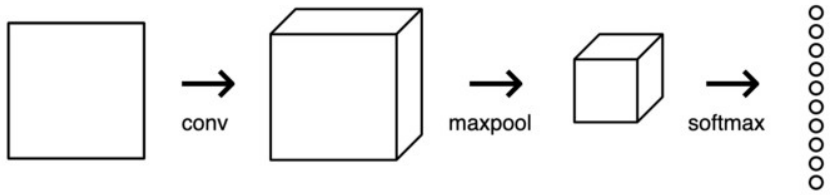
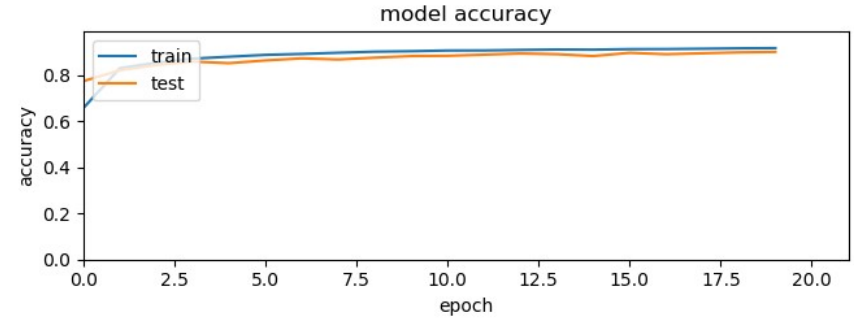
Reconfigurable logic based acceleration of convolutional neural network training

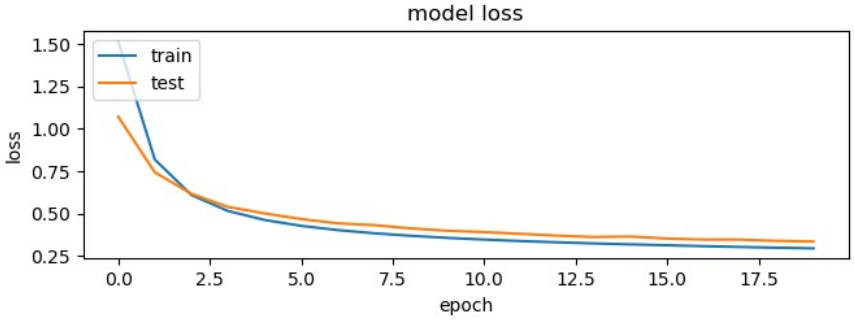
Model description:

The current network for CNN training with Mini-Batch Gradient Descent is developed on pure python using the NumPy library and trained with the MNIST data set. It consists of a Conv layer with 12 3x3 filters initialized with He initialization using Relu as activation function, a Maxpool layer with pool size 2, and a fully connected layer using Softmax as activation function. To measure the accuracy of the network, the Categorical Cross-Entropy Loss function is used. The training parameters that were used for the results presented are: batch size= 128, learning rate=0.01, 10000 training samples, and 1000 test samples.

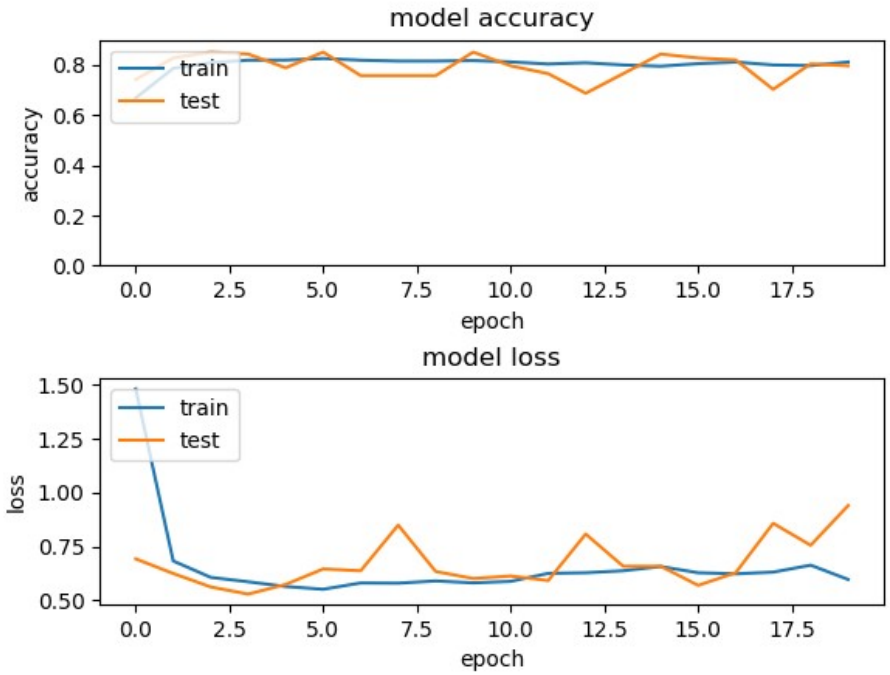


Using the network described a Keras model was implemented so comparisons could be made. The run of that model gave the following results for a 20 epochs test:





Running the same network on the NumPy based implementation, the results are the following:



We observe similar behavior between the 2 implementations. While the training is close to the same results with slightly worst accuracy and loss, the testing results appear to be a bit unstable. It's worth noting that the NumPy version can give a bit better results in case we use batch size 100, but it is common for power of 2 batch sizes to offer better runtime so 128 was the one used to execute this example.

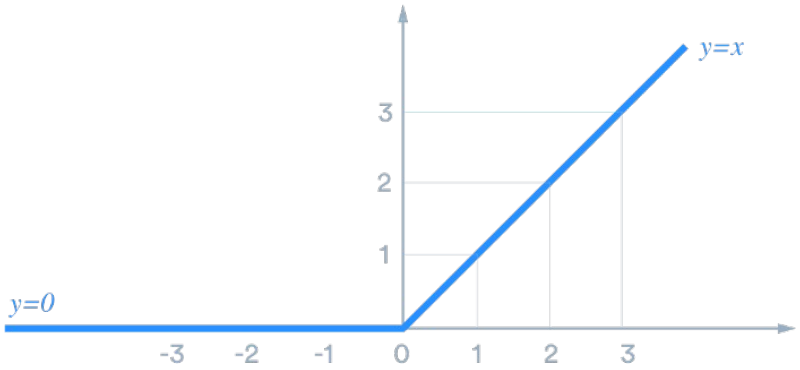
Conv Layer:

The first layer of the network is a Conv layer with 12 3x3 filters. The input of this layer is four-dimensional tensor with shape [batch, channel, height, width]=[batch, 1, 28, 28] and the output is [batch,12,26,26]. Once the Convolution is completed, the output goes through the ReLu activation function and the result is forwarded to the next layer.

On this layer, a convolution is applied to small regions of an image, sampling the values of pixels in this region, and converting it into a single pixel. It is applied to each region of pixels in the image, to produce a new image. The idea is that pixels in the new image incorporate information about the surrounding pixels, thus reflecting how well a feature is represented in that area.

Using a 12 3x3 filters means that we will have a total of 108 weights, which alongside with biases, are used for classification during the forward phase and are updated during the backpropagation phase. These weights are initialized using the He initialization. This initialization technique was used to counter the vanishing/exploding weights problem, which resulted in the network's performance slowly declining to 0. Another positive effect was that since normal distribution was replaced by the He initialization, the network results were improved after the 2nd epoch for around 50 % accuracy to 75%.

ReLU is the most commonly used activation function in neural networks, especially in CNNs. ReLU stands for rectified linear unit, it is defined as $y=max(0,x)$. Visually, it looks like the following:



It's cheap to compute as there is no complicated math. The model can therefore take less time to train or run.

Maxpool Layer:

The convolutional layers aren't supposed to reduce the size of the image significantly. Instead, they make sure that each pixel reflects its neighbors. This makes it possible to perform downscaling, through pooling, without losing important information.

A widespread method to do so is max pooling, in other words using the maximum value from a cluster of neurons at a previous layer. Indeed, max-pooling layers have a size and a width. Unlike convolution layers, they are applied to the 2-dimensional depth slices of the image, so the resulting image is of the same depth, just of a smaller width and height by dividing them by the pool size. The presented network uses pool size 2. As a result, the output of the Maxpool Layer is [batch,12,13,13].

Fully Connected Layer:

The fully-connected layer is a combination of a flattening layer and a dense layer using Softmax as the activation function. The input is flattened into a feature vector and passed through a network of neurons to predict the output probabilities.

The rows are concatenated to form a long feature vector. The feature vector is then passed through the dense layer and is multiplied by the layer's weights, summed with its biases, and passed through a 10 nodes Softmax, each representing each digit since we are working with MNIST. This layer will have 12*13*13 = 2028 weights connected to each node, so a total of 20280 weights, which are initialized with the He initialization as well.

Softmax turns arbitrary real values into probabilities. The math behind it is pretty simple: given some numbers,

1. Raise e to the power of each of those numbers.
2. Sum up all the exponential. This result is the *denominator*.
3. Use each number's exponential as its *numerator*.

$$s(x_i)=\frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_i}}$$

After the softmax transformation is applied, the digit represented by the node with the highest probability will be the output of the CNN!

Loss Function:

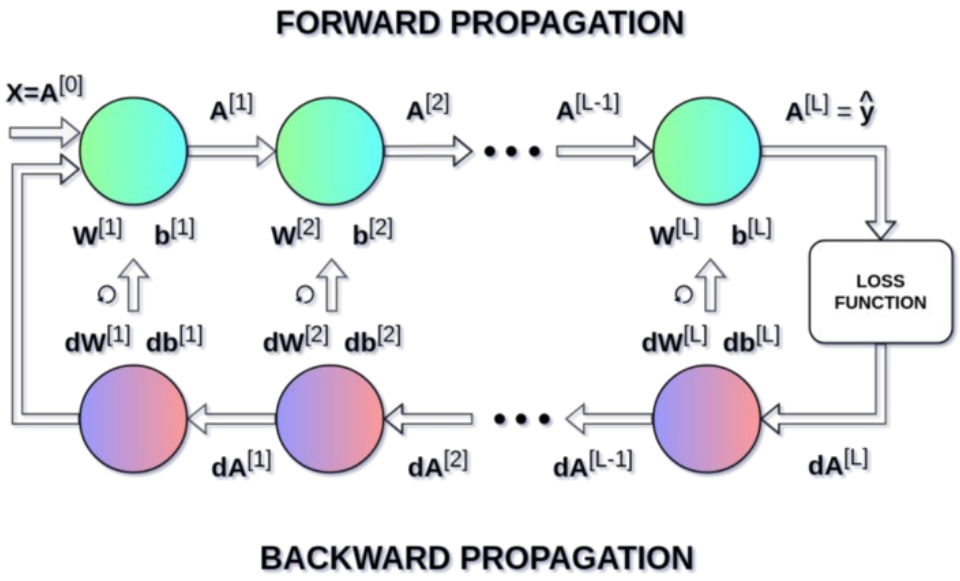
Once the forward part of the CNN is done executed, we have to check how accurate the results were. A loss function has to be implemented to compare the results with the label of each picture that went through the network. A common loss function to use when predicting multiple output classes is the Categorical Cross-Entropy Loss function, defined as follows:

$$H(y,\hat{y})=\sum_i y_i \log \frac{1}{\hat{y}_i}=-\sum_i y_i \log \hat{y}_i$$

ŷ: CNN's prediction, **y**: the desired output label.
Since we are working with batches, we need to make predictions over multiple examples, so the average of the loss over all examples needs to be calculated.

Backpropagation:

When we use a feed-forward neural network to accept an input x and produce an output ŷ, information flows forward through the network. The inputs x provide the initial information that then propagates up to the hidden units at each layer and finally produces ŷ. This is called forward propagation. During training, forward propagation can continue onward until it produces a scalar cost J(θ). The back-propagation algorithm, often simply called backprop, allows the information from the cost to then flow backward through the network, to compute the gradient. Computing an analytical expression for the gradients straight forward, but numerically evaluating such an expression can be computationally expensive. The back-propagation algorithm does so using a simple and inexpensive procedure.



The parameters of the neural network are adjusted according to the following formulae:

$$W^{[l]}=W^{[l]}-a\,dW^{[l]}$$
$$b^{[l]}=b^{[l]}-a\,db^{[l]}$$

a represents learning rate, which allows us to control the value of the performed adjustment. A low learning rate can result in a very slow learning network and a high one can result in to not be able to hit the minimum. The parameters **dW** and **db** are calculated using the chain rule, partial derivatives of loss function with respect to **W**, and **b**. The size of **dW** and **db** are the same as that of **W** and **b** respectively. These variables are calculated following the formulas:

$$dZ^{[l]} = dA^{[l]} * g'(Z^{[l]})$$

$$dW^{[l]} = \frac{dL}{dW^{[l]}} = \frac{1}{m} dZ^{[l]} A^{[l-1]T}$$

$$db^{[l]} = \frac{dL}{db^{[l]}} = \frac{1}{m} \sum_{i=1}^{m} dZ^{[l](i)}$$

$$dA^{[l]} = \frac{dL}{dA^{[l-1]}} = W^{[l]T} dZ^{[l]}$$

Z: output of a layer
A: activation output of the corresponding layer
m: number of examples from the training set
g': derivative of the non-linear activation function

The initial gradient that is passed back to the FC layer comes from the computation of the loss function. It is multiplied with the derivative of the Softmax function and the result is used to calculate the dW and db of the FC layer along with the gradient that will be passed to the Maxpool layer. Before this gradient is passed backward, it first needs to be unflatten - reshaped back to its original dimensions

A Maxpool layer can't be trained because it doesn't have any weights, but we still need to implement a method for it to calculate gradients. During the forward pass, the Max Pooling layer takes an input volume and halves its width and height dimensions by picking the max values over 2x2 blocks. The backward pass does the opposite: we'll double the width and height of the loss gradient by assigning each gradient value to where the original max value was in its corresponding 2x2 block. Each gradient value is assigned to where the original max value was, and every other value is zero. The result is being passed backward to the Conv layer.

Conv layer is working similarly to the FC layer. The gradient is first multiplied with the derivative of the ReLu function and the result is used to calculate the dW and db of the FC layer along with the gradient that will be passed to the next layer.

Precision Testing:

An interesting point of discussion whether or not by using a lower precision point, we get worse results. A lot of research is focused on using low floating point precision on both training and inference.

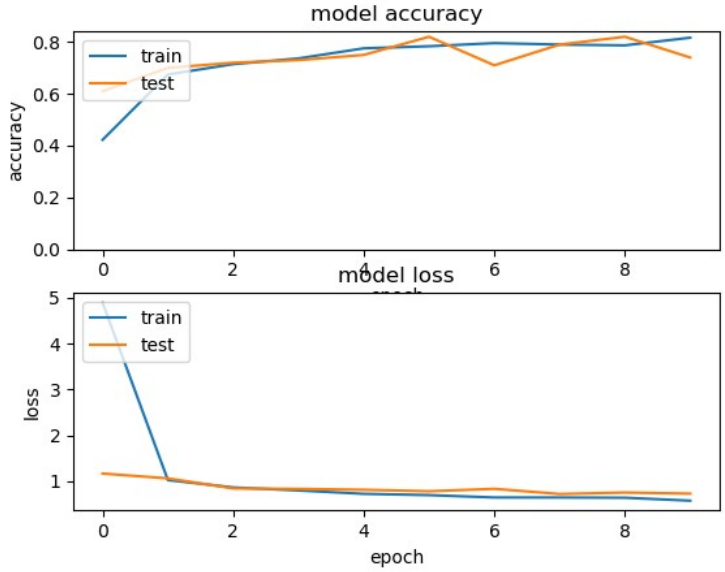
Using the NumPy made network, a series of tests were run for different combinations of floating-point. By default, NumPy is using 64-bit float numbers and it can support down to 16 bit. The network that was used for these experiments was identical to the one that was described previously, but with the exception that the batch size was 100 and the number of training samples 2000 and test samples 500.

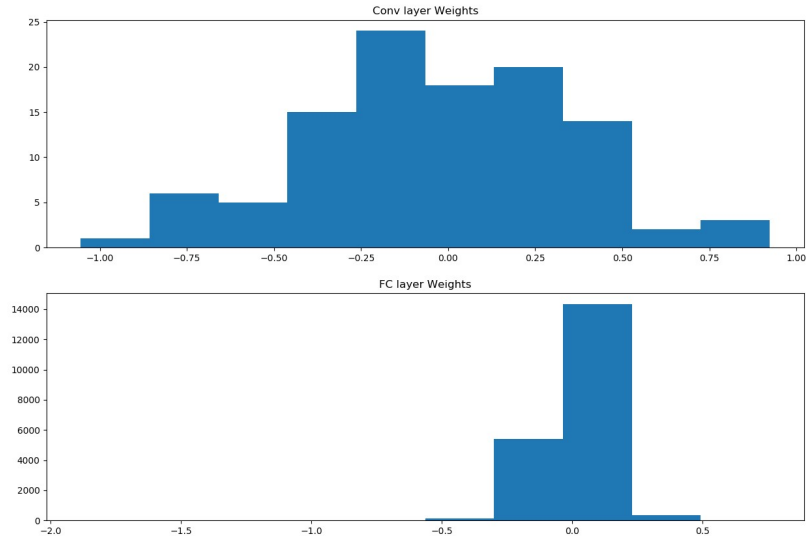
The following table represents a review of the results:

Precision Used	Time	1st Epoch Train	1st Epoch Test	5th Epoch Train	5th Epoch Test	10th Epoch Train	10th Epoch Test
fw32bw32	61.89min	Loss:4.89 acc:0.42	Loss:1.17 acc:0.61	Loss:0.70 acc:0.78	Loss:0.78 acc:0.82	Loss:0.58 acc:0.82	Loss:0.73 acc:0.74
fw16bw32	62.27 min	Loss:4.46 acc:0.40	Loss:1.04 acc:0.68	Loss:0.73 acc:0.77	Loss:0.73 acc:0.83	Loss:0.54 acc:0.84	Loss:0.64 acc:0.77
fw16bw16	57.02 min	Loss:4.00 acc:0.45	Loss:0.96 acc:0.72	Loss:0.67 acc:0.79	Loss:0.60 acc:0.85	Loss:0.51 acc:0.83	Loss:0.51 acc:0.86

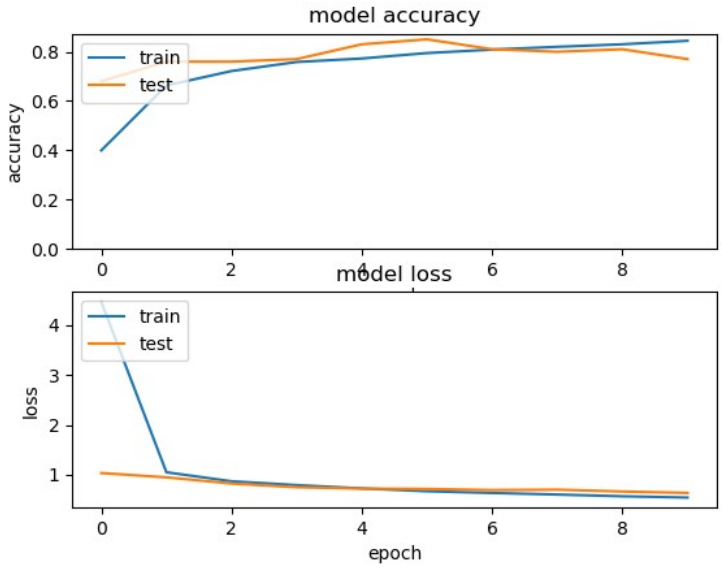
As we can see the results remain close to each other, but the execution time is a bit better. The pictures following will give us a better overview of this as they represent the accuracy and loss of the network over each epoch but also for every run we get to see the distribution of the weights.

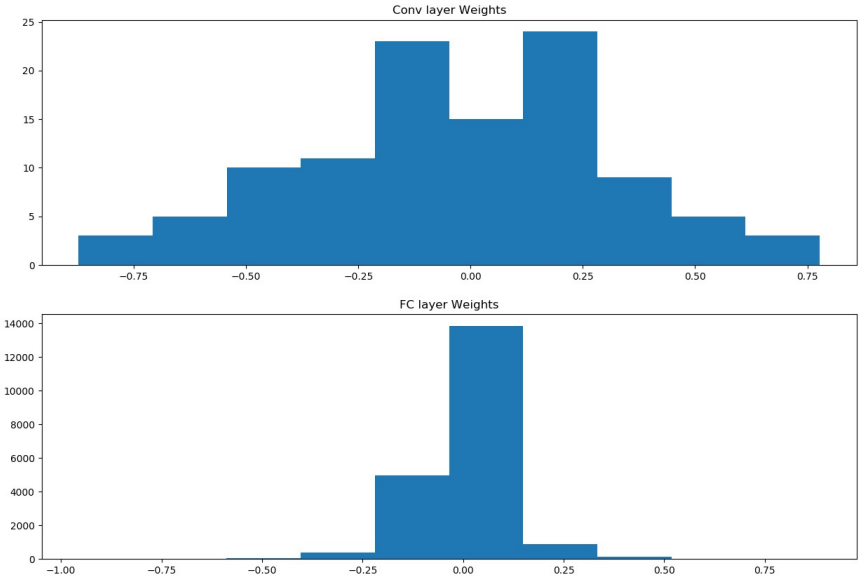
32 bits float forward and backward





16 bits float forward and 32 bits backward





16 bits float forward and backward