



UNIVERSITÀ DI PISA

Artificial Intelligence and Data Engineering
Symbolic and Evolutionary Artificial Intelligence

Project documentation

TEAM MEMBERS:

Fabiano Pilia
Elisa De Filomeno

Contents

| | | |
|----------|---|-----------|
| 1 | Introduction | 2 |
| 1.1 | State-of-Art | 2 |
| 1.2 | Posit Numbers | 2 |
| 1.3 | Tensorflow | 4 |
| 2 | Installing Tensorflow | 5 |
| 3 | Experiments | 7 |
| 3.1 | MNIST | 7 |
| 3.1.1 | Download and prepare the dataset | 8 |
| 3.1.2 | First model: DNN | 8 |
| 3.1.3 | Using different activation functions | 12 |
| 3.1.4 | Second model: CNN | 14 |
| 3.1.5 | Third model: CNN with more layers, Relu and Random Uni- form Initialiser | 20 |
| 3.1.6 | Fourth model: Adding dropout | 23 |
| 3.1.7 | Analysing the dropout layer | 27 |
| 3.1.8 | Fourth model with the new Dropout Layer | 31 |
| 3.1.9 | LeNet-5 | 33 |
| 3.1.10 | MNIST Conclusion and summary | 38 |
| 3.2 | CIFAR-10 | 39 |
| 3.2.1 | Download and prepare the dataset | 39 |
| 3.2.2 | First CNN model | 40 |
| 3.2.3 | CIFAR10 Conclusion and summary | 51 |
| 3.3 | Fine-tuning | 51 |
| 4 | Conclusion and Future Work | 53 |

Chapter 1

Introduction

This project aims to study the extension of Tensorflow with posit support using cpp-Posit both for training and inference. In the following experiments we will compare the results of utilizing float32 and posit160 representations in Deep Neural Networks (DNNs) and Convolutional Neural Networks (CNNs). Our goal is to examine the performance differences when using posits as an alternative to floating-point numbers. We will also verify the proper functionality of neural networks defined using the customized TensorFlow framework developed to support posits. This framework can be found at the following [Link](#), during the experiments we encountered some problems when working with posits, we tried to modify the source code of the extension of TensorFlow to solve them.

1.1 State-of-Art

A strategic aspect of high-performance computing applications is the efficient representation of the information, typically composed of real numbers. The standard for number representation is the IEEE floating point, however, there have been some problems with this format. Floats are not portable across architectures and the hardware (the FPU, floating point unit) is very complex and resource costing. Moreover the IEEE 754 standard is never fully implemented in any architecture and there may be exceptions (like underflow, overflow) that handling is troublesome.

To solve all these problems a new number representation format has been created and studied: the Posit number representation. This new data type is designed to replace the standard floating point.

1.2 Posit Numbers

Posit numbers were introduced and created by John L. Gustafson in 2017. Unlike floating-point numbers, which use the three fields sign (1 bit), mantissa (23 bits), and exponent (8 bits) to represent a wide range of values with varying precision, posits use a different encoding scheme. Posit format uses four fields to represent numbers:

a sign (1bit), a regime (variable length), an exponent e (at most es bits), and a fraction f (variable length). A posit number $\langle nbits, esbits \rangle$ has two configuration parameters: $nbits$ which is the total number of bits, and $esbits$ which is the maximum number of bits of the exponent. In the following picture we can see an illustration of a posit $\langle 32, 11 \rangle$ data type:

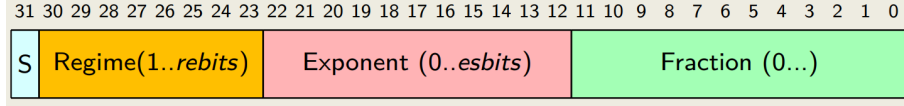


Figure 1.1: Illustration of a posit $\langle 32, 11 \rangle$ data type

The regime is responsible for determining the scale of the number and has a variable length that spans from 1 to $nbits - 1$. The exponent specifies the power of 2 to which the regime is applied. The fraction bits store the fractional part of the number. In general, a real number represented with high accuracy using the posit representation takes fewer bits than with the floating-point representation.

Given a posit X encoding the real number r can be obtained with the following formula:

$$r = \begin{cases} 0, & \text{if } X = 0 \\ \text{NaN}, & \text{if } X = -2^{(nbits-1)} \\ \text{sign}(X) \times useed^k \cdot 2^e \cdot (1 + f), & \text{otherwise} \end{cases}$$

Figure 1.2: Real number r represented by a posit encoding

In particular:

- $useed$ is given by 2^{esbits}
- $\text{sign}(X)$ is given by the first bit of the posit representation
- k is given by the following formula:

$$k = \begin{cases} -l, & \text{if } b = 0 \\ l - 1, & \text{otherwise} \end{cases}$$

Figure 1.3: k encoding

In the above formula:

- l represents the regime length and it is the number of subsequent identical bits in the regime
- b is the value of the first opposite bit after the previous sequence of identical bits.

Posit numbers can be projected on a circle, the *posit ring*:

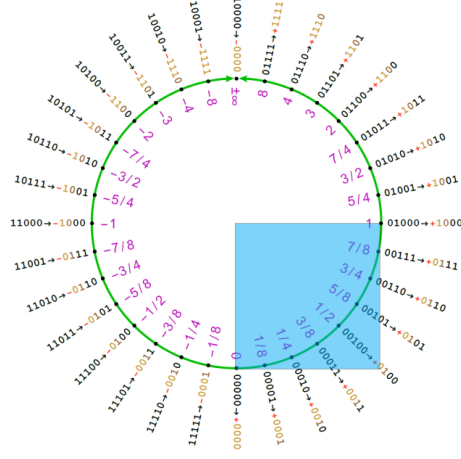


Figure 1.4: Posit ring

The interval $[0,1]$ provides the highest level of accuracy and the numbers represented in this region have the smallest resolution, which is defined as the difference between two consecutive numbers. From analyzing the cumulative density function of the weight distribution for the dataset *CIFAR-10* it was revealed that the majority of weights should fall within the range of -1 to 1. Consequently, the weights will predominantly reside in the region where the posit representation offers the highest decimal accuracy.

1.3 Tensorflow

TensorFlow is a free and open-source software library developed by the *Google Brain* team for internal Google usage. It provides a comprehensive ecosystem of tools, libraries, and resources for building and deploying machine learning models.

TensorFlow is widely used for a variety of tasks and in particular for training and inference of deep neural networks.

It supports both training and inference of models on various hardware platforms, including CPUs, GPUs, and TPUs (Google's Tensor Processing Units) and can be used with many programming languages, including Python, JavaScript, C++, and Java.

Chapter 2

Installing Tensorflow

For this project we've used our personal computer with Windows 11 operating system, in this chapter we will explain how we prepared the environment to develop the project.

Firstly we installed Docker Desktop, then We used Windows Subsystem for Linux (WSL2) to download and configure a Ubuntu 22.04 LTS terminal environment on our personal machines. At this point we installed Visual Studio Code in order to code and debug inside the remote Docker container and connected to our Linux distribution.

Inside the container we've cloned the following git repository:

<https://github.com/federicorossifr/tensorposit.git>

We've downloaded the *cpp_posit* zip from a past project in order to continue their work. Then we've performed the following steps:

1. Copy `cppposit_private` in `home/USER/cppWorkspace`
2. Change in `tensorposit/env.sh` from `federico/` to `USER/` to obtain:

```
-v/home/fabiano/cppWorkspace/cppposit_private:/cppposit_private
```

3. Remove the `-config=opt` from the `tensorposit/tensorflow/build.sh` script and add the limitation of the resources, obtaining the following command: We build tensorflow using the command:

```
bazel build --local_ram_resources=7000 --local_cpu_resources=4  
//tensorflow/tools/pip_package:build_pip_package
```

```
./bazel-bin/tensorflow/tools/pip_package/build_pip_package /mnt
```

4. Start a container using running **env.sh**
5. From inside the container move to the tensorflow folder and run **build.sh** and wait for the completion: it requires a lot of time, approximately a day.

6. The previous step generates a .whl file inside the mnt folder of the container; this file must be moved from the container to outside in the cppWorkspace folder, to do it we've used the following command:

```
docker cp CONTAINER-ID:mnt/TENSORFLOW-FILE.whl ~/cppWorkspace
```

Tensorflow 2.6.0 requires python 3.8 to run, so it must be the default python version for Ubuntu, but since our version of Ubuntu is the 22.04, then the python version was 3.10, so we've uninstalled everything and installed python 3.8 and pip for python 3.8, otherwise tensorflow doesn't work. By testing tensorflow we obtained several problems related to dependencies solved in the following way:

- tensorflow 2.6.0 requires numpy 1.19.2, pandas 1.3.5 works.
- python3.8 -m pip install scikit-learn
- python3.8 -m pip uninstall scipy
- python3.8 -m pip install scipy==1.8
- install matplotlib 3.6
- uninstall PIL and Pillow
- install pillow

At this point we were able to import and use tensorflow to perform the experiments.

Chapter 3

Experiments

In this chapter we'll present the experiments and the results obtained with the posits; more specifically, we'll use the same model with the same initial weights and we'll train it with both the float32 and the posit160. We'll discuss the problems we encountered, how we solved them, and the ones we didn't solve. We've started from the results of another project trying to solve the problems in the following figure.

| Layer | Does it work with the posit? |
|--------------------|------------------------------|
| Dense | Yes |
| Flatten | Yes |
| Dropout | No |
| Conv1D | Yes |
| MaxPooling | Yes |
| Conv2D | No |
| BatchNormalization | No |
| LSTM | Yes |

Figure 3.1: Previous project results

3.1 MNIST

The **MNIST dataset** is a widely used benchmark dataset in computer vision and machine learning. It consists of a collection of **70,000** grey-scale images of hand-written digits (0-9) with a resolution of *28x28 pixels*. These images are divided into a **training set** of *60,000* examples and a **test set** of *10,000* examples. The dataset is **labelled**, meaning that each image has an associated ground truth label indicating the digit it represents.

It is a standard dataset for evaluating and comparing the performance of different machine learning algorithms, particularly in the digit classification task. Many researchers and educators use MNIST as a starting point for developing and testing new models and techniques in computer vision.

The problem to be solved with this dataset is to predict which digit is shown in each image.

The image below shows some examples from the MNIST dataset:



Figure 3.2: Images of handwritten digits present in the MNIST dataset

3.1.1 Download and prepare the dataset

In order to use the *MNIST dataset*, we need to retrieve it and pre-process it so that it can be used by our model. The dataset is available in the `tensorflow.keras.datasets.mnist` module and using the `load_data()` we've obtained two tuples of NumPy arrays: `(x_train, y_train)`, `(x_test, y_test)`. The first element in the tuple is the image, the second is its label.

The images in both sets are in the range `[0,255]`, so we've rescaled them to the range `[0,1]` to avoid computing with large numbers and to make better use of the posit representation.

To obtain for all the experiments the same random initialization of the weights we've used a seed.

3.1.2 First model: DNN

The first model used is a DNN consisting of a flat layer to transform the `(28,28,1)` input into a `(28*28,)` input, a dense hidden layer with 128 units equipped with a **Relu** activation function and a final layer, since we are performing a 10-class classification, is a dense layer with 10 units with a softmax activation function. The model is the following:

```
model = models.Sequential()
model.add(layers.Flatten(input_shape=(28, 28, 1)))
model.add(layers.Dense(128, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

Since we're solving a multi-class classification the **loss function** used is the *categorical cross-entropy*; the metric is the *accuracy*.

In the following we can see the compilation and execution settings:

```
model.compile(optimizer='adam',
              loss=k.losses.categorical_crossentropy,
              metrics=['accuracy'])

history = model.fit(train_images,
                   train_labels,
                   epochs=10,
                   batch_size=128,
                   validation_split=0.1)
```

Float32 results

Using the configuration with the **float32** we've obtained the following results during the training phase:

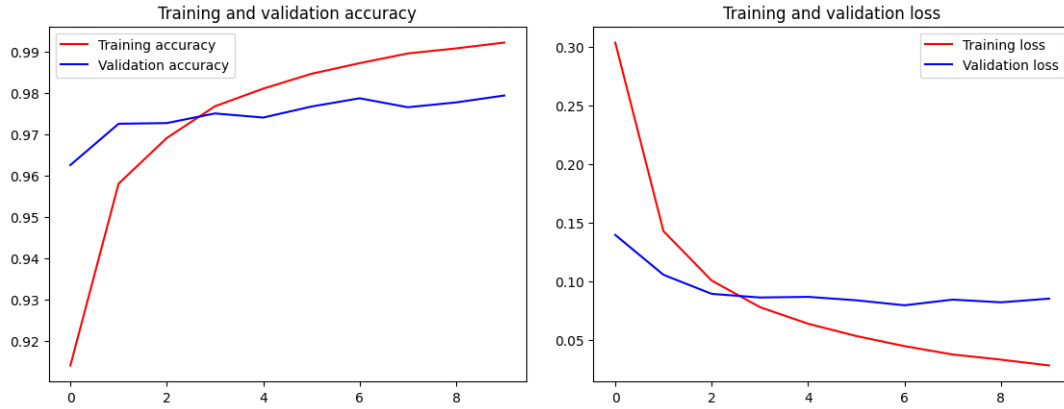


Figure 3.3: Training and validation accuracy and loss plots of the first model trained using the float32

As we can see from figure 3.3 we've obtained good results for both training and validation accuracy, while the actual values achieved are presented in table 3.1.

| | accuracy | loss |
|------------|----------|--------|
| training | 0.9921 | 0.0280 |
| validation | 0.9793 | 0.0850 |
| test | 0.9782 | 0.0863 |

Table 3.1: Results of the training and test of the model

Posit results

In order to use the posits we need to set the default float type:

```
import tensorflow.python.keras as k
k.backend.set_floatx("posit160")
```

During this experiment we've obtained strange results:

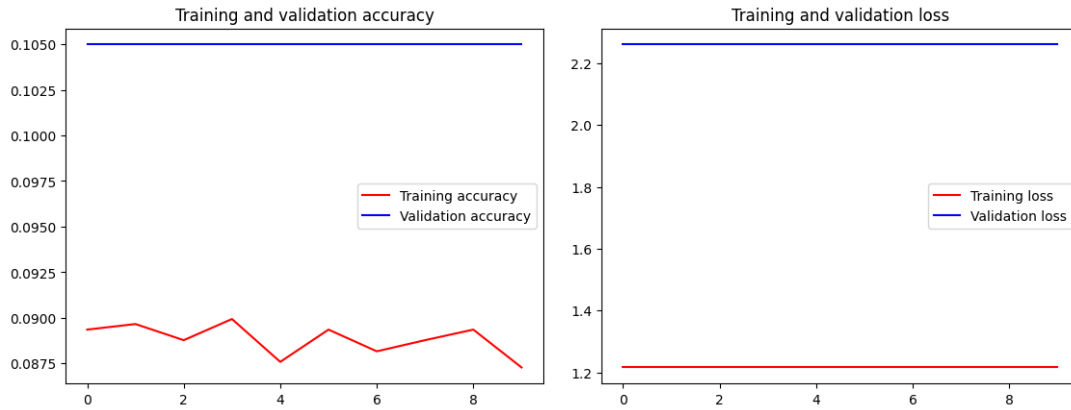


Figure 3.4: Training and validation accuracy and loss plots of the first model trained using the posit160

As we can see the validation accuracy, the validation loss and the training loss remains always equal to their starting values, respectively **0.0894**, **2.2607** and **1.2158**.

This problem means that the **weights are not updated** during the training phase remaining at the same values of weights obtained by random initialization.

We've proceeded to investigate the weights retrieving it from the hidden layer, in figure 3.5:

```
array([[ -0.0804138, -0.0808105, -0.0809021, ..., -0.0810242, -0.0809021,
        -0.0805054],
       [ -0.0801697, -0.079895, -0.079895, ..., -0.0800781, -0.0803528,
        -0.0800781],
       [ -0.0804749, -0.0804138, -0.0810242, ..., -0.0804443, -0.0803528,
        -0.080719],
       ...,
       [ -0.0810242, -0.0808411, -0.0800476, ..., -0.0802612, -0.0801086,
        -0.0809326],
       [ -0.0809021, -0.0802612, -0.0801392, ..., -0.0809326, -0.0803833,
        -0.0809021],
       [ -0.0804138, -0.0798645, -0.0809937, ..., -0.0801697, -0.0802917,
        -0.0805664]], dtype=posit160)
```

Figure 3.5: Weights of the model after the training using the posits

We can see from the figure that the weights are **posit160**, confirming that we're using the correct expected representation, but the most important fact is that all

the weights are negative numbers.

As activation function for the hidden layer units we're using the **ReLU** (figure 3.6), which outputs 0 for values in input *less than or equal to 0*.

This is the problem with our model: since the weights are always negative and they multiply positive numbers, the resulting **weighted sum**, which is the input for a generic hidden unit, is **negative**. Since the weights are updated using a **gradient descent** method and the gradient of the *ReLU* for values less than or equal to 0 is **0**, then the gradient contribution to be added to each weight is **always 0**.

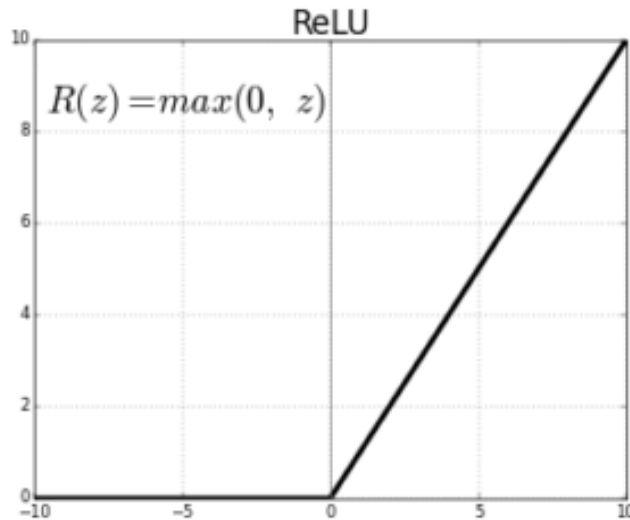


Figure 3.6: The plot of the ReLU activation function

3.1.3 Using different activation functions

To solve the problem we've decided to use the **tanh** activation function instead of the *ReLU*. The *tanh* is defined in the interval **[-1,1]** and generalize the **sign** function making it derivable around 0, see figure 3.7.

We must use it with care, since if we use several hidden layers we may incur in the **vanishing of the gradient**; This problem is due to the fact that when the error is propagated backwards to calculate the weight update, there are several multiplications of numbers between 0 and 1, resulting in a weight update close to 0 causing the stall of the weights, which is in fact the same result we're trying to solve.

The same problem is present if we use the **sigmoid** function, obtaining several values between 0 and 0.25 multiplied by each other, obtaining an irrelevant update of the weight.

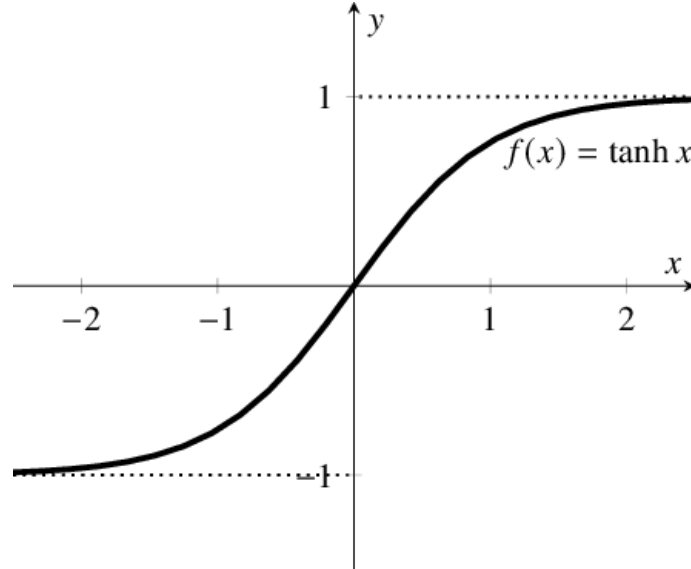


Figure 3.7: The plot of the Tanh activation function

Posit results using the *tanh*

In figure 3.8 we can see the training plots using the **tanh** as activation function.

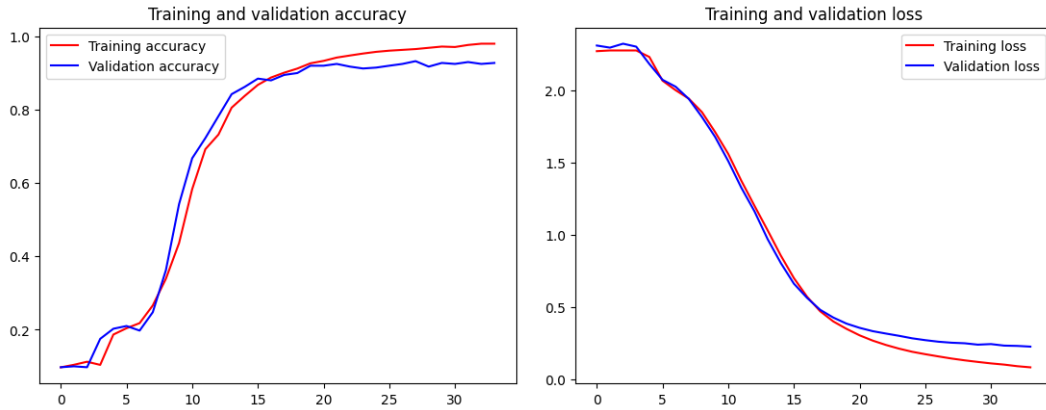


Figure 3.8: Training and validation accuracy and loss plots of the first model trained using the posit160 and tanh as activation function

The results obtained are presented in table 3.2.

| | accuracy | loss |
|------------|----------|--------|
| training | 0.9800 | 0.0804 |
| validation | 0.9275 | 0.2239 |
| test | 0.9199 | 0.2345 |

Table 3.2: Results of the training and test of the model

From the table and the figure we can see that we reach good values for the

training, validation and test loss. The training accuracy increases slowly respect to the float32 case using the Relu, in fact here we've trained the network for *30 epochs* reaching only the 98% of accuracy. Better values of accuracy on the training set can be achieved by training the network for more epochs. To compare it with the float case we've trained the model using the *tanh* also for the posits.

Float32 results using the *tanh*

In figure 3.8 we can see the training plots using the **tanh** as activation function in case of floats.

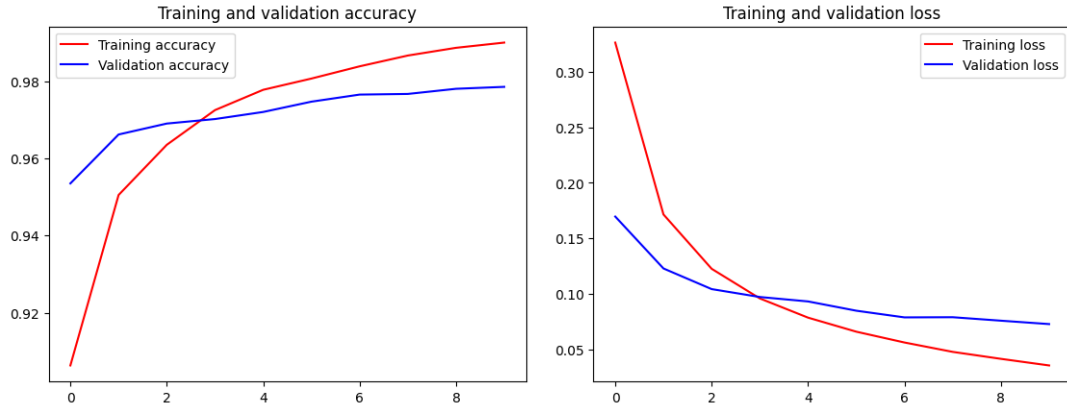


Figure 3.9: Training and validation accuracy and loss plots of the first model trained using the float32 and tanh as activation function

The results obtained are presented in table 3.3.

| | accuracy | loss |
|------------|----------|--------|
| training | 0.9900 | 0.0355 |
| validation | 0.9785 | 0.0728 |
| test | 0.9762 | 0.0798 |

Table 3.3: Results of the training and test of the model

3.1.4 Second model: CNN

For the second experiment with the *MNIST dataset* we've used a CNN. Our purpose is to test if the 2D layers work with the posit representation. We start by using the network from the first experiment (reducing the number of neurons due to the limitation of using only the CPU) as a classifier, connected to a CNN composed of two pairs of **Conv2D** and **MaxPooling2D** layers. We've used the **tanh** as the activation function, which may cause the problem of the vanishing of the gradient, but we need it in order to test the results with the float32 with the posit160.

```

model = models.Sequential()

model = models.Sequential()

model.add(layers.Conv2D(16, (3, 3), activation='tanh', input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(32, (3, 3), activation='tanh'))
model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Flatten())
model.add(layers.Dense(64, activation='tanh'))
model.add(layers.Dense(10, activation='softmax'))

```

Float32 results

We've obtained good results with the float32, after only 5 epochs the accuracy reaches high values and the loss goes to low values. We can see the plot in the following figure:

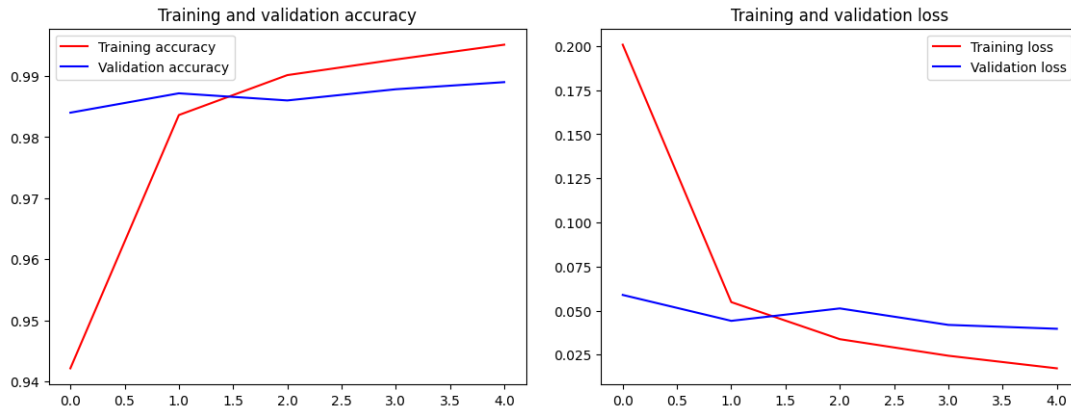


Figure 3.10: Training and validation accuracy and loss plots of the second model trained using the float32

The values reached at the end of the training and after testing the model can be seen in the following table:

| | accuracy | loss |
|------------|----------|--------|
| training | 0.9951 | 0.0173 |
| validation | 0.9890 | 0.0397 |
| test | 0.9879 | 0.0366 |

Table 3.4: Results of the training and test of the CNN

As we can see, when we use a CNN instead of a DNN, we get better results, in

fact the CNN performs very well in tasks related to images, that's the one we're trying to solve.

Posit Results

The plot of the training phase for the case using the posits is shown in figure 3.11. What we can see from this image is that **the losses are never updated**, this means that the weights are not moved from their original position.

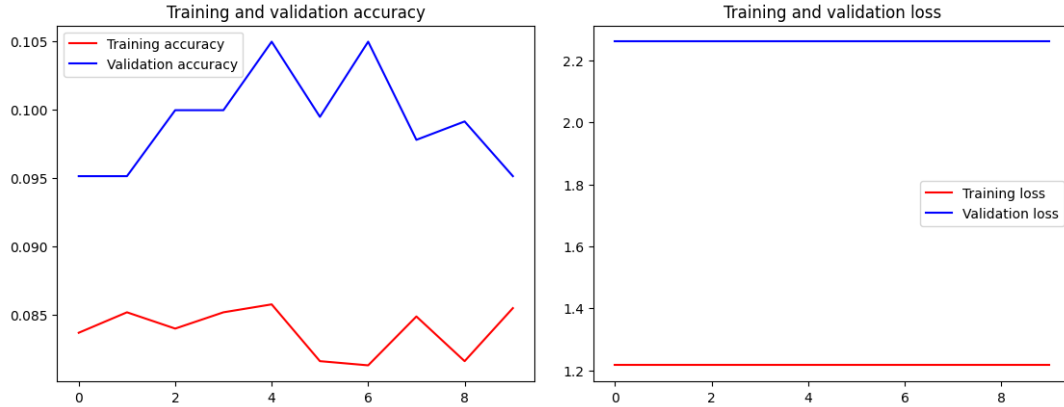


Figure 3.11: Training and validation accuracy and loss plots of the second model trained using the posit

Since we are using the *hyperbolic tangent*, the fact that the weights are not updated is not due to the fact that the update of the weights is equal to 0 because of the use of *Relu* during the first test.

In order to try to find what is the issue using the CNN we've tried different approaches.

Changing the weights initializer: using the *random normal initialiser*

Initially, we considered *inspecting the weights*, in particular we obtained the weights randomly assigned during the compilation of the model, i.e. the *initial weights*, and compared them with the weights obtained after training the model for the epochs necessary to bring the accuracy and loss into *stall*. The weights are shown in Figure 3.12 and Figure 3.13 respectively.

From the first figure we can see that the randomly initialised weights are all **negative**, another peculiarity is the fact that almost all of them **start with -0.14**.

This led us to think of a problem with the **initialisation of the weights**, a problem already observed during the first test which rendered the Relu activation function unusable. From the second figure, we can see that the weights are indeed updated, but they all remain negative.

```
[array([[[[-0.140869, -0.141602, -0.141785, -0.140625, -0.141357,
-0.140442, -0.140259, -0.141479, -0.141357, -0.140869,
-0.141968, -0.141968, -0.140869, -0.140503, -0.140686,
-0.141174, -0.141052, -0.141541, -0.140015, -0.141174,
-0.141724, -0.140442, -0.140015, -0.141541, -0.141846,
-0.141052, -0.142029, -0.140991, -0.141357, -0.141724,
-0.140991, -0.140503]],

[[-0.141296, -0.141907, -0.141113, -0.14032, -0.139954,
-0.140503, -0.140137, -0.14093, -0.139893, -0.14093,
-0.140564, -0.139832, -0.141724, -0.141418, -0.141296,
-0.141418, -0.141541, -0.141541, -0.139832, -0.141602,
-0.140503, -0.141113, -0.140686, -0.140686, -0.141663,
-0.14093, -0.140747, -0.140991, -0.141357, -0.14032,
-0.141174, -0.140808]],

[[-0.140442, -0.140259, -0.140686, -0.141907, -0.141785,
-0.140747, -0.141907, -0.140259, -0.141418, -0.140747,
-0.140198, -0.140137, -0.141968, -0.141357, -0.140137,
-0.139893, -0.141174, -0.140381, -0.140747, -0.140259,
-0.140686, -0.140686, -0.140869, -0.142029, -0.140808,
-0.140869, -0.14093, -0.140381, -0.141357, -0.14032,
-0.140076, -0.142029]]],

...
[-0.205994, -0.20697, -0.206116, ..., -0.206604, -0.207031,
-0.206421],
[-0.20575, -0.205627, -0.207581, ..., -0.206482, -0.207703,
-0.206787]], dtype=posit160),
array([0, 0, 0, 0, 0, 0, 0, 0, 0], dtype=posit160)]
```

Figure 3.12: Weights before the training

```
[array([[[[-0.203064, -0.203247, -0.203247, -0.202942, -0.203491,
-0.202698, -0.202515, -0.203064, -0.203613, -0.203125,
-0.203552, -0.20343, -0.203186, -0.202759, -0.202942,
-0.202759, -0.203247, -0.203064, -0.202271, -0.203186,
-0.202881, -0.202698, -0.202271, -0.203674, -0.203125,
-0.203369, -0.20343, -0.203186, -0.203613, -0.203247,
-0.203125, -0.20282]],

[[-0.185425, -0.186401, -0.185608, -0.184509, -0.184509,
-0.185059, -0.184692, -0.185425, -0.184082, -0.185059,
-0.18512, -0.184326, -0.185669, -0.185608, -0.185425,
-0.185486, -0.186035, -0.185669, -0.184387, -0.185669,
-0.184998, -0.185608, -0.185242, -0.184875, -0.185791,
-0.18512, -0.185242, -0.18512, -0.185547, -0.184448,
-0.185242, -0.184937]],

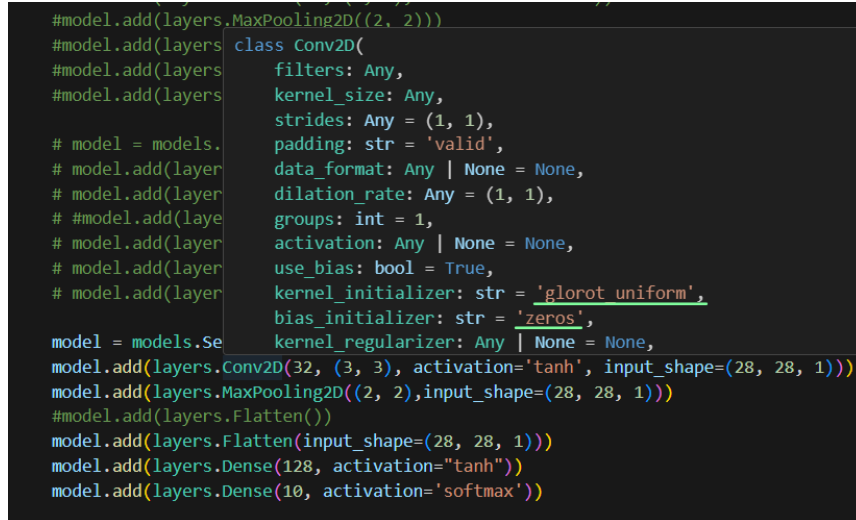
[[-0.193481, -0.194519, -0.194397, -0.193787, -0.193909,
-0.193054, -0.194153, -0.19458, -0.193359, -0.19458,
-0.194458, -0.194458, -0.193909, -0.193481, -0.19397,
-0.193726, -0.193542, -0.194702, -0.193665, -0.194092,
-0.194519, -0.192749, -0.192993, -0.194275, -0.195007,
-0.193298, -0.194702, -0.192749, -0.193542, -0.19458,
-0.192505, -0.19397]]],

...
-0.382568],
[-0.380981, -0.381836, -0.377808, ..., -0.381714, -0.383911,
-0.382935]], dtype=posit160),
array([-0.179993, -0.181213, -0.174683, -0.180908, -0.181824, -0.182495,
-0.182312, -0.180115, -0.181702, -0.181335], dtype=posit160)]
```

Figure 3.13: Weights after the training

Consequently, we decided to analyse the **default method** used for initialising the weights in both *convolutional* and *dense* layers, in order to also try to solve the problem that plagued Relu in the first test.

The **kernel initialiser** used is the *glorot uniform* as we can see in the following figure.



```
#model.add(layers.MaxPooling2D((2, 2)))
#model.add(layers.Conv2D(
#model.add(layers
#model.add(layers
    filters: Any,
    kernel_size: Any,
    strides: Any = (1, 1),
    padding: str = 'valid',
    data_format: Any | None = None,
    dilation_rate: Any = (1, 1),
    groups: int = 1,
    activation: Any | None = None,
    use_bias: bool = True,
    kernel_initializer: str = 'glorot_uniform',
    bias_initializer: str = 'zeros',
    kernel_regularizer: Any | None = None,
model = models.Sequential(
model.add(layers.Conv2D(32, (3, 3), activation='tanh', input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2), input_shape=(28, 28, 1)))
#model.add(layers.Flatten())
model.add(layers.Flatten(input_shape=(28, 28, 1)))
model.add(layers.Dense(128, activation='tanh'))
model.add(layers.Dense(10, activation='softmax'))
```

Figure 3.14: Weights after the training

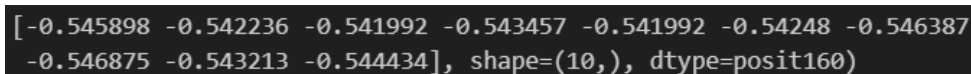
The **Glorot uniform initializer** is also called Xavier uniform initializer. It draws samples from a uniform distribution within $[-limit, limit]$, where $limit = \sqrt{6 / (fan_in + fan_out)}$ (fan_in is the number of input units in the weight tensor and fan_out is the number of output units).

The Glorot initializer is used also for the Dense layers.

We've used the following code to test the *GlorotUniform function*, as we can see we have used just for the test an array of 10 numbers and their type is *posit160*:

```
initializer = tensorflow.python.keras.initializers.GlorotUniform()
values = initializer(shape=(10,), dtype="posit160")
print(values)
```

An example of output of the Glorot function used outside a keras layer is the following:



```
[-0.545898 -0.542236 -0.541992 -0.543457 -0.541992 -0.54248 -0.546387
 -0.546875 -0.543213 -0.544434], shape=(10,), dtype=posit160)
```

Figure 3.15: Results obtained from the Glorot function using posits

As we can see, the results are all negative numbers and the first two decimal are always the same (-0.54 in this case). An important aspect we discovered is that

if the type is not explicitly passed as an argument, this function does not generate posits by default.

In the next figure we can see the same output of the Glorot, but using the float32 as dtype:

```
[ 0.48643327  0.30375326 -0.10060766  0.5301838  -0.00153506  0.00609881
 0.4547181   0.32007796  0.08421242 -0.23402122], shape=(10,), dtype=float32)
```

Figure 3.16: Results obtained from the Glorot function using the floats

Instead of using the glorot initialiser, we decided to implement our own initialiser to make sure that the weights were really randomly initialised 'posits'. In the following code we can see the function that will be used as kernel initializer:

```
from tensorflow import random
random.set_seed(1024)

def random_normal_init(shape, dtype=None):
    return random.normal(shape, dtype="posit160")
```

We've tested the code and the output is the following:

```
[-0.789551 -0.259888 -1.29736 0.352051 0.87207 1.00049 0.447876 -0.539551
 -0.806396 -1.05664], shape=(10,), dtype=posit160)
```

Figure 3.17: Results obtained from the random normal function

As we can see the results are distributed uniformly in a larger interval which includes also positive numbers, solving the problem observed using the *Glorot function*. We've decided to set as kernel initializer the *random uniform initializer*; the model obtained is the following:

```
model = models.Sequential()

model.add(layers.Conv2D(32, (3, 3),
                        activation='tanh',
                        input_shape=(28, 28, 1),
                        kernel_initializer=random_normal_init))

model.add(layers.MaxPooling2D((2, 2), input_shape=(28, 28, 1)))
model.add(layers.Flatten(input_shape=(28, 28, 1)))

model.add(layers.Dense(128,
```

```

activation="tanh",
kernel_initializer=random_normal_init))

model.add(layers.Dense(10,
activation='softmax',
kernel_initializer=random_normal_init))

```

We then trained the network, but due to the fact that we have limitations due to CPU usage that in this case was performing very poorly, we reduced the training set samples from 60000 to 2000 and the test set was consequently reduced to 200 samples. The number of samples used during training is small, so we do not expect the network to generalise well, as we expect to run into the case of overfitting, but this test is important not so much to obtain good results, but to test whether the use of the new kernel initialiser solves the stalling problem observed earlier. In the figure 3.19 we can see the plot of the training and validation accuracies and losses. We can see that using the *random uniform initialiser* we've solved the problem of the stall. We've tested the same network with the same kernel initializer also using the float32, the time performances will be presented in the final table.

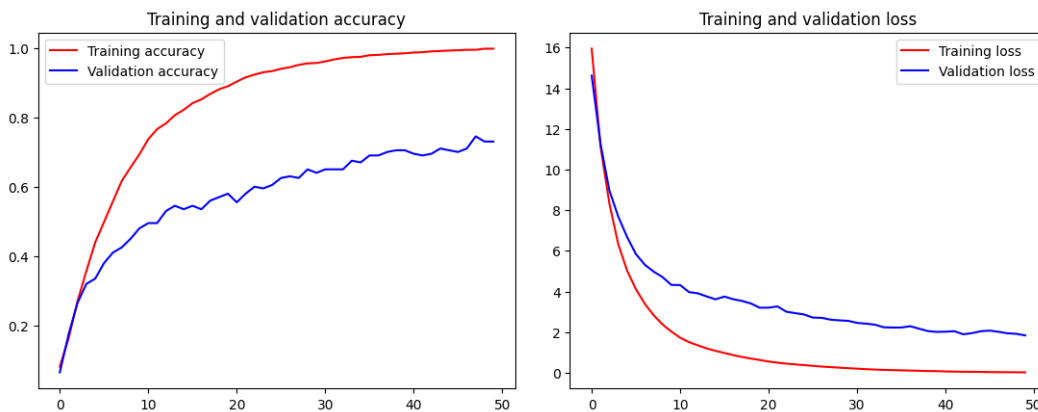


Figure 3.18: Training and validation accuracy and loss plots of the second model trained using the posit with the *random uniform initialiser*

3.1.5 Third model: CNN with more layers, Relu and Random Uniform Initialiser

The test we performed next is to test a more complex CNN with Relu activation function both in case of float32 and posits, the weights are granted to be the same thanks to the usage of the same **seed**. The network is composed by three pairs of conv2D and maxpooling2D layers, and a classifier at the end:

```

model = models.Sequential()

model.add(layers.Conv2D(32, (3, 3),

```

```

        activation='relu',
        input_shape=(28, 28, 1),
        kernel_initializer=random_normal_init))
model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(32, (3, 3),
        activation='relu',
        kernel_initializer=random_normal_init))
model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(32, (3, 3),
        activation='relu',
        kernel_initializer=random_normal_init))
model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Flatten())
model.add(layers.Dense(10, activation='softmax',
        kernel_initializer=random_normal_init))

```

Posit results

The plots obtained during the training using the posit can be seen in figure 3.19: we can see that now the *Relu* can be used as activation function.

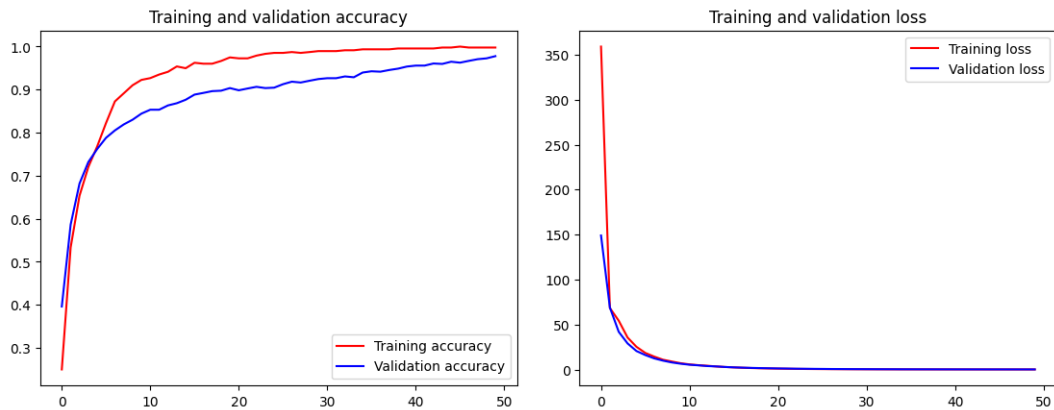


Figure 3.19: Training and validation accuracy and loss plots of the third model using the posits

Using this model we've reached very good results (see table 3.5) in terms of loss and accuracy, but the problem was the amount of time it took to train the network, as it took **222 minutes** to train using only a third of the samples present in the dataset. We can see that training the network on 20000 samples **negatively** affects the ability to generalise and recognise images never seen before, to solve this we only need to use the complete dataset, but due to the limitation and the time required

we’ve just used this undersampling. The important thing here is that the *Relu* **works correctly** with the chosen kernel initialiser.

| | accuracy | loss |
|------------|----------|--------|
| training | 0.9978 | 0.0462 |
| validation | 0.9778 | 0.0838 |
| test | 0.8269 | 0.1071 |

Table 3.5: Results of the training and test of the third model using the posits

Now we’ve to check these results against the ones obtained using the float32.

Float32 results

Using the same settings we’ve trained the model using the floats. We obtained similar results to those obtained with posits, but in this case the training **speed was much better**: we went from **222** minutes with posits to **3** minutes with floats. The plots and the results are respectively in figure 3.20 and in table 3.6.

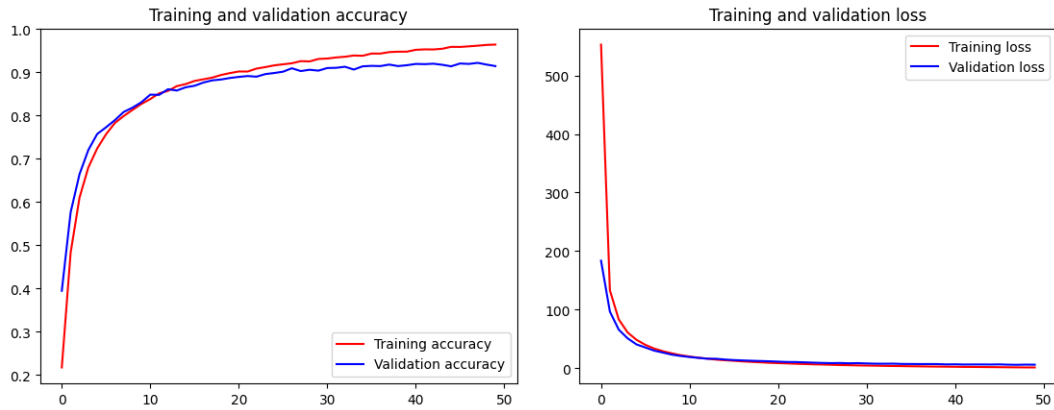


Figure 3.20: Training and validation accuracy and loss plots of the third model using the float32

| | accuracy | loss |
|------------|----------|--------|
| training | 0.9642 | 1.4976 |
| validation | 0.9145 | 6.1793 |
| test | 0.9116 | 7.6489 |

Table 3.6: Results of the training and test of the third model using the float32

Respect to the case with posits, we need more epochs to converge and the loss values are higher.

3.1.6 Fourth model: Adding dropout

For the next test, we used the same model as for the third test, but with the addition of a dropout layer. In this test, we intend to observe whether the dropout works correctly using posits, so the test set is reduced only to 2000 samples to speed up the process. Since the architecture is the same, the following is just the summary of the model with the new added layer:

| Layer (type) | Output Shape | Param # |
|--------------------------------|--------------------|---------|
| conv2d (Conv2D) | (None, 26, 26, 32) | 320 |
| max_pooling2d (MaxPooling2D) | (None, 13, 13, 32) | 0 |
| conv2d_1 (Conv2D) | (None, 11, 11, 32) | 9248 |
| max_pooling2d_1 (MaxPooling2D) | (None, 5, 5, 32) | 0 |
| conv2d_2 (Conv2D) | (None, 3, 3, 32) | 9248 |
| max_pooling2d_2 (MaxPooling2D) | (None, 1, 1, 32) | 0 |
| dropout (Dropout) | (None, 1, 1, 32) | 0 |
| flatten (Flatten) | (None, 32) | 0 |
| dense (Dense) | (None, 10) | 330 |
| Total params: 19,146 | | |
| Trainable params: 19,146 | | |
| Non-trainable params: 0 | | |

We've trained the network obtaining the following results (figure 3.21):

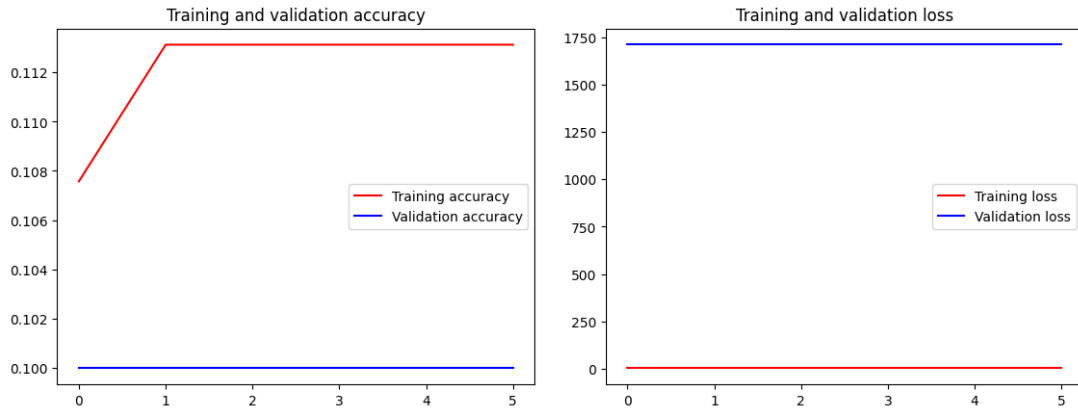


Figure 3.21: Training and validation accuracy and loss plots of the second model trained using the posit with the a dropout layer

From the plots we can see that the network **doesn't learn** with the dropout layer. It's the same problem that we've obtained before, solved using a new *weights initializer*. We've started to investigate the problem by observing the weights **before** and **after** the training, and we can see them respectively in figure 3.22 and in figure 3.23.

```
[array([[[[0.614258, 1.66406, 1.53076, -0.784668, 0.142151, 1.05859,
          1.01807, 1.09521, 1.8877, -0.141479, 0.861572, -0.953369,
          0.0431213, -2.97363, -1.39062, -0.467529, 1.1748, -0.572998,
          2.06543, 0.693115, 1.04834, -1.24219, -0.0320129, 0.852783,
          -1.98779, 0.441772, 0.564209, 0.723389, -0.0661316, -0.62085,
          -1.09912, -0.539551]],

         [[-1.3501, -0.0612488, -0.79126, 0.63623, 0.605713, -0.174133,
          -1.44189, 1.24414, -1.97217, 1.38916, 1.30029, -1.5835,
          0.845459, 1.22754, -0.482666, 0.744629, 0.93042, -0.344849,
          -0.482178, -1.53613, 0.391479, 1.771, 1.38428, -0.319336,
          -0.616943, 0.482422, -1.55957, -0.603271, 0.24585, 0.119385,
          -0.829102, 0.651611]],

         [[1.0957, -0.20752, 0.0724182, -1.59766, -0.711182, -0.817871,
          1.45508, -0.504639, -0.759033, 0.434937, 1.49658, -0.211792,
          0.0694885, 1.06934, -1.74365, 0.0834351, -1.87842, -0.916992,
          -1.44238, 0.346558, -0.368408, -1.59277, -0.174072,
          -0.157349, 0.14447, -0.256348, 1.78857, 1.65967, 0.810303,
          1.11816, -2.74805, -0.132874]]],

        [[[-0.614258, -0.261108, -0.286377, 0.300659, -0.0980225,
          -0.717041, -2.41406, -0.74292, 0.794922, 0.00938416,
          -0.857178, -2.11133, -2.40137, 1.98975, 0.435791, -1.63965,
          ...
          [2.31641, 1.1543, 2.12891, 0.557617, -0.175964, 0.554932, 1.71582,
          -1.11523, -0.158508, 0.843994],
          [-0.0855103, -1.70117, -0.349487, -1.6416, -0.684326, 1.31494,
          -0.171387, -0.010704, -1.80469, 0.112396]], dtype=posit160),
        array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0], dtype=posit160)]
```

Figure 3.22: Weights before the training

```
[array([[[[0.614258, 1.66406, 1.53076, -0.784668, 0.142151, 1.05859,
1.01807, 1.09521, 1.8877, -0.141479, 0.861572, -0.953369,
0.0431213, -2.97363, -1.39062, -0.467529, 1.1748, -0.572998,
2.06543, 0.693115, 1.04834, -1.24219, -0.0320129, 0.852783,
-1.98779, 0.441772, 0.564209, 0.723389, -0.0661316, -0.62085,
-1.09912, -0.539551]]],

[[-1.3501, -0.0612488, -0.79126, 0.63623, 0.605713, -0.174133,
-1.44189, 1.24414, -1.97217, 1.38916, 1.30029, -1.5835,
0.845459, 1.22754, -0.482666, 0.744629, 0.93042, -0.344849,
-0.482178, -1.53613, 0.391479, 1.771, 1.38428, -0.319336,
-0.616943, 0.482422, -1.55957, -0.603271, 0.24585, 0.119385,
-0.829102, 0.651611]]],

[[1.0957, -0.20752, 0.0724182, -1.59766, -0.711182, -0.817871,
1.45508, -0.504639, -0.759033, 0.434937, 1.49658, -0.211792,
0.0694885, 1.06934, -1.74365, 0.0834351, -1.87842, -0.916992,
-1.44238, 0.346558, -0.368408, -1.59277, -0.174072,
-0.157349, 0.14447, -0.256348, 1.78857, 1.65967, 0.810303,
1.11816, -2.74805, -0.132874]]],

[[[-0.614258, -0.261108, -0.286377, 0.300659, -0.0980225,
-0.717041, -2.41406, -0.74292, 0.794922, 0.00938416,
-0.857178, -2.11133, -2.40137, 1.98975, 0.435791, -1.63965,
...
[-0.0855103, -1.70117, -0.349487, -1.6416, -0.684326, 1.31494,
-0.171387, -0.010704, -1.80469, 0.112396]], dtype=posit160),
array([-0.000845909, 0.00347137, -0.00062561, -0.00140953, 0.0022583,
-0.00391388, -0.00069809, 0.00398254, -0.00333405, 0.000995636],
dtype=posit160)]
```

Figure 3.23: Weights after the training

From the images we can see that the weights of the first layers are not updated, so there is a problem with the backpropagation of the error using the gradient descent method, while the last layers are updated. We thought that the weights after the dropout layer were updated, probably just once at the first epoch, and the ones before weren't. Using this network it is difficult to see if this is actually the problem, therefore we've decided to use a very simple network with a number of weights that can be observed. We want to see if it is the dropout layer that breaks the network.

The model used is the following, also in this case we only present the summary of the model, the parameters are the same as before, except for the number of filters, which is equal to 1, to obtain very few weights:

| Layer (type) | Output Shape | Param # |
|--------------------------------|-------------------|---------|
| conv2d (Conv2D) | (None, 26, 26, 1) | 10 |
| max_pooling2d (MaxPooling2D) | (None, 13, 13, 1) | 0 |
| conv2d_1 (Conv2D) | (None, 11, 11, 1) | 10 |
| max_pooling2d_1 (MaxPooling2D) | (None, 5, 5, 1) | 0 |

| | | |
|--------------------------------|-----------------|----|
| conv2d_2 (Conv2D) | (None, 3, 3, 1) | 10 |
| max_pooling2d_2 (MaxPooling2D) | (None, 1, 1, 1) | 0 |
| dropout (Dropout) | (None, 1, 1, 1) | 0 |
| flatten (Flatten) | (None, 1) | 0 |
| dense (Dense) | (None, 10) | 20 |
| Total params: 50 | | |
| Trainable params: 50 | | |
| Non-trainable params: 0 | | |

The results of the training are the same as before. Then we've compared the weights before and after the training: the layers **before** the dropout layer are not updated, while the ones **after** are updated.

We've moved the dropout layer after the first two pairs of conv2D-MaxPooling2D, the summary of the model is the following:

| Layer (type) | Output Shape | Param # |
|---------------------------------|-------------------|---------|
| conv2d_10 (Conv2D) | (None, 26, 26, 1) | 10 |
| max_pooling2d_9 (MaxPooling2D) | (None, 13, 13, 1) | 0 |
| conv2d_11 (Conv2D) | (None, 11, 11, 1) | 10 |
| dropout_2 (Dropout) | (None, 11, 11, 1) | 0 |
| max_pooling2d_10 (MaxPooling2D) | (None, 5, 5, 1) | 0 |
| conv2d_12 (Conv2D) | (None, 3, 3, 1) | 10 |
| max_pooling2d_11 (MaxPooling2D) | (None, 1, 1, 1) | 0 |
| flatten_3 (Flatten) | (None, 1) | 0 |
| dense_3 (Dense) | (None, 10) | 20 |
| Total params: 50 | | |

```
Trainable params: 50
Non-trainable params: 0
```

We've analysed the results: the weights of the **convolutional base** are not updated independently of the position of the dropout layer, while the weights of the **dense layer** are updated only once during the first epoch. Therefore, we can place the dropout layer wherever we want with no change in the weights that make up the convolutional base: we can conclude that the positioning of the dropout layer is not the problem.

3.1.7 Analysing the dropout layer

We analysed the dropout layer directly in order to observe which steps are performed to achieve the result and what is the output of a dropout layer in case of posits. The Dropout layer randomly sets input units to **0** with a frequency of **rate** (a parameter to be passed) at each step during training time, which helps prevent overfitting. Inputs *not set to 0* are scaled up by $1/(1 - \text{rate})$, so are multiplied by this scaling up factor (for 0.5 as rate, the factor is 2). The behaviour of the dropout layer used as a standalone layer is the following, using the *float32* as dtype:

```
random.set_seed(0)
layer = k.layers.Dropout(.2, input_shape=(2,))
data = np.arange(10).reshape(5, 2).astype("float32")
print(data)

[[0.  1.]
 [2.  3.]
 [4.  5.]
 [6.  7.]
 [8.  9.]]

outputs = layer(data, training=True)
print(outputs)

tf.Tensor(
[[ 0.    1.25]
 [ 2.5   3.75]
 [ 5.    6.25]
 [ 0.    8.75]
 [10.   11.25]], shape=(5, 2), dtype=float32)
```

What we can see is that some values of the input are zeroed, while the others are scaled up to obtain the same sum of the elements as before.

Then we did the same test but with the posits, to do this we set the default float type to the posits and the seed to the same as the floats, to make the experiment repeatable:

```
random.set_seed(0)
k.backend.set_floatx("posit160")
```

Then we applied the same steps used for the floats:

```
layer = k.layers.Dropout(.2, input_shape=(2,))
data = np.arange(10).reshape(5, 2).astype("posit160")
data

array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7],
       [8, 9]], dtype=posit160)

outputs = layer(data, training=True)
print(outputs)

tf.Tensor(
[[0 0]
 [0 0]
 [0 0]
 [0 0]
 [0 0]], shape=(5, 2), dtype=posit160)
```

We have obtained a matrix whose elements are all 0, so during the call to the dropout layer something unexpected happens in the calculation using posits.

Keras source code

In order to solve the problem, we decided to inspect the source code of the dropout layer available at the following link:

<https://github.com/keras-team/keras/blob/v2.6.0/keras/layers/core.py>

The dropout class is defined inside the *core.py* module, we've created a new module in order to test it and we've tried running all the steps in its call method (the one that generates the output) to see where the problem occurs. The call method is the following:

```

def call(self, inputs, training=None):
    if training is None:
        training = K.learning_phase()

    def dropped_inputs():
        return nn.dropout(
            inputs,
            noise_shape=self._get_noise_shape(inputs),
            seed=self.seed,
            rate=self.rate)

    output = control_flow_util.smart_cond(
        training, dropped_inputs,
        lambda: array_ops.identity(inputs))
    return output

```

In the call method is present a function called **dropped_inputs** that return the output of another function called **nn.dropout** defined in the module **tensorflow.python.ops**. Since this function is used as *predicate* to generate the masked output, then we've tested it using the floats and the posits obtaining the same results as the previous test: there is a problem in the function **nn.dropout**.

We've then analysed the code of the *nn.dropout* function: it takes as input a **matrix X**, a **rate** that is the probability to put at 0 an element in the matrix, the **shape** of the *mask* to be generated and a **seed** to make the results replicable. The main steps of the function are the following:

1. It generates a random tensor matrix **R** with shape equal to the *noise shape*, the *seed* equal to the passed seed and the *dtype* predefined, in our case the posits.
2. Since the values from the matrix are uniformly distributed in the [0,1] interval the mask is generated using the following formula:

$$M_{ij} = \begin{cases} 1, & \text{if } R_{ij} \geq \text{rate} \\ 0, & \text{otherwise} \end{cases} \quad (3.1)$$

The random tensor is generated using the function:

random_ops.random_uniform from the *tensorflow.python.ops* module.

3. The input matrix is multiplied by this matrix.
4. The resulting matrix is scaled up by a factor of $1/(1 - \text{rate})$ in order to keep the same sum of elements of the starting matrix **X**

An example is the following:

$$X = \begin{bmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \\ 6 & 7 \\ 8 & 9 \end{bmatrix} R = \begin{bmatrix} 0.1160804 & 0.32475173 \\ 0.17018151 & 0.39640236 \\ 0.7649511 & 0.62501967 \\ 0.85889864 & 0.06789052 \\ 0.27476895 & 0.690626 \end{bmatrix} \rightarrow M = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \rightarrow Result = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 8 & 10 \\ 12 & 0 \\ 0 & 18 \end{bmatrix}$$

We can see that the survived elements from the starting matrix are doubled, since using a rate equal to **0.5**, the scaling up factor is equal to $1/(1 - 0.5) = 2$, which is used to multiply the elements of the Result matrix. The previous example was obtained using the *float32* as default data type for the backend.

An example using the posits is the following:

$$X = \begin{bmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \\ 6 & 7 \\ 8 & 9 \end{bmatrix} R = \begin{bmatrix} 0.00466156 & 0.00633621 \\ 0.00363541 & 0.0035553 \\ 0.0073967 & 0.00303268 \\ 0.00217438 & 0.00353241 \\ 0.00321198 & 0.00325775 \end{bmatrix} \rightarrow M = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \rightarrow Result = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$$

We can clearly see that the problem is in the generation of the **random tensor matrix R** using the **random_uniform** function. The random uniform function is generating values not uniformly distributed in the interval [0,1] when the data type passed to the function is **posit160**. The values have the same format that we've seen in the previous experiments, that are the causes of the other problems as well.

We can conclude that there is a **huge** problem, in the generation of **random uniform distributed** values using the posits and the function of tensorflow.

We've used the same function that we've used to generate the random weights for the kernel of the **conv2D** layers and it works, but in this case we can't use the previous function since we need to generate uniform values in the interval [0,1] and apply the comparison with the rate to generate the mask.

We've solved the problem by not using the posits as data type for the generation of the matrix, but to use always the *float32*: this is not a problem since the matrix **R** is used only to generate the matrix **M** by comparing its element with the **rate**. The dropout function code is modified in order to call the **random_uniform** function with the data type **float32** when the floatx type is *posit160*.

The results using the changed **dropout** function are the following (using the posits):

$$X = \begin{bmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \\ 6 & 7 \\ 8 & 9 \end{bmatrix} R = \begin{bmatrix} 0.2390374 & 0.92039955 \\ 0.05051243 & 0.49574447 \\ 0.8355223 & 0.02647042 \\ 0.08811307 & 0.4566604 \\ 0.76883924 & 0.7376363 \end{bmatrix} \rightarrow M = \begin{bmatrix} 0 & 1 \\ 0 & 0 \\ 1 & 0 \\ 0 & 0 \\ 1 & 1 \end{bmatrix} \rightarrow Result = \begin{bmatrix} 0 & 2 \\ 0 & 0 \\ 8 & 0 \\ 0 & 0 \\ 16 & 18 \end{bmatrix}$$

Using this configuration the problem of the dropout layer is solved.

3.1.8 Fourth model with the new Dropout Layer

Now we've to test the new dropout function on the starting network; in order to do this we've created a module named **dropout.py** containing the definition of a **Dropout class** that uses the modified *dropout function* defined in the same module. The summary of the model is the following (we've used the same configuration of the previous model, but we've added the modified dropout layer):

| Layer (type) | Output Shape | Param # |
|--------------------------------|--------------------|---------|
| conv2d_6 (Conv2D) | (None, 26, 26, 32) | 320 |
| max_pooling2d_6 (MaxPooling2D) | (None, 13, 13, 32) | 0 |
| conv2d_7 (Conv2D) | (None, 11, 11, 32) | 9248 |
| max_pooling2d_7 (MaxPooling2D) | (None, 5, 5, 32) | 0 |
| dropout_1 (Dropout) | (None, 5, 5, 32) | 0 |
| conv2d_8 (Conv2D) | (None, 3, 3, 32) | 9248 |
| max_pooling2d_8 (MaxPooling2D) | (None, 1, 1, 32) | 0 |
| flatten_2 (Flatten) | (None, 32) | 0 |
| dense_2 (Dense) | (None, 10) | 330 |
| Total params: 19,146 | | |
| Trainable params: 19,146 | | |
| Non-trainable params: 0 | | |

We've trained the network obtaining the following results (figure 3.24):

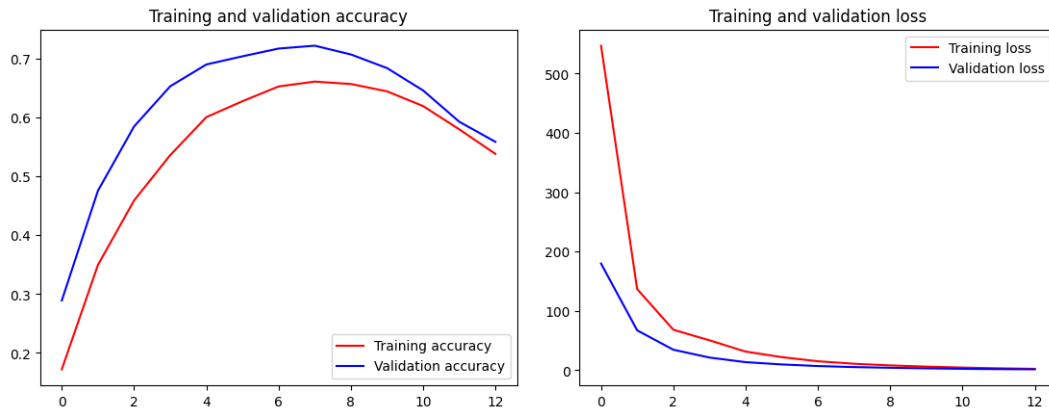


Figure 3.24: Training and validation accuracy and loss plots of the model with the custom dropout layer using the posits

The network achieved these results after 50 minutes.

While training it using the floats we've obtained the following results (figure 3.25):), in **50** seconds:

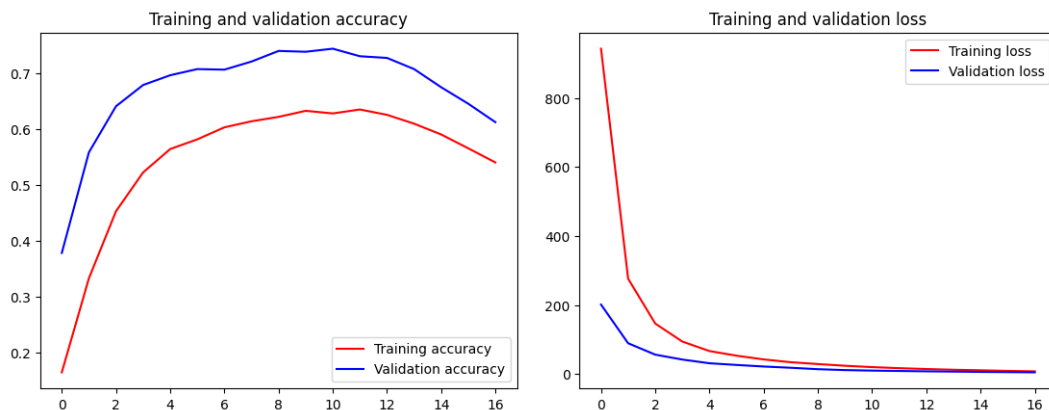


Figure 3.25: Training and validation accuracy and loss plots of the model with the custom dropout layer using the float32

We are clearly in a case of **underfitting** induced by the use of a useless dropout layer. Our goal was to see if the new dropout layer worked properly obtaining a **positive** result.

Using the MNIST dataset even with downsampling is not compatible with the use of dropout, which deteriorates the performance of the model.

3.1.9 LeNet-5

LeNet-5 is a convolutional neural network structure proposed by LeCun in 1998 that was specifically designed for handwritten digit recognition. The architecture of LeNet-5 consists of seven layers, including three convolutional layers followed by two fully connected layers and two subsampling layers. We decided to try this architecture since it is more complex than the ones we used for the previous experiments, we want to see if it works fine even with posit numbers. In the following picture we can see its architecture:

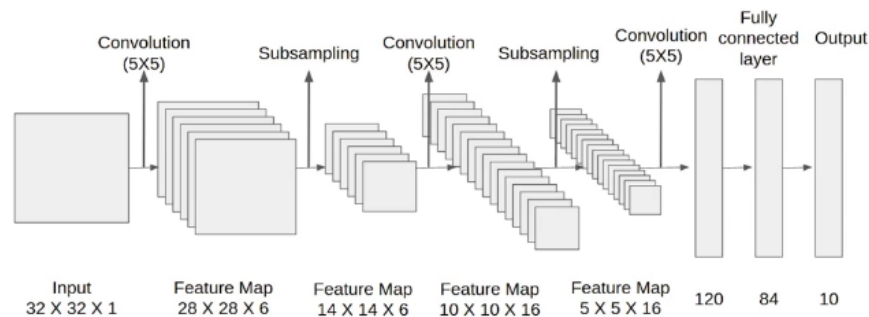


Figure 3.26: Lenet-5 architecture

We defined the model using keras, the only difference with the architecture showed in picture 3.26 is that the input shape is 28x28x1.

```
lenet_5_model = keras.models.Sequential([
    keras.layers.Conv2D(6, kernel_size=5, strides=1, activation='tanh',
                        input_shape=train_images[0].shape, padding='same'), #C1
    keras.layers.AveragePooling2D(), #S2
    keras.layers.Conv2D(16, kernel_size=5, strides=1,
                        activation='tanh', padding='valid'), #C3
    keras.layers.AveragePooling2D(), #S4
    keras.layers.Conv2D(120, kernel_size=5, strides=1,
                        activation='tanh', padding='valid'), #C5
    keras.layers.Flatten(), #Flatten
    keras.layers.Dense(84, activation='tanh'), #F6
    keras.layers.Dense(10, activation='softmax') #Output layer
])
```

We tried this model using both float and posit numbers. When using **float** numbers the model performed extremely well, after 10 epochs it reached a validation accuracy of **98%** and a validation loss of **0.069**. However for the posits we encountered an error in the layer *AveragePooling2D*. We report the error:

```

InvalidArgumentError: No OpKernel was registered to support Op 'AvgPool' used by {{node sequential_5/average_pooling2d_10/AvgPool}}
Registered devices: [CPU]
Registered kernels:
device='XLA_CPU_JIT'; T in [DT_FLOAT, DT_DOUBLE, DT_BFLOAT16, DT_HALF]
device='CPU'; T in [DT_BFLOAT16]
device='CPU'; T in [DT_HALF]
device='CPU'; T in [DT_FLOAT]
device='CPU'; T in [DT_DOUBLE]

[[sequential_5/average_pooling2d_10/AvgPool]] [Op:__inference_train_function_5113]

```

Figure 3.27: AveragePooling2D error

It says that $T=DT_POSIT160$ is not in the list of Registered kernels for the layer and so it can't be used. So we decided to change the architecture and use *MaxPooling2D* layers instead of *AveragePooling2D*. The model summary is:

| Layer (type) | Output Shape | Param # |
|-------------------------------|---------------------|---------|
| input_1 (InputLayer) | [(None, 28, 28, 1)] | 0 |
| conv2d (Conv2D) | (None, 28, 28, 6) | 156 |
| max_pooling2d (MaxPooling2D) | (None, 14, 14, 6) | 0 |
| conv2d_1 (Conv2D) | (None, 10, 10, 16) | 2416 |
| max_pooling2d_1 (MaxPooling2) | (None, 5, 5, 16) | 0 |
| conv2d_2 (Conv2D) | (None, 1, 1, 120) | 48120 |
| flatten (Flatten) | (None, 120) | 0 |
| dense (Dense) | (None, 84) | 10164 |
| dense_1 (Dense) | (None, 10) | 850 |
| Total params: 61,706 | | |
| Trainable params: 61,706 | | |
| Non-trainable params: 0 | | |

Figure 3.28: Lenet-5 model summary

Float32 results

Using the configuration with the **float32** we've obtained the following results during the training phase:

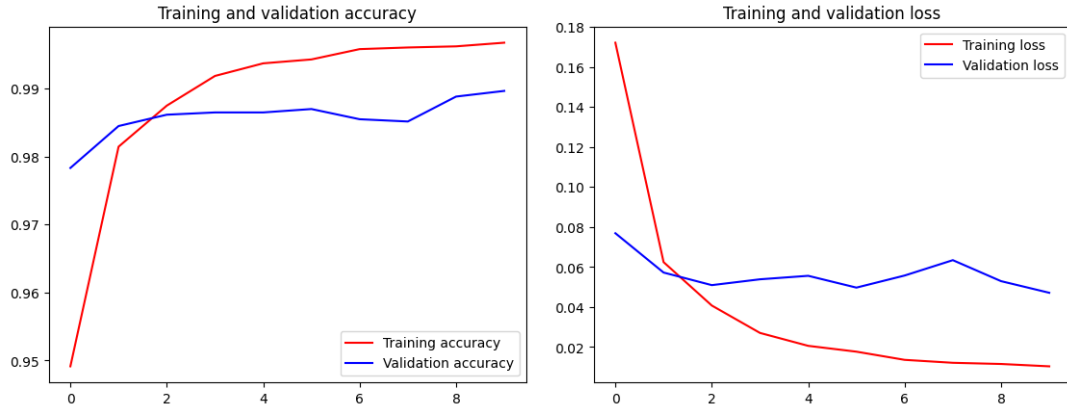


Figure 3.29: Training and validation accuracy and loss plots of the Lenet-5 model trained using the float32

As we can see from figure 3.29 we've obtained good results for both training and validation accuracy, while the actual values achieved are presented in table 3.7.

| | accuracy | loss |
|------------|----------|--------|
| training | 0.9968 | 0.0103 |
| validation | 0.9897 | 0.0471 |
| test | 0.9890 | 0.0406 |

Table 3.7: Results of the training and test of the model

The performances of the model using *MaxPooling2D* layers are still really good, just like the lenet-5 model with *AveragePooling2D* layers we reach a validation accuracy of nearly **99%** and a validation loss of **0.05**. The total time the model took to complete the training phase of 10 epochs is **229** seconds (3.82 minutes), in average 14ms/step.

Posit results

For this experiment we used the same model but added *kernel_initializer=random_normal_init* in the parameters of the *Conv2D* and *Dense* layers for the same reasons as we explained before. We report the results obtained:

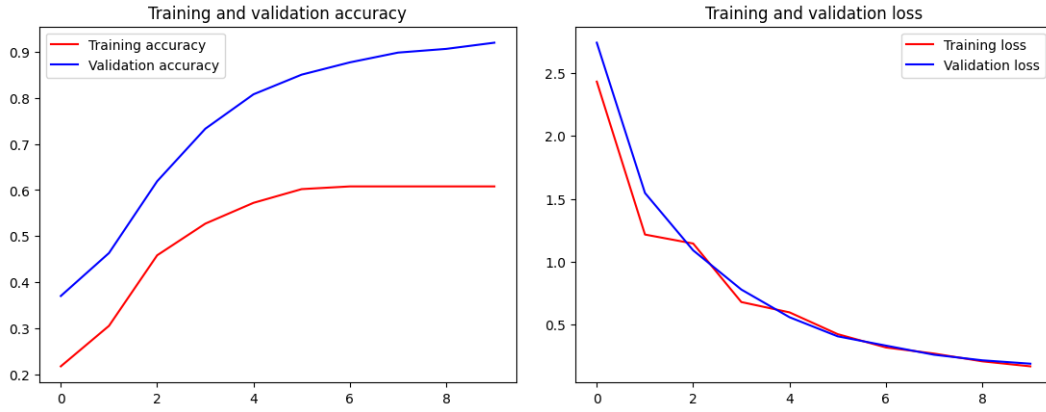


Figure 3.30: Training and validation accuracy and loss plots of the model trained using the posit160

In the results we can notice how the model performs better in the validation set than in the training set, and we can see that it needs more than 9 epochs to exceed 90% of performance. The total time the model took to complete the training phase of 10 epochs is **3464** seconds (57.7 minutes), in average 834ms/step (352 seconds for each epoch).

The actual values achieved are presented in table 3.8.

| | accuracy | loss |
|------------|----------|--------|
| training | 0.6079 | 0.1680 |
| validation | 0.9199 | 0.1890 |
| test | 0.7500 | 0.1929 |

Table 3.8: Results of the training and test of the model

We can see that the performances aren't as good as in the case of floats, in the training phase the model needs 9 epochs to exceed 90% while with floats it already surpassed it from the very first 2 epochs. Even the generalization power is a worse, reaching in the test set only **75%** of accuracy against **99%** with floats. In general its performance are good but not as much as floats, however we can see that the model performance didn't stall at a certain point, but would probably continue to increase with more epochs. We trained again with 30 epochs and obtained the following results:

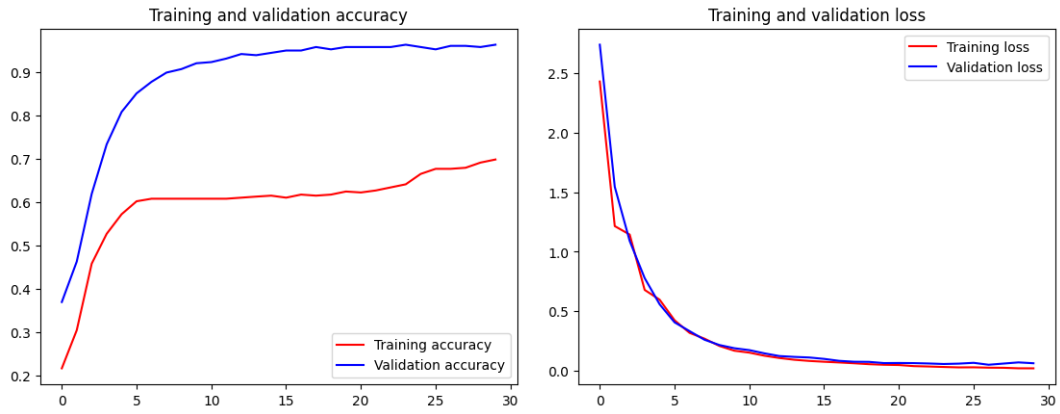


Figure 3.31: Training and validation accuracy and loss plots of the model trained using the posit160

Table 3.9 presents the actual values achieved.

| | accuracy | loss |
|------------|----------|--------|
| training | 0.6982 | 0.0199 |
| validation | 0.9626 | 0.0627 |
| test | 0.8684 | 0.0607 |

Table 3.9: Results of the training and test of the model

As expected with more epochs the model improved its performance reaching a validation accuracy of **96%** and a test accuracy of **87%**. The results however are still not as good as when the model used float and exceeded **98%** of accuracy in both the validation and test set.

3.1.10 MNIST Conclusion and summary

The test on the LeNet-5 was the last experiment conducted on the MNIST dataset, we solved the problems presented in the previous project and found a problem that most likely concerns the implementation of the function that uniformly generates samples in a range. In the following table we can see a summary of the results obtained from the experiments on the MNIST dataset and the performance in terms of time achieved on both the floats and the posit implementations.

| Model | | Val Accuracy | Val Loss | Test Accurac | Test Loss |
|----------------------|-----------------|--------------|----------|--------------|-----------|
| Simple DNN | Float32 | 0.9785 | 0.0728 | 0.9900 | 0.0355 |
| | Posit160 | 0.9275 | 0.2239 | 0.9800 | 0.0804 |
| CNN with more layers | Float32 | 0.9145 | 6.1793 | 0.9642 | 1.4976 |
| | Posit160 | 0.9778 | 0.0838 | 0.9978 | 0.0462 |
| CNN + dropout | Float32 | 0.6125 | 4.5410 | 0.5404 | 7.4505 |
| | Posit160 | 0.5591 | 1.6060 | 0.5386 | 2.4463 |
| LeNet-5 | Float32 | 0.9897 | 0.0103 | 0.9897 | 0.0471 |
| | Posit160 | 0.9626 | 0.0607 | 0.8684 | 0.0607 |

Comparison of MINST Model Training Times

| Model | | Number of Epochs | Avg Training Time per epoch | Tot. Training Time |
|----------------------|-----------------|------------------|-----------------------------|--------------------|
| Simple DNN | Float32 | 10 | 3 s | 33 s |
| | Posit160 | 34 | 2 s | 82 s |
| Simple CNN | Float32 | 25 | 4 s | 96 s |
| | Posit160 | 50 | 2 s | 100 s |
| CNN with more layers | Float32 | 50 | 4 s | 200 s |
| | Posit160 | 50 | 270 s | 13500 s |
| CNN + dropout | Float32 | 17 | 3 s | 50 s |
| | Posit160 | 13 | 232 s | 3018 s |
| LeNet-5 | Float32 | 10 | 22.9 s | 229 s |
| | Posit160 | 30 | 361 s | 10830 s |

3.2 CIFAR-10

CIFAR-10 is a dataset for image classification consisting of **60000** 32×32 color images in 10 classes, with **6000** images per class. In total there are **5000** images for the training and **1000** images for the testing. The classes include common objects such as airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks.

In the following picture we can see an example of an image for each class of the dataset:

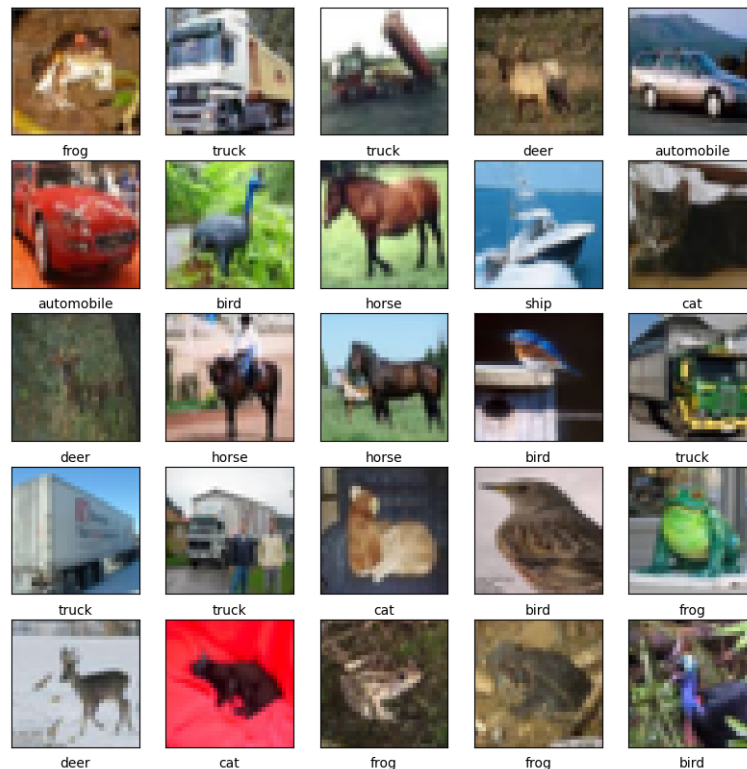


Figure 3.32: cifar-10 classes and images

We decided to try the CIFAR-10 dataset since it's more complex compared to MNIST due to the fact that we deal with RGB-colored images.

3.2.1 Download and prepare the dataset

In order to use the *CIFAR-10 dataset*, we need to retrieve it and pre-process it so that it can be used by our model. The dataset is available in the *tensorflow.keras.datasets.cifar10* module and using the *load_data()* we've obtained two tuples of NumPy arrays: (train_images, train_labels), (test_images, test_labels). The first elements of the tuples are the images for training and testing respectively, the second elements are their labels.

The images in both sets are in the range **[0,255]**, so we've rescaled them to the range **[0,1]** to avoid computing with large numbers and to make better use of the

posit representation.

For our experiments we decided to use from the dataset only **10000** images for the training set and **1000** images for the test set, in order to speed up the training times.

3.2.2 First CNN model

We decided to use as the first model a CNN consisting of three *Conv2D* and *MaxPooling2D* layers, and finally a *Flatten* and two *Dense* layers. We used as activation functions all *relu* functions, only the final *Dense* layer has a *linear* activation, it also has 10 units since we have 10 classes. One difference with the models we used for the MINST dataset is the input shape, in this case the shape is (32, 32, 3) since the images are 32×32 with 3 channels (RGB) since are colored images. In the MINST dataset we had gray-scaled images, so with only 1 channel.

The code used for the model is the following:

```
inputs = k.Input(shape=(32, 32, 3))
x = k.layers.Conv2D(32, kernel_size=3, strides=1, activation='relu')(inputs)
x = k.layers.MaxPooling2D((2, 2))(x)
x = k.layers.Conv2D(64, kernel_size=3, strides=1, activation='relu')(x)
x = k.layers.MaxPooling2D((2, 2))(x)
x = k.layers.Conv2D(64, kernel_size=3, strides=1, activation='relu')(x)
x = k.layers.Flatten()(x)
x = k.layers.Dense(64, activation='relu')(x)
outputs = k.layers.Dense(10, activation='linear')(x)
model = k.Model(inputs, outputs)
```

the model summary is:

| Layer (type) | Output Shape | Param # |
|--------------------------------|---------------------|---------|
| input_1 (InputLayer) | [(None, 32, 32, 3)] | 0 |
| conv2d (Conv2D) | (None, 30, 30, 32) | 896 |
| max_pooling2d (MaxPooling2D) | (None, 15, 15, 32) | 0 |
| conv2d_1 (Conv2D) | (None, 13, 13, 64) | 18496 |
| max_pooling2d_1 (MaxPooling2D) | (None, 6, 6, 64) | 0 |
| conv2d_2 (Conv2D) | (None, 4, 4, 64) | 36928 |
| flatten (Flatten) | (None, 1024) | 0 |
| dense (Dense) | (None, 64) | 65600 |
| dense_1 (Dense) | (None, 10) | 650 |
| Total params: 122,570 | | |
| Trainable params: 122,570 | | |
| Non-trainable params: 0 | | |

Figure 3.33: CNN model summary for the cifar10 dataset

Finally we compiled and fitted the model with the following code:

```
model.compile(optimizer='adam',
              loss=k.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

history = model.fit(train_images, train_labels, epochs=10, batch_size=128,
                    validation_split=0.1, shuffle=False)
```

The 10 label classes are provided as integers, this is why we used as **loss function** the *SparseCategoricalCrossentropy(from_logits=True)*, the metric is the *accuracy*. An alternative was to use *CategoricalCossentropy* loss with labels in *one-hot representation*.

The argument *from_logits=True* indicates that our class predictions are logit tensors and not probability distributions. In general if the output layer has a *softmax* activation, *from_logits* should be *False*, otherwise *from_logits* should be *True*. In this case the output layer has a *linear* activation, so *from_logits* is *True*.

Float32 results

Using the configuration with the **float32** we've obtained the following results during the training phase:

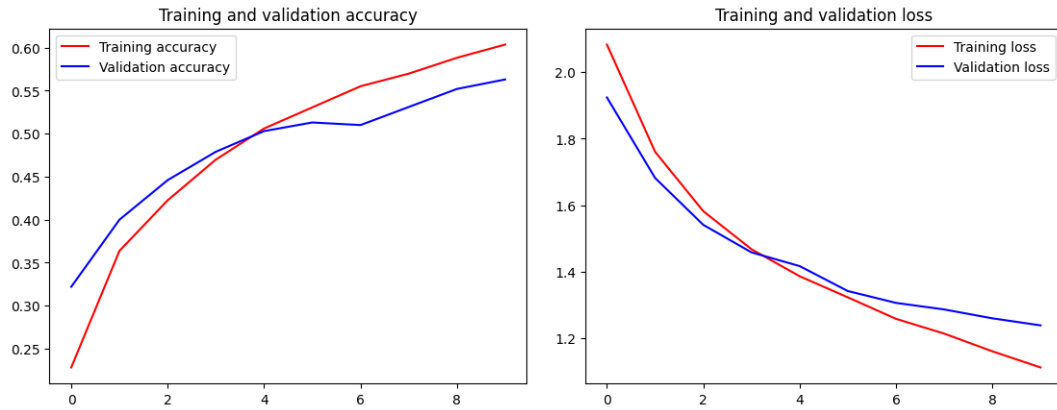


Figure 3.34: Training and validation accuracy and loss plots of the model trained using the float32

As we can see from figure 3.34 we've obtained good results for both training and validation accuracy, while the actual values achieved are presented in table 3.10.

| | accuracy | loss |
|------------|----------|--------|
| training | 0.5913 | 1.1581 |
| validation | 0.5400 | 1.2720 |
| test | 0.5496 | 1.2686 |

Table 3.10: Results of the training and test of the model

The training time of this model was very fast, taking only **4** seconds per epoch, **40** seconds in total for the 10 epochs. The model is able to learn well and reached **54%** of accuracy on the validation set and nearly **55%** of accuracy on the test set. The results are great considering we have 10 classes and we didn't use all the images in the dataset for the training.

Posit results

In order to use the posits we need to set the default float type:

```
import tensorflow.python.keras as k
k.backend.set_floatx("posit160")
```

We decided to use *kernel_initializer=random_normal_init* to initialize weights inside our Conv2D and Dense layers in this way:

```
inputs = k.Input(shape=(32, 32, 3))
x = k.layers.Conv2D(32, kernel_size=3, strides=1, activation='relu',
                    kernel_initializer=random_normal_init)(inputs)
x = k.layers.MaxPooling2D((2, 2))(x)
x = k.layers.Conv2D(64, kernel_size=3, strides=1, activation='relu',
                    kernel_initializer=random_normal_init)(x)
x = k.layers.MaxPooling2D((2, 2))(x)
x = k.layers.Conv2D(64, kernel_size=3, strides=1, activation='relu',
                    kernel_initializer=random_normal_init)(x)
x = k.layers.Flatten()(x)
x = k.layers.Dense(64, activation='relu',
                    kernel_initializer=random_normal_init)(x)
outputs = k.layers.Dense(10, activation='linear',
                          kernel_initializer=random_normal_init)(x)
model = k.Model(inputs, outputs)
```

We trained the model for 30 epochs, during this experiment we've obtained the following results:

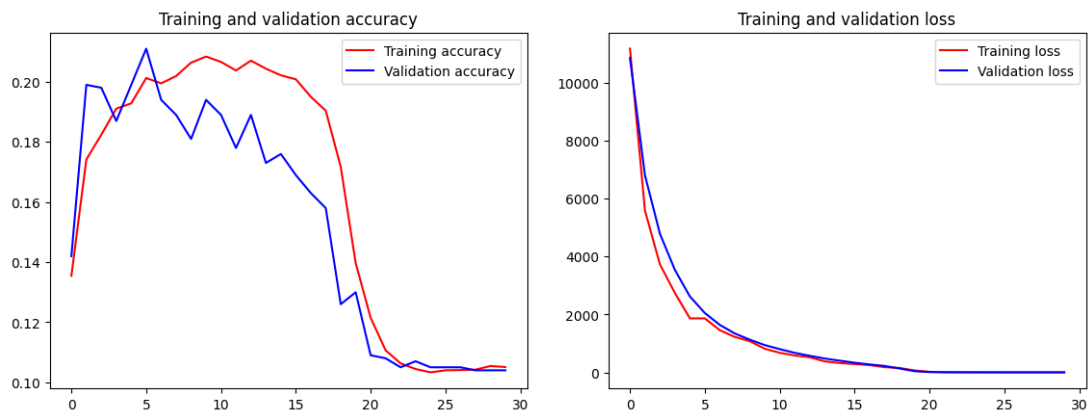


Figure 3.35: Training and validation accuracy and loss plots of the CNN model trained using the posit160

Looking at the accuracy plot can see how in the first 15 epochs the model seems to learn but after that the accuracy drops at the very low score of **10%** of accuracy. The model isn't able to generalize and performs poorly even in the training set. The loss curve on the other hand shows a great improvement with more epochs.

| | accuracy | loss |
|------------|----------|--------|
| training | 0.1051 | 2.2061 |
| validation | 0.1040 | 2.5391 |
| test | 0.0900 | 2.2793 |

Table 3.11: Results of the training and test of the model

These results are not comparable with the floats results, even the training times are considerably higher. To be sure that the *SparseCrossentropy* loss function worked correctly we made another experiment using the same model but with *CategoricalCossentropy* loss and labels in *one-hot representation*. To change the representation of our labels we used the function *to_categorical* in *tensorflow.python.keras.utils.np_utils*. We obtained the same exact results so the model with *SparseCrossentropy* loss function works correctly. We decided then to try training with more training examples to see if the performance would improve. Unfortunately, we couldn't train for many epochs since the training time is significantly slow, in this experiment it took on average **493** seconds per epoch, so in total more than **4** hours to complete the training.

Posit results with more training samples

We decided to increase the training samples to *50000* and the testing samples to *5000*. We trained for 10 epochs obtaining the following results:

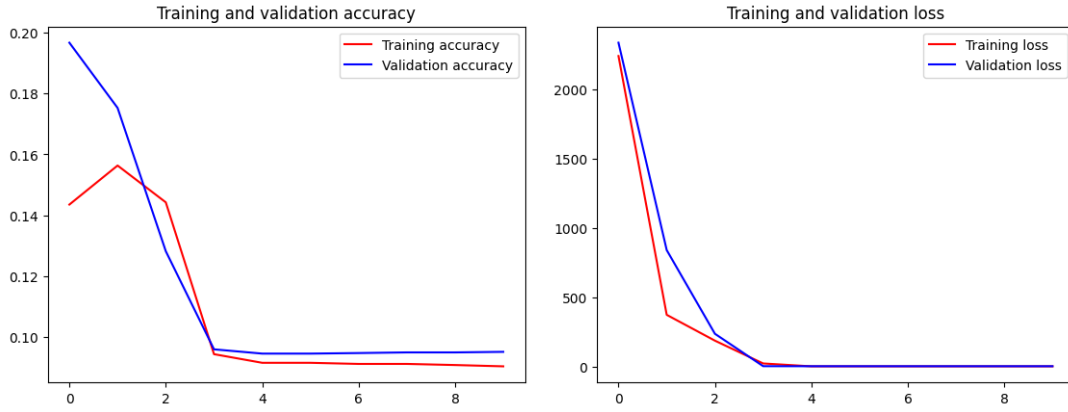


Figure 3.36: Training and validation accuracy and loss plots of the CNN model trained using the posit160

| | accuracy | loss |
|------------|----------|--------|
| training | 0.0904 | 1.4585 |
| validation | 0.0951 | 2.2627 |
| test | 0.1013 | 2.1152 |

Table 3.12: Results of the training and test of the model

The results are very similar to the ones previously obtained, this means the problem is not in the number of samples in the training set.

Second model: LeNet-5 CNN with MaxPooling2D instead of AveragePooling2D layers

We decided to use a different CNN model to see if there is a different behaviour, for this experiment we used *LeNet-5* architecture but with *MaxPooling2D* layers instead of *AveragePooling2D* layers since they didn't work with posits. The code of the model is the following:

```
inputs = k.Input(shape=(32, 32, 3))
x = k.layers.Conv2D(6, kernel_size=5, strides=1, activation='tanh',
padding='same')(inputs) #C1
x = k.layers.MaxPooling2D()(x)
x = k.layers.Conv2D(16, kernel_size=5, strides=1, activation='tanh',
padding='valid')(x) #C3
x = k.layers.MaxPooling2D()(x)
x = k.layers.Conv2D(120, kernel_size=5, strides=1, activation='tanh',
padding='valid')(x) #C5
x = k.layers.Flatten()(x) #Flatten
x = k.layers.Dense(84, activation='tanh')(x) #F6
```

```

outputs = k.layers.Dense(10, activation='softmax')(x) #Output layer

model = k.Model(inputs, outputs)

```

the model summary is:

| Layer (type) | Output Shape | Param # |
|--------------------------------|---------------------|---------|
| input_1 (InputLayer) | [(None, 32, 32, 3)] | 0 |
| conv2d (Conv2D) | (None, 32, 32, 6) | 456 |
| max_pooling2d (MaxPooling2D) | (None, 16, 16, 6) | 0 |
| conv2d_1 (Conv2D) | (None, 12, 12, 16) | 2416 |
| max_pooling2d_1 (MaxPooling2D) | (None, 6, 6, 16) | 0 |
| conv2d_2 (Conv2D) | (None, 2, 2, 120) | 48120 |
| flatten (Flatten) | (None, 480) | 0 |
| dense (Dense) | (None, 84) | 40404 |
| dense_1 (Dense) | (None, 10) | 850 |
| Total params: 92,246 | | |
| Trainable params: 92,246 | | |
| Non-trainable params: 0 | | |

Figure 3.37: CNN model summary for the cifar10 dataset

Finally we compiled and fitted the model with the following code:

```

model.compile(optimizer='adam',
              loss=k.losses.SparseCategoricalCrossentropy(from_logits=False),
              metrics=['accuracy'])

history = model.fit(train_images, train_labels, epochs=10, batch_size=128,
                    validation_split=0.1, shuffle=False)

```

We set *from_logits=False* since we have a *softmax* activation in the last *Dense* layer.

Float32 results

Using the configuration with the **float32** we've obtained the following results during the training phase:

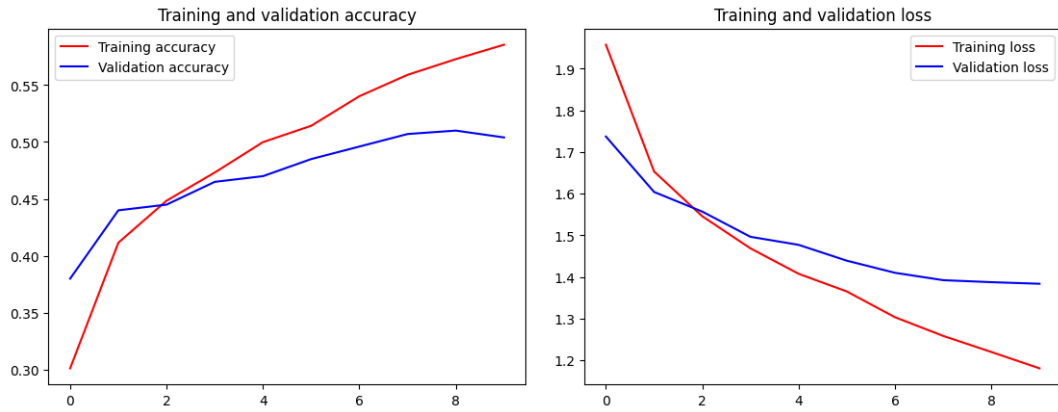


Figure 3.38: Training and validation accuracy and loss plots of the second CNN model trained using the float32

As we can see from figure 3.38 we've obtained good results for both training and validation accuracy, while the actual values achieved are presented in table 3.13.

| | accuracy | loss |
|------------|----------|--------|
| training | 0.5854 | 1.1812 |
| validation | 0.5040 | 1.3841 |
| test | 0.4840 | 1.4287 |

Table 3.13: Results of the training and test of the second CNN model

The training time of this model was very fast, taking only **2** seconds per epoch, **20** seconds in total for the 10 epochs. The model is able to learn well and reached **50%** of accuracy on the validation set and nearly **49%** of accuracy on the test set. The results are still great even if a bit worse than with the previous CNN model, which obtained **54%** of accuracy on the validation set and **55%** of accuracy on the test set.

Posit results

We trained for 30 epochs obtaining the following results:

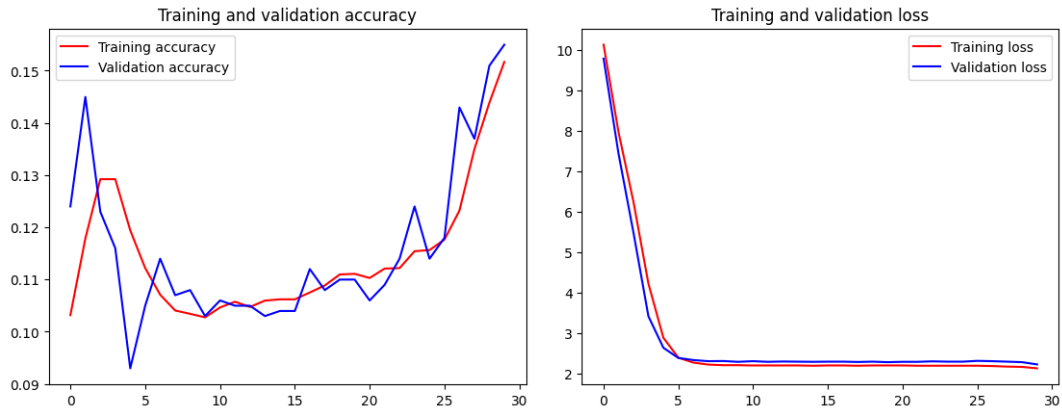


Figure 3.39: Training and validation accuracy and loss plots of the second CNN model trained using the posit160

| | accuracy | loss |
|------------|----------|--------|
| training | 0.1517 | 2.1348 |
| validation | 0.1550 | 2.2314 |
| test | 0.1540 | 2.1992 |

Table 3.14: Results of the training and test of the model

The results are still similar to the ones with the previous CNN, looking at the validation and test accuracy model still performs very poorly reaching only a **15%** of accuracy, the loss on the other hand remains good with low values (**2.2** for both validation and test loss).

Third model: simpler CNN with only one Conv2D layer

We decided to use a simpler CNN model to see if there is a problem with complex CNNs containing many Convolutional and MaxPooling layers. We used the same architecture as in the previous experiment but only removed the last two *Conv2D* and *MaxPooling2D* layers.

The code of the model is the following:

```
inputs = k.Input(shape=(32, 32, 3))
x = k.layers.Conv2D(6, kernel_size=3, strides=1, activation='tanh',
                    padding='same')(inputs)
x = k.layers.MaxPooling2D()(x)
x = k.layers.Flatten()(x)
x = k.layers.Dense(84, activation='tanh')(x)
outputs = k.layers.Dense(10, activation='softmax')(x)
model = k.Model(inputs, outputs)
```

The model summary is:

| Layer (type) | Output Shape | Param # |
|------------------------------|---------------------|---------|
| input_1 (InputLayer) | [(None, 32, 32, 3)] | 0 |
| conv2d (Conv2D) | (None, 32, 32, 6) | 168 |
| max_pooling2d (MaxPooling2D) | (None, 16, 16, 6) | 0 |
| flatten (Flatten) | (None, 1536) | 0 |
| dense (Dense) | (None, 84) | 129108 |
| dense_1 (Dense) | (None, 10) | 850 |
| Total params: 130,126 | | |
| Trainable params: 130,126 | | |
| Non-trainable params: 0 | | |

Figure 3.40: third CNN model summary for the cifar10 dataset

Finally we compiled and fitted the model with the following code:

```
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False),
              metrics=['accuracy'])
history = model.fit(train_images, train_labels, epochs=10, batch_size=128,
                    validation_split=0.1, shuffle=False)
```

We set *from_logits=False* since we have a *softmax* activation in the last *Dense* layer.

Float32 results

Using the configuration with the **float32** we've obtained the following results during the training phase:

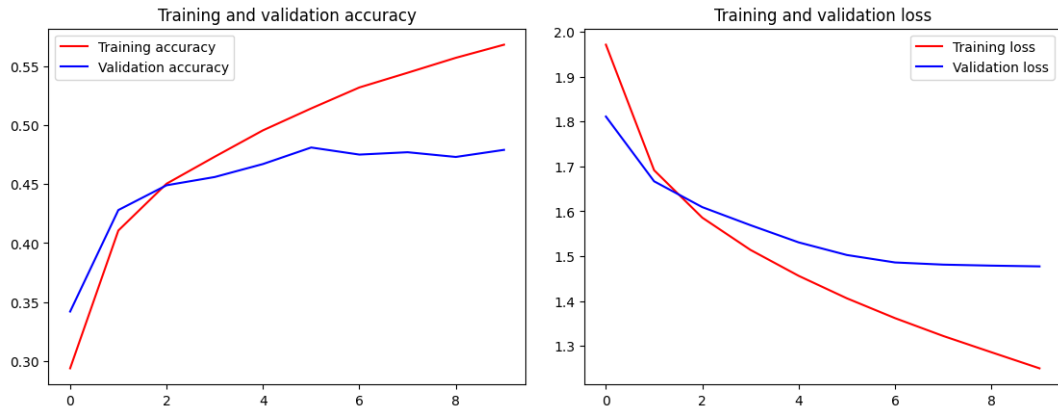


Figure 3.41: Training and validation accuracy and loss plots of the third CNN model trained using the float32

As we can see from figure 3.41 we've obtained good results for both training and validation accuracy, while the actual values achieved are presented in table 3.15.

| | accuracy | loss |
|------------|----------|--------|
| training | 0.5681 | 1.2502 |
| validation | 0.4790 | 1.4774 |
| test | 0.4580 | 1.5090 |

Table 3.15: Results of the training and test of the third CNN model

The training time of this model was very fast, taking only **1** second per epoch, **10** seconds in total for all 10 epochs. The model reached **48%** of accuracy on the validation set and nearly **46%** of accuracy on the test set. The results are comparable to the ones obtained with the second model and worse than the first one which obtained **54%** of accuracy on the validation set and **55%** of accuracy on the test set.

Posit results

We trained for 70 epochs obtaining the following results:

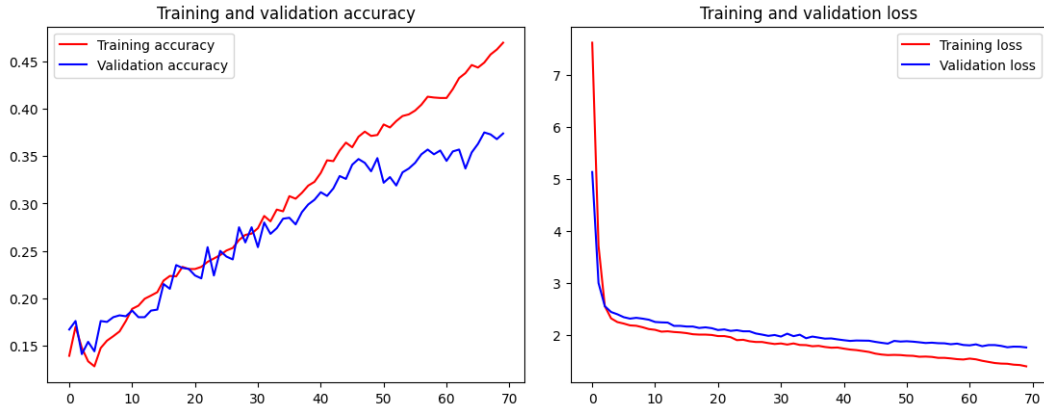


Figure 3.42: Training and validation accuracy and loss plots of the third CNN model trained using the posit160

| | accuracy | loss |
|------------|----------|--------|
| training | 0.4697 | 1.3877 |
| validation | 0.3739 | 1.7520 |
| test | 0.3860 | 1.7739 |

Table 3.16: Results of the training and test of the model

In this experiment the model finally learned correctly and has a generalization power! It reached a validation accuracy of **37%** and a test accuracy of nearly **39%**. The results are still worse than the ones obtained with float numbers, with **48%** of accuracy on the validation set and **46%** on the test set. Looking only at the loss, the model results are better, it reached **1.75** of loss on the validation set and **1.77** on the test set. With float we had very similar results, just a bit better (**1.5** of loss on both the validation and test set).

3.2.3 CIFAR10 Conclusion and summary

In conclusion after all these experiments we believe that the problem with the first model was the use of the *relu* activation function in consecutive convolutional layers, probably the weighted sum of the inputs became consistently negative and the output of the *relu* function always zero, also the gradient during backpropagation became zero and the weights weren't updated.

In the second model we believe the problem was that the use of the *tanh* activation function in consecutive convolutional layers led to the vanishing gradient problem. Consequently, the first two models performed very poorly and weren't able to have the generalization power achieved by the third model.

In the following table we can see a summary of the results obtained from the experiments on the CIFAR10 dataset and the performance in terms of time achieved on both the floats and the posit implementations.

| Model | | Val Accuracy | Val Loss | Test Accuracy | Test Loss |
|------------|-----------------|--------------|----------|---------------|-----------|
| First CNN | Float32 | 0.5400 | 1.2720 | 0.5496 | 1.2686 |
| | Posit160 | 0.1040 | 2.5391 | 0.0900 | 2.2793 |
| LeNet5 CNN | Float32 | 0.5040 | 1.3841 | 0.4840 | 1.4287 |
| | Posit160 | 0.1550 | 2.2314 | 0.1540 | 2.1992 |
| Simple CNN | Float32 | 0.4790 | 1.4774 | 0.4580 | 1.5090 |
| | Posit160 | 0.3739 | 1.7520 | 0.3860 | 1.7739 |

| Model | | N of Epochs | Avg Training Time per epoch | Tot. Training Time |
|------------|-----------------|-------------|-----------------------------|--------------------|
| First CNN | Float32 | 10 | 4 s | 40 s |
| | Posit160 | 30 | 493 s | 14790 s |
| LeNet5 CNN | Float32 | 10 | 2 s | 40 s |
| | Posit160 | 30 | 147 s | 4410 s |
| Simple CNN | Float32 | 10 | 1 s | 10 s |
| | Posit160 | 70 | 43 s | 3010 s |

3.3 Fine-tuning

We report the experiment to use a pre-trained model, in this case the VGG16 model to fine-tune a dataset containing images of cats and dogs for their image

recognition. The dataset was taken from https://storage.googleapis.com/mledu-datasets/cats_and_dogs_filtered.zip and the idea was to observe the behavior of a model that had been pre-trained with floats when working with posits. However even with very simple CNNs and few training images the training times were too long and it was impossible to continue the experiment.

Chapter 4

Conclusion and Future Work

We report a table to summarize the *Keras* layers that did and didn't work with posit numbers.

| Keras Layer | Does it work with the posit? |
|--------------------|------------------------------|
| Dense | Yes |
| Flatten | Yes |
| Dropout | Yes |
| Conv1D | Yes |
| MaxPooling2D | Yes |
| AveragePooling2D | No |
| Conv2D | Yes |
| BatchNormalization | Yes |
| RandomFlip | No |
| RandomRotation | No |
| RandomZoom | No |

In particular for the layers that didn't work we encountered the error that posits data type wasn't registered to the list of registered data types in some operations used in these layers, for example in the *AvgPool* operation in the *AveragePooling* layer.

We also believe there is an error in the implementation of the function `random_ops.random_uniform` from the `tensorflow.python.ops` module, present in the following link:

<https://github.com/federicorossifr/tensorposit/blob/master/tensorflow/tensorflow/core/ops/random>

This is probably the reason we had problems with *Conv2D* and *Dense* layers with the default value of `kernel_initializer="glorot_uniform"`, and the reason why *Dropout* layer couldn't work before the modifications we made as explained before.