NeuralNetworks_professor

October 27, 2025

1 Introduction to Neural Networks and Pytorch

```
Notebook version: 0.5. (Oct 27, 2025)

Authors: Jerónimo Arenas García (jarenas@ing.uc3m.es)
         Jesús Cid-Sueiro (jcid@tsc.uc3m.es)
```

[1]:
```
# from IPython.core.display import HTML
# HTML("""
# <style>
# body {
#   counter-reset: section subsection;
# }
# h2 {
#   counter-reset: subsection;
# }
# h2:before {
#     counter-increment: section;
#     content: "Section " counter(section) ". ";
# }
# h3:before {
#     counter-increment: subsection;
#     content: counter(section) "." counter(subsection) " ";
# }
# </style>
# """)
```

```
Changes: v.0.1. (Nov 14, 2020) - First version
         v.0.2. (Nov 5, 2021) - Structuring code, revisiting formulation
         v.0.3. (Nov, 1, 2022) - Revisiting text.
         v.0.4. (Nov, 4, 2024) - General notebook updates
         v.0.5. (Oct 27, 2025)

Pending changes:
    Add an example with dropout
    Add theory about CNNs
    Define some functions to simplify code cells
```

[2]:
```
import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline

size = 14
params = {'legend.fontsize': 'Large',
          'axes.labelsize': size,
          'axes.titlesize': size,
          'xtick.labelsize': size*0.75,
          'ytick.labelsize': size*0.75}
plt.rcParams.update(params)
```

1.1 1. Introduction and purpose of this Notebook

1.1.1 1.1. About Neural Networks

- Neural Networks (NN) have become the state of the art for many machine learning problems
  - Natural Language Processing
  - Computer Vision
  - Image Recognition
- They are in widespread use for many applications, e.g.,
  - Language translation
  - Automatic speech recognition (Hey Siri! DNN overview)
  - Autonomous navigation (Facebook Robot Autonomous 3D Navigation)
  - Automatic plate recognition

Feed Forward Neural Networks have been around since 1960 but only recently (last 10-15 years) have they met their expectations, and improve other machine learning algorithms

- Computation resources are now available at large scale
- Cloud Computing (AWS, Azure)
- From MultiLayer Perceptrons to Deep Learning
- Big Data sets
- This has also made possible an intense research effort resulting in
  - Topologies better suited to particular problems (CNNs, RNNs, Transformers, Attention)
  - New training strategies providing better generalization

In parallel, Deep Learning Platforms have emerged that make design, implementation, training, and production of DNNs feasible for everyone

1.1.2 1.2. Scope of this notebook

- To provide just overview of most important NNs and DNNs concepts
- Connecting with already studied methods as starting point (mainly logistic regression)
- Introduction to PyTorch
- Set the basis for learning about more advanced topologies for Natural Language Processing

1.1.3 1.3. Outline

1. Introduction and purpose of this Notebook
2. Introduction to Neural Networks
3. Implementing Deep Networks with PyTorch

1.1.4 1.4. Other resources

- We point here to external resources and tutorials that are excellent material for further study of the topic
- Most of them include examples and exercises using numpy and PyTorch
- This notebook uses examples and other material from some of these sources

Tutorial	Description
	Very general tutorial including videos and an overview of top deep learning platforms
	Very complete book with a lot of theory and examples for MxNET, PyTorch, and TensorFlow
	Official tutorials from the PyTorch project. Contains a 60 min overview, and a very practical *learning PyTorch with examples* tutorial
	Kaggle tutorials covering an introduction to Neural Networks using Numpy, and a second one offering a PyTorch tutorial

In addition to this, PyTorch MOOCs can be followed for free in main sites: edX, Coursera, Udacity

Preliminary work Complete the following tutorials for basic knowledge of pyTorch

- Introduction to PyTorch tensors
- Introduction to automatic differentiation with PyTorch
- PyTorch tutorial available in Google Colab (Complete just until de XOR example)

1.2 2. Datasets

Along this notebook, we will run some experiments to solve classification problems using two image datasets, that we name "digits" and "DogCats"

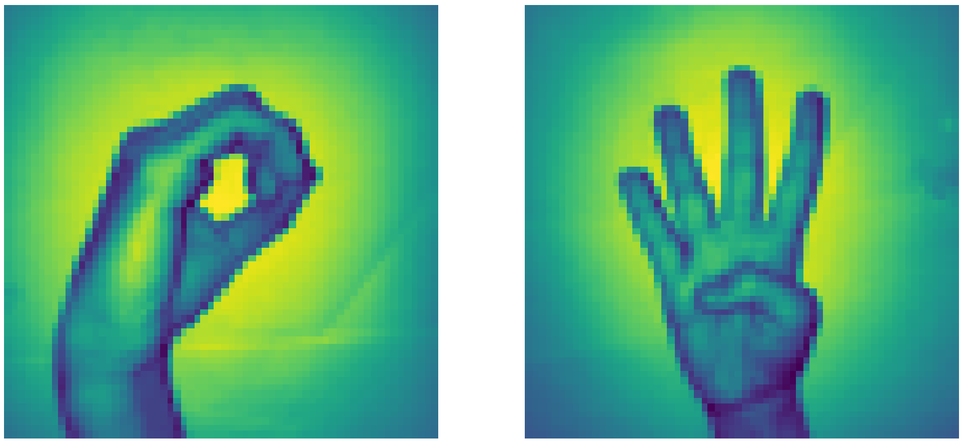
1.2.1 Digits: a sign language digits data set

- Dataset is taken from Kaggle and used in the above referred tutorial
- 2062 digits in sign language. 64×64 images
- Problem with 10 classes. One hot encoding for the label matrix
- Input data are images, we create also a flattened version

[3]:
```
# Load images and labels
digitsX = np.load('./data/Sign-language-digits-dataset/X.npy')
digitsY = np.load('./data/Sign-language-digits-dataset/Y.npy')

# Flatten images (to get 1-dimensional inputs
K = digitsX.shape[0]
```

```
img_size = digitsX.shape[1]
digitsX_flatten = digitsX.reshape(K,img_size*img_size)

print('Size of Input Data Matrix:', digitsX.shape)
print('Size of Flattened Input Data Matrix:', digitsX_flatten.shape)
print('Size of label Data Matrix:', digitsY.shape)

# Show sample images
selected = [260, 1400]
plt.subplot(1, 2, 1), plt.imshow(digitsX[selected[0]].reshape(img_size,␣
 ↪img_size)), plt.axis('off')
plt.subplot(1, 2, 2), plt.imshow(digitsX[selected[1]].reshape(img_size,␣
 ↪img_size)), plt.axis('off')
plt.show()
print('Labels corresponding to figures:', digitsY[selected,])
```

```
Size of Input Data Matrix: (2062, 64, 64)
Size of Flattened Input Data Matrix: (2062, 4096)
Size of label Data Matrix: (2062, 10)
```



```
Labels corresponding to figures: [[0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]]
```

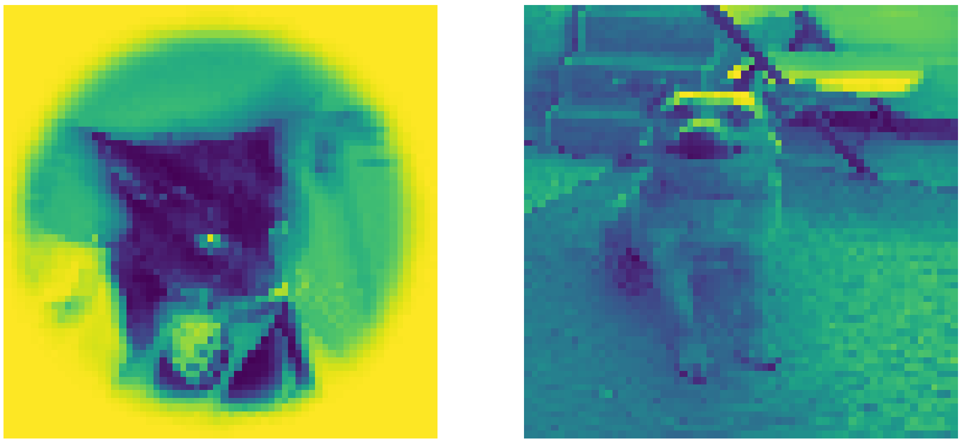
1.2.2 DogCats: a dataset of dogs and cat images

- Dataset is taken from Kaggle
- 25000 pictures of dogs and cats
- Binary problem
- Input data are images, we create also a flattened version
- Original images are RGB, and arbitrary size
- Preprocessed images are 64×64 and gray scale

```
[4]: # Preprocessing of original Dogs and Cats Pictures
     # Adapted from
     # https://medium.com/@mrgarg.rajat/
      ↪kaggle-dogs-vs-cats-challenge-complete-step-by-step-guide-part-1-a347194e55b1
     # RGB channels are collapsed in GRAYSCALE
     # Images are resampled to 64x64
     # This code has been used to generate the adapted dataset used in this␣
      ↪notebook, that is stored in
     # ./data/DogsCats/ .
     # You can uncomment this code to re-generate the dataset, if needed.
     """
     import os, cv2  # cv2 -- OpenCV

     train_dir = './data/DogsCats/train/'
     rows, cols = 64, 64
     train_images = sorted([train_dir+i for i in os.listdir(train_dir)])

     def read_image(file_path):
         image = cv2.imread(file_path, cv2.IMREAD_GRAYSCALE)
         return cv2.resize(image, (rows, cols),interpolation=cv2.INTER_CUBIC)

     def prep_data(images):
         m = len(images)
         X = np.ndarray((m, rows, cols), dtype=np.uint8)
         y = np.zeros((m,))
         print("X.shape is {}".format(X.shape))

         for i,image_file in enumerate(images) :
             image = read_image(image_file)
             X[i,] = np.squeeze(image.reshape((rows, cols)))
             if 'dog' in image_file.split('/')[-1].lower():
                 y[i] = 1
             elif 'cat' in image_file.split('/')[-1].lower():
                 y[i] = 0

             if i%5000 == 0 :
                 print(f"Proceed {i} of {m}")

         return X,y

     X_train, y_train = prep_data(train_images)
     np.save(X.npy', X_train)
     np.save('./data/DogsCats/Y.npy', y_train)
     """
     None
```

```
[5]: # Load images and labels
DogsCatsX = np.load('./data/DogsCats/X.npy')
DogsCatsY = np.load('./data/DogsCats/Y.npy')

# Flatten images to get 1D inputs
K = DogsCatsX.shape[0]
img_size = DogsCatsX.shape[1]
DogsCatsX_flatten = DogsCatsX.reshape(K,img_size*img_size)

print('Size of Input Data Matrix:', DogsCatsX.shape)
print('Size of Flattened Input Data Matrix:', DogsCatsX_flatten.shape)
print('Size of label Data Matrix:', DogsCatsY.shape)

# Show sample images
selected = [260, 16000]
plt.subplot(1, 2, 1), plt.imshow(DogsCatsX[selected[0]].reshape(img_size,␣
 ↪img_size)), plt.axis('off')
plt.subplot(1, 2, 2), plt.imshow(DogsCatsX[selected[1]].reshape(img_size,␣
 ↪img_size)), plt.axis('off')
plt.show()
print('Labels corresponding to figures:', DogsCatsY[selected,])
```

```
Size of Input Data Matrix: (25000, 64, 64)
Size of Flattened Input Data Matrix: (25000, 4096)
Size of label Data Matrix: (25000,)
```



```
Labels corresponding to figures: [0. 1.]
```

Now we define a function that, given the dataset name, prepares the data for binary or multiclass classification. The data are normalized and split into two sets for training and validation. This method will be used later to select the appropriate datasets

```
[6]: from sklearn.preprocessing import MinMaxScaler
     from sklearn.model_selection import train_test_split

     def get_dataset(dataset_name, forze_binary=False):
         """
         Loads the selected dataset, among two options: DogsCats or digits.

         If dataset_name == 'digits', you can take a dataset with two classes only,
         using forze_binary == True
         """

         if dataset_name == 'DogsCats':
             X = DogsCatsX_flatten
             y = DogsCatsY
         elif dataset_name == 'digits':
             if forze_binary:
                 # Zero and Ones are one hot encoded in columns 1 and 4
                 X0 = digitsX_flatten[np.argmax(digitsY, axis=1)==1,]
                 X1 = digitsX_flatten[np.argmax(digitsY, axis=1)==4,]
                 X = np.vstack((X0, X1))
                 y = np.zeros(X.shape[0])
                 y[X0.shape[0]:] = 1
             else:
                 X = digitsX_flatten
                 y = digitsY
         else:
             print("-- ERROR: Unknown dataset")
             return

         # Joint normalization of all data. For images [-.5, .5] scaling is frequent
         min_max_scaler = MinMaxScaler(feature_range=(-.5, .5))
         X = min_max_scaler.fit_transform(X)

         # Generate train and validation data, shuffle
         X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2,␣
      ↪random_state=42, shuffle=True)

         return X_train, X_val, y_train, y_val
```

1.3 3. Introduction to Neural Networks

In this section, we will implement neural networks from scratch using Numpy arrays (i.e., no PyTorch will be used in this section)

- No need to learn any new Python libraries
- But we need to deal with complexity of multilayer networks
- Low-level implementation will be useful to grasp the most important concepts concerning DNNs

- Back-propagation
- Activation functions
- Loss functions
- Optimization methods
- Generalization
- Special layers and configurations

1.3.1 3.1. A Single-Layer Neural Network for binary classification

3.1.1. Architecture One of the simplest neural network architectures for binary classification is shown in the figure

The main components are:

- A **linear combination** of the input features is computed to produce the intermediate output

$$o = \mathbf{w}^\top \mathbf{x} + b$$

- An **activation function**, which maps the linear combination to values in a bounded range, to produce the *soft* prediction

$$q = g(o)$$

  A common choice for binary classification is the logistic function, which provides probabilistic predictions $q \in [0, 1]$,

$$q = \text{logistic}(o) = \frac{1}{1 + \exp(-o)}.$$

  However, other activation functions are possible.

- A **binary threshold**, transforming the *soft* prediction into a *hard* decision (the class prediction) in $\{0, 1\}$. Following the probabilistic interpretation of the soft prediction, a common choice is to apply a threshold $\frac{1}{2}$, so that

$$\hat{y} = \left[\begin{array}{ll} 1, & \text{if } q \geq \frac{1}{2} \\ 0, & \text{if } q < \frac{1}{2} \end{array}\right.$$

We will define a `forward` method to implement the computation of the soft prediction, q. To do so, we define a method to implement the logistic function, too.

[7]:
```python
# Define some useful functions
def logistic(t):
    """
    Computes the logistic function
    """
    return 1.0 / (1 + np.exp(-t))

def forward(w,b,x):
    """
    Computes the network output
    """
    # return logistic(x.dot(w) + b)
    return logistic(x @ w + b)
```

For binary classification, our goal is to fit the weights so that the hard predictions are correct. Therefore, a natural measure of the classification performance is the accuracy, defined as the average number of correct decisions.

[8]:
```python
def accuracy(y, q):
    return np.mean(y == (q >= 0.5))
```

3.1.2. Loss functions The accuracy is a good measure for the evaluation of the classifiers, but it is not useful to define the learning algorithm. This is because learning algorithms for neural networs are mostly based on gradient-based optimization techniques. The thresholding function is not differentiable at $\frac{1}{2}$ and its derivative is zero elsewhere. Therefore, the derivatives of the accuracy with respect to the weights are not useful to guide learning.

For this reason, we need a **loss function**, that is, a measure of discrepancy between the true class, y, and the soft prediction q,

$$\ell(y, q)$$

that could be used for training. A basic learning algorithm will try to minimize the **empirical risk**, defined as cumulative loss over the whole training set

$$R(\mathbf{w}, b) = \sum_{k=0}^{K-1} \ell(y_k, q_k)$$

Many losses have been proposed for neural networks. Some examples are: * **Square error**: $\ell_2(y, q) = (y-q)^2$ * **Absolute error**: $\ell_1(y, q) = |y - q|$ * **Cross entropy**: $\ell_{\text{CE}}(y, q) = -y\log(q) - (1-y)\log(1-q)$

For binary classification, cross entroy is the most common choice.

3.1.3. Logistic Regression vs Single Layer NN Any neural network with probabilistic soft decisions defines a parametric probability model of the data. For the single-layer NN, the parametric model will be

$$P(y = 1|\mathbf{w}, b, \mathbf{x}) = g(\mathbf{w}^\top \mathbf{x} + b)$$

Therefore, we can train a neural network following a probabilistic approach. For instance, the negative log likelihood will be given by

$$\begin{aligned}
\text{NLL}(\mathbf{w}, b) &= -\sum_{k=0}^{K-1} \log(P(y_k|\mathbf{w}, b, \mathbf{x})) && (1)\\
&= -\sum_{k=0}^{K-1} (y_k \log(P(1|\mathbf{w}, b, \mathbf{x})) + (1-y_k)\log(P(0|\mathbf{w}, b, \mathbf{x}))) && (2)\\
&= -\sum_{k=0}^{K-1} (y_k \log(q_k) + (1-y_k)\log(1-q_k)) && (3)\\
&= \sum_{k=0}^{K-1} \ell_{\text{CE}}(y_k, q_k) && (4)
\end{aligned}$$

which shows that the empirical risk of the cross entropy is the NLL and, thus, optimizing the cross entropy provides ML estimates of the weights.

This also shows that a single-layer NN with logistic activation and cross-entropy loss is completely equivalent to a logistic regression model adjusted with ML.

3.1.4. Training In order to find parameters $\mathbf{w}$ and b, we will minimize the NLL via gradient descent optimization.

The gradient computation can be simplified using the **chain rule**

$$\begin{aligned}
\frac{\partial \text{NLL}}{\partial \mathbf{w}} &= \frac{\partial \text{NLL}}{\partial q} \cdot \frac{\partial q}{\partial o} \cdot \frac{\partial o}{\partial \mathbf{w}} && (5)\\
&= \sum_{k=0}^{K-1} \left[\frac{1-y_k}{1-q_k} - \frac{y_k}{q_k}\right] q_k(1-q_k)\mathbf{x}_k && (6)\\
&= \sum_{k=0}^{K-1} (q_k - y_k)\mathbf{x}_k && (7)\\
\frac{\partial \text{NLL}}{\partial b} &= \sum_{k=0}^{K-1} (q_k - y_k) && (8)
\end{aligned}$$

Therefore, the gradient descent rules are

$$\mathbf{w}_{n+1} = \mathbf{w}_n + \rho_n \sum_{k=0}^{K-1} (y_k - q_{k,n})\mathbf{x}_k$$

$$b_{n+1} = b_n + \rho_n \sum_{k=0}^{K-1} (y_k - q_{k,n}),$$

where $q_{k,n}$ is the probabilistic prediction for sample k. It depends on n because it depends on the weights, which change at each iteration.

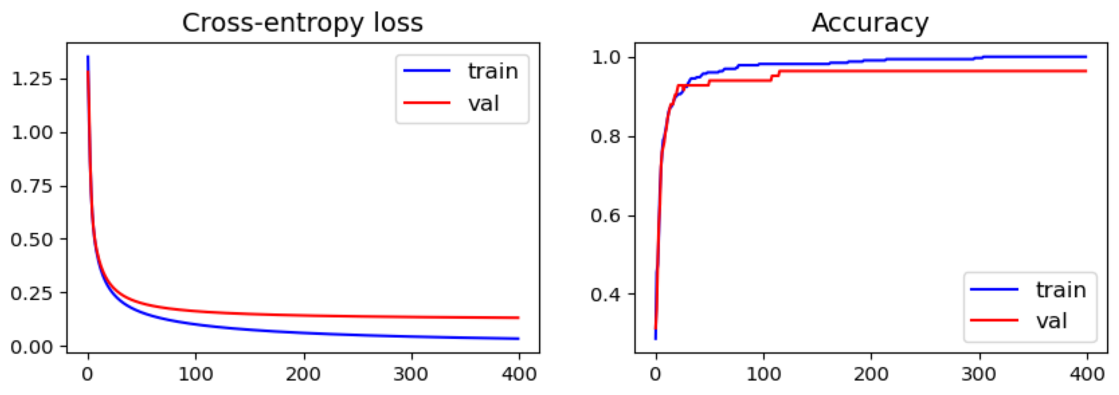
[9]:
```python
def backward(y, q, x):
    """
    Computes the gradient of the loss function for a single sample x with
    ouput y_hat, given label y.
    """
    # w_grad = x.T.dot((1-y)*q - y*(1-q))/len(y)
    # b_grad = np.sum((1-y)*q - y*(1-q))/len(y)
    w_grad = x.T @ (q - y) / len(y)
    b_grad = np.mean(q - y)
    return w_grad, b_grad

def loss(y, q):
    return - (y @ np.log(q) + (1 - y) @ np.log(1 - q)) / len(y)
```

3.1.5. Testing the single layer NN Now, we will test the behavior of the single-layer NN with the given datasets

[10]:
```python
# Load normalized data
X_train, X_val, y_train, y_val = get_dataset('digits', forze_binary=True)

# Neural Network Training
epochs = 400
rho = .05    # Use this setting for Sign Digits Dataset

# Parameter initialization
w = .1 * np.random.randn(X_train.shape[1])
b = .1 * np.random.randn(1)

loss_train = np.zeros(epochs)
loss_val = np.zeros(epochs)
acc_train = np.zeros(epochs)
acc_val = np.zeros(epochs)

for epoch in np.arange(epochs):
    print(f"-- Epoch {epoch + 1} out of {epochs}    \r", end="")
    q_train = forward(w, b, X_train)
    q_val = forward(w, b, X_val)
    w_grad, b_grad = backward(y_train, q_train, X_train)
    w = w - rho * w_grad
    b = b - rho * b_grad

    loss_train[epoch] = loss(y_train, q_train)
    loss_val[epoch] = loss(y_val, q_val)
    acc_train[epoch] = accuracy(y_train, q_train)
    acc_val[epoch] = accuracy(y_val, q_val)
```

```
-- Epoch 400 out of 400
```

[11]:
```python
plt.figure(figsize=(10,3))
plt.subplot(1, 2, 1), plt.plot(loss_train, 'b'), plt.plot(loss_val, 'r'), plt.
 ↪legend(['train', 'val']),
plt.title('Cross-entropy loss')
plt.subplot(1, 2, 2), plt.plot(acc_train, 'b'), plt.plot(acc_val, 'r'), plt.
 ↪legend(['train', 'val']),
plt.title('Accuracy')
plt.show()
```



Exercise 1 Study the behavior of the algorithm changing the number of epochs and the learning rate

```
[ ]: # <Write your code here>
```

Exercise 2 Repeat the analysis for the other dataset, trying to obtain as large an accuracy value as possible. What do you believe are the reasons for the very different performance for both datasets?

```
[ ]: # <Write your code here>
```

Linear logistic regression allowed us to review a few concepts that are key for Neural Networks:

- Network topology (In this case, a linear network with one layer)
- Activation functions
- Parametric approach ($\mathbf{w}/b$)
- Parameter initialization
- Obtaining the network prediction using *forward* computation
- Loss function
- Parameter gradient calculus using *backward* computation
- Optimization method for parameters update (here, GD)

1.3.2 3.2. Single-Layer Neural Networks for Multiclass Classification

3.2.1. Multiclass problems and one-hot encoding The single-layer NN can be easily extended to problems with $M \geq 2$ classes, $0, 1, \dots, M-1$.

To do so, we will represent classes using one-hot encoding, that is, M-dimensional vectors with zero componentes unless for a value 1 in the position indicated by the class.

For instance, classes in $\{0, 1, 2, 3\}$ will be represented by vectors

$$\begin{pmatrix}1\\0\\0\\0\end{pmatrix}\begin{pmatrix}0\\1\\0\\0\end{pmatrix}, \begin{pmatrix}0\\0\\1\\0\end{pmatrix} \text{ and } \begin{pmatrix}0\\0\\0\\1\end{pmatrix},$$

respectively.

Thus, both the true-class, $\mathbf{y}$, and the prediction, $\hat{\mathbf{y}}$, will be one-hot M-dimensional vectors .

3.2.2. Architecture A natural extension of the single layer NN to multiple classes is shown in the figure

The components of the multiclass model are multidimensional extensions of those of the single-layer NN for the binary classification problem:

- A **linear combination** is computed per each class. Note that, defining the matrix $\mathbf{W} = (\mathbf{w}_0|\mathbf{w}_1|\cdots|\mathbf{w}_{M-1})^\top$, we can write
$$\mathbf{o} = \mathbf{W}\mathbf{x} + \mathbf{b}$$
- **Activation function**. The **softmax** function is the most common choice. It is a multidimensional generalization of the logistic function (invented in 1959 by the social scientist R. Duncan Luce) and defined as
$$q_i = \frac{\exp(o_i)}{\sum_{j=0}^{M-1} \exp(o_j)}, \tag{9}$$
and it provides probabilistic soft predictions because
$$0 \le q_i \le 1$$
$$\sum_{j=0}^{M-1} q_j = 1$$
The derivatives of the softmax components, that will be required for training, are given by
$$\frac{\partial q_i}{\partial o_i} = q_i(1 - q_i) \tag{10}$$
$$\frac{\partial q_i}{\partial o_j} = -q_i q_j, \qquad j \neq i \tag{11}$$
- **Class prediction**: the final transformation maps the probabilistic predictions into a class prediction in one-hot form. Following the probabilistic interpretation of the soft prediction, we can use the **hardmax** function, which outputs a zero vector with a unit value in the ouput corresponding to the highest probabilistic input, that is,
$$\hat{y}_i = \left[\begin{array}{ll} 1, & \text{if } q_i = \max_j q_j \\ 0, & \text{otherwise} \end{array}\right.$$
The classifier is still linear, in the sense that
$$\text{hardmax}(\mathbf{q}) = \text{hardmax}(\mathbf{o}) = \text{hardmax}(\mathbf{W}\mathbf{x} + \mathbf{b})$$

3.2.3. Loss function The losses defined for the binary case can be easily extended to the multiclass setting:

The multi-class version of the cross entropy is defined as * **Square error**: $\ell_2(\mathbf{y}, \mathbf{q}) = \|\mathbf{y} - \mathbf{q}\|^2$ * **Absolute error**: $\ell_1(\mathbf{y}, \mathbf{q}) = \|\mathbf{y} - \mathbf{q}\|_1$ * **Cross entropy**: $\ell_{\text{CE}}(\mathbf{y}, \mathbf{q}) = -\sum_{j=0}^{M-1} y_j \log(q_j)$

We will implement the cross entropy. For evaluation purposes, the accuracy will be used

[12]:
```python
def accuracy(y, q):
    return np.mean(np.argmax(y, axis=1) == np.argmax(q, axis=1))

def loss(y, q):
    return - np.sum(y * np.log(q))
```

3.2.4. Probabilistic model As in the binary case, any neural network architecture with a probabilistic activation function defines a parametric probability model. For the architecture in the figure, such model is given by

$$P(y_i = 1|\mathbf{x}, \mathbf{W}, \mathbf{b}) = q_i$$

$$\mathbf{q} = \text{softmax}(\mathbf{W}\mathbf{x} + \mathbf{b})$$

Consequently, the negative log-likelihood is identical to the empirical risk defined by the cross entropy, that is

$$\text{NLL}(\mathbf{W}, \mathbf{b}) = \sum_{k=0}^{K-1} \ell_{\text{CE}}(\mathbf{y}_k, \mathbf{q}_k)$$

We will define a method to compute the softmax activation, and a `forward` method to compute the soft prediction from the inputs

[13]:
```python
# Define some useful functions
def softmax(t):
    """Compute softmax values for each sets of scores in t."""
    e_t = np.exp(t)
    return e_t / e_t.sum(axis=1, keepdims=True)

def forward(w, b, x):
    # Compute the soft prediction of the network
    return softmax(x @ w.T + b.T)
```

3.2.4. Training The Gradient Descent learning rules are given by

$$\mathbf{W}_{n+1} = \mathbf{W}_n - \rho_n \sum_{k=0}^{K-1} \frac{\partial l(\mathbf{y}_k, \mathbf{q}_k)}{\partial \mathbf{W}}$$

$$\mathbf{b}_{n+1} = \mathbf{b}_n - \rho_n \sum_{k=0}^{K-1} \frac{\partial l(\mathbf{y}_k, \mathbf{q}_k)}{\partial \mathbf{b}}$$

Applying the chain rule, and using the derivatives of the softmax function, the derivatives can be computed as follows:

$$\begin{aligned}
\frac{\partial l(\mathbf{y},\mathbf{q})}{\partial \mathbf{W}} &= \frac{\partial l(\mathbf{y},\mathbf{q})}{\partial \mathbf{o}} \cdot \frac{\partial \mathbf{o}}{\partial \mathbf{W}} && (12)\\
&= \sum_{i=0}^{M-1} \frac{\partial l(\mathbf{y},\mathbf{q})}{\partial o_i} \cdot \frac{\partial o_i}{\partial \mathbf{W}} && (13)\\
&= \frac{\partial l(\mathbf{y},\mathbf{q})}{\partial \mathbf{o}} \cdot \mathbf{x}^\top && (14)\\
&= \frac{\partial \mathbf{q}}{\partial \mathbf{o}} \cdot \frac{\partial l(\mathbf{y},\mathbf{q})}{\partial \mathbf{q}} \cdot \mathbf{x}^\top && (15)\\
&= \begin{bmatrix} q_1(1-q_1) & -q_1q_2 & \cdots & -q_1q_{M-1} \\ -q_2q_1 & q_2(1-q_2) & \cdots & -q_2q_{M-1} \\ \vdots & \vdots & \ddots & \vdots \\ -q_{M-1}q_1 & -q_{M-1}q_2 & \cdots & q_{M-1}(1-q_{M-1}) \end{bmatrix} \begin{bmatrix} -y_1/q_1 \\ -y_2/q_2 \\ \vdots \\ -y_{M-1}/q_{M-1} \end{bmatrix} \mathbf{x}^\top && (16)\\
&= (\mathbf{q}-\mathbf{y})\mathbf{x}^\top && (17)\\
& && (18)\\
\frac{\partial l(\mathbf{y},\mathbf{q})}{\partial \mathbf{b}} &= \mathbf{q}-\mathbf{y} && (19)
\end{aligned}$$

Thus, the gradient descent learning rules are

$$\mathbf{W}_{n+1} = \mathbf{W}_n + \rho_n \sum_{k=0}^{K-1} (\mathbf{y}_k - \mathbf{q}_{k,n}) \cdot \mathbf{x}_k^\top$$

$$\mathbf{b}_{n+1} = \mathbf{b}_n + \rho_n \sum_{k=0}^{K-1} (\mathbf{y}_k - \mathbf{q}_{k,n})$$

where $\mathbf{q}_{k,n}$ is the soft prediction for sample k. It depends on n because the soft prediction depends on the weights, which change at each iteration.

[14]:
```python
def backward(y, q, x):
    # Calcula los gradientes
    W_grad = (q - y).T @ x / len(y)
    b_grad = (q - y).T.mean(axis=1, keepdims=True)
    return W_grad, b_grad
```

3.2.5. Testing the multi-class single-layer NN Now, we will test the behavior of the multiclass NN with the `digits` dataset.

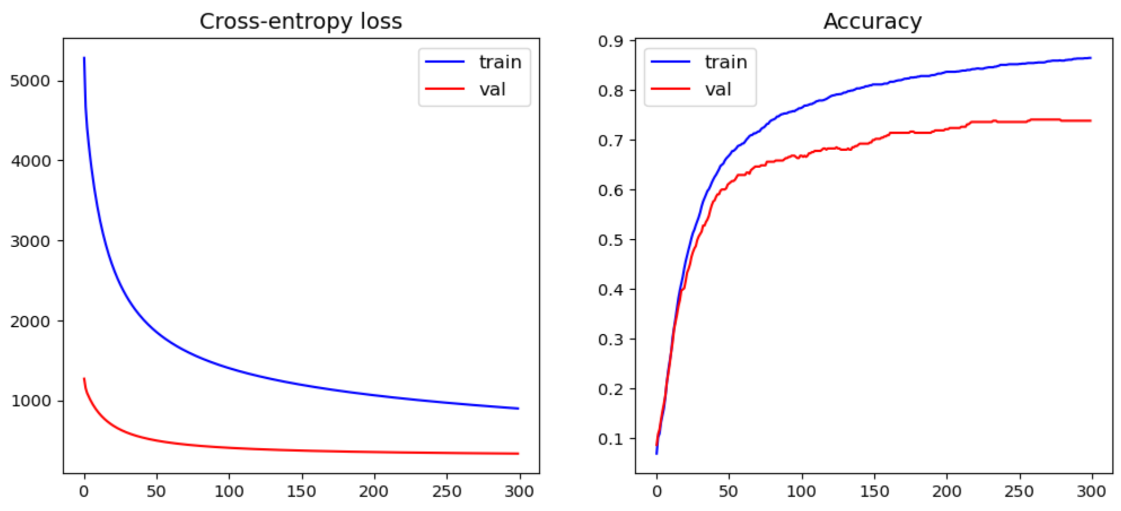
[15]:
```python
dataset = 'digits'
X_train, X_val, y_train, y_val = get_dataset('digits')
```

[16]:
```python
# Neural Network Training

epochs = 300
rho = .1
```

```
#Parameter initialization
W = .1 * np.random.randn(y_train.shape[1], X_train.shape[1])
b = .1 * np.random.randn(y_train.shape[1], 1)

loss_train = np.zeros(epochs)
loss_val = np.zeros(epochs)
acc_train = np.zeros(epochs)
acc_val = np.zeros(epochs)

for epoch in np.arange(epochs):
    print(f"Epoch {epoch + 1} out of {epochs}   \r", end="")
    q_train = forward(W, b, X_train)
    q_val = forward(W, b, X_val)
    W_grad, b_grad = backward(y_train, q_train, X_train)
    W = W - rho * W_grad
    b = b - rho * b_grad

    loss_train[epoch] = loss(y_train, q_train)
    loss_val[epoch] = loss(y_val, q_val)
    acc_train[epoch] = accuracy(y_train, q_train)
    acc_val[epoch] = accuracy(y_val, q_val)
```

```
Epoch 300 out of 300
```

[17]:
```
plt.figure(figsize=(12,5))
plt.subplot(1, 2, 1), plt.plot(loss_train, 'b'), plt.plot(loss_val, 'r'),
plt.legend(['train', 'val']), plt.title('Cross-entropy loss')
plt.subplot(1, 2, 2), plt.plot(acc_train, 'b'), plt.plot(acc_val, 'r'),
plt.legend(['train', 'val']), plt.title('Accuracy')
plt.show()
```



Exercise 3 Study the behavior of the algorithm changing the number of epochs and the learning rate

```
[ ]: # Write your code here
```

Exercise 4 Obtain the confusion matrix, and study which classes are more difficult to classify

```
[ ]: # Write your code here
```

Exercise 5 Think about the differences between using this 10-class network, vs training 10 binary classifiers, one for each class

```
[ ]: # Write your response here
```

As in linear logistic regression note that we covered the following aspects of neural network design, implementation, and training:

- Network topology (In this case, a linear network with one layer and M ouptuts)
- Activation functions (softmax activation)
- Initialization of parameters ($\mathbf{W}$, $\mathbf{b}$)
- Obtaining the network prediction using *forward* computation
- Loss function
- Gradient calculus using *backward* computation
- Optimization method for parameters update (here, GD)

1.3.3 3.3. Multi Layer Networks (Deep Networks)

Previous networks are constrained in the sense that they can only implement linear classifiers: the boundary decision of a binary single-layer NN is linear (an hyperplane) and the boundary sepearating each pair of classes in a multi-class single-layer NN is also linear.

As in logistic regression, we can easily apply the single-layer NN to non-linear classification problems by using fixed non-linear transformations of the inputs: $\mathbf{z} = \mathbf{f}(\mathbf{x})$, as the inputs to the linear layer. However, a fixed non-linear transformation limits the adaptability of the network to different datasets.

An interesting alternative is to parametrize the transformation using one or more non-linear layers of neurons. This is the central idea of the **multi-layer perceptron** (MLP).

- When counting layers, we normally ignore the input layer, since there is no computation involved
- Intermediate layers are normally referred to as "**hidden layers**"
- **Non-linear activations** result in an overall non-linear classifier
- We can still use **gradient descent optimization** as long as the derivatives of the loss function with respect to all parameters exist.
- This is already **deep learning**. We can have two layers or more, each with different numbers of neurons. But as long as derivatives with respect to parameters can be calculated, the network can be optimized
- **Structural optimization**: Finding an appropriate number of layers for a particular problem, as well as the number of neurons per layer, requires exploration

- The more data we have for training the network, the more parameters we can afford, making feasible the use of more complex topologies

3.3.1. Example: a 2-layer network for binary classification

Network topology The forward computation graph, shown in the figure, illustrates the computation steps that produce the network prediction and the loss computation

- **Hidden layers**: one hidden layer with n_h neurons with hyperbolic tangent activation. The hyperbolic tangent is just a shifted version of the logistic function producing outputs in the interval $[-1, 1]$ ($\tanh(o) = 2\text{logistic}(o) - 1$). It does not produce probabilistic outputs, but they are not needed at intermediate layers.
- **Output layer**: a single neuron with logistic activation function.
- **Loss function**: Cross-entropy

The network equations are, thus:

$$\mathbf{h} = \tanh(\mathbf{o}^{(1)}) = \tanh\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right)$$
$$q = \text{logistic}(o) = \text{logistic}\left({\mathbf{w}^{(2)}}^\top \mathbf{h} + b^{(2)}\right)$$

(where the hyperbolic tangent of a vector is computed component-wise). They are implemented in the forward method, below.

[18]:
```python
# Define some useful functions
def logistic(t):
    return 1.0 / (1 + np.exp(-t))

def forward(W1, b1, w2, b2, x):
    # Compute the network output
    h = 2 * logistic(x.dot(W1.T) + b1) - 1
    q = logistic(h.dot(w2) + b2)
    # Return also hidden units value for backward gradient step
    return h, q
```

Training We will train the neural network by applying the gradient descent learning rule to the minimization of the NLL (i.e. the cumulative cross entropy).

To do so, we need to compute the derivatives of the loss with respect to every network parameter. We will do it by applying extensively the chain rule:

- **Output layer** weights: the derivatives are the same that we have computed for the single-layer NN, since the dependency of the loss on the output layer weights is the same (we just need to use $\mathbf{h}$ instead of $\mathbf{x}$):

$$\mathbf{w}_{n+1}^{(2)} = \mathbf{w}_n^{(2)} + \rho_n \sum_{k=0}^{K-1} (y_k - q_{k,n})\mathbf{h}_{k,n}$$

$$b^{(2)}_{n+1} = b^{(2)}_n + \rho_n \sum_{k=0}^{K-1} (y_k - q_{k,n})$$

- **Hidden layer** weights: we need to use the chain rule (we ignore dimensions and rearrange at the end)

$$\tag{20}$$

$$\frac{\partial \ell_{\text{CE}}(y,q)}{\partial \mathbf{W}^{(1)}} = \frac{\partial \ell_{\text{CE}}(y,q)}{\partial o} \cdot \frac{\partial o}{\partial \mathbf{h}} \cdot \frac{\partial \mathbf{h}}{\partial \mathbf{o}^{(1)}} \cdot \frac{\partial \mathbf{o}^{(1)}}{\partial \mathbf{W}^{(1)}} \tag{21}$$

$$= (q-y)\left[\mathbf{w}^{(2)} \odot (\mathbf{1}-\mathbf{h})^2\right] \mathbf{x}^\top \tag{22}$$

where $\odot$ denotes component-wise multiplication and the square after $(\mathbf{1}-\mathbf{h})$ should be computed component-wise. (Note, also, that $\frac{\partial \mathbf{o}^{(1)}}{\partial \mathbf{W}^{(1)}}$ is actually a three dimensional matrix (i.e. a *tensor*). To apply the chain rule properly, the multiplications in the above equation must represent the adequate tensor products)

$$\tag{23}$$

$$\frac{\partial \ell_{\text{CE}}(y,q)}{\partial \mathbf{b}^{(1)}} = \frac{\partial \ell_{\text{CE}}(y,q)}{\partial o} \cdot \frac{\partial o}{\partial \mathbf{h}} \cdot \frac{\partial \mathbf{h}}{\partial \mathbf{o}^{(1)}} \cdot \frac{\partial \mathbf{o}^{(1)}}{\partial \mathbf{b}^{(1)}} \tag{24}$$

$$= (q-y)\left[\mathbf{w}^{(2)} \odot (\mathbf{1}-\mathbf{h})^2\right] \tag{25}$$

- GD update rules become

$$\mathbf{W}^{(1)}_{n+1} = \mathbf{W}^{(1)}_n + \rho_n \sum_{k=0}^{K-1} (y_k - q_{k,n}) \left[\mathbf{w}^{(2)} \odot (\mathbf{1}-\mathbf{h}_{k,n})^2\right] \mathbf{x}_k^\top$$

$$\mathbf{b}^{(1)}_{n+1} = \mathbf{b}^{(1)}_n + \rho_n \sum_{k=0}^{K-1} (y_k - q_{k,n}) \left[\mathbf{w}^{(2)} \odot (\mathbf{1}-\mathbf{h}_{k,n})^2\right]$$

[19]:
```python
def backward(y, q, h, x, w2):
    #Calculate gradients
    w2_grad = h.T.dot(q - y) / len(y)
    b2_grad = np.sum(q - y) / len(y)
    W1_grad = ((w2[np.newaxis,] * ((1 - h)**2) * (q - y)[:,np.newaxis]).T.
 ↪dot(x)) / len(y)
    b1_grad = ((w2[np.newaxis,] * ((1 - h)**2) * (q - y)[:,np.newaxis]).
 ↪sum(axis=0)) / len(y)
    return w2_grad, b2_grad, W1_grad, b1_grad

def accuracy(y, q):
    return np.mean(y == (q >= 0.5))

def loss(y, q):
    return - np.sum(y * np.log(q) + (1 - y) * np.log(1 - q)) / len(y)
```

3.3.2. The back-propagation algorithm The process that we have followed to compute the loss derivatives with respect to the weights can be extended to networks with an arbitrary number of layers.

Note that derivatives are computed backwards: from the last layer to the first hidden layer, so that we can use intermediate computations at a some layer to compute derivatives at layers that are further back.

For this reason, the gradient descent method is called the **back-propagation** algorithm.

3.3.3. Testing the 2-layer network Now we are ready to evaluate the two layer network

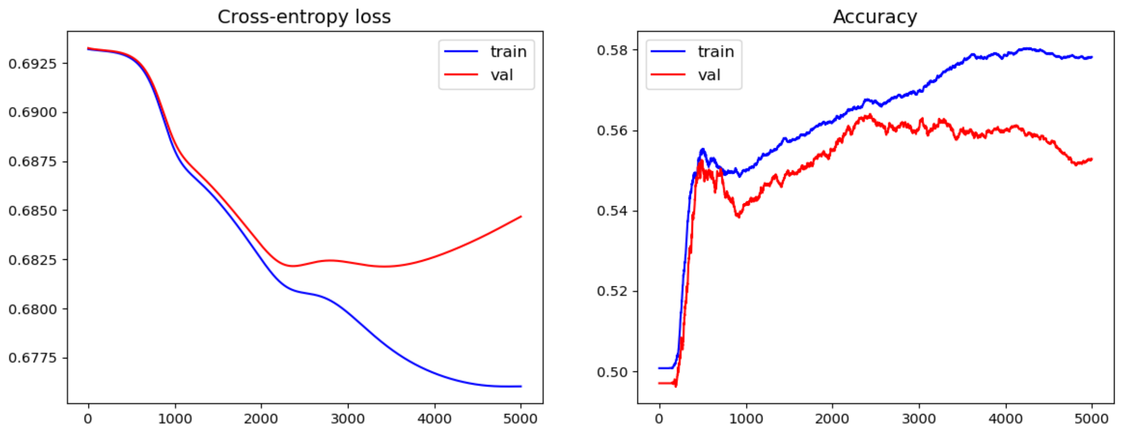
[20]:
```python
def evaluate_model(
    X_train, X_val, y_train, y_val, n_h=5, epochs=1000, rho=.005):

    W1 = .01 * np.random.randn(n_h, X_train.shape[1])
    b1 = .01 * np.random.randn(n_h)
    w2 = .01 * np.random.randn(n_h)
    b2 = .01 * np.random.randn(1)

    loss_train = np.zeros(epochs)
    loss_val = np.zeros(epochs)
    acc_train = np.zeros(epochs)
    acc_val = np.zeros(epochs)

    for epoch in np.arange(epochs):
        print(f'Current epoch: {epoch + 1}  \r', end="")

        h, q_train = forward(W1, b1, w2, b2, X_train)
        dum, q_val = forward(W1, b1, w2, b2, X_val)
        w2_grad, b2_grad, W1_grad, b1_grad = backward(y_train, q_train, h,␣
 ↪X_train, w2)
        W1 = W1 - rho/10 * W1_grad
        b1 = b1 - rho/10 * b1_grad
        w2 = w2 - rho * w2_grad
        b2 = b2 - rho * b2_grad

        loss_train[epoch] = loss(y_train, q_train)
        loss_val[epoch] = loss(y_val, q_val)
        acc_train[epoch] = accuracy(y_train, q_train)
        acc_val[epoch] = accuracy(y_val, q_val)

    return loss_train, loss_val, acc_train, acc_val
```

Results in Dogs vs Cats dataset

[21]:
```python
dataset = 'DogsCats'

X_train, X_val, y_train, y_val = get_dataset(dataset)
```

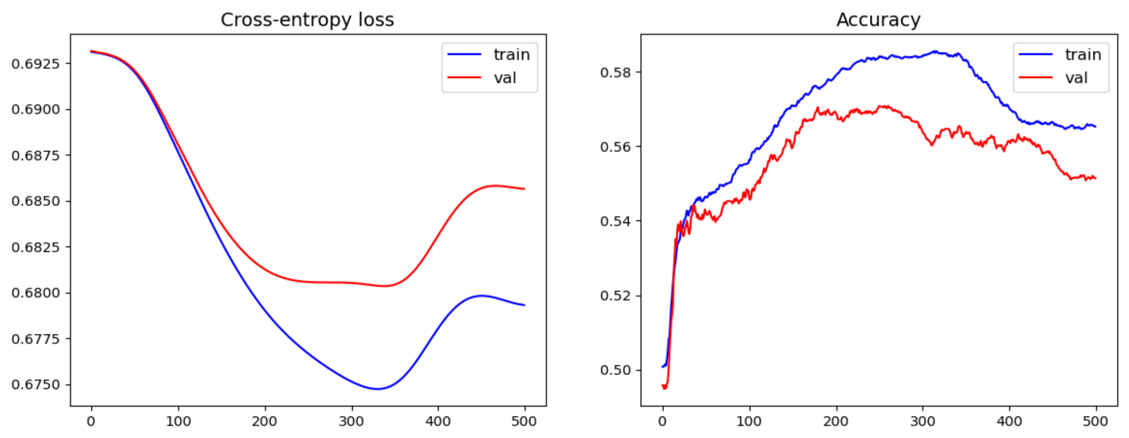
```
loss_train, loss_val, acc_train, acc_val = evaluate_model(
    X_train, X_val, y_train, y_val, n_h=5, epochs=5000, rho=0.05)

plt.figure(figsize=(14,5))
plt.subplot(1, 2, 1), plt.plot(loss_train, 'b'), plt.plot(loss_val, 'r'),
plt.legend(['train', 'val']), plt.title('Cross-entropy loss')
plt.subplot(1, 2, 2), plt.plot(acc_train, 'b'), plt.plot(acc_val, 'r'),
plt.legend(['train', 'val']), plt.title('Accuracy')
plt.show()
```

```
Current epoch: 5000
```



[27]:
```
dataset = 'DogsCats'

X_train, X_val, y_train, y_val = get_dataset(dataset)
loss_train, loss_val, acc_train, acc_val = evaluate_model(
    X_train, X_val, y_train, y_val, n_h=5, epochs=500, rho=0.5)

plt.figure(figsize=(14,5))
plt.subplot(1, 2, 1), plt.plot(loss_train, 'b'), plt.plot(loss_val, 'r'),
plt.legend(['train', 'val']), plt.title('Cross-entropy loss')
plt.subplot(1, 2, 2), plt.plot(acc_train, 'b'), plt.plot(acc_val, 'r'),
plt.legend(['train', 'val']), plt.title('Accuracy')
plt.show()
```

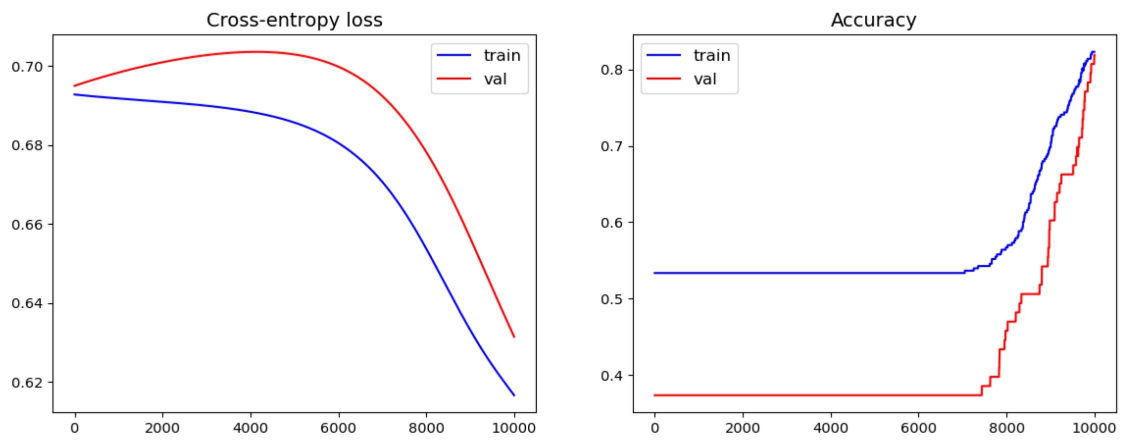
```
Current epoch: 500
```



Results in Binary Sign Digits Dataset

[28]:
```
dataset = 'digits'
X_train, X_val, y_train, y_val = get_dataset(dataset, forze_binary=True)
loss_train, loss_val, acc_train, acc_val = evaluate_model(
    X_train, X_val, y_train, y_val, n_h=5, epochs=10000, rho=0.001)

plt.figure(figsize=(14,5))
plt.subplot(1, 2, 1), plt.plot(loss_train, 'b'), plt.plot(loss_val, 'r'),
plt.legend(['train', 'val']), plt.title('Cross-entropy loss')
plt.subplot(1, 2, 2), plt.plot(acc_train, 'b'), plt.plot(acc_val, 'r'),
plt.legend(['train', 'val']), plt.title('Accuracy')
plt.show()
```

```
Current epoch: 10000
```



Exercise 6 Train the network using other settings for:

- The number of epochs
- The learning step
- The number of neurons in the hidden layer

You may find **divergence issues** for some settings

- Related to the use of the hyperbolic tangent function in the hidden layer (numerical issues)
- This is also why learning step was selected smaller for the hidden layer
- **Optimized libraries rely on certain modifications to obtain more robust implementations**

```
[ ]: # Write your solution here
```

Exercise 7 Try to solve both problems using the scikit-learn implementation of the MLP

- You can also explore other activation functions
- You can also explore other solvers to speed up convergence
- You can also adjust the size of minibatches
- Take a look at the *early_stopping* parameter

```
[ ]: # Write your solution here
```

1.3.4 3.4. Activation Functions

The MLP with two layers that we have used as an example contains sigmoid-type activation functions (logistic or hyperbolic tangent), which produce bounded outputs.

A major inconvenient of these kind of activations is that their derivatives vanish for large values of the input. As a consequence, learning can get stucked in *flat* regions of the parameter space.

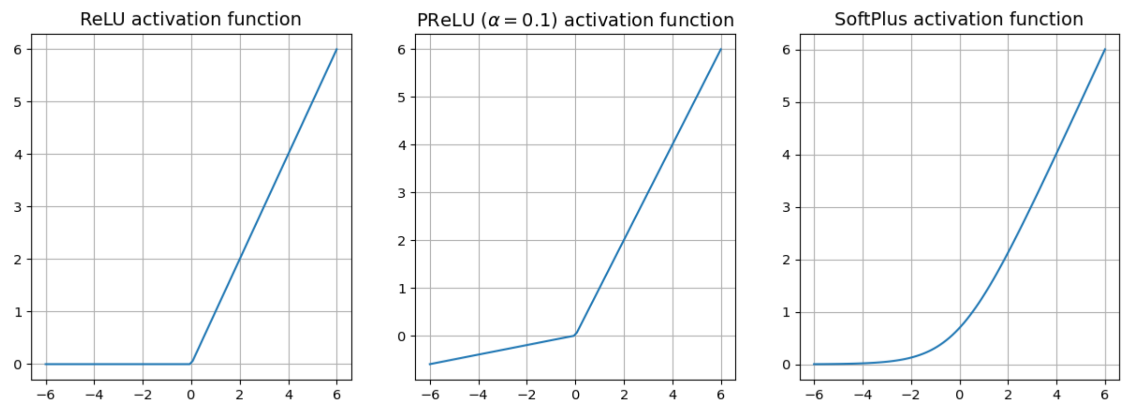
Activation functions must be non-linear (otherwise, all network layers could be colapsed into a single one), but they do not need to be neither probabilistic nor bounded (with the possible exception of the final layer).

For this reason, many other activation functions have been proposed. Some examples are

- **ReLU** (Rectified Linear Unit):
  - $\mathrm{ReLU}(t) = \max(0, t)$.
  - It is a one-side linear function. Its derivative is the step function.
- **PReLU** (Parametric Rectified Linear Unit):
  - $\mathrm{PReLU} = \max(\alpha t, t)$.
  - A modification of the ReLU that replaces the constant term 0 by a linear term with an adjustable parameter, that avoids zero derivatives. For $\alpha = 0.01$, it is named **Leaky ReLU**.
- **Softplus**:
  - $\mathrm{softplus}(t) = \log(1 + \exp(t))$.
  - Its derivative is the logistic function. It is a "soft" version of the ReLU: for large $|t|$, $\mathrm{softplus}(t) \approx \mathrm{ReLU}(t)$

(you can refer to the pytorch documentation or the Wikipedia to see many other examples)

[22]:
```
x_array = np.linspace(-6,6,100)
relu = np.clip(x_array, 0, a_max=None)
softplus = np.log(1 + np.exp(x_array))
LeakyLU = np.clip(x_array, 0.1 * x_array, a_max=None)

fig, axs = plt.subplots(1, 3)
fig.set_figwidth(15)
axs[0].plot(x_array, relu)
axs[1].plot(x_array, LeakyLU)
axs[2].plot(x_array, softplus)
axs[0].grid()
axs[1].grid()
axs[2].grid()
axs[0].set_title('ReLU activation function')
axs[1].set_title('PReLU ($\\alpha=0.1$) activation function')
axs[2].set_title('SoftPlus activation function')
plt.show()
```



Surprisingly, as explained in the Dive into Deep Learning book, the most popular choice for the hidden layers is the ReLU: despite its simplicity, it has shown good performance on many predictive tasks. Morever, despite it derivative is zero on one side, ReLU has demonstrated to mitigate the problem of vanishing gradients that seriously affected sigmoid-based neural networks.

1.3.5 3.5. Multi Layer Networks for Regression

Deep Learning networks can be used to solve regression problems with the following common adjustments

- Linear activation for the output unit
- Square loss (or other than the cross entropy):

$$\ell(y, \hat{y}) = (y - \hat{y})^2, \qquad \text{where} \qquad y, \hat{y} \in \mathbb{R}$$

1.4 4. Implementing Deep Networks with PyTorch

- Pytorch is a Python library that provides different levels of abstraction for implementing deep neural networks
- The main features of PyTorch are:
  - Definition of numpy-like **n-dimensional tensors**. They can be stored in (or moved to) GPU for **parallel execution** of operations
  - **Automatic calculation of gradients**, making *backward gradient calculation* transparent to the user
  - **Pre-defined components**: common loss functions, different types of NN layers, optimization methods, data loaders, etc, simplifying NN implementation and training
  - Provides **different levels of abstraction**, thus a good balance between flexibility and simplicity
- This notebook provides just a basic review of the main concepts necessary to train NNs with PyTorch taking materials from:
  - Learning PyTorch with Examples, by Justin Johnson
  - What is *torch.nn* really?, by Jeremy Howard
  - Pytorch Tutorial for Deep Learning Lovers, by Kaggle user kanncaa1

1.4.1 4.1. Installation and PyTorch introduction

- PyTorch can be installed with or without GPU support
  - If you have an Anaconda installation, you can install from the command line, using the instructions of the project website
- PyTorch is also preinstalled in Google Collab with free GPU access
  - Follow RunTime -> Change runtime type, and select GPU for HW acceleration
- Please, refer to Pytorch getting started tutorial for a quick introduction regarding tensor definition, GPU vs CPU storage of tensors, operations, and bridge to Numpy

1.4.2 4.2. Torch tensors (very) general overview

We can create tensors with different construction methods provided by the library, either to create new tensors from scratch or from a Numpy array

[23]:
```
import torch

x = torch.rand((100,200))
digitsX_flatten_tensor = torch.from_numpy(digitsX_flatten)

print(x.type())
print(digitsX_flatten_tensor.size())
```

```
torch.FloatTensor
torch.Size([2062, 4096])
```

- Tensors can be converted back to numpy arrays
- Note that in this case, a tensor and its corresponding numpy array **will share memory**

Operations and slicing use a syntax similar to numpy

[24]:
```python
print('Size of tensor x:', x.size())
print('Tranpose of vector has size', x.t().size()) #Transpose and compute size
print('Extracting upper left matrix of size 3 x 3:', x[:3,:3])
print(x.mm(x.t()).size())  #mm for matrix multiplications
xpx = x.add(x)
xpx2 = torch.add(x,x)
print((xpx != xpx2).sum())   # Since all are equal, count of different terms is␣
 ↪zero
```

```
Size of tensor x: torch.Size([100, 200])
Tranpose of vector has size torch.Size([200, 100])
Extracting upper left matrix of size 3 x 3: tensor([[0.0815, 0.7430, 0.5832],
        [0.3206, 0.5126, 0.7064],
        [0.6637, 0.8015, 0.6468]])
torch.Size([100, 100])
tensor(0)
```

- Adding underscore performs operations "*in place*", e.g., `x.add_(y)`

If a GPU is available, tensors can be moved to and from the GPU device Operations on tensors stored in a GPU will be carried out using GPU resources and will typically be highly parallelized

[33]:
```python
#Select device depending on GPU

def get_best_device():
    if torch.cuda.is_available():  # NVIDIA GPU
        return torch.device("cuda")
    elif hasattr(torch.backends, "mps") and torch.backends.mps.is_available(): ␣
 ↪# Apple Silicon GPU
        return torch.device("mps")
    elif hasattr(torch.version, "hip") and torch.version.hip is not None:  #␣
 ↪AMD ROCm GPU
        return torch.device("cuda")  # ROCm uses CUDA device type internally
    else:
        print("No supported GPU backend found → using CPU.")
        return torch.device("cpu")


device = get_best_device()

x = torch.randn(3, 3).to(device)
y = x + x
```

1.4.3 4.3. Automatic gradient calculation

PyTorch tensors have a property `requires_grad`. When true, PyTorch automatic gradient calculation will be activated for that variable

- In order to compute these derivatives numerically, PyTorch keeps track of all operations carried out on these variables, organizing them in a forward computation graph.
- When executing the `backward()` method, derivatives will be calculated
- However, this should only be activated when necessary, to save computation

[34]:
```
x.requires_grad = True
y = (3 * torch.log(x)).sum()
y.backward()
print(x.grad[:2,:2])
print(3/x[:2,:2])

x.requires_grad = False
x.grad.zero_()
print('Automatic gradient calculation is deactivated, and gradients set to␣
 ↪zero')
```

```
tensor([[  1.5003,   2.2902],
        [  2.7504, -11.7111]], device='mps:0')
tensor([[  1.5003,   2.2902],
        [  2.7504, -11.7111]], device='mps:0', grad_fn=<MulBackward0>)
Automatic gradient calculation is deactivated, and gradients set to zero
```

Exercise 8 **1.1.** Initialize a tensor `x` with the upper right 5×10 submatrix of flattened digits. Activate xas a variable required for gradient computation.

[29]:
```
# Write your solution here
# <SOL>
x = torch.from_numpy(digitsX_flatten[:5, :10])
x.requires_grad = True
# </SOL>

print(x)
```

```
tensor([[0.4667, 0.4745, 0.4784, 0.4824, 0.4863, 0.4902, 0.4941, 0.4941, 0.5059,
         0.5137],
        [0.5961, 0.6078, 0.6196, 0.6314, 0.6431, 0.6471, 0.6588, 0.6667, 0.6824,
         0.6941],
        [0.5882, 0.6039, 0.6196, 0.6314, 0.6431, 0.6549, 0.6667, 0.6824, 0.7020,
         0.7098],
        [0.5569, 0.5686, 0.5843, 0.6000, 0.6118, 0.6196, 0.6275, 0.6392, 0.6549,
         0.6627],
        [0.5804, 0.5765, 0.5922, 0.6078, 0.6157, 0.6353, 0.6431, 0.6471, 0.6588,
         0.6706]], requires_grad=True)
```

1.2. Compute output vector `y` as the component-wise square root of `x`

[30]:
```
# y = <FILL IN>
y = torch.sqrt(x)
print(y)
```

```
tensor([[0.6831, 0.6888, 0.6917, 0.6945, 0.6973, 0.7001, 0.7029, 0.7029, 0.7113,
         0.7167],
        [0.7721, 0.7796, 0.7872, 0.7946, 0.8020, 0.8044, 0.8117, 0.8165, 0.8260,
         0.8331],
        [0.7670, 0.7771, 0.7872, 0.7946, 0.8020, 0.8093, 0.8165, 0.8260, 0.8378,
         0.8425],
        [0.7462, 0.7541, 0.7644, 0.7746, 0.7822, 0.7872, 0.7921, 0.7995, 0.8093,
         0.8141],
        [0.7618, 0.7593, 0.7695, 0.7796, 0.7847, 0.7971, 0.8020, 0.8044, 0.8117,
         0.8189]], grad_fn=<SqrtBackward0>)
```

1.3. Compute scalar value `z` as the sum of all elements in `y` squared. You can easily test if it is equal to the sum of all elements in `x`.

```
[31]: # z = <FILL IN>
z = (y**2).sum()

print(z)
```

```
tensor(30.1882, grad_fn=<SumBackward0>)
```

1.4. Compute the derivatives of `z` with the `backward` method, an check if they are correct.

Note: The backward method can only be run on scalar variables

```
[39]: # Write your solution here
# <SOL>

# This is correct, but it is under comments to avoid a run error in the next␣
 ↪cell.
# z.backward()
# print(x.grad)

# </SOL>
```

1.5. If you try to run the last cell multiple times, yoy will likely get an error. Implement the necessary modifications so that you can run the backward method multiple times, but the gradient does not change from run to run

```
[32]: # Write your solution here
# <SOL>
z.backward(retain_graph=True)
print(x.grad)
x.grad.zero_()

# </SOL>
```

```
tensor([[1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.]])
```

```
[32]: tensor([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
              [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
              [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
              [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
              [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]])
```

1.4.4 4.4. Feed Forward Neural Network using PyTorch

In this section we will change our code for neural network evaluation and training to use tensors instead of numpy arrays. We will work with the `sign digits` dataset.

We will introduce all concepts using a single layer perceptron (softmax regression), and then implement networks with additional hidden layers

4.4.1. Using Automatic differentiation We start by loading the data, and converting to tensors.

- As a first step, we refactor our code to use tensor operations
- We do not need to pay too much attention to particular details regarding tensor operations, since these will not be necessary when moving to higher PyTorch abstraction levels
- We do not need to implement gradient calculation. PyTorch will take care of that

```
[35]: from sklearn.preprocessing import MinMaxScaler
      from sklearn.model_selection import train_test_split

      dataset = 'digits'

      # Joint normalization of all data. For images [-.5, .5] scaling is frequent
      min_max_scaler = MinMaxScaler(feature_range=(-.5, .5))
      X = min_max_scaler.fit_transform(digitsX_flatten)

      # Generate train and validation data, shuffle
      X_train, X_val, y_train, y_val = train_test_split(X, digitsY, test_size=0.2,␣
       ↪random_state=42, shuffle=True)

      # Convert to Torch tensors
      X_train_torch = torch.from_numpy(X_train)
      X_val_torch = torch.from_numpy(X_val)
      y_train_torch = torch.from_numpy(y_train)
      y_val_torch = torch.from_numpy(y_val)
```

```
[36]: # Define some useful functions
      def softmax(t):
          """Compute softmax values for each sets of scores in t"""
          return t.exp() / t.exp().sum(-1).unsqueeze(-1)
```

```
def model(w,b,x):
    # Compute the probabilistic prediction
    return softmax(x.mm(w) + b)

def accuracy(y, q):
    return (y.argmax(axis=-1) == q.argmax(axis=-1)).float().mean()

def nll(y, q):
    return -(y * q.log()).mean()
```

Note that:

- Syntaxis is a bit different because input variables are tensors, not arrays
- This time we did not need to implement the backward function

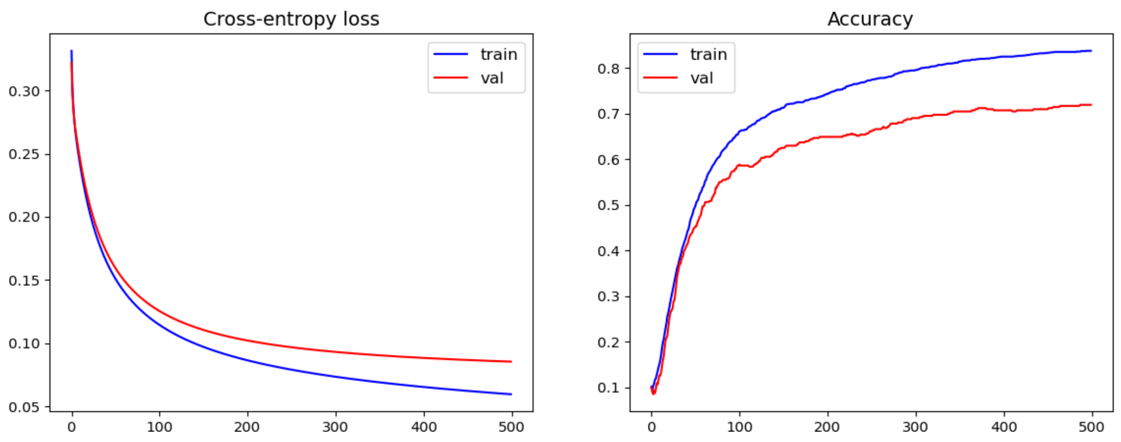
[37]:
```
# Parameter initialization
W = .1 * torch.randn(X_train_torch.size()[1], y_train_torch.size()[1])
W.requires_grad_()
b = torch.zeros(y_train_torch.size()[1], requires_grad=True)

epochs = 500
rho = .5

loss_train = np.zeros(epochs)
loss_val = np.zeros(epochs)
acc_train = np.zeros(epochs)
acc_val = np.zeros(epochs)
```

[38]:
```
# Network training
for epoch in range(epochs):

    print(f'Current epoch: {epoch + 1}  \r', end="")

    # Compute network output and cross-entropy loss
    pred = model(W, b, X_train_torch)
    loss = nll(y_train_torch, pred)

    # Compute gradients
    loss.backward()

    # Deactivate gradient automatic updates
    with torch.no_grad():
        #Computing network performance after iteration
        loss_train[epoch] = loss.item()
        acc_train[epoch] = accuracy(y_train_torch, pred).item()
        pred_val = model(W, b, X_val_torch)
        loss_val[epoch] = nll(y_val_torch, pred_val).item()
        acc_val[epoch] = accuracy(y_val_torch, pred_val).item()
        #Weight update
        W -= rho * W.grad
        b -= rho * b.grad

        # Reset gradients
        W.grad.zero_()
        b.grad.zero_()
```

```
Current epoch: 500
```

It is important to deactivate gradient updates after the network has been evaluated on training data, and gradients of the loss function have been computed: - Validation data should never be used for updating the network parameters - Save computation to accelerate the code

[39]:
```
plt.figure(figsize=(14,5))
plt.subplot(1, 2, 1), plt.plot(loss_train, 'b'), plt.plot(loss_val, 'r'), plt.
 ↪legend(['train', 'val']), plt.title('Cross-entropy loss')
plt.subplot(1, 2, 2), plt.plot(acc_train, 'b'), plt.plot(acc_val, 'r'), plt.
 ↪legend(['train', 'val']), plt.title('Accuracy')
plt.show()
```



4.4.2. Using torch *nn* module PyTorch *nn* module provides many attributes and methods that make the implementation and training of Neural Networks simpler

- `nn.Module` and `nn.Parameter` allow to implement a more concise training loop
- `nn.Module` is a PyTorch class that will be used to encapsulate and design a specific neural network, thus, it is central to the implementation of deep neural nets using PyTorch
- `nn.Parameter` allow the definition of trainable network parameters. In this way, we will simplify the implementation of the training loop.

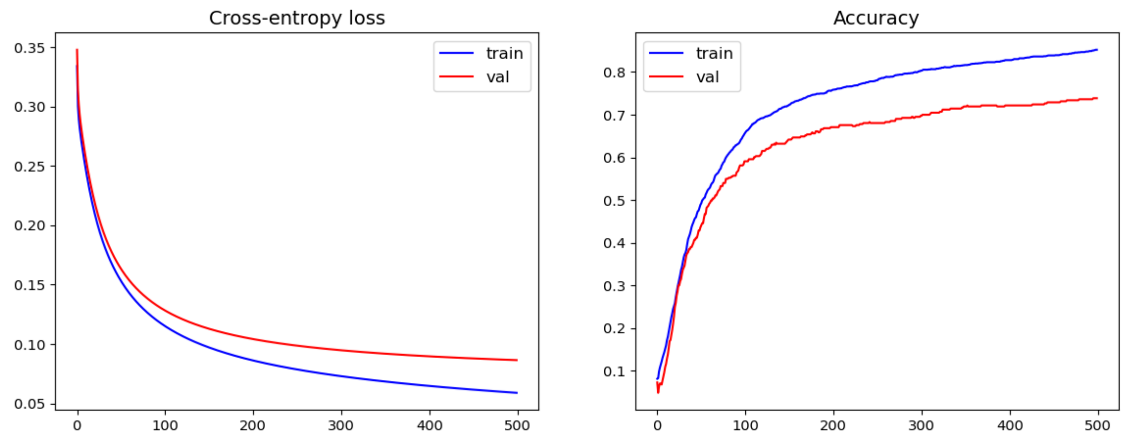
- All parameters defined with `nn.Parameter` will have `requires_grad = True`

[40]:
```
from torch import nn

class my_multiclass_net(nn.Module):
    def __init__(self, nin, nout):
        """ This method initializes the network parameters
        Parameters nin and nout stand for the number of input parameters␣
 ↪(features in X)
        and output parameters (number of classes) """
        super().__init__()
        self.W = nn.Parameter(.1 * torch.randn(nin, nout))
        self.b = nn.Parameter(torch.zeros(nout))

    def forward(self, x):
        return softmax(x.mm(self.W) + self.b)

    def softmax(t):
        """Compute softmax values for each sets of scores in t"""
        return t.exp() / t.exp().sum(-1).unsqueeze(-1)
```

[41]:
```
my_net = my_multiclass_net(X_train_torch.size()[1], y_train_torch.size()[1])

epochs = 500
rho = .5

loss_train = np.zeros(epochs)
loss_val = np.zeros(epochs)
acc_train = np.zeros(epochs)
acc_val = np.zeros(epochs)

for epoch in range(epochs):

    print(f'Current epoch: {epoch + 1}  \r', end="")

    #Compute network output and cross-entropy loss
    pred = my_net(X_train_torch)
    loss = nll(y_train_torch, pred)

    #Compute gradients
    loss.backward()

    #Deactivate gradient automatic updates
    with torch.no_grad():
        #Computing network performance after iteration
        loss_train[epoch] = loss.item()
        acc_train[epoch] = accuracy(y_train_torch, pred).item()
        pred_val = my_net(X_val_torch)
        loss_val[epoch] = nll(y_val_torch, pred_val).item()
        acc_val[epoch] = accuracy(y_val_torch, pred_val).item()

        #Weight update
        for p in my_net.parameters():
            p -= p.grad * rho
        #Reset gradients
        my_net.zero_grad()
```

```
Current epoch: 500
```

[42]:
```
plt.figure(figsize=(14,5))
plt.subplot(1, 2, 1), plt.plot(loss_train, 'b'), plt.plot(loss_val, 'r'), plt.
 ↪legend(['train', 'val']), plt.title('Cross-entropy loss')
plt.subplot(1, 2, 2), plt.plot(acc_train, 'b'), plt.plot(acc_val, 'r'), plt.
 ↪legend(['train', 'val']), plt.title('Accuracy')
plt.show()
```



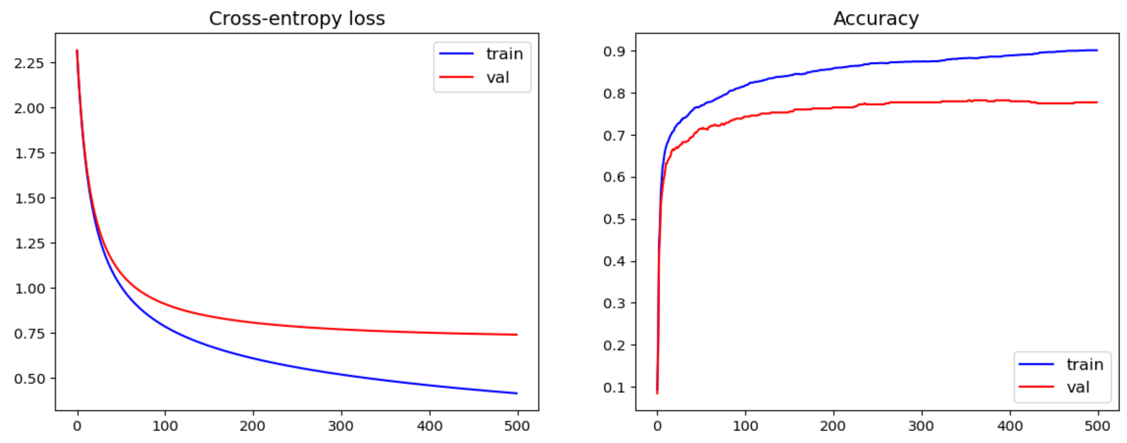
- `nn.Module` comes with several kinds of pre-defined layers, thus making it even simpler to implement neural networks
- We can also import the Cross Entropy Loss from `nn.Module`. When doing so:
  - We do not have to compute the softmax, since the `nn.CrossEntropyLoss` already does so
  - `nn.CrossEntropyLoss` receives two input arguments, the first is the output of the network, and the second is the true label as a 1-D tensor (i.e., an array of integers, one-hot encoding should not be used)

[43]:
```
from torch import nn

class my_multiclass_net(nn.Module):
    def __init__(self, nin, nout):
        """Note that now, we do not even need to initialize network parameters␣
↪ourselves"""
        super().__init__()
        self.lin = nn.Linear(nin, nout)

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

loss_func = nn.CrossEntropyLoss()
```

[44]:
```
my_net = my_multiclass_net(X_train_torch.size()[1], y_train_torch.size()[1])

epochs = 500
rho = .1

loss_train = np.zeros(epochs)
loss_val = np.zeros(epochs)
acc_train = np.zeros(epochs)
acc_val = np.zeros(epochs)

for epoch in range(epochs):

    print(f'Current epoch: {epoch + 1}  \r', end="")

    #Compute network output and cross-entropy loss
    pred = my_net(X_train_torch)
    loss = loss_func(pred, y_train_torch.argmax(axis=-1))

    #Compute gradients
    loss.backward()

    #Deactivate gradient automatic updates
    with torch.no_grad():
        #Computing network performance after iteration
        loss_train[epoch] = loss.item()
        acc_train[epoch] = accuracy(y_train_torch, pred).item()
        pred_val = my_net(X_val_torch)
        loss_val[epoch] = loss_func(pred_val, y_val_torch.argmax(axis=-1)).
↪item()
        acc_val[epoch] = accuracy(y_val_torch, pred_val).item()

        #Weight update
        for p in my_net.parameters():
            p -= p.grad * rho
        #Reset gradients
        my_net.zero_grad()
```

```
Current epoch: 500
```

[45]:
```
plt.figure(figsize=(14,5))
plt.subplot(1, 2, 1), plt.plot(loss_train, 'b'), plt.plot(loss_val, 'r'), plt.
 ↪legend(['train', 'val']), plt.title('Cross-entropy loss')
plt.subplot(1, 2, 2), plt.plot(acc_train, 'b'), plt.plot(acc_val, 'r'), plt.
 ↪legend(['train', 'val']), plt.title('Accuracy')
plt.show()
```



Note that a faster convergence is observed in this case. It is actually due to a more convenient initialization of the hidden layer

4.4.3. Network Optimization We cover in this subsection two different aspects about network training using PyTorch:

- Using `torch.optim` allows an easier and more interpretable encoding of neural network training, and opens the door to more sophisticated training algorithms
- Using minibatches can speed up network convergence

`torch.optim` provides two convenient methods for neural network training:

- `opt.step()` updates all network parameters using current gradients
- `opt.zero_grad()` resets all network parameters

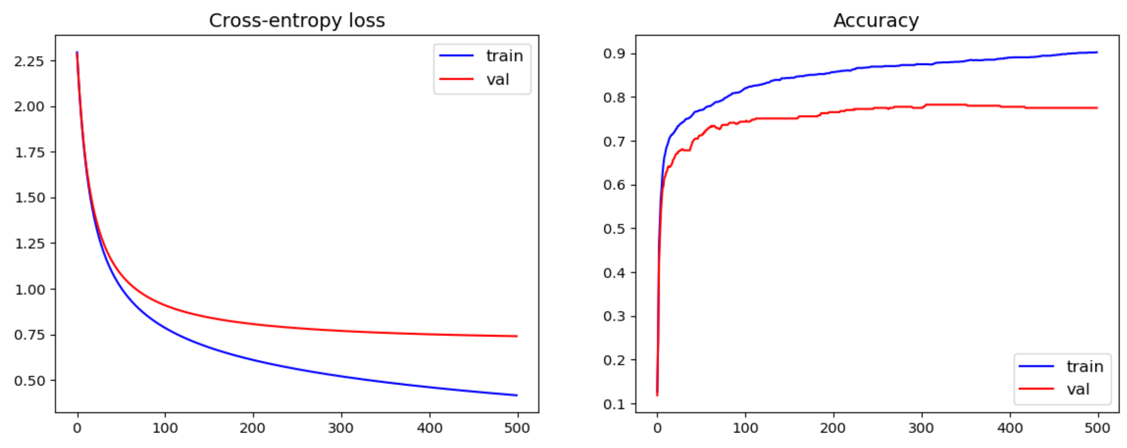
[46]:
```
from torch import optim

my_net = my_multiclass_net(X_train_torch.size()[1], y_train_torch.size()[1])
opt = optim.SGD(my_net.parameters(), lr=0.1)

epochs = 500

loss_train = np.zeros(epochs)
loss_val = np.zeros(epochs)
acc_train = np.zeros(epochs)
```

```
acc_val = np.zeros(epochs)

for epoch in range(epochs):

    print(f'Current epoch: {epoch + 1}  \r', end="")

    # Compute network output and cross-entropy loss
    pred = my_net(X_train_torch)
    loss = loss_func(pred, y_train_torch.argmax(axis=-1))

    # Compute gradients
    loss.backward()

    # Deactivate gradient automatic updates
    with torch.no_grad():
        #Computing network performance after iteration
        loss_train[epoch] = loss.item()
        acc_train[epoch] = accuracy(y_train_torch, pred).item()
        pred_val = my_net(X_val_torch)
        loss_val[epoch] = loss_func(pred_val, y_val_torch.argmax(axis=-1)).
 ↪item()
        acc_val[epoch] = accuracy(y_val_torch, pred_val).item()

    opt.step()
    opt.zero_grad()
```

```
Current epoch: 500
```

Note network optimization is carried out outside `torch.no_grad()` but network evaluation (other than forward output calculation for the training patterns) still needs to deactivate gradient updates

[47]:
```
plt.figure(figsize=(14,5))
plt.subplot(1, 2, 1), plt.plot(loss_train, 'b'), plt.plot(loss_val, 'r'), plt.
 ↪legend(['train', 'val']), plt.title('Cross-entropy loss')
plt.subplot(1, 2, 2), plt.plot(acc_train, 'b'), plt.plot(acc_val, 'r'), plt.
 ↪legend(['train', 'val']), plt.title('Accuracy')
plt.show()
```



Exercise 9 Implement network training with two modifications:

- Replace the SGD optimization method by the Adam algorithm. You can refer to the official documentation and get help on this and other methods.
- Implement and adaptive learning rate using `torch.optim.lr_scheduler` (for instance, you can try the exponentialLR scheduler).

[54]:
```
# Write your solution here
# <SOL>

# </SOL>
```

4.4.4. The DataLoader method. Using SGD with minibatches Each epoch of the previous implementation of network training was actually implementing Gradient Descent * In Stochastic Gradient Descent (SGD) only small subset of training patterns (a *minibatch*) are used at every iteration * In each epoch we iterate over all training patterns sequentially selecting non-overlapping *minibatches* * Overall, convergence is usually faster than when using standard Gradient Descent * Torch provides methods that simplify the implementation of this strategy

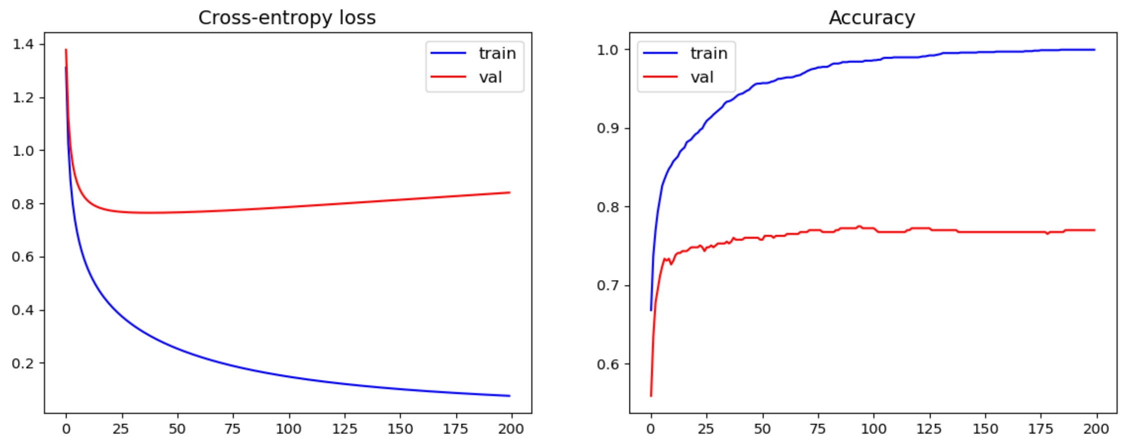
[48]:
```
from torch.utils.data import TensorDataset, DataLoader

train_ds = TensorDataset(X_train_torch, y_train_torch)
train_dl = DataLoader(train_ds, batch_size=64)
```

[49]:
```
from torch import optim

my_net = my_multiclass_net(X_train_torch.size()[1], y_train_torch.size()[1])
opt = optim.SGD(my_net.parameters(), lr=0.1)

epochs = 200

loss_train = np.zeros(epochs)
```

```
loss_val = np.zeros(epochs)
acc_train = np.zeros(epochs)
acc_val = np.zeros(epochs)

for epoch in range(epochs):

    print(f'Current epoch: {epoch + 1}  \r', end="")

    for xb, yb in train_dl:

        #Compute network output and cross-entropy loss for current minibatch
        pred = my_net(xb)
        loss = loss_func(pred, yb.argmax(axis=-1))

        #Compute gradients and optimize parameters
        loss.backward()
        opt.step()
        opt.zero_grad()

    #At the end of each epoch, evaluate overall network performance
    with torch.no_grad():
        #Computing network performance after iteration
        pred = my_net(X_train_torch)
        loss_train[epoch] = loss_func(pred, y_train_torch.argmax(axis=-1)).
 ↪item()
        acc_train[epoch] = accuracy(y_train_torch, pred).item()
        pred_val = my_net(X_val_torch)
        loss_val[epoch] = loss_func(pred_val, y_val_torch.argmax(axis=-1)).
 ↪item()
        acc_val[epoch] = accuracy(y_val_torch, pred_val).item()
```

```
Current epoch: 200
```

[50]:
```
plt.figure(figsize=(14,5))
plt.subplot(1, 2, 1), plt.plot(loss_train, 'b'), plt.plot(loss_val, 'r'), plt.
 ↪legend(['train', 'val']), plt.title('Cross-entropy loss')
plt.subplot(1, 2, 2), plt.plot(acc_train, 'b'), plt.plot(acc_val, 'r'), plt.
 ↪legend(['train', 'val']), plt.title('Accuracy')
plt.show()
```



4.4.4. Multi Layer networks using `nn.Sequential` PyTorch simplifies considerably the implementation of neural network training, since we do not need to implement derivatives ourselves

We can also make a simpler implementation of multilayer networks using `nn.Sequential` function

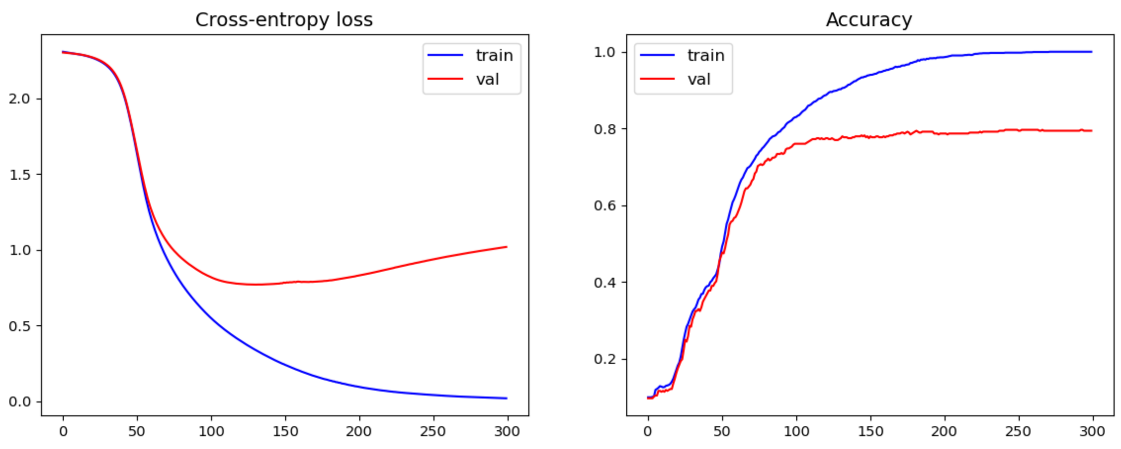
It returns directly a network with the requested topology, including parameters **and forward evaluation method**

[62]:
```
my_net = nn.Sequential(
    nn.Linear(X_train_torch.size()[1], 200),
    nn.ReLU(),
    nn.Linear(200,50),
    nn.ReLU(),
    nn.Linear(50,20),
    nn.ReLU(),
    nn.Linear(20,y_train_torch.size()[1])
)

opt = optim.SGD(my_net.parameters(), lr=0.01)
```

[63]:
```
epochs = 300

loss_train = np.zeros(epochs)
loss_val = np.zeros(epochs)
acc_train = np.zeros(epochs)
acc_val = np.zeros(epochs)

for epoch in range(epochs):

    print(f'Current epoch: {epoch + 1}  \r', end="")

    for xb, yb in train_dl:
        #Compute network output and cross-entropy loss for current minibatch
        pred = my_net(xb)
        loss = loss_func(pred, yb.argmax(axis=-1))

        #Compute gradients and optimize parameters
        loss.backward()
        opt.step()
        opt.zero_grad()

    #At the end of each epoch, evaluate overall network performance
    with torch.no_grad():
        #Computing network performance after iteration
        pred = my_net(X_train_torch)
        loss_train[epoch] = loss_func(pred, y_train_torch.argmax(axis=-1)).
 ↪item()
        acc_train[epoch] = accuracy(y_train_torch, pred).item()
        pred_val = my_net(X_val_torch)
        loss_val[epoch] = loss_func(pred_val, y_val_torch.argmax(axis=-1)).
 ↪item()
        acc_val[epoch] = accuracy(y_val_torch, pred_val).item()
```

```
Current epoch: 300
```

[64]:
```
plt.figure(figsize=(14,5))
plt.subplot(1, 2, 1), plt.plot(loss_train, 'b'), plt.plot(loss_val, 'r'), plt.
 ↪legend(['train', 'val']), plt.title('Cross-entropy loss')
plt.subplot(1, 2, 2), plt.plot(acc_train, 'b'), plt.plot(acc_val, 'r'), plt.
 ↪legend(['train', 'val']), plt.title('Accuracy')
plt.show()
```



```
[65]: print('Validation accuracy with this net:', acc_val[-1])
```

```
Validation accuracy with this net: 0.7941888570785522
```

1.4.5 4.5. Generalization

As the number of neurons and parameters in a neural network model grows, the training process can incur in **over-fitting** issues: the model learns to perform exceptionally well on the training data but fails to **generalize** effectively to new, unseen data: the models captures noise and specific details in the training data, rather than the underlying patterns. Overfit models have poor predictive performance on data they haven't seen before because they essentially memorize the training data rather than learning the true relationships.

A standard procedure to avoid overfiting is cross validation. We can train neural network configurations with different complexity, and select the most appropriate using a validation set, or through n-fold cross validation. However, the number of possible configurations (number of layers, neurons per layer, etc) may be too large, and combining cross-validation with other techniques is usually more efficient. These are some of them:

- **Regularization by a penalty term**: the loss function is extended with an additional term penalizing large weights. Two common examples of regularization for an empirical risk $R(\mathbf{w})$ are :
  - L2 regularization: $R(\mathbf{w}) + \lambda\|\mathbf{w}\|^2$
  - L1 regularization: $R(\mathbf{w}) + \lambda\|\mathbf{w}\|_1$
- **Early stopping**: It involves monitoring the model's performance on a validation dataset during training. If the validation performance starts to degrade (e.g., validation loss increases), training is stopped early to avoid overfitting and save the model with the best performance on the validation data.
- **Dropout regularization**: During training, dropout randomly "drops out" (deactivates) a fraction of neurons or units in a layer, preventing co-adaptation of neurons and reducing overfitting. It helps the model generalize better by making it more robust and less reliant on specific neurons during prediction. During inference, dropout is typically turned off, and all neurons are used.

Image Source

- **Data augmentation**: it involves applying random transformations to the training data to create additional training examples. These transformations can include operations like rotation, scaling, cropping, flipping, brightness adjustments, and more. By exposing the network to a wider variety of training examples, data augmentation helps the model learn to be invariant to these transformations and generalize better to unseen data.