Deep Learning in Practice, Practical Session 1: Hyper-parameters and training basics with PyTorch

Maxence Gélard
maxence.gelard@student-cs.fr

1. Architectures details of the best models

1.1. Classification problem

For the classification problem, aiming at recognizing handwritten digits, the model that got the best performance is a convolutional network with the following architecture:

- **3 convolutional layers**, using kernels of size 5 and stride 1 (no padding). The number of output channels for each layer was respectively 16, 32 and 64. Each of these layer were followed by a *ReLU* activation and a *MaxPooling* layer (with kernel of size 2).
- **1 dense layer** with 256 input units and 10 (= number of classes) output units.

This network has been trained using the ***cross-entropy*** loss (so no *Softmax* is used in the network as *PyTorch* directly deals with the logits for the cross entropy), which has been optimized with ***Adam*** with a ***learning rate*** of 0.01 (and I kept the default values for the other parameters : $\beta_1 = 0.9$, $\beta_2 = 0.99$ and $\epsilon = 10^{-8}$). I have been using a batch size of 1000 and I have been training the network for 100 epochs.

Performance of the model on the USPS dataset (accuracy): Train accuracy: 99.98% - Validation accuracy: 98.45% - Test accuracy: 97.1%.

1.2. Regression problem

For the regression problem, aiming at predicting selling prices from a set of 4 features per data point, the best model that I managed to get is a dense model (only *Linear* layers) with the following hyperparameters:

- **Number of neurons per layers**: 200 neurons.
- **Layers architectures**: 1 input layer and **5 hidden layers**.
- **Activations**: *ReLu* functions.

This network has been trained using the ***Mean Squared error function***, and optimized using the ***Adam*** optimizer with the ***learning rate*** set to 0.01. Notice that in order to avoid numerical overflows, the features have been scaled ($X_i = \frac{X_i - X_{i,min}}{X_{i,max} - X_{i,min}}$ for all features i), and I also scaled the target ($y = \frac{y}{y_{max} - y_{min}}$, no shift with y_{min} seemed to be necessary). The model has been trained for 100 epochs, using a batch size of 100.

Performance of the model on the Houses Prices dataset (Mean Squared Error): Train MSE: 0.0028 - Validation MSE: 0.0033 - **Test MSE: 0.0030**

2. Hyperparameters tuning discussion

2.1. Time / Performance trade-offs

For the regression problem, even if the models has 5 hidden layers with 200 neurons each, it remain globally a pretty shallow network, and only takes 5 *seconds* to be trained. For the classification task, the best model being convolutional, and takes 6 minutes to be trained (it is expected to takes more time to be trained that a fully connected network). The trainings have been done on my local machine with a *2,3 GHz Intel Core i5* processor. Given the performance of the two models these training time looks reasonable. However, achieving this performance required a more time-consuming hyperparameters tuning that took around 1 day of computing resources, to which I added an analysis time of the results ($\geq$ 1 day cumulative). I did this tuning in two steps, first a manual tuning to understand the effect of each parameters (discussed in the second part), and then a more automated ("grid-search like") search has been implemented both to find the best models as well as to dispose of different training configurations to analyze for deeply the influence of the parameters. Nevertheless, this tuning was worth it as for example, on the classification task, I succeeded in increasing the test accuracy of more than 8% compared to the initial model that was provided.

2.2. Impact of relevant parameters

Below are given some of the analysis that I have been doing in order to understand the influence of hyperparameters.

• Architecture: deeper / more complex provide better expressivity?

I started by analyzing the impact of the choice of architecture. The first thing that I observed is that a more complex model (more layers, more neurons) allows the model to get more expressivity and thus maximizes train accuracy. However, too deep networks also lead to a bigger gap between train and validation accuracy ($\geq$ additional 2-3% when adding 5 more layers to the regression network with MSE loss), so in this case regularization (like *Dropout*), but the data being not too complex, this wasn't necessary here to get correct performance. Below (Figure 1) is given a plot to provide more details on the architecture analysis.

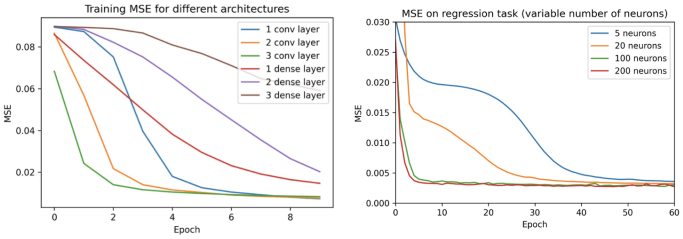

Figure 1. Influence of architecture on training loss

On the left hand side (classification task), I compared the impact of train loss with the number and types (dense / convolutionnal) of layers. First, we notice that convolutionnal layers quickly perform better on the train set, which is understandable as they carry more information about the data (pixels localisation...), and

adding more layers even more decreases the loss. However, for the dense layers, adding more of them lead to worse training loss, and more epochs are required to train them. On the right hand side (regression problem), we get another hint about the gain in expressivity when adding more neurons. In both cases, using a more complex architecture, though providing better results, also increases the training time.

• **Optimization parameters**: Afterwards, I investigated the impact of different optimization parameters on the training process and performance. Below are given 4 plots drawn from this analysis (first two look at the influence of learning rate on the training of dense networks - respectively 1 and 5 layers - 100 neurons for regression - same for the third one but for a convolutional network for classification - while fourth analyzes the influence of the optimizer):

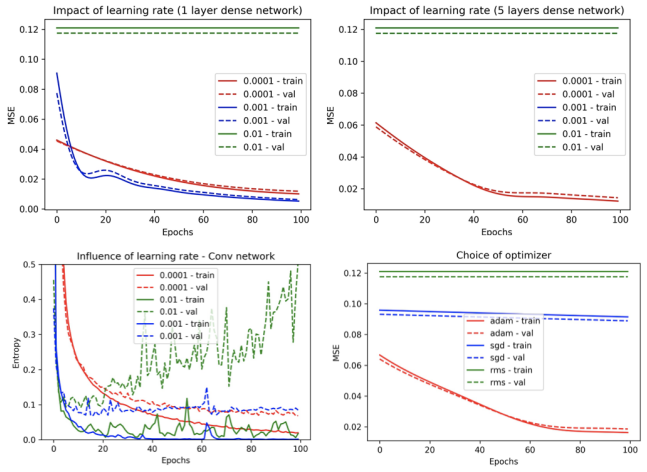

Figure 2. Influence of optimization parameters on training process

One can notice that for the dense network (in both cases) setting a too high learning rate (0.01) leads to a model that even didn't learn anything / diverged. For smaller learning rate, we get different behaviors depending on the considered architectures. For the shallower architecture (1 layer), the two others learning rate (0.001 and 0.0001) provide similar convergence properties, even if the larger one (0.001) shows some overshoot at 20 epochs (can be due to the overshoot in gradient descent that we observed for high learning rate). For the deeper network (5 layers), we have a pretty correct convergence with 0.0001, but we get a divergence (blue and green curved overlap) with 0.001, possibly due to the fact that a more complex model leads to sharper loss function, so a higher learning rate can lead to some divergence. The same analysis has been carried out for the classification problem (third plot) and leads to similar conclusions, with the extension, that we see a more noticeable divergence of the validation loss for high learning rates, while keeping a convergence on the training loss. Finally, I looked at the influence of the choice of the optimizer (here for the regression case - using a learning rate of 0.0001), and we do observe that Adam provides a better convergence of the training loss. Another considerable advantage of Adam (compared to classical SGD), not represented here, is that it allowed to get better performance (lower training loss for example), and also provided smoother training (thanks to the momentum on the gradients used in the optimization) thus making it easier to train deeper models.

I also analyzed the influence of batch size: with small batch sizes (typically 10), one gets a very noisy training, with a predominance of stochasticity. On the other hand, while increasing batch size, one get a smoother learning curve, but sometimes at the cost of a worse performance, especially in terms of generalisation (for a fixed number of epochs). That could be explained by some cancellation in the gradients of a given large batch. I noticed that increasing the learning rate while increasing the batch size tends to provide better results, which might confirm the fact that the problem was some gradient cancellation. However, one of the main advantage of using larger batch size is the gain is speed, with for example on the regression problem, I observed a reduction factor around 8 by passing from batch size 10 to 100 (the relative time is not exactly linear).

• **Role of the loss - MSE / Entropy and MSE / Gaussian Loss**:

Starting with the classification task, I started with training my different models using the Mean Squared Error loss. However, for a classification problem, let's say a binary classification problem, we have the following statistical model: $y_i \sim \mathcal{B}(\sigma(x_i^T w))$ with $\mathcal{B}$ the Bernoulli law. A maximum likelihood estimator will lead to the minimisation of the binary **cross-entropy** and not the MSE (and this can be extended to our multi-class case). So it is with no surprise that we get better performance with this loss (even if the difference are tiny): 96% (MSE) vs. 97.1% (Entropy) for test accuracy.

Concerning the regression problem, I have made several observation concerning the **Gaussian Loss**. Firstly, in general, the Gaussian Loss is always a bit higher than the MSE, which sounds plausible as it needs to find a trade-off between the parts in $log(\sigma_i^2)$ and $\frac{1}{2\sigma_i^2}(y_i - \mu_i)^2$. Moreover, when the training was more sensible to hyperparameters when using the Gaussian Loss and a smaller learning rate had to be used (typically 0.001 instead of 0.01). This is namely due to the fact that it is harder to predict well the variance instead of only giving the point-wise prediction. I also noticed that I had to use at least 5 layers to be able to predict a variance which was accurate (and avoid pathological case where it was equal to 1 everywhere for example). Nevertheless, even if my best model was the one trained with MSE, I got very close results with this loss, keeping the same architecture (Test MSE: 0.0030 when trained with MSE / 0.0034 when trained with Gaussian Loss). Additionally, below is given the plot of the variance of the prediction, with respect to the feature *GrLiveArea*:

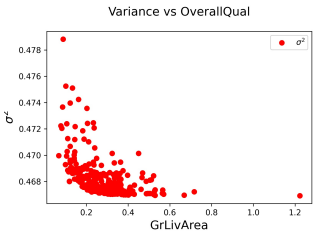

Figure 3. Variance of the prediction vs. feature *GrLiveArea*

One can observe that the variance is a decreasing function of this feature, which is explainable by the fact that high values of *GrLiveArea* occurs less so are more discrimnative, hence a better confidence of the network at those data points.