Python implementation - numpy

We will create a module that will include a few files with different components:

- *"neuralnet"* – it will include the main class to create a neural network that will "comunicate" with the other modules;
- "layers" – it will include the layers that will make the NNs (dense, etc);
- "activations" – it will include the activation layers (sigmoid, tahn, relu, etc);
- "optimizer" – it will include the training algorithms used to update the parameters of the NNs during training (SGD);
- "losses" – it will include the loss functions to optimize during training (CrossEntropy, MeanSquaredError, etc)
- “data” – dataset (similar to previous ones)
- “metrics” - error metrics

These components will include some core implementations, but also make up a base for further developments

Python implementation - Layers

- We will first implement a base layer with some shared and **abstract** methods for all other specific layers.
- class Layer:
  - methods:
    - forward_propagation – abstract method that computes the output of a layer given the input;
    - backward_propagation – abstract method that back propagates the output error;
    - output_shape – abstract method that returns the output shape of the layer;
    - parameters – abstract mehtod that returns the number of parameters of the layer;
    - set_input_shape – sets the input shape of the layer;
    - input_shape – returns the input_shape of the layer;
    - layer_name – returns the name of the layer;

Python implementation – Dense Layer

- A dense layer, also known as a fully connected layer, connects each neuron to every neuron in the previous and next layers, forming a dense network of connections.
- class DenseLayer(Layer):
  - arguments:
    - n_units – number of neurons in the NN;
    - input_shape – tuple with the input_shape (for tabular data it will be (n_features,))
  - estimated parameters:
    - input – the layer input;
    - output – the layer output;
    - weights – the layer weights;
    - biases – the layer biases;
  - methods:
    - initialize – initializes the layer with random weiights and biases and with the optimizer;
    - parameters – returns the number of parameters of the layer;
    - forward_propagation - perform forward propagation on a given input;
    - **backward_propagation - perform backward propagation on the given output error;**
    - output_shape - returns the shape of the output of the layer.

Complete this function !

Python implementation – Activation layers

- class ActivationLayer(Layer):
  - methods:
    - forward_propagation - perform forward propagation on a given input by applying the activation function;
    - backward_propagation - perform backward propagation on the given output error by applying the derivative of the activation layer on the input and multiplying it with the output error;
    - activation_function – abstract method that computes the activation function on the given input (used on the forward propagation);
    - derivative – abstract method that computes the derivative of the activation function on the given input (used on the back propagation);
    - output_shape - returns the shape of the output of the layer (which is the same as the input_shape);
    - parameters – returns the number of parameters of the layer (returns 0 as activation layers do not have learnable parameters).

PYTHON IMPLEMENTATION

Python implementation – Activation layers

- class ReLUActivation(ActivationLayer):
  - methods:
    - activation_function – applies the ReLU function on the input (argument)
    - **derivative – applies the ReLU derivative on the input (argument)**

Complete these functions !

- class SigmoidActivation(ActivationLayer);
  - methods:
    - activation_function – applies the sigmoid function on the input (argument)
    - **derivative – applies the sigmoid derivative on the input (argument)**

Complete these functions !

Python implementation – Loss functions

- We will first implement a base loss function class with two abstract methods for all other specific layers (the class will not need an __init__).
- class LossFunction:
  - methods:
    - loss – abstract method that computes the loss based on the true and predicted labels
    - derivative - abstract method that computes the derivative of the loss based on the true and predicted labels
- class MeanSquaredError(LossFunction):
    - loss - compute the mean squared error loss function;
    - **derivative - compute the derivative of the mean squared error loss function**
- class BinaryCrossEntropy(LossFunction):
    - loss - compute the binary cross-entropy loss function;
    - **derivative - compute the derivative of the binary cross-entropy loss.**

Complete these functions !

Complete these functions !

Python implementation – optimizer

- Optimizers are algorithms that adjust the model's parameters during training to minimize the loss function.
- We will implement the SGD optimization algorithm, with the option of considering the momentum

  - arguments:
    - learning_rate – the learning rate to use for updating the weights;
    - momentum – the momentum to use for updating the weights.
  - estimated_patameters:
    - retained_gradient – accumulated gradient from previous epochs;
  - methods:
    - update - Update the retained_gradient, computes and returns the updated weights;

Python implementation – neural network

- Now that we have all the components we can build a NN. The NeuralNetwork class will connect all the other classes together.
- class NeuralNetwork:
  - arguments:
    - epochs - the number of epochs to train the neural network;
    - batch_size - the batch size to use for training the neural network;
    - optimizer - the optimizer to use for training the neural network;
    - learning_rate - the learning rate to use for training the neural network;
    - momentum – parameter of momentum;
    - verbose - whether to print the loss and the metric at each epoch;
    - loss - the loss function to use for training the neural network;
    - metric - the metric to use for training the neural network;

Python implementation – neural network

- class NeuralNetwork:
  - estimated parameters:
    - layers – the layers of the NN.
    - history – dictionary containning the loss and metric at each epoch.
  - methods:
    - add – add a layer to the NN.
    - get_mini_batches – handle training batches yielding sets of exemples for training
    - forward_propagation – forward propagation of the full feedforward NN, going through all layers to compute final output for a set of examples
    - **backward_propagation – backward propagation of the full feedforward NN for a set of examples, going through all layers in reverse order to calculate errors and update weights**
    - fit – train the NN (will call both forward and backward propagation).
    - predict – make predictions on new data (calls forward propagation);
    - score – evaluate the NN on a given dataset (calls predict);

Complete this function !

Python implementation – neural network

⑩ NeuralNetwork.fit:

- arguments:
  - dataset – the dataset to train the NN on
- algorithm:
  1. Compute mini batches of size self.batch_size;
  2. Perform forward propagation for all layers;
  3. Compute the loss derivative error;
  4. Backpropagate the error through all layers using backward propagation;
  5. Repeat steps 2, 3 and 4 for all mini batches;
  6. Compute the loss based on true labels and predictions;
  7. Compute the metric based on true labels and predictions;
  8. Save the loss and metric in the history dictionary;
  9. Print the metric and loss if verbose is set to True;
  10. Repeat the previous steps for all epochs.

Training algorithms in pytorch

Jupyter Notebook:
….ipynb

Several algorithms have been proposed and are available in pytorch; the notebook shows its use in the **MNIST** case including the algorithms: Adam, RMSprop, SGD

Training algorithms include a set of parameters (dependent on the algorithm) which may be tuned/ optimized where the most relevant is the **learning rate (α)**

Overfitting in pytorch

Jupyter Notebook:
….ipynb

MNIST

- Controlo de complexidade
- **Dropout**
- **L1/L2**
- **Early stopping**

IMDB dataset - exercise

Jupyter Notebook:
….ipynb

IMDB dataset

Dataset with texts of reviews in IMDB about movies; classified into 2 classes: positive or negative; 25k reviews for training + 25k for test; balanced – 50% positive and negative examples
We will only consider the 10k words that are most common

This dataset will be used here, considering a one-hot encoding scheme where the 10K most common words will be considered as binary features.

So, if a word appears in a review the value of the feature will be 1, if not it will be 0

Notice that in this way, the order of the words is neglected

DNNs for the IMDB

- DNN – 2 hidden layers
- 16 neurons in each; RelU
- Output layer – sigmoid – binary classification

```
# Overfitting-prone dense network
hidden_nodes = 64
class IMDBNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(max_words, hidden_nodes),
            nn.ReLU(),
            nn.Linear(hidden_nodes, hidden_nodes),
            nn.ReLU(),
            nn.Linear(hidden_nodes, 1),
            nn.Sigmoid()
        )

    def forward(self, x):
        return self.net(x)

model = IMDBNet().to(device)
```

DNNs for the IMDB

```
# Loss + optimizer
criterion = nn.BCELoss()
optimizer = optim.RMSprop(model.parameters(), lr=0.01)
```

Loss function – binary cross entropy (used in logistic regression)

```
# Validation split

x_train, x_val, y_train, y_val = train_test_split(
    x_train, train_labels, test_size=10000, random_state=42, stratify=train_labels
)
```

```
# Convert to tensors

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

x_train = torch.tensor(x_train).to(device)
y_train = torch.tensor(y_train).unsqueeze(1).to(device)

x_val = torch.tensor(x_val).to(device)
y_val = torch.tensor(y_val).unsqueeze(1).to(device)
```

Defines validation set

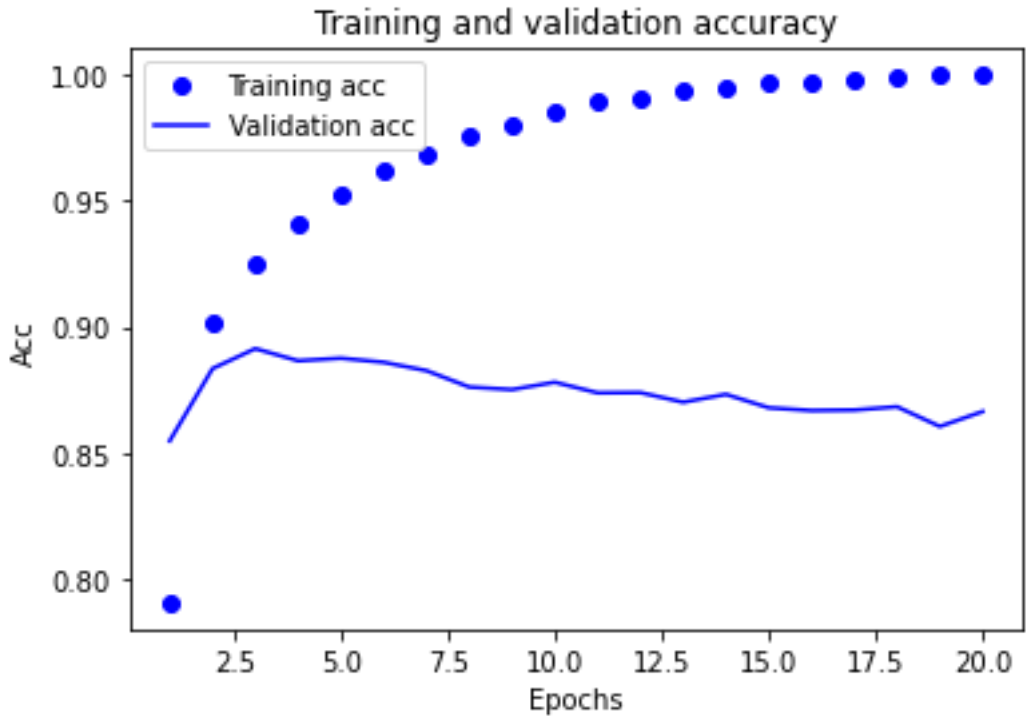
DNNs for the IMDB

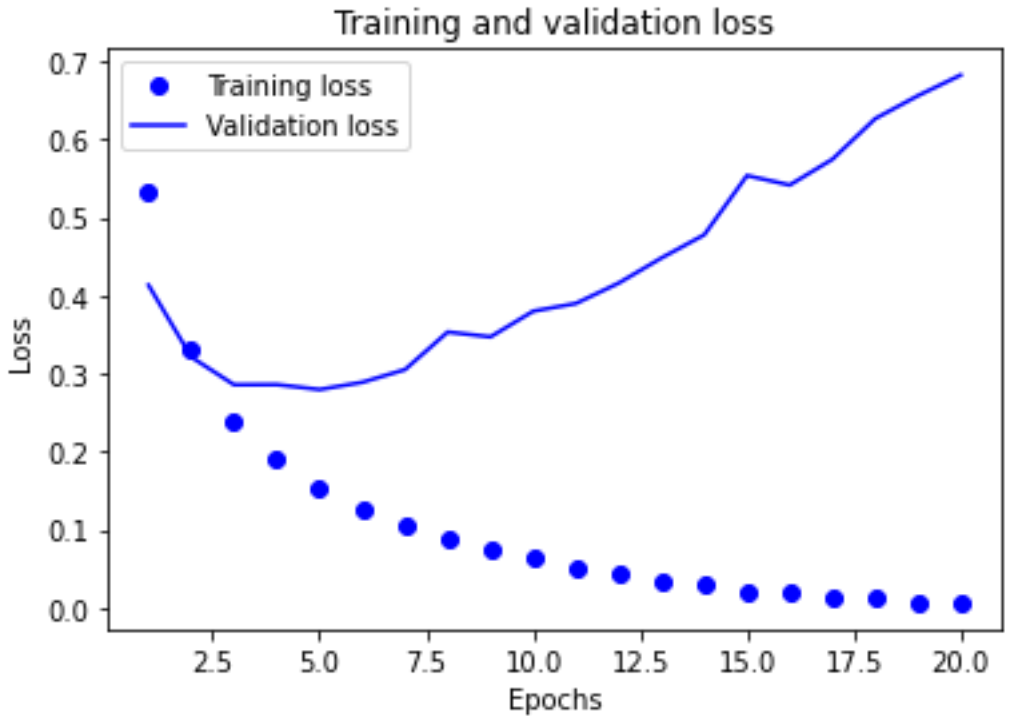
- Model training

```
# Training loop
epochs = 20
batch_size = 512
train_losses = []
val_losses = []

for epoch in range(epochs):
    perm = torch.randperm(x_train.size(0))
    epoch_loss = 0
    for i in range(0, x_train.size(0), batch_size):
        idx = perm[i:i + batch_size]

        xb = x_train[idx]
        yb = y_train[idx]

        preds = model(xb)
        loss = criterion(preds, yb)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        epoch_loss += loss.item()

    train_losses.append(epoch_loss)

    with torch.no_grad():
        val_preds = model(x_val)
        val_loss = criterion(val_preds, y_val)
        val_losses.append(val_loss.item())

    print(f"Epoch {epoch+1:02d} | Train loss: {epoch_loss:.3f} | Val loss: {val_loss:.3f}")
```

DNN for IMDB

Plot of accuracy and loss in training and validation sets

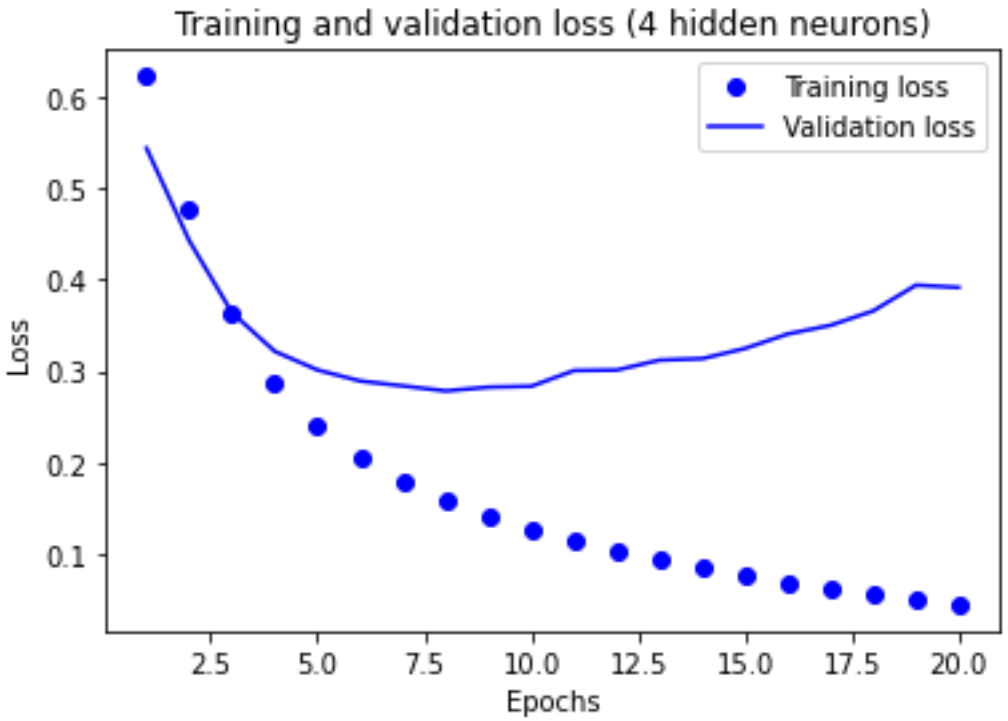


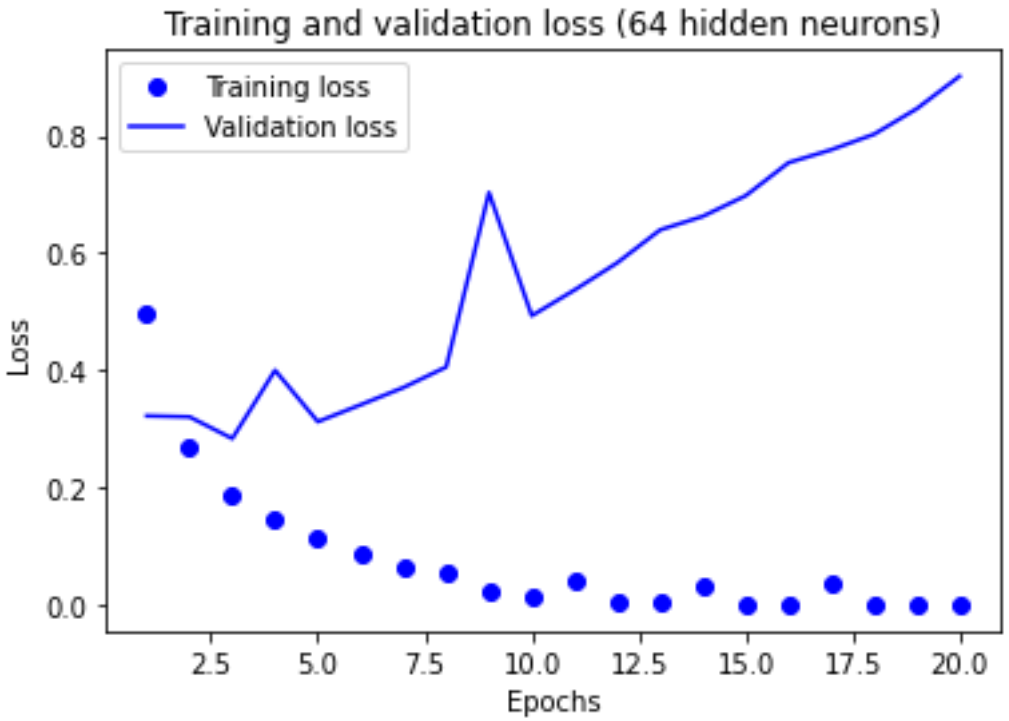


What do these graphs suggest ?

Overfitting: IMDB

Controlling the model's capacity



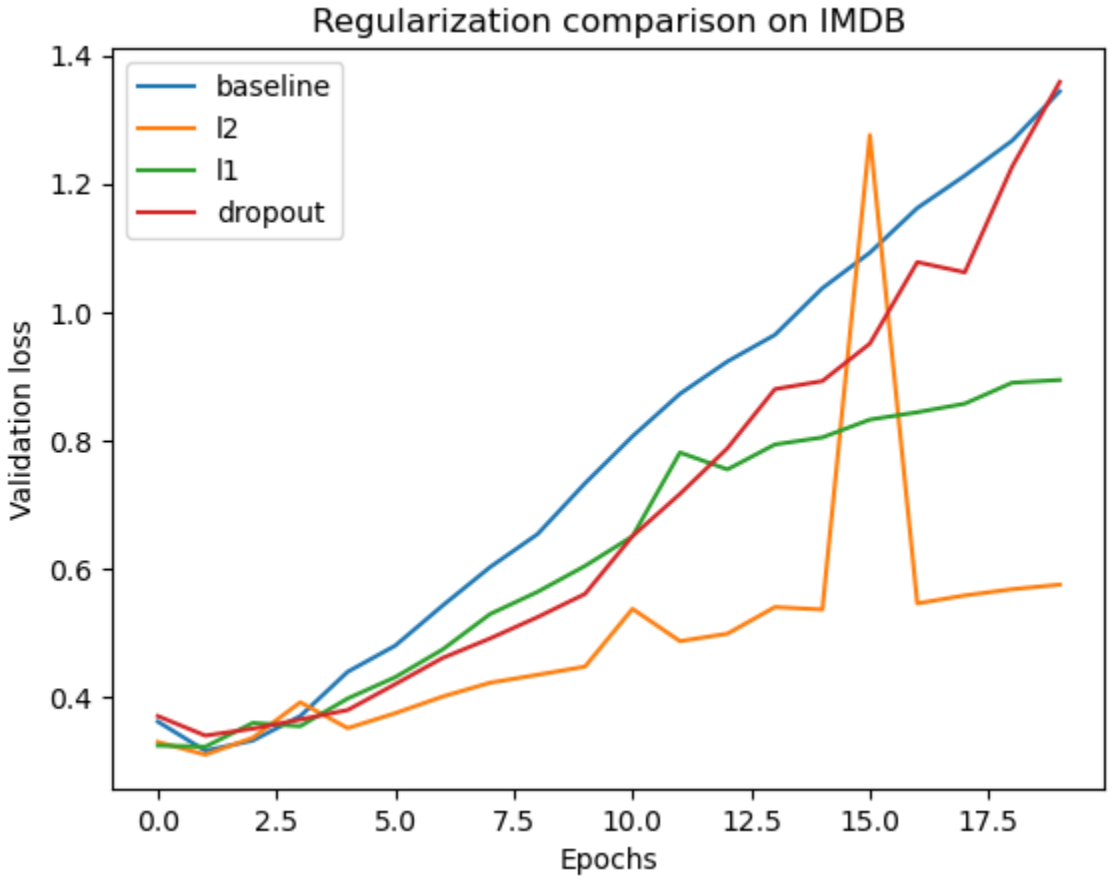


Overfitting: IMDB

Regularization & Dropout

```python
def train_model(x_train, y_train, x_val, y_val,
                l1_lambda=0.0,
                l2_lambda=0.0,
                dropout=0.0,
                epochs=20,
                batch_size=512):

    device = x_train.device
    model = IMDBNet(dropout=dropout).to(device)

    criterion = nn.BCELoss()
```

```python
trained_models = {}
results = {}

trained_models["baseline"], _, results["baseline"] = train_model(x_train, y_train, x_val, y_val)

trained_models["l2"], _, results["l2"] = train_model(
    x_train, y_train, x_val, y_val,
    l2_lambda=1e-3
)

trained_models["l1"], _, results["l1"] = train_model(
    x_train, y_train, x_val, y_val,
    l1_lambda=1e-5
)

trained_models["dropout"], _, results["dropout"] = train_model(
    x_train, y_train, x_val, y_val,
    dropout=0.5
)
```



Overfitting: IMDB

```
=== Test set performance ===
baseline | Test loss: 1.4191 | Test accuracy: 0.8460
l2       | Test loss: 0.5953 | Test accuracy: 0.8427
l1       | Test loss: 0.9019 | Test accuracy: 0.8433
dropout  | Test loss: 1.1257 | Test accuracy: 0.8609
```