Homework 1 - Supervised Deep Learning

Simionato Giuseppe (2029013)

Academic Year 2021-2022

1 REGRESSION

1.1 Introduction

In this section is presented the implementation and test of a simple neural network for solving a supervised problem: a regression task. The goal is to train a neural network to approximate an unknown function:

$$f : \mathbb{R} \to \mathbb{R}$$
$$x \to y = f(x)$$
$$\text{network}(x) \approx f(x)$$

As training point, there are only noisy measures from the target function: $\hat{y} = f(x) + noise$.
The work of this first task follows these steps:

- Analysis and **pre-processing** of the dataset
- Definition of the **model** and of the useful functions for the training
- First **training and testing** of the model
- Analysis of the trained network
- **Hyper-parameters** optimization both using manual and automatic implementation
- **K-fold** cross validation with the best parameters

Regularization methods

Some regularization methods have been used in the code and they will be discussed in the following:

- batch normalization and dropout in the layers
- adam optimizer
- **sparsity** (L1 regularizer), this method is implemented thanks to a function defined by scratch which compute the sum of the module of all parameters. The result of this function is multiplied by a parameter and added to the loss
- the **weight_decay** parameter in the optimizer which applies L2 regularization while initialising optimizer. This adds regularization term to the loss function, with the effect of shrinking the parameter estimates, making the model simpler and less likely to overfit
- **weights initialization** where the weights of the network are initialized depending on the layer's type

1.2 Methods

Dataset Analysis
If the goal is to train a neural network to approximate an unknown function, it is of paramount importance to know how the dataset is made. After loading the data, they are stored in a pandas dataframe to allow an easier manipulation and a better visualization with tables: the main features of the dataset are reported (1, 2). The number of total points for the two dataset is the same, 100, but a part of the training set will be used for the validation. The training data seem to be more distributed along the abscissa axis. In fact, the two sets are both centered in zero (as we can see from the mean), but the std of the former is bigger than the one of the latter even if the max and min are similar.

The training and testing points are already divided in two sets and are represented, thanks to the bokeh library, in an interactive plot in Figure 3. There is in green the training set and in red the test one. Looking at the plot, we can notice a first important thing: not all the regions of the domain are covered by the training points or in other words the two sets are not distributed uniformly along the x-axis. For our algorithm, it will be difficult to learn properly the trend of the function around these regions, as for example near the sample x = -2.

The downloaded dataset are then processed and prepared in order to have the data ready for training the model. A 'torch.utils.data.Dataset' class is created to load and manipulate the data in a more flexible and easy way. It contains the following methods:

- __init__ to initialize the dataset. It is loaded using pandas and stored in a dataframe.
- __len__ returns the size of the dataset.
- __getitem__ to support indexing. It is used to get the i-th sample thanks to the pandas dataframe property *iloc*

It is also created the 'ToTensor' class to convert the data into a pytorch tensor very important to exploit the functionalities of the pytorch models.
Now that we have a dataset to work with and we have defined the transformation to apply to our data, we can use the 'transform.Compose' to combine them in a unique variable. In this case, we only convert the data to tensor. Thanks to the dataset class, we can apply directly the transformations when the data are loaded, enabling a faster manipulation of the samples. Before using the Dataloader, we split the training dataset in two parts: one for the training (80% of the whole dataset) and one for the validation (20% of the whole dataset). The function used for the splitting is *sklearn.model_selection.train_test_split* provided by the sklearn library ([1]). The data are shuffled and taken randomly with a specific seed to allow reproducible results.

Architecture

The **architecture of the model** used for the regression task is composed by the input layer, three hidden linear layers and the output layer. The layers are followed by the **activation function**, in this case we have chosen the ReLU function. Each hidden layer includes also the **dropout layer** which randomly sets input units to 0 with a given probability p at each step during training time. This procedure helps to prevent overfitting. Inputs not set to 0 are scaled up by 1/(1 - p) such that the sum over all inputs is unchanged.

After the first three layers there should be a **batchnormalization layer** which applies a transformation that maintains the mean output close to 0 and the output standard deviation close to 1. This layer has been removed because it does not lead to some beneficial effects for the regularization of the training: indeed we expected a stabilization of the learning process and a dramatically reduction of the number of training epochs required to train the network, but probably with the limitation of the parameters this layer prevented them to learn properly the target function.

The class of the network is made also of two functions:

- **training_step** : it is used to compute the loss during the training procedure and allows to track the gradient
- **validation_step** : it is used to compute the loss during the validation/test procedure and thanks to the *torch.tensor.detach* method the gradient is not computed. This method constructs a new view on a tensor which is declared not to need gradients, i.e., it is to be excluded from further tracking of operations.

We can notice that this is not a deep architecture and the main reason lies in the available dataset. The samples are not too much for training deep architectures with too many parameters: in this way we want to prevent the overfitting due to a too complex model.

After the implementation of the model are defined the functions to train and evaluate it.
We decided to separate the function where the loop over the epochs is performed from the function where the validation dataset (or test dataset) is evaluated. This is done mainly because the two procedures even if are similar they are deeply different: indeed when we compute the loss over the training set we are tracking the gradient and updating the parameters. If the two procedures are separated the code is more flexible for possible changes.

- **evaluate**: this function is used, as already mentioned, to compute the loss without keeping track of the gradient and so it has the unique goal to evaluate the performances of our model
- **fit**: when we call this function we are training the model and for this reason the input parameters have to include the *model*, the *train and validation dataloaders*, the *epochs*, but also some important hyper-parameters to properly tune to obtain the best performances, like the *optimizer*, the *learning rate (lr)*, the *weights decay*(L2 regularizer), the loss function and the *sparsity parameter* (L1 regularizer). The fit function does not return in output only the list of all training and validation losses, but also the number of epoch and validation loss of the minimum value that the error has reached. The last parameter, *verbose*, enable the plot of the function inferred by the algorithm with the training and test data.

The other two functions here defined are used for the regularization:

- **initialize_weights**: if the layer is *convolutional*, the xavier initialization method assignes to the weights a random number with a uniform probability distribution between the range $-(1/\sqrt{n})$ and $1/\sqrt{n}$, where n is the number of inputs to the node. If the layer is *linear*, the kaiming initialization assignes to the weights a random value sampled from a zero-centered Gaussian distribution with standard deviation of $\sqrt{2/n}$. Finally, if there is a *batchnormalization* layer the value of the weights are initialized to a constant value equal to one
- **sparse_loss**: this function compute the value to add to the loss as already described for the L1 regularization

Hyperparameters optimization

The optimization of the hyperparameters is implemented in two ways. The first is a **manual optimization** through some *for loop* where as a gridsearch exhaustively considers all parameter combinations. The second method uses an **automatic optimization** library: Optuna. Optuna is an open source hyperparameter optimization framework to automate hyperparameter search which allows a parallelization over multiple threads or processes without modifying code. It needs an objective function, which returns a numerical value to evaluate the performance of the hyperparameters, in this case the validation loss, and decide where to sample the upcoming trials. The default sampler in Optuna the Tree-structured Parzen Estimater (TPE) is a form of Bayesian Optimization. Optuna uses TPE to search more efficiently than a grid search, by choosing points closer to previous good results. Thanks to many trials it is able to find the best combination of hyperparameters.

Model evaluation

The loss for this task is the Mean Squared Error (MSE).
The evaluation of the model is done in two training sessions: one before the hyperparameters optimization and one after. The **first training** is done to see how the architecture is simply able to infer the function and to also see if everything works properly. The **final training** is done with the best hyperparameters and using the k-fold cross validation (k=5) to obtain a better evaluation of the performances and more stable results. The cross validation reduces also the impact of the splitting of the dataset which can be the cause of an unbalanced training set with the main consequence of a not good generalized trained model. Indeed with very few samples, it is easy to occur in an unbalanced dataset after the splitting where the majority of the data are in the same region of the domain.

1.3 Results

The **results of the first training** confirms that the model works in the right way. Looking at the plot in Figure 4, we can notice that the training is done quite well except for a small region where the dataset is not defined, as we already mentioned. In fact, the performances are good and the inferred function fit properly the curve. The GIF shows us also another important thing: in the first epochs the function change faster to fit better all the data, but at a certain point the variations are very small and it is changed one region at time accordingly with the samples. The loss are also reported in Figure 4 and we can say that the model is not overfitting our data. The loss for the test set is: **0.338**.

The activation of the different layers of the network highlights what we already showed: the model is not learning by heart but is trying to catch the main features of the dataset and to set the parameters in order to infer properly the target function. This is also underlined by the plot of the inferred function which is able to describe the regions where the training points are not present and it is not intersecting all the samples perfectly. Furthermore, the activation are not the same for different value of the input, but the network responds in different way for different stimuli: hence we can say that it is able to discriminate one stimulus from another.

The **results of the final training** are quite similar to the previous one, but thanks to the hyperparameters optimization the loss is smaller and the activation for every layer is more distinct for different inputs. We can affirm also this time that the training has been performed in the proper way. The best hyperparameters are {'Nh1': 80, 'Nh2': 200, 'p': 0.0, 'lr': 0.001, 'weight_decay': 1e-05, 'opt': <class 'torch.optim.adam.Adam'>, 'batch_size': 50} which lead to a best loss of **0.130**. Looking at the plot in Figure 5, we notice that all the regions are quite well described by the curve however there are some zones which show a little bit of overfitting, as for example around x=0. This problem could be reduced with a better tuning of the regularization techniques or with a different number of training epochs (or with the introduction of the earlystopping). Finally, from the histograms of the weights (6, 7), we see how the value of the weights are distributed after the two training. The first histogram is very similar for both procedures, but the last two hidden layers histograms after the final training are less picked around zero, and so there are a bigger variance around the most frequent value. The last layer weights are instead more diffused in the first training, where in the second are more diffused near the most frequent value.

2 CLASSIFICATION

2.1 Introduction

The goal of the second task is to train a neural network that maps an input image (from fashionMNIST dataset, images of Zalando's article) to one of ten classes, hence this is a multi-class classification problem with mutually exclusive classes which are: *t-shirt, trouser, pullover, dress, coat, sandal, shirt, sneaker, bag, boot.* The workflow of this task is the same of the previous one and to achieve the best performances are implemented the following techniques:

- data augmentation: techniques used to increase the amount of data by adding slightly modified copies of already existing data
- convolutional layers
- regularization methods: are the same used for the regression task except for the L1 regularizer because we have seen that it has the only consequence of getting worse results in this classification task

Because of this is a classification task over images, we have also operated a data manipulation to get a more robust and generalized algorithm.

2.2 Methods

Dataset Analysis

The pre processing of the dataset is simply composed by some transformations and as for the regression task the loaded training set is splitted in two parts: one for the train procedure and one for the validation. The main difference respect to the previous task lies in the **augmentation**. The train dataset is loaded twice and are applied different transformations: the first time the samples are converted into Tensors and is applied a normalization. The goal of normalization is to have a similar range of values so that the gradients are less frequent zero during training and leading them to have the same distribution, we ensure that the channel information can be mixed and updated during the back propagation using the same learning rate. The same transformations are applied also to the test set. The second training set is composed by images random rotated and horizontally or vertically flipped: this dataset is added to the previous one to form a larger and more various dataset.

Architecture

The chosen architecture is a Convolutional Neural Network with two convolutional blocks made of a convolutional layer, a batch normalization layer, the activation function (ReLU) and a maxpooling with size 2x2. After these layers there is a flatten to pass from two dimension to one then we have two linear layers. The input of this model are 28x28 grayscale images (one channel) and the output is a one dimensional vector of size 10 as the number of classes where each element is the probability of belonging to a specific class. The classification of an input sample is done giving the most probable class of the output vector. For the training and testing procedure are defined two functions external to the CNN class:

- **train_epoch:** is defined to compute the loss for the train dataloader, indeed are tracked the gradient and updated the weights of the architecture. This is used over many epochs in the training procedure. Some important parameters are taken in input, as the loss, the optimizer and the L1 regularization parameter.
- **test_epoch:** the model is set in the evaluation mode to compute the performances without modifying the model weights. This is used over many epochs in the validation and test procedure.

The total number of trainable parameters is: 1,475,338

Model evaluation

The performances of this classification are evaluated using the Cross Entropy loss and this is used to improve the results over the training. It is also computed the accuracy (the number of correctly classified samples respect to the total samples) but it is not used choose the direction of weight's updating. Like in the regression task, the model has been trained in two sessions: one before the hyperparameters optimization and one after. In this way, we can obtain a fast feedback to see if the architecture and all the settings are working well and then we can train the model to archive the best performances. The hyperparameters optimization is done with Optuna over 20 trials to find the best values of: the batch size, the probability in the dropout, the learning rate, the type of optimizer, the weight decay, the number of filters in the two convolutional layers and the number of neurons in the two linear layers. After finding the best hyperparameters, we train the model using the k-fold cross validation technique with k=5.

2.3 Results

The test results after the two training are reported in the following table.

	Test Loss	**Test Accuracy**
First training	0.327	88.6 %
Final training	0.299	89.6 %

We can see that they are both good results, but thanks to the hyperparameters optimization there is an improvement in the performances in the final loss and accuracy. The list of the best hyperparameters value is: *batch_size*: 300, *p* (dropout probability): 0.4, *lr*: 0.0085, *optimizer*: Adam, *weight_decay*: 0.00005, *filters1*: 35, *filters2*: 35, *linear1*: 375, *linear2*: 105. The epochs in the first training are 15 instead of in the final training are 20 because in the former case is not important a complete trained model but only the trend of the performances.

Loss and accuracy trends: looking at the trends of the **first training** (Figure 8), we can see that the accuracy is always increasing even if there are some oscillations and it has not reached any stationary state. We can affirm that the model is not overfitting our data, but more important for this first training is that the architecture is able to learn properly and it does not seem to be too complex for this task. The trend of the loss is confirming this conclusions, even if it is less stable than the accuracy, it is still decreasing. In particular we notice that it is decreasing at the end of the training which is of paramount importance if we are looking for the minimum of the loss.
The plots of the final training (Figure 9) has a good trend, it is reported only the one one the last fold because they have more or less the same trend. The same considerations made for the first training are

valid also in this case, in particular we can notice the curves of the validation set that are oscillation less than in the first training.

Confusion matrix: starting from the **first training** and thanks to the confusion matrix in Figure 10, we can see where the algorithm made the main mistakes in the classification and what are the classes more difficult to distinguish from each other. In fact, the shirt is with a probability of 0.13 misclassified as a t-shirt, but this is not strange because they are very similar and even for a human is not always easy to discriminate them. Another difficult distinction is between the coat and the pullover, indeed also these two samples are not easy to distinguish, but it is interesting to notice that if from one side the coat is not always clearly identified, the pullover is rare that it is confused with a coat. Probably the reason is that the model has found better the main features of the pullover but the characteristic property of the coat are not yet found.
The confusion matrix of the **final training** (Figure 11) shows us the same results of the previous training with some improvements in the classification of the coat, the pullover and the t-shirt, but also with some worse performance in the discrimination of the shirt.

Filters and layer's response: if we look in more detail how the network works after the final training, we notice that the response of the **first convolutional layer** for a given image is the same of some common filters used in computer vision, we can recognize the horizontal and vertical edge detectors, the . This is not surprising if we know that the filter operation is a convolution and so the convolutional layers must act at the same way. It is more interesting the fact that these filters operates in order to extract information to discriminate one class from the other and there is not one filter that is acting like another: they are all trying to detect different features using different convolutions. The **second convolutional layer** is doing something more complicated. The filters in this layer are operating on the results of the previous ones and so they are looking for features on another level, more deeply. The result seems to be a specialization in some areas where there are filters looking for a specif feature in the image and probably a features that allow to the model to distinguish a specific class from the others. If there was a discriminating feature unique for each class we should probably see it in one of the convolution and the classification will not have any mistake.

Label space visualization: it is reported in Figure 12 the space of the label reduced from ten to two dimensions thanks to the t-sne technique (in the notebook there is the interactive plot obtained with *Plotly Express*). We notice that the test samples of the same label (same color) are grouped in clusters, this highlights how the network is able to catch some common features from the images of the same class, but also we see how it is difficult to distinguish input from similar classes. The trousers (red) are clearly apart from the other samples and for example the cluster of the boots (light blue), of the sneakers (pink) and of the sandal (green) are close to each other, probably because the model recognize them similar, indeed they are all footwear. The shirt, the t-shirt, the pullover and the coat are really mixed together and this is because even for us some samples are very similar and they have a lot of property in common. In the end, we can affirm that for classification purposes it is not good that these cluster intersect each other, but on the other hand it is not wrong because they are really similar and the network understands it. What the model has to do is to find more characterizing property that are unique for each class and allow to unequivocally distinguish every single class.

References

[1] `https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html`

Appendix

	input	label
count	100.000000	100.000000
mean	0.535661	2.826761
std	3.511726	1.962990
min	-4.915863	-3.742970
25%	-3.255942	1.677844
50%	0.037716	2.663394
75%	4.143882	3.727911
max	4.977516	7.199304

Figure 1: Train dataset

	input	label
count	100.000000	100.000000
mean	-0.088111	2.572386
std	2.924814	1.697217
min	-4.999868	-2.999869
25%	-2.733220	1.781779
50%	0.029972	2.715547
75%	2.208296	3.767334
max	4.894875	6.335996

Figure 2: Test dataset

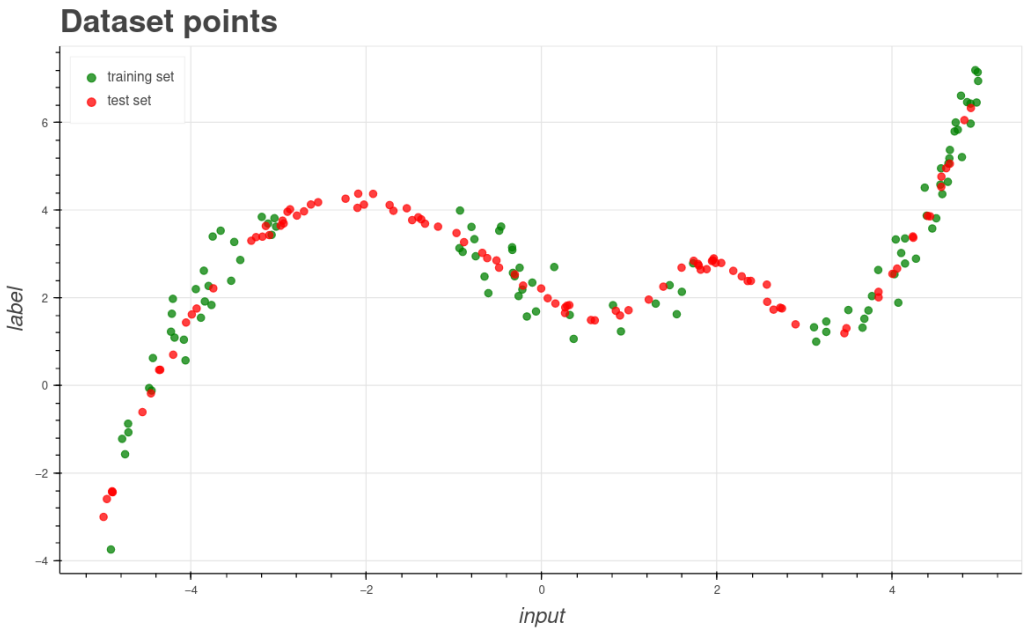

Figure 3: Dataset representation

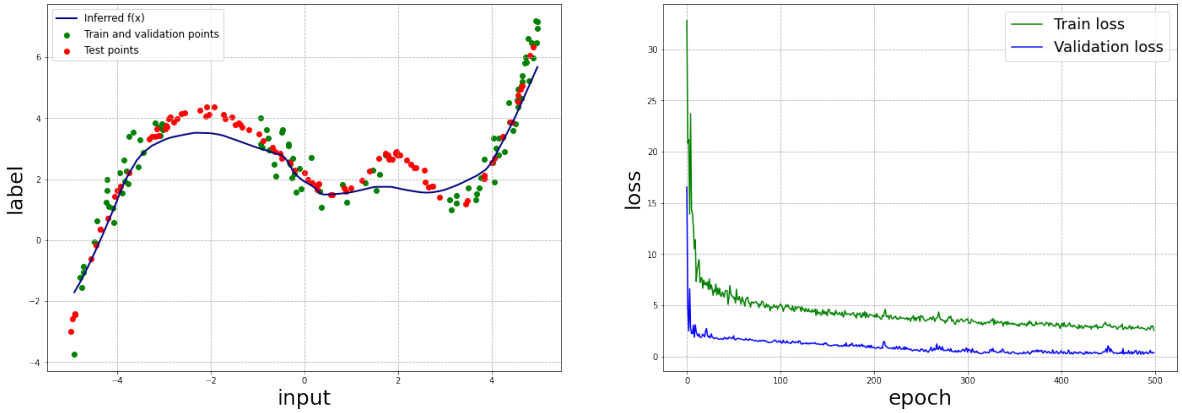

Figure 4: Loss of first training in regression

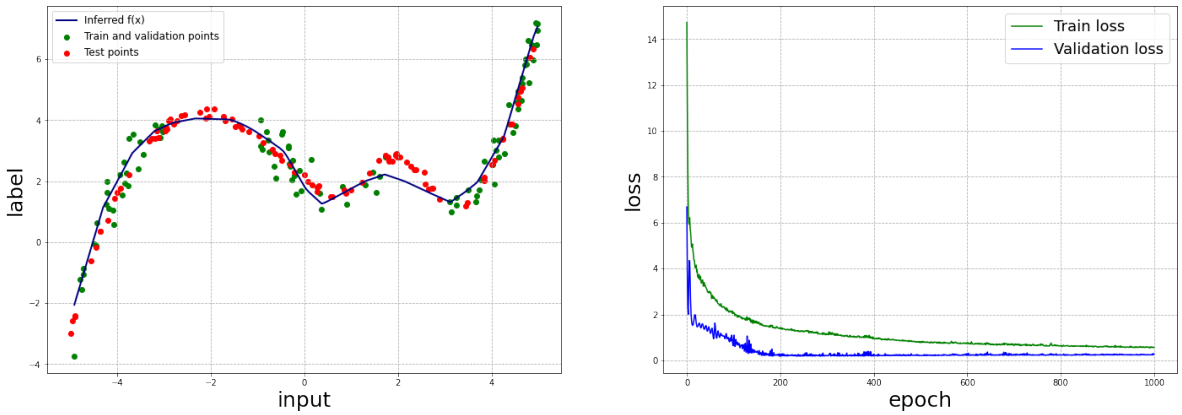

Figure 5: Loss of final training in regression (last fold)

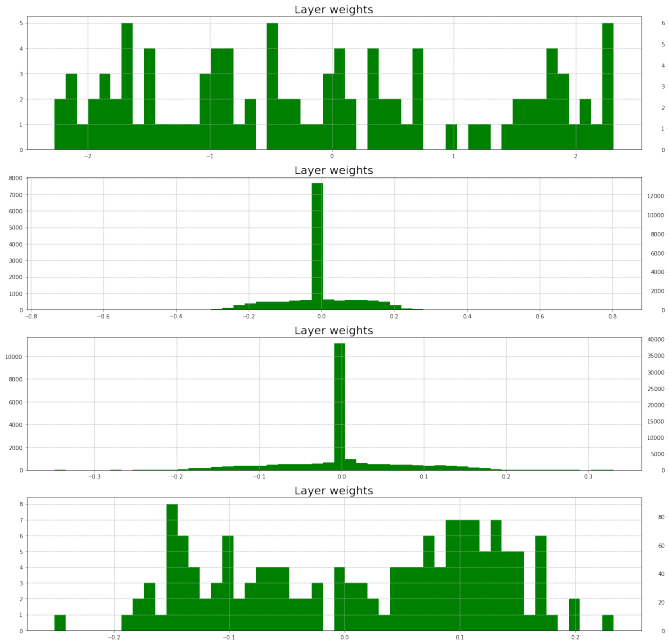

Figure 6: Weights of first training

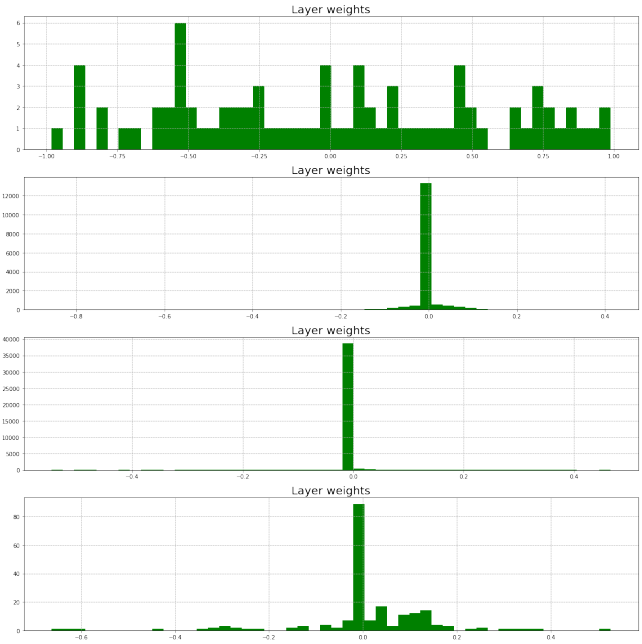

Figure 7: Weights of final training

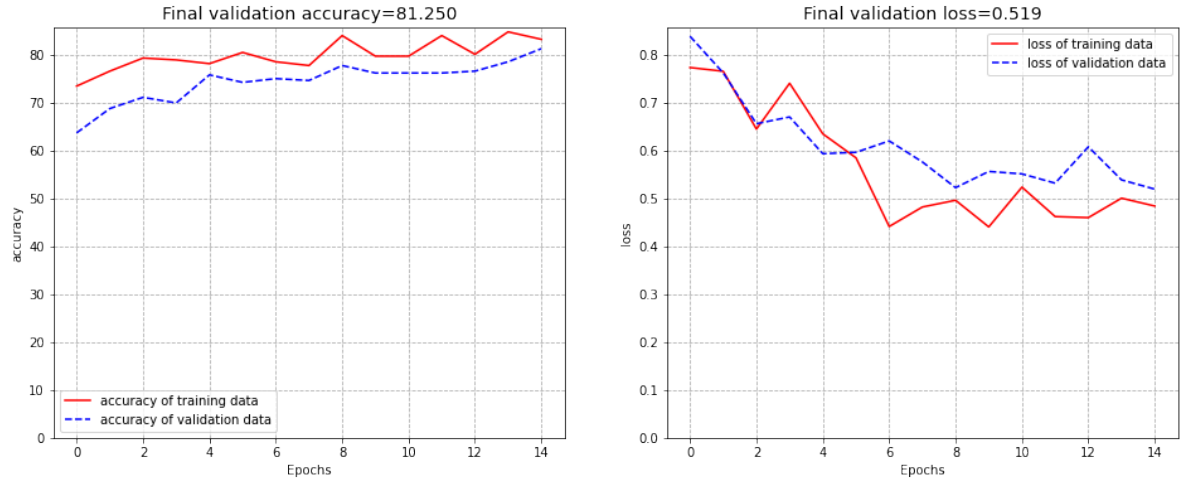

Figure 8: Loss of first training in classification

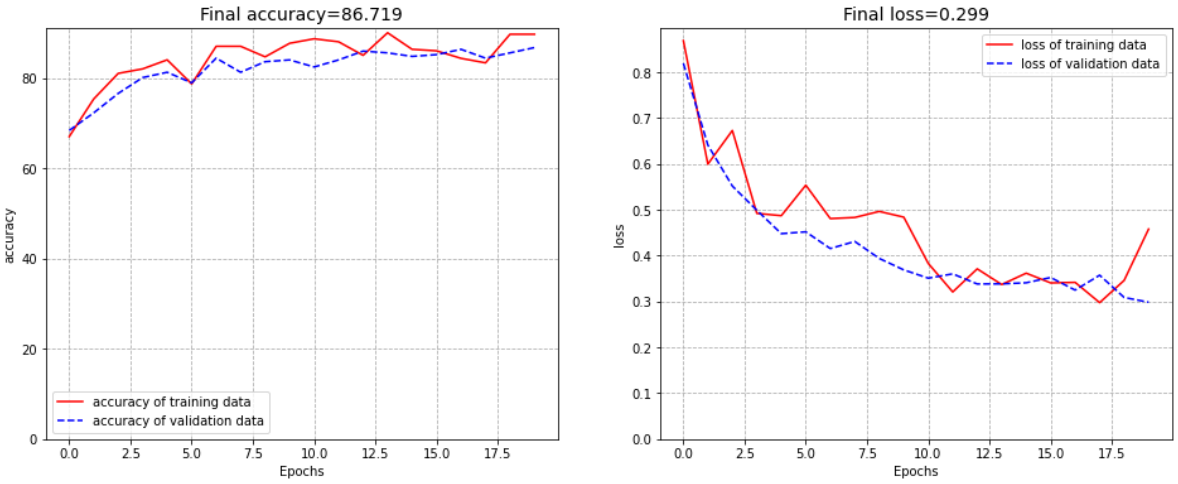

Figure 9: Loss of final training in classification (last fold)

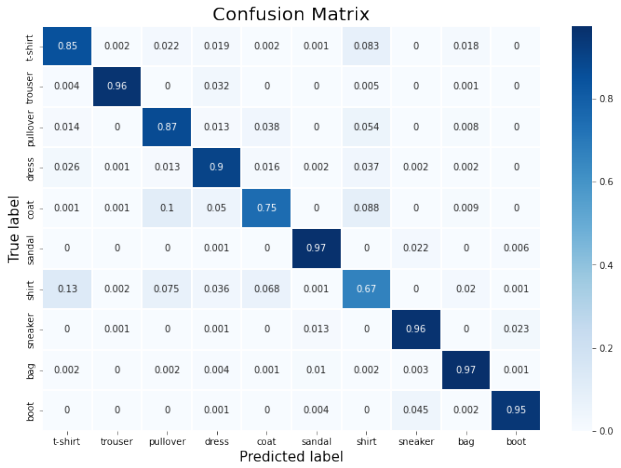

Figure 10: Confusion matrix first training

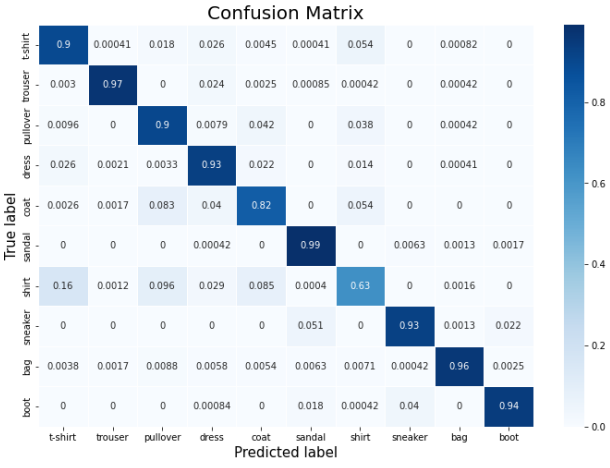

Figure 11: Confusion matrix final training

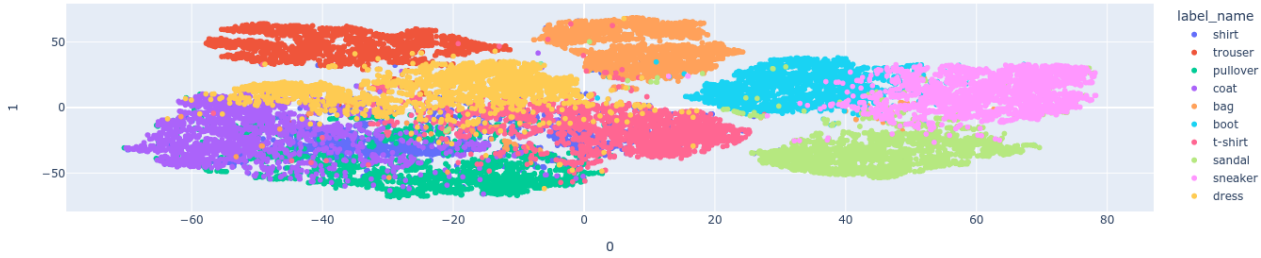

Figure 12: Label space visualization (t-SNE)