TÉCNICO LISBOA

Instituto Superior Técnico

Aprendizagem Profunda / Deep Learning

MEEC

Deep Learning Homework 1, 2022/2023

Students:
Afonso Brito Caiado Alemão | 96135
Rui Pedro Canário Daniel | 96317

Group **8**

Teachers:
André Martins, Ben Peters, Margarida Campos

December 22, 2022

Contents

0 Contribution of each member

Each member of the group contributed to the whole realization of this project.

1 Image classification with linear classifiers and neural networks

In this exercise, we implement a linear classifier for a simple image classification problem, using the Kuzushiji-MNIST dataset, which contains handwritten cursive images of 10 characters from the Hiragana writing system (used for Japanese). Examples of images in this dataset are shown in Fig. 1.

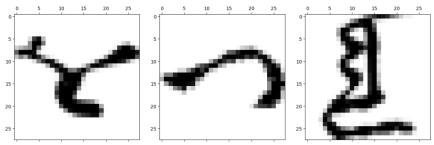


Figure 1: Examples of images from the Kuzushiji-MNIST dataset.

1.1

1.1.1 (a)

We implement the `update_weights()` method of the `Perceptron` class in *hw1-q1.py*. In Perceptron algorithm, for each training example (x_i, y_i), it is predicted $\hat{y}_i = \arg\max_{y \in \mathcal{Y}} \Big(w_y \cdot \phi(x_i) \Big)$. Next, it is performed the update of the weigths: if $\hat{y}_i$ is different from y_i, then we increase the weight of gold class updating $w_{y_i} \leftarrow w_{y_i} + \eta\phi(x_i)$ and we decrease the weight of incorrect class updating $w_{\hat{y}_i} \leftarrow w_{\hat{y}_i} - \eta\phi(x_i)$.

Our code implementation is reported below.

```
class Perceptron(LinearModel):
    def update_weights(self, x_i, y_i, **kwargs):
        """
        x_i (n_features): a single training example
        y_i (scalar): the gold label for that example
        other arguments are ignored
        """
        # Q1.1a
        eta = 1
        # Sign function.
        y_hat = np.argmax(self.W.dot(x_i))
        if y_hat != y_i:
            # Perceptron update.
            self.W[y_i, :] += eta * x_i
            self.W[y_hat, :] -= eta * x_i
```

We trained 20 epochs of our model on the training set. Its performance on the validation set and test set is reported in the plot of the accuracies as a function of the epoch number in Fig. 2.

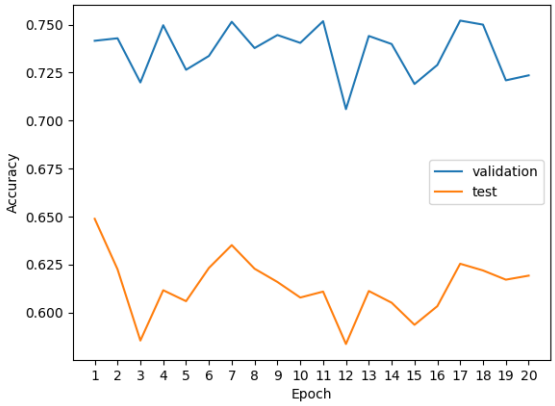


Figure 2: Accuracy of training set and of validation set as a function of the epoch number for Perceptron.

1.1.2 (b)

In this question the goal is to solve the same problem with Logistic Regression (without regularization), using stochastic gradient descent as training algorithm with a fixed learning rate $\eta = 0.001$.

Multinomial logistic regression defines conditional probability as given by (1) and it learns weights to maximize conditional log-likelihood of training data as given in (2) and (3). Loss function in logistic regression is given by (4) and its gradient by (5). We denote e_y the one-hot vector representation of class y, given by (6).

$$P_W(y|x) = \frac{e^{w_y \cdot \phi(x)}}{Z_x}, \quad Z_x = \sum_{y' \in \mathcal{Y}} e^{(w_{y'} \cdot \phi(x))} \tag{1}$$

$$\hat{W} = \arg\max_W \log \left(\prod_{t=1}^{N} P_W(y_t|x_t) \right) = \arg\min_W - \sum_{t=1}^{N} \log P_W(y_t|x_t) \tag{2}$$

$$\hat{W} = \arg\min_W \sum_{t=1}^{N} L(W; (x_t, y_t)) = \arg\min_W \sum_{t=1}^{N} \left(\log \sum_{y'_t} e^{w_{y'_t} \cdot \phi(x_t)} - w_{y_t} \cdot \phi(x_t) \right) \tag{3}$$

$$L(W; (x, y)) = \log \sum_{y'} e^{w_{y'} \cdot \phi(x)} - w_y \cdot \phi(x) \tag{4}$$

$$\nabla_W L(W; (x, y)) = \sum_{y'} P_W(y'|x) e_{y'} \phi(x)^T - e_y \phi(x)^T \tag{5}$$

$$e_y = [0, ..., 0, 1, 0, ..., 0]^T \text{ , 1 in } i\text{-th position} \tag{6}$$

In stochastic gradient descent we make one update for each training example (x_t, y_t). So, instead of summing across all data points we adapt the learning rule, for one example only, in (7).

$$W \leftarrow W - \eta \nabla_W L(W^k; (x_t, y_t)) = W + \eta \left(e_y \phi(x)^T - \sum_{y'} P_W(y'|x) e_{y'} \phi(x)^T \right) \tag{7}$$

In order to solve it, we implemented the `update_weights()` method in the `LogisticRegression` class. Our code implementation is reported below.

```
class LogisticRegression(LinearModel):
    def update_weight(self, x_i, y_i, learning_rate=0.001):
        """
        x_i (n_features): a single training example
        y_i: the gold label for that example
        learning_rate (float): keep it at the default value for your plots
        """
        # Q1.1b
        # Label scores according to the model (num_labels x 1).
        label_scores = self.W.dot(x_i)[:, None]
        # One-hot vector with the true label (num_labels x 1).
        y_one_hot = np.zeros((np.size(self.W, 0), 1))
        y_one_hot[y_i] = 1
        # Softmax function.
```

```
        # This gives the label probabilities according to the model (num_labels x 1).
        label_probabilities = np.exp(label_scores) / np.sum(np.exp(label_scores))
        # SGD update. W is num_labels x num_features.
        self.W += learning_rate * (y_one_hot - label_probabilities) * x_i[None, :]
```

We trained 20 epochs of our model on the training set. Its performance on the validation set and test set is reported in the plot of the accuracies as a function of the epoch number in Fig. 3.

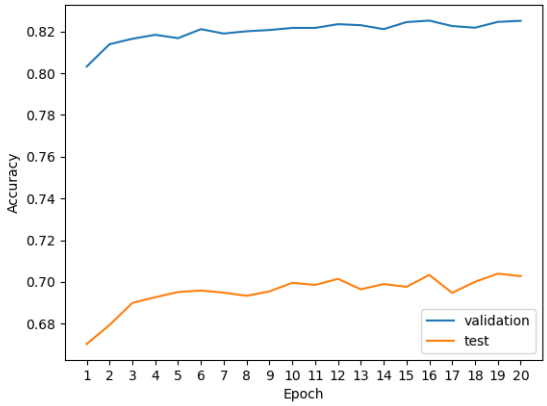


Figure 3: Accuracy of training set and of validation set as a function of the epoch number for Logistic Regression.

1.2

In this section we implement a multi-layer perceptron (a feed-forward neural network) using as input the original feature representation, i.e., simple independent pixel values.

1.2.1 (a)

Perceptron is an instance of a linear classifier. If a training set is separable by some margin, it will find the weights that separates the data (a separating hyperplane). However, it does not pick the weights to maximize the margin.

Multi-layer perceptrons (MLPs) are universal function approximators that need to be trained. Stochastic gradient descent is an effective training algorithm. This is possible with the gradient backpropagation algorithm and also dropout regularization is effective to avoid overfitting. A MLP contains one or more hidden layers (apart from one input and one output layer). While a single layer perceptron can only learn linear functions, a MLP can also learn non-linear functions.

It was proved that Perceptron, with only one neuron, can't be applied to non-linear data. The MLP was developed to overcome this limitation because it is a neural network where the mapping between inputs and output can be non-linear.

While in the Perceptron the neuron must have an activation function that imposes a threshold, like ReLU or sigmoid, neurons in a MLP can use any arbitrary activation function.

MLPs with non-linear activations are more expressive than the simple perceptron because in MLP each of the hidden units computes some representation of the input and propagates forward that representation which increases the expressive power of the network, yielding more complex, non-linear, functions/classifiers. However if the MLP is composed by layers of linear activation units, this is equivalent to a single linear layer, so no expressive power increase by using multiple layers.

1.2.2 (b)

Without using any neural network toolkit, we implement a multi-layer perceptron with a single hidden layer to solve this problem, including the gradient backpropagation algorithm which is needed to train the model. We use 200 hidden units, a ReLU activation function for the hidden layers, and a multinomial logistic loss (also called cross-entropy) in the output layer, and train the model with stochastic gradient descent with a learning rate $\eta = 0.001$. Biases are initialized with zero vectors and values in weight matrices with $w_{ij} \sim N(\mu, \sigma^2)$ using $\mu = 0.1$ and $\sigma^2 = 0.1^2$.

We start by checking the matrix dimensions of the connection weights and the biases in (8), knowing that for each input we have 784 features, and that there is one hidden layer with 200 units and the output layer have 10 units because there are 10 possible classes.

$$W^{(1)}_{200\text{x}784}\,, \quad W^{(2)}_{10\text{x}200}\,, \quad b^{(1)}_{200\text{x}1}\,, \quad b^{(2)}_{10\text{x}1} \tag{8}$$

In the next phase, it will be performed forward propagation given by (11) and (12). We define $h^{(0)} = x$ for convenience. The ReLU activation function $g(z)$, given by (9), is not differentiable for $z = 0$, but we define $g'(z) = 0$ for $z = 0$, for convenience. So $g'(z)$ is given by (10).

$$g(z) = \max(0, z) \tag{9}$$

$$g'(z) = \begin{cases} 0, & \text{if } z \leq 0 \\ 1, & \text{otherwise} \end{cases} \tag{10}$$

$$z^{(1)} = W^{(1)} h^{(0)} + b^{(1)} \ , \quad h^{(1)} = g(z^{(1)}) \tag{11}$$

$$\hat{y} = z^{(2)} = W^{(2)} h^{(1)} + b^{(2)} \ , \quad p = \text{softmax}(z^{(2)}) \tag{12}$$

Each output unit uses softmax to compute label probabilities p_i, as in (13). We use a constant $m = \max z_i$ in order to avoid overflow.

$$p_i = \frac{e^{z_i}}{\sum_{k=1}^{d} e^{z_k}} = \frac{e^{z_i - m}}{\sum_{i=k}^{d} e^{z_k - m}} \tag{13}$$

Then, it will be performed the backward phase.
Given the true output y (a one-hot vector), the cross-entropy loss is given by (14).

$$L(y,p) = -\sum_{k=1}^{d} y_k \ \log \ p_k = -\sum_{k=1}^{d} y_k \left(z_k - \ \log \ \sum_j e^{z_j} \right) = -\sum_{k=1}^{d} y_k z_k + \left(\sum_{k=1}^{d} y_k \right) \ \log \ \sum_j e^{z_j} = -y^T z + \ \log \ \sum_j e^{z_j} \tag{14}$$

We need to derive all functions in our network. The gradient in the output layer is given by (15), the gradients of hidden layer parameters are given by (16), the gradients of hidden layer below are given by (17) and the gradients of hidden layer below (before activation $h^{(l)} = g(z^{(l)})$) are given by (18).

$$\frac{\partial L(y,p)}{\partial z^{(L)}} = p - y \tag{15}$$

$$\frac{\partial L(y,p)}{\partial W^{(l)}} = \frac{\partial L(y,p)}{\partial z^{(l)}} h^{(l-1)^T} \ , \quad \frac{\partial L(y,p)}{\partial b^{(l)}} = \frac{\partial L(y,p)}{\partial z^{(l)}} \tag{16}$$

$$\frac{\partial L(y,p)}{\partial h^{(l-1)}} = W^{(l)^T} \frac{\partial L(y,p)}{\partial z^{(l)}} \tag{17}$$

$$\frac{\partial L(y,p)}{\partial z^{(l-1)}} = \frac{\partial L(y,p)}{\partial h^{(l-1)}} \odot g'(z^{(l-1)}) \tag{18}$$

These quantities are computed recursively from the last layer to the first one, in (19), (20), (21), (22) and (23).

$$\frac{\partial L(y,p)}{\partial z^{(2)}} = p - y \tag{19}$$

$$\frac{\partial L(y,p)}{\partial W^{(2)}} = \frac{\partial L(y,p)}{\partial z^{(2)}} h^{(1)^T} \ , \quad \frac{\partial L(y,p)}{\partial b^{(2)}} = \frac{\partial L(y,p)}{\partial z^{(2)}} \tag{20}$$

$$\frac{\partial L(y,p)}{\partial h^{(1)}} = W^{(2)^T} \frac{\partial L(y,p)}{\partial z^{(2)}} \tag{21}$$

$$\frac{\partial L(y,p)}{\partial z^{(1)}} = \frac{\partial L(y,p)}{\partial h^{(1)}} \odot g'(z^{(1)}) \tag{22}$$

$$\frac{\partial L(y,p)}{\partial W^{(1)}} = \frac{\partial L(y,p)}{\partial z^{(1)}} h^{(0)^T} \ , \quad \frac{\partial L(y,p)}{\partial b^{(1)}} = \frac{\partial L(y,p)}{\partial z^{(1)}} \tag{23}$$

Finally, the last phase is to perform the updates, starting with the first layer, in (24), (25) and (26).

$$W^{(1)} = W^{(1)} - \eta \frac{\partial L(y,\hat{y})}{\partial W^{(1)}} \ , \quad b^{(1)} = b^{(1)} - \eta \frac{\partial L(y,\hat{y})}{\partial b^{(1)}} \tag{24}$$

$$W^{(2)} = W^{(2)} - \eta \frac{\partial L(y,\hat{y})}{\partial W^{(2)}} \ , \quad b^{(2)} = b^{(2)} - \eta \frac{\partial L(y,\hat{y})}{\partial b^{(2)}} \tag{25}$$

$$W^{(3)} = W^{(3)} - \eta \frac{\partial L(y,\hat{y})}{\partial W^{(3)}} \ , \quad b^{(3)} = b^{(3)} - \eta \frac{\partial L(y,\hat{y})}{\partial b^{(3)}} \tag{26}$$

In order to achieve that, we implemented the `MLP` class in *hw1-q1.py*. Our code implementation is reported below.

```
class MLP(object):
    def __init__(self, n_classes, n_features, hidden_size, num_layers):
        # Initialize an MLP with a single hidden layer.

        # num_layers = 1
```

```
        self.n_classes = n_classes

        # Single hidden layer MLP with 200 hidden units.
        # First is input size, last is output size.
        units = [n_features, hidden_size, n_classes]

        # Initialize all self.W and self.b randomly.
        # mu = 0.1 and sigma^2 = 0.1^2
        # sigma * np.random.randn(...) + mu

        W1 = .1 * np.random.randn(units[1], units[0]) + 0.1
        b1 = np.zeros(units[1])
        W2 = .1 * np.random.randn(units[2], units[1]) + 0.1
        b2 = np.zeros(units[2])

        self.W = [W1, W2]
        self.b = [b1, b2]

    def predict(self, X):
        # Compute the forward pass of the network. At prediction time, there is
        # no need to save the values of hidden nodes, whereas this is required
        # at training time.
        predicted_labels = []
        for x in X:
            output, _ = self.forward(x)
            y_hat = self.predict_label(output)
            predicted_labels.append(y_hat)
        predicted_labels = np.array(predicted_labels)
        return predicted_labels

    def evaluate(self, X, y):
        """
        X (n_examples x n_features)
        y (n_examples): gold labels
        """
        # Identical to LinearModel.evaluate()
        y_hat = self.predict(X)
        y_hat = np.argmax(y_hat, axis=1)
        n_correct = (y == y_hat).sum()
        n_possible = y.shape[0]
        return n_correct / n_possible

    def train_epoch(self, X, y, learning_rate=0.001):
        # Encode labels as one-hot vectors.
        one_hot = np.zeros((np.size(y, 0), self.n_classes))
        for i in range(np.size(y, 0)):
            one_hot[i, y[i]] = 1
        y = one_hot
        total_loss = 0
        for x_i, y_i in zip(X, y):
            output, hiddens = self.forward(x_i)
            loss = self.compute_loss(output, y_i)
            total_loss += loss
            grad_weights, grad_biases = self.backward(x_i, y_i, output, hiddens)
            self.update_parameters(grad_weights, grad_biases, eta=learning_rate)
        print(f'Total Loss = {total_loss}')

    def forward(self, x):
        num_layers = len(self.W)

        hiddens = []
        for i in range(num_layers):
            h = x if i == 0 else hiddens[i-1]
```

```
            z = self.W[i].dot(h) + self.b[i]
            if i < num_layers-1:  # Assume the output layer has no activation.
                hiddens.append(np.maximum(0, z))
        output = z
        # For classification this is a vector of logits (label scores).
        # For regression this is a vector of predictions.
        return output, hiddens

    def compute_label_probabilities(self, output):
        # softmax transformation.
        m = max(output)
        probs = np.exp(output - m) / np.sum(np.exp(output - m))
        return probs

    def compute_loss(self, output, y):
        # softmax transformation.
        probs = self.compute_label_probabilities(output)
        loss = -y.dot(np.log(probs))
        return loss

    def predict_label(self, output):
        # The most probable label is also the label with the largest logit.
        y_hat = np.zeros_like(output)
        y_hat[np.argmax(output)] = 1
        return y_hat

    def backward(self, x, y, output, hiddens):
        num_layers = len(self.W)

        # softmax transformation.
        probs = self.compute_label_probabilities(output)
        grad_z = probs - y  # Grad of loss wrt last z.

        grad_weights = []
        grad_biases = []
        for i in range(num_layers-1, -1, -1):
            # Gradient of hidden parameters.
            h = x if i == 0 else hiddens[i-1]
            grad_weights.append(grad_z[:, None].dot(h[:, None].T))
            grad_biases.append(grad_z)

            # Gradient of hidden layer below.
            grad_h = self.W[i].T.dot(grad_z)

            # Gradient of hidden layer below before activation.
            # relu: grad = 0 if z < 0, or grad = 1 if z > 0
            g_der = np.zeros(h.shape[0])
            for i in range(h.shape[0]):
                g_der[i] = 0 if h[i] <= 0 else 1
            grad_z = grad_h * g_der

        grad_weights.reverse()
        grad_biases.reverse()
        return grad_weights, grad_biases

    def update_parameters(self, grad_weights, grad_biases, eta):
        num_layers = len(self.W)
        for i in range(num_layers):
            self.W[i] -= eta*grad_weights[i]
            self.b[i] -= eta*grad_biases[i]
```

Then we train 20 epochs of our model on the training set. Its performance on the validation and test set is reported in the plot of the accuracies as a function of the epoch number in Fig. 4.

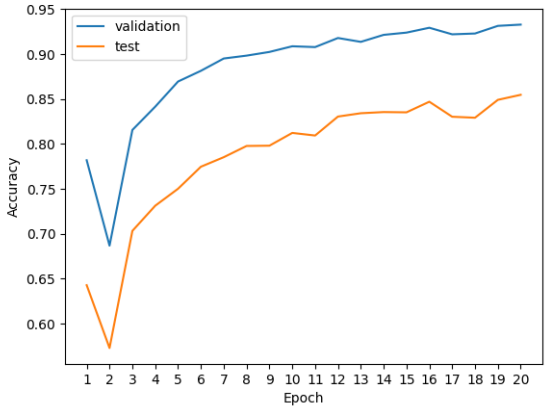


Figure 4: Accuracy of training set and of validation set as a function of the epoch number for MLP.

2 Image classification with an autodiff toolkit

In this question, we implement the gradient backpropagation using a deep learning framework with automatic differentiation: Pytorch.

2.1

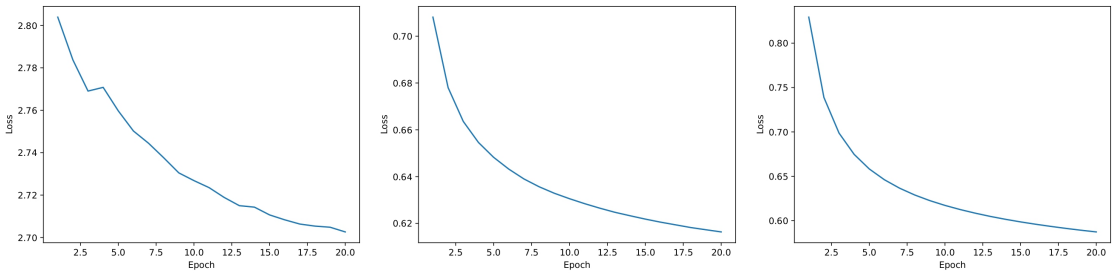
Initially we implement a linear model with logistic regression, using stochastic gradient descent as training algorithm with batch size of 1.

In order to solve it, we implemented the method `train_batch()` and the class `LogisticRegression`'s `__init__()` and `forward()` methods. Our code implementation is reported below.

```
class LogisticRegression(nn.Module):

    def __init__(self, n_classes, n_features, **kwargs):
        """
        n_classes (int)
        n_features (int)

        The __init__ should be used to declare what kind of layers and other
        parameters the module has.
        """
        super(LogisticRegression, self).__init__()
        self.layer = nn.Linear(n_features, n_classes)

    def forward(self, x, **kwargs):
        """
        x (batch_size x n_features): a batch of training examples

        Every subclass of nn.Module needs to have a forward() method. forward()
        describes how the module computes the forward pass. In a log-linear
        model like this, for example, forward() needs to compute the logits
        y = Wx + b, and return y (the nn.CrossEntropyLoss take the
        softmax of y for us). We only need to define the
        forward pass -- this is enough for it to figure out how to do the
        backward pass.
        """
        x = self.layer(x)

        return x


def train_batch(X, y, model, optimizer, criterion, **kwargs):
    """
    X (n_examples x n_features)
    y (n_examples): gold labels
    model: a PyTorch defined model
    optimizer: optimizer used in gradient step
    criterion: loss function
```

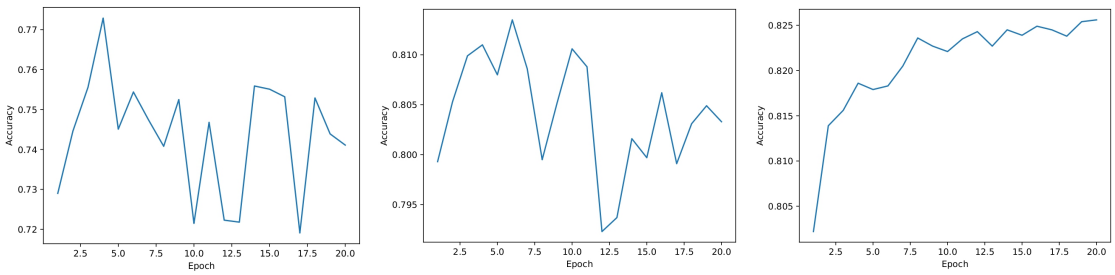
```
    To train a batch, the model needs to predict outputs for X, compute the
    loss between these predictions and the "gold" labels y using the criterion,
    and compute the gradient of the loss with respect to the model parameters.

    This function should return the loss.
    """
    # clear the gradients
    optimizer.zero_grad()
    # compute the model output
    yhat = model(X)
    # calculate loss
    loss = criterion(yhat, y)
    # credit assignment
    loss.backward()
    # update model weights
    optimizer.step()

    return loss.item()
```

We trained 20 epochs of the model and tune the learning rate for the validation data, using the following values: $\{0.001, 0.01, 0.1\}$. The plots of the training loss and the validation accuracy, both as a function of the epoch number, are represented for each leaning rate in Fig. 5 and in Fig. 6.



(a): Learning rate = 0.1. (b): Learning rate = 0.01. (c): Learning rate = 0.001.

Figure 5: Training loss as a function of the epoch number for different values of learning rate.



(a): Learning rate = 0.1. (b): Learning rate = 0.01. (c): Learning rate = 0.001.

Figure 6: Validation accuracy as a function of the epoch number for different values of learning rate.

The results are reported in Table 1.

Table 1: Results of the model for tuning the learning rate.

Learning Rate	Final validation accuracy	Final accuracy on the test set
0.1	0.7411	0.6133
0.01	0.8033	0.6806
0.001	**0.8256**	**0.7019**

The best configuration in terms of final validation accuracy is the **learning rate** equal to **0.001**. In this case the final accuracy on the test set is equal to 0.7019.

2.2

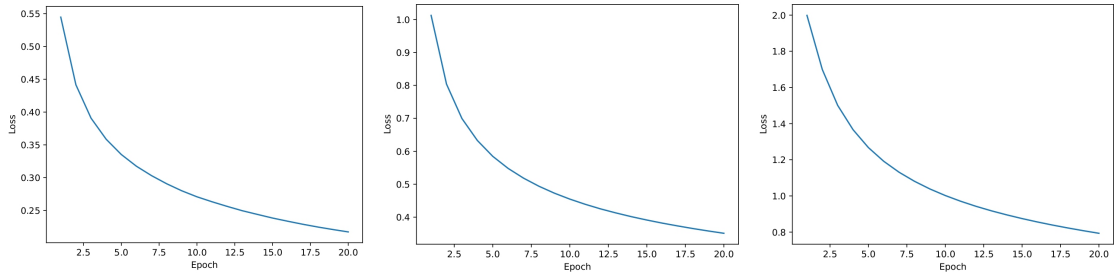
We implement a feed-forward neural network using dropout regularization. In order to solve it, we implement the class `FeedforwardNetwork`'s `__init__()` and `forward()` methods. Our code implementation is reported below.

```
class FeedforwardNetwork(nn.Module):
    def __init__(
            self, n_classes, n_features, hidden_size, layers,
            activation_type, dropout, **kwargs):
        """
        n_classes (int)
        n_features (int)
        hidden_size (int)
        layers (int)
        activation_type (str)
        dropout (float): dropout probability

        As in logistic regression, the __init__ here defines a bunch of
        attributes that each FeedforwardNetwork instance has. Note that nn
        includes modules for several activation functions and dropout as well.
        """
        super(FeedforwardNetwork, self).__init__()
        self.layers = layers

        if (activation_type == 'tanh'):
            self.activation = nn.Tanh()
        else:
            self.activation = nn.ReLU()

        self.model = nn.Sequential()

        if layers == 0:
            self.model.append(nn.Linear(n_features, n_classes))
        else:
            self.model.append(nn.Linear(n_features, hidden_size))
            self.model.append(self.activation)
            self.model.append(nn.Dropout(dropout))

            for i in range(self.layers - 1):
                self.model.append(nn.Linear(hidden_size, hidden_size))
                self.model.append(self.activation)
                self.model.append(nn.Dropout(dropout))

            self.model.append(nn.Linear(hidden_size, n_classes))

    def forward(self, x, **kwargs):
        """
        x (batch_size x n_features): a batch of training examples

        This method needs to perform all the computation needed to compute
        the output logits from x. This will include using various hidden
        layers, pointwise nonlinear functions, and dropout.
        """
        return self.model(x)
```

The default values are: number of epochs = 20; learning rate = 0.01; hidden size = 100; dropout = 0.3; batch size = 16; activation = ReLU; optimizer = SGD.

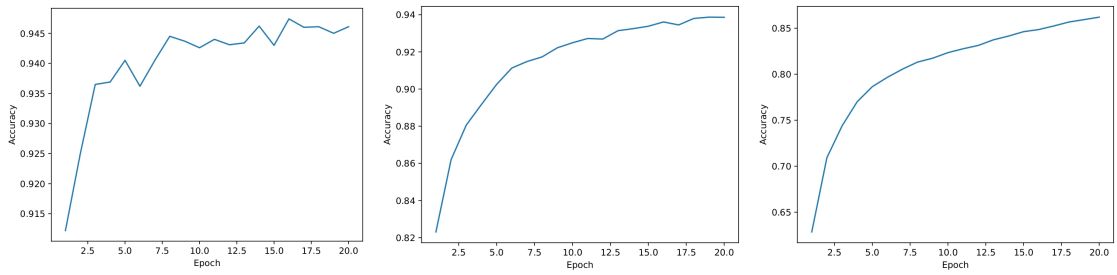
We tune each of the hyperparameters {learning rate, hidden size, dropout probability, activation function} while leaving the remaining at their default value.

We trained the model for number of epochs = 20, hidden size = 100, dropout = 0.3, batch size = 16, activation = ReLU, optimizer = SGD and tune the learning rate for the validation data, using the following values: $\{0.001, 0.01, 0.1\}$. The plots of the training loss and the validation accuracy, both as a function of the epoch number, are represented for each leaning rate in Fig. 7 and in Fig. 8.



(a): Learning rate = 0.1. (b): Learning rate = 0.01. (c): Learning rate = 0.001.

Figure 7: Training loss as a function of the epoch number for different values of learning rate.



(a): Learning rate = 0.1. (b): Learning rate = 0.01. (c): Learning rate = 0.001.

Figure 8: Validation accuracy as a function of the epoch number for different values of learning rate.

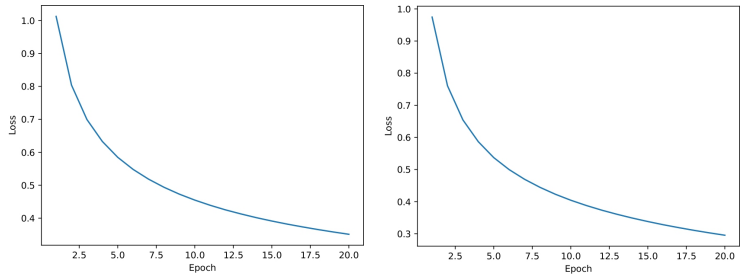
The results are reported in Table 2.

Table 2: Results of the model for tuning the learning rate.

Number of Epochs	Batch Size	Optimizer	Learning Rate	Hidden Size	Dropout	Activation	Final validation accuracy	Final accuracy on the test set
20	**16**	**SGD**	**0.1**	**100**	**0.3**	**ReLU**	**0.9461**	**0.8696**
20	16	SGD	0.01	100	0.3	ReLU	0.9386	0.8636
20	16	SGD	0.001	100	0.3	ReLU	0.8621	0.7410

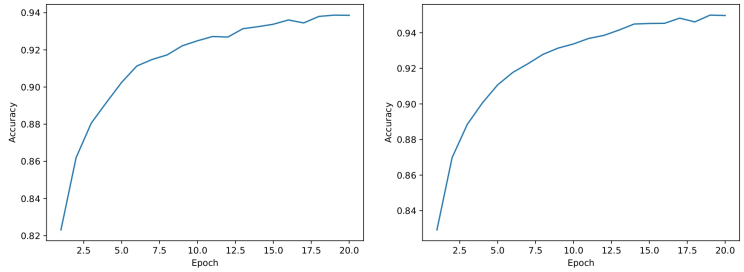
So in terms of final validation accuracy, with the remaining hyperparameters at their default value, the **tuned learning rate** is equal to **0.1** and the final accuracy on the test set is equal to 0.8696.

Next, we trained the model for number of epochs = 20, learning rate = 0.01, dropout = 0.3, batch size = 16, activation = ReLU, optimizer = SGD and tune the hidden size for the validation data, using the following values: $\{100, 200\}$. The plots of the training loss and the validation accuracy, both as a function of the epoch number, are represented for each hidden size in Fig. 9 and in Fig. 10.



(a): Hidden size = 100. (b): Hidden size = 200.

Figure 9: Training loss as a function of the epoch number for different values of hidden size.



(a): Hidden size = 100. (b): Hidden size = 200.

Figure 10: Validation accuracy as a function of the epoch number for different values of hidden size.

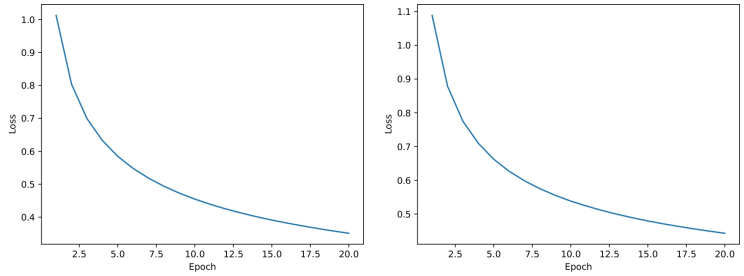
The results are reported in Table 3.

Table 3: Results of the model for tuning the hidden size.

Number of Epochs	Batch Size	Optimizer	Learning Rate	Hidden Size	Dropout	Activation	Final validation accuracy	Final accuracy on the test set
20	16	SGD	0.01	100	0.3	ReLU	0.9386	0.8636
20	**16**	**SGD**	**0.01**	**200**	**0.3**	**ReLU**	**0.9497**	**0.8803**

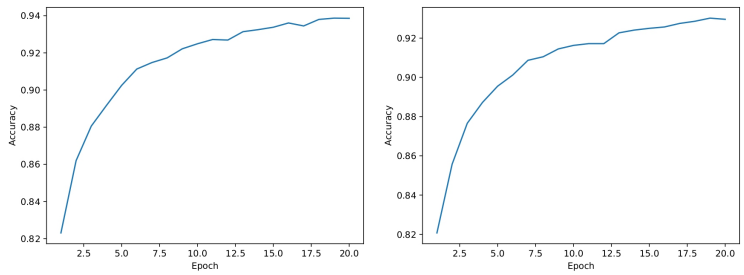
So in terms of final validation accuracy, with the remaining hyperparameters at their default value, the **tuned hidden size** is equal to **200** and the final accuracy on the test set is equal to 0.8803.

Then, we trained the model for number of epochs = 20, learning rate = 0.01, hidden size = 100, batch size = 16, activation = ReLU, optimizer = SGD and tune the dropout probability for the validation data, using the following values: $\{0.3, 0.5\}$. The plots of the training loss and the validation accuracy, both as a function of the epoch number, are represented for each dropout probability in Fig. 11 and in Fig. 12.



(a): Dropout probability = 0.3. (b): Dropout probability = 0.5.

Figure 11: Training loss as a function of the epoch number for different values of dropout probability.



(a): Dropout probability = 0.3. (b): Dropout probability = 0.5.

Figure 12: Validation accuracy as a function of the epoch number for different values of dropout probability.

The results are reported in Table 4.

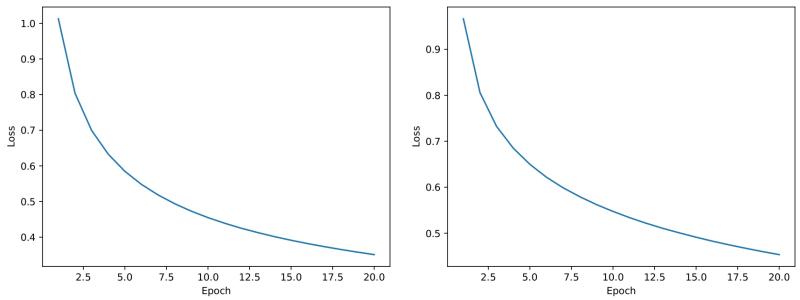
So in terms of final validation accuracy, with the remaining hyperparameters at their default value, the **tuned dropout probability** is equal to **0.3** and the final accuracy on the test set is equal to 0.8636.

Then, we trained the model for number of epochs = 20, learning rate = 0.01, hidden size = 100, dropout = 0.3, batch size = 16, optimizer = SGD and tune the activation function for the validation data, using the following values: {ReLU, tanh}.

Table 4: Results of the model for tuning the dropout probability.

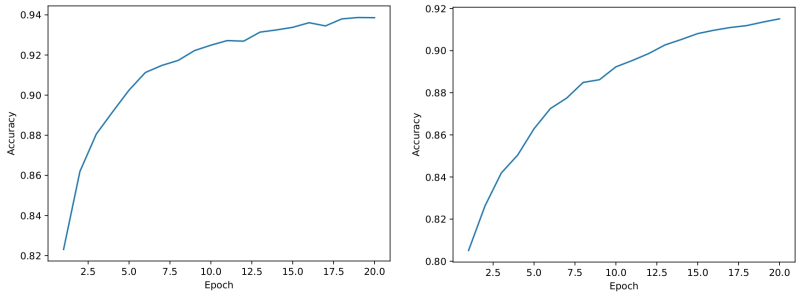
Number of Epochs	Batch Size	Optimizer	Learning Rate	Hidden Size	Dropout	Activation	Final validation accuracy	Final accuracy on the test set
20	**16**	**SGD**	**0.01**	**100**	**0.3**	**ReLU**	**0.9386**	**0.8636**
20	16	SGD	0.01	100	0.5	ReLU	0.9296	0.8414

The plots of the training loss and the validation accuracy, both as a function of the epoch number, are represented for each activation function in Fig. 13 and in Fig. 14.



(a): Activation function = ReLU. (b): Activation function = tanh.

Figure 13: Training loss as a function of the epoch number for different activation function.



(a): Activation function = ReLU. (b): Activation function = tanh.

Figure 14: Validation accuracy as a function of the epoch number for different activation function.

The results are reported in Table 5.

Table 5: Results of the model for tuning the activation function.

Number of Epochs	Batch Size	Optimizer	Learning Rate	Hidden Size	Dropout	Activation	Final validation accuracy	Final accuracy on the test set
20	**16**	**SGD**	**0.01**	**100**	**0.3**	**ReLU**	**0.9386**	**0.8636**
20	16	SGD	0.01	100	0.3	tanh	0.9151	0.8210

So in terms of final validation accuracy, with the remaining hyperparameters at their default value, the **tuned activation function** is equal to **ReLU** and the final accuracy on the test set is equal to 0.8636.

Finally, we trained the model with the best configuration, i.e, for number of epochs = 20, batch size = 16, optimizer = SGD and with the tuned hyperparameters learning rate = 0.1, hidden size = 200, dropout probability = 0.3 and activation function = ReLU. The plots of the training loss and the validation accuracy, both as a function of the epoch number, are represented in Fig. 15 and in Fig. 16.

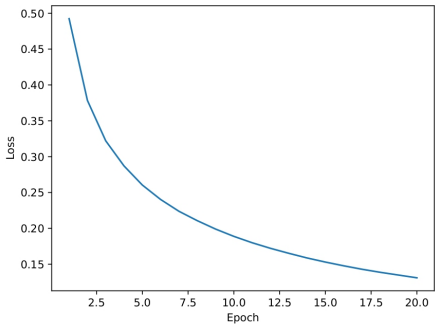


Figure 15: Training loss as a function of the epoch number for the best configuration.

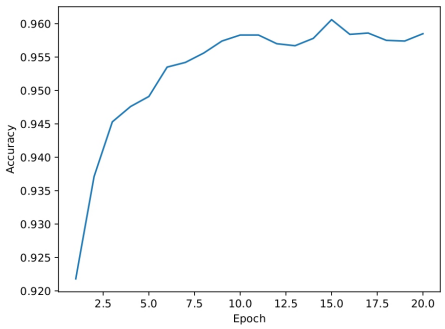


Figure 16: Validation accuracy as a function of the epoch number for the best configuration.

In this case, the final validation accuracy is equal to 0.9585 and the final accuracy on the test set is equal to 0.8958. Furthermore, if we train until epoch 15, the final validation accuracy is equal to 0.9606 and the final accuracy on the test set is equal to 0.8988, so in this case we should create an Early Stopping monitor that will stop training when the validation loss is not improving, over the best configuration, in a few consecutive epochs (e.g. 10 epochs of patience).

We also apply another process in order to tune each of the hyperparameters {learning rate, hidden size, dropout probability, activation function} where we try all the possible combinations between them. The obtained results are reported in Table 6.

Table 6: Results for all combinations of tuning hyperparameters.

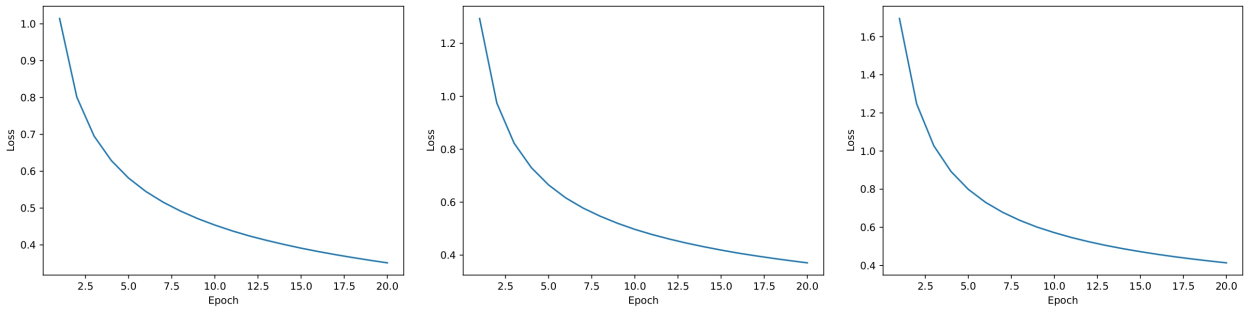
Number of Epochs	Batch Size	Optimizer	Learning Rate	Hidden Size	Dropout	Activation	Final validation accuracy	Final accuracy on the test set
20	16	SGD	0.001	100	0.3	ReLU	0.8621	0.7410
20	16	SGD	0.001	100	0.3	tanh	0.8278	0.6978
20	16	SGD	0.001	100	0.5	ReLU	0.8582	0.7326
20	16	SGD	0.001	100	0.5	tanh	0.8222	0.6896
20	16	SGD	0.001	200	0.3	ReLU	0.8699	0.7495
20	16	SGD	0.001	200	0.3	tanh	0.8250	0.6981
20	16	SGD	0.001	200	0.5	ReLU	0.8678	0.7459
20	16	SGD	0.001	200	0.5	tanh	0.8220	0.6936
20	16	SGD	0.01	100	0.3	ReLU	0.9386	0.8636
20	16	SGD	0.01	100	0.3	tanh	0.9151	0.8210
20	16	SGD	0.01	100	0.5	ReLU	0.9296	0.8414
20	16	SGD	0.01	100	0.5	tanh	0.8882	0.7776
20	16	SGD	0.01	200	0.3	ReLU	0.9497	0.8803
20	16	SGD	0.01	200	0.3	tanh	0.9237	0.8319
20	16	SGD	0.01	200	0.5	ReLU	0.9465	0.8702
20	16	SGD	0.01	200	0.5	tanh	0.8985	0.7934
20	16	SGD	0.1	100	0.3	ReLU	0.9461	0.8696
20	16	SGD	0.1	100	0.3	tanh	0.9314	0.8565
20	16	SGD	0.1	100	0.5	ReLU	0.9351	0.8505
20	16	SGD	0.1	100	0.5	tanh	0.9126	0.8245
20	**16**	**SGD**	**0.1**	**200**	**0.3**	**ReLU**	**0.9585**	**0.8958**
20	16	SGD	0.1	200	0.3	tanh	0.9495	0.8849
20	16	SGD	0.1	200	0.5	ReLU	0.9535	0.8882
20	16	SGD	0.1	200	0.5	tanh	0.9326	0.8490

In this case, the best configuration is also for number of epochs = 20, batch size = 16, optimizer = SGD and with the tuned hyperparameters learning rate = 0.1, hidden size = 200, dropout probability = 0.3 and activation function = ReLU. The final validation accuracy is equal to 0.9585 and the final accuracy on the test set is equal to 0.8958.

2.3

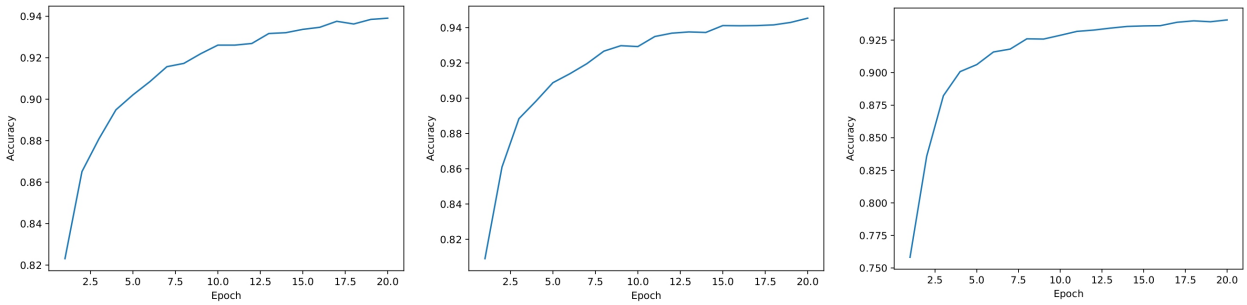
Using the feed-forward neural network implemented in the last question we tested the model with 2 and 3 layers using all of the hyperparameters with their default values (as in Table 1 of [2]).

So, we trained the model for number of epochs = 20, batch size = 16, optimizer = SGD, learning rate = 0.01, hidden size = 100, dropout probability = 0.3 and activation function = ReLU and tune the number of hidden layers for the validation data, using the following values: $\{1, 2, 3\}$. The plots of the training loss and the validation accuracy, both as a function of the epoch number, are represented for each number of hidden layers in Fig. 17 and in Fig. 18.



(a): Number of hidden layers = 1. (b): Number of hidden layers = 2. (c): Number of hidden layers = 3.

Figure 17: Training loss as a function of the epoch number for different values of number of hidden layers, for the remaining hyperparameters with their default values.



(a): Number of hidden layers = 1. (b): Number of hidden layers = 2. (c): Number of hidden layers = 3.

Figure 18: Validation accuracy as a function of the epoch number for different values of number of hidden layers, for the remaining hyperparameters with their default values.

The results are reported in Table 7.

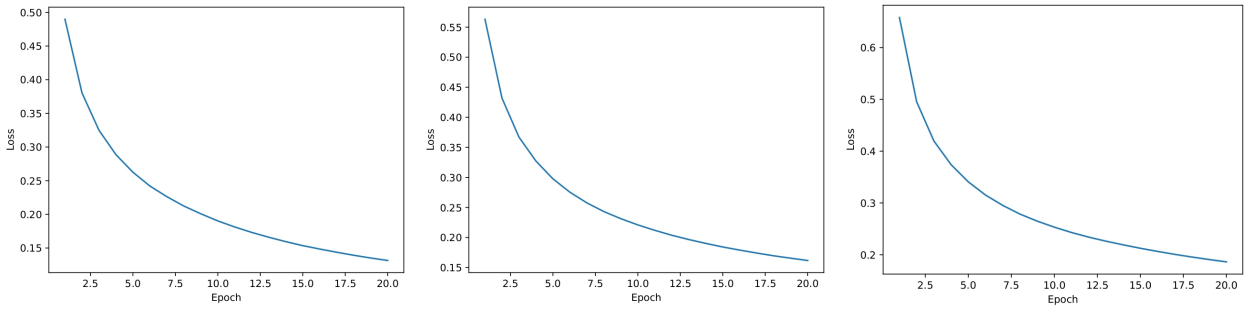
Table 7: Results of the model for tuning the number of layers, for the remaining hyperparameters with their default values.

Number of Epochs	Batch Size	Optimizer	Learning Rate	Hidden Size	Dropout	Activation	Number of layers	Final validation accuracy	Final accuracy on the test set
20	16	SGD	0.001	100	0.3	ReLU	1	0.9391	0.8600
20	**16**	**SGD**	**0.001**	**100**	**0.3**	**ReLU**	**2**	**0.9454**	**0.8724**
20	16	SGD	0.001	100	0.3	ReLU	3	0.9405	0.8616

So in terms of final validation accuracy, with the remaining hyperparameters as with their default values, the best configuration is the **number of hidden layers** equal to **2** and the final accuracy on the test set is equal to 0.8724.

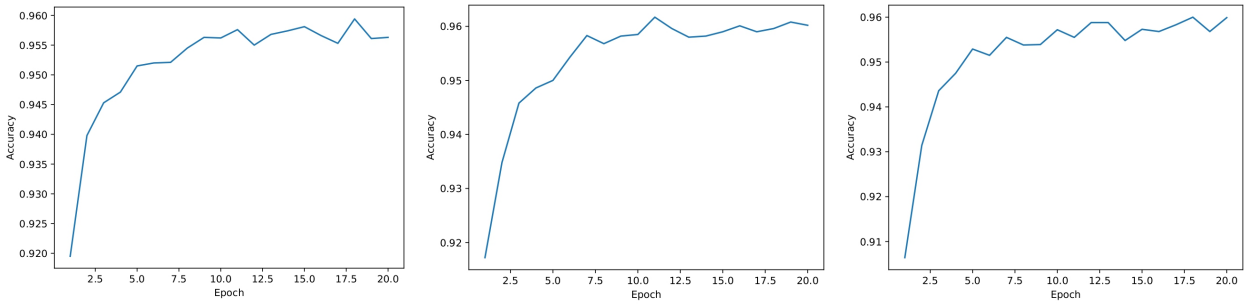
Then, we perform the same process using all of the hyperparameters as the best configuration we previously found in 2.2.

So, we trained the model for number of epochs = 20, batch size = 16, optimizer = SGD and with the tuned hyperparameters learning rate = 0.1, hidden size = 200, dropout probability = 0.3 and activation function = ReLU and tune the number of hidden layers for the validation data, using the following values: $\{1, 2, 3\}$. The plots of the training loss and the validation accuracy, both as a function of the epoch number, are represented for each number of hidden layers in Fig. 19 and in Fig. 20.



(a): Number of hidden layers = 1. (b): Number of hidden layers = 2. (c): Number of hidden layers = 3.

Figure 19: Training loss as a function of the epoch number for different values of number of hidden layers, for the remaining hyperparameters at their best configuration obtained in 2.2.



(a): Number of hidden layers = 1. (b): Number of hidden layers = 2. (c): Number of hidden layers = 3.

Figure 20: Validation accuracy as a function of the epoch number for different values of number of hidden layers, for the remaining hyperparameters at their best configuration obtained in 2.2.

The results are reported in Table 8.

Table 8: Results of the model for tuning the number of layers, for the remaining hyperparameters at their best configuration obtained in 2.2.

Number of Epochs	Batch Size	Optimizer	Learning Rate	Hidden Size	Dropout	Activation	Number of layers	Final validation accuracy	Final accuracy on the test set
20	16	SGD	0.1	200	0.3	ReLU	1	0.9563	0.8991
20	**16**	**SGD**	**0.1**	**200**	**0.3**	**ReLU**	**2**	**0.9602**	**0.9008**
20	16	SGD	0.1	200	0.3	ReLU	3	0.9599	0.8984

So in terms of final validation accuracy, with the remaining hyperparameters at their best configuration obtained in 2.2, the best configuration is the **number of hidden layers** equal to **2** and the final accuracy on the test set is equal to 0.9008.

In the real world, it is necessary to do hyperparameter tuning for the different network architectures, but it is not required for this assignment.

3 Multi-layer perceptron with quadratic activations

In this exercise, we consider a feed-forward neural network with a single hidden layer and a quadratic activation function, $g(z) = z^2$. This choice of activation, unlike other popular activation functions such as tanh, sigmoid, or ReLU, can be tackled as a linear model via a reparametrization. We assume a univariate regression task, where the predicted output $\hat{y} \in \mathbb{R}$ is given by $\hat{y} = v^T h$, where $h \in \mathbb{R}^K$ are internal representations, given by $h = g(Wx)$, $x \in \mathbb{R}^D$ is a vector of input variables, and $\Theta = (W, v) \in \mathbb{R}^{K \times D} \times \mathbb{R}^K$ are the model parameters.

3.1

We show in (27), (28), (29) and (30) that $h = A_\Theta \phi(x)$ for a certain feature transformation $\phi : \mathbb{R}^D \to \mathbb{R}^{\frac{D(D+1)}{2}}$ independent of Θ and with $A_\Theta \in \mathbb{R}^{K \times \frac{D(D+1)}{2}}$. That is, h is a linear transformation of $\phi(x)$.

$$Wx = \begin{bmatrix} w_{11} & \dots & w_{1D} \\ \vdots & \ddots & \vdots \\ w_{K1} & \dots & w_{KD} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_D \end{bmatrix} = \begin{bmatrix} w_{11}x_1 + w_{12}x_2 + \dots + w_{1D}x_D \\ \vdots \\ w_{K1}x_1 + w_{K2}x_2 + \dots + w_{KD}x_D \end{bmatrix} \tag{27}$$

$$h = g(Wx) = \begin{bmatrix} (w_{11}x_1 + w_{12}x_2 + \cdots + w_{1D}x_D)^2 \\ \vdots \\ (w_{K1}x_1 + w_{K2}x_2 + \cdots + w_{KD}x_D)^2 \end{bmatrix} \tag{28}$$

$$h = \begin{bmatrix} w_{11}^2x_1^2 + w_{12}^2x_2^2 + \cdots + w_{1D}^2x_D^2 + 2w_{11}w_{12}x_1x_2 + \cdots + 2w_{11}w_{1D}x_1x_D + \cdots + 2w_{1(D-1)}w_{1D}x_{D-1}x_D \\ \vdots \\ w_{K1}^2x_1^2 + w_{K2}^2x_2^2 + \cdots + w_{KD}^2x_D^2 + 2w_{K1}w_{K2}x_1x_2 + \cdots + 2w_{K1}w_{KD}x_1x_D + \cdots + 2w_{K(D-1)}w_{KD}x_{D-1}x_D \end{bmatrix} \tag{29}$$

$$h = \begin{bmatrix} w_{11}^2 & w_{12}^2 & \dots & w_{1D}^2 & w_{11}w_{12} & \dots & w_{11}w_{1D} & \dots & w_{1(D-1)}w_{1D} \\ \vdots & \vdots & & \vdots & \vdots & & \vdots & & \vdots \\ w_{K1}^2 & w_{K2}^2 & \dots & w_{KD}^2 & w_{K1}w_{K2} & \dots & w_{K1}w_{KD} & \dots & w_{K(D-1)}w_{KD} \end{bmatrix} \begin{bmatrix} x_1^2 \\ x_2^2 \\ \vdots \\ x_D^2 \\ 2x_1x_2 \\ \vdots \\ 2x_1x_D \\ \vdots \\ 2x_{D-1}x_D \end{bmatrix} = A_\Theta\phi(x) \tag{30}$$

The mapping ϕ and the matrix A_Θ are given by (31) and (32).

$$\phi(x) = \begin{bmatrix} x_1^2 \\ x_2^2 \\ \vdots \\ x_D^2 \\ 2x_1x_2 \\ \vdots \\ 2x_1x_D \\ \vdots \\ 2x_{D-1}x_D \end{bmatrix} \tag{31}$$

$$A_\Theta = \begin{bmatrix} w_{11}^2 & w_{12}^2 & \dots & w_{1D}^2 & w_{11}w_{12} & \dots & w_{11}w_{1D} & \dots & w_{1(D-1)}w_{1D} \\ \vdots & \vdots & & \vdots & \vdots & & \vdots & & \vdots \\ w_{K1}^2 & w_{K2}^2 & \dots & w_{KD}^2 & w_{K1}w_{K2} & \dots & w_{K1}w_{KD} & \dots & w_{K(D-1)}w_{KD} \end{bmatrix} \tag{32}$$

Every row of A_Θ has the number of elements given by (33), as expected.

$$D + \binom{D}{2} = D + \frac{D!}{(D-2)!2!} = D + \frac{D(D-1)(D-2)!}{2(D-2)!} = \frac{D(D+1)}{2} \tag{33}$$

3.2

Based on the previous claim, $\hat{y}$ is also a linear transformation of $\phi(x)$ because by (34) we can write $\hat{y}(x; c_\Theta) = c_\Theta^T\phi(x)$ for some $c_\Theta \in \mathbb{R}^{\frac{D(D+1)}{2}}$.

$$\hat{y} = v^Th = v^TA_\Theta\phi(x) = c_\Theta^T\phi(x)\ , \quad c_\Theta = A_\Theta^Tv \tag{34}$$

So, this is not a linear model in terms of the original parameters Θ, since $\phi(x)$ is independent of Θ, and c_Θ is not a linear function of Θ because each element of c_Θ is a linear combination of the elements in one of the columns of A_Θ and all entries of A_Θ are not linear in terms of W (they are quadratic in terms of W), so not linear in terms of Θ.

3.3

Assuming $K \geq D$, for any real vector $c \in \mathbb{R}^{\frac{D(D+1)}{2}}$ we prove that there is a choice of the original parameters $\Theta = (W, v)$ such that $c_\Theta = c$. In this case c_Θ is given by (35) and (36).

$$c_\Theta = A_\Theta^T v = \begin{bmatrix} w_{11}^2 & w_{21}^2 & \dots & w_{K1}^2 \\ w_{12}^2 & w_{22}^2 & \dots & w_{K2}^2 \\ \vdots & \vdots & & \vdots \\ w_{1D}^2 & w_{2D}^2 & \dots & w_{KD}^2 \\ w_{11}w_{12} & w_{21}w_{22} & \dots & w_{K1}w_{K2} \\ \vdots & \vdots & & \vdots \\ w_{11}w_{1D} & w_{21}w_{2D} & \dots & w_{K1}w_{KD} \\ \vdots & \vdots & & \vdots \\ w_{1(D-1)}w_{1D} & w_{2(D-1)}w_{2D} & \dots & w_{K(D-1)}w_{KD} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_K \end{bmatrix} \tag{35}$$

$$c_\Theta = \begin{bmatrix} w_{11}^2 v_1 + w_{21}^2 v_2 + \dots + w_{K1}^2 v_K \\ w_{12}^2 v_1 + w_{22}^2 v_2 + \dots + w_{K2}^2 v_K \\ \vdots \\ w_{1D}^2 v_1 + w_{2D}^2 v_2 + \dots + w_{KD}^2 \\ w_{11}w_{12}v_1 + w_{21}w_{22}v_2 + \dots + w_{K1}w_{K2}v_K \\ \vdots \\ w_{11}w_{1D}v_1 + w_{21}w_{2D}v_2 + \dots + w_{K1}w_{KD}v_K \\ \vdots \\ w_{1(D-1)}w_{1D}v_1 + w_{2(D-1)}w_{2D}v_2 + \dots + w_{K(D-1)}w_{KD}v_K \end{bmatrix} \tag{36}$$

We can also define a matrix C given by 37, where $\text{rank}(C)$ refers to the number of linearly independent rows or columns in the matrix C. Then, c_Θ is given by 38, where $\text{vech}(C)$ is a vector which the elements are the elements in the lower triangular part of C.

$$\{C = W^T \text{Diag}(v) W \mid W \in \mathbb{R}^{K\times D},\ v \in \mathbb{R}^K\} = \{C \in \mathbb{R}^{D\times D} \mid \text{rank}(C) \leq K\} \tag{37}$$

$$c_\Theta = \text{vech}(C) \tag{38}$$

This means that assuming $K \geq D$ this set equals $\mathbb{R}^{D\times D}$, from which we can obtain any $c_\Theta \in \mathbb{R}^{\frac{D(D+1)}{2}}$. So this is a linear model in terms of c_Θ because $\hat{y}(x; c_\Theta) = c_\Theta^T \phi(x)$, and we can equivalently parameterize the model with c_Θ instead of Θ.

On the other hand, if $K < D$, $\text{rank}(C)$ may be greater than K, and in that case we can't recuperate $\Theta = (W, v)$. So, for $K < D$ an equivalent parameterization may not exist.

3.4

Supposing we are given training data $\mathcal{D} = \{(x_n, y_n)\}_{n=1}^N$ with $N > \frac{D(D+1)}{2}$. We want to minimize the squared loss given by (39).

$$L(c_\Theta; \mathcal{D}) = \frac{1}{2}\sum_{n=1}^N \big(\hat{y_n}(x_n; c_\Theta) - y_n\big)^2 = \frac{1}{2}\sum_{n=1}^N \big(c_\Theta^T \phi(x_n) - y_n\big)^2 = \frac{1}{2}\sum_{n=1}^N \big(\phi^T(x_n) c_\Theta - y_n\big)^2 = \frac{1}{2}\sum_{n=1}^N L_n(c_\Theta; \mathcal{D}) \tag{39}$$

A function $f : \mathbb{R}^n \to \mathbb{R}$ is convex if $f((1-\alpha) + \alpha y) \leq (1-\alpha) f(x) + \alpha f(y)$ for all $x, y \in \mathbb{R}^n$ and $0 \leq \alpha \leq 1$. Let f be C^2, i.e., $\nabla^2 f$ is continuous. Then, f is convex if and only if $\nabla^2 f \geq 0$.

Using the theorem that states that if $t_i \geq 0$ and $g_i : \mathbb{R}^n \to \mathbb{R}$ is convex for $i = 1, 2, ..., m$, then $t_1 g_1 + t_2 g_2 + ... + t_m f_m : \mathbb{R}^n \to \mathbb{R}$ is convex, we conclude that $L(c_\Theta; \mathcal{D})$ is convex if all $L_n(c_\Theta; \mathcal{D})$ are convex because $L(c_\Theta; \mathcal{D})$ is a nonnegative linear combination of $L_1, L_2, \dots, L_n$.

$L_n(c_\Theta; \mathcal{D}) = g_n \circ h_n$ with $g_n(z) = z^2$ and $h_n(c_\Theta; \mathcal{D}) = \phi^T(x_n) c_\Theta - y_n$. So, as g_n is convex and h_n is an affine map, then $L_n(c_\Theta; \mathcal{D})$ is convex for all n, which implies that $L(c_\Theta; \mathcal{D})$ is convex. Therefore, $\nabla^2 L(c_\Theta; \mathcal{D}) \geq 0$, i.e, the Hessian is semi-positive definite.

If $L(c_\Theta; \mathcal{D})$ is convex and c_Θ is a local minimizer, then it is also a global minimizer. So in this case, being unique the solution of $\nabla L(c_\Theta; \mathcal{D}) = 0$ (given by (40), (41) and (42)) and knowing that $L(c_\Theta; \mathcal{D})$ is convex, there is a global minimum. The closed form solution for c_Θ is given by (42) if $\sum_{n=1}^N \phi(x_n)\phi^T(x_n)$ is invertible (next, we will proof that for this case).

$$\nabla L(c_\Theta; \mathcal{D}) = \sum_{n=1}^N \phi(x_n)\big(\phi^T(x_n) c_\Theta - y_n\big) \tag{40}$$

$$\nabla L(c_\Theta; \mathcal{D}) = 0 \Leftrightarrow \sum_{n=1}^N \phi(x_n)\big(\phi^T(x_n) c_\Theta - y_n\big) = 0 \Leftrightarrow \sum_{n=1}^N \phi(x_n)\phi^T(x_n) c_\Theta = \sum_{n=1}^N \phi(x_n) y_n \tag{41}$$

$$\nabla L(c_\Theta; \mathcal{D}) = 0 \Leftrightarrow c_\Theta = \left[\sum_{n=1}^{N} \phi(x_n)\phi^T(x_n)\right]^{-1} \sum_{n=1}^{N} \phi(x_n) y_n \tag{42}$$

In order to proof that $\sum_{n=1}^{N} \phi(x_n)\phi^T(x_n)$ is invertible for $N > \frac{D(D+1)}{2}$, we define the matrix $X \in \mathbb{R}^{N \times \frac{D(D+1)}{2}}$ such that X have $\phi(x_n)$ as rows and we assume that X has full column-rank.

$$\sum_{n=1}^{N} \phi(x_n)\phi^T(x_n) = X^T X \tag{43}$$

The columns of $X^T X$ are linearly independent if the equation $X^T X u = 0$ has only the trivial solution, for $u \in \mathbb{R}^{\frac{D(D+1)}{2}}$.

If $Xu = 0$, then $X^T X u = 0$, because $X^T X u = 0 \Leftrightarrow u^T X^T X u = 0 \Leftrightarrow \|Xu\|_2^2 = 0 \Leftrightarrow Xu = 0$.

If $N < \frac{D(D+1)}{2}$, then $\exists u \in \mathbb{R}^{\frac{D(D+1)}{2}}$, $u \neq 0$, such that $Xu = 0$, and so $X^T X u = 0$, which means that $X^T X$ is not invertible.

If $N \geq \frac{D(D+1)}{2}$, then there are $\frac{D(D+1)}{2}$ linearly independent columns of $X^T X$ (X has full column-rank), so $X^T X$ is invertible.

Therefore, it is proved that $X^T X = \sum_{n=1}^{N} \phi(x_n)\phi^T(x_n)$ is invertible if $N > \frac{D(D+1)}{2}$.

Let $y = (y_1, \ldots, y_N)$. Our problem is a least squares problem where we want to minimize $\frac{1}{2}\|Xc_\Theta - y\|_F^2$ with respect to c_Θ. That way, the closed form solution for c_Θ (global minimum) is also given by 44.

$$c_\Theta = \left(X^T X\right)^{-1} X^T y \tag{44}$$

Global minimization is usually intractable for feedforward neural networks because neural networks embrace non-convexity and local minima. For example, gradient descent will eventually converge to a stationary point of the function, regardless of convexity. If the function is convex, this will be a global minimum, but if not, it could be a local minimum or even a saddle point.

The global minimum represents the optimal solution, but fairly recent research on neural-network loss surfaces suggests that high-complexity networks may actually benefit from local minima, because a network that finds the global minimum will be overtrained and therefore will be less effective when processing new input samples.

On the other hand, in the context of real neural-network applications, saddle points in the error surface might be a serious concern to successful training.

Our problem is special because we can write $\hat{y}(x; c_\Theta) = c_\Theta^T \phi(x)$ which leads to $L(c_\Theta; \mathcal{D})$ being convex. There is a unique local minimizer, which is also the unique global minimizer. So, since feedforward neural networks find a local minimizer, then in this case, the solution is the global minimizer.

References

[1] Deep Learning Slides - Slides of Theoretical Classes 2022/2023, 1st Semester (MEEC), by A. Martins, F. Melo, M. Figueiredo;

[2] 1st Semester 2022/2023, Deep Learning (IST, 2022-23), Homework 1, by André Martins, Ben Peters, Margarida Campos;

[3] Deep Learning Practical Lectures - 2022/2023, 1st Semester (MEEC);

[4] Towards Data Science: "Multilayer Perceptron Explained with a Real-Life Example and Python Code: Sentiment Analysis". Available: `https://towardsdatascience.com/multilayer-perceptron-explained-with-a-real-life-example-and-python-code-sentiment-analysis-cb408ee93141`;

[5] All About Circuits: "Understanding Local Minima in Neural-Network Training". [Online]. Available: `https://www.allaboutcircuits.com/technical-articles/understanding-local-minima-in-neural-network-training/`;