HW03 [ECE 720]

Digvijay Anand
200478940

Q1. Calculate the minimum time needed to drive a signal across a chip over a distance of 1 cm using repeaters.

Solution:

Let's consider a network of unit length inverter model to create a model for the delay of a wire made up a network of m, unit length inverters with other constraints/parameters as

- a wire of length, L,
- output resistance of repeater or driver, R_d
- input capacitance of repeater or driver, C_d
- resistance per-unit length of wire, r
- capacitance per-unit length of wire, c
- ratio of C_{in}/C_{int}, γ

We can estimate the optimal delay, $t_{p,min}$ and t_{p1} (delay with single load) using:

$$t_{p,min} = \left(1.38 + 1.02\sqrt{1+\gamma}\right)L\sqrt{R_d C_d r c}$$

$$t_{p1} = t_{p0}(1 + 1/\gamma) = 0.69\, R_d\, C_d\, (\gamma + 1)$$

Using above steps, we get the following for wire length, **L = 1 cm = 10_000 µm**:

Layer	c (fF/µm)	r (Ω/µm)	**opt. delay = $t_{p,min}$ (ps/cm)**	**t_{p1} (fs)**
M1	0.18	11.7	7.8	553
Intermediate	0.16	11.8	7.4	553
Global	0.18	3.0	**3.9**	553

with, R_d (Ω-um) = 548, C_d (fF/um) = 0.61, γ = 1.4

Thus the minimum time needed to drive a signal across a chip over a distance of 1 cm using repeaters = 3.9 ps/cm.

Q2. CAEML-PPA Dataset

a. Train an Area-Delay **predictor model**:
Solution: **Design: 'rocket'** **Training size: 150**

Modifications [**in bold**]:
hw03/p2/dataset.py:

```
...
class CAEMLPPADataset(Dataset):
	def __init__(self,design='rocket', num_samples = 500):
		self.features = ['Tclk','MaxTran','Uncertainty','Fanout']
		self.targets = ['Area','Cpath']
		self.featuresData=np.array([])
		self.targetsData=np.array([])
		csvfile = "./data/"+design+".csv"
		pd_csv = pd.read_csv(csvfile,skiprows=1)
		data = pd_csv.values
		data = data[:num_samples]
		print("\n[dataset size in dataset.py] Dataset size = ",
		data.shape)
...
```

hw03/p2/train.py:

```
...
# First     argument gives the number of training epochs
# Second    argument gives the number of dataset samples
# Third     argument gives the percentage of test dataset
if len(sys.argv)>1:
	epochs = int(sys.argv[1])
	num_samples = int(sys.argv[2])
	test_size = float(sys.argv[3])
else:
	epochs = 0
	num_samples = 0
	test_size = 0
if epochs > 0:
	print(f"Training for \t\t\t{epochs} epochs")
if num_samples > 0:
	print(f"====== Total dataset size \t{num_samples} samples")
if test_size > 0:
	print(f"====== Test size is \t\t{test_size*100}% \t\t=
	{num_samples*test_size} samples")

# Get cpu or gpu device for training
device = "cuda" if torch.cuda.is_available() else "cpu"
print("Using {} device".format(device))
```

```
random_seed=0
crossvalIter = ShuffleSplit(n_splits=50, test_size=test_size,
random_state=random_seed)
# crossvalIter = KFold(n_splits=10,shuffle=True,
random_state=random_seed)
torch.manual_seed(random_seed)
# Load Dataset
fullDS=CAEMLPPADataset(design='rocket', num_samples=num_samples)
crossvalDS=fullDS
...
```

$ python3 train.py {# of epochs} {# of samples} {test_size}
Note: bash argument num_samples in train.py overrides the one in dataset.py

Case I. Changing the total dataset size in dataset.py and train.py

$ python3 train.py 500 **150** 0.7

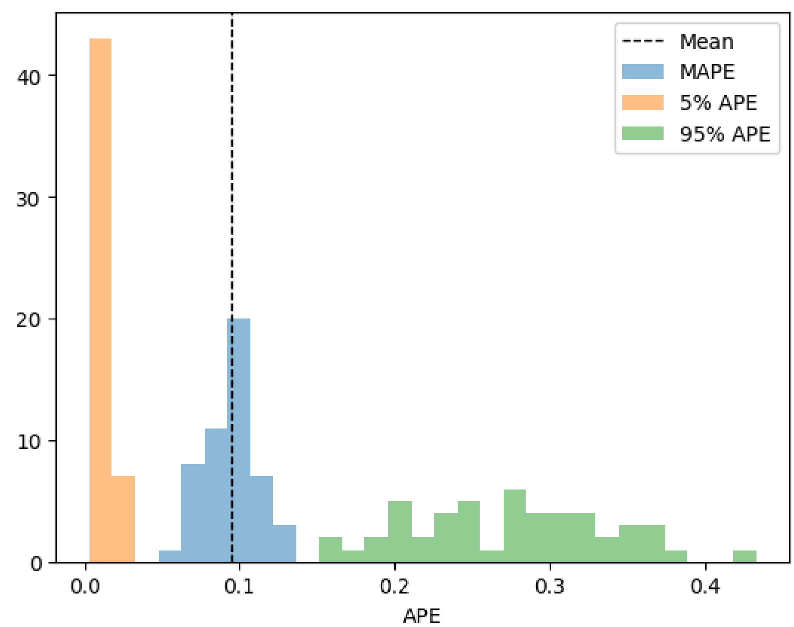

Fig. [Case I] error-summary image (500 150 0.7)

Overall dnMAPE mean = 0.094469
5%-95% confidence mean = 0.011282 - 0.274619
Overall mean TL/VL = 8.9223e-01

Case II. Changing only the training size in train.py

$ python3 train.py 500 **500** 0.7

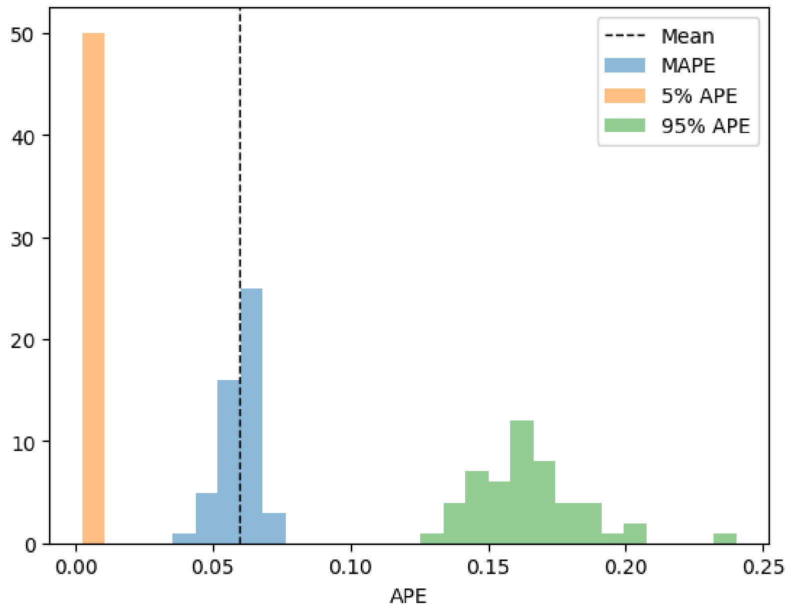

Fig. [Case II] error-summary image (500 500 0.7)

Overall dnMAPE mean	**= 0.059613**
5%-95% confidence mean	= 0.005604 - 0.164654
Overall mean TL/VL	= 9.1620e-01

Comparison:

Based on the error and performance metric,

- In the first case, the average prediction error is **9.45%**, while in the second case, it is lower at **5.96%**, indicating better prediction accuracy in the second case.

- The confidence interval in the second case (0.56% - 16.47%) is narrower, meaning less variability and more consistent predictions, compared to the first case (1.13% - 27.46%).

- In the first case, the ratio is **0.8922**, and in the second, it is **0.9162**. Both show good generalization, with the second case slightly better at balancing performance on training and validation data.

This implies that larger sample size is essential for better accuracy, performance and generalization. So, moving forward, I will use the model.pt obtained from **Case II**.

b. <u>Train an Area-Delay **adapter model**</u>:

<u>Solution</u>:

Here, I took the base model as mymodel4.pt. This particular split of the model has the best performance with the following attributes:

TL/VL	= 0.95219	log_nMSE	= -3.237674
Confidence_5%	= 0.000617	Confidence_95%	= 0.08649
dnMAPE	= 0.021596	90%_var	= 0.085873
# of epochs	= 2_500	test_size	= 0.7
Sample size	= 500	design	= rocket

While other mymodel{#split}.pt may be good in terms of dnMAPE or TL/VL, or log_nMSE, they show contrasting behavior across these attributes which raises concerns on their generalization capability.

<u>Step I</u>: cp mymodel4,pt mymodel.pt

<u>Step II</u>: $ python3 retrain.py **{# of epochs} {# of samples} {test_size}**
<u>hw03/p2/retrain.py</u> **Modified**

```
...
# First     argument gives the number of training epochs
# Second    argument gives the number of dataset samples
# Third     argument gives the percentage of test dataset

if len(sys.argv)>1:
      epochs = int(sys.argv[1])
      num_samples = int(sys.argv[2])
      test_size = float(sys.argv[3])
else:
      epochs = 0
      num_samples = 0
      test_size = 0
if epochs > 0:
      print(f"Re-training for \t\t\t{epochs} epochs")
if num_samples > 0:
      print(f"====== Total dataset size \t{num_samples} samples")
if test_size > 0:
      print(f"====== Test size is \t\t{test_size*100}% \t\t=
{num_samples*test_size} samples")


# Get cpu or gpu device for training
device = "cuda" if torch.cuda.is_available() else "cpu"
print("Using {} device".format(device))
```

```
random_seed=0
crossvalIter = ShuffleSplit(n_splits=50, test_size=test_size,
random_state=random_seed)
torch.manual_seed(random_seed)

# Load Dataset
# fullDS=CAEMLPPADataset(design='rocket_tiny')

fullDS=CAEMLPPADataset(design='cnn', num_samples=num_samples)
crossvalDS=fullDS
...
```

Step III: $ python3 retrain_script.py – test_size TEST_SIZE
I wrote a python script to gather statistics for a desired number of 500 training epoch runs required to predict the maximum utility.

hw03/p2/retrain_script.py

```
import os
import shutil
import glob
import argparse

# Parse command-line argument for test_size
parser = argparse.ArgumentParser(description='Run retraining with a
specified test_size.')
parser.add_argument(
    '--test_size',
    type=float,
    required=True,
    help='Example: --test_size 0.7'
)
args = parser.parse_args()
test_size = args.test_size
# Define the parameters for retraining
retrain_cmd     = "python3 retrain.py 500 500 {} > {}"
mv_png          = "adapter-error-summary.png"
new_png         = "adapter-error-summary_{}_{}.png"
sim_txt         = "adapter_dnMAPE_below_5_{:02d}_{}.txt"

#####################################
# Loop for 2*500 epochs to          #
# achieve atleast 5% dnMAPE score   #
# with maximum test_size            #
#                                   #
#####################################
for i in range(2):
```

```
    iteration_folder = f"q22_it_{i:02d}_{int(test_size*100)}"
    if not os.path.exists(iteration_folder):
        os.makedirs(iteration_folder)

    sim_filename = sim_txt.format(i, int(test_size*100))

    command = retrain_cmd.format(test_size, sim_filename)
    print(f"Executing: {command}")
#     break
    os.system(command)

    new_png_name = new_png.format(i, int(test_size*100))
    mv_cmd = f"mv {mv_png} {new_png_name}"
    print(f"Renaming: {mv_png} to {new_png_name}")
    os.system(mv_cmd)
    shutil.move(new_png_name, iteration_folder)
    shutil.move(sim_filename, iteration_folder)

    # Move all generated files to the iteration folder
    pt_files = glob.glob('retrain*.png')
    for pt_file in pt_files:
        shutil.move(pt_file, iteration_folder)
        print(f"Moved {pt_file} to {iteration_folder}")
    print(f"Listing contents of {iteration_folder}:")
    os.system(f"ls {iteration_folder}")
print("Script execution completed.")
```

Step IV: We can now very, the test_size parameter to identify the best adapter model for ‘cnn’ with base model trained on ‘rocket’:

# of training epochs = 500:   (dnMAPE=**overall** denormalized Mean Absolute Percentage Error)

test_size	dnMAPE	test_size	dnMAPE	test_size	dnMAPE	test_size	dnMAPE
0.70	0.054233	0.72	0.054717	0.75	0.063635	0.78	0.063441
0.80	0.063897	0.85	0.067470	0.90	0.092769	0.95	0.107649
0.81	0.065123	0.86	0.068780	0.91	0.096351	0.96	0.113202
0.82	0.063209	0.87	0.081960	0.92	0.090512	0.97	0.119798
0.83	0.064737	0.88	0.090249	0.93	0.102040	0.98	0.150846
0.84	0.065398	0.89	0.089487	0.94	0.103765	0.99	0.225826

Although, the dnMAPE obtained from the above runs are not so close to 0.5 but, since, we only trained for 500 epochs, we can increase the performance by increasing training rounds.

<u># of training epochs = 1000</u>: (dnMAPE=**overall** denormalized Mean Absolute Percentage Error)

test_size	dnMAPE	test_size	dnMAPE	test_size	dnMAPE	test_size	dnMAPE
0.80	0.040990	0.85	0.047046	0.90	0.052425	0.95	0.069468
0.81	0.040235	0.86	0.049247	0.91	0.054888	0.96	0.077326
0.82	0.040763	0.87	0.059218	**0.92**	**0.055621**	0.97	0.090888
0.83	0.042575	0.88	0.050205	0.93	0.060249	0.98	0.124393
0.84	0.044123	0.89	0.050399	0.94	0.064405	0.99	0.210835

<u>Results</u>:

With 1000 epochs of training, we get dnMAPE close to the required value of 0.5. Using a single hidden layer in the base-model and adapter model is sufficient for our current task of predicting area and delay based on 4 input features.

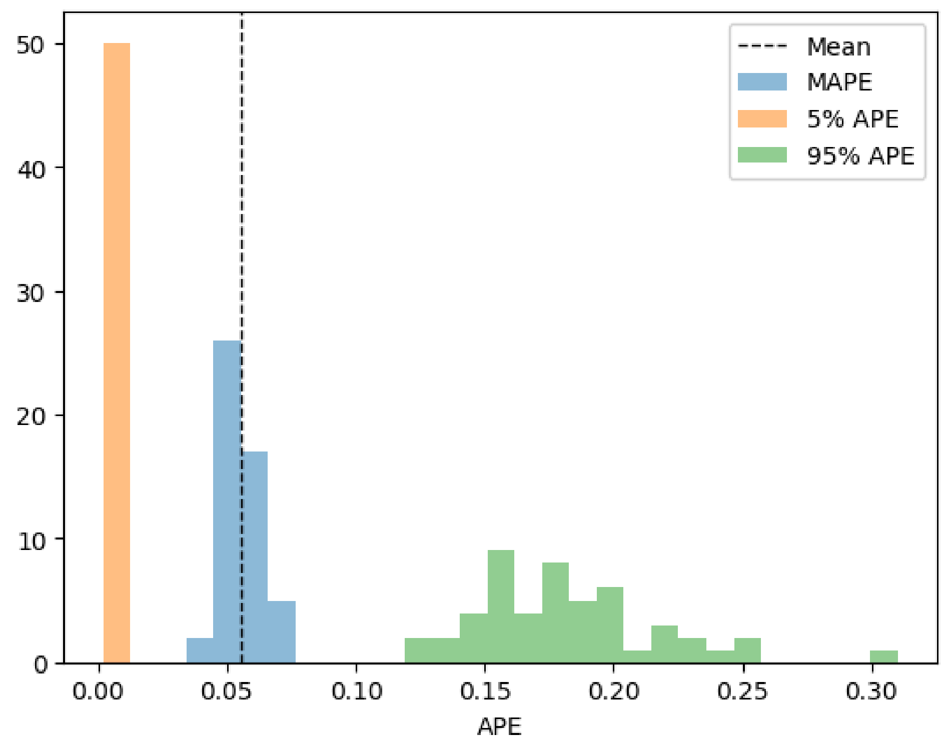

Fig. hidden-1 adapter error-summary image (500 500 0.92)

Issues:

Adding more layers increases the risk of overfitting or underfitting, as the TL/VL ratio fluctuates across designs and often deviates significantly from 1 for small datasets.

Out of the various splits in the myadapter{# of split}.pt series, **myadapter48.pt** stands out as the optimal choice with the following attributes:

TL/VL	= 0.189921	log_nMSE	= -4.007313
Confidence_5%	= 0.002399	Confidence_95%	= 0.174835
dnMAPE	**= 0.041211**	90%_var	= 0.172556
# of epochs	= 1_000	**test_size**	**= 0.92**
Sample size	= 500	design	= cnn

c. Simple scaling adapter model:

Solution:
Using the above attributes to design the simple scale and shift model:

test_size	**= 0.92;**		
base-design	**= rocket;**	**design**	**= cnn;**
# of epochs	**= 15*500;**	**# of Output layer**	**= 1;**

hw03/p2/model.py **[new] adapter model class**

```
...
#################################
# Scale and Shift adapter       #
# based on Low-Rank Adaptation  #
# := y = alpha*x + beta         #
#                               #
#################################

class ScaleandShiftAdapter(torch.nn.Module):
    def __init__(self,in_dim=4,
out_dim=1,base_layers=[128,256,512,256,64]):
        super(ScaleandShiftAdapter, self).__init__()
        self.in_dim = in_dim
        self.out_dim = out_dim
        self.alpha = torch.nn.Parameter(torch.ones(out_dim))
        self.beta = torch.nn.Parameter(torch.ones(out_dim))
        self.base_layers = base_layers
        self.base_model =
MyModel(self.in_dim,self.out_dim,self.base_layers)
        self.outLayer = torch.nn.Linear(self.out_dim, self.out_dim,
dtype=torch.double)

    def forward(self,x):
        y = self.base_model(x)
        y_ssa = self.alpha * y + self.beta
```

```
        x_out = self.outLayer(y_ssa)
        return y_ssa


    def resetWeights(self):
        self.outLayer.reset_parameters()
```

hw03/p2/retrain.py: **[new] adapter model call**

```
...
# Initialize Model
# model =
Adapter(in_dim=len(fullDS.features),out_dim=len(fullDS.targets),layers=[
64],base_layers=[64]).to(device)

model =
ScaleandShiftAdapter(in_dim=len(fullDS.features),out_dim=len(fullDS.tar
gets),base_layers=[64]).to(device)
model.resetWeights()
...
```

Report:

The scale and shift adapter offers an efficient approach for building lightweight modules that can be integrated into a pre-trained base model, enabling adaptation to new tasks without the need to fine-tune the entire model.

This technique significantly lowers parameter overhead by approximating the functionality of a linear layer using only scalar parameters.

Attributes of scale and shift adapter**:**

Overall dnMAPE mean = 0.085424
5%-95% confidence mean = 0.005793 - 0.141921
Overall mean TL/VL = 0.933781

TL/VL	= 1.11021	log_nMSE	= -3.146609
Confidence_5%	= 0.005058	Confidence_95%	= 0.126281
dnMAPE	**= 0.078064**	90%_var	= 0.130223
# of epochs	= 15_000	**test_size**	**= 0.92**
Sample size	= 500	design	= cnn

Results:

With 15000 epochs of training, we get the following error metric plot:

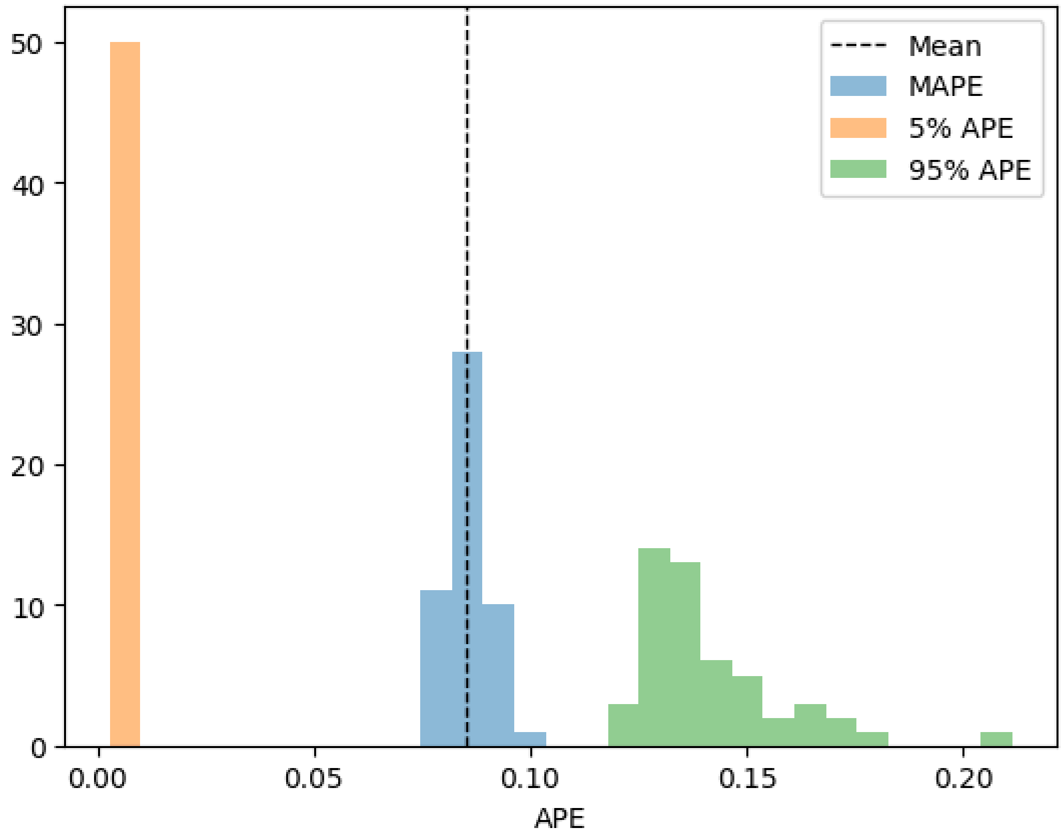

Fig. scale and shift adapter error-summary image (500 500 0.92)
For last 500 epoch training run

Comparison with part (b):

With the same number of cross-validation runs, the scale and shift model shows an error of approximately 7.8% in terms of dnMAPE, which is comparable to the hidden-1 adapter's error of around 5%. The average error for the scale and shift adapter is about 8.5%, indicating a small difference which can be handled through a base model with a larger number of layers.

Although the error percentage has increased, the confidence interval is significantly more concentrated, even after running for an order of magnitude higher epochs. This gives motivation for improved generalization characteristics.