Training a simple Classifier

Hardware for Artificial Intelligence Fundamentals - Lab

Hardware for Artificial Intelligence Group

TECHNISCHE UNIVERSITÄT DARMSTADT

Winter 2024
Sheet 1

In this exercise, you will learn to train a simple classifier. We are going to use the **scikit-learn**[1] library for this purpose. You will have to use the library's online documentation to figure out how to use the suggested functions. Additionally, there are many tutorials on its website if you want further examples.

Task 1.1: Our Dataset

Our first dataset consists of measurements for different species of penguins, found around the Palmer research station in Antarctica[2]. There are three different species: *Adelie*, *Chinstrap*, and *Gentoo* (see figure 1). For each penguin its *species*, *weight*, *flipper length*, *culmen length*, and *culmen depth* are recorded. (Culmen is basically the bill of the penguin.) Our goal is to predict the penguin species based on those measurements.

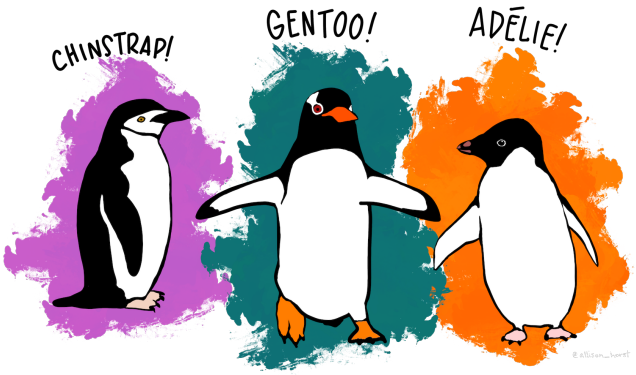

Figure 1: Species of penguins in the dataset. Artwork by @allison_horst.

1.1a) Downloading the data

The first step is to load the dataset into our Python script. Our chosen dataset is available on **OpenML**[3] so we can use sklearns bultin `fetch_openml` function, as shown in the following code snippet.

```
# retrieve dataset as pandas frame
from sklearn.datasets import fetch_openml
penguins = fetch_openml(name='penguins', parser="auto", as_frame=True).frame
```

The `as_frame=True` argument allows us to receive the result as a pandas dataframe[4], which allows for easy manipulation of tabular data.

[1] https://scikit-learn.org/
[2] https://allisonhorst.github.io/palmerpenguins/
[3] https://openml.org/search?type=data&status=active&id=42585&sort=runs
[4] https://pandas.pydata.org/pandas-docs/stable/reference/frame.html

Training a simple Classifier

1.1b) Preparing the data

1. Several rows/samples of the penguin data are missing values, which are marked `NA` when using pandas dataframes. Use the function `dropna` in combination with the `axis` argument to remove all rows that are missing values.
2. Select and separate the required features. Our target is the column **species**. For the input we want the columns **culmen_length_mm**, **culmen_depth_mm**, **flipper_length_mm**, and **body_mass_g**. Create a new dataframe for the target and another one for the input.
3. The dataset comes with no predetermined test and training set. Use sklearns `train_test_split` function to split the data accordingly. A test size of $40\,\%$ seems to be a good choice. Do you need to use the `stratify` argument?

Task 1.2: Decision Tree

For the first classifier, we are going to train a Decision Tree.

1.2a) Training a Decision Tree Classifier

Use sklearns `DecisionTreeClassifier` class and its `fit` method to train a Tree on our training data.

1.2b) Testing the Classifier

Now we need to test our classifier. Predict the penguin species for each sample in the test dataset using your tree. Use the `accuracy_score` function to determine the accuracy of your predictions, which should be somewhere in the $90 - 95\,\%$ range.

1.2c) Randomness

Run your script multiple times. Did you notice that your accuracy slightly changes with each run? Many tasks in machine learning require random decisions, for example, the `train_test_split` function determines the split randomly but also the decision tree training makes use of randomness. To have repeatable results it is good practice to fix the random seed[5], the initial value which is used by the pseudo-random number generator. Most of the sklearn functions accept the `randomstate` argument to achieve that.

1.2d) Visualization

Use the function `export_graphviz` to export a representation of your decision tree. The output will be in the graphviz dot language format. You can either convert it into an image using `dot -Tpng tree.dot -o tree.png`, if you have the tools installed, or use an online converter[6].

[5] `https://en.wikipedia.org/wiki/Random_seed`
[6] `https://dreampuf.github.io/GraphvizOnline/`

Training a neural Network

Hardware for Artificial Intelligence Fundamentals - Lab

Hardware for Artificial Intelligence Group

TECHNISCHE UNIVERSITÄT DARMSTADT

Winter 2024
Sheet 2

Today's goal is to train our first neural Network. We will continue to use the dataset from the first exercise, but this time we are going to use **PyTorch**[1] for training.

Task 2.1: Our Dataset

Our mission is still the classification of penguins based on measurements. Go ahead and download the data as done in exercise *1.1a*, before continuing.

2.1a) Data preparation for neural Networks

There are two "problems" we did not have to care about whilst training the decision tree. But they are relevant for neural Networks and PyTorch.

1. The target is a string, but this time we need to encode our target as a number.
2. Measurements from penguins are within a range that starts far from zero.[2] Training our network will be more successful if we transform measurements into the range $[0, 1]$.

Sklearn provides tools for both our problems: `LabelEncoder` and `MinMaxScaler` classes from `sklearn.preprocessing`. The `MinMaxScaler` shall be used on the input data to ensure each feature is in the range $[0, 1]$. Similarly the `LabelEncoder` will transform the target data to class numbers. Both classes contain the `fit_transform` method, which performs initialization of the object and transformation of the data in one step.

After that is the perfect time to repeat the steps from exercise *1.1b* and split the data into training and test data.

2.1b) PyTorch DataLoader

PyTorch comes with a container class that handles all the batching (and much more): the `DataLoader`. To initialize the DataLoader it requires the dataset to be in an appropriate format. Unfortunately, the data we receive from the sklearn utilities does not work out of the box. The following snippet provides you with a class that handles the conversion between sklearn and PyTorch, as well as an example of how to use it.

```
class SklearnDataSet(torch.utils.data.Dataset):
    def __init__(self, x, y):
        self.x = torch.from_numpy(x).float()
        self.y = y

    def __len__(self):
        return self.x.size(dim=0)

    def __getitem__(self, idx):
```

[1] `https://pytorch.org/`
[2] Have a look at the data, there are no weightless penguins in Antarctica.

```
        return self.x[idx], self.y[idx]

data_test = SklearnDataSet(X_test, y_test)
test = torch.utils.data.DataLoader(data_test)
```

Create DataLoaders for your training and your test data. For your training DataLoader you should use a batch size of 5 and turn on shuffling of the data between epochs.

Task 2.2: Training a neural Network

2.2a) Defining a Network

To create your network, create a class that inherits from `torch.nn.Module`. It is important to call the constructor of the parent class, as this will ensure all the parameters of your layer will be registered. The ordering of the layers and activation functions is then implemented in the `forward(self, x)` method, x being the batch input data. An example is shown in the following snippet.

```
import torch.nn as nn
torch.manual_seed(42)

class Network(nn.Module):
    def __init__(self):
        super().__init__()
        self.f = nn.Linear(2, 1)

    def forward(self, x):
        return self.f(x)

model = Network()
```

We are going to train a fully connected linear network, with two hidden layers, one output layer, and a ReLU activation function for each hidden layer. The layers contain 4, 4, and 3 neurons, in that order.

2.2b) Training loop

To train our network we need to decide which optimizer and which loss function to use. We choose Stochastic Gradient Descent with a learning rate of 0.1 and Cross Entropy Loss, as introduced in the lecture.

```
lossFunc = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.1)
```

The following training loop structure can be used regardless of the chosen loss function, optimizer and model. It should work for most models and can easily be adapted.

```
epoch = 2
model.train()
for _ in range(epoch):
    for _, (inputs, targets) in enumerate(train):
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = lossFunc(outputs, targets)

        loss.backward()
        optimizer.step()
```

First, the model is put into training mode. It means that gradient calculation is turned on, which we need for back-propagation. Then we iterate for our number of chosen epochs and for each epoch we iterate over our training DataLoader.

In the inner loop, the actual training step for one batch happens. The gradient information of the optimizer is first reset and then a prediction is made using our model. Prediction and actual targets are passed to the loss function, which then performs the calculation backward through the network. Finally, the optimizer updates the model weights.

Modify the training loop in such a way, that you can plot the training loss across the epochs. The X-axis should contain the epoch number and the Y-axis should contain the average loss of that epoch. You can use matplotlib[3] for this. Choose a suitable number of epochs for your training.

Task 2.3: Testing a neural Network

2.3a) Accuracy

This time we are not going to use a sklearn function to determine the accuracy. Instead, we use the `MultiClassAccuracy` from torchmetrics[4]. We will use that class in future exercises as well, so now is a good time to get acquainted.

Whilst instantiating the class object you have to specify the number of classes (in our case that is 3, because there are three different species of penguins). To insert data you have to call the `update(prediction, target)` with the model prediction and the actual target respectively. At the end you can call the `compute()` method, which returns the accuracy. Modify the snippet below to determine the accuracy.

```
with torch.inference_mode():
    for _, (inputs, targets) in enumerate(test):
        # TODO use model to make prediction
        # TODO update your accuracy object

    # TODO print final accuracy
```

The first line of the snippet makes sure, that your model is in inference mode. No gradients will be calculated. Then it is simply a matter of iterating over the test DataLoader. Your final accuracy should be $98\,\%$ or more.

2.3b) Accessing the model parameters

Are you curious about the trained parameter values of your model? Go ahead and run the following snippet.

```
print(model.state_dict())
```

It should print all your model's parameters. The state_dict of a model can also be used to save and load values from a file. Useful if your model is large enough that training requires more than a couple of seconds.

[3] https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.plot.html
[4] https://lightning.ai/docs/torchmetrics/stable/

Training a neural Network for MNIST

Hardware for Artificial Intelligence Fundamentals - Lab

Hardware for Artificial Intelligence Group

TECHNISCHE UNIVERSITÄT DARMSTADT

Winter 2024
Sheet 3

In this exercise, we are going to move on from our beloved penguin dataset and try to classify images. Image classification is a lot harder than our previous task, and therefore requires a larger neural Network. To be able to train the model in a shorter time, we will use the GPU for training.

Task 3.1: The MNIST Dataset

The MNIST dataset consists of handwritten digits from 0 to 9. Each image contains a single digit and has a resolution of 28×28 pixel in grayscale. The training set consists of $60,000$ images and the testing set of $10,000$ images. A few examples from the training set are shown in figure 1.

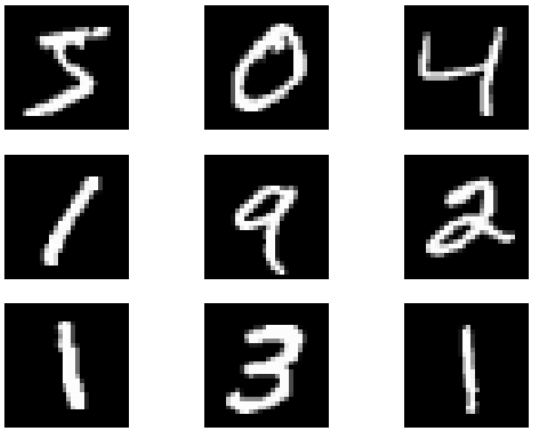

Figure 1: Example of digits in the MNIST dataset.

3.1a) Downloading and preparing the data

MNIST is so popular for training neural networks that it is already included with the torchvision package. The following listing shows how to create the DataLoader for the testing set. You should already be familiar with the concept of datasets and dataloaders from the previous exercise.

```
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

tf = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])
test = DataLoader(datasets.MNIST(download=True, train=False, transform=tf))
```

Let's take a look at the arguments for the MNIST dataset. `download=True` specifies that we want to download the dataset if it is not already present in the current (or otherwise specified) folder. To select the testing dataset we specified `train=False`. Another thing we are specifying is transformations that will be applied to the data.

First, we convert the images (integer 0-255) to tensors (floating point) in the range 0-1 using the `ToTensor()` transformation. Tensors are like matrices but also allow to keep track of gradients, which makes them a foundational building block of pytorch. Lastly, the images are normalized. The parameters of `Normalize(...)` are the mean and the standard deviation.

You can now create the training DataLoader.

Task 3.2: Training our Network

3.2a) Defining the Network

You can modify the model from the previous exercise, but the number of layers should be sufficient. The inputs of the first layer need to be adjusted according to the image size and the number of neurons in the last layer to the number of classes. Adjust the number of neurons in the hidden layers to 25 each.

Our dataset consists of two-dimensional images, but the linear layers of the model only work with one-dimensional data. To flatten the data you should use an instance of `nn.Flatten()`, which you need to instantiate within the constructor and then use during the `forward(..)` method.

3.2b) Using the GPU

Our model has a lot more parameters to train[1] this time, therefore we are using the GPU. The following listing shows how to move tensors and models to a specific device. Unless specified pytorch will always create a model/tensor on the CPU.

```
# select a device, if cuda is available use it, otherwise use the CPU
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

a = torch.tensor()
b = a.to(device)

model = Network().to(device)
```

We start by selecting a cuda (=Nvidia GPU) device[2] if one is available, otherwise we fall back to the CPU. The tensor a is created and moved to our device using the `to(device)` method. For tensors this method returns a copy of the tensor on the specified device. That means tensor a is still on the CPU but tensor b is on the specified device.

Similarly, the model is moved to the device. (For models the `to(device)` returns a copy of the model on the device but also moves the original model to the device.)

3.2c) Training loop

The training loop is equivalent to the one from the previous exercise. You only need to move the `inputs/targets` tensors to the device as well.

For this model, you might want to use the Adam optimizer instead of stochastic gradient descent and a lower learning-rate, as shown in the following listing.

[1] Layer 1: $(28 \cdot 28 + 1) \cdot 25 = 19625$, layer 2: $(25 + 1) \cdot 25 = 650$, layer 3: $(25 + 1) \cdot 10 = 260$, total: 20535

[2] In this snippet we are using the "cuda:0" device, which is simply the first one. Our GPU Server has four available cuda devices, therefore you might want to use another index (1-3) to select a less utilized device.

```
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
```

Task 3.3: Testing the Network

3.3a) Accuracy

Similar to the training loop, there is very little you need to adjust in your testing routine from the previous exercise. Besides the tensors you also need to move your instance of `MulticlassAccuracy`.

Your accuracy should be above $90\,\%$, otherwise, you need to adjust your training to improve it.

3.3b) Saving the model parameters

Training takes a lot of time for larger models, therefore you might want to save it for later use.

```
# saving the state dict
torch.save(model.state_dict(), "myModel.pt")

# restoring model parameters from a saved file
model.load_state_dict("myModel.pt")
```

Pruning a neural Network

Hardware for Artificial Intelligence Fundamentals - Lab

Hardware for Artificial Intelligence Group

TECHNISCHE UNIVERSITÄT DARMSTADT

Winter 2024
Sheet 4

Some connections/weights in a neural network make almost no contribution to the overall result. It makes sense to remove those connections to save computational cost, so-called pruning.

Task 4.1: Pruning after training

We start by pruning a model after training. We will use global unstructured pruning, as explained in the lecture. Which weights[1] to prune will be decided upon the L1-norm.

4.1a) Load your network

For now, we continue with the network from the previous exercise. Load your saved weights. Test the accuracy again if you don't remember it.

4.1b) Prune it

Prune $60\,\%$ of your network using the following snippet. You need to adapt the `parameters_to_prune` to your model.

```python
import torch.nn.utils.prune as prune
parameters_to_prune = (
    (model.fc1, 'weight'),
    # ...
)

prune.global_unstructured(
    parameters_to_prune,
    pruning_method=prune.L1Unstructured,
    amount=0.4,
)
```

Test your accuracy again. Notice a decrease?

4.1c) Finetuning

One common approach to reduce the accuracy loss due to pruning is finetuning. The pruned model is again trained for a few epochs, often with a reduced learning rate.

Finetune your model and test the accuracy again. Is it better than before finetuning?

[1] We are going to ignore bias for now.

Task 4.2: Pruning whilst training

Finetuning recovers some of the accuracy of a pruned network. Another approach is iteratively pruning and finetuning for smaller amounts.

4.2a) Iterative pruning

We are going to train the network from scratch this time. Modify your training code from last time in such a way that every n epochs a small amount is pruned.

The total pruning amount should also add up to $60\,\%$ as in the previous task. It is usually better to do some pruning every 5-10 epochs instead of every epoch.

Test your model, is the accuracy better than the finetuned model?

Task 4.3: Save your model

Pytorch is using Matrix multiplication to compute linear layers. It is not possible to "remove" connections by removing entries in a matrix, therefore pytorch is keeping track of those removed connections in a separate matrix, the mask. The mask consists of 0s (removed) and 1s (not removed) and is elementwise multiplied with the weight matrix. To save and afterward successfully load the model from a file, we need to combine the mask and weight matrix.

```
# display the model parameter names
print(model.state_dict().keys())
```

Note the `.weight_orig` and `.weight_mask` keys in the output of the previous listing.

The `remove` function takes care of the job for us.

```
prune.remove(model.fc1, 'weight')  # repeat for your layers
```

Logic Synthesis

Hardware for Artificial Intelligence Fundamentals - Lab

Hardware for Artificial Intelligence Group

TECHNISCHE UNIVERSITÄT DARMSTADT

Winter 2024
Sheet 5

In this exercise we are going to create an accelerator for a single neuron. We will use industry standard electronic design automation tools, which are installed on the lab PCs. For guidance on using the lab PCs please read the provided document “How to use the lab PCs”.

Task 5.1: Describing Adders and Multipliers in Hardware

As discussed during the lecture, there are benefits to embedding weights directly into to the hardware. We will used signed integers for this exercise, you do not need to worry about shifting the zero point.

5.1a) Adder

Create a verilog module for an adder. The bit-with for inputs (a, b) and output (y) should be 16 bit[1]. Use the template below.

Listing 1: add.v

```
module ADD (
    input wire signed [15:0] a,
    input wire signed [15:0] b,
    output wire signed [15:0] y
);
    // your code here
endmodule
```

5.1b) Multiplier

Create a verilog module for a multiplier. The bit-with for the inputs (a, b) should be 8 bit and therefore the result (y) is 16 bit wide.

Listing 2: mul.v

```
module MUL (
    input wire signed [7:0] a,
    input wire signed [7:0] b,
    output wire signed [16:0] y
);
    // your code here
endmodule
```

[1]Theoretically, the addition of to 16 bit numbers could result in a 17 bit number. We simply ignore this for now in our design.

5.1c) Synthesis

To create an ASIC from our modules we need a technology library, which describes the available gates. The library is available on moodle, the following script describes to Design Compiler what it is suppsed to do: compile our module.

Listing 3: synthesis.tcl

```
set target_library NangateOpenCellLibrary_typical.db ; # tell synopsis which library to use
set link_library NangateOpenCellLibrary_typical.db

read_file add.v ; # read specified input file
link ; # link design with library
compile_ultra ; # compile design with extra effort
write -f verilog -hierarchy -output netlist_add.v ; # save result as verilog netlist
quit
```

Use `module load syn/2017.09-SP3` to make Synopsys Design Compiler available in your shell. You can then execute the compilation script with `dc_shell -f synthesis.tcl`.

Have a look at the netlist. There is no instance of the adder anymore, instead everything is replaced by the modules from the library we used. Go ahead and compile the netlist for the multiplier as well.

5.1d) Simulation

Testing a hardware design is a difficult process. It is not feasible to test all input combinations and transitions, instead we provide you with a testvector. Download the testvector and testbench from moodle and place it into the same directory as your netlists.

Listing 4: testbench.tcl

```
vlib work ; # create library work
vlog testbench_add.v ; # add file to library
vlog add.v
vsim work.testbench_add ; # simulate module testbench_add in lib work
run 10 us ; # run simulation for 10 us
quit -f
```

Use `module load modelsim` to load the modelsim package. You can now execute the testbench `vsim -c -do "do testbench.tcl"`. It will tell you if your adder is working as expected or not. Modify the testbench for the multiplier and its testvector and test it as well.

Task 5.2: Creating a Neuron

Using our multiplier and adder circuits we want to create a neuron. It will have 4 inputs and the given weights (w_i) and bias (b) in table 1. The neuron should also have a ReLU function.

Table 1: Neuron parameters

Param	value
w_0	-5
w_1	10
w_2	27
w_3	-13
b	7

5.2a) Implementation

Extend the code below to implement the complete neuron. Use your multiplication and addition module for math operations. Do not forget to add code for the ReLU.

Listing 5: neuron.v

```
module NEURON (
    input wire signed [7:0] inp0,
    input wire signed [7:0] inp1,
    input wire signed [7:0] inp2,
    input wire signed [7:0] inp3,
    output wire signed [15:0] out
);
    parameter [7:0] w0 = -8'sd5;  // s: signed, d: decimal

    // your code here

endmodule
```

5.2b) Testing

Copy and modify the testbench/testvector[2] from the previous task to test your neuron implementation.

[2]Tip: you can use python to generate your testvector. `https://numpy.org/doc/stable/reference/generated/numpy.binary_repr.html`

Power and Area

Hardware for Artificial Intelligence Fundamentals - Lab

Hardware for Artificial Intelligence Group

TECHNISCHE UNIVERSITÄT DARMSTADT

Winter 2024
Sheet 6

The most important metrics for an ASIC are Power, Performance, and Area (PPA), which we will evaluate in this exercise. First we examine the Power and Area of a multiplier with two inputs. After that we fix the weight into the multiplier design and observe the optimizations done during synthesis.

Task 6.1: Ordinary Multiplier

For this part we will reuse the multiplier design from the previous exercise. Two signed 8 bit inputs and one 16 bit output.

6.1a) Synthesis

Unfortunately we need to synthesize the circuit again. Besides the netlist we also want to create an area report, which tells us how much area the synthesized circuit is actually requiring. For our power analysis we will need to know the delays of the gates and wires, which we save as a Standard Delay Format (SDF) file.

Listing 1: synthesis.tcl

```
set target_library NangateOpenCellLibrary_typical.db ; # tell synopsis which library to use
set link_library NangateOpenCellLibrary_typical.db

read_file add.v ; # read specified input file
link ; # link design with library
compile_ultra ; # compile design with extra effort
write -f verilog -hierarchy -output netlist_add.v ; # save result as verilog netlist
write_sdf sdf_full.sdf  ; # create SDF file
report_area > area-report.txt  ; # create area report
quit
```

Inspect the area report. We are interested in total cell area, which is measured in μm^2.

6.1b) Simulation

In order to determine the power usage we need to simulate the design to obtain switching activity of the gates. To do this we use modelsim and a testbench, just like in the last exercise.

Listing 2: simulation.tcl

```
vlib work
vlog testbench_mul.v
vlog netlist_mul.v
vlog NangateOpenCellLibrary.v
vsim -sdftype /testbench_mul/dut=sdf_full.sdf work.testbench  ; # annotate multiplier with sdf file
power add /testbench_mul/dut/*
run 10 us
```

Power and Area

```
power report -all -bsaif power_full.saif  ; # create power report
quit -f
```

For simulation we specify our SDF file to annotate delays and export a Switching Activity Interchange Format (SAIF) file which contains the switching information from simulation.

6.1c) Power Analysis

Now that we have obtained the switching information from simulation, we are able to determine the actual power usage. For analysis we use design compiler, as shown in the following script.

Listing 3: analysis.tcl

```
set target_library NangateOpenCellLibrary_typical.db  ; # specify library
set link_library NangateOpenCellLibrary_typical.db

read_verilog netlist_full.v  ; # read netlist
read_saif -input power_full.saif -instance_name testbench/mul  ; # read SAIF
report_power -analysis_effort high > report.txt  ; # create power report
quit
```

Open the report. The last row of the contained table tells us the relevant information.

Task 6.2: Fixed Multiplier

In this part of the exercise we will evaluate what happens when the weight of a multiplication is embedded and therefore synthesized into the multiplier design.

Our goal is to create an area/power report for every weight in the range $[-127, 127]$. To acheive this we create a parametrized multiplier design, then set the parameter. This design then gets synthesized, simulated and analyzed. We repeat this for every integer in the desired range.

6.2a) Multiplier design

The new multiplier will have only one 8 bit input but the 16 bit output remains unchanged. Extend the verilog module listed below.

Listing 4: mul_fixed.v

```
module MUL (
    input wire signed [7:0] a,
    output wire signed [16:0] y
);
    parameter WEIGHT = 0;  // parameters require a default value in verilog
    wire signed [7:0] b = WEIGHT;

    // your code here
endmodule
```

To instantiate a module with a parameter you would do it as in the following code snippet.

```
MUL #(.WEIGHT(7)) dut (/* connections here */);
```

6.2b) Synthesis, Simulation, and Analysis

In previous tasks we have already used scripts to automate design compiler and modelsim. Those scripts were written in the Tool Command Language (TCL), which is also capable of loops, variables and running external commands. We will make use of those features to automate the steps for our parameter.

First let us create the script for modelsim. Use the testbench provided on moodle, testvector is the same as in the previous task.

Listing 5: sim_fixed.tcl

```
vlib work
vlog testbench_fixed.v
vlog $1
vlog NangateOpenCellLibrary.v
vsim -sdftype /mul=$2 work.testbench
power add /testbench/mul/*
run 10 us
power report -all -bsaif $3
quit
```

You may notice the `$1, $2, $3` in the code. Those are arguments which are passed on the commandline to the script when invoking modelsim.

Next we need to create a script for design compiler. It will read the library and multiplier design. And then in a loop it will synthesize the design with a given parameter, execute our modelsim script, and then read back the SAIF file for analysis.

Create directories “logs” and “gen” before running the script.

Listing 6: synth_fixed.tcl

```
set target_library NangateOpenCellLibrary_typical.db
set link_library NangateOpenCellLibrary_typical.db

analyze -f verilog mul_fixed.v  ; # read file but do nothing else with it
for {set factor -127} {$factor <= 127} {incr factor} {  ; # loop over factor range
    set fSdf "gen/sdf_$factor.sdf"  ; # generate variables for filenames
    set fNetlist "gen/netlist_$factor.v"
    set fSaif "gen/power_$factor.saif"

    # create netlist and sdf
    elaborate -parameters WEIGHT=$factor MUL  ; # create design with parameter weight set
    rename_design $current_design "MUL_FIXED"  ; # rename design to MUL_FIXED (otherwise it's new name
    contains the parameter value. e.g., MUL_n127)

    compile_ultra
    report_area > "logs/area_$factor.txt"
    write -f verilog -hierarchy -output $fNetlist  ; # create netlist
    write_sdf $fSdf  ; # create sdf

    if { [ catch {
        # execute modelsim with arguments (filenames)
        exec vsim -- -c -do "do sim_fixed.tcl $fNetlist $fSdf $fSaif"

        # analyze result
        read_saif -input $fSaif -instance_name testbench/dut
    } err ] } {
        puts "No cells for weight $factor"
    }
    report_power -analysis_effort high > "logs/power_$factor.txt"

    # remove design
    remove_design "MUL_FIXED"
}
quit
```

You may wonder why there is a `catch` statement around the modelsim execution and reading the SAIF back in. For certain parameters (e.g., 0, 2) the design does not include any cells from the library because everything gets optimized away. In those cases modelsim tries to annotate cells but none exist, additionally there will be no switching to simulate.

6.2c) Plotting

Use Matplotlib[1] to create power and area plots. The X-Axis should contain the used factor and the Y-Axis the required total cell area/total power. You may want to use the following python functions to automatically parse the reports.

```
import glob
import re

def matchArea(data, keys):
    return [re.search(f"{key}:\\s+(.+)", data).group(1) for key in keys]

for file in glob.glob('logs/area_*.txt'):
    with open(file) as f:
        values = matchArea(f.read(), [
            'Combinational area', 'Buf/Inv area',
            'Noncombinational area', 'Total cell area'
        ])
        print(values)

def matchPower(data):
    vp = "([0-9\\.+-e]+)\\s[mnu]W"
    r = re.search(f"Total\\s+{vp}\\s+{vp}\\s+{vp}\\s+{vp}", data)
    return r.groups()

for file in glob.glob('logs/power_*.txt'):
    with open(file) as f:
        values = matchPower(f.read())
```

[1] https://matplotlib.org/stable/tutorials/pyplot.html

Quantization

Hardware for Artificial Intelligence Fundamentals - Lab

Hardware for Artificial Intelligence Group

TECHNISCHE UNIVERSITÄT DARMSTADT

Winter 2024
Sheet 7

The goal is to replace the floating point weights and operations by integers. This usually comes with latency and storage benefits. Additionally, integer arithmetic hardware is much cheaper than floating point hardware.

Task 7.1: Using a Library

Use the Optimum Quanto[1] Library to quantize your neural network from the second exercise.

7.1a) Post Training Quantization

Load your network and use the training data to calibrate the activations. Weights and biases can be quantized based on their magnitude. But the magnitude of the activations is dependant on the actual data, therefore we use the training data to determine the required ranges.

```
from quanto import quantize, qint8, Calibration

# instantiate and load your model here

# Quantization
quantize(model, weights=qint8, activations=qint8)

# Calibration
with torch.inference_mode():
    with Calibration(momentum=0.9):
        model(samples)  # you probably need to change this line
                        # the model should be applied to all training samples once
```

Test the accuracy of the now quantized model.

7.1b) Finetuning

To make up for quantization loss, it is a common strategy to train the quantized model for a couple more epochs.

Finetune for one epoch and test your accuracy again.

[1] https://github.com/huggingface/optimum-quanto

7.1c) Freeze Weights

Quanto is still storing the weights as floating point and converts them on the fly to integers. This is required to allow for training. To convert the weights to integers you may use the freeze function.

```
from optimum.quanto import freeze

freeze(model)

print(model.lin1.weight)  # change according to your layer names
```

Take a look at your integer values.

Task 7.2: Manual Quantization

This time we will do the post training quantization manually. One big issue is the growth of the activation size. If we do have 8 bit inputs and weights, the neurons of the first layer will produce a 16 bit output. The second layer will already have 24 bit outputs and so on. Real world accelerators use a scaling step after each layer, in which the activations are scaled back to 8 bit. But determining suitable scaling factors would be a bit too much for this lab, therefore we will start by training a neural network with a single layer.

7.2a) Single Layer Network

Use the provided network structure below. Train it on the penguin dataset like in exercise 2. Choose a suitable number of epochs/learningrate to yield a good accuracy.

```
class Network(nn.Module):
    def __init__(self):
        super().__init__()
        self.f1 = nn.Linear(4, 3)

    def forward(self, x):
        return self.f1(x)
```

7.2b) Quantization

Now quantize your network. We will use symmetric quantization, signed 8 bit, and zeropoint should be 0.

Given that $Z = 0$, it follows that $q_{\text{min}} = -q_{\text{max}}$ and $r_{\text{min}} = -r_{\text{max}}$. The 8 bit maximum value is $q_{\text{max}} = 127$, therefore $q_{\text{min}} = -127$. The biggest magnitude of weight/bias will dictate the scaling factor.

$$S = \frac{\max\left(|weight_i|, |bias_j|\right)}{q_{\text{max}}}$$

```
import numpy as np
# convert to numpy
weight, bias = model.f1.weight.detach().numpy(), model.f1.bias.detach().numpy()

# determine absolute maximum
# determine Scaling factor
# convert weight/bias to integer values
```

We need those values for the next exercise, so you probably want to save them.

Neural Network in Hardware

Hardware for Artificial Intelligence Fundamentals - Lab

Hardware for Artificial Intelligence Group

TECHNISCHE UNIVERSITÄT DARMSTADT

Winter 2024
Sheet 8

Convert the single layer neural network from the previous exercise into custom hardware. User your multiplier and adder modules.

Task 8.1: Network

8.1a) Hardware Description

Use the weights you obtained afer manual quantization in the last exercise. It is up to you if you want to create neuron modules or instantiate the multipliers with embedded weights directly in the network module.

Listing 1: network.v

```
module Network (
    input wire signed [7:0] inp0,
    input wire signed [7:0] inp1,
    input wire signed [7:0] inp2,
    input wire signed [7:0] inp3,
    output wire signed [15:0] out0,
    output wire signed [15:0] out1,
    output wire signed [15:0] out2
);

    // your code here

endmodule
```

8.1b) Simulation

Download the testbench from moodle and use modelsim to simulate your network accelerator.

Listing 2: simulation.tcl

```
vlib work
vlog testbench_network.v
vlog network.v
vlog add.v
vlog mul_fixed.v
# if you have decided to create additional modules you need to add the files here
vsim work.testbench_network
run 10 us
quit -f
```

You may notice a significant accuracy drop reported by the testbench. The reason for that is the coarse quantization approach we took. If you inspect the weights of the model quantized with the optimum quanto library again, you will notice that they use an individual scaling factor per neuron. This allows for a more finegrained control than our approach which looks at all weights/biases at the same time.