Université de Genève

Deep Learning

14x013

DL Mini Projects

authors:

Azeem Arshad

Petter Stahle

Faysal Saber

Emails:

muhammad.arshad.2@etu.unige.ch

petter.stahle@etu.unige.ch

faysal.saber@etu.unige.ch

Gitlab project repo:

`https://gitlab.unige.ch/Petter.Stahle/deeplearningminiproject.git`

UNIVERSITÉ DE GENÈVE

FACULTÉ DES SCIENCES

Département d'informatique

Contents

Project 1: Classification, weight sharing, auxiliary losses

1 Introduction

In this project we implement and compare CNN architectures based on weight sharing and auxiliary loss. Our data-set contains pairs of images with their classes (what digit they respectively represent) and the target which we will try to predict, i.e. whether the first digit in the pair is lesser or equal to the second.

Name	**Tensor dimension**	**Type**	Content
`train_input`	$N \times 2 \times 14 \times 14$	`float32`	Images
`train_target`	N	`int64`	Class to predict $\in \{0, 1\}$
`train_classes`	$N \times 2$	`int64`	Classes of the two digits $\in \{0, \dots, 9\}$
`test_input`	$N \times 2 \times 14 \times 14$	`float32`	Images
`test_target`	N	`int64`	Class to predict $\in \{0, 1\}$
`test_classes`	$N \times 2$	`int64`	Classes of the two digits $\in \{0, \dots, 9\}$

Figure 1: overview of the datset

2 CNN Models

2.1 Basic Model

The first model uses no weight sharing. Each channel of the input (2x14x14) is passed through a different CNN as shown in the left part of figure 2 below. Next, the output is concatenated and passed through a series of fully connected layers. Finally the output $\in [0, 1]$ determines whether the first image's digit is larger or the second image's digit is larger as shown in the right part of figure 2.

Figure 2: Basic model without weight sharing or auxiliary loss. Right part follows left part.

2.2 CNN with Weight Sharing

The second model uses weight sharing, which means that each channel of the input (2x14x14) is passed through the same CNN as opposed to the first model as shown in figure 2 below. The following steps are similar to the first model, so the output is concatenated and passed through a series of fully connected layers.

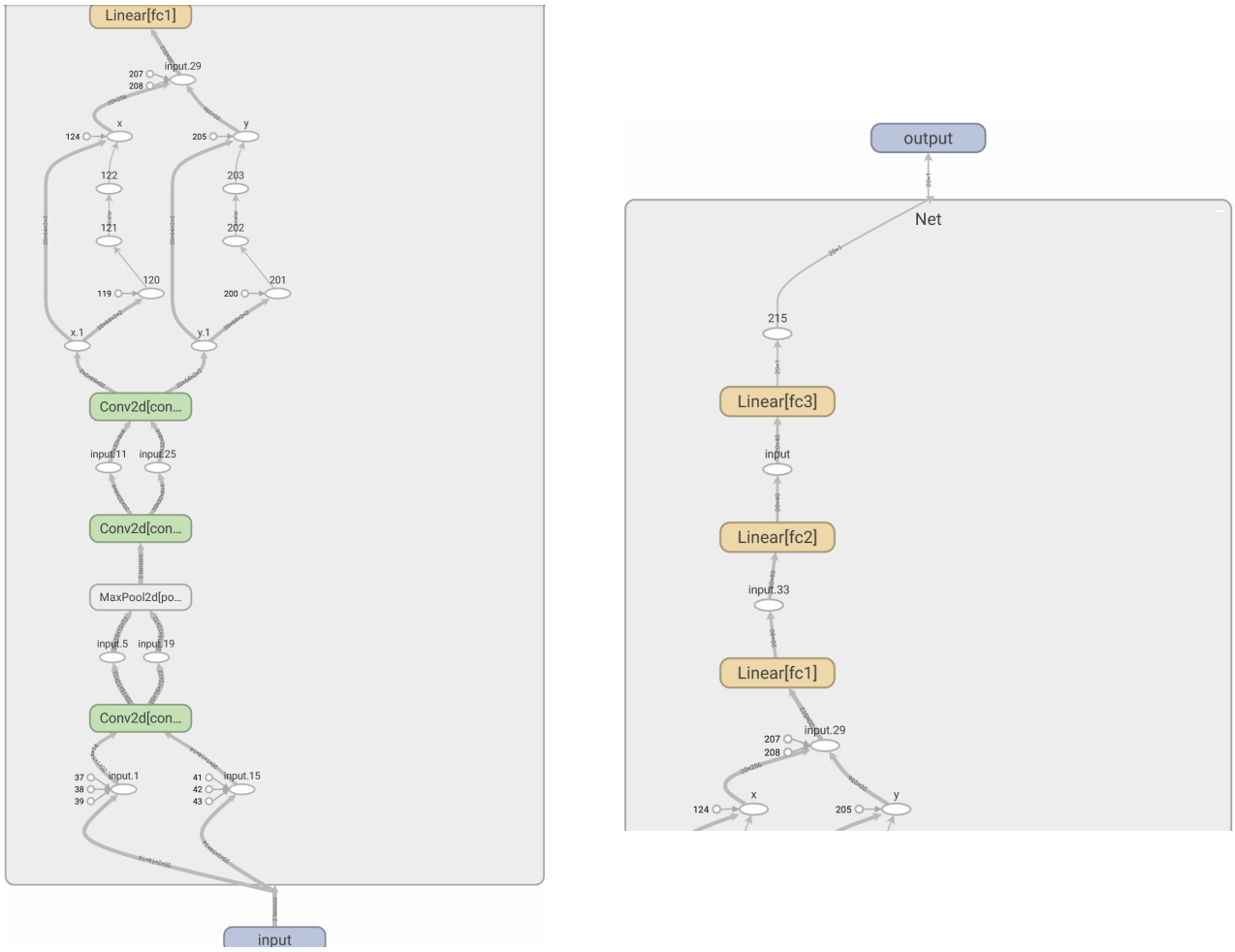

Figure 3: Weight Sharing model

2.3 CNN with auxiliary loss

Auxiliary loss takes advantage of the classes of each digit given in the dataset. As we can see in the diagram below, We start by classifying the digits by passing each tensor through three convolutional layers. We then pass them through three fully connected layers. This will return us the embedded images x and y (in the right part of figure 4 and code: **emb_x** and **emb_y**). We use the `log_softmax` over these embedded images, that we will then use to compute our auxiliary loss. In parallel, we concatenate the embedded images that we input into two new extra Linear layers before using the sigmoid function to predict the main target.

When training the model, the loss would simply be the addition of the auxiliary losses and the loss obtained with the classes and target outputs. We add a smaller weighting factor on the losses of the images and a higher factor on the loss from the main target. Better and more stable results were obtained when increasing the weighting factor of the main target loss.

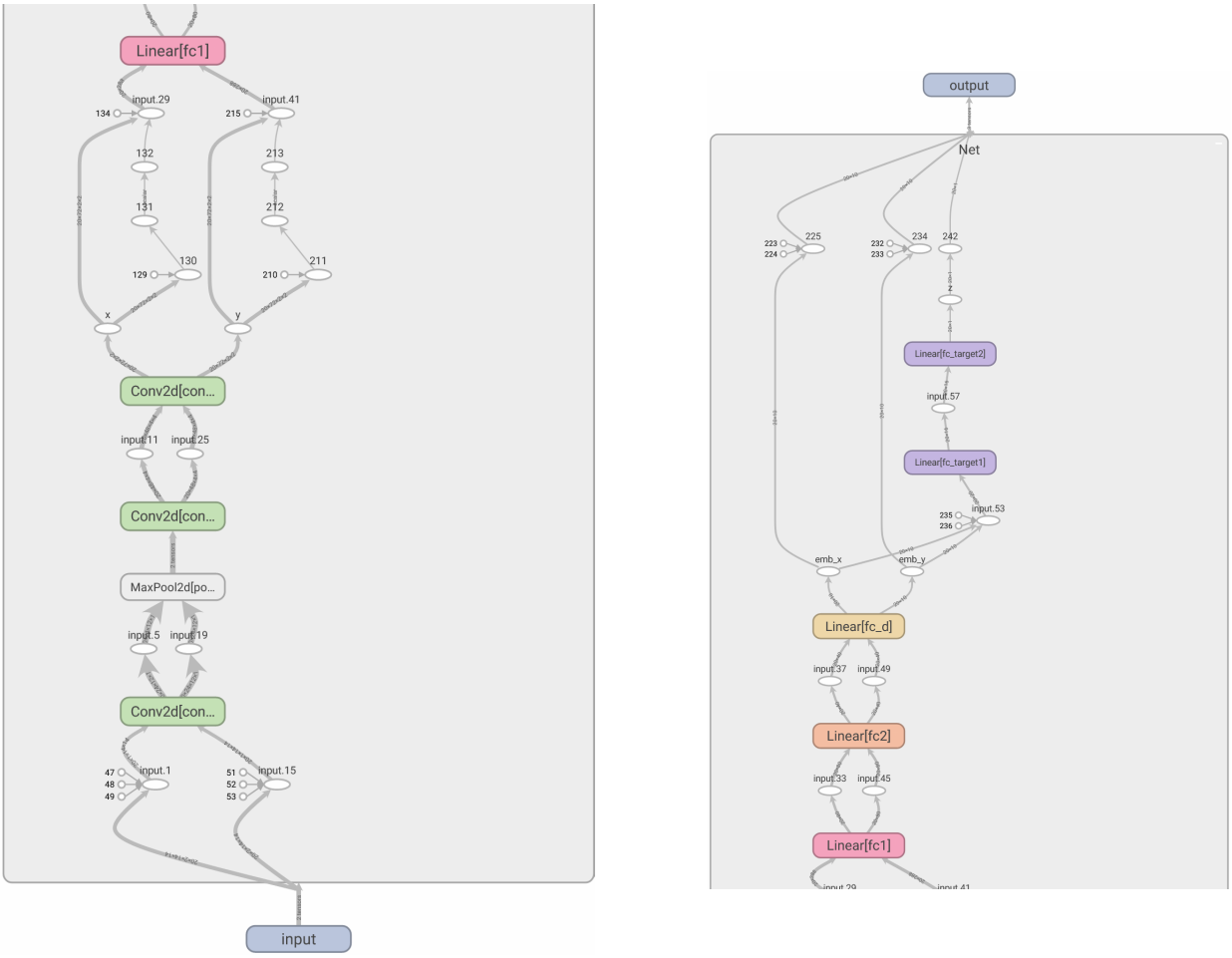

Figure 4: Auxiliary Loss model

3 Results

We run each model over 10 rounds and obtain the following accuracies over the test set with their respective mean and standard deviation. We can additionally observe the evolution of accuracies and loss over 25 epochs.

	Baseline (Mean)	**Basic model**	**Weight Sharing**	**Auxiliary loss**
Mean [%]	52.6	82.870	85.210	84.730
std	-	1.311	1.141	3.738

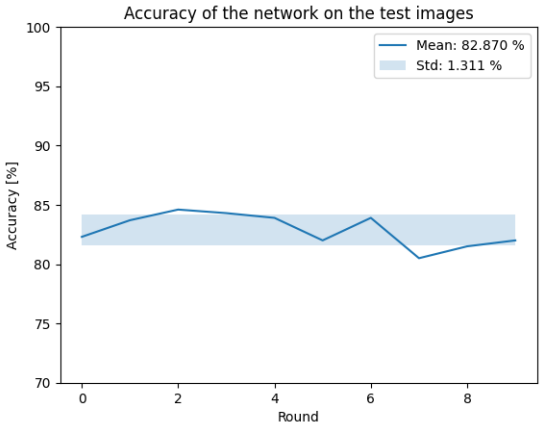

Figure 5: Basic model

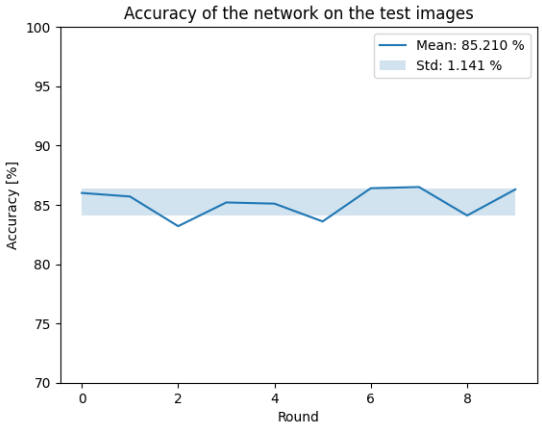

Figure 6: Weight sharing model

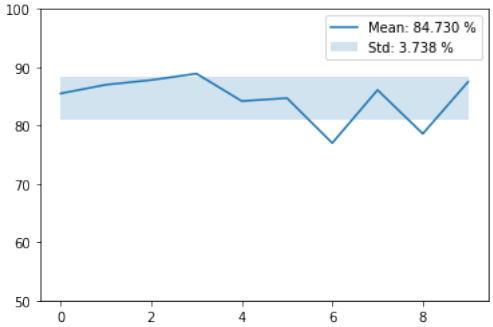

Figure 7: Auxiliary loss model

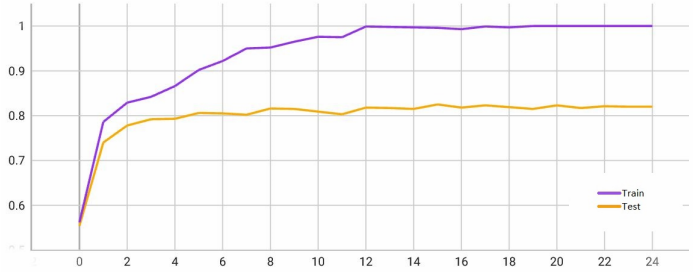

Figure 8: Basic model without weight sharing

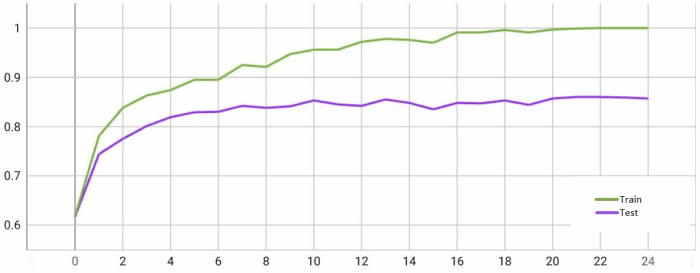

Figure 9: Weight sharing

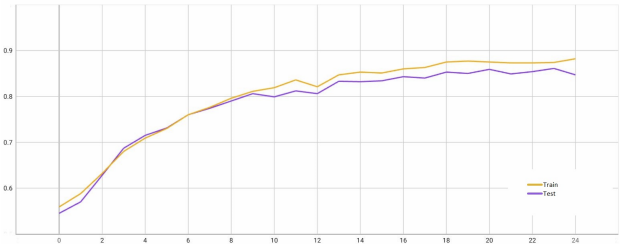

Figure 10: Auxiliary loss

Figure 11: Accuracy evolution over 25 epochs

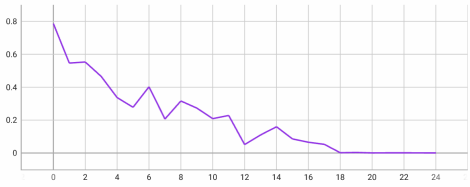

Figure 12: Basic model without weight sharing

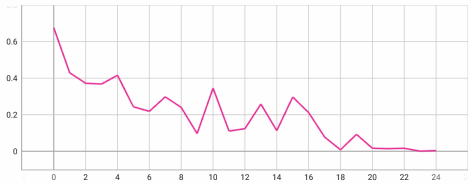

Figure 13: Weight sharing

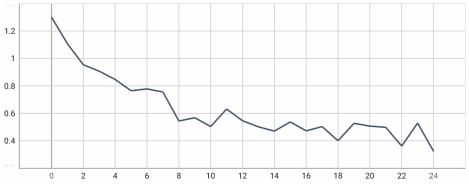

Figure 14: Auxiliary loss

Figure 15: Loss evolution over 25 epochs

3.1 Remarks

Accuracy (figures 8, 9) and loss (12, 13) stabilises after 19th epoch for both the basic model and the weight sharing model. The same applies to the Auxiliary model (figures 10, 14) to some degree since in general it is the least stable among the 3 models. It is worth mentioning Auxiliary loss (model 3) achieves a higher maximum accuracy over weight sharing (model 2) as shown in figures 6 and 7. Despite achieving higher maximum accuracy, Auxiliary Auxiliary loss model results are less stable than the other 2 models as it achieved a std of 3.47 as opposed to 1.31 and 1.14 achieved by basic model and weight sharing models respectively.

4 Conclusion

In conclusion, we are able to observe the benefits of leveraging the use of weight sharing and auxiliary loss as they resulted in a higher accuracy on average over the basic model. For future work, it might be useful to explore the usage of standard famous networks such as AlexNet.

Project 2: Mini deep-learning framework

1 Framework Structure

We implemented our own deep learning framework in the python module `Project2/Module.py`. This module contains the implementations for the following layers, each of them having their own class. Each class inherits from the base class below:

```
    class Layer(object):
    def __init__(self, n_inputs: int, n_outputs: int) -> None:
        self.n_inputs = n_inputs
        self.n_outputs = n_outputs
        self.weights = torch.randn(n_inputs, n_outputs)
        self.bias = torch.randn(n_outputs)
        # Gradient with respect to weights
        self.gradwrtw = torch.zeros(n_inputs, n_outputs)
        # Gradient with respect to bias
        self.gradwrtb = torch.zeros(n_outputs)

    def forward(self , x: torch.Tensor) -> torch.Tensor:
        raise NotImplementedError

    def backward(self , *gradwrtoutput):
        raise NotImplementedError
figfig
    def param(self):
        return []
```

- LinearLayer (Fully connected layer)

  It is a layer where all the nodes in the layer has a connection to all the nodes in the previous layer. It is essentially a matrix multiplication.

- Activation layers:

  – ReLU:
  $$f(x) = max(0, x)$$

  – Tanh:
  $$tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

  – Sigmoid:
  $$\sigma(z) = \frac{1}{1 + e^{-z}}$$

- LossMSE:

  The **loss function** we implement is the Mean Squared Error (MSE), i.e.
  $$MSE = \frac{1}{m}\sum_{i=1}^{m}(y^{(i)} - \hat{y}^{(i)})^2$$

  where m is the number of samples, $y(i)$ the ground truth label for i-th sample and $y^(i)$ the predicted label for i-th sample.

- Sequential

  This object inherits from the Layer object, but represents a Multi-Layer Perceptron which is itself a sequence of layers.

We implement a separate base class `Optimizer` which instantiates the class `SGD`. This gives us the freedom to code extra optmizers for the future.

```
class SGD(Optimizer):
    def __init__(self, model: Sequential, lr: float):
        super().__init__(model, lr)
    def step(self):
        for layer in self.model.layers:
            layer.weights -= self.lr * layer.gradwrtw.T
            layer.bias -= self.lr * layer.gradwrtb
```

where `gradwrtw` and `gradwrtb` are respectively the gradient w.r.t. the weights and bias.

2 Test

In order to test our framework, we generated a train and test dataset of 1000 2D points in the range $[0, 1]^2$ with labels 0 if outside the unit circle, and 1 if within. After this, we defined a sequential model with our framework, as shown below:

```
model = m.Sequential(
    m.LinearLayer(2, 25),
    m.ReLU(),
    m.LinearLayer(25, 25),
    m.Tanh(),
    m.LinearLayer(25, 25),
    m.Tanh(),
    m.LinearLayer(25, 25),
    m.Tanh(),
    m.LinearLayer(25, 1),
    m.Sigmoid())
```

Following this, we trained it over 100 epochs, with batches of size 100, with a learning rate of 0.01. This got us an accuracy over the test set of 96.4%. The resulting predictions are plotted in the image below in fig. 16.

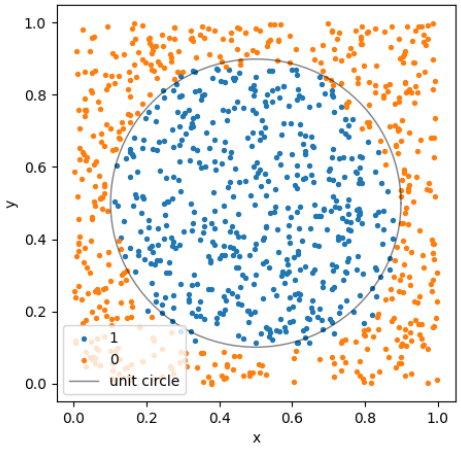

Figure 16: Predictions of test model.

References

[1] Iddo Drori. *The Science of Deep Learning*. Cambridge University Press, 2022. http://www.dlbook.org.