Politecnico di Torino

III Facoltà di Ingegneria

Special Project

Integrated Systems Architecture

Master degree in Electrical Engineering

Authors: Group25

Claudia Golino, Mattia Mirigaldi

January 1, 2023

Contents

CHAPTER 1

Introduction

1.1 Neural networks overview

1.1.1 Fully connnected network

A neural network is a method of artificial intelligence that teach computer how to elaborate data through a process that mimics the way human brain operates.
Neural networks can help computers make intelligent decisions with limited human assistance. This is because they can learn and model the relationships between input and output data that are nonlinear and complex, the neural network can adapt to changing input and generates the best possible results without needing to redesign the output criteria.

Neural Networks (NNs) are modeled as collections of neurons that are connected in an acyclic graph. In other words, the outputs of some neurons can become inputs to other neurons. Cycles are not allowed since that would imply an infinite loop in the forward pass of a network.
For regular neural networks, the most common layer type is the fully-connected layer in which neurons between two adjacent layers are fully pairwise connected, but neurons within a single layer share no connections. If number of layers ≥ 3 then NN is called Deep Neural Network (DNN).

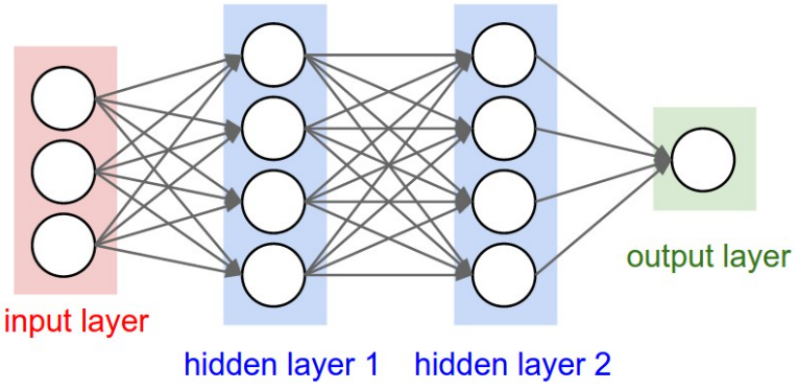


Figure 1.1: 3-layer neural network with three inputs, two hidden layers of 4 neurons each and one output layer. Connections (synapses) between neurons across layers, but not within a layer

The neuron is a linear classifier, meaning that it classify data into labels based on the linear combination (weighted sum) of input features. Furthermore, the neuron doesn't just output that weighted sum, since the computation associated with a cascade of neurons would then be a simple linear algebra operation. Instead is applied a non-linear function on weighted sum of the input values thaht causes a neuron to generate an output only if the inputs cross some threshold. To align brain-inspired termi-

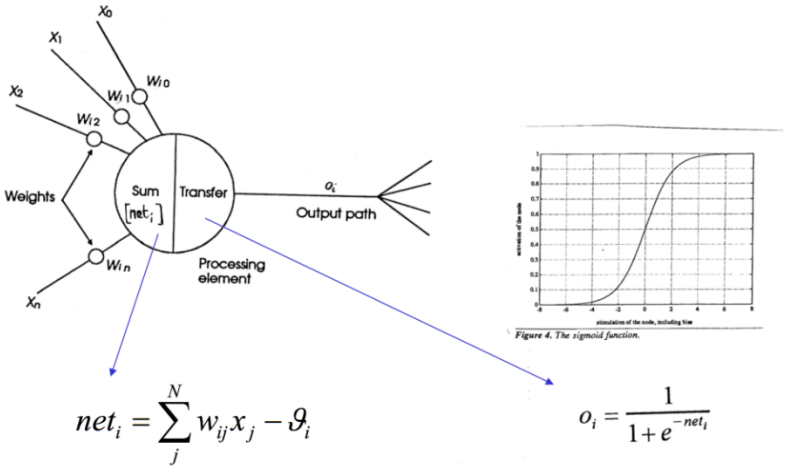


Figure 1.2: Mathematical model of a neuron

nology with neural networks, the outputs of the neurons are often referred to as activations, and the synapses are often referred to as weights. In this report is used the activation/weight nomenclature. Equation 1.1 shows an example of the computation at each layer, where W_{ij}, x_i and y_j are the weights, input activations and output activations, respectively, and f$(\cdot)$ is a non-linear function.

$$\text{layer} : y_j = f(\sum_{i=1}^{3} W_{ij} * x_i + b) \tag{1.1}$$

The network learn how to extract high level features through statistical learning from a large pool of

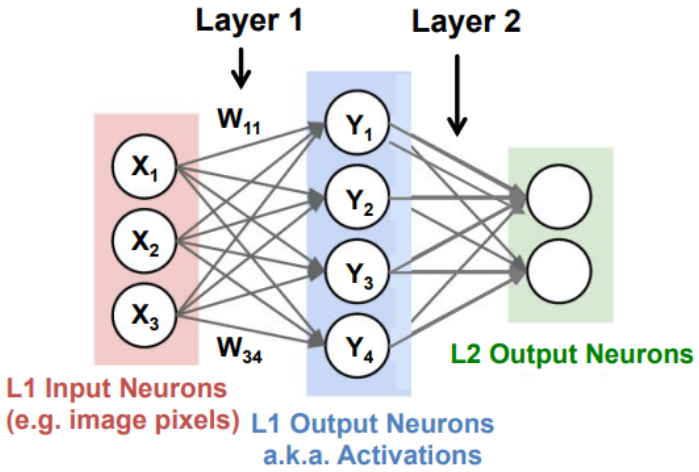


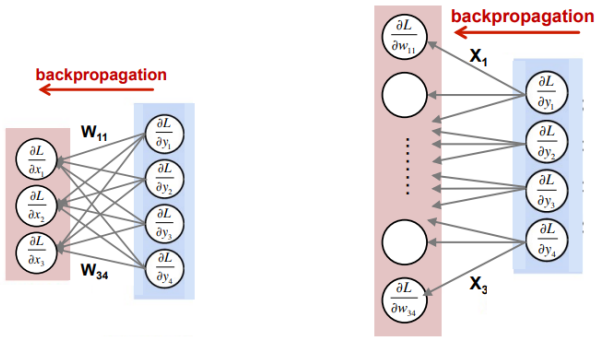
Figure 1.3: Neurons model used in NNs

labelled data, in a process called *training*. During the training the network determine the value of the weights (and bias).Once trained, the program can perform its task by computing the output of the network using the weights determined during the training process. Running the program with these weights is referred to as *inference*.
Taking as example DNNs used in image classification the network is trained by giving in input an image and the output of the DNN is a vector of scores, one for each object class; the class with the highest score indicates the most likely class of object in the image. The overarching goal for training a DNN is to determine the weights that maximize the score of the correct class and minimize the scores of the incorrect classes. When training the network the correct class is often known because it is given for the images used for training (i.e., the training set of the network). The gap between the ideal correct scores and the scores computed by the DNN based on its current weights is referred to as the loss (L). Thus the goal of training DNNs is to find a set of weights to minimize the average loss over a large training set.
When training a network, the weights (w_{ij}) are usually updated using a hill-climbing optimization process called gradient descent. A multiple of the gradient of the loss relative to each weight, which is the partial derivative of the loss with respect to the weight, is used to update the weight.

$$w_{ij}^{t+1} = w_{ij}^{t} - \alpha * \frac{\delta L}{\delta w_{ij}} \tag{1.2}$$

Where α is called the learning rate.

This gradient indicates how the weights should change in order to reduce the loss. The process is repeated iteratively to reduce the overall loss. An efficient way to compute the partial derivatives of the gradient is through a process called *backpropagation*. Backpropagation, which is a computation derived from the chain rule of calculus, operates by passing values backwards through the network to compute how the loss is affected by each weight. This backpropagation computation is, in fact, very similar in form to the computation used for inference as shown in 1.4 Thus, techniques for efficiently performing inference can sometimes be useful for performing training.
A variety of techniques are used to improve the efficiency and robustness of training. For example,



(a) Compute the gradient of the loss relative to the filter inputs

(b) Compute the gradient of the loss relative to the weights

Figure 1.4: Example of backpropagation through a neural network

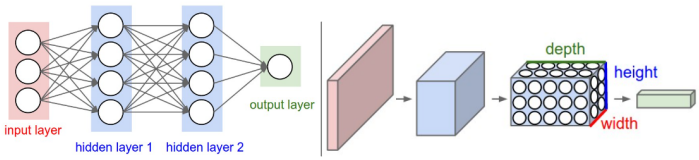
often the loss from multiple sets of input data, i.e., a batch, are collected before a single pass of weight update is performed; this helps to speed up and stabilize the training process.

1.1.2 Convolutional Neural Networks

A common form of DNNs is Convolutional Neural Nets (CNNs), ConvNet architectures makes the explicit assumption that the inputs are images, which allows us to encode certain properties into the architecture.

Regular Neural Nets don't scale well to full images. Taking as example the CIFAR-10 dataset, images are only of size 32x32x3 (32 wide, 32 high, 3 color channels), so a single fully-connected neuron in a first hidden layer of a regular Neural Network would have 32*32*3 = 3072 weights. This amount still seems manageable, but clearly this fully-connected structure does not scale to larger images. For example, an image of more respectable size, e.g. 200x200x3, would lead to neurons that have 200*200*3 = 120,000 weights. Clearly, this full connectivity is wasteful and the huge number of parameters would quickly lead to overfitting.
Overfitting is a statistical phenomenon, not only relevant to NNs, which indicates that the algorithm has almost "memorized" training data instead of "learning" them (training error is much smaller than your test error).

Unlike a regular Neural Network, the layers of a ConvNet have neurons arranged in 3 dimensions: width, height, depth. The neurons in a layer will only be connected to a small region of the layer before it, instead of all of the neurons in a fully-connected manner.Taking as example CIFAR-10 the final output layer would have dimensions 1x1x10, because by the end of the ConvNet architecture we will reduce the full image into a single vector of class scores, arranged along the depth dimension (fig. 1.1.2)



Left: A regular 3-layer Neural Network. Right: A ConvNet arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers. Every layer of a ConvNet transforms the 3D input volume to a 3D output volume of neuron activations. In this example, the red input layer holds the image, so its width and height would be the dimensions of the image, and the depth would be 3 (Red, Green, Blue channels).

Simple ConvNet is a sequence of layers, and every layer of ConvNet transforms one volume of activations to another through a differentiable function.
The three main types of layers used to build ConvNet architectures are :

- CONV layer : will compute the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and a small region they are connected to in the input volume.
- RELU layer : will apply an elementwise activation function, such as the max(0,x) thresholding at zero. This leaves the size of the volume unchanged.
- POOL layer : will perform a downsampling operation along the spatial dimensions (width, height), resulting in a volume reduction
- FC (fully-connected) layer: will compute the class scores, each neuron in this layer will be connected to all the numbers in the previous volume.

These layers are stacked to form a full ConvNet architecture.
In this way, ConvNets transform the original image layer by layer from the original pixel values to the final class scores. Taking as example the CIFAR-10 the output volume will have size [1x1x10],

where each of the 10 numbers correspond to a class score, such as among the 10 categories of CIFAR-10.

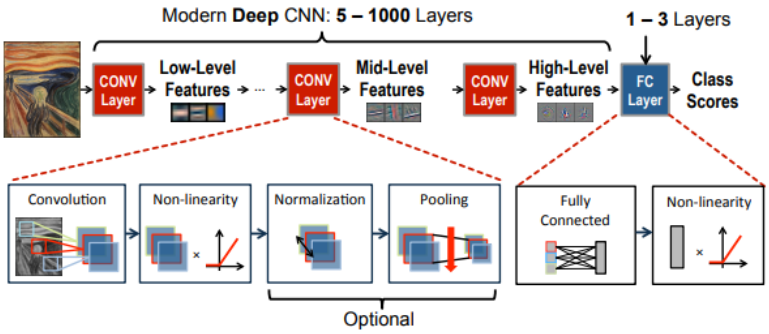


Figure 1.5: Layers in a convolutional neural network

The Conv layer is the core building block of a Convolutional Network that does most of the computational heavy lifting.
Instead of connecting neurons to all neurons in previous volume, each neuron is connected to only a local region of the input volume. The spatial extent of this connectivity is a hyperparameter called the receptive field of the neuron (equivalentlythis is the filter size). The extent of the connectivity along the depth axis is always equal to the depth of the input volume.

Example : Supposing that the input volume has size [32x32x3], (e.g. an RGB CIFAR-10 image). If the receptive field (or the filter size) is 5x5, then each neuron in the Conv Layer will have weights to a [5x5x3] region in the input volume, for a total of 5*5*3 = 75 weights (and +1 bias parameter). Notice that the extent of the connectivity along the depth axis must be 3, since this is the depth of the input volume.

Three hyperparameters control the size of the output volume: the depth, stride and zero-padding.

- Depth of the output volume : it corresponds to the number of filters we would like to use, each learning to look for something different in the input. For example, if the first Convolutional Layer takes as input the raw image, then different neurons along the depth dimension may activate in presence of various oriented edges, or blobs of color. The set of neurons that are all looking at the same region of the input is colled a depth column.
- Stride with which we slide the filter : when the stride is 1 then we move the filters one pixel at a time. When the stride is 2 then the filters jump 2 pixels at a time as we slide them around. This will produce smaller output volumes spatially.
- Zero-padding : may be convenient to pad the input volume with zeros around the border. The nice feature of zero padding is that it will allow us to control the spatial size of the output volumes.

The spatial size of the output volume can be computed as a function of the input volume size (W), the receptive field size of the Conv Layer neurons (F), the stride with which they are applied (S), and the amount of zero padding used (P) on the border.
The spatial size then will be equal to :

$$W_{out} = H_{out} = \frac{W - F + 2 * P}{S + 1} \tag{1.3}$$

For example for a 7x7 input and a 3x3 filter with stride 1 and pad 0 we would get a 5x5 output. With stride 2 we would get a 3x3 output. To summarize, the Conv Layer:

- Accepts a volume of size $W_{in} \times H_{in} \times D_{in}$
- Requires four hyperparameters
  - their spatial extent F
  - the stride S
  - the amount of zero padding P
  - number of filters K
- Produces a volume of size $W_{out} \times H_{out} \times D_{out}$, where :
  - $W_{out} = \frac{Win-F+2P}{S+1}$
  - $H_{out} = \frac{Win-F+2P}{S+1}$ (i.e. width and height are computed equally by symmetry)
  - $D_{out} = \text{K}$

With parameter sharing, constraining the neurons in each depth slice to use the same weights and bias, it introduces F*F*D1 weights per filter, for a total of (F*F*D1)*K weights and K biases. constrain the neurons in each depth slice to use the same weights and bias.

1.2 Project goal

In recent years, methods based on deep neural networks(DNNs) have become the standard approach for complex problems as computer vision. However a good accuracy of CNNs comes at the cost of high computational complexity, typically in the order of billions of multiply and accumulate (MAC) operations for recent state of the art networks.
CNNs are usually trained on GPUs using floating points precision for the data (weights and activations), GPUs are very efficient when processing high amount of data in a massively parallelized way, such as during the training, but are inherently energy hungry and infeasible for deployment on battery powered devices. By contrast, FPGAs and ASIC are extremely efficient and can be designed around the workload that will be processed at the edge.
A recent approach is to co-design both the CNN and its accelerator, carefully tailoring the computation requirements and resources availability with the target task accuracy. Xilinx's researchers developed a framework called Brevitas, built upon Pytorch, to train low bit-width quantized CNNs with the target accelerator in mind. The trained CNN model is then passed to another framework called FINN (Xilinx again) which generates an accelerator based on a systolic array that can execute the network efficiently, respecting the performance targets determined by the designer. The result is a bitstream that can be mounted on a FPGA accelerator, providing an easy way to design and deploy efficient CNNs.
In this project is shown the workflow of implementing a quantized convolutional neural network accelerator on a Xilinx FPGA using a python-HLS toolchain. The hardware accelerator for the CNN models is generated with FINN, a systolic array generator, and deployed on a Xilinx FPGA. The toolchain is written in Python, C++ and HLS

1.3 Report organization

The report is organized in chapters, one for each exercise (from 1 to 5) each first focus on the problem statement and then presenting the results.
In the report can be found :

- For exercise 1 through 4 the code of CNN models used during the experiments, plus training hyper-parameters, pre- and post-processing of data, special constructs/classes/functions used outside or within the CNN model during the training process
- In exercise 5 is included the code of the jupyter notebooks that has been modified in order to port the model from FINN onto the ZCU104 development board. Can also be found a table with all hardware metrics extracted from FINN

1.4 Exercise 0

This first exercise present an introductory script with the basic functions and definitions required to implement the training loop of a simple neural network for MNIST.
The MNIST dataset consist of 60000 train images and 10000 test images of digits and was used to develop the first convolutional neural network for handwritten digit recognition.
To run the python scripts is used as IDE anaconda, then the following commands are executed to install the required packages:

$ pip install brevitas

$ conda install -c conda-forge onnx

$ conda install tqdm

$ pip3 install onnxoptimizer

$ conda install –x conda-forge tensorflow

The training script used is essentially composed of 3 elements: datatest pipeline, training function and test function.

- Dataset pipeline: loads the data, apply pre-processing functions such as normalization, padding, shuffling, cropping, data augmentation, split and so on. The dataset pipeline is necessary to make all the input data coherent (for instance, same dimension) and feed it to the processing engines to avoid bottlenecks during the training process.
- Training function: for all the elements wrapped in a single batch, executes a forward pass, computes the error between the neural network predictions and the ground truth with a loss function, back propagates the error and evaluate the gradient, then updates the weight. It also returns the accuracy and loss.
- Test function: for all the elements wrapped in a single batch, execute a forward pass, counts or evaluates the correct predictions, returns the accuracy and loss.

Data does not always come in its final processed form that is required for training machine learning algorithms. We use transforms to perform some manipulation of the data and make it suitable for training. The torchvision.transforms module offers several commonly-used transforms out of the box (*from torchvision.transforms import transforms*). this exercise the transform applied is ToTensor that

converts a PIL image or NumPy ndarray into a FloatTensor and scales the image's pixel intensity values in the range [0., 1.] (normalization). The tensor is conceptually identical to a numpy array but unlike it Tensors can utilize GPUs to accelerate their numeric computation, GPUs often provide speedups of 50x or greater
To enable easy access to the samples and laberls Pythorch provides a primitive : *DataLoader*, the dataloader function wraps the dataset objects definitions (in this case the MNIST dataset) and offer an easy API to load the data from the memory to the system memory, then apply pre-processing on CPU, and finally move it to the device that executes the training and test processes.
To measure the compatibility between prediction and ground truth in these exercises is used the Cross Entropy Loss, the data loss L is computed as the average over the data losses of every example.

$$L = \frac{1}{N} * \sum_i L_i \tag{1.4}$$

Since we are dealing with a classification problem, single correct label (out of a fixed set), using the Softmax classifier that uses the cross-entropy loss leads to faster training as well as improved generalization

$$L_i = -\log(\frac{e^{f_{y_i}}}{\sum_i e^{f_j}}) \tag{1.5}$$

Where f_{y_i} is the output vector of the NN, while f_j is the expected vector (score vector).
Other possible loss functions are the L1loss, CrossEntropyLoss, SGD, RMSprop and Adam.
Once the analytic gradient is computed with backpropagation, the gradients are used to perform a parameter update. The simplest form of update is the SDG (Stochastic Gradient Descent), the parameters ared changedalong the negative gradient direction (since the gradient indicates the direction of increase, but we usually wish to minimize a loss function).
Assuming a vector of parameters x and the gradient dx, the SDG update has the form:

$$x+ = -learning\_rate * \delta x \tag{1.6}$$

Where the learning_rate is a hyperparameter, if too low then are no made progess on the loss function. With deep networks another approach is the *momentum update*, where the optimization process can be seen as equivalent to the process of simulating the parameter vector (i.e. a particle with zero initial velocity in some location) as rolling on the landscape. This physic view suggests an update in which the gradient only directly influences the velocity, which in turn has an effect on the position:

$$\begin{aligned} v &= mu * v - learning\_rate * \delta x \\ x+ &= v \end{aligned} \tag{1.7}$$

The v variable is initialized at zero,the additional hyperparameter (mu) is referred to as momentum (its typical value is about 0.9).

The python script also checks how accurate the neural network is for each class since with some models that might have a really high accuracy on a subset of the dataset classes, while underperforming on others, which might result on a high overall test accuracy even if the model is not good.

```
if __name__ == '__main__':
    import torch, torchvision, copy
    from torch import nn
    from torch.utils.data import DataLoader
    from torchvision import datasets
    from torchvision.transforms import transforms
    import numpy as np
    from tqdm import tqdm
```

```
9     from torch_neural_networks_library import default_model  # NN models will be
      defined in this python file
10    from pathlib import Path
11    from find_num_workers import find_num_workers
12    from torch.utils.tensorboard import SummaryWriter
13
14    Path("./runs/exercise_0").mkdir(parents=True, exist_ok=True)  # check if runs
      directory for tensorboard exist, if not create one
15
16    writer = SummaryWriter(log_dir='runs/exercise_0')
17    # applied ToTensor transformation
18    transform = transforms.Compose([transforms.ToTensor()])
19
20    # Set download to True if you are running this script for the first time,
      otherwise set it to False
21    training_data = datasets.MNIST(
22        root="data",
23        train=True,
24        download=False,   # set to false if the dataset has been downloaded already
25        transform=transform,
26    )
27
28    # Set download to True if you are running this script for the first time,
      otherwise set it to False
29    test_data = datasets.MNIST(
30        root="data",
31        train=False,
32        download=False,   # set to false if the dataset has been downloaded already
33        transform=transform,
34    )
35
36    batch_size = 16
37
38    #best_workers = find_num_workers(training_data=training_data, batch_size=
      batch_size)
39    best_workers = 0   # change this number with the one from the previous function
      and keep using that for future runs
40
41    # DataLoader
42    train_dataloader = DataLoader(training_data, batch_size=batch_size, shuffle=True,
      num_workers=best_workers, pin_memory=torch.cuda.is_available())
43    test_dataloader = DataLoader(test_data, batch_size=batch_size, shuffle=True,
      num_workers=best_workers, pin_memory=torch.cuda.is_available())
44
45    for X, y in test_dataloader:  # Print the shape of the input and output data
46        print("Shape of X [N, C, H, W]: ", X.shape, X.dtype)
47        print("Shape of y: ", y.shape, y.dtype)
48        break
49    print(test_data.classes)
50
51    # writer function to initialize Tensorboard
52    dataiter = iter(copy.deepcopy(test_dataloader))
53    images, labels = dataiter.next()
54    img_grid = torchvision.utils.make_grid(images)
55    writer.add_image(str(batch_size)+'_mnist_images', img_grid)
56
57    # Set cpu or gpu device for training.
58    device = "cuda" if torch.cuda.is_available() else "cpu"
59    print("Using {} device".format(device))
60
61    model = default_model()
62    print(model)
```

```
    writer.add_graph(model, images)
    """It is necessary that both the model and the tensors involved in the inference/training are on the same device"""
    model.to(device)
    model_parameters = filter(lambda p: p.requires_grad, model.parameters())
    params = sum([np.prod(p.size()) for p in model_parameters])   # count learnable parameters
    memory = params * 32 / 8 / 1024 / 1024  # evaluate total parameter memory
    print("this model has ", params, " parameters")
    print("total weight memory is %.4f MB" %(memory))

    # loss function and optimizer instantation
    loss_fn = nn.CrossEntropyLoss()
    optimizer = torch.optim.SGD(model.parameters(), lr=5e-4)

    # training function
    def train(dataloader, model, loss_fn, optimizer, epoch):
        size = len(dataloader.dataset)
        model.train()
        for batch, (X, y) in enumerate(dataloader):
            X, y = X.to(device), y.to(device)

            # Forward propagation
            pred = model(X)  # Compute prediction error
            loss = loss_fn(pred, y)  # Compute loss

            # Backpropagation
            optimizer.zero_grad()  # prepares the gradient by setting it to zero for all tensors
            loss.backward()  # computes the gradient of the graph, in this case the collection of layers
            optimizer.step()  # performs a parameter update using the method instantiated before, in this case the SGD

            if batch % 1000 == 0:
                loss, current = loss.item(), batch * len(X)
                print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
                writer.add_scalar('training loss', loss / 1000, epoch * len(dataloader) + batch)
        return loss

    # test function
    def test(dataloader, model, loss_fn):
        size = len(dataloader.dataset)
        num_batches = len(dataloader)
        model.eval()
        test_loss, correct = 0, 0
        with torch.no_grad():
            for X, y in dataloader:
                X, y = X.to(device), y.to(device)
                pred = model(X)
                test_loss += loss_fn(pred, y).item()
                correct += (pred.argmax(1) == y).type(torch.float).sum().item()
        test_loss /= num_batches
        correct /= size
        print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss: {test_loss:>8f} \n")
        return correct

    epochs = 5
    best_correct = 0
    best_model = []
```

```
118
119     print("Use $ tensorboard --logdir=runs/exercise_0 to access training statistics")
120     Path("./saved_models").mkdir(parents=True, exist_ok=True)
121
122     for t in tqdm(range(epochs)):
123         print(f"Epoch {t+1}\n-------------------------------")
124         loss = train(train_dataloader, model, loss_fn, optimizer, t)
125         current_correct = test(test_dataloader, model, loss_fn)
126         writer.add_scalar('test accuracy', current_correct, t)
127         torch.save({
128             'epoch': t,
129             'model_state_dict': model.state_dict(),
130             'optimizer_state_dict': optimizer.state_dict(),
131             'loss': loss,
132             'test_acc': current_correct,
133         }, "./saved_models/default_model.pth")
134         print("Saved PyTorch Model State to model.pth")
135
136     writer.close()
137
138     #  Class accuracy test loop
139     classes = test_data.classes
140     correct_pred = {classname: 0 for classname in classes}
141     total_pred = {classname: 0 for classname in classes}
142
143     with torch.no_grad():
144         for X, y in test_dataloader:
145             images, labels = X.to(device), y.to(device)
146             outputs = model(images)
147             _, predictions = torch.max(outputs, 1)
148             for label, prediction in zip(labels, predictions):
149                 if label == prediction:
150                     correct_pred[classes[label]] += 1
151                 total_pred[classes[label]] += 1
152
153     min_correct = [0,110]
154     for classname, correct_count in correct_pred.items():
155         accuracy = 100 * float(correct_count) / total_pred[classname]
156         if min_correct[1] >= int(accuracy):
157             min_correct = [classname, accuracy]
158         print("Accuracy for class {:5s} is: {:.1f} %".format(classname, accuracy))
159
160     lowest_class_accuracy = min_correct[1]
161
162     print("Worst class accuracy is %.4f for class %s" %(min_correct[1], min_correct
    [0]))
```

Some of the hyperparameters are :

- The epochs value : defines how many times the training loop is run over the entire dataset. The epoch size can be tuned to select the minimum number that allows the model to converge to a stable and reliable accuracy. Values too small or too big can lead to under/over fitting.
- The batch size: sets how many inputs are processed at once during each forward and backward propagation pass. High batch values improve the training speed, but require more memory.(n.b. also model parameters require memory, thus large networks require a lot of memory even with small batches sizes).

When training a model we typically want to pass samples in “minibatches”, reshuffle the data at every epoch to reduce model overfitting, and use Python’s multiprocessing to speed up data retrieval.

The default neural network model consist of three hidden layers, each FC layer and followed by an activation function. It is defined in file *torch_neural_networks_library.py*

```
class default_model(nn.Module):
def __init__(self):
    super(default_model, self).__init__()
    self.flatten = nn.Flatten()
    self.linear1 = nn.Linear(28*28, 512)
    self.linear2 = nn.Linear(512, 512)
    self.linear3 = nn.Linear(512, 10)
    self.act = nn.ReLU()

def forward(self, x):
    out = self.flatten(x)
    out = self.linear1(out)
    out = self.act(out)
    out = self.linear2(out)
    out = self.act(out)
    out = self.linear3(out)
    return out
```

There is no explicit parameter initialization, by default the weights are initialized by drawing them from a gaussian distribution with standard deviation of $\frac{2}{\sqrt{n}}$ where n is the number of inputs to the neuron. [7]

Worst class accuracy is 65.0224 % for class 5 - five
Accuracy per class results are :

Class	Accuracy [%]
class 0 - zero	96.1
class 1 - one	97.3
class 2 - two	83.0
class 3 - three	83.0
class 4 - four	85.3
class 5 - five	65.0
class 6 - six	89.9
class 7 - seven	88.0
class 8 - eight	79.2
class 9 - nine	73.6

CHAPTER 2

Exercise 1: training loop optimization for a 2-layer network.

Starting from the default neural network presented in the intro, the goal of this exercise is to maximise the accuracy by tweaking the training setup.
The parameters that can modified to get a model that has the best trade off between accuracy, model parameters and model size are :

- pre-processing functions
- batch size
- epoch size
- learnable parameters initialization of the NN model
- parameters of the optimizer, the optimizer is fixed

2.1 Neural Network model

The given Neural Network consist in 4-layers model (3 hidden layers and one output layer) with a Relu activation function used after every Fully connected layer, as shown in Figure 2.1.
The flatten layer is used to flatten input by reshaping it into a one-dimensional tensor, while the linear ones are the Fully connected layers and are used to convert the dimensions of the previous layer to desired one.
The Pytorch class reported below is the default Neural Network used in this first exercise.

```
class default_model(nn.Module):
    def __init__(self):
        super(default_model, self).__init__()
        self.flatten = nn.Flatten()
        self.linear1 = nn.Linear(28*28, 512)
        self.linear2 = nn.Linear(512, 512)
        self.linear3 = nn.Linear(512, 10)
        self.act = nn.ReLU()


    def forward(self, x):
        out = self.flatten(x)
        out = self.linear1(out)
```

```
14          out = self.act(out)
15          out = self.linear2(out)
16          out = self.act(out)
17          out = self.linear3(out)
18          return out
```

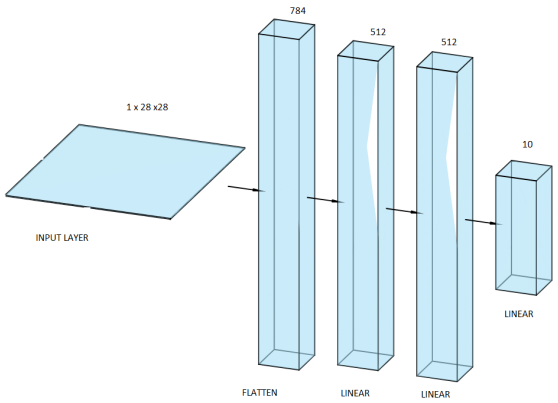


Figure 2.1: Default Neural Network model

2.2 Optimization Setup

Pre-processing functions

In order to have better performances a good model is necessary but also the quality of the data is important. In general the preprocessing helps to make the data consistent by eliminating any duplicates, irregularities in the data, normalizing the data to compare, and improving the accuracy of the result.
The preprocessing is only computed on the training data, and then applied to the test data, not across the entire dataset and then the data is divided into train/val/test splits.
The scaling preprocessing is important because it facilitates optimization, especially optimization using gradient descent[12].Using a scaling method that produces values on both sizes of zero, such as $[-1,1]$ scaling is preferred: normalization helps because it ensures that there are both positive and negative values used as inputs for the next layer, which makes learning more flexible.
This type of preprocessing is not strictly necessary, in fact MNIST commonly use $[0, 1]$ range.

This operation is done by the means of the *Transform* class which support many input transformations, including "Compose", "ToTensor", "Normalize", "Resize", "RandomCrop", etc.
The used input pre-processing transformations to improve accuracy are:

Compose : used to combine more transforms together;

ToTensor : used to converts the torchvision datasets from PIL Images to torch.FloatTensor of shape (C x H x W), in this case (1 x 28 x 28), in the range [0, 1];

Normalize used to normalizes each dimension so that the range is [-1, 1].

The corresponding code line is:

```
transform_train = transforms.Compose([transforms.ToTensor(), transforms.Normalize
((0.5, ), (0.5,)),]),
# where the parameters mean and std of the normalization are passed as 0.5, 0.5 to
 obtain an image equal to (input - mean) / std
```

Batch size

The batch size is the number of images that will be used in the gradient estimation process. It influences how much GPU or system memory is required, but also influences how fast the optimizer can converge to the optima.The number of batch sizes should be a power of 2 to take full advantage of the GPUs processing. Values too big or too small will slow down the training or even cause a crash sometimes.
Checking the average loss and iteration time displayed in the console during the training, it can be noticed:

- the relationship between the batch size and the noise in the loss function[13]:
  for a small batch size the "wiggle" in the loss will be relatively high, instead when the batch size is the full dataset the wiggle will be minimal because every gradient update should be improving the loss function monotonically (unless the learning rate is set too high). In fact small batch size can have a significant regularization effect because of its high variance[15].

- the high correlation between batch size and learning rate[8]:
  When the learning rate is high, the large batch size performs better than with small learning rate. Also small batch size will require a small learning rate to prevent it from overshooting the minima [6].

It's known from image classification studies that the higher the batch size the higher the network accuracy[11] but our results concluded that a higher batch size does not usually achieve high accuracy. Lowering the learning rate and decreasing the batch size will allow the network to train better, especially in the case of fine-tuning.
The ***batch size was tuned to 32***, according to the memory capacity of our processing unit.

Learnable parameters initialization

One common problem with Neural Networks is vanishing and exploding gradient descent during the forward propagation. To solve these issues, one solution could be to initialize the parameters carefully, in particular **weight initialization**.
If weights are initialized correctly, the optimization of loss function will be achieved in the least time otherwise loss gradients will either be too large or too small, and the network will take more time converging to a minimum using gradient descent (or even not converge at all).
The best practise to:

- use RELU or leaky RELU as the activation function, as they both are relatively robust to the vanishing or exploding gradient problems;

- use Heuristics for weight initialization, depending on the chosen non-linear activation function. For RELU activation function the best heuristic is called He-et-al/Kaiming Initialization.

In fact Zero initialization causes the neuron to memorize the same functions almost in each iteration, by making hidden layers symmetric.
Random initialization can break the symmetry, however initializing weight with much high or low value can result in slower optimization, so an extra scaling factor can be required (see Xavier initialization[7]

which scales weights by $\frac{1}{\sqrt{N}}$ from a uniform distribution in [-1, 1], He initialization[4]).
The benefits are that all the heuristics do not vanish or explode too quickly (as the weights are neither too much bigger than 1 nor too much less than 1) and they help to avoid slow convergence and ensure that we do not keep oscillating off the minima.
The kaiming initialization is usually implemented by scaling normal or uniform distributions of weights. Using an uniform distribution, the model yields relatively bigger gradients than using a normal distribution[5], so we decided to use the ***torch.nn.init.kaiming_uniform*** class, which fills the weights with values according to the method described in [7] using a uniform distribution. The resulting weights will have values sampled from $\mathcal{U}(-\text{bound}, \text{bound})$ where bound is:

$$\text{bound} = \text{gain} \times \sqrt{\frac{3}{\text{fan\_mode}}} \tag{2.1}$$

where the parameter "fan_mode" is 'fan_in' by default and preserves the magnitude of the variance of the weights in the forward pass, the parameter "$gain$" depends on the given nonlinearity function and in the Relu fuction correspond to $\sqrt{2}$.
In particular we decided to use the default pytorch weight initialization method, that is

```
1     init.kaiming_uniform_(self.weight, a=math.sqrt(5))
```

Where the parameter "a" is the negative slope of the rectifier used after this layer (only used with 'leaky_relu').

Biases initialization: biases can be safely initialized to 0 because bias will depend only on the linear activation of that layer, but not depend on the gradients of the deeper layers.

Optimizer's parameters

The class SDG, that implements Stochastic Gradient Descendet optimizer in Pytorch, has several parameters[10]: learning rate, momentum, dampening, weight decay (L2 penalty) and nesterov(boolean).
The default parameters are 0 for all parameters except for the learning rate which is required. The pythorch implementation can also enable the Nesterov momentum, set False by default, that ensures much faster convergence (linear rather than sublinear) compared to SDG.
The default learning rate proposed for this exercise is set to **lr=5e-4**.
A learning rate of 5e-4, would mean that weights in the network are updated 5e-4 * (estimated weight error) or 0.05% of the estimated weight error each time the weights are updated.
Typical values for a neural network with standardized inputs (or inputs mapped to the (0,1) interval) are less than 1 and greater than 10^{-6} and a default value of 0.1 typically works for standard multi-layer neural networks[2]. So the default learning rate can appropriate for a fast convergence, but we proceeded to do a cross-validation to find an optimal value:
we performed a sensitivity analysis of the learning rate for the chosen model, also called a grid search. This can help to both highlight an order of magnitude where good learning rates may reside, as well as describe the relationship between learning rate and performance.
Typically, a grid search involves picking values approximately on a logarithmic scale, e.g., a learning rate taken within the set {0.1, 0.01, 10^{-3}, 10^{-4} , 10^{-5}}[6]. Moreover a typical setting for the momentum is a value greater than 0.0 and less than one, where common values such as 0.5, 0.9 and 0.99 are used in practice[6].
The optimum value found for the learning rate is **0.5**

Epoch size

By changing the epochs parameter the number of times the model is trained over the entire dataset is changed.
In general if it is needed to speed-up the training without increasing the epochs from the default value it is possible to act on the learning rate. The number of epochs that our model requires to reach the optima or oscillate around is about **3**.
The model requires 1 epoch to get past 80% accuracy with an average loss of 0.11. After 3 epochs the model reaches an accuracy of 97.3% with an average loss of 0.09.

2.3 Performance

Our model performance is compared with the performance of two default models: one trained with the default script (without changing any parameter) and one trained with an optimized script, i.e. one model trained by tuning the training loop only (learning rate, data pre processing).
The score is evaluated as:

$$score = \frac{our\_model\_minimum\_class\_accuracy}{default\_model\_minimum\_accuracy} * A + \frac{default\_epochs}{your\_epochs} * B \tag{2.2}$$

- The coefficients are: A = 0.6, B = 0.4
- Default minimum class accuracy = 5.9417, default epochs = 5
- Default optimized minimum class accuracy = 96.0357, default optimized epochs = 3

Results

- score against default training script = 10.1308
- score against optimized training script = 0.9855
- model parameters: 669706
- total weight in memory: 2.5547 MB

Recap optimizations :

Pre-processing	ToTensor() + Normalize((0.5,) (0.5,))
Batch size	32
Weight initialization	Relu as activation function Weights values from a gaussian distribution with $\sigma = \sqrt{\frac{2}{N}}$ N : inputs to neuron
Bias initialization	0
SGD learning rate	0.5
Epoch size	3

Accuracy per class

Worst class accuracy is 93.7220 for class 5 - five

Class	Accuracy [%]
class 0 - zero	98.9
class 1 - one	99.4
class 2 - two	98.5
class 3 - three	97.3
class 4 - four	98.2
class 5 - five	93.7
class 6 - six	97.9
class 7 - seven	96.4
class 8 - eight	95.6
class 9 - nine	96.0

CHAPTER 3

Exercise 2: design of a CNN for digit classification and fashion MNIST

The goal for this exercise is to write a neural network for MNIST dataset that uses a maximum of 6 layers with learnable parameters. Compared to previous exercise, the model now can be fully customized by modifying the Neural Network class, e.g the layer type and the dimensions of kernel, input and output volume.
In order to retrieve a model that has the best trade off between accuracy, model parameters and model size the following parameters can be modified:

- pre-processing functions
- batch size
- epoch size
- Neural Network model
- learnable parameters of the NN model
- parameters of the optimizer, the optimizer is fixed

3.1 Neural Network model

The default model of exercise 1 is simple, but over-parametrized and slow to train and execute for a simple task like digit recognition, since it uses fully-connected layers only.
In the adopted model Convolutional layers are used because are particularly suited for image processing tasks, such as handwritten digits recognition. The CNN model adopted uses a simplified layers pattern of the suggested formula[14] which describes the most common form of a ConvNet architecture :

$$INPUT - [[CONV - RELU] * N - POOL?] * M - [FC - RELU] * K - FC, \tag{3.1}$$

Where the * indicates repetition, and the POOL? indicates an optional pooling layer.

In particular it's composed by 2 Convolutional layers, each followed by a Relu activation function and a MaxPool2d layer and it doesn't have normalization layers; after this sequence there is a final FClayer to retrieve in output the 10 classes.

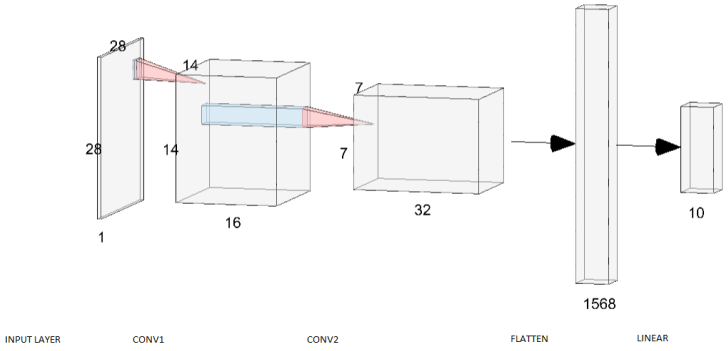


Figure 3.1: Convolutional Neural Network model

CNN for MNIST dataset

The input has dimensions [N, C, H, W] = [32, 1, 28, 28], with :

- N : batch size
- C : n°channels
- H : height
- W : width

MNIST dataset is composed by 28x28 grayscale images.

The first **Convolutional layer** has the following parameters:

- in_channels=1;
- out_channels=16;
- kernel_size=5 (F);
- stride=1(S);
- padding=2 (P).

The output 2D size will be : $\lfloor \frac{W-F+2P}{S} \rfloor + 1$ that in this case is $\lfloor \frac{28-5+2*2}{1} \rfloor + 1 = 28$, so the output volume will be 28x28x16.

The following layer is the **Relu layer**:
this layer does not change the size of the volume and it applies an elementwise activation function on the input performing the function *f(x)=max(0,x)*. Models that use this activation function are easier to train and often achieve better performance.

The following **MaxPool layer** reduce the spatial size to reduce the amount of parameters and computation in the network, and hence to also control overfitting.
In particular the most common downsampling operation is max and MaxPooling has been shown to work better in practice than average pooling. The parameters are kernel size = 2 (F) and stride = 2 (S), in which every depth slice in the input is downsampled by 2 along both width and height, discarding 75% of the activations, larger receptive fields will be too destructive. The depth dimension

remains unchanged.
More specifically the Maxpooling layer with kernel size = 2 (F) and stride = 2 (S) and input volume $W_{in} * H_{in} * D_{in}$ will produce an output volume $W_{out} * H_{out} * D_{out}$:

$$W_{out} = \frac{W_{in}-F}{S} + 1 = 14 \ , \ H_{out} = \frac{W_{in}-F}{S} + 1 = 14 \ , \ D_{out} = D_{in} = 16.$$

N.B. In Pytorch the stride of the window is equal to the kernel size by default, so it can be omitted.
Then another **Convolutional - Relu- MaxPool sequence of layers** is present:
in this case the changed parameters are in_channels (=16) and out_channels (=32) of the Convolutional layer, so the output volume will be $W_{out} = H_{out} = 7, D_{out} = 32$.
The number of channels in the second convolutional layer is increased with the respect to the one in the first convolutional layer, in order to increase the depth of the matrices involved in the convolutions.
Lastly the output volume is flatten to (32 * 7 * 7) before entering in the Fully Connected layer. In this case "*x.view(x.size(0), -1)*" is equivalent to use nn.Flatten() function.
In the **Fully Connected layer** a non linear transformation is applied to the (32 * 7 * 7) input vector and it produces an output vector of 10, which correspond to the 10 classes.
The Pytorch class reported below shows how the Neural Network was implemented for this exercise.

```
class CNN_mnist(nn.Module):
    def __init__(self):
        super(CNN_mnist, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(
                in_channels=1,
                out_channels=16,
                kernel_size=5,
                stride=1,
                padding=2,
            ),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(16, 32, 5, 1, 2),
            nn.ReLU(),
            nn.MaxPool2d(2),
        )
        self.linear = nn.Linear(32 * 7 * 7, 10)
    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = x.view(x.size(0), -1)
        out = self.linear(x)
        return out
```

3.2 Optimization Setup

The optimization is very similar to the one performed in exercise 1:

- batch size=32;
- epoch size=3;
- optimizer learning rate=3e-2;

3.3 Performance

The score is evaluated as:

$$score = (\frac{D\_model\_size}{model\_size}) * A + (\frac{model\_min\_class\_accuracy}{D\_model\_min\_accuracy}) * B + (\frac{D\_model\_parameters}{model\_parameters}) * C + (\frac{D\_epochs}{epochs}) * D \quad (3.2)$$

Where the D in apex stands for "Default"
- The coefficients are: A = 0.2, B = 0.3, C = 0.3, D= 0.2
- default model: size = 2.555MB, min class accuracy = 5.9417, parameters = 669706, epochs = 5
- default optimized model: size = 2.555MB, min class accuracy = 96.0357, parameters = 669706, epochs = 3
- optimized CNN model: size = 0.1229 MB, min class accuracy = 98.2161, parameters = 32218, epochs = 5

Results

- score against default training script from exercise 1 = 16.7922
- score against optimized training script from exercise 1 = 12.0742
- model parameters: 28938
- total weight in memory: 0.1104 MB

Accuracy per class

Test Accuracy: 98.8%, Average loss: 0.038252
Worst class accuracy is 96.7899 for class 7 - seven

Class	Accuracy [%]
class 0 - zero	99.5
class 1 - one	99.2
class 2 - two	99.6
class 3 - three	98.9
class 4 - four	98.6
class 5 - five	99.2
class 6 - six	98.9
class 7 - seven	96.8
class 8 - eight	98.8
class 9 - nine	98.6

CNN for Fashion MNIST dataset

The FashionMNIST dataset, which is a MNIST-like dataset of 28*28 gray images of clothes is a slightly more challenging dataset than MNIST dataset.
As it shares the same image size and the structure of training and testing splits the input still has

dimensions [N, C, H, W] = [32, 1, 28, 28].
The same model of the previous Neural Network is used for this dataset, but the training parameters differ in the Optimizer parameters (learning rate=0.005, momentum=0.9) and epochs (increased to 5).

3.3.1 Results

Test Accuracy: 89.0%, Average loss: 0.298976
Worst class accuracy is 75.6000 for class Shirt

Score for this exercise against default model from exercise 1 = 25.9469
Score for this exercise against optimized training script from exercise 1 = 12.1035

Class	Accuracy [%]
T-shirt/top	75.9
Trouser	99.2
Pullover	85.8
Dress	81.6
Coat	84.8
Sandal	98.7
Shirt	75.6
Sneaker	93.4
Bag	98.2
Ankle boot	97.1

As expected the FashionMNIST is more challenging to achieve a high accuracy.

CHAPTER 4

Exercise 3: design of a CNN for image classification

The goal for this exercise is to write a Neural Network for CIFAR10 dataset, that is a small dataset of 32*32*3 color (RGB) images, labeled with an integer corresponding to 1 of 10 classes.
Various architectures and parameters have been tried in order to achieve the best trade off between accuracy, model parameters and model size. To reduce the vanishing gradients problem ReLU and He weight initialization are used.
To reduce over-fitting are introduced normalization layers to handle the complexity of the NN.
Vanishing gradients can occur when optimization becomes stuck at a certain point due to a gradient that is too small to progress. Gradient clipping can prevent these gradient issues from messing up the parameters during training.

4.1 Neural Network model

The CNN model adopted uses as layers pattern the formula :

$$INPUT - [CONV - RELU - POOL] * 2 - [FC - RELU] * 1 - FC \tag{4.1}$$

In particular it's composed by 2 Convolutional layers, each followed by a Batch Normalization and a Relu activation function. There is a single CONV layer between every POOL layer. After this sequence there are two Fully connected layers to retrieve in output the final 10 classes.
With respect to the CNN of the second exercise there are some differences in the:

- Input dimensions of the data and output dimensions of the intermediate layers;
- FC layer numbers (One more FC layer followed by a Relu layer)
- Dropout layers after POOL layers
- Batch Normalization layer added after every CONV and FC layer followed by a Relu;

Dimensions

The input has dimensions [N, C, H, W] = [32, 3, 32, 32], with

- N : batch size
- C : n°channels

- H : height

- W : width

So the output 2D size of the **first Convolutional layer** will be : $\lfloor \frac{W-F+2P}{S} \rfloor + 1$ that in this case is $\lfloor \frac{32-5+2*2}{1} \rfloor + 1 = 32$, so the output volume will be 32x32x16.
The **Relu and Batch Normalization layers** don't change the size of the volume.
The following **MaxPool layer** with kernel size = 2 (F) and stride = 2 (S) and input volume $W_{in} * H_{in} * D_{in}$ will produce an output volume $W_{out} * H_{out} * D_{out}$ in order to reduce the spatial size such as:

$$W_{out} = \frac{W_{in}-F}{S} + 1 = 16 \ , \ H_{out} = \frac{W_{in}-F}{S} + 1 = 16 \ , \ D_{out} = D_{in} = 16.$$

Then another **Convolutional - Pool sequence of layers** is present:
so the output volume will be $W_{out} = H_{out} = 8, D_{out} = 32$.

Fully Connected layers

The output volume is flatten to (32 * 8 * 8) before entering in the Fully Connected layers.
The **first Fully Connected layer** takes the inputs from the feature analysis and applies weights to predict the correct label. In this layer a non linear transformation is applied to the (32 * 8 * 8) input vector and it produces an output vector of 128.
The **Fully connected output layer**, preceded by a NORM- Relu- Dropout sequence of layers, gives the final probabilities for each label. In this layer the output vector is 10, which correspond to the 10 classes.
Two-layer Fully Connected NNs may perform better than single-layer because it increases the capacity of the network.
In fact the higher the number of fully connected layers, the more complex and powerful the NN, but the higher the risks of overfitting and the higher the cost of memory, because it contains most of model parameters.

Dropout layers

Dropout is a regularization technique that prevents neural networks from overfitting by randomly dropping nodes out of the network while during training in each iteration. In fact, while regularization methods reduce overfitting by modifying the cost function, dropout modify the network itself.
It has a regularizing effect as the remaining nodes must adapt to pick-up the slack of the removed nodes; resulting in multiple independent internal representations being learned by the network.
The amount of nodes removed is specified as a parameter.In this case,Dropout layers are added after each Max Pooling layer and after the Fully Connected layer, and use a fixed dropout rate of 20% (e.g. retain 80% of the nodes).

Normalization layers

Normalization layers normalize the input data by first subtracting its mean μ, then dividing it by its standard deviation σ. Batch Normalization is a technique which takes care of normalizing the input of each layer to make the training process faster and more stable. During inference on new data, the batch normalization acts as a simple linear transformation of previous layer data.
In practice, it is an extra layer generally added after the computation layer and before the non-linearity. In our case it is added between every CONV/FC layer and Relu layer.

The normalization layers are used to build a high accuracy NN because, as the different features are on a similar scale, they help to stabilize the gradient descent step. The Batch Norm layers were added in this NN to speed-up the training process, because they allowed us to use larger learning rates/help the model converge faster for a given learning rate.

Pytorch implementation

The Pytorch class reported below shows how the Neural Network was implemented for this exercise.

```
class CNN_cifar10(nn.Module):
    def __init__(self):
        super(CNN_cifar10, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(
                in_channels=3,
                out_channels=16,
                kernel_size=5,
                stride=1,
                padding=2,
            ),
            nn.BatchNorm2d(16),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(16, 32, 5, 1, 2),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.MaxPool2d(2),
        )
        self.dense = nn.Sequential(
            nn.Linear(32 * 8 * 8, out_features=128),
            nn.BatchNorm1d(128),
            nn.ReLU(),
            )
        self.linear2 = nn.Linear(128, 10)
        self.drop25 = nn.Dropout(0.25)
        self.drop50 = nn.Dropout(0.5)
    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = self.drop25(x)
        x = x.view(x.size(0), -1)
        x = self.dense(x)
        x = self.drop50(x)
        out = self.linear2(x)
        return out
```

4.2 Optimization Setup

The optimization setup in this exercise is comprehensive of the following parameters that have been tuned to reach the optima:

- pre-processing functions (see below);
- batch size =32;
- epoch size = 40 (after this n°of epochs the accuracy remain stable);
- optimizer (see below);

Pre-processing functions

The preprocessing is implemented as a chain of transformations directly in the data loader, by the means of *transforms.Compose*. The functions applied are normalization and data augmentation.

Normalization
The dataset is normalized so that each channel of the training set has zero mean and unitary standard deviation.
By normalizing each channel with the same distribution, channel information can be updated through gradient descent using the same learning rate. Furthermore using linear activation functions around 0 ± 1, neurons tend to have nonzero gradients, so learn sooner.
The values of mean and standard deviation for the training set are computed offline and applied to the train and test set with *transforms.Normalize* function. The values found for the mean are (0.4914, 0.4822, 0.4465) per channel and standard deviation (0.2470, 0.2435, 0.2616) per channel.

Data augmentation
Data augmentation consists in making copies of the data in the training set with small random modifications.
The types of random augmentations that could be useful tend to preserve the image features with careful distortions. The most common types of augmentations include horizontal flips, shifting, zooming or cropping of the image.
The transforms applied to the train set are random flipping and random cropping, by the means of *RandomHorizontalFlip()* and *RandomCrop()* functions.
RandomHorizontalFlip() without arguments will simply randomly flip the image horizontally with probability 0.5.
RandomCrop() contains the size and padding parameters.The size parameter is a sequence (or integer) indicating the output size of RandomCrop. Padding parameter indicates the padding on each border of the image. If a sequence of length 4 is provided then the padding correspond respectively to the left, top, right and bottom borders.
Data augmentation is implemented because it has a regularizing effect as it both expands the training dataset and allows the model to learn the same general features, although in a more generalized manner.
The corresponding line is

```
1     train_transform =
2     transforms.Compose([  transforms.RandomHorizontalFlip(),
3                           transforms.RandomCrop(size=32, padding=[0, 2, 3, 4]),
4                           transforms.ToTensor(),
5                           transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2470,
      0.2435, 0.2616))
6                         ])
```

Optimizer

As well as SGD, another popular optimizer that is worth trying is the Adam optimizer:
it is an improvement to traditional gradient descent algorithms since has an adaptive learning rate method which modulates the learning rate of each weight based on the magnitudes of its gradients.
Weights that receive high gradients will have their effective learning rate reduced, while weights that receive small or infrequent updates will have their effective learning rate increased.

It is implemented as :

$$\begin{aligned} m &= \beta_1 * m + (1 - \beta_1) * \delta x \\ v &= \beta_2 * v + (1 - \beta_2) * (\delta x^2) \\ x+ &= -\frac{learning\_rate * m}{\sqrt{v} + \epsilon} \end{aligned} \tag{4.2}$$

Recommended values in the [9] are eps = 1e-8, beta1 = 0.9, beta2 = 0.999.
In practice Adam is currently recommended as the default algorithm to use, compared to other adaptive learning rate methods as Adagrad and RMSprop. Adagrad has frequent updates in the learning rate which make it better than SGD, especially with sparse data, but the updates don't get monotonically smaller; while RMSprop shows similar results of SGD with momentum, but uses another algorithm to calculate the gradients instead of the raw (and noisy) gradient vector dx.
Adam has a faster computation time and fewer parameters to tune with respect to SGD, that is hardly used in applications now due to its slow computation speed.

4.3 Performance

For this task the score is evaluated compared to a small custom model trained on an ultrabook CPU, a deeper one trained on a RTX 3090, and a ResNet20 trained exactly as described in the paper "Deep Residual Learning for Image Recognition"
The score is evaluated as:

$$\begin{aligned} score = & (\frac{default\_model\_size}{model\_size}) * A \\ & + (\frac{model\_min\_class\_accuracy}{default\_model\_min\_accuracy} + \frac{model\_test\_accuracy}{deafult\_test\_accuracy}) * B \\ & + (\frac{default\_model\_parameters}{model\_parameters}) * C \\ & + (\frac{default\_epochs}{epochs}) * D \end{aligned} \tag{4.3}$$

The values of A, B and C are A = 0.2, B = 0.3, C = 0.3, D= 0.2

- cpu model: size = 0.3965MB, min class accuracy = 73.9, test accuracy = 86.9, parameters = 103946, epochs = 100
- gpu model: size = 1.1747MB, min class accuracy = 78.6, test accuracy = 89.8, parameters = 307946, epochs = 320
- ResNet20: size = 1.1205MB, min class accuracy = 77.6, test accuracy = 89.2, parameters = 293738, epochs = 320

The cpu model was trained on a 8-core Intel laptop for 100 epochs (around 3 hours), batch size 32
The gpu models were trained on a Nvidia RTX3090 for 320 epochs (around 50 minutes), batch size 256

Results

- score against cpu model = 0.9283
- score against gpu model = 2.3809

- score against ResNet20 model = 2.3583
- model parameters: 1.0603 MB
- total weight in memory: 277962

Accuracy per class

Worst class accuracy is 58.8000 for class cat
Accuracy: 77.0%, Average loss: 0.684176

Class	Accuracy [%]
class airplane	78.5
class automobile	88.9
class bird	67.0
class cat	58.8
class deer	70.5
class dog	70.2
class frog	84.6
class horse	82.0
class ship	88.4
class truck	80.6

4.4 ResNet

Deeper NN are difficult to train due to vanishing gradient- as the gradient is back-propagated to earlier layers, repeated multiplication may make the gradient shrink to 0 and so preventing any weights update.
ResNet essentially solved this problem by using skip connections.With the skip connection, the output changes from h(x) = f(wx +b) to h(x) = f(x) + x. These skip connections help as they allow an alternate shortcut path for the gradients to flow through.

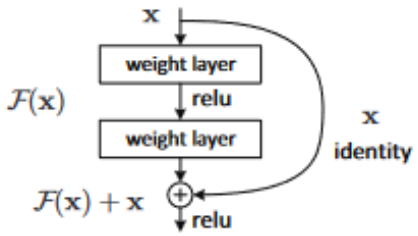


Figure 4.1: Skip connection in ResNet

There are 3 main components that make up the ResNet:

- input layer (conv1 + max pooling) (Usually referred to as layer 0)
- ResBlocks (layer1 $\sim$ layer 5)
- final layer

ResBlock is the most important layer and there are two main types of blocks used in ResNet, depending mainly on whether the input and output dimensions are the same or different.

- Identity Block: When the input and output activation dimensions are the same.
- Convolution Block: When the input and output activation dimensions are different from each other.

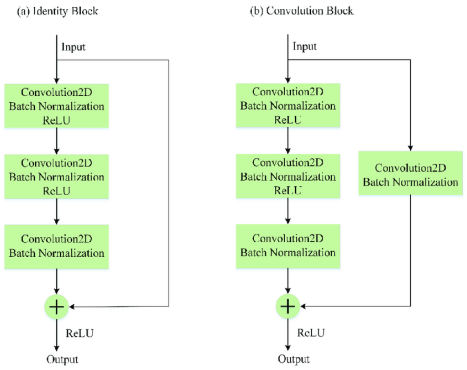


Figure 4.2: ResBlock

```
    class ResBlock(nn.Module):
    def __init__(self, in_channels, out_channels, downsample):
        super().__init__()
        if downsample:
            self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=2,
     padding=1)
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=2),
                nn.BatchNorm2d(out_channels)
            )
        else:
            self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=1,
     padding=1)
            self.shortcut = nn.Sequential()

        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1,
    padding=1)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.bn2 = nn.BatchNorm2d(out_channels)

    def forward(self, input):
        shortcut = self.shortcut(input)
        input = nn.ReLU()(self.bn1(self.conv1(input)))
        input = nn.ReLU()(self.bn2(self.conv2(input)))
        input = input + shortcut
        return nn.ReLU()(input)
```

The implemented ResNet has a *Layer0* consisting of a 7*7 convolution and a 3*3 max pooling, two resblocks and the final layer consisting of a global average pooling (gap) and a fully connected layer (fc).
It takes in input the number of channels in input, the residual block defined previously and can be set the output classes.

```
    class ResNet18(nn.Module):
    def __init__(self, in_channels, resblock, outputs=10):
```

```
        super().__init__()
        self.layer0 = nn.Sequential(
            nn.Conv2d(in_channels, 64, kernel_size=7, stride=2, padding=3),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU()
        )

        self.layer1 = nn.Sequential(
            resblock(64, 64, downsample=False),
            resblock(64, 64, downsample=False)
        )

        self.layer2 = nn.Sequential(
            resblock(64, 128, downsample=True),
            resblock(128, 128, downsample=False)
        )

        self.gap = torch.nn.AdaptiveAvgPool2d(1)
        self.fc = torch.nn.Linear(128, outputs)

    def forward(self, input):
        input = self.layer0(input)
        input = self.layer1(input)
        input = self.layer2(input)
        input = self.gap(input)
        input = input.view(input.size(0), -1)
        input = self.fc(input)

        return input
```

Results

Network trained for 10 epochs.

- score against cpu model = 2.3335
- score against gpu model = 6.8670
- score against ResNet20 model = 6.8597
- model parameters: 685322
- total weight in memory: 2.6143 MB

Accuracy per class

Worst class accuracy is for class bird is 62.8 %
Accuracy: 78.7 %, Average loss: 0.630265

Class	Accuracy [%]
class airplane	87.9
class automobile	85.7
class bird	62.8
class cat	68.8
class deer	79.9
class dog	63.4
class frog	89.7
class horse	73.6
class ship	90.6
class truck	84.3

CHAPTER 5

Exercise 4: quantization of previous CNN models with fake-quantize methods, port to brevitas and export to FINN

In deployment scenario, where the CNN model has to be executed on a resource constrained device with low energy consumption, floating points units are not feasible and usually not available. Possible solution is to quantize operators, in this exercises the models for Fashion-MNIST and CIFAR10 are re-written using only quantized operators.
Quantization is an important technique since NN high accuracy is often achieved by over-parametrizing them so there's ample opportunity for reducing bit-precision without impacting accuracy.
The training loop is the same as the one used previously, so

- batch size = 32;
- epoch size = 5;

Performances evaluated as :

$$\begin{aligned} default\_score =& \frac{2.55}{memory} * 0.2 + \frac{class\_accuracy_{min}}{5.9417} * 0.3 + \frac{669706.0}{params} * 0.3 + \frac{5}{epochs} * 0.2 \\ optimized\_score =& \frac{2.55}{memory} * 0.2 + \frac{class\_accuracy_{min}}{96.0357} * 0.3 + \frac{669706.0}{params} * 0.3 + \frac{3}{epochs} * 0.2 \end{aligned} \tag{5.1}$$

5.1 Quantized operators with Pytorch custom classes

The CNNs are re-written re-written using the custom layers provided in *quantization.py*, the best trade-off between model size and accuracy has been obtained using an 8-bit quantizations in all layers (int8 datatype). Uniform integer quantization enables to compute matrix multiplications and convolutions in the integer domain, allowing to use high throughput integer math pipelines.
In uniform quantization to map input value that lies x$\in[\alpha,\beta]$,from a large set, to a small set [$-2^b - 1$, $2^b - 1$-1] there are only two possible transformation functions : f(x) = s * x + z and its special case f(x) = s * x, where x, s, z $\in$ R. These two choices are called affine and scale, respectively.
Affine quantization

$$\begin{aligned} s =& \frac{2^b - 1}{\alpha - \beta} \\ z =& - round(\beta * s) - 2^{b-1} \end{aligned} \tag{5.2}$$

where s is the scale factor while z is the zero point to which the real value 0 is mapped. In the 8-bit case, $\frac{255}{\alpha-\beta}$, z = -round(β*s).
The quantize operation is then defined as :

$$\text{clip}(x, l, u) \begin{cases} l, & x < l \\ x, & l \leq x \leq u \\ u, & x > u \end{cases}$$

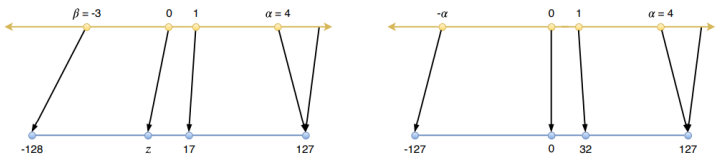
$$x_q = \text{quantize}(x, b, s, z) = \text{clip}(\text{round}(s \cdot x + z), -2^{b-1}, 2^{b-1} - 1)$$

where round() rounds to the nearest integer.

Scale quantization

Scale quantization performs range mapping with only a scale transformation, in the symmetric variant of scale quantization the input range and integer range are symmetric around zero. This means that for int8 we use the integer range [-127, 127], opting not to use the value -128 in favor of symmetry. For 8-bit quantization, losing one out of 256 representable values is insignificant, but for lower bit quantization the trade-off between representable values and symmetry should be re-evaluated.
Scale quantization of a real value x, with a chosen representable range [-α, α], producing a b-bit integer value x_q

$$\begin{aligned} s &= \frac{2^b - 1}{\alpha} \\ x_q = quantize(x, b, s) &= clip(round(s * x), -2^{b-1} + 1, 2^{b-1} - 1) \end{aligned} \tag{5.3}$$



(a) Affine quantization (b) Scale quantization

Figure 5.1: Quantization mapping of real values to int8

While both affine and scale quantization enable the use of integer arithmetic, affine quantization leads to more computationally expensive inference.

PArameterized Clipping acTivation (PACT)

PACT is a technique to quantize activations, it enables neural networks to work well with ultra low precision weights and activations without any significant accuracy degradation. PACT uses an activation clipping parameters α that is optimized during training to find the right quantization scale.
Basically ReLu layer is replaced with the following:

$$y = PACT(x) = 0.5(|x| - |x - \alpha| + \alpha) = \begin{cases} 0, & x \in (-\infty, 0) \\ x, & x \in [0, \alpha) \\ \alpha, & x \in [\alpha, +\infty) \end{cases}$$

Figure 5.2: PACT equation.

Where α is the clipping level (range of activation [0,α] that is dinamically adjusted via gradient-descent training.
The higher the α the more it resembles a ReLu Actfn, better to use large value of α and apply regularization to reduce it during training. Below is shown how α changes during full-precision training of CIFAR10 ResNet20 starting with an initial value of 10 and using the L2-regularization. It can be observed that α converges to a value much smaller than initial value as training proceeds.

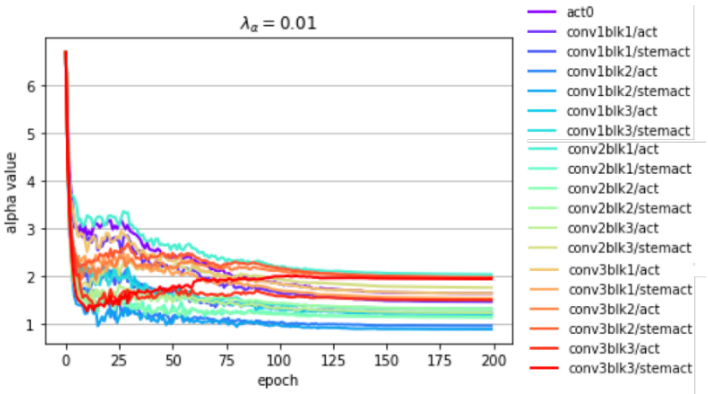


Figure 5.3: Evolution of α during training on CIFAR-10

Another important trick to reduce significantly performance degradation due to quantization is to leave the first and last layers un-quanitzed.

Normalization layer folding
The normalization layer used in the Cifar10 NN is the Batch Norm, which normalizes the input of each layer by first subtracting to the batch its mean μ, then dividing it by its standard deviation σ, as in the equation 5.4. Further scale and shift are not required because the data have been preprocessed to have to a mean of 0 and a standard deviation of 1.

$$\begin{aligned} \mu_{\mathcal{B}} &\leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \\ \sigma_{\mathcal{B}}^2 &\leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \\ \widehat{x}_i &\leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \\ y_i &\leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \end{aligned} \tag{5.4}$$

During inference, the batch normalization acts as a simple linear transformation of a convolution, that is also a linear transformation.

In order to avoid using FP32 arithmetic both operations can be merged into a single linear transformation by folding the normalization layer into convolution: the weight and bias of the convolution are rearranged as in the equation 5.5.

$$\begin{aligned} w_{\text{fold}} &= \gamma \cdot \frac{W}{\sqrt{\sigma^2 + \epsilon}} \\ b_{\text{fold}} &= \gamma \cdot \frac{b - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta \end{aligned} \tag{5.5}$$

In fact the convolution operation followed by the batch normalization operation can be expressed, for an input x, as 5.6:

$$\begin{aligned} z &= W * x + b \\ \text{out} &= \gamma \cdot \frac{z - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta \end{aligned} \tag{5.6}$$

The batch norm layers can been replaced by an Identity layer, and the weights of the convolutions have been modified accordingly to the equation 5.5. The convolution bias is included in this formula, but it is not useful as the convolutional layer is followed by the normalization. Thus, the bias term is set to zero. By doing this folding the model will have fewer parameters nd faster inference without any performance degradation.

QUANTIZED CNN FASHION-MNIST

```
from torch import nn
from torch.nn import Module
import quantization as cs
import torch.nn.functional as F

########## FMNIST NN quantized with Pytorch custom ##########
#PACT WITH QUANTIZATION
[conv>relu>maxpool]x2->flatten>linear
class Ex4_FMNIST(nn.Module):
    def __init__(self):
        super(Ex4_FMNIST, self).__init__()
        self.weight_bit = 8
        self.act_bit = 8
        self.bias_bit = 8
        self.quant_method = 'scale'
        self.alpha_coeff = 10.0
        self.quantization = True
        self.conv1 = cs.Conv2d(in_channels=1, out_channels=16, kernel_size=(5, 5),
                        stride=(1, 1), padding=(2, 2), act_bit=self.act_bit,
                        weight_bit=self.weight_bit, bias_bit=self.bias_bit,
                        quantization=self.quantization, quant_method=self.quant_method
    )
        self.conv2 = cs.Conv2d(in_channels=16, out_channels=32, kernel_size=(5, 5),
                        stride=(1, 1), padding=(2, 2), act_bit=self.act_bit,
                        weight_bit=self.weight_bit, bias_bit=self.bias_bit,
                        quantization=self.quantization, quant_method=self.quant_method
    )
        self.act = cs.ReLu(alpha=self.alpha_coeff, act_bit=self.act_bit,
                        quantization=self.quantization)
        self.dense = cs.Linear(in_channels=32*7*7, out_channels=10, act_bit=self.
    act_bit,
                        weight_bit=self.weight_bit, bias_bit=self.bias_bit,
                        quantization=self.quantization, quant_method=self.quant_method
    )
        self.pool = nn.MaxPool2d(2)

```

```
33
34      def forward(self, inputs):
35          if self.quantization:
36              inputs = cs.quantization_method[self.quant_method](inputs, -2 **
37                          (self.act_bit-1) + 1, 2 ** (self.act_bit-1) - 1)
38
39          out = self.conv1(inputs)
40          out = self.act(out)
41          out = self.pool(out)
42          out = self.conv2(out)
43          out = self.act(out)
44          out = self.pool(out)
45          out = out.reshape(out.shape[0], -1)
46          out = self.dense(out)
47          return out
```

ACCURACY RESULTS

FashionMNIST un-quantized	**FashionMNIST with custom quantized classes**
Optimizer SGD with learning rate=0.075	**Optimizer** SGD with learning rate=0.075
Score against default model from exercise 1 = 23.47224 against optimized training script from exercise 1 = 12.0544	**Score** against default model from exercise 1 = 23.4173 against optimized training script from exercise 1 = 12.1741
Test Accuracy: 89.4%, Average loss: 0.300765	**Test** Accuracy: 87.1%, Average loss: 0.358906
Accuracy per class Accuracy for class T-shirt/top is: 90.3 % Accuracy for class Trouser is: 97.4 % Accuracy for class Pullover is: 92.4 % Accuracy for class Dress is: 93.5 % Accuracy for class Coat is: 70.5 % Accuracy for class Sandal is: 97.9 % Accuracy for class Shirt is: 62.3 % Accuracy for class Sneaker is: 95.6 % Accuracy for class Bag is: 98.6 % Accuracy for class Ankle boot is: 95.9 %	**Accuracy per class** Accuracy for class T-shirt/top is: 84.7 % Accuracy for class Trouser is: 97.5 % Accuracy for class Pullover is: 87.9 % Accuracy for class Dress is: 89.4 % Accuracy for class Coat is: 67.7 % Accuracy for class Sandal is: 95.8 % Accuracy for class Shirt is: 61.4 % Accuracy for class Sneaker is: 94.6 % Accuracy for class Bag is: 96.4 % Accuracy for class Ankle boot is: 95.7 %
Worst class accuracy Worst class accuracy is 62.3000 for class Shirt	**Worst class accuracy** Worst class accuracy is 61.4000 for class Shirt

Table 5.1: Performance for FashionMNIST quantized with custom Pytorch classes

5.1.1 QUANTIZED CNN CIFAR-10

```
1  ###### CIPHAR-10 NN quantized with Pytorch custom ######
2  class CIFAR10_quant(nn.Module):
3      def __init__(self):
4          super(CIFAR10_quant, self).__init__()
5          self.weight_bit = 8
6          self.act_bit = 8
7          self.bias_bit = 8
8          self.quant_method = 'scale'
9          self.alpha_coeff = 10.0
10
11         self.quantization = True
12
13         self.conv1 = cs.Conv2d(in_channels=3, out_channels=16, kernel_size=(5, 5),
       stride=(1, 1),
14                                padding=(2, 2),
```

```
                                act_bit=self.act_bit, weight_bit=self.weight_bit,
    bias_bit=self.bias_bit,
                                quantization=self.quantization, quant_method=self.
    quant_method)
        self.conv2 = cs.Conv2d(in_channels=16, out_channels=32, kernel_size=(5, 5),
    stride=(1, 1),
                                padding=(2, 2),
                                act_bit=self.act_bit, weight_bit=self.weight_bit,
    bias_bit=self.bias_bit,
                                quantization=self.quantization, quant_method=self.
    quant_method)

        self.act1 = cs.ReLu(alpha=self.alpha_coeff, act_bit=self.act_bit, quantization
    =self.quantization)
        self.act2 = cs.ReLu(alpha=self.alpha_coeff, act_bit=self.act_bit, quantization
    =self.quantization)
        self.act3 = cs.ReLu(alpha=self.alpha_coeff, act_bit=self.act_bit, quantization
    =self.quantization)
        self.flatten = nn.Flatten()
        self.dense1 = cs.Linear(in_channels=32 * 8 * 8, out_channels=128, act_bit=self
    .act_bit,
                                weight_bit=self.weight_bit,
                                bias_bit=self.bias_bit, quantization=self.quantization
    ,
                                quant_method=self.quant_method)
        self.dense2 = cs.Linear(in_channels=128, out_channels=10, act_bit=self.act_bit
    ,
                                weight_bit=self.weight_bit,
                                bias_bit=self.bias_bit, quantization=self.quantization
    ,
                                quant_method=self.quant_method)
        self.pool = nn.MaxPool2d(2)
        self.batch16 = nn.BatchNorm2d(16)
        self.batch32 = nn.BatchNorm2d(32)
        self.batch128 = nn.BatchNorm1d(128)
        self.drop1 = nn.Dropout(0.25)
        self.drop2 = nn.Dropout(0.5)

    # [conv>batchnomr>relu>maxpool]x2>dropout25->flatten>linear>batchnorm>relu>
    dropout50>linear
    def forward(self, x):
        if self.quantization:
            x = cs.quantization_method[self.quant_method](x, -2 ** (self.act_bit - 1)
    + 1, 2 ** (self.act_bit - 1) - 1)

        out = self.conv1(x)
        out = self.batch16(out)
        out = self.act1(out)
        out = F.max_pool2d(out, 2)
        out = self.conv2(out)
        out = self.batch32(out)
        out = self.act2(out)
        out = F.max_pool2d(out, 2)
        out = self.drop1(out)
        out = self.flatten(out)
        out = self.dense1(out)
        out = self.batch128(out)
        out = self.act3(out)
        out = self.drop2(out)
        out = self.dense2(out)
        return out
```

ACCURACY RESULTS

CIFAR-10 un-quantized	CIFAR-10 with custom quantized classes
Optimizer Adam with learning rate = 0.001, β = [0.9, 0.999], $\epsilon = 10^{-7}$	**Optimizer** Adam with learning rate = 0.001, β = [0.9, 0.999], $\epsilon = 10^{-7}$
Score against cpu model = 0.9186 against gpu model = 2.3718 against ResNet20 model = 2.3490	**Score** against cpu model = 0.9337 against gpu model = 2.3860 against ResNet20 model = 2.3634
Test Accuracy: 76.4 %, Average loss: 0.684244	**Test** Accuracy: 78.2%, Average loss: 0.643598
Accuracy per class Accuracy for class airplane is : 81.8 % Accuracy for class automobile is : 86.0 % Accuracy for class bird is : 61.3 % Accuracy for class cat is : 56.4 % Accuracy for class deer is : 70.4 % Accuracy for class dog is : 71.8 % Accuracy for class frog is : 84.9 % Accuracy for class horse is : 82.7 % Accuracy for class ship is : 86.8 % Accuracy for class truck is : 82.4%	**Accuracy per class** Accuracy for class airplane is: 77.0 % Accuracy for class automobile is: 86.4 % Accuracy for class bird is: 62.8 % Accuracy for class cat is: 60.1 % Accuracy for class deer is: 79.0 % Accuracy for class dog is: 71.7 % Accuracy for class frog is: 87.9 % Accuracy for class horse is: 77.8 % Accuracy for class ship is: 90.0 % Accuracy for class truck is: 89.5 %
Worst class accuracy Worst class accuracy is 56.4 % for class cat	**Worst class accuracy** Worst class accuracy is 60.1 % for class cat

Table 5.2: Performance for quantized CNN CIFAR-10 with custom Pytorch classes

5.2 Quantize networks using Brevitas classes

The goal of this project is lastly to generate from the Quantized Neural Network (QNN) an highly efficient FPGA accelerator, in order to do so is used the FINN compiler. The approach suggested by FINN is to first train the QNN in Brevistas and then export to FINN-ONNX.
Brevitas is a PyTorch library for quantization-aware training, it implements a set of building blocks at different levels of abstraction to model a reduced precision hardware data-path at training time. The CNN model is trained to perform inference with quantized weights and activations, using the quantization-aware training (QAT) capabilities offered by Brevitas.

5.2.1 BREVITAS QUANTIZED CNN FMNIST

```
from torch import nn
from torch.nn import Module
import quantization as cs
import brevitas.nn as qnn
from brevitas.quant import Int8Bias as BiasQuant
import torch.nn.functional as F
from brevitas.core.quant import QuantType
from brevitas.quant import Uint8ActPerTensorFloat as ActQuant
from brevitas.quant import Int8WeightPerTensorFloat as WeighQuant
class Ex4_FMNIST_brevitas(nn.Module):
    def __init__(self):
        super(Ex4_FMNIST_brevitas, self).__init__()
        self.weight_bit = 8
        self.act_bit = 8
        self.bias_bit = 8
        self.alpha_coeff = 10.0
```

```
        self.quant_inp = qnn.QuantIdentity(bit_width=self.act_bit,
                           return_quant_tensor=True)
        self.conv1 = qnn.QuantConv2d(in_channels=1, out_channels=16, kernel_size=(5,
    5),
                           stride=(1, 1), padding=(2, 2), weight_bit_width=self.
    weight_bit, bias=False,
                           bias_quant=BiasQuant, return_quant_tensor=True)
        self.conv2 = qnn.QuantConv2d(in_channels=16, out_channels=32, kernel_size=(5,
    5),
                           stride=(1, 1), padding=(2, 2), weight_bit_width=self.
    weight_bit,bias=False,
                           bias_quant=BiasQuant, return_quant_tensor=True)
        self.act1 = qnn.QuantReLU(bit_width=self.act_bit, return_quant_tensor=True)
        self.act2 = qnn.QuantReLU(bit_width=self.act_bit, return_quant_tensor=True)
        self.flatten=nn.Flatten(),
        self.dense = qnn.QuantLinear(in_features=32*7*7, out_features=10, bias=True,
                           weight_bit_width=self.weight_bit,bias_quant=BiasQuant,
                           return_quant_tensor=False)
    def forward(self, x):
        out = self.quant_inp(x)
        out = self.act1(self.conv1(out))
        out = self.pool(out)
        out = self.act2(self.conv2(out))
        out = self.pool(out)
        out = self.flatten(out)
        out = self.dense(out)
        return out
```

Results

FashionMNIST with custom classes	**FashionMNIST with Brevitas classes**
Optimizer SGD with learning rate=0.075	**Optimizer** SGD with learning rate=0.005, momentum=0.9
Score against default model from exercise 1 = 23.4173 against optimized training script from exercise 1 = 12.1741	**Score** against default model from exercise 1 = 24.3332 against optimized training script from exercise 1 = 12.0646
Test Accuracy: 87.1%, Average loss: 0.358906	**Test** Accuracy: 89.7%, Average loss: 0.299013
Accuracy per class Accuracy for class T-shirt/top is: 84.7 % Accuracy for class Trouser is: 97.5 % Accuracy for class Pullover is: 87.9 % Accuracy for class Dress is: 89.4 % Accuracy for class Coat is: 67.7 % Accuracy for class Sandal is: 95.8 % Accuracy for class Shirt is: 61.4 % Accuracy for class Sneaker is: 94.6 % Accuracy for class Bag is: 96.4 % Accuracy for class Ankle boot is: 95.7 %	**Accuracy per class** Accuracy for class T-shirt/top is: 81.8 % Accuracy for class Trouser is: 98.2 % Accuracy for class Pullover is: 84.7 % Accuracy for class Dress is: 89.6 % Accuracy for class Coat is: 88.1 % Accuracy for class Sandal is: 98.2 % Accuracy for class Shirt is: 67.0 % Accuracy for class Sneaker is: 96.9 % Accuracy for class Bag is: 97.9 % Accuracy for class Ankle boot is: 94.4 %
Worst class accuracy Worst class accuracy is 61.4000 for class Shirt	**Worst class accuracy** Worst class accuracy is 67.0000 for class Shirt

Table 5.3: Performance for FashionMNIST

5.2.2 BREVITAS QUANTIZED CNN CIFAR-10

```
from torch import nn
from torch.nn import Module
import quantization as cs
import brevitas.nn as qnn
from brevitas.quant import Int8Bias as BiasQuant
import torch.nn.functional as F
from brevitas.core.quant import QuantType
from brevitas.quant import Uint8ActPerTensorFloat as ActQuant
from brevitas.quant import Int8WeightPerTensorFloat as WeighQuant
class cifar_quant_brevitas(nn.Module):
    def __init__(self):
        super(cifar_quant_brevitas, self).__init__()
        self.weight_bit = 8
        self.act_bit = 8
        self.bias_bit = 8

        self.alpha_coeff = 10.0

        self.quant_inp = qnn.QuantIdentity(bit_width=self.act_bit, return_quant_tensor=True)
        self.conv1 = qnn.QuantConv2d(in_channels=3, out_channels=16, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2),
                                     bias=False, weight_bit_width=self.weight_bit, weight_quant=WeighQuant,
                                     bias_quant=BiasQuant, return_quant_tensor=True)
        self.relu1 = qnn.QuantReLU(
            bit_width=self.act_bit, act_quant=ActQuant, return_quant_tensor=True)
        self.conv2 = qnn.QuantConv2d(in_channels=16, out_channels=32, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2),
                                     bias=False, weight_bit_width=self.weight_bit, bias_quant=BiasQuant,
                                     weight_quant=WeighQuant, return_quant_tensor=True)
        self.relu2 = qnn.QuantReLU(
            bit_width=self.act_bit, act_quant=ActQuant, return_quant_tensor=True)
        self.relu3 = qnn.QuantReLU(
            bit_width=self.act_bit, act_quant=ActQuant, return_quant_tensor=True)

        self.dense = qnn.QuantLinear(in_features=32*8*8, out_features=128, bias=False, weight_bit_width=self.weight_bit,
                                     weight_quant=WeighQuant, bias_quant=BiasQuant, return_quant_tensor=True)
        #self.pool = qnn.QuantMaxPool2d(2)
        self.flatten = nn.Flatten()
        self.fc1 = qnn.QuantLinear(in_features=128, out_features=10, bias=True, weight_bit_width=self.weight_bit,
                                     weight_quant=WeighQuant, bias_quant=BiasQuant, return_quant_tensor=False)
        self.pool = qnn.QuantMaxPool2d(2)
        self.drop1 = qnn.QuantDropout(p=0.25)
        self.drop2 = qnn.QuantDropout(p=0.5)
        self.drop3 = qnn.QuantDropout(p=0.2)
        self.batch16 = nn.BatchNorm2d(16)
        self.batch32 = nn.BatchNorm2d(32)
        self.batch64 = nn.BatchNorm1d(64)
        self.batch128 = nn.BatchNorm1d(128)
    def forward(self, x):
        out = self.quant_inp(x)
        out = self.conv1(out)
        out = self.batch16(out)
        out = self.relu1(out)
        out = F.max_pool2d(out, kernel_size=2, stride=2)
```

```
        out = self.conv2(out)
        out = self.batch32(out)
        out = self.relu2(out)
        out = F.max_pool2d(out, kernel_size=2, stride=2)
        out = self.drop1(out)
        out = out.view(x.size(0), -1)
        out = self.dense(out)
        out = self.batch128(out)
        out = self.relu3(out)
        out = self.drop2(out)
        out = self.fc1(out)
        return out
```

Results

CIFAR-10 with custom quantized classes	**CIFAR-10 with Brevitas classes**
Optimizer Adam with learning rate = 0.001, β = [0.9, 0.999], $\epsilon = 10^{-7}$	**Optimizer** Adam with learning rate = 0.001, β = [0.9, 0.999], $\epsilon = 10^{-7}$
Score against cpu model = 0.9337 against gpu model = 2.3860 against ResNet20 model = 2.3634	**Score** against cpu model = 0.9373 against gpu model = 2.3894 against ResNet20 model = 2.3669
Test Accuracy: 78.2%, Average loss: 0.643598	**Test** Accuracy: 77.6 %, Average loss: 0.656111
Accuracy per class Accuracy for class airplane is: 77.0 % Accuracy for class automobile is: 86.4 % Accuracy for class bird is: 62.8 % Accuracy for class cat is: 60.1 % Accuracy for class deer is: 79.0 % Accuracy for class dog is: 71.7 % Accuracy for class frog is: 87.9 % Accuracy for class horse is: 77.8 % Accuracy for class ship is: 90.0 % Accuracy for class truck is: 89.5 %	**Accuracy per class** Accuracy for class airplane is: 82.2 % Accuracy for class automobile is: 90.0 % Accuracy for class bird is: 63.6 % Accuracy for class cat is: 61.0 % Accuracy for class deer is: 74.7 % Accuracy for class dog is: 63.4 % Accuracy for class frog is: 83.4 % Accuracy for class horse is: 82.8 % Accuracy for class ship is: 88.3 % Accuracy for class truck is: 86.6 %
Worst class accuracy Worst class accuracy is 60.1 for class cat	**Worst class accuracy** Worst class accuracy is 61 for class cat

Table 5.4: Performance for CIFAR10 quantized with Brevitas

5.3 Exporting in ONNX format

ONNX is an open format built to represent machine learning models, and the FINN compiler expects an ONNX model as input. Two of the Brevitas-exported ONNX variants that can be ingested by FINN:

- FINN-ONNX: Quantized weights exported as tensors with additional attributes to mark low-precision datatypes. Quantized activations exported as MultiThreshold nodes.
- QONNX: All quantization is represented using Quant, BinaryQuant or Trunc nodes. QONNX must be converted into FINN-ONNX by finn.transformation.qonnx.convert_qonnx_to_finn

The only model exported to FINN is the fashion MNIST, lastly to export it :

```
from brevitas.export import FINNManager, BrevitasONNXManager

    model = quantModel()
    ...
    ## training ##
    ## validate ##
    ...
    in_tensor = (1, 1, 28, 28)
    build_dir = os.getcwd() + "/finn_model/"+'model_MNIST_quant_'
    FINNManager.export(model.to("cpu"), input_shape=in_tensor, export_path=build_dir +
     '_finn.onnx')
    BrevitasONNXManager.export(model.cpu(), input_shape=in_tensor, export_path=
    build_dir + '_brevitas.onnx')
```

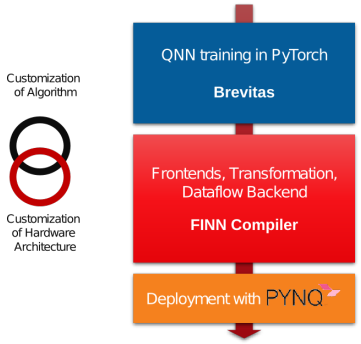
Lastly the ONNX model can be visualized with Netron, which is a visualizer for neural networks and allows interactive investigation of network properties.

```
    from finn.util.visualization import showInNetron
    showInNetron(model_filename)
```

CHAPTER 6

Exercise 5: deployment of a small CNN to a custom accelerator generated with FINN

FINN is an experimental framework from Xilinx Research Labs to explore deep neural network inference on FPGAs. It specifically targets quantized neural networks, with emphasis on generating dataflow-style architectures customized for each network. The resulting FPGA accelerators are highly efficient and can yield high throughput and low latency. The key idea in such architectures is to parallelize across layers as well as within layers by dedicating a proportionate amount of compute resources to each layer. This is done by mapping each layer to a Vivado HLS description, parallelizing each layer's implementation to the appropriate degree and using on-chip FIFOs to link up the layers to create the full accelerator.



The accelerator is build upon the brevitas network of FMNIST dataset, see appendix 8.

Dataset	FMNIST with input (1,28,28)
Optimzer	Adam with learning rate = 0.001, $\beta = [0.9, 0.999]$, $\epsilon = 10^{-7}$
Epoch	10
Loss	0.239489
Accuracy	89.83 %

The network is trained as in 8 and some changes were applied in order to meet all constraints given by the hardware generation. As we can see in Fig.6.1 we can visualize the generated network in Netron.

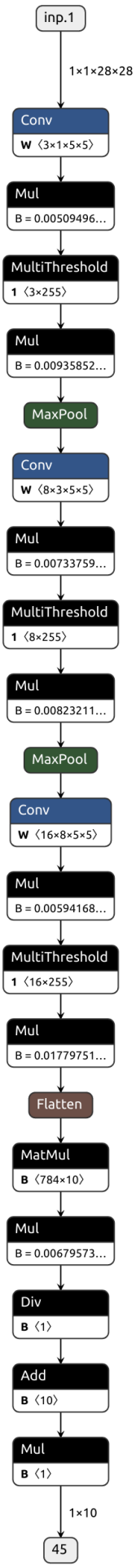


Figure 6.1: Netron representation of the exported NN.

6.1 FINN verification

After the network is trained, it is imported into FINN and verified with FINN compiler (8). To start the FINN compiler first have to be set the following environment variables :

$ export FINN_XILINX_PATH = <installation path of Xilinx tool>

$ export FINN_XILINX_VERSION = <Xilinx verion>

$ export PLATFORM_REPO_PATH = <path to Vitis platform files>

$ export FINN_HOST_BUILD_DIR = <set to your build directory>

Before of running the verification, the FINN-ONNX model is prepared to make it easier to process by adding pre and post processing directly in the ONNX graph. In this case, the preprocessing step divides the input uint8 data by 255 so the inputs are bounded between [0, 1]. The postprocessing step takes the output of the network and returns the index (0-9) of the image category with the highest probability (top-1).
The model preparation consist also in tidy-up transformations: all the intermediate tensors need to have statically defined shapes to transform the model into a synthesizable hardware description. Following the tutorial at the link[1] we applied the streamlining transformations, partitioning, conversion to HLS Layers, by anticipating the building process.
Streamlining transformations are a series of mathematical simplifications that allow us to get rid of floating point scaling operations. In fact in FINN, convolutions are implemented with the lowering approach: they are converted into matrix-matrix multiply operations, where one of the matrices is generated by sliding a window over the input image.
At the end the layers that can put into the FPGA are converted into their HLS equivalents and separated into a dataflow partition.

Lastly to verify the model are executed same inputs on Brevitas and FINN model, used the *finn.core.onnx_exec* function to execute the exported FINN-ONNX on the inputs (only for verification).

6.2 Building

Verified that the model is correct, it can be build the accelerator. The FINN compiler has a build_dataflow tool which takes in input the specific build info as a configuration dict and then invokes all the necessary steps to make the dataflow build happen. The build configuration is specified by an instance of *finn.builder.build_dataflow_config.DataflowBuildConfig*, to configure the performance (and correspondingly, the FPGA resource footprint) of the generated dataflow accelerator there are two ways :

- basic : Set a target performance and let the compiler figure out the per-node parallelization settings.
- advanced : Specify a separate .json as folding_config_file that lists the degree of parallelization (as well as other hardware options) for each layer

In our case we dealt only with the basic approach, for which are needed to setup only

- target_fps: target inference performance in frames per second. Note that target may not be achievable due to specific layer constraints, or due to resource limitations of the FPGA.

- synth_clk_period_ns: target clock frequency (in nanoseconds) for Vivado synthesis. e.g. synth_clk_period_ns=5.0 will target a 200 MHz clock. Note that the target clock period may not be achievable depending on the FPGA part and design complexity.

6.2.1 Launch a build : only estimate reports

The build can produce many different outputs, and some of them can take a long time (e.g. bitfile synthesis for a large network). Initially when working on generating a new accelerator and exploring the different performance options generating the bitfile is not recommended. Thus, in the beginning is preferred to just select the estimate reports as the output products.
As can be seen in 8 the generate_outputs only contains ESTIMATE_REPORTS, also the steps uses a value of estimate_only_dataflow_steps that skips steps like HLS synthesis to provide a quick estimate from analytical model
In folder "*estimates_output_dir*" there are various reports generated as .json file, among them can be check :

file	description	value
estimate_network_performance.json	analytical estimates for the performance and latency	$critical\_path\_cycles$: 96444, max_cycles : 19760, $max\_cycles\_node\_name$: "$ConvolutionInputGenerator_0$", $estimated\_throughput\_fps$: 5060.728744939272, $estimated\_latency\_ns$: 964440.0
estimate_layer_cycles.json	estimated number of clock cycles each layer will take	$'FMPadding\_Batch\_0'$: 1024, $'ConvolutionInputGenerator\_0'$: 19760, $'StreamingFCLayer\_Batch\_0'$: 11760, $'StreamingMaxPool\_Batch\_0'$: 1176, $'FMPadding\_Batch\_1'$: 972, $'ConvolutionInputGenerator\_1'$: 14970, $'StreamingFCLayer\_Batch\_1'$: 7840 $'StreamingMaxPool\_Batch\_1'$: 294, $FMPadding\_Batch\_2'$: 968, $'ConvolutionInputGenerator\_2'$: 10240, $'StreamingFCLayer\_Batch\_2'$: 19600, $'StreamingFCLayer\_Batch\_3'$: 7840
estimate_layer_resources.json	total resource estimates	$'BRAM\_18K'$: 12.0, $'LUT'$: 44044.0, $'URAM'$: 0.0, $'DSP'$: 0.0

Table 6.1: Estimated reports

FINN attempts to parallelize each layer such that they all take a similar number of cycles, and less than the corresponding number of cycles that would be required to meet target_fps. Additionally by summing up all layer cycle estimates one can obtain an estimate for the overall latency of the whole network.
To determine whether the current configuration will fit into a particular FPGA can be check the the resource requirements in each layer (file "estimate_layer_resources.json"). If too high for the FPGA then a solution could be lower the target_fps.

6.2.2 Launch a Build: generating the accelerator

There are different ways to generate the accelerator, since the goal is to generate an IP component that can be used in other projects the STITCHED_IP output product is the correct choice. This build last circa 10-15 minutes and at the end the accelerator will be exported as a stitched IP block design.

Among the reports generated by the output products, different from the ones generated by ESTIMATE_REPORTS:

file	description	value
ooc_synth_and_timing.json	post-synthesis and maximum clock frequency estimate for the accelerator	
rtlsim_performance.json	steady-state throughput and latency for the accelerator	"cycles": 46680, "runtime[ms]": 0.4668, "throughput[images/s]": 2142.245, "DRAM_in_bandwidth[Mb/s]": 1.679, "DRAM_out_bandwidth[Mb/s]": 0.085, "fclk[mhz]": 100.0, "N": 1, "latency_cycles": 46680

Table 6.2: IP reports

The node-by-node hardware configuration determined by the FINN compiler, including FIFO depths, parallelization settings (PE/SIMD) and others is reported in file final_hw_config.json. To further optimize the build (the "advanced" method aforementioned) can be used this .json file as the folding_config_file for a new run to use it as a starting point for further exploration and optimizations.

6.2.3 Launch a build : generate the ZYNQ bitfile

Lastly can be generated the bitfile for the FPGA, in our case has been generated for ZCU104 development board. This final build other than generating the bitfile will generate also the python driver that lets us execute the accelerator on PYNQ platforms with simply numpy i/o.
In the end all the files needed to be copied on the target board will be contained in the *deploy* folder. To test the accelerator on the board, we'll put a copy of the dataset and a premade Python script that validates the accuracy into the driver folder, then make a zip archive of the whole deployment folder.

6.3 Deployment on ZYNQ board

To connect to the board check the link, in our case the he IP name of the pynq board is pynq-zcu104, IP address is 192.168.166.58 and so to connect simply connect navigate to link "*http: 192.168.166.58*. After connecting a notebook is runned 8 where the deploy folder previously generated is unzipped and then is run the *validate.py*file to check the accelerator correctness8.
In our case the final accuracy is 89.69 %.
The metrics of the implemented accelerator :

runtime[ms]	6.882190704345703
throughput[images/s]	4649.682255941246
DRAM_in_bandwidth[Mb/s]	3.6453508886579367
DRAM_out_bandwidth[Mb/s]	0.18598729023764982
fclk[mhz]	99.999
batch_size	32
fold_input[ms]	0.05435943603515625
pack_input[ms]	0.05507469177246094
copy_input_data_to_device[ms]	0.30541419982910156
copy_output_data_from_device[ms]	0.10704994201660156
unpack_output[ms]	86.59172058105469
unfold_output[ms]	0.032901763916015625

Table 6.3: Metrics on ZCU104 board

CHAPTER 7

Conclusion

Nowadays Neural Networks (NN) -enabled machines are becoming fundamental to solve complex problem. One class of NNs used in computer vision problems is the Convolutional NN, where is made the explicit assumption that the inputs are images. However a good accuracy of CNNs comes at the cost of high computational complexity that makes them infeasible for deployment on resource-constrained devices.
Since CNNs are usually trained on GPUs using floating points precision for the data a solution is to quantize operators that allows inference to be carried out using integer-only arithmetic, which can be implemented more efficiently than floating point inference on commonly available integer-only hardware. However too aggrevise quantization may lead to performance degradation.
Xilinx's researchers developed a framework called Brevitas, built upon Pytorch, to train low bit-width quantized CNNs with the target accelerator in mind. The trained CNN model is then passed to another framework called FINN (Xilinx again) which generates an accelerator based on a systolic array that can execute the network efficiently, respecting the performance targets determined by the designer. The result is a bitstream that can be mounted on a FPGA accelerator, providing an easy way to design and deploy efficient CNNs.

In this project has been designed and implemented a quantized CNN for fashion-MNIST dataset that has the best trade off between accuracy, model parameters and model size.

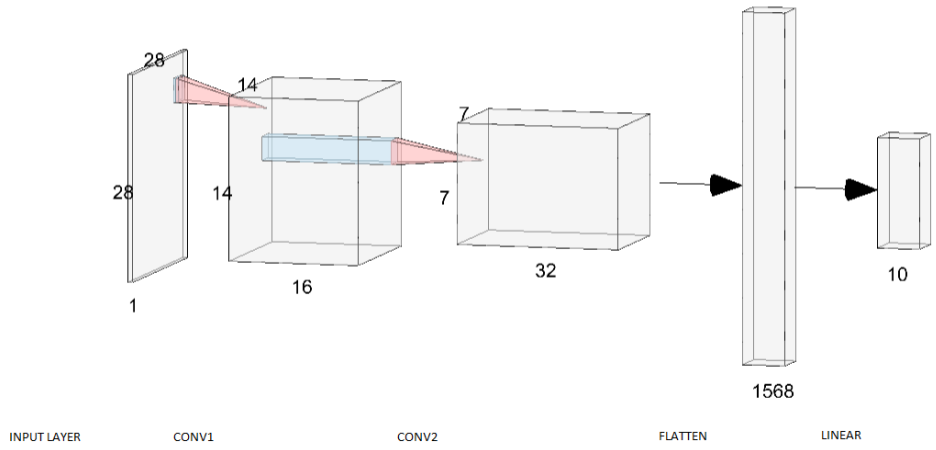


Figure 7.1: Conv Net for FMNIST dataset

Initially the CNN has been designed and trained in pytorch (see 3.3) achieving :

- Test accuracy : 89.4 %
- Average loss : 0.300765

Then the network has been quantized using Brevitas (check 5.2) and trained to recover accuracy degradation due to quantization.he best trade-off between model size and accuracy has been obtained using an 8-bit quantizations in all layers (int8 datatype) while to quantize activation has been used the PACT technique ([3]), achieving :

- Test accuracy : 89.7 %
- Average loss : 0.299013

Since the FINN compiler expects an ONNX model as input, then the model has been exported as ONNX model using Brevitas export classes.
In FINN has been built the accelerator based on the model and then deployed on the ZCU104 development board. On the board the accelerator has been test with the *validate.py* script achieving a final accuracy of 89.69 %.

CHAPTER 8

Appendix

8.1 Training the quantized network with Brevitas

In [1]:

```
import onnx
import torch
```

In [2]:

```
#load dataset

import torch, torchvision, copy
from torch import nn
from torchvision import datasets
from torchvision.transforms import transforms
import numpy as np
from tqdm import tqdm

transform_train = transforms.Compose([transforms.ToTensor()])
transform_test = transforms.Compose([transforms.ToTensor()])
training_data = datasets.FashionMNIST(root="/workspace/finn/notebooks/mnist_ex/data2/Fash
ionMNIST/raw", train=True, download=False, transform=transform_train)
test_data = datasets.FashionMNIST(root="/workspace/finn/notebooks/mnist_ex/data2/FashionM
NIST/raw", train=False, download=False, transform=transform_test)
print("Samples in each set: train = %d, test = %s" % (len(training_data), len(test_data)
))
print("Shape of one input sample: " +  str(training_data[0][0].shape))
```

```
Samples in each set: train = 60000, test = 10000
Shape of one input sample: torch.Size([1, 28, 28])
```

In [3]:

```
#divide in batches

from torch.utils.data import DataLoader, Dataset
batch_size = 32
# dataset loaders
train_dataloader = DataLoader(training_data, batch_size=batch_size, shuffle=True)
test_dataloader = DataLoader(test_data, batch_size=batch_size, shuffle=False)
count = 0
for x,y in train_dataloader:
    print("Input shape for 1 batch: " + str(x.shape))
    print("Label shape for 1 batch: " + str(y.shape))
    count += 1
    if count == 1:
        break
```

```
Input shape for 1 batch: torch.Size([32, 1, 28, 28])
Label shape for 1 batch: torch.Size([32])
```

In [4]:

```
#model declaration

from torch import nn
from torch.nn import Module
import brevitas.nn as qnn
from brevitas.quant import Int8Bias as BiasQuant
from brevitas.quant import Uint8ActPerTensorFloat as ActQuant
from brevitas.quant import Int8WeightPerTensorFloat as WeighQuant
import torch.nn.functional as F

# Setting seeds for reproducibility
torch.manual_seed(0)

weight_bitx = 8
act_bitx = 8
bias_bitx = 8
```

```
model = nn.Sequential(
    #qnn.QuantIdentity(bit_width=act_bitx, return_quant_tensor=True),
    qnn.QuantConv2d(in_channels=1, out_channels=3, kernel_size=(5, 5), stride=(1, 1), pa
dding=(2, 2),
                                    bias=False, weight_bit_width=weight_bitx, weight_qu
ant=WeighQuant,
                                    bias_quant=BiasQuant, return_quant_tensor=True),
    qnn.QuantReLU(bit_width=act_bitx, act_quant=ActQuant, return_quant_tensor=True),
    nn.MaxPool2d(kernel_size=2, stride=2),
    qnn.QuantConv2d(in_channels=3, out_channels=8, kernel_size=(5, 5), stride=(1, 1), pa
dding=(2, 2),
                                    bias=False, weight_bit_width=weight_bitx, bias_quan
t=BiasQuant,
                                    weight_quant=WeighQuant, return_quant_tensor=True),
    qnn.QuantReLU(bit_width=act_bitx, act_quant=ActQuant, return_quant_tensor=True),
    nn.MaxPool2d(kernel_size=2, stride=2),
    qnn.QuantConv2d(in_channels=8, out_channels=16, kernel_size=(5, 5), stride=(1, 1), p
adding=(2, 2),
                                    bias=False, weight_bit_width=weight_bitx, bias_quan
t=BiasQuant,
                                    weight_quant=WeighQuant, return_quant_tensor=True),
    qnn.QuantReLU(bit_width=act_bitx, act_quant=ActQuant, return_quant_tensor=True),
    nn.Flatten(),
    qnn.QuantLinear(in_features=16*7*7, out_features=10, bias=True, weight_bit_width=wei
ght_bitx,
                                   weight_quant=WeighQuant, bias_quant=BiasQuant, return
_quant_tensor=False)
)
```

In [5]:

```
#train and test functions
import torch
from sklearn.metrics import accuracy_score

def train(model, train_loader, optimizer, criterion):
    losses = []
    # ensure model is in training mode
    model.train()

    for i, data in enumerate(train_loader, 0):
        inputs, target = data
        optimizer.zero_grad()

        # forward pass
        output = model(inputs.float())
        loss = criterion(output, target)#.unsqueeze(1))

        # backward pass + run optimizer to update weights
        loss.backward()
        optimizer.step()

        # keep track of loss value
        losses.append(loss.data.numpy())

    return losses



def test(model, test_loader):
    # ensure model is in eval mode
    model.eval()
    y_true = []
    y_pred = []

    with torch.no_grad():
        for data in test_loader:
            inputs, target = data
            output_orig = model(inputs.float())
            # run the output through sigmoid
            #output = torch.sigmoid(output_orig)
```

```
            # compare against a threshold of 0.5 to generate 0/1
            #pred = (output.detach().numpy() > 0.5) * 1
            target = target.float()
            y_true.extend(target.tolist())
            y_pred.extend(output_orig.argmax(1).reshape(-1).tolist())

    return accuracy_score(y_true, y_pred)
```

In [6]:

```
#training settings
num_epochs = 10
lr = 0.001

def display_loss_plot(losses, title="Training loss", xlabel="Iterations", ylabel="Loss")
:
    x_axis = [i for i in range(len(losses))]
    plt.plot(x_axis,losses)
    plt.title(title)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.show()

# loss criterion and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=lr, betas=(0.9, 0.999))
```

In [7]:

```
#training
import numpy as np
from sklearn.metrics import accuracy_score
from tqdm import tqdm, trange

# Setting seeds for reproducibility
torch.manual_seed(0)
np.random.seed(0)

running_loss = []
running_test_acc = []
t = trange(num_epochs, desc="Training loss", leave=True)

for epoch in t:
        loss_epoch = train(model, train_dataloader, optimizer,criterion)
        test_acc = test(model, test_dataloader)
        t.set_description("Training loss = %f test accuracy = %f" % (np.mean(loss_epoch)
, test_acc))
        t.refresh() # to show immediately the update
        running_loss.append(loss_epoch)
        running_test_acc.append(test_acc)
```

```
Training loss = 0.239489 test accuracy = 0.898300: 100%|██████████| 10/10 [03:18<00:00, 1
9.82s/it]
```

In [8]:

```
#testing
test(model, test_dataloader)
```

Out[8]:

```
0.8983
```

In [9]:

```
# Save the Brevitas model to disk
torch.save(model.state_dict(), "state_dict_self-trained.pth")
```

In [10]:

```
import brevitas.onnx as bo
```

```
from brevitas.quant_tensor import QuantTensor

ready_model_filename = "model_fmnist_notebook.onnx"
input_shape = (1, 1, 28, 28)


bo.export_finn_onnx(
    model, export_path=ready_model_filename, input_shape=input_shape
)

print("Model saved to %s" % ready_model_filename)
```

```
Model saved to model_fmnist_notebook.onnx
```

In [ ]:

```
```

8.2 Import network in FINN and verify

In [1]:

```
import onnx
import torch
```

In [2]:

```
build_dir = "/workspace/finn/notebooks/mnist_ex/verification"
```

In [3]:

```
import brevitas.onnx as bo
from finn.util.visualization import showInNetron
from finn.util.test import get_test_model_trained
from finn.transformation.streamline import Streamline
from finn.transformation.lower_convs_to_matmul import LowerConvsToMatMul
from finn.transformation.bipolar_to_xnor import ConvertBipolarMatMulToXnorPopcount
import finn.transformation.streamline.absorb as absorb
from finn.transformation.streamline.reorder import MakeMaxPoolNHWC, MoveScalarLinearPastI
nvariants, MoveFlattenPastAffine
from finn.transformation.infer_data_layouts import InferDataLayouts
from finn.transformation.general import RemoveUnusedTensors

import finn.transformation.fpgadataflow.convert_to_hls_layers as to_hls
from finn.transformation.fpgadataflow.create_dataflow_partition import (
    CreateDataflowPartition,
)
from finn.transformation.move_reshape import RemoveCNVtoFCFlatten
from finn.custom_op.registry import getCustomOp
from finn.transformation.infer_data_layouts import InferDataLayouts
```

In [4]:

```
#import model .onnx into FINN
from finn.core.modelwrapper import ModelWrapper
ready_model_filename = "model_fmnist_notebook.onnx"
model_for_sim = ModelWrapper(ready_model_filename)
```

In [5]:

```
#member fuctions used to extract information about the structure and properties of the ON
NX model
from finn.core.datatype import DataType

finnonnx_in_tensor_name = model_for_sim.graph.input[0].name
finnonnx_out_tensor_name = model_for_sim.graph.output[0].name
print("Input tensor name: %s" % finnonnx_in_tensor_name)
print("Output tensor name: %s" % finnonnx_out_tensor_name)
finnonnx_model_in_shape = model_for_sim.get_tensor_shape(finnonnx_in_tensor_name)
finnonnx_model_out_shape = model_for_sim.get_tensor_shape(finnonnx_out_tensor_name)
print("Input tensor shape: %s" % str(finnonnx_model_in_shape))
print("Output tensor shape: %s" % str(finnonnx_model_out_shape))
finnonnx_model_in_dt = model_for_sim.get_tensor_datatype(finnonnx_in_tensor_name)
finnonnx_model_out_dt = model_for_sim.get_tensor_datatype(finnonnx_out_tensor_name)
print("Input tensor datatype: %s" % str(finnonnx_model_in_dt.name))
print("Output tensor datatype: %s" % str(finnonnx_model_out_dt.name))
print("List of node operator types in the graph: ")
print([x.op_type for x in model_for_sim.graph.node])
```

```
Input tensor name: inp.1
Output tensor name: 45
Input tensor shape: [1, 1, 28, 28]
Output tensor shape: [1, 10]
Input tensor datatype: FLOAT32
Output tensor datatype: FLOAT32
List of node operator types in the graph:
['Conv', 'Mul', 'MultiThreshold', 'Mul', 'MaxPool', 'Conv', 'Mul', 'MultiThreshold', 'Mul
', 'MaxPool', 'Conv', 'Mul', 'MultiThreshold', 'Mul', 'Flatten', 'MatMul', 'Mul', 'Div',
'Add', 'Mul']
```

```
'Add', 'Mul']
```

Note that the output tensor is (as of yet) marked as a float32 value, even though we know the output is binary. This will be automatically inferred by the compiler in the next step when we run the `InferDataTypes` transformation.

2. Network preparation: Tidy-up transformations

prepare our FINN-ONNX model. In particular, all the intermediate tensors need to have statically defined shapes.

Graph transformations in FINN. **transform the model into a synthesizable hardware description.**

In [6]:

```
from finn.transformation.general import GiveReadableTensorNames, GiveUniqueNodeNames, Rem
oveStaticGraphInputs
from finn.transformation.infer_shapes import InferShapes
from finn.transformation.infer_datatypes import InferDataTypes
from finn.transformation.fold_constants import FoldConstants

verif_model_filename = build_dir + "/model_fmnist_notebook_verified.onnx"

model_for_sim.set_tensor_datatype(model_for_sim.graph.input[0].name, DataType["UINT8"])
model_for_sim.save(verif_model_filename)

model_for_sim = model_for_sim.transform(InferShapes())
model_for_sim = model_for_sim.transform(FoldConstants())
model_for_sim = model_for_sim.transform(GiveUniqueNodeNames())
model_for_sim = model_for_sim.transform(GiveReadableTensorNames())
model_for_sim = model_for_sim.transform(RemoveStaticGraphInputs())
model_for_sim.save(build_dir + "/model_fmnist_notebook_tidy.onnx")
```

Let's view our ready-to-go model after the transformations. Note that all intermediate tensors now have their shapes specified (indicated by numbers next to the arrows going between layers). Additionally, the datatype inference step has propagated quantization annotations to the outputs of `MultiThreshold` layers (expand by clicking the + next to the name of the tensor to see the quantization annotation) and the final output tensor.

In [0]:

```
showInNetron(build_dir + "/model_fmnist_notebook_tidy.onnx")
```

adding pre and post processing directly in the ONNX graph. In this case, the preprocessing step divides the input uint8 data by 255 so the inputs are bounded between [0, 1]. The postprocessing step takes the output of the network and returns the index (0-9) of the image category with the highest probability (top-1).

In [7]:

```
#PREPROCESSING
from finn.util.pytorch import ToTensor
from finn.transformation.merge_onnx_models import MergeONNXModels

model_for_sim = ModelWrapper(build_dir+"/model_fmnist_notebook_tidy.onnx")
global_inp_name = model_for_sim.graph.input[0].name
ishape = model_for_sim.get_tensor_shape(global_inp_name)
# preprocessing: torchvision's ToTensor divides uint8 inputs by 255
totensor_pyt = ToTensor()
chkpt_preproc_name = build_dir + "/model_fmnist_pre.onnx"
bo.export_finn_onnx(totensor_pyt, ishape, chkpt_preproc_name)
# join preprocessing and core model
pre_model = ModelWrapper(chkpt_preproc_name)
model_for_sim = model_for_sim.transform(MergeONNXModels(pre_model))
# add input quantization annotation: UINT8
global_inp_name = model_for_sim.graph.input[0].name
model_for_sim.set_tensor_datatype(global_inp_name, DataType["UINT8"])
```

```
/workspace/finn-base/src/finn/transformation/infer_data_layouts.py:119: UserWarning: Assu
ming 4D input is NCHW
```

```
  warnings.warn("Assuming 4D input is NCHW")
```

In [8]:

```
finnonnx_in_tensor_name = model_for_sim.graph.input[0].name
finnonnx_out_tensor_name = model_for_sim.graph.output[0].name
print("Input tensor name: %s" % finnonnx_in_tensor_name)
print("Output tensor name: %s" % finnonnx_out_tensor_name)
finnonnx_model_in_shape = model_for_sim.get_tensor_shape(finnonnx_in_tensor_name)
finnonnx_model_out_shape = model_for_sim.get_tensor_shape(finnonnx_out_tensor_name)
print("Input tensor shape: %s" % str(finnonnx_model_in_shape))
print("Output tensor shape: %s" % str(finnonnx_model_out_shape))
finnonnx_model_in_dt = model_for_sim.get_tensor_datatype(finnonnx_in_tensor_name)
finnonnx_model_out_dt = model_for_sim.get_tensor_datatype(finnonnx_out_tensor_name)
print("Input tensor datatype: %s" % str(finnonnx_model_in_dt.name))
print("Output tensor datatype: %s" % str(finnonnx_model_out_dt.name))
print("List of node operator types in the graph: ")
print([x.op_type for x in model_for_sim.graph.node])
```

```
Input tensor name: global_in
Output tensor name: global_out
Input tensor shape: [1, 1, 28, 28]
Output tensor shape: [1, 10]
Input tensor datatype: UINT8
Output tensor datatype: FLOAT32
List of node operator types in the graph:
['Div', 'Conv', 'Mul', 'MultiThreshold', 'Mul', 'MaxPool', 'Conv', 'Mul', 'MultiThreshold
', 'Mul', 'MaxPool', 'Conv', 'Mul', 'MultiThreshold', 'Mul', 'Flatten', 'MatMul', 'Mul',
'Div', 'Add', 'Mul']
```

In [9]:

```
#POSTPROCESSING
from finn.transformation.insert_topk import InsertTopK
from finn.transformation.infer_shapes import InferShapes

# postprocessing: insert Top-1 node at the end
model_for_sim = model_for_sim.transform(InsertTopK(k=1))
chkpt_name = build_dir+"/model_fmnist_pre_post.onnx"
# tidy-up again
model_for_sim = model_for_sim.transform(InferShapes())
model_for_sim = model_for_sim.transform(FoldConstants())
model_for_sim = model_for_sim.transform(GiveUniqueNodeNames())
model_for_sim = model_for_sim.transform(GiveReadableTensorNames())
model_for_sim = model_for_sim.transform(InferDataTypes())
model_for_sim = model_for_sim.transform(RemoveStaticGraphInputs())
model_for_sim.save(chkpt_name)
```

In [10]:

```
finnonnx_in_tensor_name = model_for_sim.graph.input[0].name
finnonnx_out_tensor_name = model_for_sim.graph.output[0].name
print("Input tensor name: %s" % finnonnx_in_tensor_name)
print("Output tensor name: %s" % finnonnx_out_tensor_name)
finnonnx_model_in_shape = model_for_sim.get_tensor_shape(finnonnx_in_tensor_name)
finnonnx_model_out_shape = model_for_sim.get_tensor_shape(finnonnx_out_tensor_name)
print("Input tensor shape: %s" % str(finnonnx_model_in_shape))
print("Output tensor shape: %s" % str(finnonnx_model_out_shape))
finnonnx_model_in_dt = model_for_sim.get_tensor_datatype(finnonnx_in_tensor_name)
finnonnx_model_out_dt = model_for_sim.get_tensor_datatype(finnonnx_out_tensor_name)
print("Input tensor datatype: %s" % str(finnonnx_model_in_dt.name))
print("Output tensor datatype: %s" % str(finnonnx_model_out_dt.name))
print("List of node operator types in the graph: ")
print([x.op_type for x in model_for_sim.graph.node])
```

```
Input tensor name: global_in
Output tensor name: global_out
Input tensor shape: [1, 1, 28, 28]
Output tensor shape: [1, 1]
Input tensor datatype: UINT8
Output tensor datatype: UINT32
```

```
List of node operator types in the graph:
['Div', 'Conv', 'Mul', 'MultiThreshold', 'Mul', 'MaxPool', 'Conv', 'Mul', 'MultiThreshold
', 'Mul', 'MaxPool', 'Conv', 'Mul', 'MultiThreshold', 'Mul', 'Flatten', 'MatMul', 'Mul',
'Div', 'Add', 'Mul', 'TopK']
```

In [15]:

```
showInNetron(build_dir+"/model_fmnist_pre_post.onnx")
```

```
Serving '/workspace/finn/notebooks/mnist_ex/verification/model_fmnist_pre_post.onnx' at h
ttp://0.0.0.0:8081
```

Out[15]:

In [11]:

```
model_for_sim = ModelWrapper(build_dir + "/model_fmnist_pre_post.onnx")
model_for_sim = model_for_sim.transform(MoveScalarLinearPastInvariants())
model_for_sim = model_for_sim.transform(Streamline())
model_for_sim = model_for_sim.transform(LowerConvsToMatMul())
model_for_sim = model_for_sim.transform(MakeMaxPoolNHWC())

model_for_sim = model_for_sim.transform(absorb.AbsorbTransposeIntoMultiThreshold())

model_for_sim = model_for_sim.transform(MakeMaxPoolNHWC())
model_for_sim = model_for_sim.transform(absorb.AbsorbTransposeIntoMultiThreshold())

model_for_sim = model_for_sim.transform(Streamline())
# absorb final add-mul nodes into TopK
model_for_sim = model_for_sim.transform(absorb.AbsorbScalarMulAddIntoTopK())
model_for_sim = model_for_sim.transform(InferDataLayouts())
model_for_sim = model_for_sim.transform(RemoveUnusedTensors())

model_for_sim.save(build_dir + "/model_fmnist_stream.onnx")
```

In [20]:

```
showInNetron(build_dir+"/model_fmnist_stream.onnx")
```

```
Stopping http://0.0.0.0:8081
Serving '/workspace/finn/notebooks/mnist_ex/verification/model_fmnist_stream.onnx' at htt
p://0.0.0.0:8081
```

Out[20]:

In [12]:

```
# HLS
model_for_sim = ModelWrapper(build_dir + "/model_fmnist_stream.onnx")

# choose the memory mode for the MVTU units, decoupled or const
mem_mode = "decoupled"


model_for_sim = model_for_sim.transform(to_hls.InferQuantizedStreamingFCLayer(mem_mode))
# TopK to LabelSelect
model_for_sim = model_for_sim.transform(to_hls.InferLabelSelectLayer())
# input quantization (if any) to standalone thresholding


model_for_sim = model_for_sim.transform(to_hls.InferStreamingMaxPool())
model_for_sim = model_for_sim.transform(to_hls.InferPool_Batch())

model_for_sim = model_for_sim.transform(to_hls.InferThresholdingLayer())
model_for_sim = model_for_sim.transform(to_hls.InferConvInpGen())
model_for_sim = model_for_sim.transform(to_hls.InferStreamingMaxPool())
# get rid of Reshape(-1, 1) operation between hlslib nodes
model_for_sim = model_for_sim.transform(RemoveCNVtoFCFlatten())
# get rid of Tranpose -> Tranpose identity seq
model_for_sim = model_for_sim.transform(absorb.AbsorbConsecutiveTransposes())
# infer tensor data layouts
model_for_sim = model_for_sim.transform(InferDataLayouts())


model_for_sim = model_for_sim.transform(InferDataTypes())
model_for_sim = model_for_sim.transform(RemoveStaticGraphInputs())
model_for_sim = model_for_sim.transform(RemoveUnusedTensors())
model_for_sim.save(build_dir + "/model_fmnist_final.onnx")
parent_model = model_for_sim.transform(CreateDataflowPartition())
parent_model.save(build_dir + "/dataflow_parent.onnx")
sdp_node = parent_model.get_nodes_by_op_type("StreamingDataflowPartition")[0]
sdp_node = getCustomOp(sdp_node)
dataflow_model_filename = sdp_node.get_nodeattr("model")
# save the dataflow partition with a different name for easier access
dataflow_model = ModelWrapper(dataflow_model_filename)
dataflow_model.save(build_dir + "/dataflow_model.onnx")
```

In [23]:

```
from finn.util.visualization import showInNetron

showInNetron(build_dir + "/model_fmnist_final.onnx")
```

```
Stopping http://0.0.0.0:8081
Serving '/workspace/finn/notebooks/mnist_ex/verification/model_fmnist_final.onnx' at http
```

```
://0.0.0.0:8081
```

Out[23]:

3. Load the Dataset and the Brevitas Model

We'll use some example data from the Fashion mnist dataset (from the previous notebook) to use as inputs for the verification.

In [3]:

```
#import model .onnx into FINN
from finn.core.modelwrapper import ModelWrapper
build_dir="./verification"
model_for_sim = ModelWrapper(build_dir + "/model_fmnist_notebook_tidy.onnx")
from dataset_loading import mnist
```

In [4]:

```
bsize=32
dataset_root="/workspace/finn/notebooks/mnist_ex/data2/FashionMNIST/raw"

trainx, trainy, testx, testy, valx, valy = mnist.load_mnist_data(dataset_root, download=
False, one_hot=False)

test_imgs = testx
test_labels = testy
total = test_imgs.shape[0]
n_batches = int(total / bsize)
limit = n_batches*bsize
test_imgs = test_imgs[:limit,:,:].reshape(n_batches, bsize, -1)
test_labels = test_labels[:limit,].reshape(n_batches, bsize)
print(test_imgs.shape)
```

```
(312, 32, 784)
```

In [5]:

```
finnonnx_in_tensor_name = model_for_sim.graph.input[0].name
finnonnx_out_tensor_name = model_for_sim.graph.output[0].name
print("Input tensor name: %s" % finnonnx_in_tensor_name)
print("Output tensor name: %s" % finnonnx_out_tensor_name)
finnonnx_model_in_shape = model_for_sim.get_tensor_shape(finnonnx_in_tensor_name)
finnonnx_model_out_shape = model_for_sim.get_tensor_shape(finnonnx_out_tensor_name)
```

```
print("Input tensor shape: %s" % str(finnonnx_model_in_shape))
print("Output tensor shape: %s" % str(finnonnx_model_out_shape))
finnonnx_model_in_dt = model_for_sim.get_tensor_datatype(finnonnx_in_tensor_name)
finnonnx_model_out_dt = model_for_sim.get_tensor_datatype(finnonnx_out_tensor_name)
print("Input tensor datatype: %s" % str(finnonnx_model_in_dt.name))
print("Output tensor datatype: %s" % str(finnonnx_model_out_dt.name))
print("List of node operator types in the graph: ")
print([x.op_type for x in model_for_sim.graph.node])
```

```
Input tensor name: global_in
Output tensor name: global_out
Input tensor shape: [1, 1, 28, 28]
Output tensor shape: [1, 10]
Input tensor datatype: UINT8
Output tensor datatype: FLOAT32
List of node operator types in the graph:
['Conv', 'Mul', 'MultiThreshold', 'Mul', 'MaxPool', 'Conv', 'Mul', 'MultiThreshold', 'Mul
', 'MaxPool', 'Conv', 'Mul', 'MultiThreshold', 'Mul', 'Flatten', 'MatMul', 'Mul', 'Div',
'Add', 'Mul']
```

Let's also bring up the network we trained in Brevitas from the previous notebook. We'll compare its outputs to what is generated by FINN.

In [6]:

```
#MODEL

from torch import nn
from torch.nn import Module
import brevitas.nn as qnn
from brevitas.quant import Int8Bias as BiasQuant
from brevitas.quant import Uint8ActPerTensorFloat as ActQuant
from brevitas.quant import Int8WeightPerTensorFloat as WeighQuant
import torch.nn.functional as F

# Setting seeds for reproducibility
torch.manual_seed(0)

weight_bitx = 8
act_bitx = 8
bias_bitx = 8

model = nn.Sequential(
    #qnn.QuantIdentity(bit_width=act_bitx, return_quant_tensor=True),
    qnn.QuantConv2d(in_channels=1, out_channels=3, kernel_size=(5, 5), stride=(1, 1), pa
dding=(2, 2),
                                    bias=False, weight_bit_width=weight_bitx, weight_qu
ant=WeighQuant,
                                    bias_quant=BiasQuant, return_quant_tensor=True),
    qnn.QuantReLU(bit_width=act_bitx, act_quant=ActQuant, return_quant_tensor=True),
    nn.MaxPool2d(kernel_size=2, stride=2),
    qnn.QuantConv2d(in_channels=3, out_channels=8, kernel_size=(5, 5), stride=(1, 1), pa
dding=(2, 2),
                                    bias=False, weight_bit_width=weight_bitx, bias_quan
t=BiasQuant,
                                    weight_quant=WeighQuant, return_quant_tensor=True),
    qnn.QuantReLU(bit_width=act_bitx, act_quant=ActQuant, return_quant_tensor=True),
    nn.MaxPool2d(kernel_size=2, stride=2),
    qnn.QuantConv2d(in_channels=8, out_channels=16, kernel_size=(5, 5), stride=(1, 1), p
adding=(2, 2),
                                    bias=False, weight_bit_width=weight_bitx, bias_quan
t=BiasQuant,
                                    weight_quant=WeighQuant, return_quant_tensor=True),
    qnn.QuantReLU(bit_width=act_bitx, act_quant=ActQuant, return_quant_tensor=True),
    nn.Flatten(),
    qnn.QuantLinear(in_features=16*7*7, out_features=10, bias=True, weight_bit_width=wei
ght_bitx,
                                   weight_quant=WeighQuant, bias_quant=BiasQuant, return
_quant_tensor=False)
)
trained_state_dict = torch.load("state_dict_self-trained.pth")
```

```
#trained_state_dict = torch.load("state_dict.pth")["models_state_dict"][0]
model.load_state_dict(trained_state_dict, strict=False)
```

Out[6]:

```
<All keys matched successfully>
```

In [7]:

```
def inference_with_brevitas(current_inp):
    input_tensors = current_inp.astype(np.float32)
    input_tensors = torch.from_numpy(input_tensors)
    brevitas_output = model.forward(input_tensors)
    y_pred=brevitas_output.argmax(1)
    return y_pred
```

4. Compare FINN & Brevitas execution

Let's make helper functions to execute the same input with Brevitas and FINN. For FINN, we'll use the `finn.core.onnx_exec` function to execute the exported FINN-ONNX on the inputs. Note that this ONNX execution is for verification only; not for accelerated execution.

Recall that the data values from the dataset are uint8 vectors, so we' don't have to preprocess it before we can use it for verifying the ONNX model.

In [8]:

```
import finn.core.onnx_exec as oxe

def inference_with_finn_onnx(current_inp):
    finnonnx_in_tensor_name = model_for_sim.graph.input[0].name
    finnonnx_model_in_shape = model_for_sim.get_tensor_shape(finnonnx_in_tensor_name)
    finnonnx_out_tensor_name = model_for_sim.graph.output[0].name
    #print(finnonnx_model_in_shape)
    current_inp=current_inp.astype(np.float32)
    # reshape to expected input (add 1 for batch dimension)
    current_inp = current_inp.reshape(finnonnx_model_in_shape)
    # create the input dictionary
    input_dict = {finnonnx_in_tensor_name : current_inp}
    # run with FINN's execute_onnx
    output_dict = oxe.execute_onnx(model_for_sim, input_dict)
    #get the output tensor
    finn_output = output_dict[finnonnx_out_tensor_name]
    finn_output= torch.from_numpy(finn_output.argmax(1))
    return finn_output
```

Now we can call our inference helper functions for each input and compare the outputs.

In [9]:

```
import numpy as np
from tqdm import trange

n_verification_inputs=32
verify_range = trange(n_verification_inputs, desc="FINN execution", position=0, leave=Tr
ue)
model.eval()

ok = 0
nok = 0

for i in verify_range:
    current_inp = test_imgs[0][i].reshape(1, 1, 28, 28)
    brevitas_output = inference_with_brevitas(current_inp)
    finn_output = inference_with_finn_onnx(current_inp)
    #print(brevitas_output, finn_output)
    # compare the outputs
    ok += 1 if finn_output == brevitas_output else 0
```

```
    nok += 1 if finn_output != brevitas_output else 0
    verify_range.set_description("ok %d nok %d" % (ok, nok))
    verify_range.refresh()
    #print(finn_output)
```

```
ok 32 nok 0: 100%|██████████| 32/32 [00:03<00:00, 10.50it/s]
```

In [10]:

```
if ok == n_verification_inputs:
    print("Verification succeeded. Brevitas and FINN-ONNX execution outputs are identical
")
else:
    print("Verification failed. Brevitas and FINN-ONNX execution outputs are NOT identica
l")
```

```
Verification succeeded. Brevitas and FINN-ONNX execution outputs are identical
```

This concludes our second notebook. In the next one, we'll take the ONNX model we just verified all the way down to FPGA hardware with the FINN compiler.

8.3 Build accelerator

In [1]:

```
import brevitas.onnx as bo
from brevitas.quant_tensor import QuantTensor

ready_model_filename = "model_mnist_fpga.onnx"
```

In [4]:

```
from finn.util.visualization import showInNetron

showInNetron(ready_model_filename)
```

```
Serving 'model_mnist_fpga.onnx' at http://0.0.0.0:8081
```

Out[4]:

In [2]:

```
import finn.builder.build_dataflow as build
import finn.builder.build_dataflow_config as build_cfg
import os
import shutil

model_file = "verification/model_fmnist_final.onnx"
#model_file = "model_v4noid_verified.onnx"

estimates_output_dir = "output_estimates_only"



#Delete previous run results if exist
if os.path.exists(estimates_output_dir):
    shutil.rmtree(estimates_output_dir)
    print(os.path.abspath(estimates_output_dir))
    print("Previous run results deleted!")


cfg_estimates = build.DataflowBuildConfig(
    output_dir          = estimates_output_dir,
    mvau_wwidth_max     = 80,
    target_fps          = 1000000,
    synth_clk_period_ns = 10.0,
    fpga_part               = "xczu7ev-ffvc1156-2-e",
```

```
    steps               = build_cfg.estimate_only_dataflow_steps,
    generate_outputs=[
        build_cfg.DataflowOutputType.ESTIMATE_REPORTS,
    ]
)
```

```
/workspace/finn/notebooks/mnist_ex/output_estimates_only
Previous run results deleted!
```

In [3]:

```
%%time
build.build_dataflow_cfg(model_file, cfg_estimates)
```

```
Building dataflow accelerator from verification/model_fmnist_final.onnx
Intermediate outputs will be generated in /home/mmirigaldi/finn_temp_mmirigaldi
Final outputs will be generated in output_estimates_only
Build log is at output_estimates_only/build_dataflow.log
Running step: step_qonnx_to_finn [1/8]
Running step: step_tidy_up [2/8]
Running step: step_streamline [3/8]
Running step: step_convert_to_hls [4/8]
Running step: step_create_dataflow_partition [5/8]
Running step: step_target_fps_parallelization [6/8]
Running step: step_apply_folding_config [7/8]
Running step: step_generate_estimate_reports [8/8]
Completed successfully
CPU times: user 959 ms, sys: 9.81 ms, total: 969 ms
Wall time: 966 ms
```

Out[3]:

```
0
```

In [4]:

```
! ls {estimates_output_dir}
```

```
auto_folding_config.json  intermediate_models  time_per_step.json
build_dataflow.log   report
```

In [5]:

```
! ls {estimates_output_dir}/report
```

```
estimate_layer_config_alternatives.json  estimate_network_performance.json
estimate_layer_cycles.json   op_and_param_counts.json
estimate_layer_resources.json
```

In [6]:

```
! cat {estimates_output_dir}/report/estimate_network_performance.json
```


```
{
  "critical_path_cycles": 96444,
  "max_cycles": 19760,
  "max_cycles_node_name": "ConvolutionInputGenerator_0",
  "estimated_throughput_fps": 5060.728744939272,
  "estimated_latency_ns": 964440.0
}
```


In [7]:

```
import json
def read_json_dict(filename):
    with open(filename, "r") as f:
        ret = json.load(f)
    return ret
```

In [8]:

```
read_json_dict(estimates_output_dir + "/report/estimate_layer_cycles.json")
```

Out[8]:


```
{'FMPadding_Batch_0': 1024,
 'ConvolutionInputGenerator_0': 19760,
 'StreamingFCLayer_Batch_0': 11760,
 'StreamingMaxPool_Batch_0': 1176,
 'FMPadding_Batch_1': 972,
 'ConvolutionInputGenerator_1': 14970,
 'StreamingFCLayer_Batch_1': 7840,
 'StreamingMaxPool_Batch_1': 294,
 'FMPadding_Batch_2': 968,
 'ConvolutionInputGenerator_2': 10240,
 'StreamingFCLayer_Batch_2': 19600,
 'StreamingFCLayer_Batch_3': 7840}
```


In [9]:

```
read_json_dict(estimates_output_dir + "/report/estimate_layer_resources.json")
```

Out[9]:


```
{'FMPadding_Batch_0': {'BRAM_18K': 0,
  'BRAM_efficiency': 1,
  'LUT': 0,
  'URAM': 0,
  'URAM_efficiency': 1,
  'DSP': 0},
 'ConvolutionInputGenerator_0': {'BRAM_18K': 0,
  'BRAM_efficiency': 1,
  'LUT': 348,
  'URAM': 0,
  'URAM_efficiency': 1,
  'DSP': 0},
 'StreamingFCLayer_Batch_0': {'BRAM_18K': 2,
  'BRAM_efficiency': 0.016276041666666668,
  'LUT': 12702,
  'URAM': 0,
  'URAM_efficiency': 1,
  'DSP': 0},
 'StreamingMaxPool_Batch_0': {'BRAM_18K': 0,
  'BRAM_efficiency': 1,
  'LUT': 0,
  'URAM': 0,
  'URAM_efficiency': 1,
  'DSP': 0},
 'FMPadding_Batch_1': {'BRAM_18K': 0,
  'BRAM_efficiency': 1,
  'LUT': 0,
  'URAM': 0,
  'URAM_efficiency': 1,
  'DSP': 0},
 'ConvolutionInputGenerator_1': {'BRAM_18K': 0,
  'BRAM_efficiency': 1,
  'LUT': 348,
  'URAM': 0,
  'URAM_efficiency': 1,
  'DSP': 0},
 'StreamingFCLayer_Batch_1': {'BRAM_18K': 4,
  'BRAM_efficiency': 0.06510416666666667,
  'LUT': 15058,
  'URAM': 0,
  'URAM_efficiency': 1,
  'DSP': 0},
 'StreamingMaxPool_Batch_1': {'BRAM_18K': 0,
  'BRAM_efficiency': 1,
  'LUT': 0,
  'URAM': 0,
  'URAM_efficiency': 1,
  'DSP': 0},
 'FMPadding_Batch_2': {'BRAM_18K': 0,
  'BRAM_efficiency': 1,
  'LUT': 0,
```

```
  'URAM': 0,
  'URAM_efficiency': 1,
  'DSP': 0},
 'ConvolutionInputGenerator_2': {'BRAM_18K': 0,
  'BRAM_efficiency': 1,
  'LUT': 396,
  'URAM': 0,
  'URAM_efficiency': 1,
  'DSP': 0},
 'StreamingFCLayer_Batch_2': {'BRAM_18K': 2,
  'BRAM_efficiency': 0.6944444444444444,
  'LUT': 14758,
  'URAM': 0,
  'URAM_efficiency': 1,
  'DSP': 0},
 'StreamingFCLayer_Batch_3': {'BRAM_18K': 4,
  'BRAM_efficiency': 0.8506944444444444,
  'LUT': 434,
  'URAM': 0,
  'URAM_efficiency': 1,
  'DSP': 0},
 'total': {'BRAM_18K': 12.0, 'LUT': 44044.0, 'URAM': 0.0, 'DSP': 0.0}}
```

In [10]:

```
import finn.builder.build_dataflow as build
import finn.builder.build_dataflow_config as build_cfg
import os
import shutil

#model_file = "model_v4noid_verified.onnx"
model_file = "verification/model_fmnist_final.onnx"

rtlsim_output_dir = "output_ipstitch_ooc_rtlsim"

#Delete previous run results if exist
if os.path.exists(rtlsim_output_dir):
    shutil.rmtree(rtlsim_output_dir)
    print("Previous run results deleted!")

cfg_stitched_ip = build.DataflowBuildConfig(
    output_dir          = rtlsim_output_dir,
    mvau_wwidth_max     = 80,
    target_fps          = 100000,
    synth_clk_period_ns = 10.0,
    fpga_part               = "xczu7ev-ffvc1156-2-e",
    generate_outputs=[
        build_cfg.DataflowOutputType.STITCHED_IP,
        build_cfg.DataflowOutputType.RTLSIM_PERFORMANCE,
        #build_cfg.DataflowOutputType.OOC_SYNTH,
    ]
)
```

```
Previous run results deleted!
```

In [11]:

```
%%time
build.build_dataflow_cfg(model_file, cfg_stitched_ip)
```

```
Building dataflow accelerator from verification/model_fmnist_final.onnx
Intermediate outputs will be generated in /home/mmirigaldi/finn_temp_mmirigaldi
Final outputs will be generated in output_ipstitch_ooc_rtlsim
Build log is at output_ipstitch_ooc_rtlsim/build_dataflow.log
Running step: step_qonnx_to_finn [1/17]
Running step: step_tidy_up [2/17]
Running step: step_streamline [3/17]
Running step: step_convert_to_hls [4/17]
Running step: step_create_dataflow_partition [5/17]
Running step: step_target_fps_parallelization [6/17]
Running step: step_apply_folding_config [7/17]
Running step: step_generate_estimate_reports [8/17]
```

```
Running step: step_hls_codegen [9/17]
Running step: step_hls_ipgen [10/17]
Running step: step_set_fifo_depths [11/17]
Running step: step_create_stitched_ip [12/17]
Running step: step_measure_rtlsim_performance [13/17]
Running step: step_out_of_context_synthesis [14/17]
Running step: step_synthesize_bitfile [15/17]
Running step: step_make_pynq_driver [16/17]
Running step: step_deployment_package [17/17]
Completed successfully
CPU times: user 39.6 s, sys: 825 ms, total: 40.5 s
Wall time: 6min 57s
```

Out[11]:

```
0
```

In [12]:

```
! ls {rtlsim_output_dir}/stitched_ip
```

```
all_verilog_srcs.txt         finn_vivado_stitch_proj.xpr
finn_vivado_stitch_proj.cache        ip
finn_vivado_stitch_proj.hbs        make_project.sh
finn_vivado_stitch_proj.hw        make_project.tcl
finn_vivado_stitch_proj.ip_user_files  vivado.jou
finn_vivado_stitch_proj.srcs        vivado.log
```

In [13]:

```
! ls {rtlsim_output_dir}/report
```

```
estimate_layer_resources_hls.json  rtlsim_performance.json
```

In [14]:

```
#! cat {rtlsim_output_dir}/report/ooc_synth_and_timing.json
```

In [15]:

```
! cat {rtlsim_output_dir}/report/rtlsim_performance.json
```


```
{
  "cycles": 46680,
  "runtime[ms]": 0.4668,
  "throughput[images/s]": 2142.2450728363324,
  "DRAM_in_bandwidth[Mb/s]": 1.6795201371036845,
  "DRAM_out_bandwidth[Mb/s]": 0.0856898029134533,
  "fclk[mhz]": 100.0,
  "N": 1,
  "latency_cycles": 46680
}
```


In [16]:

```
! cat {rtlsim_output_dir}/final_hw_config.json
```


```
{
  "Defaults": {},
  "StreamingFIFO_0": {
    "ram_style": "auto",
    "depth": 32,
    "impl_style": "rtl"
  },
  "FMPadding_Batch_0": {
    "SIMD": 1
  },
  "StreamingFIFO_1": {
    "ram_style": "auto",
    "depth": 256,
    "impl_style": "rtl"
  },
  "ConvolutionInputGenerator_0": {
```

```
"ConvolutionInputGenerator_0": {
  "SIMD": 1,
  "ram_style": "distributed"
},
"StreamingDataWidthConverter_Batch_0": {
  "impl_style": "hls"
},
"StreamingFCLayer_Batch_0": {
  "PE": 1,
  "SIMD": 5,
  "ram_style": "auto",
  "resType": "lut",
  "mem_mode": "decoupled",
  "runtime_writeable_weights": 0
},
"StreamingDataWidthConverter_Batch_1": {
  "impl_style": "hls"
},
"StreamingFIFO_6": {
  "ram_style": "auto",
  "depth": 32,
  "impl_style": "rtl"
},
"StreamingDataWidthConverter_Batch_2": {
  "impl_style": "hls"
},
"StreamingFIFO_7": {
  "ram_style": "auto",
  "depth": 32,
  "impl_style": "rtl"
},
"FMPadding_Batch_1": {
  "SIMD": 1
},
"StreamingFIFO_8": {
  "ram_style": "auto",
  "depth": 256,
  "impl_style": "rtl"
},
"ConvolutionInputGenerator_1": {
  "SIMD": 1,
  "ram_style": "distributed"
},
"StreamingDataWidthConverter_Batch_3": {
  "impl_style": "hls"
},
"StreamingFCLayer_Batch_1": {
  "PE": 1,
  "SIMD": 15,
  "ram_style": "auto",
  "resType": "lut",
  "mem_mode": "decoupled",
  "runtime_writeable_weights": 0
},
"StreamingDataWidthConverter_Batch_4": {
  "impl_style": "hls"
},
"StreamingFIFO_13": {
  "ram_style": "auto",
  "depth": 32,
  "impl_style": "rtl"
},
"StreamingDataWidthConverter_Batch_5": {
  "impl_style": "hls"
},
"StreamingFIFO_14": {
  "ram_style": "auto",
  "depth": 64,
  "impl_style": "rtl"
},
"FMPadding_Batch_2": {
  "SIMD": 1
}
```

```
  },
  "StreamingFIFO_15": {
    "ram_style": "auto",
    "depth": 512,
    "impl_style": "vivado"
  },
  "ConvolutionInputGenerator_2": {
    "SIMD": 1,
    "ram_style": "distributed"
  },
  "StreamingDataWidthConverter_Batch_6": {
    "impl_style": "hls"
  },
  "StreamingFIFO_17": {
    "ram_style": "auto",
    "depth": 1024,
    "impl_style": "vivado"
  },
  "StreamingFCLayer_Batch_2": {
    "PE": 1,
    "SIMD": 8,
    "ram_style": "auto",
    "resType": "lut",
    "mem_mode": "decoupled",
    "runtime_writeable_weights": 0
  },
  "StreamingFIFO_18": {
    "ram_style": "auto",
    "depth": 256,
    "impl_style": "rtl"
  },
  "StreamingFCLayer_Batch_3": {
    "PE": 1,
    "SIMD": 1,
    "ram_style": "auto",
    "resType": "lut",
    "mem_mode": "decoupled",
    "runtime_writeable_weights": 0
  }
}
```

In [17]:

```
import finn.builder.build_dataflow as build
import finn.builder.build_dataflow_config as build_cfg
import os
import shutil

#model_file = "model_v4noid_verified.onnx"
model_file = "verification/model_fmnist_final.onnx"

final_output_dir = "output_final"

#Delete previous run results if exist
if os.path.exists(final_output_dir):
    shutil.rmtree(final_output_dir)
    print("Previous run results deleted!")

cfg = build.DataflowBuildConfig(
    output_dir          = final_output_dir,
    mvau_wwidth_max     = 80,
    target_fps          = 1000000,
    synth_clk_period_ns = 10.0,
    board               = "ZCU104",
    fpga_part               = "xczu7ev-ffvc1156-2-e",
    shell_flow_type     = build_cfg.ShellFlowType.VIVADO_ZYNQ,
    generate_outputs=[
        build_cfg.DataflowOutputType.BITFILE,
        build_cfg.DataflowOutputType.PYNQ_DRIVER,
        build_cfg.DataflowOutputType.DEPLOYMENT_PACKAGE,
    ]
)
```

```
Previous run results deleted!
```

In [18]:

```
%%time
build.build_dataflow_cfg(model_file, cfg)
```

```
Building dataflow accelerator from verification/model_fmnist_final.onnx
Intermediate outputs will be generated in /home/mmirigaldi/finn_temp_mmirigaldi
Final outputs will be generated in output_final
Build log is at output_final/build_dataflow.log
Running step: step_qonnx_to_finn [1/17]
Running step: step_tidy_up [2/17]
Running step: step_streamline [3/17]
Running step: step_convert_to_hls [4/17]
Running step: step_create_dataflow_partition [5/17]
Running step: step_target_fps_parallelization [6/17]
Running step: step_apply_folding_config [7/17]
Running step: step_generate_estimate_reports [8/17]
Running step: step_hls_codegen [9/17]
Running step: step_hls_ipgen [10/17]
Running step: step_set_fifo_depths [11/17]
Running step: step_create_stitched_ip [12/17]
Running step: step_measure_rtlsim_performance [13/17]
Running step: step_out_of_context_synthesis [14/17]
Running step: step_synthesize_bitfile [15/17]
Running step: step_make_pynq_driver [16/17]
Running step: step_deployment_package [17/17]
Completed successfully
CPU times: user 38.3 s, sys: 675 ms, total: 38.9 s
Wall time: 38min 56s
```

Out[18]:

```
0
```

In [19]:

```
! ls {final_output_dir}/bitfile
```

```
finn-accel.bit finn-accel.hwh
```

In [20]:

```
! ls {final_output_dir}/driver
```

```
driver.py  driver_base.py  finn  runtime_weights  validate.py
```

In [21]:

```
! ls {final_output_dir}/report
```

```
estimate_layer_resources_hls.json  post_synth_resources.xml
post_route_timing.rpt
```

In [22]:

```
! ls {final_output_dir}/deploy
```

```
bitfile  driver
```

In [23]:

```
! cp -r data2 {final_output_dir}/deploy/driver
```

In [0]:

```
! ls {final_output_dir}/deploy/driver
```

In [0]:

```
from shutil import make_archive
```

```
from shutil import make_archive
make_archive('deploy-fpga-bis-on-pynq', 'zip', final_output_dir+"/deploy")
```

8.4 Deploy to board

In [3]:

```
cd ./xilinx/jupyter_notebooks
```

```
/home/xilinx/jupyter_notebooks
```

In [1]:

```
! pwd
```

```
/home/xilinx/jupyter_notebooks
```

In [2]:

```
#! zip -r finn-mnist.zip finn-mnist
! unzip deploy-fpga-bis-on-pynq.zip -d deploy-fpga-bis-on-pynq
```

```
Archive:  deploy-fpga-bis-on-pynq.zip
   creating: deploy-fpga-bis-on-pynq/bitfile/
   creating: deploy-fpga-bis-on-pynq/driver/
   creating: deploy-fpga-bis-on-pynq/driver/data2/
   creating: deploy-fpga-bis-on-pynq/driver/finn/
   creating: deploy-fpga-bis-on-pynq/driver/runtime_weights/
  inflating: deploy-fpga-bis-on-pynq/driver/validate.py
  inflating: deploy-fpga-bis-on-pynq/driver/driver.py
  inflating: deploy-fpga-bis-on-pynq/driver/driver_base.py
   creating: deploy-fpga-bis-on-pynq/driver/data2/FashionMNIST/
   creating: deploy-fpga-bis-on-pynq/driver/data2/FashionMNIST/.ipynb_checkpoints/
   creating: deploy-fpga-bis-on-pynq/driver/data2/FashionMNIST/raw/
   creating: deploy-fpga-bis-on-pynq/driver/data2/FashionMNIST/raw/FashionMNIST/
  inflating: deploy-fpga-bis-on-pynq/driver/data2/FashionMNIST/raw/t10k-images-idx3-ubyte

  inflating: deploy-fpga-bis-on-pynq/driver/data2/FashionMNIST/raw/t10k-labels-idx1-ubyte

  inflating: deploy-fpga-bis-on-pynq/driver/data2/FashionMNIST/raw/train-images-idx3-ubyt
e
  inflating: deploy-fpga-bis-on-pynq/driver/data2/FashionMNIST/raw/train-labels-idx1-ubyt
e.gz
  inflating: deploy-fpga-bis-on-pynq/driver/data2/FashionMNIST/raw/train-images-idx3-ubyt
e.gz
  inflating: deploy-fpga-bis-on-pynq/driver/data2/FashionMNIST/raw/train-labels-idx1-ubyt
e
  inflating: deploy-fpga-bis-on-pynq/driver/data2/FashionMNIST/raw/t10k-images-idx3-ubyte
.gz
  inflating: deploy-fpga-bis-on-pynq/driver/data2/FashionMNIST/raw/t10k-labels-idx1-ubyte
.gz
   creating: deploy-fpga-bis-on-pynq/driver/data2/FashionMNIST/raw/FashionMNIST/processed
/
   creating: deploy-fpga-bis-on-pynq/driver/data2/FashionMNIST/raw/FashionMNIST/raw/
  inflating: deploy-fpga-bis-on-pynq/driver/data2/FashionMNIST/raw/FashionMNIST/processed
/training.pt
  inflating: deploy-fpga-bis-on-pynq/driver/data2/FashionMNIST/raw/FashionMNIST/processed
/test.pt
  inflating: deploy-fpga-bis-on-pynq/driver/data2/FashionMNIST/raw/FashionMNIST/raw/t10k-
images-idx3-ubyte
  inflating: deploy-fpga-bis-on-pynq/driver/data2/FashionMNIST/raw/FashionMNIST/raw/t10k-
labels-idx1-ubyte
  inflating: deploy-fpga-bis-on-pynq/driver/data2/FashionMNIST/raw/FashionMNIST/raw/train
-images-idx3-ubyte
  inflating: deploy-fpga-bis-on-pynq/driver/data2/FashionMNIST/raw/FashionMNIST/raw/train
-labels-idx1-ubyte.gz
  inflating: deploy-fpga-bis-on-pynq/driver/data2/FashionMNIST/raw/FashionMNIST/raw/train
-images-idx3-ubyte.gz
  inflating: deploy-fpga-bis-on-pynq/driver/data2/FashionMNIST/raw/FashionMNIST/raw/train
-labels-idx1-ubyte
  inflating: deploy-fpga-bis-on-pynq/driver/data2/FashionMNIST/raw/FashionMNIST/raw/t10k-
images-idx3-ubyte.gz
  inflating: deploy-fpga-bis-on-pynq/driver/data2/FashionMNIST/raw/FashionMNIST/raw/t10k-
labels-idx1-ubyte.gz
```

```
   creating: deploy-fpga-bis-on-pynq/driver/finn/core/
   creating: deploy-fpga-bis-on-pynq/driver/finn/util/
  inflating: deploy-fpga-bis-on-pynq/driver/finn/core/onnx_exec.py
  inflating: deploy-fpga-bis-on-pynq/driver/finn/core/rtlsim_exec.py
  inflating: deploy-fpga-bis-on-pynq/driver/finn/core/execute_custom_node.py
  inflating: deploy-fpga-bis-on-pynq/driver/finn/core/datatype.py
  inflating: deploy-fpga-bis-on-pynq/driver/finn/core/modelwrapper.py
  inflating: deploy-fpga-bis-on-pynq/driver/finn/core/throughput_test.py
  inflating: deploy-fpga-bis-on-pynq/driver/finn/core/data_layout.py
  inflating: deploy-fpga-bis-on-pynq/driver/finn/core/remote_exec.py
  inflating: deploy-fpga-bis-on-pynq/driver/finn/util/fpgadataflow.py
  inflating: deploy-fpga-bis-on-pynq/driver/finn/util/onnx.py
  inflating: deploy-fpga-bis-on-pynq/driver/finn/util/config.py
  inflating: deploy-fpga-bis-on-pynq/driver/finn/util/inference_cost.py
  inflating: deploy-fpga-bis-on-pynq/driver/finn/util/hls.py
  inflating: deploy-fpga-bis-on-pynq/driver/finn/util/basic.py
  inflating: deploy-fpga-bis-on-pynq/driver/finn/util/pyverilator.py
  inflating: deploy-fpga-bis-on-pynq/driver/finn/util/platforms.py
  inflating: deploy-fpga-bis-on-pynq/driver/finn/util/data_packing.py
  inflating: deploy-fpga-bis-on-pynq/driver/finn/util/vivado.py
  inflating: deploy-fpga-bis-on-pynq/bitfile/finn-accel.bit
  inflating: deploy-fpga-bis-on-pynq/bitfile/finn-accel.hwh
```

In [3]:

```
cd /home/xilinx/jupyter_notebooks/deploy-fpga-bis-on-pynq/driver
```

```
/home/xilinx/jupyter_notebooks/deploy-fpga-bis-on-pynq/driver
```

In [6]:

```
! python3 validate.py --batchsize 32
```

```
batch 1 / 312 : total OK 28 NOK 4
batch 2 / 312 : total OK 55 NOK 9
batch 3 / 312 : total OK 83 NOK 13
batch 4 / 312 : total OK 111 NOK 17
batch 5 / 312 : total OK 140 NOK 20
batch 6 / 312 : total OK 171 NOK 21
batch 7 / 312 : total OK 203 NOK 21
batch 8 / 312 : total OK 232 NOK 24
batch 9 / 312 : total OK 261 NOK 27
batch 10 / 312 : total OK 291 NOK 29
batch 11 / 312 : total OK 319 NOK 33
batch 12 / 312 : total OK 348 NOK 36
batch 13 / 312 : total OK 377 NOK 39
batch 14 / 312 : total OK 407 NOK 41
batch 15 / 312 : total OK 435 NOK 45
batch 16 / 312 : total OK 465 NOK 47
batch 17 / 312 : total OK 495 NOK 49
batch 18 / 312 : total OK 525 NOK 51
batch 19 / 312 : total OK 552 NOK 56
batch 20 / 312 : total OK 580 NOK 60
batch 21 / 312 : total OK 611 NOK 61
batch 22 / 312 : total OK 640 NOK 64
batch 23 / 312 : total OK 668 NOK 68
batch 24 / 312 : total OK 698 NOK 70
batch 25 / 312 : total OK 729 NOK 71
batch 26 / 312 : total OK 758 NOK 74
batch 27 / 312 : total OK 789 NOK 75
batch 28 / 312 : total OK 819 NOK 77
batch 29 / 312 : total OK 849 NOK 79
batch 30 / 312 : total OK 877 NOK 83
batch 31 / 312 : total OK 902 NOK 90
batch 32 / 312 : total OK 932 NOK 92
batch 33 / 312 : total OK 961 NOK 95
batch 34 / 312 : total OK 991 NOK 97
batch 35 / 312 : total OK 1021 NOK 99
batch 36 / 312 : total OK 1049 NOK 103
batch 37 / 312 : total OK 1079 NOK 105
batch 38 / 312 : total OK 1107 NOK 109
batch 39 / 312 : total OK 1135 NOK 113
```

```
batch 40 / 312 : total OK 1165 NOK 115
batch 41 / 312 : total OK 1194 NOK 118
batch 42 / 312 : total OK 1221 NOK 123
batch 43 / 312 : total OK 1252 NOK 124
batch 44 / 312 : total OK 1281 NOK 127
batch 45 / 312 : total OK 1311 NOK 129
batch 46 / 312 : total OK 1341 NOK 131
batch 47 / 312 : total OK 1368 NOK 136
batch 48 / 312 : total OK 1395 NOK 141
batch 49 / 312 : total OK 1427 NOK 141
batch 50 / 312 : total OK 1457 NOK 143
batch 51 / 312 : total OK 1485 NOK 147
batch 52 / 312 : total OK 1510 NOK 154
batch 53 / 312 : total OK 1539 NOK 157
batch 54 / 312 : total OK 1567 NOK 161
batch 55 / 312 : total OK 1595 NOK 165
batch 56 / 312 : total OK 1625 NOK 167
batch 57 / 312 : total OK 1655 NOK 169
batch 58 / 312 : total OK 1686 NOK 170
batch 59 / 312 : total OK 1715 NOK 173
batch 60 / 312 : total OK 1745 NOK 175
batch 61 / 312 : total OK 1774 NOK 178
batch 62 / 312 : total OK 1801 NOK 183
batch 63 / 312 : total OK 1831 NOK 185
batch 64 / 312 : total OK 1856 NOK 192
batch 65 / 312 : total OK 1886 NOK 194
batch 66 / 312 : total OK 1916 NOK 196
batch 67 / 312 : total OK 1947 NOK 197
batch 68 / 312 : total OK 1978 NOK 198
batch 69 / 312 : total OK 2006 NOK 202
batch 70 / 312 : total OK 2037 NOK 203
batch 71 / 312 : total OK 2066 NOK 206
batch 72 / 312 : total OK 2092 NOK 212
batch 73 / 312 : total OK 2120 NOK 216
batch 74 / 312 : total OK 2147 NOK 221
batch 75 / 312 : total OK 2175 NOK 225
batch 76 / 312 : total OK 2205 NOK 227
batch 77 / 312 : total OK 2236 NOK 228
batch 78 / 312 : total OK 2261 NOK 235
batch 79 / 312 : total OK 2287 NOK 241
batch 80 / 312 : total OK 2314 NOK 246
batch 81 / 312 : total OK 2343 NOK 249
batch 82 / 312 : total OK 2369 NOK 255
batch 83 / 312 : total OK 2395 NOK 261
batch 84 / 312 : total OK 2423 NOK 265
batch 85 / 312 : total OK 2451 NOK 269
batch 86 / 312 : total OK 2478 NOK 274
batch 87 / 312 : total OK 2506 NOK 278
batch 88 / 312 : total OK 2536 NOK 280
batch 89 / 312 : total OK 2561 NOK 287
batch 90 / 312 : total OK 2592 NOK 288
batch 91 / 312 : total OK 2614 NOK 298
batch 92 / 312 : total OK 2639 NOK 305
batch 93 / 312 : total OK 2667 NOK 309
batch 94 / 312 : total OK 2695 NOK 313
batch 95 / 312 : total OK 2724 NOK 316
batch 96 / 312 : total OK 2753 NOK 319
batch 97 / 312 : total OK 2782 NOK 322
batch 98 / 312 : total OK 2811 NOK 325
batch 99 / 312 : total OK 2843 NOK 325
batch 100 / 312 : total OK 2869 NOK 331
batch 101 / 312 : total OK 2896 NOK 336
batch 102 / 312 : total OK 2922 NOK 342
batch 103 / 312 : total OK 2949 NOK 347
batch 104 / 312 : total OK 2976 NOK 352
batch 105 / 312 : total OK 3003 NOK 357
batch 106 / 312 : total OK 3033 NOK 359
batch 107 / 312 : total OK 3063 NOK 361
batch 108 / 312 : total OK 3094 NOK 362
batch 109 / 312 : total OK 3121 NOK 367
batch 110 / 312 : total OK 3148 NOK 372
batch 111 / 312 : total OK 3176 NOK 376
```

```
batch 112 / 312 : total OK 3206 NOK 378
batch 113 / 312 : total OK 3236 NOK 380
batch 114 / 312 : total OK 3266 NOK 382
batch 115 / 312 : total OK 3294 NOK 386
batch 116 / 312 : total OK 3323 NOK 389
batch 117 / 312 : total OK 3352 NOK 392
batch 118 / 312 : total OK 3383 NOK 393
batch 119 / 312 : total OK 3411 NOK 397
batch 120 / 312 : total OK 3438 NOK 402
batch 121 / 312 : total OK 3467 NOK 405
batch 122 / 312 : total OK 3498 NOK 406
batch 123 / 312 : total OK 3529 NOK 407
batch 124 / 312 : total OK 3555 NOK 413
batch 125 / 312 : total OK 3584 NOK 416
batch 126 / 312 : total OK 3613 NOK 419
batch 127 / 312 : total OK 3639 NOK 425
batch 128 / 312 : total OK 3667 NOK 429
batch 129 / 312 : total OK 3695 NOK 433
batch 130 / 312 : total OK 3721 NOK 439
batch 131 / 312 : total OK 3752 NOK 440
batch 132 / 312 : total OK 3780 NOK 444
batch 133 / 312 : total OK 3808 NOK 448
batch 134 / 312 : total OK 3836 NOK 452
batch 135 / 312 : total OK 3866 NOK 454
batch 136 / 312 : total OK 3896 NOK 456
batch 137 / 312 : total OK 3926 NOK 458
batch 138 / 312 : total OK 3956 NOK 460
batch 139 / 312 : total OK 3984 NOK 464
batch 140 / 312 : total OK 4012 NOK 468
batch 141 / 312 : total OK 4042 NOK 470
batch 142 / 312 : total OK 4072 NOK 472
batch 143 / 312 : total OK 4102 NOK 474
batch 144 / 312 : total OK 4133 NOK 475
batch 145 / 312 : total OK 4162 NOK 478
batch 146 / 312 : total OK 4190 NOK 482
batch 147 / 312 : total OK 4217 NOK 487
batch 148 / 312 : total OK 4246 NOK 490
batch 149 / 312 : total OK 4273 NOK 495
batch 150 / 312 : total OK 4301 NOK 499
batch 151 / 312 : total OK 4329 NOK 503
batch 152 / 312 : total OK 4357 NOK 507
batch 153 / 312 : total OK 4386 NOK 510
batch 154 / 312 : total OK 4412 NOK 516
batch 155 / 312 : total OK 4441 NOK 519
batch 156 / 312 : total OK 4473 NOK 519
batch 157 / 312 : total OK 4501 NOK 523
batch 158 / 312 : total OK 4527 NOK 529
batch 159 / 312 : total OK 4556 NOK 532
batch 160 / 312 : total OK 4587 NOK 533
batch 161 / 312 : total OK 4616 NOK 536
batch 162 / 312 : total OK 4643 NOK 541
batch 163 / 312 : total OK 4671 NOK 545
batch 164 / 312 : total OK 4700 NOK 548
batch 165 / 312 : total OK 4727 NOK 553
batch 166 / 312 : total OK 4757 NOK 555
batch 167 / 312 : total OK 4784 NOK 560
batch 168 / 312 : total OK 4816 NOK 560
batch 169 / 312 : total OK 4844 NOK 564
batch 170 / 312 : total OK 4873 NOK 567
batch 171 / 312 : total OK 4902 NOK 570
batch 172 / 312 : total OK 4931 NOK 573
batch 173 / 312 : total OK 4955 NOK 581
batch 174 / 312 : total OK 4986 NOK 582
batch 175 / 312 : total OK 5012 NOK 588
batch 176 / 312 : total OK 5042 NOK 590
batch 177 / 312 : total OK 5072 NOK 592
batch 178 / 312 : total OK 5097 NOK 599
batch 179 / 312 : total OK 5125 NOK 603
batch 180 / 312 : total OK 5155 NOK 605
batch 181 / 312 : total OK 5184 NOK 608
batch 182 / 312 : total OK 5212 NOK 612
batch 183 / 312 : total OK 5243 NOK 613
```

```
batch 184 / 312 : total OK 5273 NOK 615
batch 185 / 312 : total OK 5304 NOK 616
batch 186 / 312 : total OK 5334 NOK 618
batch 187 / 312 : total OK 5359 NOK 625
batch 188 / 312 : total OK 5387 NOK 629
batch 189 / 312 : total OK 5413 NOK 635
batch 190 / 312 : total OK 5442 NOK 638
batch 191 / 312 : total OK 5469 NOK 643
batch 192 / 312 : total OK 5498 NOK 646
batch 193 / 312 : total OK 5524 NOK 652
batch 194 / 312 : total OK 5553 NOK 655
batch 195 / 312 : total OK 5582 NOK 658
batch 196 / 312 : total OK 5612 NOK 660
batch 197 / 312 : total OK 5641 NOK 663
batch 198 / 312 : total OK 5672 NOK 664
batch 199 / 312 : total OK 5700 NOK 668
batch 200 / 312 : total OK 5729 NOK 671
batch 201 / 312 : total OK 5757 NOK 675
batch 202 / 312 : total OK 5787 NOK 677
batch 203 / 312 : total OK 5814 NOK 682
batch 204 / 312 : total OK 5846 NOK 682
batch 205 / 312 : total OK 5873 NOK 687
batch 206 / 312 : total OK 5900 NOK 692
batch 207 / 312 : total OK 5931 NOK 693
batch 208 / 312 : total OK 5960 NOK 696
batch 209 / 312 : total OK 5987 NOK 701
batch 210 / 312 : total OK 6015 NOK 705
batch 211 / 312 : total OK 6045 NOK 707
batch 212 / 312 : total OK 6075 NOK 709
batch 213 / 312 : total OK 6104 NOK 712
batch 214 / 312 : total OK 6135 NOK 713
batch 215 / 312 : total OK 6163 NOK 717
batch 216 / 312 : total OK 6191 NOK 721
batch 217 / 312 : total OK 6221 NOK 723
batch 218 / 312 : total OK 6252 NOK 724
batch 219 / 312 : total OK 6279 NOK 729
batch 220 / 312 : total OK 6310 NOK 730
batch 221 / 312 : total OK 6339 NOK 733
batch 222 / 312 : total OK 6367 NOK 737
batch 223 / 312 : total OK 6394 NOK 742
batch 224 / 312 : total OK 6423 NOK 745
batch 225 / 312 : total OK 6451 NOK 749
batch 226 / 312 : total OK 6480 NOK 752
batch 227 / 312 : total OK 6507 NOK 757
batch 228 / 312 : total OK 6536 NOK 760
batch 229 / 312 : total OK 6566 NOK 762
batch 230 / 312 : total OK 6594 NOK 766
batch 231 / 312 : total OK 6623 NOK 769
batch 232 / 312 : total OK 6652 NOK 772
batch 233 / 312 : total OK 6681 NOK 775
batch 234 / 312 : total OK 6713 NOK 775
batch 235 / 312 : total OK 6744 NOK 776
batch 236 / 312 : total OK 6774 NOK 778
batch 237 / 312 : total OK 6803 NOK 781
batch 238 / 312 : total OK 6831 NOK 785
batch 239 / 312 : total OK 6861 NOK 787
batch 240 / 312 : total OK 6889 NOK 791
batch 241 / 312 : total OK 6919 NOK 793
batch 242 / 312 : total OK 6951 NOK 793
batch 243 / 312 : total OK 6980 NOK 796
batch 244 / 312 : total OK 7009 NOK 799
batch 245 / 312 : total OK 7035 NOK 805
batch 246 / 312 : total OK 7066 NOK 806
batch 247 / 312 : total OK 7096 NOK 808
batch 248 / 312 : total OK 7124 NOK 812
batch 249 / 312 : total OK 7151 NOK 817
batch 250 / 312 : total OK 7176 NOK 824
batch 251 / 312 : total OK 7204 NOK 828
batch 252 / 312 : total OK 7233 NOK 831
batch 253 / 312 : total OK 7263 NOK 833
batch 254 / 312 : total OK 7293 NOK 835
batch 255 / 312 : total OK 7320 NOK 840
```

```
batch 256 / 312 : total OK 7352 NOK 840
batch 257 / 312 : total OK 7382 NOK 842
batch 258 / 312 : total OK 7413 NOK 843
batch 259 / 312 : total OK 7441 NOK 847
batch 260 / 312 : total OK 7472 NOK 848
batch 261 / 312 : total OK 7500 NOK 852
batch 262 / 312 : total OK 7530 NOK 854
batch 263 / 312 : total OK 7560 NOK 856
batch 264 / 312 : total OK 7589 NOK 859
batch 265 / 312 : total OK 7617 NOK 863
batch 266 / 312 : total OK 7647 NOK 865
batch 267 / 312 : total OK 7673 NOK 871
batch 268 / 312 : total OK 7704 NOK 872
batch 269 / 312 : total OK 7731 NOK 877
batch 270 / 312 : total OK 7761 NOK 879
batch 271 / 312 : total OK 7788 NOK 884
batch 272 / 312 : total OK 7818 NOK 886
batch 273 / 312 : total OK 7840 NOK 896
batch 274 / 312 : total OK 7865 NOK 903
batch 275 / 312 : total OK 7894 NOK 906
batch 276 / 312 : total OK 7925 NOK 907
batch 277 / 312 : total OK 7956 NOK 908
batch 278 / 312 : total OK 7987 NOK 909
batch 279 / 312 : total OK 8016 NOK 912
batch 280 / 312 : total OK 8041 NOK 919
batch 281 / 312 : total OK 8072 NOK 920
batch 282 / 312 : total OK 8102 NOK 922
batch 283 / 312 : total OK 8130 NOK 926
batch 284 / 312 : total OK 8157 NOK 931
batch 285 / 312 : total OK 8186 NOK 934
batch 286 / 312 : total OK 8213 NOK 939
batch 287 / 312 : total OK 8240 NOK 944
batch 288 / 312 : total OK 8269 NOK 947
batch 289 / 312 : total OK 8296 NOK 952
batch 290 / 312 : total OK 8325 NOK 955
batch 291 / 312 : total OK 8353 NOK 959
batch 292 / 312 : total OK 8382 NOK 962
batch 293 / 312 : total OK 8412 NOK 964
batch 294 / 312 : total OK 8443 NOK 965
batch 295 / 312 : total OK 8474 NOK 966
batch 296 / 312 : total OK 8502 NOK 970
batch 297 / 312 : total OK 8530 NOK 974
batch 298 / 312 : total OK 8560 NOK 976
batch 299 / 312 : total OK 8589 NOK 979
batch 300 / 312 : total OK 8618 NOK 982
batch 301 / 312 : total OK 8648 NOK 984
batch 302 / 312 : total OK 8676 NOK 988
batch 303 / 312 : total OK 8703 NOK 993
batch 304 / 312 : total OK 8732 NOK 996
batch 305 / 312 : total OK 8763 NOK 997
batch 306 / 312 : total OK 8793 NOK 999
batch 307 / 312 : total OK 8824 NOK 1000
batch 308 / 312 : total OK 8855 NOK 1001
batch 309 / 312 : total OK 8884 NOK 1004
batch 310 / 312 : total OK 8914 NOK 1006
batch 311 / 312 : total OK 8942 NOK 1010
batch 312 / 312 : total OK 8969 NOK 1015
Final accuracy: 89.690000
```

In [7]:

```
! python3 driver.py --exec_mode throughput_test --bitfile ../bitfile/finn-accel.bit --ba
tchsize 32
```

```
Results written to nw_metrics.txt
```

In [8]:

```
! cat nw_metrics.txt
#less device.py
```

```
{'runtime[ms]': 6.882190704345703, 'throughput[images/s]': 4649.682255941246, 'DRAM_in_ba
```

ndwidth[Mb/s]': 3.6453508886579367, 'DRAM_out_bandwidth[Mb/s]': 0.18598729023764982, 'fcl
k[mhz]': 99.999, 'batch_size': 32, 'fold_input[ms]': 0.05435943603515625, 'pack_input[ms]
': 0.05507469177246094, 'copy_input_data_to_device[ms]': 0.30541419982910156, 'copy_outpu
t_data_from_device[ms]': 0.10704994201660156, 'unpack_output[ms]': 86.59172058105469, 'un
fold_output[ms]': 0.032901763916015625}

8.5 Validation code

```
# Copyright (c) 2020 Xilinx, Inc.
# All rights reserved.
#
# Redistribution and use in source and binary forms, with or without
# modification, are permitted provided that the following conditions are met:
#
# * Redistributions of source code must retain the above copyright notice, this
#   list of conditions and the following disclaimer.
#
# * Redistributions in binary form must reproduce the above copyright notice,
#   this list of conditions and the following disclaimer in the documentation
#   and/or other materials provided with the distribution.
#
# * Neither the name of Xilinx nor the names of its
#   contributors may be used to endorse or promote products derived from
#   this software without specific prior written permission.
#
# THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
# AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
# IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
# DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE
# FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
# DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
# SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
# CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
# OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
# OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

import argparse
import numpy as np
from driver import io_shape_dict
from driver_base import FINNExampleOverlay

if __name__ == "__main__":
    parser = argparse.ArgumentParser(
        description="Validate top-1 accuracy for FINN-generated accelerator"
    )
    parser.add_argument(
        "--batchsize", help="number of samples for inference", type=int, default=32
    )
    parser.add_argument(
        "--dataset", help="dataset to use (mnist of cifar10)", default="mnist"
    )
    parser.add_argument(
        "--platform", help="Target platform: zynq-iodma alveo", default="zynq-iodma"
    )
    parser.add_argument(
        "--bitfile", help='name of bitfile (i.e. "resizer.bit")', default="../bitfile/finn-
accel.bit"
    )
    parser.add_argument(
        "--dataset_root", help="dataset root dir for download/reuse",
default="/home/xilinx/jupyter_notebooks/deploy-fpga-bis-on-pynq/driver/data2/FashionMNIST/raw"
    )
    # parse arguments
    args = parser.parse_args()
    bsize = args.batchsize
    dataset = args.dataset
    bitfile = args.bitfile
    platform = args.platform
    dataset_root = args.dataset_root

    if dataset == "mnist":
        from dataset_loading import mnist

        trainx, trainy, testx, testy, valx, valy = mnist.load_mnist_data(
            dataset_root, download=False, one_hot=False
```

```
    )
elif dataset == "cifar10":
    from dataset_loading import cifar

    trainx, trainy, testx, testy, valx, valy = cifar.load_cifar_data(
        dataset_root, download=True, one_hot=False
    )
else:
    raise Exception("Unrecognized dataset")

test_imgs = testx
test_labels = testy

ok = 0
nok = 0
total = test_imgs.shape[0]

driver = FINNExampleOverlay(
    bitfile_name=bitfile,
    platform=platform,
    io_shape_dict=io_shape_dict,
    batch_size=bsize,
    runtime_weight_dir="runtime_weights/",
)

n_batches = int(total / bsize)
limit = n_batches*bsize
test_imgs = test_imgs[:limit,:,:].reshape(n_batches, bsize, -1)
test_labels = test_labels[:limit,].reshape(n_batches, bsize)
#test_imgs = test_imgs.reshape(n_batches, bsize, -1)
#test_labels = test_labels.reshape(n_batches, bsize)

for i in range(n_batches):
    ibuf_normal = test_imgs[i].reshape(driver.ibuf_packed_device[0].shape)
    exp = test_labels[i]
    driver.copy_input_data_to_device(ibuf_normal)
    driver.execute_on_buffers()
    obuf_packed = np.empty_like(driver.obuf_packed_device[0])
    driver.copy_output_data_from_device(obuf_packed)
    obuf_folded = driver.unpack_output(obuf_packed)
    obuf_normal = driver.unfold_output(obuf_folded)
    results= np.argmax(obuf_normal, axis=1)
    ret = np.bincount(results.flatten() == exp.flatten())
    nok += ret[0]
    ok += ret[1]
    print("batch %d / %d : total OK %d NOK %d" % (i + 1, n_batches, ok, nok))

acc = 100.0 * ok / (total)
print("Final accuracy: %f" % acc)
```

Bibliography

[1] finn/cnv_end2end_example.ipynb at main · xilinx/finn. https://github.com/Xilinx/finn/blob/main/notebooks/end2end_example/bnn-pynq/cnv_end2end_example.ipynb. (Accessed on 12/31/2022).

[2] Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade*, pages 437–478. Springer, 2012.

[3] Jungwook Choi, Zhuo Wang, Swagath Venkataramani, Pierce I-Jen Chuang, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. Pact: Parameterized clipping activation for quantized neural networks. *arXiv preprint arXiv:1805.06085*, 2018.

[4] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

[5] Daniel Godoy. Hyper-parameters in action! part ii — weight initializers — by daniel godoy — towards data science. https://towardsdatascience.com/hyper-parameters-in-action-part-ii-weight-initializers-35aee1a28404. (Accessed on 10/25/2022).

[6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification, 2015.

[8] Mauro Castelli Ibrahem Kandel. The effect of batch size on the generalizability of the convolutional neural networks on a histopathology dataset - sciencedirect. https://www.sciencedirect.com/science/article/pii/S2405959519303455?via%3Dihub. (Accessed on 10/22/2022).

[9] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[10] Pytorch. Sgd — pytorch 1.12 documentation. https://pytorch.org/docs/stable/generated/torch.optim.SGD.html. (Accessed on 10/26/2022).

[11] Pavlo M Radiuk. Impact of training set batch size on the performance of convolutional neural networks for diverse datasets. 2017.

[12] Timo Stöttner. Why data should be normalized before training a neural network — by timo stöttner — towards data science. https://towardsdatascience.com/why-data-should-be-normalized-before-training-a-neural-network-c626b7f66c7d. (Accessed on 10/21/2022).

[13] Cs231n convolutional neural networks for visual recognition. https://cs231n.github.io/neural-networks-3/. (Accessed on 10/22/2022).

[14] Cs231n convolutional neural networks for visual recognition. https://cs231n.github.io/convolutional-networks/#layers. (Accessed on 11/04/2022).

[15] D Randall Wilson and Tony R Martinez. The general inefficiency of batch training for gradient descent learning. *Neural networks*, 16(10):1429–1451, 2003.