PiCNN

C++ Convolutional Neural Network Library for Raspberry Pi

Philip Salmony
University of Cambridge
pms67@cam.ac.uk

Summary

PiCNN is a fast, lightweight, single-header, easy-to-use, and fully parallelisable convolutional neural network library written in C++ specifically designed for the RaspberryPi and its limited computational capabilities compared to conventional desktop computers. It enables practical machine learning in RaspberryPi programming projects, without the hassle, size, and computational requirements of installing and running larger machine learning frameworks. The concept, derivation, and future improvements of PiCNN are illustrated in depth in this report.

Contents

1 Introduction

1.1 Why PiCNN?

The idea for PiCNN originated from an Undergraduated Research Project (UROP) at the University of Cambridge's Computer Laboratory. The basis for it was to see if parallel training of a convolutional neural network across multiple Raspberry Pis was feasible and provided any performance benefits compared to training a network on a single, faster machine.

After further research, it was evident that this had not been attempted before, nor was there a suitable neural network library that could provide a hassle-free installation and run smoothly on a RaspberryPi. This initiated the development of a standalone, single-header, and light-weight convolutional neural network library for the Raspberry Pi written in C++.

The main framework has been developed, however several improvements - mainly the ability to train in parallel across a network - are to be made. This report details the features, performance, mathematical derivation, and future ideas for this library. After initial testing, the benefits of this small, portable, and efficient library compared to larger alternatives are already clear.

1.2 Features

PiCNN in its current state has the following benefits and features:

- Multithreading:
  - Feature maps are trained in parallel on separate threads to maximise single-core performance
- Free choice of network structure:
  - Layers can be added in any order, and as many as physically possible. Furthermore, the cost function can be chosen (MSE, Cross Entropy, KL Divergence), as well as the activation function per layer (ReLU, TanH, etc.).
- Low-memory usage and low-CPU usage:
  - For example, training with the MNIST dataset requires only 10MB of memory and utilises 25% of the Raspberry Pi's CPU
- Single-header, light-weight, and fast implementation in other projects:
  - Incorporation into other projects requires only the inclusion of one header file (PiCNN.h). The total size of all files is less than 35kB!
- Compiles quickly on RaspberryPi:
  - PiCNN only uses the C++ standard libraries and has a very small size on disk - this enables very rapid compilation, especially compared to other machine learning libraries.
- Includes datahandling features:
  - Such as data import (from CSV), normalisation (min-max), one-hot conversion, etc.

1.3 Alternatives: TensorFlow and Tiny-DNN

One of the biggest, most-used machine learning libraries - especially for neural networks - is TensorFlow. TensorFlow works best on larger machines with GPUs. However, TensorFlow can be installed on a Raspberry Pi using several tricks (see: https://github.com/samjabrahams/tensorflow-on-raspberry-pi). Firstly, this requires a lot of patience and uses a lot of the Raspberry Pi's resources. Secondly, training models using TensorFlow on such a low-power machine is time-consuming and does not lead to satisfactory results - even if it is possible.

Another option is to use other and smaller machine learning libraries such as Tiny-DNN (see: https://github.com/tiny-dnn/tiny-dnn). However, even this substantially smaller library is still not optimised for Raspberry Pis, as it is simply a smaller framework designed to work on normal desktop machines. After testing, not only the performance, but also simply the installation and compilation of the Tiny-DNN files is subpar and tedious.

Therefore, after exploring these options mentioned above, PiCNN proves to be a useful and easy-to-install library for neural network training and inference on single-board computers such as the Raspberry Pi.

2 Mathematical Derivation

The following section provides a detailed explanation of the mathematical derivation of all parts of the convolutional neural network.

2.1 Backpropagation

Training the neural network involves comparing the actual output after a forward pass to the desired outputs. This error is quantified by the means of a cost function (see section 2.4. below). This error is then fed back through the neural network starting from the output layer and ending at the input layer. This is achieved in theory and in practice with partial derivatives and application of the chain rule.
After the backward pass, the backpropagated errors are used to adjust the weights and biases in the network in such a way as to minimise the the overall error at the output. This is done by employing optimisation techniques discussed in section 2.3 below.

2.2 Notation

To aid understanding, Table 1 below shows the notation used in all mathematical derivations.

Symbol	Meaning
C	Cost function
$y^l_{m,n}$	Output of layer l at row m, column n
$w^l_{m,n}$	Weight of layer l at row m, column n
$b^l_{m,n}$	Bias of layer l at row m, column n
$\sigma(x)$	Activation function evaluated at x
$\delta^l_{m,n}$	'Delta': $\frac{\partial C}{\partial y^l_{m,n}}$. Rate of change of cost function w.r.t. output of layer at row m, column n
η	Learning rate
μ	Momentum coefficient
y^L_m,t^L_m	Output at final layer of m$^{\text{th}}$neuron, Target value at final layer of m$^{\text{th}}$neuron

Table 1: Notation

2.3 Optimisation (Stochastic Gradient Descent)

Backpropagation gives us an expression for the rate of change of the cost function w.r.t. a layer's outputs (∇C_y), i.e. the partial derivative $\frac{\partial C}{\partial y^l_i}$ (also known as the delta at that output).
Again, using the chain rule, we can determine expressions for the rate of change of the cost function w.r.t. weights (∇C_w) and biases (∇C_b) in the network. These partial derivatives are then used to adjust the current values of the weights and biases such as to minimise the cost function.

A popular method of achieving this is by gradient descent: The gradient of a function gives us the direction of steepest ascent, therefore to minimise the cost function, we need to move in the opposite direction to the gradient.

The weight update rule can be expressed mathematically as follows:

$$w_{new} = w_{old} - \eta \cdot \frac{\partial C}{\partial w}$$

Where η is the learning rate, which determines the step size taken in direction of the gradient. A too large value for η could cause the weight update to 'jump' over the minimum cost, whereas a too small value for η could result in a long time for convergence.

An additional technique employed in PiCNN is called momentum (or inertia), which enables increased performance and faster localisation of optima:

$$w_{new} = w_{old} - \left(\eta \cdot \frac{\partial C}{\partial w} + \mu \cdot \Delta w\right)$$

Where Δw is the change in weight and μ is the momentum constant.

2.4 Cost Function

The cost function is a measure of error between the desired outputs (or targets) and the actual outputs present after a forward pass at the output layer of the neural network. The overall aim

in training the neural network is to minimise the cost function, thus decreasing the overall error present at the outputs.

A typical performance measure, or measure of error, is the mean squared error (MSE). It is used as one of the most basic cost functions in many neural networks and is also used as the cost function in PiCNN. It has the following form:

$$C_{MSE} = \frac{1}{2 \cdot N_1 \cdot N_2} \sum_{i=1}^{N_1} \sum_{j=1}^{N_2} \left(y_{i,j}^L - t_{i,j}^L\right)^2$$

For backpropagation it is necessary to know the partial derivative describing how the cost function changes with respect to the individual outputs of the neural network's final layer. This is commonly referred to as a 'delta' (δ).

$$\frac{\partial C}{\partial y_{i,j}^L} = \delta_{i,j}^L = \frac{1}{N_1 \cdot N_2} \cdot \left(y_{i,j}^L - t_{i,j}^L\right)$$

2.5 Activation Function

Activation functions are the origin of non-linearities in the neural network and enable the network to learn non-linear mappings. The benefits and disadvantages of many activation functions have been discussed in the literature, however PiCNN only utilises the most suitable two activation functions Tanh and ReLU. The justification for this and their properties are described in further detail below.

2.5.1 Tanh

As an example, the fully connected layers can use the hyperbolic tangent as the activation function. In comparison, the traditional Sigmoid activation function saturates too easily for relatively small input activation. Furthermore, input values are mapped to values between 0 and 1, whereas the hyperbolic tangent maps inputs to a range between -1 and +1 with a linear region about its center. The function and its derivative is defined below as follows:

$$\sigma(x) = tanh(x) = \frac{2}{1 + e^{-2x}} - 1$$

$$\sigma'(x) = 1 - \sigma^2(x)$$

2.5.2 ReLU

The convolutional layers traditionally use the ReLU activation function, as it provides sparse activation, efficient gradient propagation, efficient computation, as well as being scale-invariant. The function and its derivative is defined below as follows:

$$\sigma(x) = \begin{cases} x, & x > 0 \\ 0, & otherwise \end{cases}$$

$$\sigma'(x) = \begin{cases} 1, & x > 0 \\ 0, & otherwise \end{cases}$$

2.6 Tensors

PiCNN uses tensors as its main datatype. Tensors can be seen as 3D volumes, or rather as a collection of 2D matrices. The number of 2D matrices in a tensor determines its dimension. Furthermore, a collection of tensors is known as a TensorArray in PiCNN.

2.7 Numerical Gradients ('Sanity Check')

To check the validity of the mathematical derivation in sections described below, a first order numerical gradient check by employing finite differences is used. The network is fedforward once and the overall cost evaluated. Then a single weight in the network is adjust by a small amount h and the cost evaluated again after a forward pass. The following finite difference formula is used to approximate the derivative:

$$\frac{\partial F(x_1, x_2, ..., x_n)}{\partial x_i} \approx \frac{F(x_1, x_2, ..., x_i + h, ..., x_n) - F(x_1, x_2, ..., x_n)}{h}$$

Doing this across all weights in the network leads to knowing if the mathematical derivation via backpropagation is indeed correct.

2.8 Input Layer

The input layer takes a n-dimensional tensor, for example a 3-channel RGB image stored as double values from 0 to 255, and maps it to the dimension of the feature maps selected for the network

2.9 Convolutional Layer

The convolutional layer convolves the input tensor with a weight matrix to extract distinctive features. As opposed to fully connected layers, convolutional layers employ weight-sharing and therefore drastically reduce the number of learnable parameters and with that the computational cost. The convolutional layers in PiCNN perform 'valid' convolutions without applying zero-padding to the input.

2.9.1 Output

The output of each point is determined by the 2D convolution of the weight matrix with the input matrix added to a scalar bias term which is then passed through an activation function. This operation is performed across all dimensions of the input tensor, with different weight matrices and bias terms for each feature map.

$$y^l_{m,n} = \sigma\left(\sum_j \sum_i y^{l-1}_{i,j} \cdot w^l_{m-i,n-j} + b^l\right) = \sigma\left(z^l_{m,n}\right)$$

Where the activation function, chosen to be ReLU, and its derivative is:

$$\sigma(z^l_{m,n}) = \begin{cases} z^l_{m,n}, & z^l_{m,n} > 0 \\ 0, & otherwise \end{cases}$$

$$\sigma'(z^l_{m,n}) = \frac{\partial y^l_{m,n}}{\partial z^l_{m,n}} = \begin{cases} 1, & z^l_{m,n} > 0 \\ 0, & otherwise \end{cases}$$

2.9.2 Partial Derivatives

$$\frac{\partial C}{\partial y^l_{m,n}} = \delta^l_{m,n}, \qquad \frac{\partial z^l_{m,n}}{\partial b^l} = 1, \qquad \frac{\partial z^l_{m,n}}{\partial w^l_{a,b}} = y^{l-1}_{m-a,n-b}, \qquad \frac{\partial z^l_{m,n}}{\partial y^{l-1}_{a,b}} = w^l_{m-a,n-b}$$

2.9.3 Cost Function Gradients

To reduce the length of notation in the following equations, let us define $\epsilon^l_{m,n}$ (epsilon) as the delta at an output point $\delta^l_{m,n}$ multiplied by the derivative of the activation function evaluated at the input to the activation function $\sigma'(z^l_{m,n})$:

$$\epsilon^l_{m,n} = \delta^l_{m,n} \cdot \sigma'(z^l_{m,n})$$

The cost function derivative w.r.t. the bias reduces to the sum over all epsilons, as the bias is a single scalar added to every point at the output. The double sum is over the height and width of the output matrix.

$$\frac{\partial C}{\partial b^l} = \sum_m \sum_n \frac{\partial C}{\partial y^l_{m,n}} \cdot \frac{\partial y^l_{m,n}}{\partial z^l_{m,n}} \cdot \frac{\partial z^l_{m,n}}{\partial b^l} = \sum_m \sum_n {\epsilon^l}_{m,n}$$

The cost function derivative w.r.t. an element in the weight matrix is a 2D convolution of the epsilons with the input activation to the convolutional layer. Again, the double sum is over the height and width of the output matrix.

$$\frac{\partial C}{\partial w^l_{a,b}} = \sum_m \sum_n \frac{\partial C}{\partial y^l_{m,n}} \cdot \frac{\partial y^l_{m,n}}{\partial z^l_{m,n}} \cdot \frac{\partial z^l_{m,n}}{\partial w^l_{a,b}} = \sum_m \sum_n \epsilon^l_{m,n} \cdot y^{l-1}_{m-a,n-b}$$

The final cost function derivative w.r.t. the input activation is our backpropagated delta term. This also is a 2D convolution of the epsilons, however this time it is convolved with the weight matrix of the convolutional layer. The double sum is also taken over the height and width of the output matrix.

$$\frac{\partial C}{\partial y^{l-1}_{a,b}} = \delta^{l-1}_{a,b} = \sum_m \sum_n \frac{\partial C}{\partial y^l_{m,n}} \cdot \frac{\partial y^l_{m,n}}{\partial z^l_{m,n}} \cdot \frac{\partial z^l_{m,n}}{\partial y^{l-1}_{a,b}} = \sum_m \sum_n \epsilon^l_{m,n} \cdot w^l_{m-a,n-b}$$

2.10 Pooling Layer

The pooling layer is used to reduce the vertical and horizontal dimensions of the incoming tensor, without affecting the depth. It is useful as it provides translation invariance and also reduces the number of weights required in the network.

The pooling function was chosen to be max pooling, which looks at a pxp section of the input, takes from it the maximum value and places it in a 1x1 grid at the output. The total reduction in size of a stride p max pooling layer, given a square input image of size nxn, is therefore $\lfloor \frac{n}{p} \rfloor$.

2.11 Flatten Layer

The flatten layer converts the incoming tensor from 2D layers, such as convolution or pooling, to 1D column vectors used in the fully connected layers. The tensor is reshaped from DxRxC to 1x(D*R*C)x1.

2.12 Fully Connected Layer

Fully connected layers are usually at the end of a convolutional neural network and perform the classification. In PiCNN they are also known as the 'Dense' layer. Every neuron from the previous layer is connected via a weight to each neuron in the fully connected layer. Furthermore, every neuron also has a bias term.

2.12.1 Output

The output of the fully connected layer is the dot product of the weight matrix, connecting each neuron of the previous layer to the neurons of this layer, with the inputs to the layer added to a bias term at each neuron. This sum is then passed through a non-linear activation function.

$$y_i^l = \sigma\left(\sum_j w_{ij}^l \cdot y_j^{l-1} + b_i^l\right) = \sigma\left(z_i^l\right)$$

Where the activation function and its derivative, chosen to be the hyperbolic tangent (tanh), is as follows:

$$\sigma(z_i^l) = tanh(z_i^l) = \frac{2}{1+e^{-2z_i^l}} - 1$$

$$\sigma'(z_i^l) = \frac{\partial y_{m,n}^l}{\partial z_{m,n}^l} = 1 - \sigma^2(z_i^l)$$

2.12.2 Partial Derivatives

$$\frac{\partial C}{\partial y_i^l} = \delta_i^l, \qquad \frac{\partial z_i^l}{\partial b_i^l} = 1, \qquad \frac{\partial z_i^l}{\partial w_{ij}^l} = y_j^{l-1}, \qquad \frac{\partial z_i^l}{\partial y_j^{l-1}} = w_{ij}^l$$

2.12.3 Cost Function Gradients

As with the convolutional layer and its cost function gradients above, to simply notifation let us defineϵ_i^l (epsilon) as the delta at an output point δ_i^l multiplied by the derivative of the activation function evaluated at the input to the activation function $\sigma'(z_i^l)$:

$$\epsilon_i^l = \delta_i^l \cdot \sigma'(z_i^l)$$

The cost function gradients necessary for backpropagation and weight update are therefore as follows:

$$\frac{\partial C}{\partial b_i^l} = \frac{\partial C}{\partial y_i^l} \cdot \frac{\partial y_i^l}{\partial z_i^l} \cdot \frac{\partial z_i^l}{\partial b_i^l} = \epsilon_i^l$$

$$\frac{\partial C}{\partial w_{ij}^l} = \frac{\partial C}{\partial y_i^l} \cdot \frac{\partial y_i^l}{\partial z_i^l} \cdot \frac{\partial z_i^l}{\partial w_{ij}^l} = \epsilon_i^l \cdot y_j^{l-1}$$

$$\frac{\partial C}{\partial y_j^{l-1}} = \sum_i \frac{\partial C}{\partial y_i^l} \cdot \frac{\partial y_i^l}{\partial z_i^l} \cdot \frac{\partial z_i^l}{\partial y_j^{l-1}} = \epsilon_i^l \cdot w_{ij}^l$$

3 Performance

Initial tests were performed using the MNIST dataset, which includes roughly 60,000 28x28 pixel images of handwritten digits from 0 to 9. The network structure and an excerpt of the training performance per epoch can be seen in *Figure 1* below. The validation accuracy on this training accuracy quickly reaches above 90% after approximately 2 minutes of training. This emphasizes the point that good results can be achieved quickly using just this small, single-header library.

```
Loading training data and labels and normalising...
Creating neural network...
Network structure:
[Network] In: 28x28 | Out: 10x1 | Feature Maps: 3
[Layer 1]: Type: i | Out: 3x28x28
[Layer 2]: Type: c | Out: 3x26x26
[Layer 3]: Type: p | Out: 3x13x13
[Layer 4]: Type: c | Out: 3x11x11
[Layer 5]: Type: p | Out: 3x5x5
[Layer 6]: Type: f | Out: 1x75x1
[Layer 7]: Type: d | Out: 1x25x1
[Layer 8]: Type: d | Out: 1x10x1
Training (Max. Epochs: 100, Batch Size: 100, Min. Error: 0.001, Momentum: 0.01)...
Epoch: 24/100 | Training Error: 0.0453123 | Validation Accuracy: 49.2% | Learning Rate: 0.0685818 | Time Taken: 1.927s
Epoch: 25/100 | Training Error: 0.0503527 | Validation Accuracy: 53.8% | Learning Rate: 0.0610545 | Time Taken: 1.957s
Epoch: 26/100 | Training Error: 0.0455566 | Validation Accuracy: 69.1% | Learning Rate: 0.0360182 | Time Taken: 1.93s
Epoch: 27/100 | Training Error: 0.0297028 | Validation Accuracy: 70.8% | Learning Rate: 0.0332364 | Time Taken: 1.927s
Epoch: 28/100 | Training Error: 0.0251964 | Validation Accuracy: 75.9% | Learning Rate: 0.0248909 | Time Taken: 1.93s
Epoch: 29/100 | Training Error: 0.0245082 | Validation Accuracy: 80.9% | Learning Rate: 0.0167091 | Time Taken: 1.931s
Epoch: 30/100 | Training Error: 0.0226364 | Validation Accuracy: 82.2% | Learning Rate: 0.0145818 | Time Taken: 2.005s
Epoch: 31/100 | Training Error: 0.0191025 | Validation Accuracy: 83.2% | Learning Rate: 0.0129455 | Time Taken: 2.031s
```

Figure 1: Example MNIST Training Excerpt

Further testing is in progress, with larger more complicated datasets and varying network structures. The results and performance sections will be updated accordingly.

4 Future Improvements

PiCNN in its current state is fully functional but can be improved on. The following future improvements, listed in no particular order, will be included in future versions:

- Choice of number of feature maps per layer:
  - Currently, the number of feature maps is identical for all pooling and convolutional layers. This will be made variable for each layer.
- FFT convolutions:
  - Large 2D convolutions can be significantly sped up by applying a fast fourier transform to both the input and weight matrices, then multiplying in the frequency domain, and finally inverting the result with an IFFT. For a 1D convolution, the performance benefit is $n \cdot \log(n)$ compared to n^2 (standard convolution).
- Parallel training of network across multiple Raspberry Pis:
  - A big benefit of Raspberry Pis is their cost. Being able to train a neural network in parallel across many such devices (using for example connections via TCP), could bring a drastic performance benefit.
- Choice of optimisation technique:
  - Currently, only stochastic gradient descent (SGD) with momentum is used for optimisation. In future, ADAM, ADAGRAD, and others will be implemented as a choice for the user.
- Data import:
  - Data can only be loaded from text files, or CSVs. Additional benefits could be gotten from being able to import image files directly.
- Save and load models from XML files:
  - Trained models cannot be saved currently. Trained models in future can be saved in the XML format and reloaded into a program for further training or inference.
- Additional layer types:
  - Dropout, Radial Basis Functions, etc.
- Multithreading on multiple CPU cores:
  - Multi-threading is currently implemented, however threads are not specifically assigned to invididual cores. In future, this could be implemented to further increase the performance.

5 Appendix

5.1 Example Usage

To illustrate the usage of the PiCNN library, the following source code demonstrates how to train a simple convolutional neural network using the MNIST dataset to classify handwritten digits. After having loaded the dataset using the DataHandler class and having set up the network structure, the actual training of the network requires only a single line of code.

```
#include "PiCNN.h"

using namespace std;

int main() {

    cout << "Loading training data and labels and normalising.\n";

    DataHandler dh;
    TensorArray train_data;
    TensorArray train_labels;

    //Load training data from CSV, reshape to 28x28
    train_data = dh.reshape(dh.readCSV("mnist_train_data.csv", ","), 28, 28);

    //Normalise values (0-255) to be between 0 and 1
    train_data = dh.normalise_minmax(train_data);

    //Load training labels from CSV and convert to one-hot vectors
    train_labels = dh.onehot(dh.readCSV("mnist_train_labels.csv", ","), 10);

    cout << "Creating neural network.\n";

    //Create neural network of input size 28x28 and 3 feature maps
    PiCNN c(28, 28, 3);

    //Add layers
    //Convolutional layer, kernel size 3, activation function is ReLU
    c.addConv(3, 'relu');

    //Max-Pooling layer, stride 2
    c.addPool(2);

    //Flatten to 3D volume to 1D array
    c.addFlatten();

    //Add fully-connected layer with 50 neurons, activation function is Tanh
    c.addDense(50, 'tanh');

    //Add final softmax layer with 10 output classes
    c.addDense(10, 'softmax');
```

```
    //Print network structure
    cout << "Network structure:\n";
    c.print();

    //Training parameters
    int batch_size = 100;
    int max_epochs = pow(10, 2); //Maximum number of iterations
    double min_error = pow(10, -3); //Target error
    double lrate = 0.1; //Learning rate
    double mom = 0.01; //Momentum

    cout << "Starting training...\n";

    //Train network
    c.train(train_data, train_labels, lrate,
                mom, max_epochs, min_error, batch_size);

    //Final validation after training
    cout << "Validation...\n";

    int num_correct = 0;
    for (int i = 0; i < train_data.size(); i++) {
        Tensor output = c.feedforward(train_data[i]);
        num_correct += (dh.compare_onehot(output, train_labels[i]) ? 1 : 0);
    }

    cout << "Correct: " << num_correct << "/" << train_data.size() << endl;
    cout << "Done!\n";

    return 0;

}
```

5.2 C++ Source Code

The source code is freely available on GitHub using the following link:

https://github.com/pms67/PiCNN/tree/v2.1