```
# Projet TP groupe 3
# PCA_ANN_2_couches
```

```
# Nous avons utilisé les étapes suivantes
# 1. Réduction de la dimension des variables avec PCA
# 2. Nous avons utilisé un ANN pour notre apprentissage
```

```
# Importation des librairies

begin
    using DelimitedFiles
    using Plots
    using Statistics
    using LinearAlgebra
    using Flux

    using Flux: onehotbatch
    using Flux: @epochs
    using PlutoUI
end
```

```
# Importation des données d'entrainement
```

```
Int64[5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,    more
```

```
begin
    path_dataset = "UCI HAR Dataset/"
    dataset_train = readdlm(path_dataset * "train/X_train_new.txt", ' ')
    X_train = dataset_train[:, 1:561]
    y_train = Int.(dataset_train[:,563])
end
```

```
7352×561 Array{Float64,2}:
 0.288585  -0.0202942   -0.132905   -0.995279  …  -0.841247  0.179941  -0.0586269
 0.278419  -0.0164106   -0.12352    -0.998245     -0.844788  0.180289  -0.0543167
 0.279653  -0.0194672   -0.113462   -0.99538      -0.848933  0.180637  -0.0491178
 0.279174  -0.0262006   -0.123283   -0.996091     -0.848649  0.181935  -0.0476632
 0.276629  -0.0165697   -0.115362   -0.998139     -0.847865  0.185151  -0.0438923
 0.277199  -0.0100979   -0.105137   -0.997335  …  -0.849632  0.184823  -0.0421264
 0.279454  -0.0196408   -0.110022   -0.996921     -0.85215   0.18217   -0.04301
 ⋮                                              ⋱                       ⋮
 0.237966  -0.00108781  -0.148326   -0.218949     -0.797272  0.234996   0.048907
 0.299665  -0.0571934   -0.181233   -0.195387     -0.791883  0.238604   0.0498191
 0.273853  -0.00774933  -0.147468   -0.235309     -0.77184   0.252676   0.0500526
 0.273387  -0.0170106   -0.0450218  -0.218218     -0.779133  0.249145   0.0408112
 0.289654  -0.018843    -0.158281   -0.219139  …  -0.785181  0.246432   0.0253395
 0.351503  -0.0124231   -0.203867   -0.26927      -0.783267  0.246809   0.0366948
```

```
X_train
```

```
Int64[5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,    more
```

```
y_train
```

```
# Importation des données de test
```

```
Int64[5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,     more
```

```julia
begin
    dataset_test = readdlm(path_dataset * "test/X_test_new.txt", ' ')
    X_test = dataset_test[:, 1:561]
    y_test = Int.(dataset_test[:,563])
end
```

```
2947×561 Array{Float64,2}:
 0.257178  -0.0232852  -0.0146538  -0.938404  …  -0.720009  0.276801  -0.0579783
 0.286027  -0.0131634  -0.119083   -0.975415     -0.698091  0.281343  -0.083898
 0.275485  -0.0260504  -0.118152   -0.993819     -0.702771  0.280083  -0.0793462
 0.270298  -0.0326139  -0.11752    -0.994743     -0.698954  0.284114  -0.077108
 0.274833  -0.0278478  -0.129527   -0.993852     -0.692245  0.290722  -0.0738568
 0.27922   -0.0186204  -0.113902   -0.994455  …  -0.689816  0.294896  -0.0684707
 0.279746  -0.018271   -0.104      -0.995819     -0.690085  0.295282  -0.0670653
 ⋮                                            ⋱                        ⋮
 0.192275  -0.0336426  -0.105949   -0.354841     -0.646754  0.28215    0.181152
 0.310155  -0.0533913  -0.0991087  -0.287866     -0.651732  0.274627   0.184784
 0.363385  -0.039214   -0.105915   -0.305388     -0.655181  0.273578   0.182412
 0.349966   0.0300774  -0.115788   -0.329638     -0.655357  0.274479   0.181184
 0.237594   0.0184669  -0.0964989  -0.323114  …  -0.659719  0.264782   0.187563
 0.153627  -0.0184365  -0.137018   -0.330046     -0.66008   0.263936   0.188103
```

```julia
X_test
```

```
Int64[5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,     more
```

```julia
y_test
```

Utilisation de PCA pour réduire la dimension des données

Algorithme :

1. Centrer les données
2. Construire la matrice de covariance Σ
3. Décomposer cette matrice en vecteur propres,valeur propres {vi, λi}
4. Ordonner les valeurs propres par ordre décroissant
5. Le sous-espace de dimension q qui représente 99% de la variance des données est utilisé pour le modèle ANN

```julia
md"#### Utilisation de PCA pour réduire la dimension des données

Algorithme :

1. Centrer les données
2. Construire la matrice de covariance Σ
3. Décomposer cette matrice en vecteur propres,valeur propres {vi, λi}
4. Ordonner les valeurs propres par ordre décroissant
5. Le sous-espace de dimension q qui représente 99% de la variance des données est utilisé pour le modèle ANN
"
```

```
μ = 1×561 Array{Float64,2}:
     0.274488  -0.0176954  -0.109141  -0.605438  …  -0.489547  0.058593  -0.0565147
```

```julia
# Calcul de la moyenne de chaque colonne
μ = mean(X_train, dims=1)
```

```
M = 7352×561 Array{Float64,2}:
 0.274488  -0.0176954  -0.109141  -0.605438  …  -0.489547  0.058593  -0.0565147
 0.274488  -0.0176954  -0.109141  -0.605438     -0.489547  0.058593  -0.0565147
 0.274488  -0.0176954  -0.109141  -0.605438     -0.489547  0.058593  -0.0565147
 0.274488  -0.0176954  -0.109141  -0.605438     -0.489547  0.058593  -0.0565147
 0.274488  -0.0176954  -0.109141  -0.605438     -0.489547  0.058593  -0.0565147
 0.274488  -0.0176954  -0.109141  -0.605438  …  -0.489547  0.058593  -0.0565147
 0.274488  -0.0176954  -0.109141  -0.605438     -0.489547  0.058593  -0.0565147
 ⋮                                           ⋱                        ⋮
 0.274488  -0.0176954  -0.109141  -0.605438     -0.489547  0.058593  -0.0565147
 0.274488  -0.0176954  -0.109141  -0.605438     -0.489547  0.058593  -0.0565147
 0.274488  -0.0176954  -0.109141  -0.605438     -0.489547  0.058593  -0.0565147
 0.274488  -0.0176954  -0.109141  -0.605438     -0.489547  0.058593  -0.0565147
 0.274488  -0.0176954  -0.109141  -0.605438  …  -0.489547  0.058593  -0.0565147
 0.274488  -0.0176954  -0.109141  -0.605438     -0.489547  0.058593  -0.0565147
```

```
M = repeat(μ, length(y_train), 1)
```

```
X̄ = 7352×561 Array{Float64,2}:
  0.0140964    -0.00259874   -0.0237641   …  -0.3517    0.121348  -0.00211222
  0.00393071    0.00128486   -0.0143792      -0.35524   0.121696   0.00219799
  0.00516494   -0.00177173   -0.00432067     -0.359386  0.122044   0.00739689
  0.00468582   -0.00850522   -0.0141415      -0.359102  0.123342   0.00885152
  0.00214065    0.00112577   -0.00622083     -0.358318  0.126558   0.0126224
  0.00271065    0.00759758    0.00400377  …  -0.360084  0.126229   0.0143883
  0.00496576   -0.00194535   -0.00088113     -0.362603  0.123577   0.0135047
  ⋮                                       ⋱                        ⋮
 -0.0365216     0.0166076    -0.0391849      -0.307725  0.176403   0.105422
  0.0251772    -0.039498     -0.072092       -0.302336  0.180011   0.106334
 -0.000635415   0.0099461    -0.0383273      -0.282292  0.194083   0.106567
 -0.00110075    0.000684811   0.0641192      -0.289585  0.190552   0.0973259
  0.015166     -0.00114762   -0.0491396   …  -0.295634  0.187839   0.0818542
  0.0770153     0.00527231   -0.0947261      -0.29372   0.188215   0.0932095
```

```
# Centrage des données
X̄ = X_train - M
```

```
1×561 Array{Float64,2}:
 -2.22286e-17  2.02353e-18  5.07392e-18  …  4.25243e-17  -4.83231e-19  -3.86585e-18
```

```
mean(X̄, dims=1) # La moyenne de chaque colonne est bien proche de 0
```

```
Σ = 561×561 Array{Float64,2}:
  0.00493665    0.000424552  -0.00102248   …   0.00071841    0.000553863
  0.000424552   0.0016655    -0.000182059      1.27823e-5   -0.000158369
 -0.00102248   -0.000182059   0.00320754      -0.000257575  -0.000357949
  1.95065e-5   -0.000827027  -0.000513799      0.0628822     0.0494525
 -0.00077354   -0.000921444  -0.000473728      0.0782923     0.0607733
 -0.00131251   -0.000850007  -0.000199433  …   0.059287      0.0564256
  0.00018742   -0.000764614  -0.000450244      0.0588409     0.0462727
  ⋮                                         ⋱                 ⋮
  0.00038342   -0.000247742  -0.00101143   …  -0.0017324    -0.00219228
  0.00160038    0.000446035  -0.00219142      -0.00222809   -0.00337941
  0.000968681   0.00147623   -0.000921396     -0.000834531  -0.0007546
 -0.00126785   -0.000110889   0.000248914     -0.119343     -0.0919504
  0.00071841    1.27823e-5   -0.000257575      0.0884944     0.0493952
  0.000553863  -0.000158369  -0.000357949  …   0.0493952     0.0779092
```

```
# Construction la matrice de covariance Σ
Σ = cov(X̄) # Ou B'*B/(length(y_train) - 1)
```

```
Eigen{Float64,Float64,Array{Float64,2},Array{Float64,1}}
values:
561-element Array{Float64,1}:
 -2.3818397070201475e-16
 -1.803340291544898e-16
 -3.718746783291804e-17
 -1.4953023566224664e-17
```

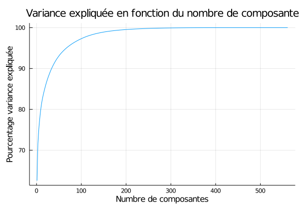
```
 -8.448811245005369e-18
 -6.961354641112536e-18
 -6.576259941595891e-18
  ⋮
  0.7081523037131603
  0.9435170027830603
  1.0437752946684833
  2.2943928417169275
  2.7350462652009018
 34.82363040636593
vectors:
561×561 Array{Float64,2}:
  1.13183e-9   -1.05098e-9   -5.56484e-10  …   0.00325696    7.15327e-5
 -7.71805e-10   6.19934e-10  -1.79635e-9      -0.000422311   0.000299848
  3.11581e-10  -7.50608e-10  -5.01163e-10     -0.000839509   0.000231385
  1.43788e-8   -2.07142e-8   -1.88137e-8       0.0171947    -0.0728429
 -1.30439e-8   -1.02185e-8    4.46897e-8      -0.0109697    -0.0830195
  7.89386e-9    6.9804e-9    -1.16705e-8   …  -0.0240049    -0.0657876
  1.10704e-10   5.47116e-11   3.31847e-9       0.0180922    -0.0684675
```

```
# Décomposer cette matrice en vecteur propres,valeur propres {vi, λi}
λ, v = eigen(Σ)
```

```
order =
 Int64[561, 560, 559, 558, 557, 556, 555, 554, 553, 552, 551, 550, 549, 548, 54
```

```
# Ordonner les valeurs propres par ordre décroissant
order = sortperm(λ, rev=true)
```

```
# Recherche du nombre de composant permettant de conserver au moins 99% de la
variance
begin
    eboli = float([0 for idx in 1:561])
    λ_drec = λ[order]
    total = sum(λ_drec)
    for i in 1:561
        som_i = sum(λ_drec[1:i])
        eboli[i] = som_i / total * 100
    end
end
```




```
plot(1:561, eboli, legend=false, xlabel="Numbre de composantes", ylabel="Pourcentage
variance expliquée", title="Variance expliquée en fonction du nombre de composante")
```

```
begin
    print("Le pourcentage de la variance conservée avec 155 composantes est : ")
    print(eboli[155])
end

# Nous alons donc utiliser 155 composantes
```

```
θ = 561×155 Array{Float64,2}:
      7.15327e-5    0.00325696    0.00245574   …  -0.0222156    -0.0262348     0.00365807
      0.000299848  -0.000422311   0.000624522     -0.00314018   -0.0166945    -0.0141309
      0.000231385  -0.000839509  -0.00036828       0.0348822     0.0449157     0.0243496
     -0.0728429     0.0171947    -0.0131377        0.0243761     0.0160184    -0.00331693
     -0.0830195    -0.0109697    -0.00976856      -0.0093758    -0.0112564    -0.0237074
     -0.0657876    -0.0240049    -0.0140109    …  -0.0108822    -0.00753059    0.00606941
```

```
 -0.0684675     0.0180922    -0.0127999        0.0179261     0.0122836    -0.00319471
  ⋮                                        ⋱
  0.00149167   -0.00594694   -0.00581047   …   0.0284113    -0.0256102     0.0150357
 -0.000804014   0.0307552     0.00201548      -0.0046152    -0.00211374    0.00347104
  0.0017       -0.0171731    -0.00763219      -0.0496163    -0.00872587    0.0174946
  0.0360833     0.0380396    -0.289237         0.00611278    0.0107237     0.0205124
 -0.0268254    -0.0383433     0.124781         0.00452719   -0.00151617    0.0219571
 -0.0220742    -0.013872      0.111505     …   0.0130679     0.0154402    -0.00138456
```

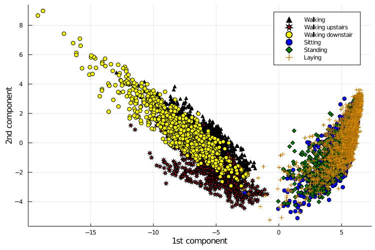
```
# Le sous-espace de dimension q qui représente 99 de la variance
θ = v[:, order[1:155]]
```

```
X_train_acp =
7352×155 Array{Float64,2}:
  5.52028  -0.290278    1.52993     …   0.0726948    0.132121      0.0355767
  5.53535  -0.0825302   1.9248          0.0210308   -0.157471      0.040921
  5.47499   0.287387    2.14464        -0.0425351   -0.00641593   -0.0151986
  5.67723   0.897031    2.01822         0.0237453   -0.0256735    -0.0585008
  5.74875   1.16295     2.13953         0.0749218    0.0384387    -0.0555716
  5.71155   1.34872     2.00397     …   0.0474181   -0.0118364    -0.0352992
  5.41052   0.574998    2.12021         0.00178789   0.0208056     0.0644773
  ⋮                                 ⋱
 -6.2104   -2.84279    -0.751203       -0.0752905    0.0106935    -0.165334
 -6.25352  -2.63677    -0.448229       -0.177855    -0.14416      -0.0860192
 -5.78232  -2.43781    -0.462731       -0.213303     0.0510888    -0.036003
 -5.85751  -3.08184    -0.671207        0.151169     0.0721317     0.109955
 -5.42109  -3.42643    -0.671243    …   0.12814      0.132078      0.120382
 -5.49797  -2.78993    -0.00572224     -0.117785    -0.0900532     0.0154308
```

```
X_train_acp = X̄ * θ
```

```
BitArray{1}: [false, false, false, false, false, false, false, false, false, fals
```

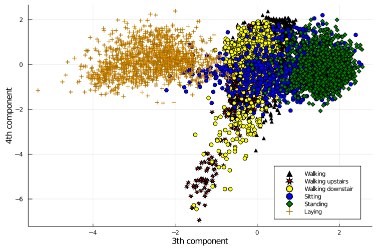
```
begin
    WALKING = (y_train .== 1)
    WALKING_UPSTAIRS = (y_train .== 2)
    WALKING_DOWNSTAIRS = (y_train .== 3)
    SITTING = (y_train .== 4)
    STANDING = (y_train .== 5)
    LAYING = (y_train .== 6)
end
```




```
# Affichage des 2 premières composantes

begin
    θ_afich = v[:, order[1:4]]
    Z = X̄ * θ
    plot(Z[WALKING, 1], Z[WALKING, 2], st=:scatter, size =(750, 500), legend=true,
label="Walking", m=:utriangle, c=:black, xlabel="1st component", ylabel="2nd
component")
    plot!(Z[WALKING_UPSTAIRS, 1], Z[WALKING_UPSTAIRS, 2], st=:scatter, legend=true,
label="Walking upstairs", m=:star7, c=:red)
    plot!(Z[WALKING_DOWNSTAIRS, 1], Z[WALKING_DOWNSTAIRS, 2], st=:scatter,
legend=true, label="Walking downstair", m=:o, c=:yellow)
    plot!(Z[SITTING, 1], Z[SITTING, 2], st=:scatter, legend=true, label="Sitting",
m=:heptagon, c=:blue)
    plot!(Z[STANDING, 1], Z[STANDING, 2], st=:scatter, legend=true,
label="Standing", m=:diamond, c=:green)
```

```
•     plot!(Z[LAYING, 1], Z[LAYING, 2], st=:scatter, legend=true, label="Laying",
  m=:a, c=:orange)
• end
```




```
• # Affichage des composantes 3 et 4
•
• begin
•     plot(Z[WALKING, 3], Z[WALKING, 4], st=:scatter, size =(750, 500), legend=true,
  label="Walking", m=:utriangle, c=:black, xlabel="3th component", ylabel="4th
  component")
•     plot!(Z[WALKING_UPSTAIRS, 3], Z[WALKING_UPSTAIRS, 4], st=:scatter, legend=true,
  label="Walking upstairs", m=:star7, c=:red)
•     plot!(Z[WALKING_DOWNSTAIRS, 3], Z[WALKING_DOWNSTAIRS, 4], st=:scatter,
  legend=true, label="Walking downstair", m=:o, c=:yellow)
•     plot!(Z[SITTING, 3], Z[SITTING, 4], st=:scatter, legend=true, label="Sitting",
  m=:heptagon, c=:blue)
•     plot!(Z[STANDING, 3], Z[STANDING, 4], st=:scatter, legend=true,
  label="Standing", m=:diamond, c=:green)
•     plot!(Z[LAYING, 3], Z[LAYING, 4], st=:scatter, legend=:bottomright,
  label="Laying", m=:a, c=:orange)
• end
```

Utilisation d'un modèle RNN

```
• md"### Utilisation d'un modèle RNN"
```

```
Y_train =
6×7352 Flux.OneHotMatrix{Array{Flux.OneHotVector,1}}:
 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  1  1  1  1  1  1  1  1  1  1  1  1
 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  0  0  0  0
 1  1  1  1  1  1  1  1  1  1  1  1  1  1     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
```

```
• # Application du OneHot encoding sur les labels d'apprentissage
• Y_train = onehotbatch(y_train, 1:6)
```

```
model = Chain(Dense(155, 64, σ), Dense(64, 6), softmax)
```

```
• # Création du modèle
• model = Chain(Dense(155, 64, σ), Dense(64,6), softmax)
```

```
θ_rnn =
Params([Float32[0.12855695 -0.0013176803 … -0.09613159 0.066836424; -0.04258428 0.048477
```

```
• # Initialisation des paramètres
• θ_rnn = Flux.params(model)
```

```
loss (generic function with 1 method)
```

```
• # Création de la fonction de perte (Entropie croisée)
• loss(x, y) = Flux.crossentropy(model(x), y)
```

```
initialLoss = 1.8093129f0
```

```
# Perte initiale
initialLoss = loss(X_train_acp', Y_train)
```

```
optimizer =  ADAM(0.001,  (0.9,  0.999),  IdDict())
```

```
# Création de l'optimizer (Adam)
optimizer = ADAM()
```

```
train (generic function with 2 methods)
```

```
# Fonction d'entrainement
function train(numEpochs=20)
    with_terminal() do
        @epochs numEpochs Flux.train!(loss, θ_rnn, [(X_train_acp', Y_train)],
optimizer; cb = () -> println(loss(X_train_acp'[:,1:100], Y_train[:,1:100])))
    end
end
```

```
[ Info: Epoch 1
1.5748048
[ Info: Epoch 2
1.5664421
[ Info: Epoch 3
1.5580957
[ Info: Epoch 4
1.5498137
[ Info: Epoch 5
1.5415965
[ Info: Epoch 6
1.5334213
[ Info: Epoch 7
1.5252517
[ Info: Epoch 8
1.5170434
[ Info: Epoch 9
1.5087519
[ Info: Epoch 10
1.5003394
[ Info: Epoch 11
```

```
train(500)
```

```
accuracy (generic function with 1 method)
```

```
accuracy(x, y) = mean(Flux.onecold(model(x)) .== y)
```

```
training_accuracy = 0.9884385201305768
```

```
training_accuracy = accuracy(X_train_acp', y_train)
```

```
Enter cell code...
```

```
2947×155 Array{Float64,2}:
  2.68674  -1.21682    0.722075    …   0.150705   -0.11832     -0.090125
  4.33126  -0.766327   1.1284          0.0511199  -0.0464086   -0.0395244
  4.98536   0.371301   1.65686        -0.0189005  -0.0728856    0.0297086
  5.09988   0.243743   1.8027         -0.108666    0.00685633   0.051577
  5.023    -0.518739   1.87108        -0.0384146  -0.0326229   -0.0581086
  4.94571  -0.522905   1.92607     …  -0.0320561  -0.0779267   -0.0146738
  5.17886   0.0320224  2.02087        -0.0409423  -0.0154981    0.0489421
  ⋮                                 ⋱
 -3.99997  -1.63732   -0.00991198      0.0123996   0.0361222   -0.097927
 -4.44716  -1.52173   -0.0984233       0.118933   -0.115774    -0.0899958
 -5.02465  -1.01597   -0.0568313       0.0361636  -0.0707252   -0.101399
 -4.5557   -1.07662   -0.153593        0.15428     0.121027    -0.207373
 -3.76462  -1.29442    0.0238255   …   0.0592178   0.0133272   -0.128423
 -3.9924   -1.04278    0.0857618      -0.0905952  -0.0769914   -0.0726717
```

```
# Application de acp sur les données de test
begin
    M_test = repeat(μ, length(y_test), 1)
    X̄_test = X_test - M_test
    X_test_acp = X̄_test * θ
end
```

test_accuracy = 0.9474041398031897

```
test_accuracy = accuracy(X_test_acp', y_test)
```