Classification, weight sharing, auxiliary losses *Deep Learning*

Karel Peeters, Lavinia Schlyter
Department of Computer Science, EPF Lausanne, Switzerland

I. Introduction

In this project we compare different neural network architectures, loss functions and training techniques on a simple computer vision problem based on the MNIST dataset. Specifically we study the effect of weight sharing and an auxiliary loss function.

The model receives the grayscale images of two MNIST digits as input and predicts if the first digit is less than or equal to the second. The input images are rescaled from the original 28×28 to 14×14 to save some computation time.

Both the train and test set contain 1000 pairs of images, the boolean target output and also which digits the images actually represent. The latter can be used for the auxiliary loss function discussed later. The code provided to us, `generate_pair_sets`, keeps digits from the train and test set separate, so there is no risk of leaking training images into the test data.

We first discuss the different techniques used, then we present our architectures and experiments and conclude on the results and further improvements.

II. Techniques

In this section we discuss the different network architectures and training techniques that are explored.

We denote a network f that takes in images a and b as tensors of shape 14×14 and returns a single output value $\hat{z}$ as:

$$\hat{z} = f_\theta(a, b)$$

Networks are designed so the output $\hat{z}$ is a single value between 0 and 1 representing the predicted probability that the correct label z is 1. To ensure that $\hat{z} \in (0, 1)$ the last layer of all networks is a Sigmoid activation function.

A. *Loss functions*

We try two different commonly used loss functions for classifications tasks: Mean Squared Error (MSE) and Binary Cross Entropy (BCE):

$$L_{z,MSE} = (\hat{z} - z)^2$$
$$L_{z,BCE} = z \log(\hat{z}) + (1 - z) \log(1 - \hat{z})$$

B. *Batch normalization*

A technique commonly used to speed up training is introducing Batch Normalization (BN) layers into a network. A BN layer normalizes the distribution of values as they pass trough the layer, with the target output mean and variance as learnable parameters. Initially it was thought that this helps reducing internal covariance shift [1], but later research found that instead the loss functions become smoother [2].

C. *Dense network*

We start with a simple fully connected neural network. This is not a good architecture for image recognition, but it serves as a baseline.

The input is both images flattened and concatenated, which yields a vector of size $2 * 14 * 14 = 392$. The network consists of a few fully connected layers each followed by a *ReLU* activation function. The output layer has size 1 and a Sigmoid activation function.

D. *Convolutional neural networks*

For deep neural networks that have images or other translation-invariant data as input convolutional layers are commonly used. A fixed size kernel slides over the input image and produces output values. The main advantages are that this architecture builds some translational invariance into the model itself, and that less weights are needed for a layer with a given input and output size.

E. *Residual connections*

One variant of a convolutional network is the ResNet [3] architecture. This architecture uses *resisual connections*, where a part of the network $y = h(x)$ is instead replaced by $y = h'(x) + x$. The hypothesis is that this new h' is easier to learn than the previous h.

The ResNet architecture uses repeated Residual blocks, which consist of a convolutional layer, followed by batch normalization and a ReLU activation function, all repeated twice. A skip connection is added which adds the input back into the network right before the final ReLU activation. All convolutional layers have kernel size 3 and pad their inputs so the output has the same shape as the input, as required by the elementwise addition.

F. *Weight sharing*

The basic model takes in both images as a single tensor. This is a missed opportunity, we know that both images are somewhat interchangeable and should have the same processing applied to them. Thus instead of a single network $f(a, b)$ that takes in both images we first use a shared network g that preprocesses both images. The final computation then looks like $f'(g(a), g(b))$, where f' is a small final network that combines the representation of the images generated by g.

This has many advantages: we encore some domain knowledge into the network architecture, the g part of the network sees twice as many inputs during training, and the final model can be smaller for the same level of expression.

G. *Auxiliary loss function*

The training data contains some extra information about the digits that is currently not being used: the correct digit labels. To use this information we add an auxiliary loss to our existing loss

Model	Epochs	train_acc	test_acc
Dense BCE	20	1.000 ± 0.000	0.782 ± 0.018
Dense MSE	20	0.993 ± 0.005	0.772 ± 0.017
Conv	80	0.888 ± 0.044	0.741 ± 0.024
Conv + BN	80	0.992 ± 0.006	0.775 ± 0.016
Conv + Augment	80	0.811 ± 0.038	0.743 ± 0.016
Duplicated	20	0.922 ± 0.011	0.794 ± 0.017
Shared	20	0.904 ± 0.011	0.823 ± 0.020
Aux $a = 0.1$	70	0.988 ± 0.005	0.881 ± 0.014
Aux $a = 10$	70	0.988 ± 0.005	0.962 ± 0.008
Residual	120	0.971 ± 0.034	0.945 ± 0.037
Resless	40	0.981 ± 0.019	0.956 ± 0.023

TABLE I: Summary of experiment results

calculation. This additional loss function behaves as a regularizer [4] by forcing the network to also learn a second task. Contrary to other types of regularization this secondary task is not really in conflict with the primary task, recognizing digits is still useful when you want to decide which one is bigger.

The network is changed to output a prediction for the digits in addition to a prediction for the boolean value of interest:

$$(\hat{z}, \hat{d_a}, \hat{d_b}) = f(a, b)$$

$\hat{d_a}$ and $\hat{d_b}$ are vectors of size 10 that at index i contain the predicted probability that the digit is i. These digit predictions are added to the loss function using cross entropy:

$$L = L_z(\hat{z}, z) + a(d_a \cdot log(\hat{d_a}) + d_b \cdot log(\hat{d_b}))$$

where L_z is the loss function used for the boolean output, and a is the auxiliary loss weight: an extra hyperparameter to control how important correctly predicting the digits is in the loss.

H. *Data augmentation*

The goal of data augmentation is to increase the size of the train data set using some domain specific knowledge.

We considered two different techniques. The first is to swap both input images and change the target output accordingly. The second is to randomly disrupt the input images with small rotations, translations or shears.

III. Experiments

In this section our experimental results are discussed. Table I contains a summary of different models and training setups tried.

All models are trained for a predetermined number of epochs, set after some manual experimenting to find the minimal number of epochs after which the test accuracy stops improving. All training uses minibatch sizes of 100. The optimizer used is Adam [5] with the default parameters.

The reported accuracies are always for the final epoch, computed with the model in evaluation, not training mode. The difference is that some layers like batch normalization or dropout behave differently in the different modes. Reporting train accuracy with the model in training mode would be faster, because you get it for free during training, but can under-represent the actual overfitting that has happened.

We should have made a separate validation set based on a separate part of the MNIST images at the start of the project, and only realized this mistake near the end. Still, results in this table are always based on a separate final run with newly generated test data so it's still hard to overfit hyperparameters on the test set.

We always report the mean and standard deviation of 10 independent training runs with randomly initialized initial model parameters and randomly generated train and test datasets. Plots show the mean value in a as a solid line, and the mean value plus or minus one standard deviation as a dashed line in the same color.

A. *Dense models*

These dense models serve as a simple baseline. We use a network with 2 hidden layers, one of size 64 and one of size 32, connected via fully connected layers and followed by ReLU activation functions.

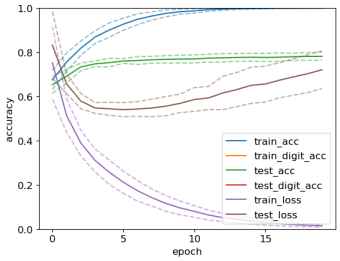

Fig. 1: *Dense BCE* training

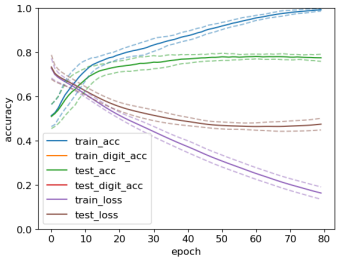

Fig. 2: *Conv + BN* training

The training process of this simple network with loss function BCE is shown in Fig. 1. Without any regularization, this model overfits very quickly on the train set, the test loss also starts going up, although performance on the test set does not start to go down by continuing training further. The test accuracy stabilizes around 78%. We tried weight decay and additional dropout layers as regularization techniques, both cause overfitting to happen later but neither increases test accuracy by more than 1%. Similarly, training a model with smaller hidden layers delays overfitting but doesn't raise test accuracy.

Using a different loss function MSE again results in test accuracies within 1% of the previous result. This observation is also true for all future models, so we don't repeat it there. For the remainder of the experiments we use BCE as the loss function.

B. *Convolutional model*

Next we try a convolutional model. Both input images are stacked to for a tensor of shape $2 \times 14 \times 14$ for the input to the model. The model chosen starts with two convolutional layers with kernel size 3 and 16 output channels, each followed by max pooling with kernel size 2. This output is processed by two fully connected layers, the first with size 32 and a ReLU activation and the final one with output size 1 and a Sigmoid activation.

The convolutions don't do any padding and don't have activation functions. We found that adding ReLU activations did not improve performance, max pooling already adds non-linearity.

The basic *Conv* model reaches a test accuracy of 74.5%. For this network we found that adding batch normalization layers after each convolutional and linear layer speeds up training a lot and even improves the test accuracy to 79.0%. Training for this model is shown in Figure 2.

Again we tried dropout and weight regularization, but they only slowed down training and did not improve test accuracy.

This experiment shows that convolutional networks are indeed a big improvement over dense networks for image processing tasks.

C. *Data augmentation*

If we train the previous network with data augmentation, specifically the digit flipping discussed before, we get lower train accuracy but the same test accuracy, the same behaviour as the regularization techniques discussed before. This shows that the performance or the model is not limited by the amount of training data, but by the model architecture itself. We found the same is true for all other architectures tried.

D. *Weight sharing*

As discussed before, we are now building networks of the form $\hat{z} = f(g(a), g(b))$, where for g we use the convolutional network from the previous model with batch normalization enabled, and tweaked so the input shape is now $1\times14\times14$ and the output size is 10. Finally f is the output model, which takes as input the output of two invocations of g stacked. We use a simple fully connected network with one hidden layer of size 20 and a ReLU activation, followed by another dense layer with output size 1 and a Sigmoid activation. The output size of g can be chosen arbitrarily, here we pick 1 for consistency with the following auxiliary loss function. Increasing this size did not have a noticeable effect on the accuracy.

Because this is a different network structure to compare the effect of weight sharing itself, we train two networks: $f(g_1(a), g_2(b)$ and $f(g(a), g(b))$. The former uses two separate instantiations of the g network with separately randomly initialized weights, we call this the duplicated network.

These models suffer a lot more from overfitting than the previous ones, here the test accuracy actually starts to go down without any regularization. For this reason we added a dropout layer with $p = 0.1$ after each convolutional layer and a dropout layer with $p = 0.5$ after each linear layer. This is enough to prevent too much training from lowering the test accuracy.

Performance of both variants is listed in Table I. The Duplicated network matches the performance of the previous convolutional network, the Shared network is either the same or slightly better. Both networks need only around 15 epochs to achieve their best performance, a lot less compared to previous networks. This makes sense since these networks start with the knowledge that both digits are independent build into their structure.

E. *Auxiliary loss function*

In this experiment we train the previous weight sharing network, including dropout, but now with the auxiliary loss added to the existing BCE loss function. Because now the outputs of the shared network g are meaningful, we can additionally plot the digit recognition accuracy.

There is an extra hyperparameter to choose here, the auxiliary loss weight a. Training for two different values, $a = 0.1$ and $a = 10$ are shown in figures 3 and 4 respectively.

This parameter turns out to be very important, values $a \geq 1$ work very well, while for lower values training appears to get stuck in an overfit local minimum, where the z accuracy is 87.0% but the digit accuracy is only 50.9%. Intuitively the network overfits on the boolean output and there is not incentive to properly identify the digits. Increasing a to for example 10 solves this issue and gets final accuracies of 96.7%.

We can conclude that this addition to the loss function is very helpful and results in models with very good test accuracies.

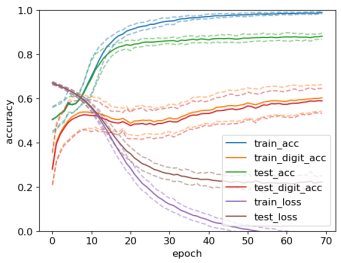

Fig. 3: Auxiliary training with $a = 0.1$

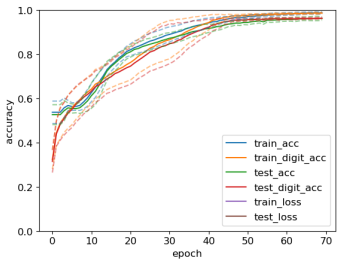

Fig. 4: Auxiliary training with $a = 10$

F. *Residual model*

Finally we try a model containing residual blocks. We replace the convolution network in the previous weight-sharing model with the following network:

- An initial convolutional layer with kernel size 3 and 32 output channels, followed by a ReLU activation.
- Two residual blocks with 32 channels.
- A max pooling with kernel size 2.
- Two residual blocks with 32 channels.
- A linear layer with flattened input of size $32 \times 7 \times 7$ and output 50 followed by a ReLU activation.
- A Dropout regularization layer.
- A linear layer with output size 10 and Softmax activation.

This *Residual* model was trained with the auxiliary loss function with weight $a = 10$. Training is very slow, converging to a test accuracy of 97.1% only after around 120 epochs. This is within margin of error of performance of the previous convolutional models.

We also tried disabling the residual connections while otherwise keeping the same architecture, we call this the *Resless* model. This version trains a lot faster, it only takes around 40 epochs to reach similar accuracies as the *Residual* model.

This is a surprising result, seemingly contradicting the result presented in [3]. We hypothesize that adding residual connections is only advantageous for very deep models, while the model used for these experiments only has 11 layers with trainable parameters.

IV. Conclusion

We tested different architectures and loss functions applied to a simple computer vision challenge. We have shown that convolutional networks trained with auxiliary losses and weight sharing perform very well, getting around 97% accuracy on test data. In particular the auxiliary loss is responsible for a big jump in accuracy if tuned correctly. Weight sharing results in a less significant improvement and mostly helps to build smaller models.

V. *

References

[1] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

[2] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How does batch normalization help optimization?, 2019.

[3] Francesco Visin, Kyle Kastner, Kyunghyun Cho, Matteo Matteucci, Aaron Courville, and Yoshua Bengio. Renet: A recurrent neural network based alternative to convolutional networks, 2015.

[4] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, 2015.

[5] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.