# GRADIENTS WITHOUT BACKPROPAGATION

**Index:**

Julia Pérez Barreales, David Ramírez Palacios

# Description of the method

The method implements a forward AD system where we use forward gradients in stochastic gradient descent optimization. The forward gradient is an unbiased estimate of the gradient which can be computed in a forward run without using backpropagation in gradient descent.

For implementing forward mode AD over our objective functions (loss functions), cross entropy, we must compute the cross entropy of the model's parameters and the jacobian vector product $J_{\text{cross entropy}}$(parameters) * v (directional derivative) where v is a perturbation vector such that their scalar components are independent and have zero mean and unit variance. We are able to do these both computations, in a single step and without computing the jacobian of the cross entropy, using jvp from functorch.

Then, we compute the forward gradient of the cross entropy by multiplying the directional derivative ($J_{\text{cross entropy}}$(parameters) * v) with the perturbation vector (v).

For training the model, we must first initialize our parameters, that are the weights and biases of the neural networks, using random guessing, and calculate the start time and the loss obtained by the random parameters over the training set. We use a while loop to train the model until the loss is less than a threshold, this threshold will be different depending on the neural network. In the case of convolutional neural networks, we have decided to change the while loop to a for loop that performs a maximum number of iterations because of the high execution time of each epoch.

Then, we calculate the forward gradients using some partial functions for each parameter. After that, we update the parameters using the learning rate, this learning rate is different depending on the neural network and we have chosen it by proving it with different values. To update the parameters, we must multiply the learning rate with the forward gradient and subtract it from the parameters, as we do in backpropagation.

To compare our method with backpropagation, we have defined some vectors that accumulate the number of misclassifications over the test set, the accuracy of the model over the test set and the cross entropy of the model over the test using the updated parameters. Besides, we compute the execution time of each iteration and we add it to the total time to compare the difference in the time execution using forward gradient and backpropagation.

In each iteration, we print the execution time and the loss obtained over the training set. We use it to compare the increase of the execution time and the decrease of the loss in each iteration compared to the backpropagation method.

Our training method returns the parameters of the model optimized, the vectors that accumulate the number of misclassifications over the test set, the accuracy of the model over the test set and the cross entropy of the model over the test using all the parameters that have been calculated in the method. We plot these vectors to observe how accuracy is increasing over the time and how the loss and the errors are decreasing over the time.

Julia Pérez Barreales, David Ramírez Palacios

# Use of the method and the code

We have implemented our code with python using Google Colab, each document is commented but the general idea is the following:

We choose a dataset and define a Neural Network model. For each NN we create two documents, in one we train the model with forward gradient (our method) and in the other one we train the model with backpropagation. We store the execution time of the training and plot the evolution of the loss, the accuracy and the misclassifications made by the model in order to compare the results of the two differently trained models.

One of the parts of the code that we struggled to complete in the beginning was defining the condition to stop the training with forward gradient. We chose to stop the training when the function with respect to which we compute the gradient is optimized.

At first we tried several optimization functions such as the Beale function, the Rosenbrock function and the Sphere function. However, the method didn't work with these functions as it diverged and we realized that the function we needed to optimize was our loss function (the Cross-Entropy). Then, we realized that the Beale function, the Rosenbrock function and the Sphere function can be used as examples to try our method but we cannot use them as optimization functions in the Neural Networks that we had defined.

Let's see in more detail the models we implemented:

## Logistic Regression

We implemented a Logistic Regression model, with one linear layer (that has 2 parameters, weight and bias) and applied the softmax function to the output of the linear layer to obtain a probability distribution.

```python
class SimpleLogisticRegression(nn.Module):
  def __init__(self, input_size, w, b):
    super().__init__()
    self.weight = nn.Parameter(w)
    self.bias = nn.Parameter(b)

  def forward(self, x):
    x = x.reshape(1, -1)
    return torch.softmax(x@self.weight + self.bias, 1)
```

When we apply our model to an example, it results in a tensor with shape [1, 3] in which the component [1, i] represents the probability of the penguin of the example belonging to the species represented by class i.

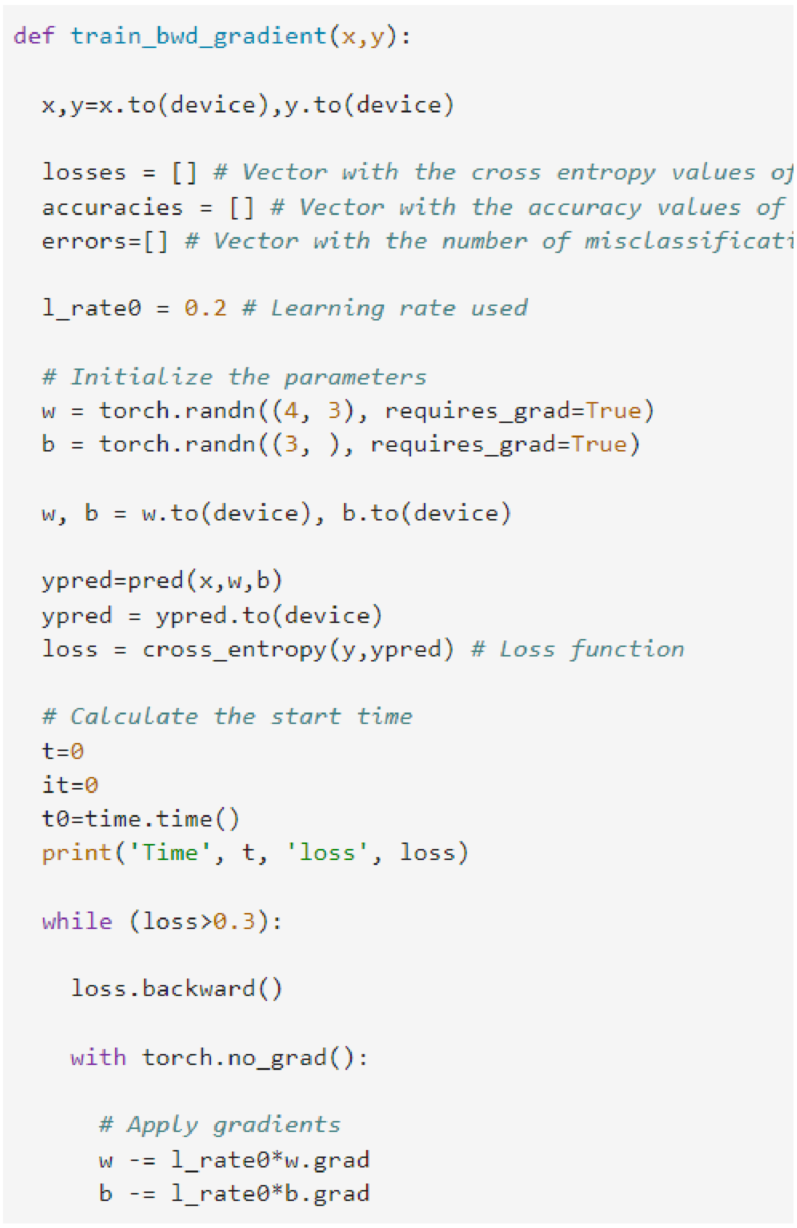
Training with forward gradient:

```
def train_fwd_gradient(x, y):
  x, y = x.to(device), y.to(device)

  losses = [] # Vector with the cross entropy values of test set
  accuracies = [] # Vector with the accuracy values of test set
  errors=[] # Vector with the number of misclassification of the test set

  l_rate0 = 0.2 # Learning rate used

  # Initialize the parameters
  w = torch.randn((4, 3),requires_grad=False)
  b = torch.randn((3, ),requires_grad=False)

  w, b = w.to(device), b.to(device)

  loss = cross_entropyW(w, y, x, b) # Loss function

  # Calculate the start time
  t=0
  it=0
  t0=time.time()
  print('Time', t, 'loss', loss)

  while (loss>0.3) :
```

```
# vw and wb are defined as a perturbation vector taken as a multivariate

vw=torch.randn(w.shape).to(device)
vw = (vw - torch.mean(vw))/torch.std(vw)
vb=torch.randn(b.shape).to(device)
vb = (vb - torch.mean(vb))/torch.std(vb)

# We define a partial function for each of the parameter
fw = partial(cross_entropyW,ytrue=y, x=x, b=b)
fb = partial(cross_entropyB,ytrue=y, x=x, w=w)

# We compute the cross entropy and the directional derivative of cross e
# This is also know as formard-mode autodiff
ftw, dtw = jvp(fw,(w, ), (vw, ))
ftb, dtb = jvp(fb,(b, ), (vb, ))

# We multiply the scalar directional derivative ∇cross_entropy(ϑ)·v with
gtw = vw*dtw
gtb = vb*dtb

# Update the parameters
w -= l_rate0*gtw
b -= l_rate0*gtb

# We calculate the number of misclassification of the test set with the
LG = SimpleLogisticRegression(4, w, b).to(device)
ypredT=torch.randn(Xtest.size(0),3).to(device)
error=0
for i in range (Xtest.size(0)):
  ypredT[i]=LG(Xtest[i].to(device))
  if (LG(Xtest[i].to(device)).argmax(1)- ytest[i])!=0:
    error = error+ 1

errors.append(error)
```

```
  # We calculate the accuracy of the test set with the updated mo
  accuracies.append(accuracy(ytest.to(device),ypredT).item())

  # We calculate the cross_entropy of the test set with the updat
  loss=cross_entropyW(w, y, x, b)
  lossT = cross_entropy(ytest,ypredT)
  losses.append(lossT.detach().item())

  #We add the execution time of the iteration
  t1=time.time()
  t+=t1-t0
  t0=t1
  it+=1

  print('Time', t, 'loss', loss)

print('Final execution time', t)
print('Number of iterations', it)
print('Mean execution time of an iteration', t/it)

return w, b,errors,accuracies,losses
```

Training with backpropagation:



```
def train_bwd_gradient(x,y):

  x,y=x.to(device),y.to(device)

  losses = [] # Vector with the cross entropy values o
  accuracies = [] # Vector with the accuracy values of
  errors=[] # Vector with the number of misclassificati

  l_rate0 = 0.2 # Learning rate used

  # Initialize the parameters
  w = torch.randn((4, 3), requires_grad=True)
  b = torch.randn((3, ), requires_grad=True)

  w, b = w.to(device), b.to(device)

  ypred=pred(x,w,b)
  ypred = ypred.to(device)
  loss = cross_entropy(y,ypred) # Loss function

  # Calculate the start time
  t=0
  it=0
  t0=time.time()
  print('Time', t, 'loss', loss)

  while (loss>0.3):

    loss.backward()

    with torch.no_grad():

      # Apply gradients
      w -= l_rate0*w.grad
      b -= l_rate0*b.grad
    # Gradients are accumulated: we need to zero them out befor
    w.grad.zero_()
    b.grad.zero_()

  # We calculate the number of misclassification of the test se
  LG = SimpleLogisticRegression(4, w, b).to(device)
  ypredT = torch.randn(Xtest.size(0),3)
  error=0
  for i in range (Xtest.size(0)):
    ypredT[i]=LG(Xtest[i].to(device))
    if (LG(Xtest[i].to(device)).argmax(1)- ytest[i])!=0:
      error = error+ 1
  errors.append(error)
  ypredT = ypredT.to(device)

  ypred=pred(x,w,b)

  # We calculate the accuracy of the test set with the updated
  accuracies.append(accuracy(ytest,ypredT).item())

  # We calculate the cross_entropy of the test set with the upd
  loss = cross_entropy(y,ypred.to(device))
  lossT = cross_entropy(ytest,ypredT)
  losses.append(lossT.detach().item())

  #We add the execution time of the iteration
  t1=time.time()
  t+=t1-t0
  t0=t1
  it+=1
  print('Time', t, 'loss', loss)

print('Final execution time', t)
print('Number of iterations', it)
print('Mean execution time of an iteration', t/it)

return w,b,errors,losses,accuracies
```

## Multilayer Neural Network

We implemented a Multilayer Neural Network model composed of three fully connected layers (each one has 2 parameters, weight and bias, so we had 6 parameters in total). We used the ReLU function as our activation function after the first two layers and applied the softmax function after the last layer.

```
class MNN(nn.Module):
  def __init__(self, input_size, fc1w, fc1b, fc2w, fc2b, fc3w, fc3b):
    super().__init__()

    self.w1 = nn.Parameter(fc1w)
    self.b1 = nn.Parameter(fc1b)

    self.w2 = nn.Parameter(fc2w)
    self.b2 = nn.Parameter(fc2b)

    self.w3 = nn.Parameter(fc3w)
    self.b3 = nn.Parameter(fc3b)

  def forward(self, x):
    x = F.relu(x@self.w1 + self.b1)
    x = F.relu(x@self.w2 + self.b2)
    x = x@self.w3 + self.b3
    x = x.reshape(1, -1)
    return torch.softmax(x, 1)
```

When we apply our model to an example, it results in a tensor with shape [1, 3] in which the component [1, i] represents the probability of the penguin of the example belonging to the species represented by class i.

Trained with forward gradient:

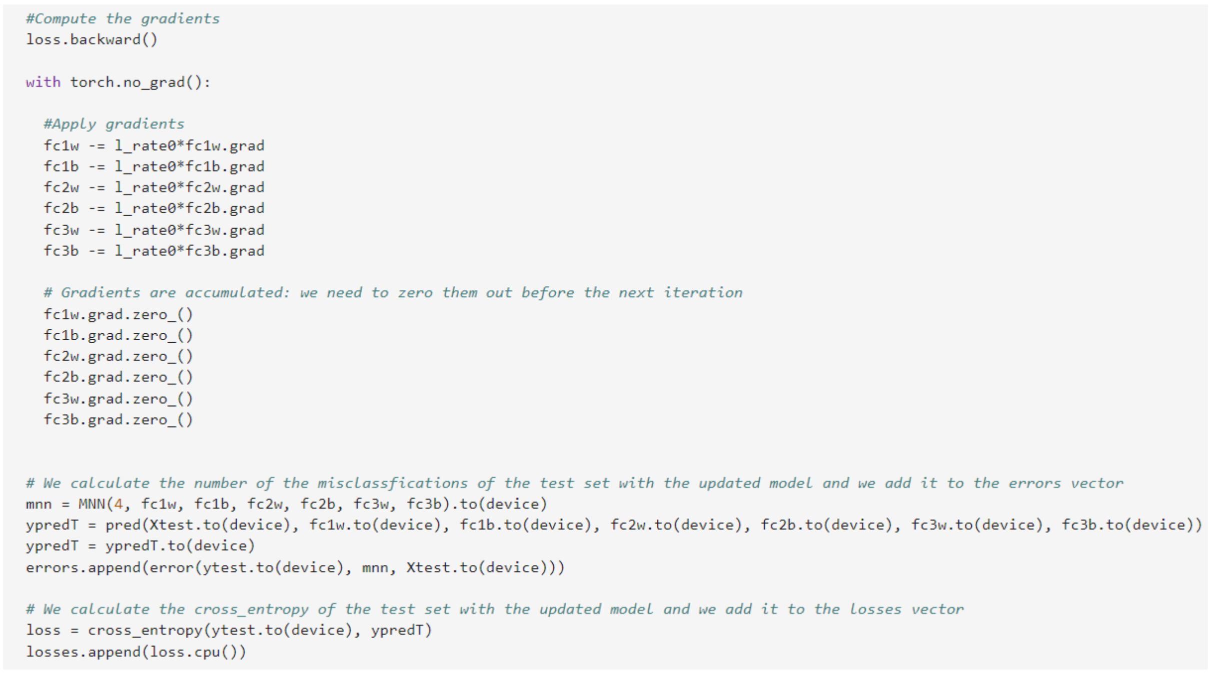
```
def train_fwd_gradient(x, y):
  x, y = x.to(device), y.to(device)

  losses = [] #Vector with the cross_entropy values of the test set
  accuracies = [] #Vector with the accuracy values of the test set
  errors = [] #Vector with the number of misclassifications of the test set

  l_rate0 = 0.05 #Learning rate used

  # Initialize the parameters
  fc1w = torch.FloatTensor(4, 64).uniform_(-1, 1)
  fc1b = torch.FloatTensor(64).uniform_(-1, 1)
  fc2w = torch.FloatTensor(64, 8).uniform_(-1, 1)
  fc2b = torch.FloatTensor(8).uniform_(-1, 1)
  fc3w = torch.FloatTensor(8, 3).uniform_(-1, 1)
  fc3b = torch.FloatTensor(3).uniform_(-1, 1)

  fc1w, fc1b, fc2w, fc2b, fc3w, fc3b = fc1w.to(device), fc1b.to(device), fc2w.to(device), fc2b.to(device), fc3w.to(device), fc3b.to(device)

  loss = cross_entropyWfc1(fc1w, fc1b, fc2w, fc2b, fc3w, fc3b, y, x) # Loss function

  t=0
  it=0
  t0 = time.time()

  while (loss>0.2) :
```

```
vfc1w=torch.randn(fc1w.shape).to(device)
vfc1w=(vfc1w - torch.mean(vfc1w))/torch.std(vfc1w)
vfc1b=torch.randn(fc1b.shape).to(device)
vfc1b=(vfc1b - torch.mean(vfc1b))/torch.std(vfc1b)
vfc2w=torch.randn(fc2w.shape).to(device)
vfc2w=(vfc2w - torch.mean(vfc2w))/torch.std(vfc2w)
vfc2b=torch.randn(fc2b.shape).to(device)
vfc2b=(vfc2b - torch.mean(vfc2b))/torch.std(vfc2b)
vfc3w=torch.randn(fc3w.shape).to(device)
vfc3w=(vfc3w - torch.mean(vfc3w))/torch.std(vfc3w)
vfc3b=torch.randn(fc3b.shape).to(device)
vfc3b=(vfc3b - torch.mean(vfc3b))/torch.std(vfc3b)

# We define a partial function for each of the parameters
fw1 = partial(cross_entropyWfc1, fc1b=fc1b, fc2w=fc2w, fc2b=fc2b, fc3w=fc3w, fc3b=fc3b, ytrue=y, x=x)
fb1 = partial(cross_entropyBfc1, fc1w=fc1w, fc2w=fc2w, fc2b=fc2b, fc3w=fc3w, fc3b=fc3b, ytrue=y, x=x)
fw2 = partial(cross_entropyWfc2, fc1b=fc1b, fc1w=fc1w, fc2b=fc2b, fc3w=fc3w, fc3b=fc3b, ytrue=y, x=x)
fb2 = partial(cross_entropyBfc2, fc1b=fc1b, fc2w=fc2w, fc1w=fc1w, fc3w=fc3w, fc3b=fc3b, ytrue=y, x=x)
fw3 = partial(cross_entropyWfc3, fc1b=fc1b, fc2w=fc2w, fc2b=fc2b, fc1w=fc1w, fc3b=fc3b, ytrue=y, x=x)
fb3 = partial(cross_entropyBfc3, fc1b=fc1b, fc2w=fc2w, fc2b=fc2b, fc3w=fc3w, fc1w=fc1w, ytrue=y, x=x)

# We compute the cross_entropy and the directional derivative of cross_entropy at each parameter in dir
# ∇cross_entropy in the process
#This is also known as forward-mode autodiff
ftw1, dtw1=jvp(fw1,(fc1w, ), (vfc1w, ))
ftb1, dtb1=jvp(fb1,(fc1b, ), (vfc1b, ))
ftw2, dtw2=jvp(fw2,(fc2w, ), (vfc2w, ))
ftb2, dtb2=jvp(fb2,(fc2b, ), (vfc2b, ))
ftw3, dtw3=jvp(fw3,(fc3w, ), (vfc3w, ))
ftb3, dtb3=jvp(fb3,(fc3b, ), (vfc3b, ))

# We multiply the scalar directional derivative ∇cross_entropy(ϑ)·v with vector v and obtain g(ϑ), the
gtw1 = vfc1w*dtw1
gtb1 = vfc1b*dtb1
gtw2 = vfc2w*dtw2
gtb2 = vfc2b*dtb2
gtw3 = vfc3w*dtw3
gtb3 = vfc3b*dtb3
```

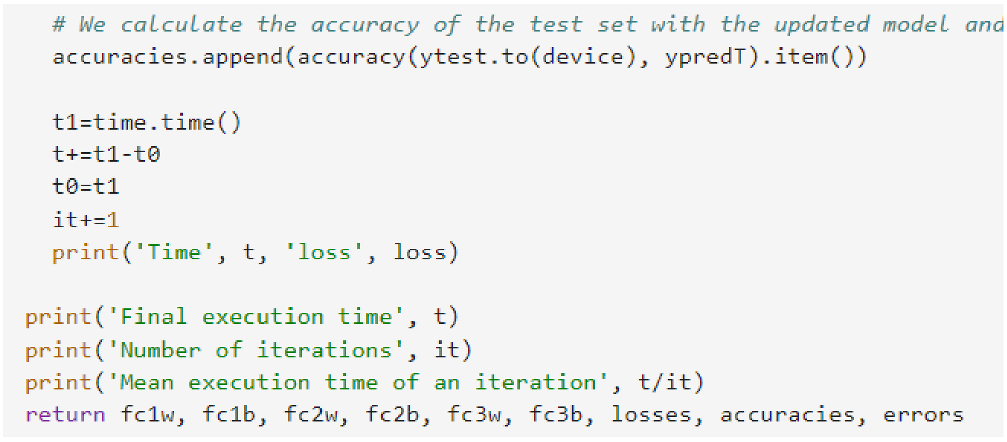
```
  # Update the parameters
  fc1w -= l_rate0*gtw1
  fc1b -= l_rate0*gtb1
  fc2w -= l_rate0*gtw2
  fc2b -= l_rate0*gtb2
  fc3w -= l_rate0*gtw3
  fc3b -= l_rate0*gtb3

  # We calculate the number of the misclassfications of the test set with the updated model and we add it to the
  mnn = MNN(4, fc1w, fc1b, fc2w, fc2b, fc3w, fc3b)
  ypred = pred(Xtest.to(device), fc1w, fc1b, fc2w, fc2b, fc3w, fc3b)
  errors.append(error(ytest, mnn, Xtest))

  # We calculate the cross_entropy of the test set with the updated model and we add it to the losses vector
  losses.append(cross_entropyWfc1(fc1w, fc1b, fc2w, fc2b, fc3w, fc3b, ytest.to(device), Xtest.to(device)).cpu())

  # We calculate the accuracy of the test set with the updated model and we add it to the accuracies vector
  accuracies.append(accuracy(ytest, ypred).item())

  # We calculate the loss
  loss = cross_entropyWfc1(fc1w, fc1b, fc2w, fc2b, fc3w, fc3b, y, x)

  # We add the execution time of the iteration
  t1=time.time()
  t+=t1-t0
  t0=t1
  it+=1

  print('Time', t, 'loss', loss)

print('Final execution time', t)
print('Number of iterations', it)
print('Mean execution time of an iteration', t/it)

return fc1w, fc1b, fc2w, fc2b, fc3w, fc3b, losses, accuracies, errors
```

Training with backpropagation:

```
def train_bwd_gradient(x, y):
  x, y = x.to(device), y.to(device)

  losses = [] #Vector with the cross_entropy values of the test set
  accuracies = [] #Vector with the accuracy values of the test set
  errors = [] #Vector with the number of misclassifications of the test set

  l_rate0 = 0.05 #Learning rate used

  #Initialize the parameters
  fc1w = torch.FloatTensor(4, 64).uniform_(-1, 1).requires_grad_(True)
  fc1b = torch.FloatTensor(64).uniform_(-1, 1).requires_grad_(True)
  fc2w = torch.FloatTensor(64, 8).uniform_(-1, 1).requires_grad_(True)
  fc2b = torch.FloatTensor(8).uniform_(-1, 1).requires_grad_(True)
  fc3w = torch.FloatTensor(8, 3).uniform_(-1, 1).requires_grad_(True)
  fc3b = torch.FloatTensor(3).uniform_(-1, 1).requires_grad_(True)

  ypred = pred(x, fc1w.to(device), fc1b.to(device), fc2w.to(device), fc2b.to(device), fc3w.to(device), fc3b.to(device))
  ypred = ypred.to(device)
  loss = cross_entropy(y, ypred) # Loss function

  it=0
  t = 0
  t0 = time.time()

  while loss>0.2:
```



```
#Compute the gradients
loss.backward()

with torch.no_grad():

  #Apply gradients
  fc1w -= l_rate0*fc1w.grad
  fc1b -= l_rate0*fc1b.grad
  fc2w -= l_rate0*fc2w.grad
  fc2b -= l_rate0*fc2b.grad
  fc3w -= l_rate0*fc3w.grad
  fc3b -= l_rate0*fc3b.grad

  # Gradients are accumulated: we need to zero them out before the next iteration
  fc1w.grad.zero_()
  fc1b.grad.zero_()
  fc2w.grad.zero_()
  fc2b.grad.zero_()
  fc3w.grad.zero_()
  fc3b.grad.zero_()


# We calculate the number of the misclassfications of the test set with the updated model and we add it to the errors vector
mnn = MNN(4, fc1w, fc1b, fc2w, fc2b, fc3w, fc3b).to(device)
ypredT = pred(Xtest.to(device), fc1w.to(device), fc1b.to(device), fc2w.to(device), fc2b.to(device), fc3w.to(device), fc3b.to(device))
ypredT = ypredT.to(device)
errors.append(error(ytest.to(device), mnn, Xtest.to(device)))

# We calculate the cross_entropy of the test set with the updated model and we add it to the losses vector
loss = cross_entropy(ytest.to(device), ypredT)
losses.append(loss.cpu())
```



```
  # We calculate the accuracy of the test set with the updated model and
  accuracies.append(accuracy(ytest.to(device), ypredT).item())

  t1=time.time()
  t+=t1-t0
  t0=t1
  it+=1
  print('Time', t, 'loss', loss)

print('Final execution time', t)
print('Number of iterations', it)
print('Mean execution time of an iteration', t/it)
return fc1w, fc1b, fc2w, fc2b, fc3w, fc3b, losses, accuracies, errors
```

## Convolutional Neural Network

We implemented a Convolutional Neural Network model formed by (in order) two convolutional 2d layers, one max pooling layer and two fully connected layers. We used the ReLU function as our activation function after the convolution layers and after the first fully connected layer and applied the softmax function after the last layer.

```python
class SimpleCNN(nn.Module):
    def __init__(self, input_size):
        super().__init__()
        self.conv1 = nn.Conv2d(input_size, 2, 3, padding=1)

        self.conv2 = nn.Conv2d(2, 4, 3, padding=1)

        self.max_pool = nn.MaxPool2d(2)
        self.fc1 = nn.Linear(4*14*14, 64)

        self.fc2 = nn.Linear(64, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = self.max_pool(x)
        x = x.reshape((-1, 4*14*14))
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        x = x.reshape(1, -1)
        return torch.softmax(x, 1)
```

When we apply our model to an example, it results in a tensor with shape [1, 10] in which the component [1, i] represents the probability of the image being the digit represented by class i (0, 1, 2, 3, 4, 5, 6, 7, 8, or 9).

Training with forward gradient:

```python
def train_fwd_gradient(train_loader):

  l_rate0 = 0.001

  losses=[]
  accuracies=[]

  cnn = SimpleCNN(1)
  fcnn, params = fc.make_functional(cnn)
  Xtrain, ytrain, ypred = pred(500, train_loader, cnn)

  Xtest, ytest, ypredtest = pred(100, test_loader, cnn)


  loss = cross_entropy(params, fcnn, ytrain, Xtrain)
  t=0
  t0 = time.time()

  for epoch in range (100):

    v = tuple([torch.randn_like(p) for p in params])

    g = partial(cross_entropy, fmodel = fcnn, ytrue=ytrain, x=Xtrain)

    loss, dt = jvp(g, (params, ), (v, ))

    with torch.no_grad():
      for j, p in enumerate(params):
        gt = v[j]*dt
        p -= l_rate0*gt

    Xtest, ytest, ypredtest = pred(100, test_loader, cnn)
    accuracies.append(accuracy(ytest,ypredtest).item())
    losses.append(cross_entropy(params,fcnn,ytest,Xtest).item())

    # We add the execution time of the iteration
    t1=time.time()
    t+=t1-t0
    t0=t1
```

```python
    print('Time', t, 'loss', loss)

  print('Final execution time', t)
  print('Mean execution time of an iteration', t/100)

  return params,losses,accuracies
```

Training with backpropagation:

```
loss = nn.CrossEntropyLoss()
opt = torch.optim.Adam(cnn.parameters())
accuracies=[]
losses=[]

t=0
t0 = time.time()
for epoch in range(100):

  cnn.train()
  for i in range(499):
    xb, yb=Xtrain[i],ytrain[i]
    xb, yb = xb.to(device), yb.to(device)

    opt.zero_grad()
    ypred = cnn(xb)
    yb = yb.reshape(1)
    l = loss(ypred, yb.long())
    l.backward()
    opt.step()

  Xtest, ytest, ypredtest = pred(100, test_loader, cnn)
  accuracies.append(accuracy(ytest,ypredtest).item())
  loss2=0
  for i in range(ypredtest.size(0)):
    loss2+=loss(ypredtest[i], ytest[i].long())
  losses.append((loss2/ypredtest.size(0)).detach().item())

  loss1 = 0
  for i in range(Xtrain.size(0)):
    ytraini = ytrain[i].reshape(1)
    loss1 += loss(cnn(Xtrain[i]), ytraini.long())/Xtrain.size(0)

  # We add the execution time of the iteration
  t1=time.time()
  t+=t1-t0
  t0=t1
```

```
  print('Time', t, 'loss', loss1)

print('Final execution time', t)
print('Mean execution time of an iteration', t/100)
```

## Main results

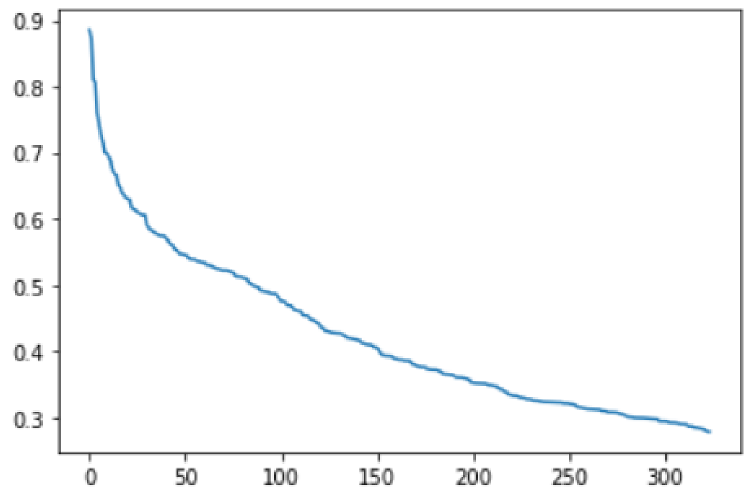
In this section, we will compare each of our NN models trained with forward gradient and the same model trained with backpropagation.
In all of the following cases we will initialize the parameters randomly but, in the Multilayer Neural Network model in particular, the random values are chosen between -1 and 1 as we have observed that the model often diverges when the values are out of this range.

## Logistic Regression

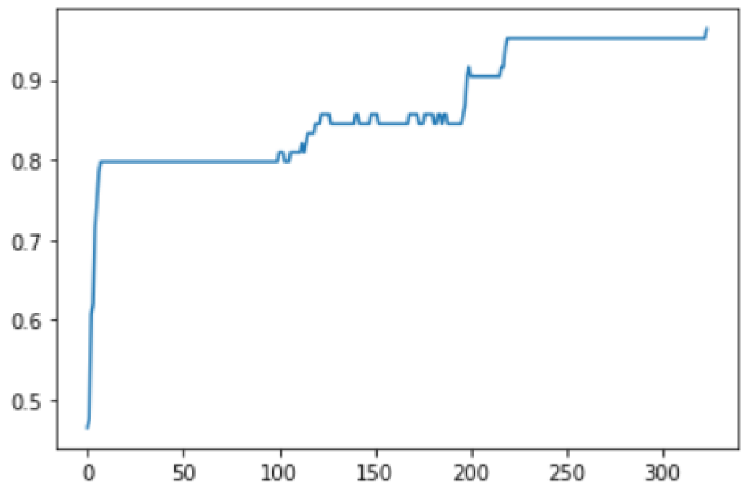
### Trained with forward gradient

Learning rate used: 0.2
Threshold used: 0.3

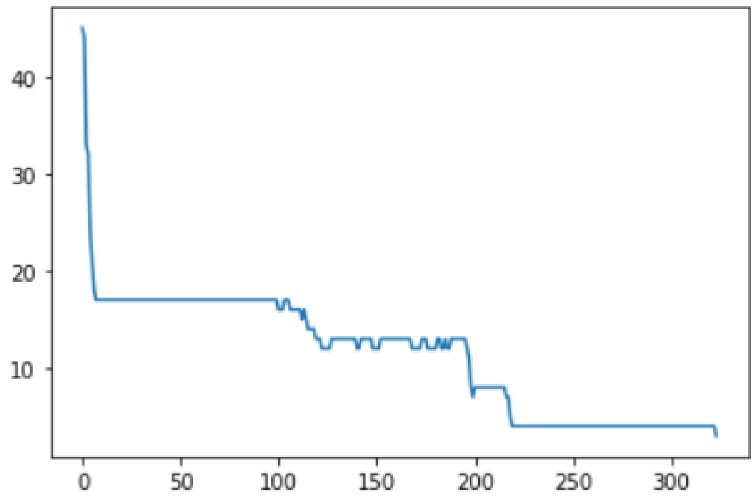
Evolution of the loss made by the model over the test set:



Evolution of the accuracy made by the model over the test set:



Evolution of the misclassifications made by the model over the test set:

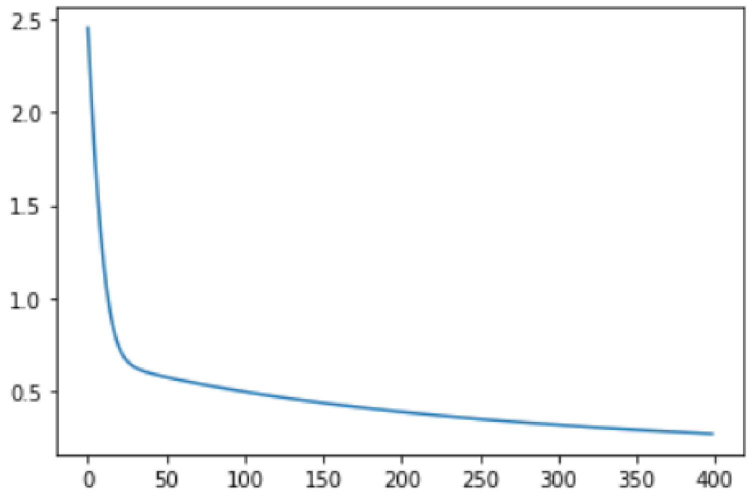


Mean execution time of an iteration: `0.24508204725053576`
Final execution time: `79.40658330917358`
Number of iterations: `324`

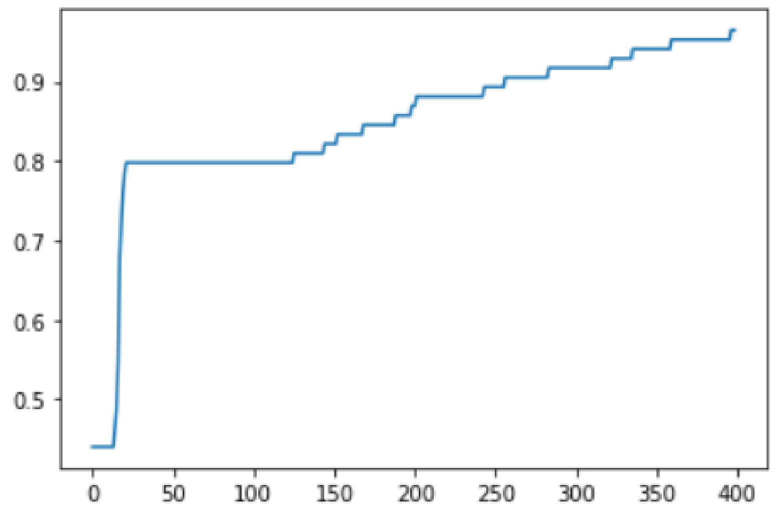
## Trained with backpropagation

Learning rate used: 0.2
Threshold used: 0.3

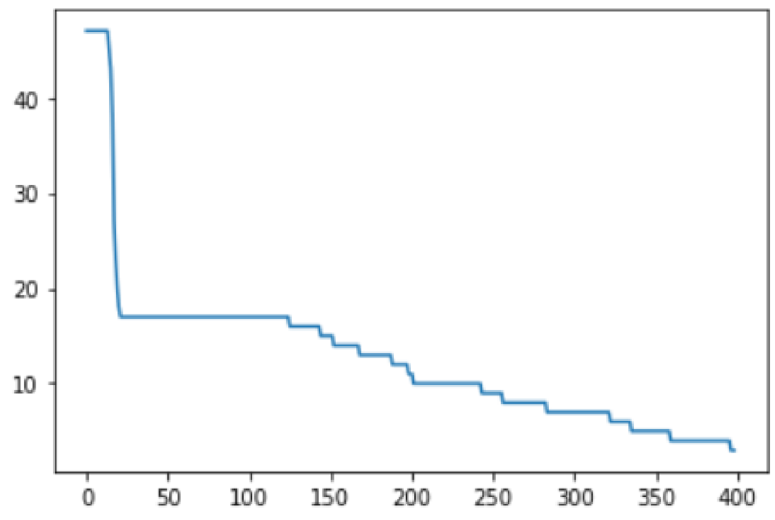
Evolution of the loss over the test set:



Evolution of the accuracy of the model over the test set:



Evolution of the misclassifications made by the model over the test set:



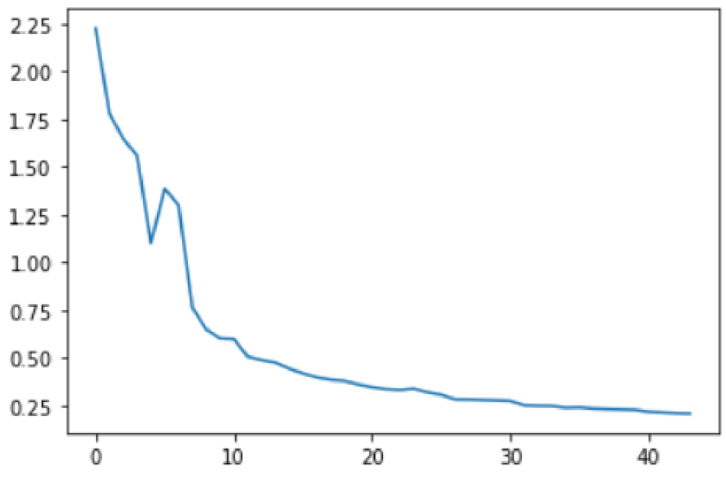
Mean execution time of an iteration: `0.08104073911681212`
Final execution time: `32.33525490760803`
Number of iterations: `399`

# Multilayer Neural Network

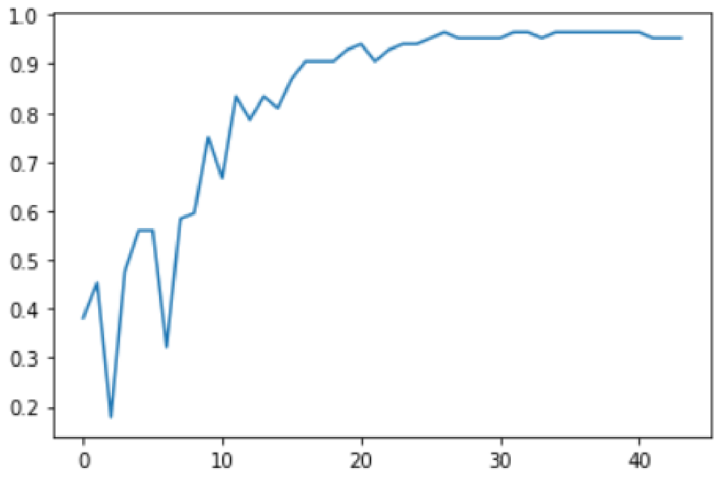
## Trained with forward gradient

Learning rate used: 0.05
Threshold used: 0.2

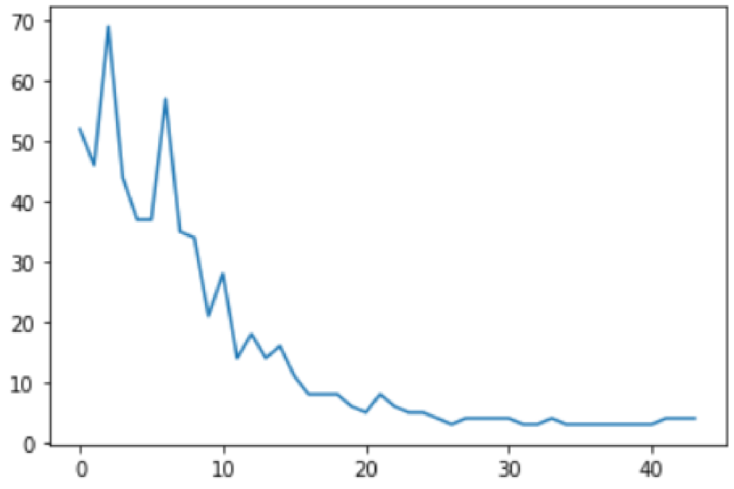
Evolution of the loss over the test set:



Evolution of the accuracy of the model over the test set:



Evolution of the misclassifications made by the model over the test set:

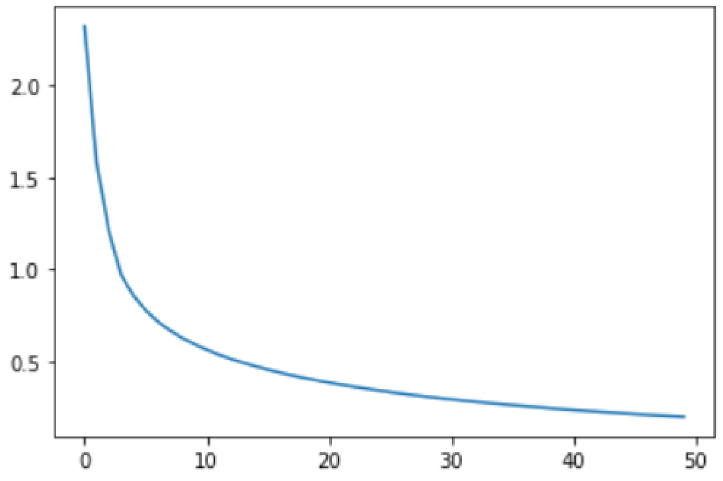


Mean execution time of an iteration: `1.2015694975852966`
Final execution time: `52.86905789375305`
Number of iterations: `44`

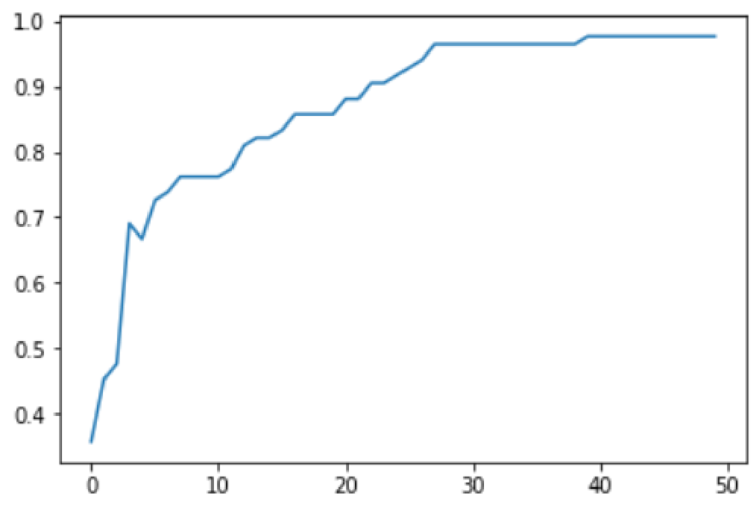
## Trained with backpropagation

Learning rate used: 0.05
Threshold used: 0.2

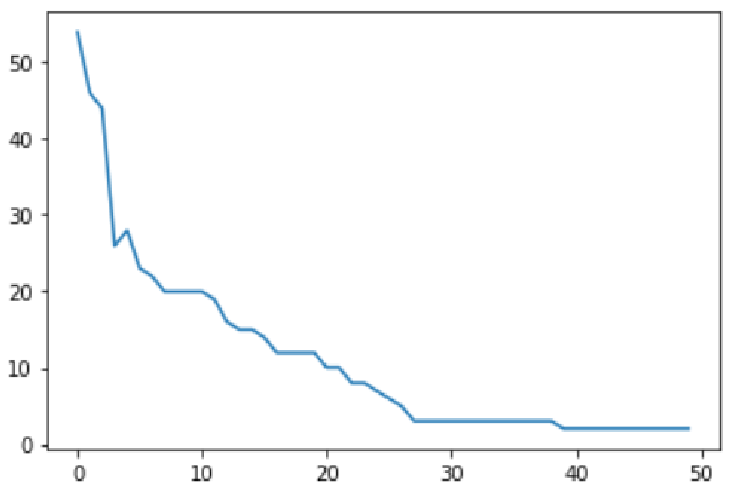
Evolution of the loss over the test set:



Evolution of the accuracy of the model over the test set:



Evolution of the misclassifications made by the model over the test set:



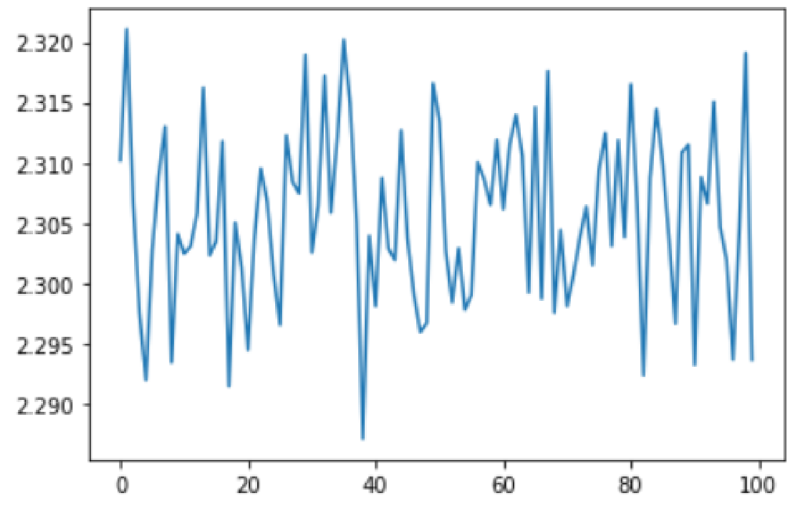
Mean execution time of an iteration: `0.06628170967102051`
Final execution time: `3.3140854835510254`
Number of iterations: `50`

# Convolutional Neural Network

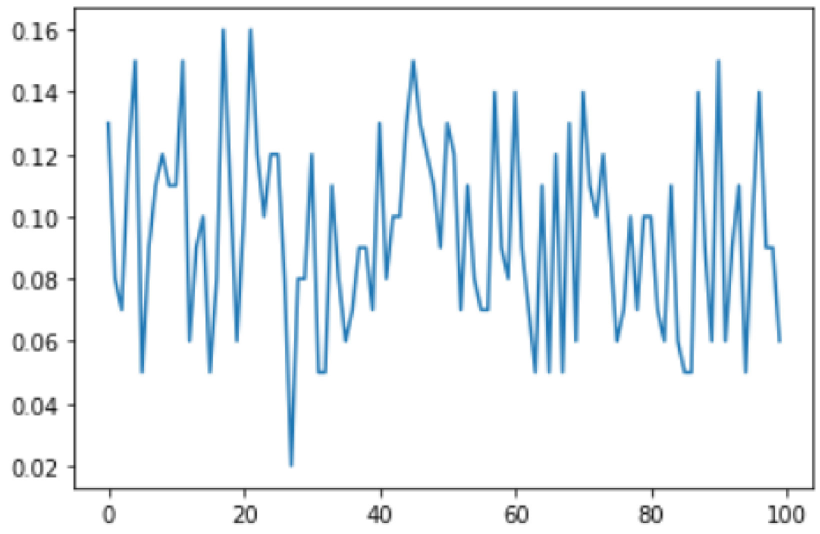
## Trained with forward gradient

Learning rate used: 0.001

Evolution of the loss over the test set:



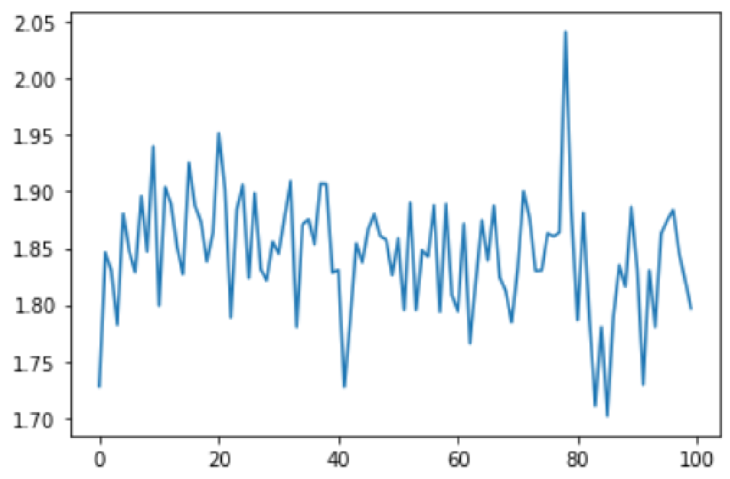
Evolution of the accuracy of the model over the test set:



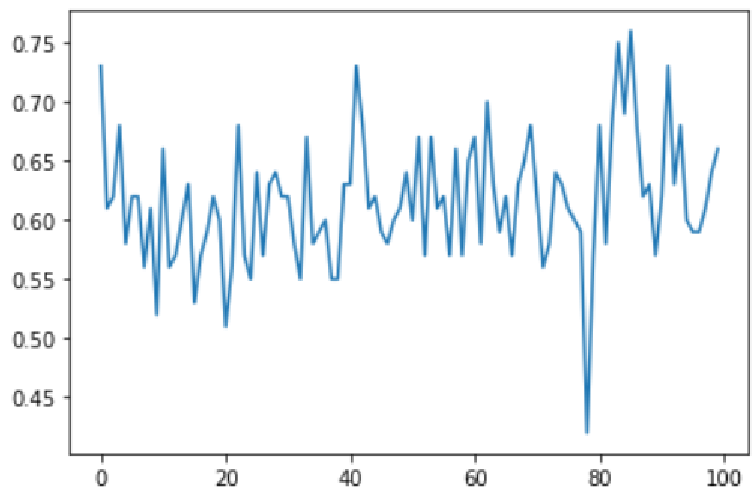
Mean execution time of an iteration: `0.9833761334419251`
Final execution time: `98.3376133441925`
Number of iterations: `100`

## Trained with backpropagation

Learning rate used: 0.001
Evolution of the loss over the test set:



Evolution of the accuracy of the model over the test set:



Mean execution time of an iteration: `1.7257762551307678`
Final execution time: `172.57762551307678`
Number of iterations: `100`

# Conclusions

## Logistic Regression model:

We have made the decision to take a big learning rate (0.2) to speed up the convergence of the model. We tried with other smaller learning rates but the time execution was very high. We have made the decision to take a bigger threshold than in multilayer neural networks to also speed up the convergence of the model.

In both types of training, we obtained that the loss and misclassifications over the test set are decreasing over the time and the accuracy is increasing over the time.

The model trained with backpropagation takes much less time in the training and each iteration in the forward gradient takes longer, although the model trained with forward gradient does less iterations.

For these reasons, training with forward gradient will be more efficient than training with backpropagation for logistic regression models if we are able to reduce the execution time of each iteration.

## Multilayer Neural Network model

We have made the decision to take a smaller learning rate (0.05) because of the high convergence speed of the model. We tried with other learning rates and with this one we obtained the best results. Besides, we have made the decision to take a smaller threshold than in the Logistic Regression because of the high convergence speed of the model.

In the Multilayer Neural Network model, the model trained with backpropagation is faster than the model trained with forward gradient although the difference between training times is not as significant as it was in Logistic Regression. The model trained with forward gradient does less iterations but each iteration takes longer than the ones in the model trained with backpropagation.

In both types of training, we obtained that the loss and misclassifications over the test set are decreasing over the time and the accuracy is increasing over the time.

## Convolutional Neural Network model

We had many problems computing the forward gradient in this model because of the appearance of convolutional layers. We tried to compute it with different implementations and we think that we have computed it in a proper way. Although when we try to apply it to the convolutional neural network, its results are very poor.

In both types of training, we obtained that the loss over the test set does not decrease much and the accuracy does not increase much. The convergence of the model is too slow for this reason we have decided to change the convergence condition using the loss and a threshold for a number fixed of iterations. Training with backpropagation is more efficient in this case because the accuracy is higher and the loss is lower.

It is also worth mentioning that each iteration of training the Convolutional Neural Network with backpropagation takes longer than the ones in the model trained with forward gradient.

Based on these results, we came to the conclusion that if we are able to reduce the mean execution time of an iteration training a model with the forward gradient will be more efficient than training it with backpropagation since it does less iterations in each model. However, at the moment these two techniques give similar results and training with backpropagation is faster.