# Hyper-parameters and training basics with PyTorch

Megi Dervishi
megi.dervishi@ens.fr

## 1. Optimal Network design

### 1.1. Multiclass classification (Pb1)

The best performing architecture is shown in Figure 1 (a). The model consists of three 2D-convolution layers of kernel $(3 \times 3)$ with no padding, two 2D maxpooling layers with kernel $(2 \times 2)$, dropout layers for regularization, two fully connected layers and ReLU activation. In the end the model has a classification layer which is 1D-batch normalized to map to the final output.

The training set is composed by 6000 images, the validation set by 1291 images and test set by 2007 images. In order to avoid overfitting, I perform data augmentation by applying transformation such as resizing, random rotation up to $45^o$, random shearing with probability 0.4. The model is trained for 100 epochs with cross-entropy loss and optimized with an Adam optimizer with learning rate 0.001. The optimal batch size is 50. With the following hyper-paramters the model reaches an accuracy of **99.15%** on the validation set and **97.66%** on the unseen testing set.

### 1.2. Regression (Pb2)

The best performing architecture is shown in Figure 1 (b). It consists of 7 fully connected layers with hidden size 20, 10 and 6. Each layer is followed by a ReLU activation function. At first I normalize the data before splitting it into a train, validation and test set. The model is trained for 120 epochs with a Gaussian negative log-likelihood loss. It is also optimized with an Adam optimizer with a learning rate 0.01. The batch size is 40.
With the above hyperparameters, the loss of the model converges, reaching **-2.2859** on the validation set and **-2.2308** on the testing set.

Furthermore as seen in Figure 2 when plotting the variance predicted by the network with respect to the "Overall Quality" feature we notice that as the Overall Quality increases the variance decreases. This shows that the model is able to easily predict the price of a good quality house compared to an average quality house. This makes sense since usually good/excellent quality houses have approximately the same high price, however for average quality houses the price can fluctuate easily as it can be affected by other factors. Hence the model is less confident in predicting such houses (the variance is higher).

## 2. Hyper-parameter tuning and Discussion

Tuning the hyper-parameters is a slow process, however neccessary to get the best performance possible out of your model. As observed, the naive classification model (2 fc layers) had an accuracy of 30.98% when trained for 10 epochs with an MSE loss, SGD optimizer and 0.01 learning rate. However by simply increasing the learning rate to 0.1, the accuracy improved tremendously reaching 93.34% on the validation set.

The first step in creating the model is thinking about the problem at hand and understanding the balance between width and depth of layers. In particular for the classification problem, the use of convolution layers is important as we are dealing with images and a wider range of parameters can potentially increase the expressiveness power of the model. However there is always the risk of overfitting. Therefore it is important to "control" the number of parameters by regularization techniques. This is done using maxpooling2D layers, dropouts and batch normalization.
For the convolution layer, I have chosen a kernel of size $(3 \times 3)$. This allows me to have 3 such layers whereas a bigger sized kernel would lead to less layers. The $(3 \times 3)$ kernel reduces the image size up until 8 by 8 (see Figure 1). Obviously we can add more convolution layers however that would be counterproductive since the image is too small. On the other hand, regression (especially with the use of Gaussian negative log-likelihood loss) requires more depth than width in order to correctly estimate the variance.

At first when my model was more shallow the best performing activation functions were Sigmoid and Tanh. However as depth increased and the models became more complex ReLU was more robust to changes unlike the other two.

The more complex a model is the more you need to train it. Therefore the number of epochs changed from 10 for the naive models to approximately 100-120 epochs. However

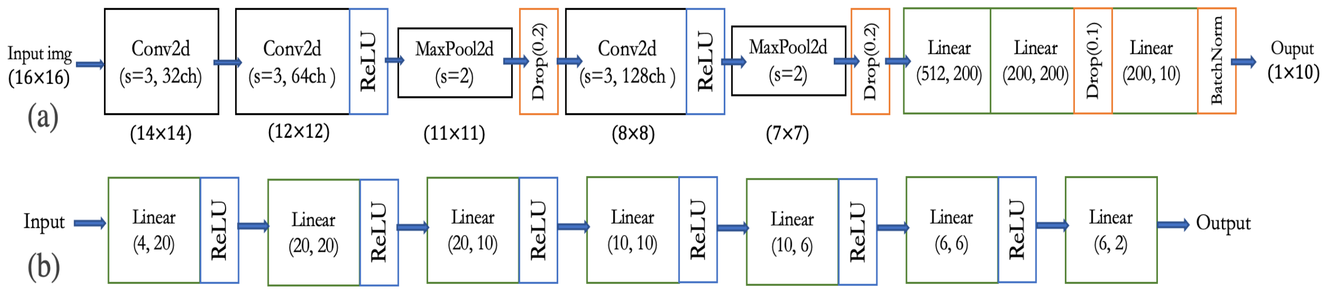

Figure 1. (a) classification model architecture, (b) regression model architecture.

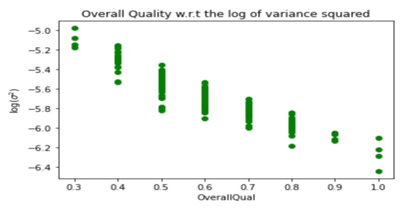

Figure 2. The evolution of the log variance squared for the regression problem.

that doesn't mean we have to over-shoot in training time as that leads to over-fitting. One way to avoid this is by implementing early-stopping.

Learning rate is responsible for the step size in the direction of gradient descent. If it is too small then the model will take a longer time to converge and would be more prone to getting stuck in a local minima. However if the learning rate is too big then the model may never reach the global minima since the learning is more unstable and can simply overshoot the minima. Another hyper-parameter that can affect the learning process is the batch size. Indeed the bigger the batch size the bigger the variance of the magnitude of the gradient and the smaller the batch size the more time it takes for the model to converge. It is important to have a balance between the learning rate and the batch size. In the experiments it was noticed that when the batch size increased i.e. increasing the variance of the magnitude of the gradient, then a better accuracy was obtained when the learning rate also increased. Increasing the learning rate ”cancels” the negative behaviour arising from the greater variance in a batch. On the contrary, reducing the batch size would increase the variance of the direction of the gradient making the learning process more noisy. And hence reducing the learning rate helps stabilizing the fluctuations of the learning process.

Furthermore a model has a different behaviour for different optimizers. In addition different optimizers act differently according to different learning rates as we can see in Figure 3. We notice that even though the optimal optimizer for the classification problem is Adam, Adam with learning rate 1 performs much worse than Adam with learning rate 0.001 assuming the same batch size and training time. We further notice that in both regression and classification the SGD optimizer has a worse performance compared to RMSProp or Adam optimizer. That could be because it would require more epochs to train and perhaps needs more ”detailed” fine-tuning i.e. better learning rates. However even though SGD could potentially generalize better, the amount of time to fine-tune the hyperparameters and train a model with an SGD optimizer would not justify the gain in accuracy. Therefore the RMSProp and Adam would be better suited; and more specifically the Adam optimizer.

Finally for classification (resp. for regression) the Cross-Entropy loss (resp. the Gaussian negative log-likelihood loss) performed better than the MSE loss. The cross-entropy loss penalizes misclassification with an exponential increase in the loss whereas the MSE loss punishes with the square. Therefore if the model missclassifies slightly a ”turtle” as a ”table”, it should be penalized heavily which is achieved by the cross-entropy loss. Hence the model learns not only to classify correctly but also to not missclassify and this leads to a better understanding i.e. better accuracy. On the other hand for a regression problem, it makes sense that the Gaussian negative log-likelihood loss (GNLL loss) has more generalization power than the MSE loss, since the MSE loss is a subcase of the GNLL loss ($\sigma = 1$). It is slightly harder to train the GNLL loss so it is important to have a deeper model and more careful hyper-parameter tuning. However this results in a better performing model.

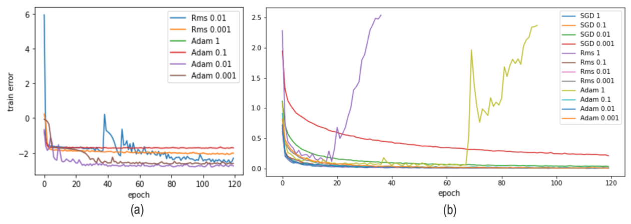

Figure 3. The evolution of training error varying optimizers,lr for (a) regression problem (b) classification problem.