Reconfigurable Logic Based Acceleration of Convolutional Neural Network

Model description:

The current network for CNN training with Mini-Batch Gradient Descent is developed on C and trained with the MNIST data set. It consists of a Convolutional layer with 12 3×3 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= 4, learning rate=0.01, momentum 0.9, decay 0.00005, epoch 20 and iterations 2000.

Conv Layer:

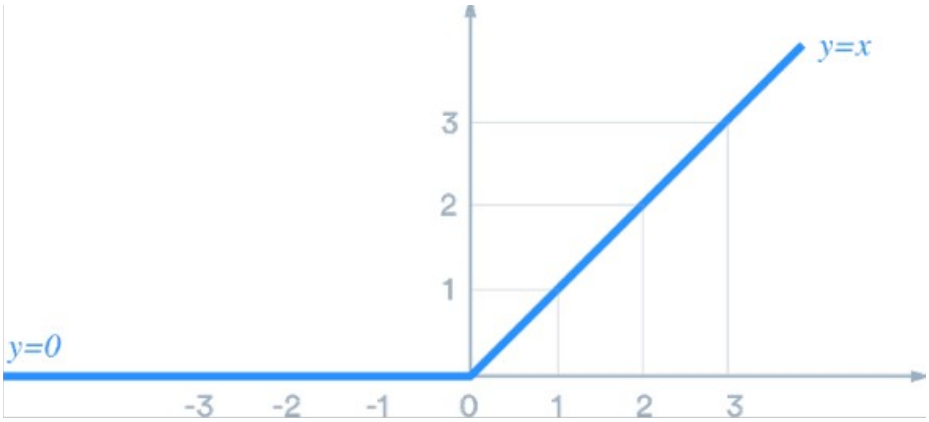
he first layer of the network is a Conv layer with 12 3×3 filters. The input of this layer is four-dimensional tensor with shape [batch, channel, height, width] = [batch, 1, 28, 28] and the output is [batch,12,26,26]. Once the Convolution is completed, the output goes through the ReLu activation function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

The initial design of the NumPy code was using the naïve implementation for a convolutional 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.

Unfortunately, this implementation is extremely slow. Matrix multiplication, or matmul, or Generalized Matrix Multiplication (GEMM), is at the heart of deep learning. It’s used in fully-connected layers, RNNs, etc., and can be used to implement convolutions too. Conv is, after all, a dot-product of the filter with input patches. If we lay out the filter into a 2-D matrix and the input patches in another, then the multiplying these 2 matrices would compute the same dot product. This laying out of the image patches into a matrix is called im2col, for image to column. We rearrange the image into columns of a matrix, so that each column corresponds to one patch where the conv filter is applied.

Using a 12 3×3 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

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:



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:

Fully Connected layers in neural networks are those layers where all the inputs from one layer are connected to every activation unit of the next layer. In most popular machine learning models, the last few layers are full connected layers that compile the data extracted by previous layers to form the final output. It is the second most time-consuming layer second to the Convolution Layer. Neurons in a fully connected layer have full connections to all activations in the previous layer, as seen in regular Neural Networks. Their activations can hence be computed with a matrix multiplication followed by a bias offset. 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:

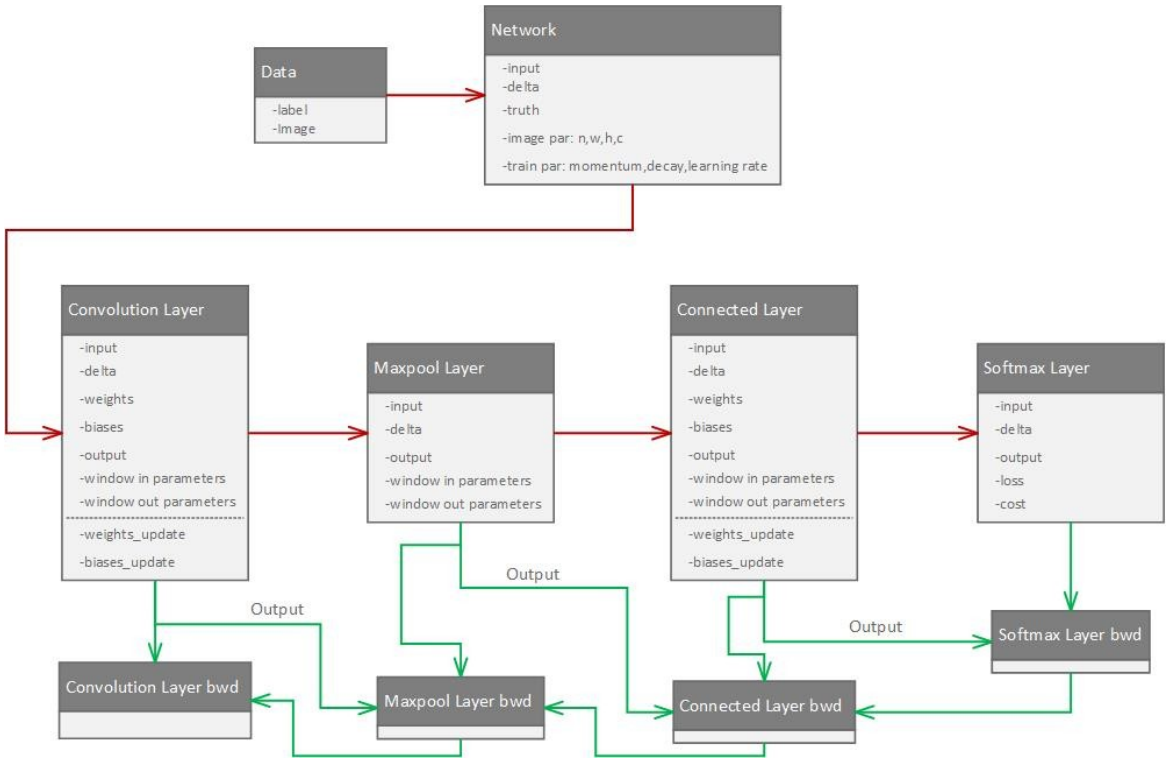
$$H(y, \hat{y}) = \sum_i y_i \log \frac{1}{\hat{y}_i} = -\sum_i y_i \log \hat{y}_i$$

$\hat{\mathbf{y}}$: CNN's prediction, $\mathbf{y}$: 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 $\hat{\mathbf{y}}$, information flows forward through the network. The inputs x provides 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 use 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:

$$dZ^{[l]} = dA^{[l]} * g'\left(Z^{[l]}\right)$$
$$dW^{[l]} = \frac{dL}{dW^{[l]}} = \frac{1}{m} dZ^{[l]} A^{[l-1]T}$$
$$db^{[l]} = \frac{dL}{db^{[l]}} = \frac{1}{m} \sum_{i=1}^{m} dZ^{[l](i)}$$
$$dA^{[l]} = \frac{dL}{dA^{[l-1]}} = W^{[l]T} dZ^{[l]}$$

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

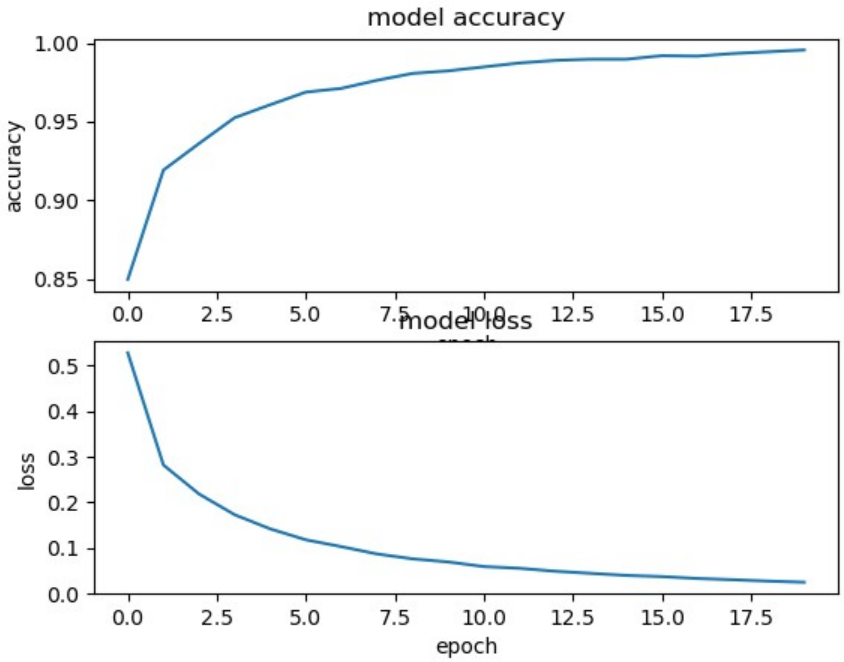
The initial gradient that is passed back to the FC layer comes from the computation of the loss function. It is multiplied with the derivative of the SoftMax function and the result is used to calculate the dW and db of the FC layer along with the gradient that will be passed to the Maxpool layer.

A Maxpool layer can't be trained, since 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.

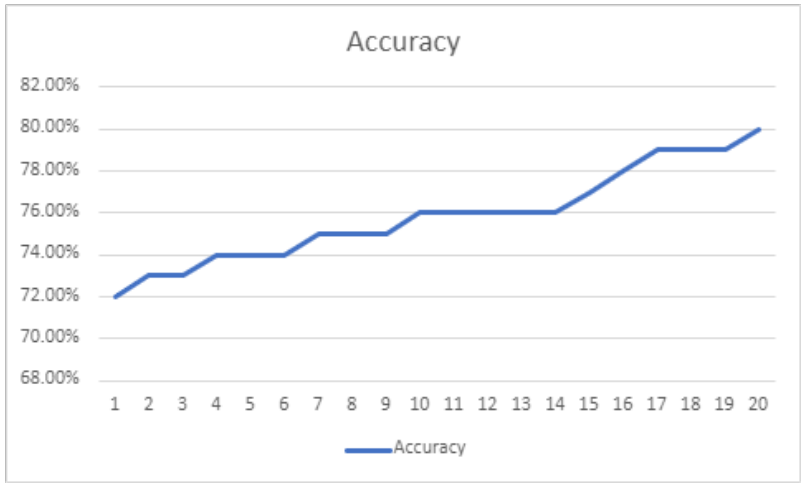
Conv layer is working similarly to the FC layer. The gradient is first multiplied with the derivative of the ReLu function and the result is used to calculate the dW and db of the conv layer along with the gradient that will be passed to the next layer.

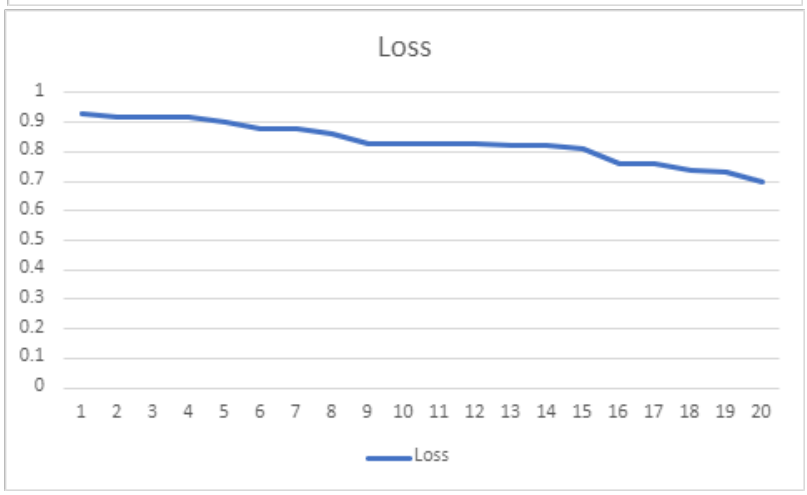
Results:

Initially, the model has been developed using Keras and then Python using NumPy. The results of the Keras model are the following:



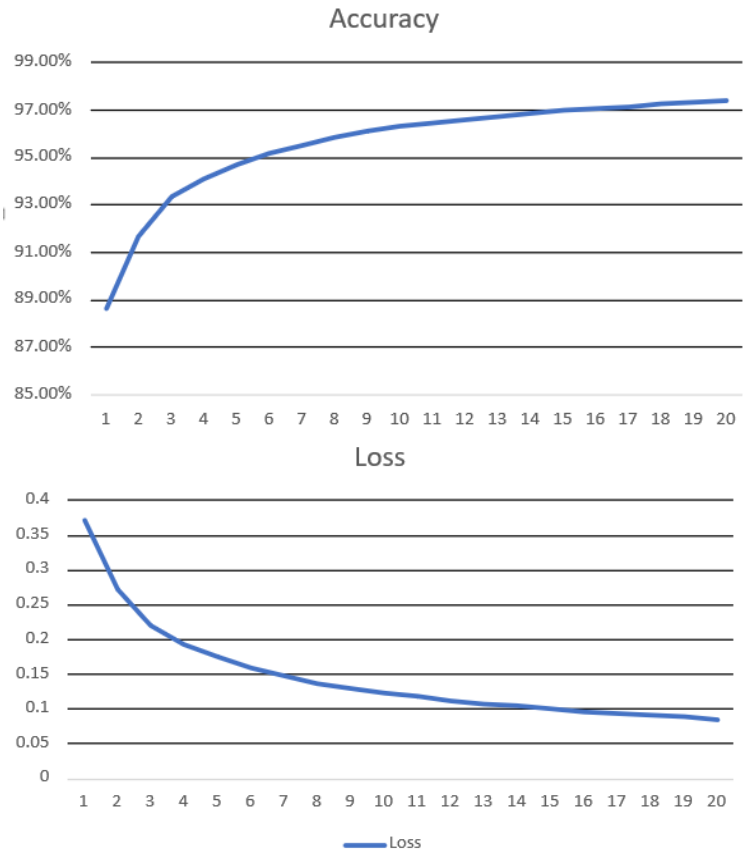
Running the same network on the NumPy based implementation, the results are the following:



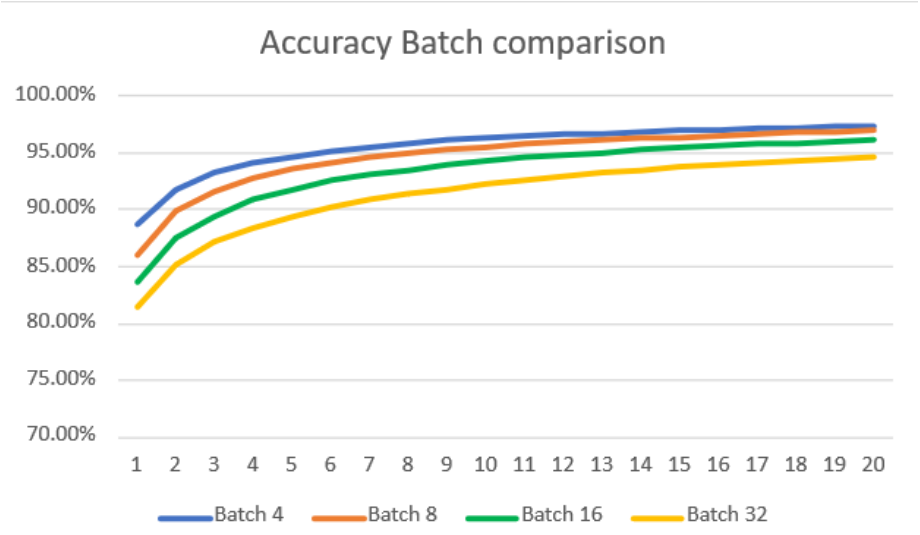


We notice that the NumPy implementation is not good enough, but it has proven through extended testing that it can achieve close to Keras model results. The major downside of this model as stated earlier though is the execution time, which was 23202 seconds. In comparison the C model executes the same network in 682 seconds.

The results produced by the C code are the following:



In the following figure, some comparisons between different batch size executions are presented, followed by a table with the execution time of each example in seconds:



Batch	Batch 8	Batch 16	Batch 32
682	642	623	615

Numpy vs Keras:

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 for 20 epochs. This test was part of the early development before the C code.

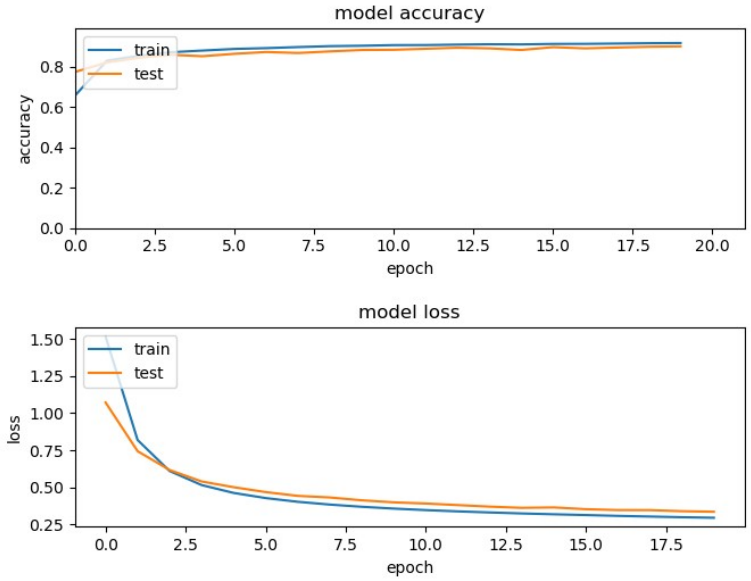


Figure 1: Keras

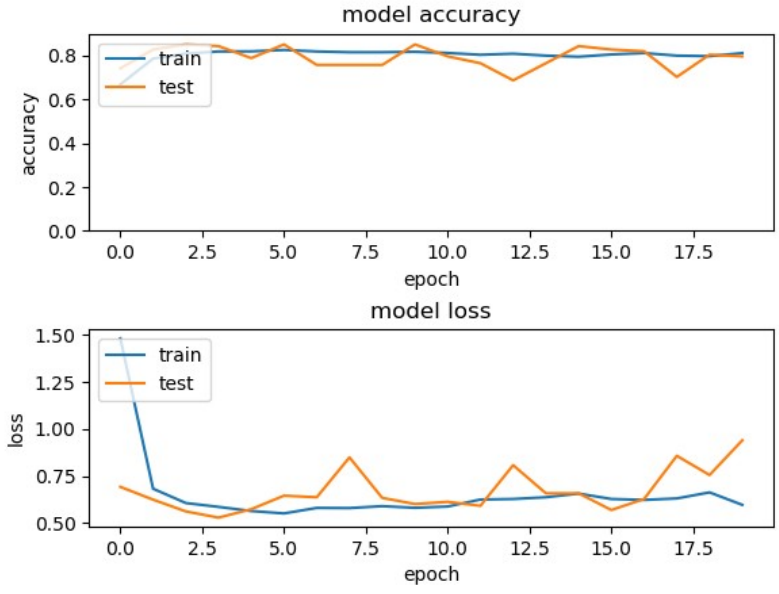


Figure 2: NumPy

Furthermore, some smaller examples were executed on the NumPy to compare the precision point results. By default, NumPy is using 64-bit float numbers and it can support down to 16 bits. 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

LeNet:

To further test the proper function of the C coded layers that can be used to create various networks, a LeNet-4 architecture has been implemented, although it has some small variations compared to the standard model. The network is consisted of 6 layers:

1. Convolution Layer with 4 filters sized 5×5, padding=0 and stride=1
2. MaxPooling Layer with size 2×2, padding=0 and stride=2
3. Convolution Layer with 16 filters sized 5×5, padding=0 and stride=1
4. MaxPooling Layer with size 2×2, padding=0 and stride=2
5. Fully Connected Layer with output size of 120
6. Fully Connected Layer with output size of 10
7. SoftMax function for results prediction

This network will have a total of 51,050 trainable parameters and 292,466 connections overall.