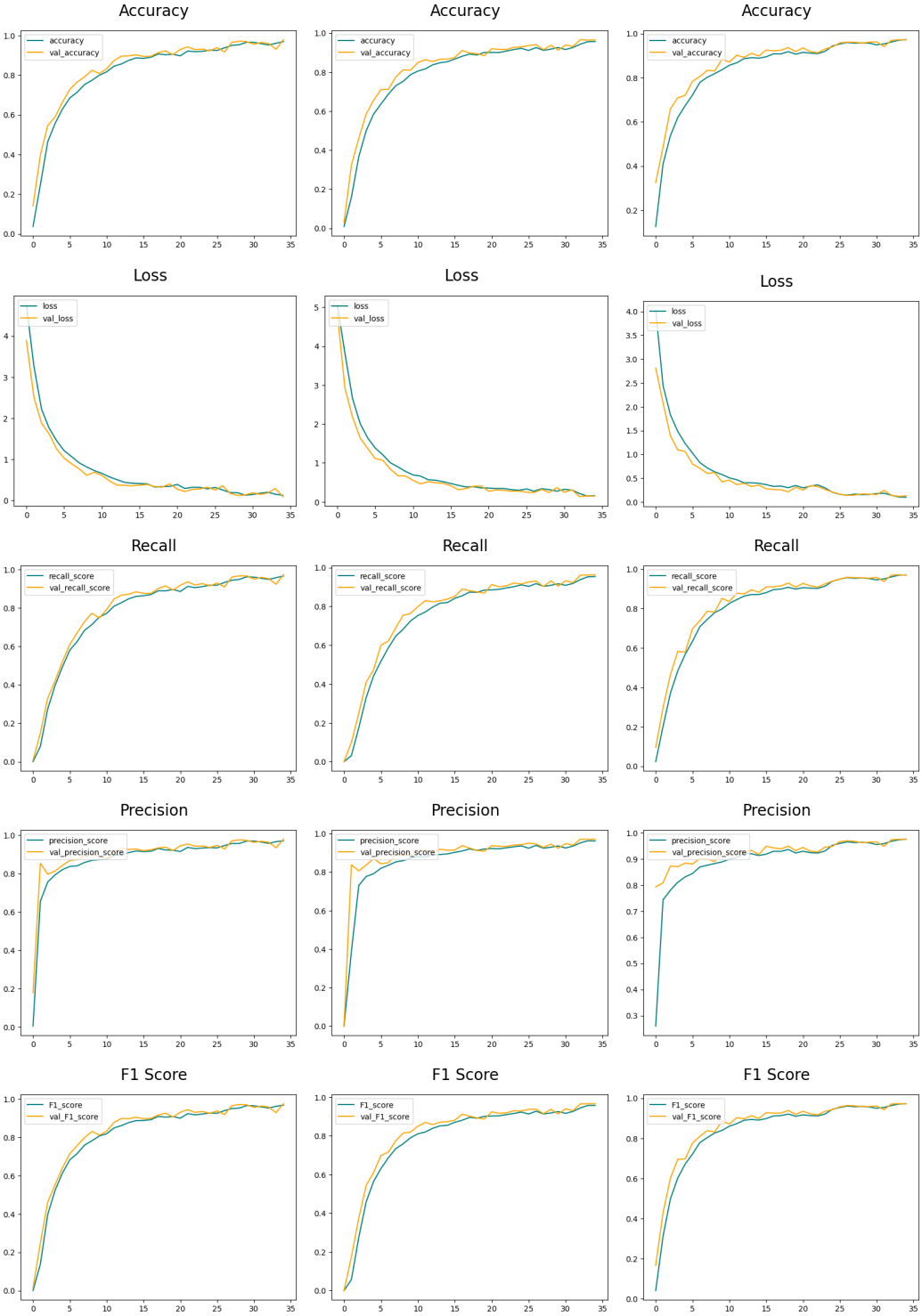
Graphs for the models can be found in the GitHub repository mentioned on the first page of this paper.

6 Models Results

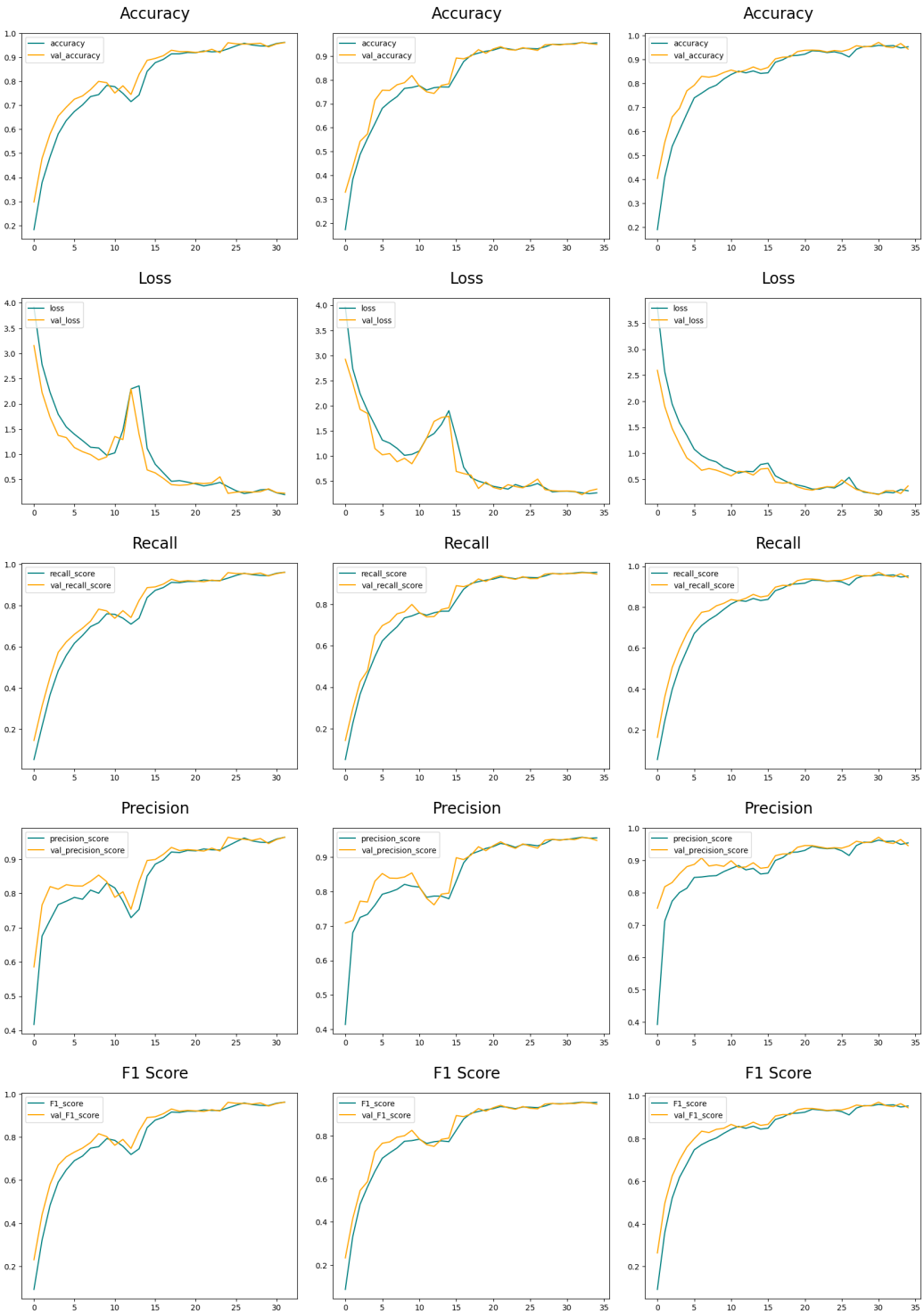
6.1 Model 1 – ReLU based



6.2 Model 2 – PReLU based



6.3 Model 3 – GeLU based



References

[1] Dan Hendrycks and Kevin Gimpel. "Bridging Nonlinearities and Stochastic Regularizers with Gaussian Error Linear Units". In: *CoRR* abs/1606.08415 (2016). arXiv: 1606.08415. URL: http://arxiv.org/abs/1606.08415.

[2] Diederik Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *International Conference on Learning Representations (ICLR)*. San Diega, CA, USA, 2015.

[3] Ilya Loshchilov and Frank Hutter. *Fixing Weight Decay Regularization in Adam*. 2018. URL: https://openreview.net/forum?id=rk6qdGgCZ.

[4] Lu Lu. "Dying ReLU and Initialization: Theory and Numerical Examples". In: *Communications in Computational Physics* 28.5 (June 2020), pp. 1671–1706. ISSN: 1991-7120. DOI: 10.4208/cicp.oa-2020-0165. URL: http://dx.doi.org/10.4208/cicp.OA-2020-0165.

[5] Keiron O'Shea and Ryan Nash. "An Introduction to Convolutional Neural Networks". In: *CoRR* abs/1511.08458 (2015). arXiv: 1511.08458. URL: http://arxiv.org/abs/1511.08458.

[6] Sebastian Ruder. "An overview of gradient descent optimization algorithms". In: *CoRR* abs/1609.04747 (2016). arXiv: 1609.04747. URL: http://arxiv.org/abs/1609.04747.

[7] Bing Xu et al. "Empirical Evaluation of Rectified Activations in Convolutional Network". In: *CoRR* abs/1505.00853 (2015). arXiv: 1505.00853. URL: http://arxiv.org/abs/1505.00853.

[8] Suorong Yang et al. *Image Data Augmentation for Deep Learning: A Survey*. 2023. arXiv: 2204.08610 [cs.CV].