MACHINE LEARNING WITH PYTHON

TEMA 5.3: Redes neuronales

AJUSTES POLINÓMICOS

Índice

- Redes neuronales
- Redes monocapa
- Redes multicapa
- Simulación
- Generalización
- Clasificación de dígitos (MNIST)
- Detección de caras

TECNICAS DE CLASIFICACIÓN

Tipos

Técnicas estadísticas: Asocian los patrones a aquella clase que minimiza la probabilidad de cometer una clasificación errónea, o lo que es lo mismo a la clase con mayor probabilidad a posteriori $P(C_K \mid x)$.

Ajustes polinómicos: En este caso el problema consiste en ajustar tantos polinómios de grado M como clases se hayan definido a un conjunto de prototipos

Redes neuronales: Resuelven el problema de clasificación mediante la superposición de funciones no lineales inspirandose en modelos biólogicos relacionados con el cerebro de los animales.

REDES NEURONALES

Definición

Una red neuronal es un dispositivo diseñado para emular la forma en la que el cerebro realiza una tarea de interés. [Haykin, 94].

Una red neuronal es una máquina capaz de realizar un procesamiento inteligente de la información.

Una nueva forma de computación inspirada en modelos biológicos. [Hilera, 95].

Redes neuronales artificiales son redes de elementos simples (usualmente adaptivos) interconectados masivamente en paralelo y con organización jerárquica las cuales intentan interactuar con el mundo real del mismo modo que lo hace el sistema nervioso biológico. [Kohonen 88].

REDES NEURONALES

Definición

La más general desde el punto de vista funcional:

Las redes neuronales son una forma matemática de resolver el mapping impuesto por el conjunto de prototipos inspirada en los modelos biólogicos del cerebro de los animales.

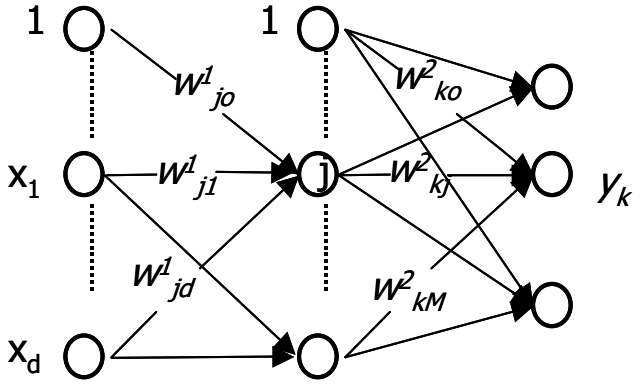
Se basan en la superposición de funciones no lineales.

REDES NEURONALES

Definición

Las redes neuronales son un conjunto de elementos de procesamiento, neuronas, interconectados y agrupados en capas de tal forma que todas las neuronas pertenecientes a la misma capa comparten una serie de propiedades.

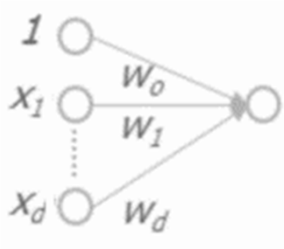
La salida de cada neurona resulta de aplicar una función de activación a la suma pesada de las entradas conectadas a dicha neurona. En el caso más general se permiten conexiones hacia delante, interacción lateral y realimentación.



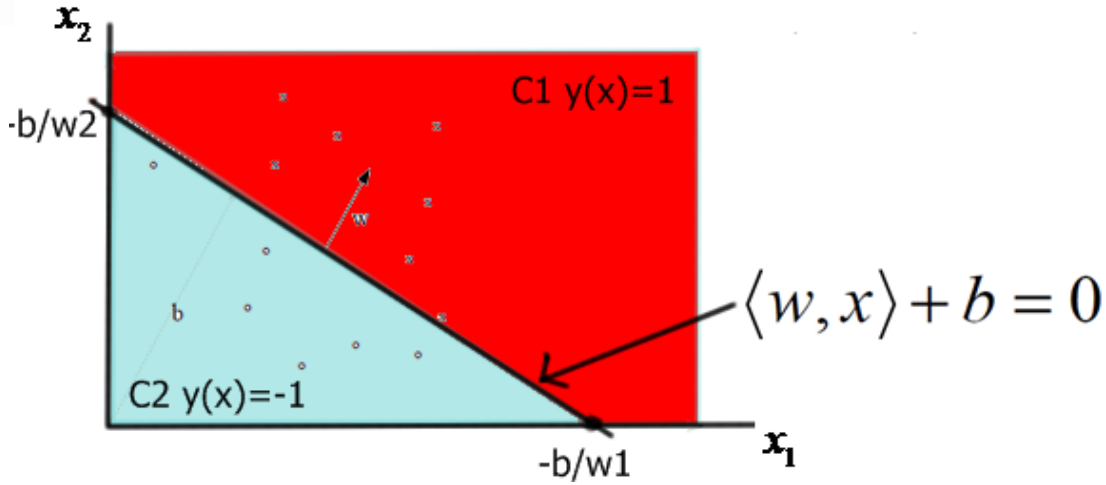
$$y_k = g^2\left(\sum_{j=0}^{M} w^2_{kj}\, g^1\left(\sum_{i=0}^{d} w^1_{ji}\, x_i\right)\right)$$

REDES MONOCAPA

Una salida



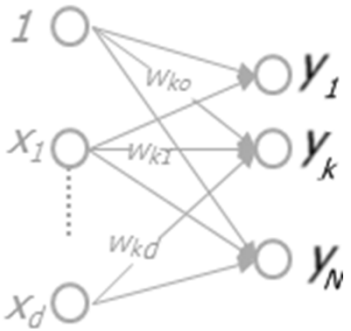
$$y(x) = sign(< w, x > +b) \text{ con } < w, x >= \sum_{i=0}^{d} w_i x_i$$



Permite separar el espacio d-dimensional en dos clases cuyos vectores se encuentran separados linealmente por un hiperplano de dimensión d-1.

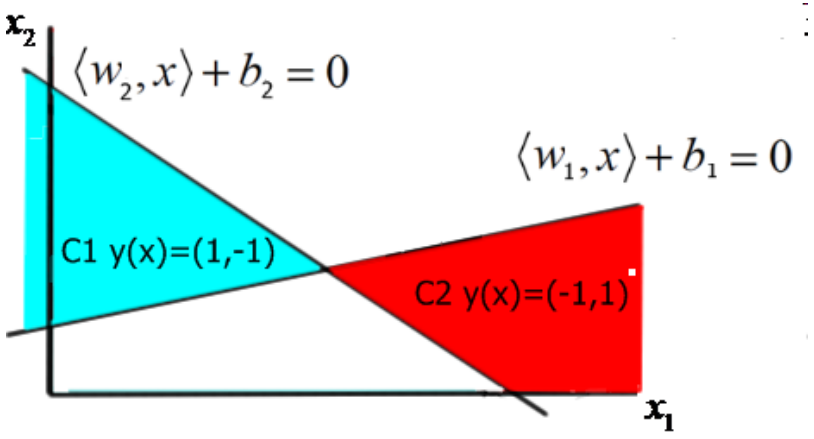
REDES MONOCAPA

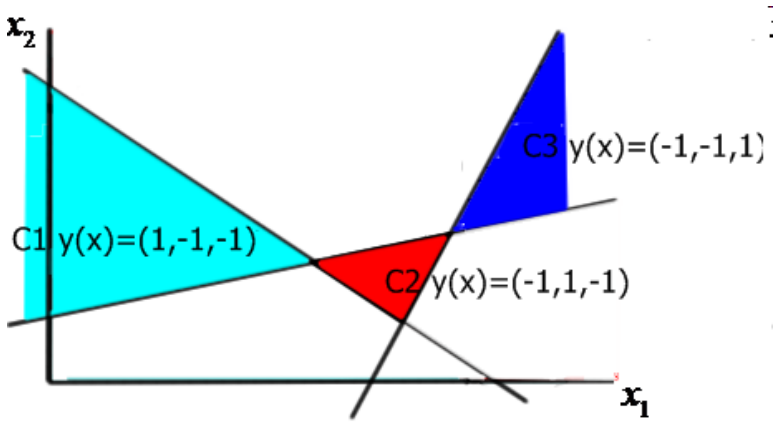
N salidas



$$y_k(x) = sign(< w_k, x > + b_k)$$

Permite separar el espacio d-dimensional en N clases cuyos vectores se encuentran separados linealmente por N hiperplanos de dimensión d-1. Salida en formato one-hot

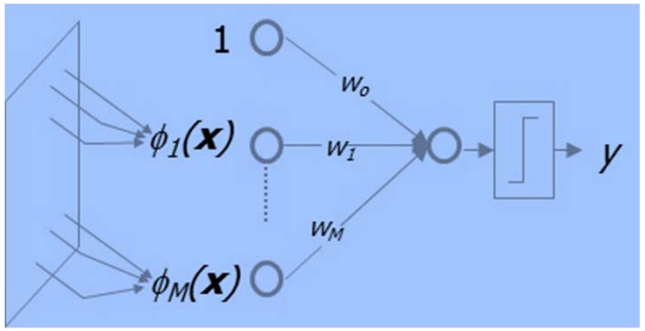




REDES MONOCAPA

El Perceptron

Propuesto por Rosenblatt en 1958. Lo aplicó a la clasificación de imágenes binarias correspondientes a formas simples (caracteres).



$$y = sign\left(< w, \phi(x) > + b\right)$$

Las funciones base realizan combinaciones lineales del vector de entrada con pesos fijos (0 y 1) y pueden utilizar funciones tipo umbral.

Puede realizar discriminaciones no lineales siempre y cuando las funciones base elegidas transformen el conjunto de patrones de entrada en un nuevo conjunto linealmente separable.

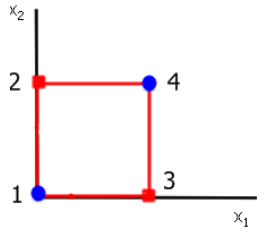
REDES MONOCAPA

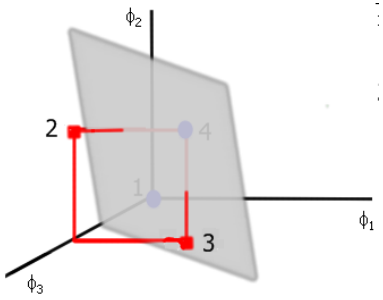
El Perceptron

En 1969 Minsky y Papert publicaron el libro Perceptrons en el que se desarrollaban desde un punto de vista matemático las limitaciones del perceptron.

Suelen resumirse en la incapacidad de un perceptron de separar regiones que no son linealmente separables. En concreto el problema de la OR-Exclusiva.

Lo cual no es del todo cierto ya que cogiendo de forma adecuada las funciones base el perceptron puede resolver problemas de separabilidad no lineal.





Clase	x_1	x_2	$\phi_1=x_1$	$\phi_2=x_2$	$\phi_3=U(x_1+x_2)$
C_1	0	0	0	0	0
C_2	0	1	0	1	1
C_2	1	0	1	0	1
C_1	1	1	1	1	1

REDES MONOCAPA

El Perceptron: Aprendizaje

Rosenblatt dedujo un algoritmo para el cálculo de la solución basada en la minimización de una función error a través de un proceso iterativo conocida como el criterio del perceptron.

Algoritmo de aprendizaje:

- 1.- Inicialización aleatoria de pesos y umbrales.
- 2.- Presentación de una par entrada salida (xn,tn).
- 3.- Calculo de la salida actual de la neurona.
- 4.- Si la clasificación del patron xn es incorrecta (y(xn)tn <=0) adaptación de los pesos.

$$w(t+1) = w(t) + \alpha \phi_n t^n$$
$$w(t+1) = w(t) + \alpha x^n t^n \quad con \quad t^n = \begin{cases} +1 & si \quad x^n \in C_1 \\ -1 & si \quad x^n \in C_2 \end{cases}$$

- 5.- Volver al paso 2 con un nuevo patrón del conjunto de entrenamiento
- 6.- Repetir el algoritmo si hay algún patrón mal clasificado.

El Perceptron: Aprendizaje

Teorema de convergencia del perceptrón: si los patrones usados para entrenar el perceptron están agrupados en clases linealmente separables el algoritmo de aprendizaje converge hacia unos valores de los pesos que posicionan un hiperplano entre las clases anteriormente mencionadas.

El Perceptron: Ejemplo

Entrenamiento del perceptron para aprender el mapping impuesto por los prototipos:

Prototipos	x_1	x_2	t
p1 o x^1	2	1	1
p2 o x^2	0	-1	1
p3 o x^3	-2	1	-1
p4 o x^4	0	2	-1

Los pesos y bias se han iniciado aleatoriamente con los valores:

$$w(0)=(-0.7,0.2) \text{ y } b(0)=0.5.$$

El Perceptron: Ejemplo

Iteración 1: Presentamos a la red el patrón de entrenamiento $p1$.

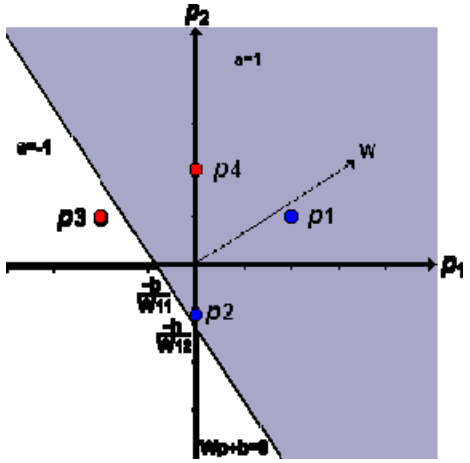
$$y(x^1) = sign(< w(0), x^1 > + b(0)) = sign\left(\begin{pmatrix} -0.7 & 0.2 \end{pmatrix} \begin{pmatrix} 2 \\ 1 \end{pmatrix} + 0.5 \right) = sign(-0.7) = -1$$

Como está mal clasificado (podría haberse deducido gráficamente en la figura 1) produce modificación de los pesos.

$$w(1) = w(0) + \alpha \cdot x^1 \cdot t^1 = \begin{pmatrix} -0.7 & 0.2 \end{pmatrix} + 2 \begin{pmatrix} 2 & 1 \end{pmatrix} 1 = \begin{pmatrix} 3.3 & 2.2 \end{pmatrix}$$

$$b(1) = b(0) + \alpha \cdot 1 \cdot t^1 = 0.5 + 2 \cdot 1 \cdot 1 = 2.5$$

Lo que transforma la superficie de decisión:



$$\left(-b/w_1 \quad -b/w_2 \right) = \left(-0.75 \quad -1.13 \right)$$

Como se observa el patrón de entrenamiento *p1* ha sido clasificado correctamente, y casualmente los patrones *p2* y *p3* fueron correctamente ubicados, pues aun no han sido presentados a la red.

El Perceptron: Ejemplo

Iteración 2: Se presenta *p2* a la red, es clasificado correctamente no modifica los pesos

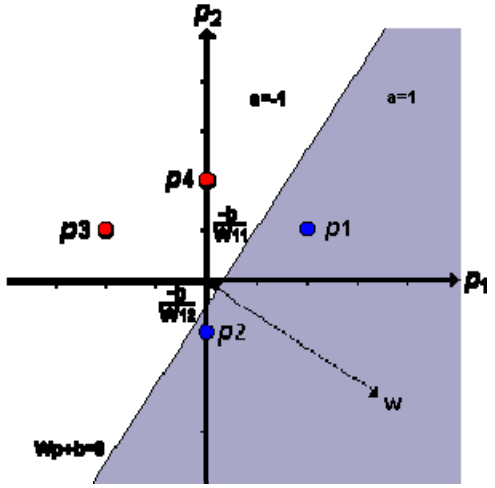
Iteración 3: Se presenta *p2* a la red, es clasificado correctamente no modifica los pesos

Iteración 4: Se presenta a la red *p4*, y es clasificado incorrectamente por lo que produce modificación de los pesos:

$$w(2) = w(1) + \alpha p_4 t_4 = \begin{pmatrix} 3.3 & 2.2 \end{pmatrix} + 2\begin{pmatrix} 0 & 2 \end{pmatrix}(-1) = \begin{pmatrix} 3.3 & -1.8 \end{pmatrix}$$
$$b(2) = b(1) + \alpha \cdot 1 \cdot t_4 = 2.5 + 2 \cdot 1 \cdot (-1) = 0.5$$

Lo que transforma la superficie de decisión:

$$\begin{pmatrix} -b/w_1 & -b/w_2 \end{pmatrix} = \begin{pmatrix} -0.15 & 0.27 \end{pmatrix}$$



Todos los patrones son clasificados correctamente.

REDES MONOCAPA

Regla delta

Si la función de activación de las neuronas son derivables, los pesos **w** se calculan mediante algoritmos que minimizan el error cuadrático (**loss**) calculado sobre un conjunto de patrones de entrada $\mathbf{x}^n$ (prototipos) y sus correspondientes salidas deseadas $\mathbf{t}^n$ (aprendizaje supervisado).

$$E(\boldsymbol{w}) = \frac{1}{2}\sum_{n=1}^{N}\sum_{k=1}^{c}\left(y_k(\boldsymbol{x}^n;\boldsymbol{w}) - t_k^n\right)^2$$

Cuando la función de activación es no lineal no es posible encontrar la solución para los pesos de forma analítica (ésto si era posible en los ajustes polinómicos).

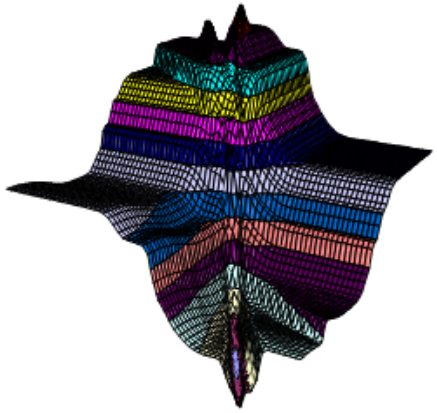
Se pueden aplicar técnicas de gradiente descendiente para encontrar el mínimo de la función error (**optimizer**).

REDES MONOCAPA

Regla delta

Se parte de un estado inicial (pesos aleatorios) y se aplican pequeños cambios en la dirección en la que el error E decrezca más rápidamente.

$$w_{kj}^{t+1} = w_{kj}^{t} - \eta \left. \frac{\partial E}{\partial w_{kj}} \right|_{\boldsymbol{w}^T}$$



A η se le denomina velocidad de aprendizaje y la elección de su valor es crítica.

La actualización de los pesos puede hacerse, *pattern by pattern*, *epoch by epoch o mini-batch,* en función de si se coge uno, todos o un subconjunto de los prototipos para calcular el error.

REDES MONOCAPA

Regla delta

Presentamos un patrón de entrada xn a la red

Calculamos el error cometido

$$E(\boldsymbol{w}) = \frac{1}{2}\sum_{k=1}^{c}\left(y_k(\boldsymbol{x}^n;\boldsymbol{w}) - t_k^n\right)^2 = \frac{1}{2}\sum_{k=1}^{c}\left(g\left(\sum_{j=0}^{d} w_{kj}x_j^n\right) - t_k^n\right)^2$$

Aplicamos el método del gradiente descendiente. $$w_{kj}^{t+1} = w_{kj}^t - \eta\left.\frac{\partial E}{\partial w_{kj}}\right|_{\boldsymbol{w}^T}$$

$$\frac{\partial E^n}{\partial w_{kj}} = g'\left(\sum_{j=0}^{d} w_{kj}x_j^n\right)\left(y_k(x^n) - t_k^n\right)x_j^n = \delta_k^n x_j^n \qquad \delta_k^n = g'\left(\sum_{j=0}^{d} w_{kj}x_j^n\right)\left(y_k(x^n) - t_k^n\right)$$

A esta regla se le denomina LMS (least mean squares), regla adalina, regla de Widrow-Hoff, y regla delta. Fue propuesta por Widrow and Hoff en 1960.

REDES MONOCAPA

Regla delta

Para funciones de activación identidad:

$$\delta_k^n = \left(y_k(x^n) - t_k^n\right)$$

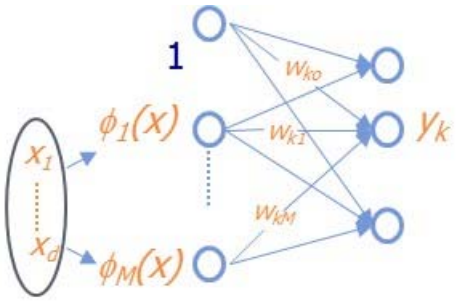
Para funciones de activación sigmoidales:

$$g(x) = \frac{1}{1+e^{-x}}$$

$$\delta_k^n = y_k(x^n)(1-y_k(x^n))\left(y_k(x^n) - t_k^n\right)$$

REDES MONOCAPA

Discriminante generalizado

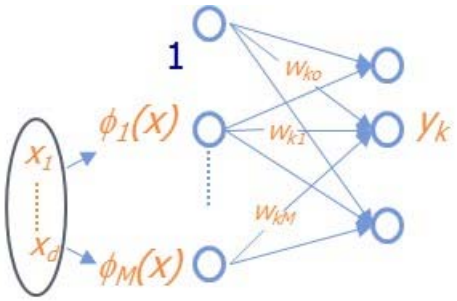


$$y_k = \sum_{j=1}^{M} w_{kj}\phi_j(x) + w_{ko}$$

Cogiendo de "forma adecuada las funciones base" el discriminante generalizado puede aproximar cualquier mapping entre dos espacios con una precisión arbitraria.

REDES MONOCAPA

Discriminante generalizado



$$y_k = \sum_{j=1}^{M} w_{kj}\phi_j(x) + w_{ko}$$

La elección de determinadas funciones con parámetros ajustables en la fase de entrenamiento (gaussianas) como funciones base da lugar a un tipo de redes neuronales denominadas RBF (radial basis functions).

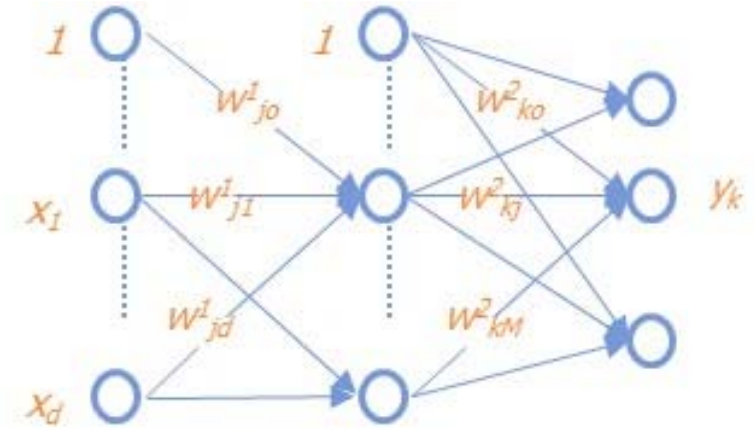
Las redes neuronales multicapa con conexionado hacia delante (feed-forward) o perceptrones multicapa se pueden considerar un caso particular de este tipo de discriminante.

REDES MULTICAPA

Definición

Las funciones base constituyen una nueva capa de elementos de procesamiento denominada capa oculta.

Puede haber más de una capa oculta



$$y_k = g^2\left(\sum_{j=0}^{M} w^2_{kj}\, g^1\left(\sum_{i=0}^{d} w^1_{ji} x_i\right)\right)$$

REDES MULTICAPA

Propiedades

Si las funciones de activación de las unidades ocultas son todas lineales entonces se puede encontrar una red equivalente pero sin unidades ocultas (sólo resuelve problemas linealmente separables).

Para producir discriminación no lineal las funciones de activación de las unidades ocultas (g^1) deben ser no lineales. Suelen utilizarse funciones de activación tipo sigmoide.

Una red feed-forward con al menos una capa oculta y función de activación sigmoidal puede aproximar cualquier mapping continuo entre dos espacios de dimensión finita suponiendo que el número de unidades ocultas es suficientemente grande.

REDES MULTICAPA

Regla delta generalizada

Cuando se quiere aplicar el gradiente descendiente a las redes feed forward multicapa aparece el problema conocido como "credit assignment". No se conoce cual es la salida optima y por lo tanto el error cometido para las unidades ocultas.

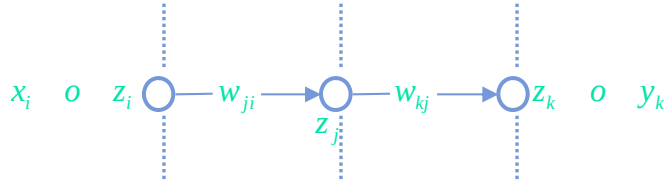
El algoritmo Back-propagation se presenta como una herramienta muy eficaz (en el sentido computacional) para el cálculo de las derivadas de la función error con respecto a los pesos en las unidades ocultas.

Este algoritmo fue presentado por Rumelhart, Hinton y Williams en torno a 1986.

REDES MULTICAPA

Regla delta generalizada

Consideremos un perceptron multicapa de una capa oculta en el que el prototipo x^n se presenta en la entrada de la red.



Primero se calculan las salidas de las neuronas de la capa oculta:

$$z_j = g^1(a_j) \quad con \quad a_j = \sum_i w_{ji} z_i = \sum_i w_{ji} x_i$$

Después se calculan las salidas de las neuronas de la capa de salida:

$$y_k = z_k = g^2(a_k) \quad con \quad a_k = \sum_j w_{kj} z_j$$

REDES MULTICAPA

Regla delta generalizada

$$E^n(\boldsymbol{w}) = \frac{1}{2}\sum_{k=1}^{c}\left(y_k(\boldsymbol{x}^n;\boldsymbol{w}) - t_k^n\right)^2 = \frac{1}{2}\sum_{k=1}^{c}\left(g^2(a_k) - t_k^n\right)^2 \qquad a_k = \sum_j w_{kj} z_j = \sum_j w_{kj} g^1(a_j)$$

Adaptación de los pesos w_{kj}:

$$\frac{\partial E^n}{\partial w_{kj}} = \frac{\partial E^n}{\partial a_k}\frac{\partial a_k}{\partial w_{kj}} = \delta_k^n z_j^n \qquad \delta_k^n = \frac{\partial E^n}{\partial a_k} = g^{2\prime}(a_k)\left(y_k(x^n) - t_k^n\right)$$

Adaptación de los pesos w_{ji}:

$$\frac{\partial E^n}{\partial w_{ji}} = \frac{\partial E^n}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}} = \delta_j x_i \qquad \delta_j = \frac{\partial E^n}{\partial a_j} = \sum_k \frac{\partial E^n}{\partial a_k}\frac{\partial a_k}{\partial a_j} = g^{1\prime}(a_j)\sum_k \delta_k w_{kj}$$

Los términos de error δ se van calculando desde las unidades de salida hacia las unidades de entrada (back-propagation).

Regla delta generalizada

Para funciones de activación de las unidades ocultas sigmoidales y las de salida lineales.

$$\delta_k = g^{2}{}'(a_k)\frac{\partial E^n}{\partial y_k} = (y_k - t_k) \qquad \delta_j = \sum_k \frac{\partial E^n}{\partial a_k}\frac{\partial a_k}{\partial a_j} = z_j(1-z_j)\sum_{k=1}^{c}\delta_k w_{kj}$$

Ajuste de los pesos:

$$\Delta w_{kj} = -\eta\delta_k z_j = -\eta(y_k - t_k)z_j$$

$$\Delta w_{ji} = -\eta\delta_j x_i = -\eta x_i z_j(1-z_j)\sum_{k=1}^{c}\delta_k w_{kj}$$

A este algoritmo de adaptación de los pesos se le conoce como regla delta generalizada

REDES MULTICAPA

Metodología de aprendizaje

1.- Inicialización aleatoria de los pesos

2.- Presentación de un patrón de entrada a la red junto con su salida deseada.

3.- Cálculo de los términos a_j y z_j para todas las neuronas de red.

La información circula hacia delante

4.- Se calculan los términos de error δ_j para todas las neuronas empezando por la capa de salida y finalizando en la primera capa de procesamiento.

El error se propaga hacia atrás

5.- Actualización de los pesos

6.- El proceso se repite hasta que el termino de error ε resulta aceptablemente pequeño u otro criterio de parada.

$$E = \sum_{n=1}^{N} \sum_{k=1}^{c} \delta_k^n$$

REDES MULTICAPA

Variantes de back-propagation

Actualización de los pesos:

- Pattern by pattern o SGD (stochastic gradient descent) suele converger mucho más rápido en cuanto a iteraciones pero mas lento de computar debido a que hay más actualizaciones de los pesos.
- Epoch by epoch o (batch gradient descent) más rápido más eficiencia computacional (aprovecha mejor las operaciones con tensores) pero con mayor tendencia a caer en mínimos locales.
- Mini-batch gradient descent. El tamaño del mini-batch es importante. Es un compromiso entre los anteriores.

Reordenación aleatoria de los prototipos en cada pasada (shuffle) es casi obligatorio.

Inicalización de los pesos: Transfer of learning.

REDES MULTICAPA

Variantes de back-propagation

Velocidad de aprendizaje fija o adaptativa

Inclusión de un termino momento

Diferentes funciones coste: Hemos utilizado el error cuadrático medio

Diferentes algoritmos de optimización: Hemos utilizado el gradiente descendiente.

Diferentes funciones de activación tanto para las capas intermedias como para la capa de salida.

REDES MULTICAPA

Variantes de back-propagation

Muchas de las elecciones van interrelacionadas:

Problem type	Last-layer activation	Loss function
Binary classification	`sigmoid`	`binary_crossentropy`
Multiclass, single-label classification	`softmax`	`categorical_crossentropy`
Multiclass, multilabel classification	`sigmoid`	`binary_crossentropy`
Regression to arbitrary values	None	`mse`
Regression to values between 0 and 1	`sigmoid`	`mse` or `binary_crossentropy`

SIMULACIÓN

sklearn.neural_network

Paquete para la simulación de redes neuronales.

No soporta GPU

Existen otros paquetes más flexibles para modelos basados en Deep-learning como TensorFlow y Keras que se ejecuta sobre TensorFlow.

Para la simulación de perceptrones multicapa entrenados con backpropagation se utiliza la clase MLPClassifier.

Muchos parámetros ajustables, número de neuronas, capas, funciones de activación, velocidad de aprendizaje, numero de iteraciones, etc.

https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html

SIMULACIÓN

sklearn.neural_network

Muchos de ellos vienen definidos por defecto.

```
sklearn.neural_network.MLPClassifier(hidden_layer_sizes=(100,                ),
activation='relu',     solver='adam',     alpha=0.0001,     batch_size='auto',
learning_rate='constant', learning_rate_init=0.001, power_t=0.5, max_iter=200,
shuffle=True, random_state=None, tol=0.0001, verbose=False, warm_start=False,
momentum=0.9,          nesterovs_momentum=True,          early_stopping=False,
validation_fraction=0.1,     beta_1=0.9,     beta_2=0.999,     epsilon=1e-08,
n_iter_no_change=10)
```

SIMULACIÓN

sklearn.neural_network

hidden_layer_sizes : tuple, length = n_layers - 2, default (100,) The ith element represents the number of neurons in the ith hidden layer.
activation : {'identity', 'logistic', 'tanh', 'relu'}, default 'relu'. Activation function for the hidden layer.

- 'identity', no-op activation, useful to implement linear bottleneck, returns f(x) = x
- 'logistic', the logistic sigmoid function, returns f(x) = 1 / (1 + exp(-x)).
- 'tanh', the hyperbolic tan function, returns f(x) = tanh(x).
- 'relu', the rectified linear unit function, returns f(x) = max(0, x)

solver : {'lbfgs', 'sgd', 'adam'}, default 'adam'. The solver for weight optimization.

- 'lbfgs' is an optimizer in the family of quasi-Newton methods.
- 'sgd' refers to stochastic gradient descent.
- 'adam' refers to a stochastic gradient-based optimizer proposed by Kingma, Diederik, and Jimmy Ba

Note: The default solver 'adam' works pretty well on relatively large datasets (with thousands of training samples or more) in terms of both training time and validation score. For small datasets, however, 'lbfgs' can converge faster and perform better.

alpha : float, optional, default 0.0001. L2 penalty (regularization term) parameter.
batch_size : int, optional, default 'auto''. Size of minibatches for stochastic optimizers. If the solver is 'lbfgs', the classifier will not use minibatch. When set to "auto", batch_size=min(200, n_samples)

SIMULACIÓN

sklearn.neural_network

learning_rate : {'constant', 'invscaling', 'adaptive'}, default 'constant'

Learning rate schedule for weight updates.

- 'constant' is a constant learning rate given by 'learning_rate_init'.
- 'invscaling' gradually decreases the learning rate at each time step 't' using an inverse scaling exponent of 'power_t'. effective_learning_rate = learning_rate_init / pow(t, power_t)
- 'adaptive' keeps the learning rate constant to 'learning_rate_init' as long as training loss keeps decreasing. Each time two consecutive epochs fail to decrease training loss by at least tol, or fail to increase validation score by at least tol if 'early_stopping' is on, the current learning rate is divided by 5.

Only used when solver='sgd'.

learning_rate_init : double, optional, default 0.001

The initial learning rate used. It controls the step-size in updating the weights. Only used when solver='sgd' or 'adam'.

power_t : double, optional, default 0.5

The exponent for inverse scaling learning rate. It is used in updating effective learning rate when the learning_rate is set to 'invscaling'. Only used when solver='sgd'.

max_iter : int, optional, default 200

Maximum number of iterations. The solver iterates until convergence (determined by 'tol') or this number of iterations. For stochastic solvers ('sgd', 'adam'), note that this determines the number of epochs (how many times each data point will be used), not the number of gradient steps.

shuffle : bool, optional, default True

Whether to shuffle samples in each iteration. Only used when solver='sgd' or 'adam'.

SIMULACIÓN

sklearn.neural_network

random_state : int, RandomState instance or None, optional, default None
If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by np.random.

tol : float, optional, default 1e-4
Tolerance for the optimization. When the loss or score is not improving by at least tol for n_iter_no_change consecutive iterations, unless learning_rate is set to 'adaptive', convergence is considered to be reached and training stops.

verbose : bool, optional, default False
Whether to print progress messages to stdout.

warm_start : bool, optional, default False
When set to True, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution.

momentum : float, default 0.9
Momentum for gradient descent update. Should be between 0 and 1. Only used when solver='sgd'.

nesterovs_momentum : boolean, default True
Whether to use Nesterov's momentum. Only used when solver='sgd' and momentum > 0.

SIMULACIÓN

sklearn.neural_network

early_stopping : bool, default False

Whether to use early stopping to terminate training when validation score is not improving. If set to true, it will automatically set aside 10% of training data as validation and terminate training when validation score is not improving by at least tol for n_iter_no_change consecutive epochs. The split is stratified, except in a multilabel setting. Only effective when solver='sgd' or 'adam'

validation_fraction : float, optional, default 0.1

The proportion of training data to set aside as validation set for early stopping. Must be between 0 and 1. Only used if early_stopping is True

beta_1 : float, optional, default 0.9

Exponential decay rate for estimates of first moment vector in adam, should be in [0, 1). Only used when solver='adam'

beta_2 : float, optional, default 0.999

Exponential decay rate for estimates of second moment vector in adam, should be in [0, 1). Only used when solver='adam'

epsilon : float, optional, default 1e-8

Value for numerical stability in adam. Only used when solver='adam'

n_iter_no_change : int, optional, default 10

Maximum number of epochs to not meet tol improvement. Only effective when solver='sgd' or 'adam'

SIMULACIÓN

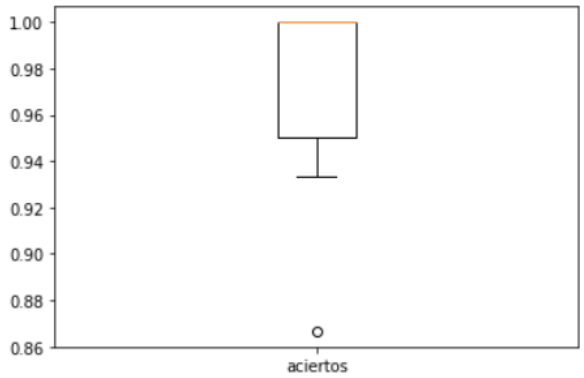
Ejemplo: nn.py

Utiliza una red neuronal de una capa oculta con 10 neuronas para resolver el problema de clasificación en el conjunto de datos iris.

```
from sklearn import datasets
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import cross_val_score, train_test_split
import matplotlib.pyplot as plt

iris = datasets.load_iris()    #carga el conjunto de datos iris.
xtrain,xtest,ytrain,ytest = train_test_split(iris.data,iris.target)
#Utilizamos sgd aunque lbfgs converge antes
#Verbose = True muestra el progreso de la red en el prompt de jupyter
model = MLPClassifier(activation='relu',hidden_layer_sizes=(10),
            learning_rate_init=0.001,max_iter=10000, n_iter_no_change=50,
            verbose=False,shuffle=True, solver='sgd', tol=0.0001)
aciertos = cross_val_score(model,iris.data,iris.target,cv=10)
print(aciertos)
plt.boxplot(aciertos,labels=['aciertos'])
plt.show()
```

```
[1.         1.         1.         0.93333333 0.86666667 1.
 0.93333333 1.         1.         1.        ]
```



El aprendizaje finaliza cuando se supera el número máximo de iteraciones o cuando el coste no mejora una cantidad mayor a tol durante n_iter_no_change iteraciones.

En SGD el número de iteraciones es el número de veces que cada patrón ha sido utilizado. En cada epoch se usa un subconjunto del conjunto de entrenamiento de dimensión batch_size ordenados aleatoriamente.

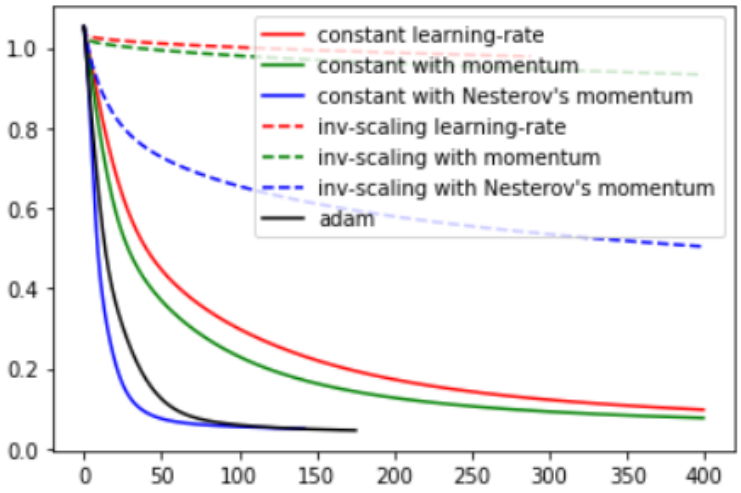
La función de activación de la capa de salida se coge automáticamente con el solver elegido.

SIMULACIÓN

Ejemplo: nn.py

La elección del solver y sus parámetros son fundamentales para acelerar la convergencia.

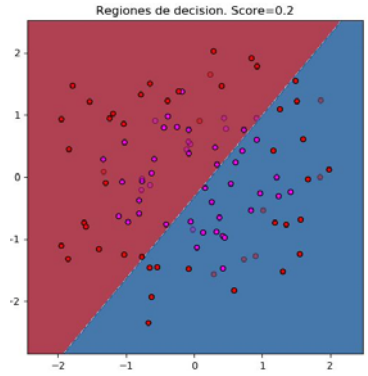
```
[{'solver': 'sgd', 'learning_rate': 'constant', 'momentum': 0,
  'learning_rate_init': 0.2},
 {'solver': 'sgd', 'learning_rate': 'constant', 'momentum': 0.3,
  'nesterovs_momentum': False, 'learning_rate_init': 0.2},
 {'solver': 'sgd', 'learning_rate': 'constant', 'momentum': .9,
  'nesterovs_momentum': True, 'learning_rate_init': 0.2},
 {'solver': 'sgd', 'learning_rate': 'invscaling', 'momentum': 0,
  'learning_rate_init': 0.2},
 {'solver': 'sgd', 'learning_rate': 'invscaling', 'momentum': .3,
  'nesterovs_momentum': True, 'learning_rate_init': 0.2},
 {'solver': 'sgd', 'learning_rate': 'invscaling', 'momentum': .9,
  'nesterovs_momentum': False, 'learning_rate_init': 0.2},
 {'solver': 'adam', 'learning_rate_init': 0.01}]
```



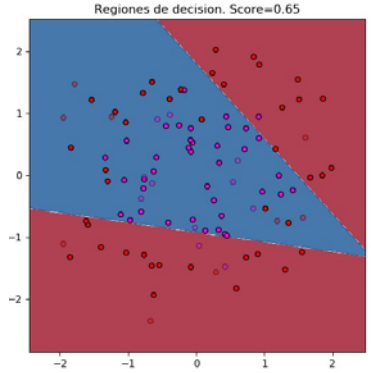
SIMULACIÓN

Ejemplo: regionesnn.py

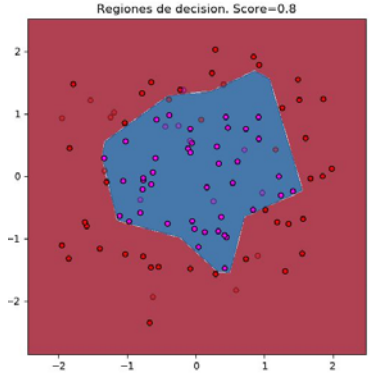
Clasificación de datos generados sintéticamente con make_circle con distintas topologías de red.



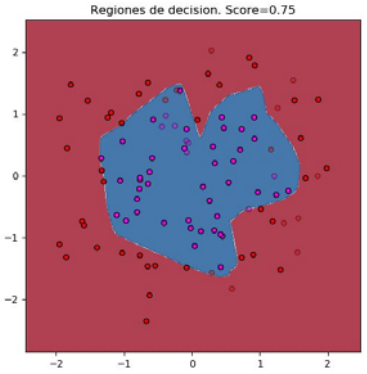
1 capa oculta de 10 neuronas con función de activación identidad



1 capa oculta de 3 neuronas con función de activación relu

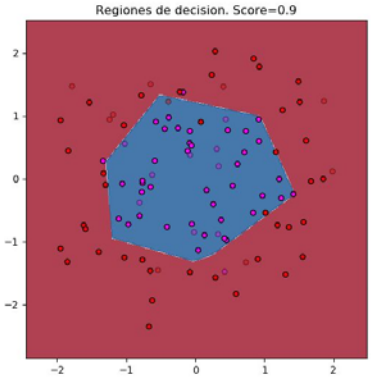


1 capa oculta de 10 neuronas con función de activación relu



1 capa oculta de 100 neuronas con función de activación relu

2 capa oculta de 10 y 5 neuronas con función de activación relu



GENERALIZACIÓN

Elección de hyperparámetros

De todos los modelos generados cual es el mejor en base a los resultados de clasificación?.

La elección de los hyperparámetros que definen la complejidad del modelo se suele hacer haciendo barridos y seleccionando los que ofrecen los mejores de resultados de clasificación.

Para ello hay que elegir una figura de mérito (score) que nos ayude a comparar dos modelos diferentes. Suele utilizarse la función loss o la capacidad de hacer predicciones correctas por el modelo (accuracy).

El paquete scikit-learn ofrece herramientas para realizar esta búsqueda.

CLASIFICACIÓN DE DÍGITOS (MNIST)

Ejemplo: nistnn.py

Utilizamos el conjunto de datos NIST del módulo: keras.datasets.

60000 dígitos manuscritos digitalizados a 28x28 para el conjunto de train y 10000 para el conjunto de test categorizados en 10 clases.

Como vector de características utilizaremos directamente el valor de los 28x28 pixels en un vector de dimensión 784.

Como figura de mérito utilizaremos el coste sobre el conjunto de entrenamiento.

Haremos un barrido para un numero de neuronas en la capa oculta de 50 a 500 en saltos de 50 neuronas.

Finalmente estimaremos los resultados de clasificación sobre el conjunto de test.

CLASIFICACIÓN DE DÍGITOS (MNIST)

Ejemplo: nistnn.py

```
from sklearn import datasets
from sklearn.neural_network import MLPClassifier
import numpy as np
import pickle
from keras.datasets import mnist

(xtrain,ytrain),(xtest,ytest)=mnist.load_data("/home/miguel/images/nist1")
#(60000,28,28) y (10000,28,28) en escala de grises con labels de 0 a 9
xtrain=xtrain.reshape(60000,784)/255
xtest=xtest.reshape(10000,784)/255
#Exploramos el valor del numero de neuronas que mejor resuelve el problema
neus=np.arange(50,501,50)
errormin=1000.
print(neus)
for numneu in neus:
        print("Simulando con: " + str(numneu) + " neuronas")
        model = MLPClassifier(activation='relu',batch_size=500,
                        early_stopping=False, hidden_layer_sizes=(numneu),
                        learning_rate='constant', learning_rate_init=0.001,
                         max_iter=10000,n_iter_no_change=10,
                         shuffle=True, solver='sgd', tol=0.0001,verbose=False)
        model.fit(xtrain,ytrain)
        print("Iteraciones: "+str(model.n_iter_))
        print("Loss: "+str(model.loss_))
        if model.loss_ < errormin:
                errormin=model.loss_
                pickle.dump(model,open("nistnn.net",'wb'))
                print("Bes net: "+str(model.loss_))
#Estimamos con la mejor red la precision sobre el conjunto de test
model = pickle.load(open("nistnn.net",'rb'))
acc = model.score(xtest,ytest)
print(acc)
```

CLASIFICACIÓN DE DÍGITOS (MNIST)

Ejemplo: nistnn.py

```
Using TensorFlow backend.
[ 50 100 150 200 250 300 350 400 450 500]
Simulando con: 50 neuronas
Iteraciones: 661
Loss: 0.07487819270993298
Bes net: 0.07487819270993298
Simulando con: 100 neuronas
Iteraciones: 667
Loss: 0.057366300634239434
Bes net: 0.057366300634239434
Simulando con: 150 neuronas
Iteraciones: 630
Loss: 0.054870139948296134
Bes net: 0.054870139948296134
Simulando con: 200 neuronas
Iteraciones: 596
Loss: 0.05568518461113431
Simulando con: 250 neuronas
Iteraciones: 621
Loss: 0.051560087699090686
Bes net: 0.051560087699090686
Simulando con: 300 neuronas
Iteraciones: 567
Loss: 0.05550992334996613
Simulando con: 350 neuronas
Iteraciones: 618
Loss: 0.05071064376368945
Bes net: 0.05071064376368945
Simulando con: 400 neuronas
Iteraciones: 598
Loss: 0.05036257447376274
Bes net: 0.05036257447376274
Simulando con: 450 neuronas
Iteraciones: 613
Loss: 0.04905933631045597
Bes net: 0.04905933631045597
Simulando con: 500 neuronas
Iteraciones: 598
Loss: 0.04880460420230898
Bes net: 0.04880460420230898
0.9772
```

Los mejores resultados se obtienen para una red de 500 neuronas en la capa oculta lo que nos debe motivar a hacer una nueva búsqueda aumentando el número máximo de neuronas.

Para cada configuración el proceso también debería repetirse varias veces debido a que el inicio de los pesos es aleatorio y se puede llegar a mínimos locales distintos.

La dimensión del vector de características (784) y el número de patrones (55000) a aprender es tan elevado que se requiere una red con muchos parámetros ajustables (muchas neuronas en la capa oculta).

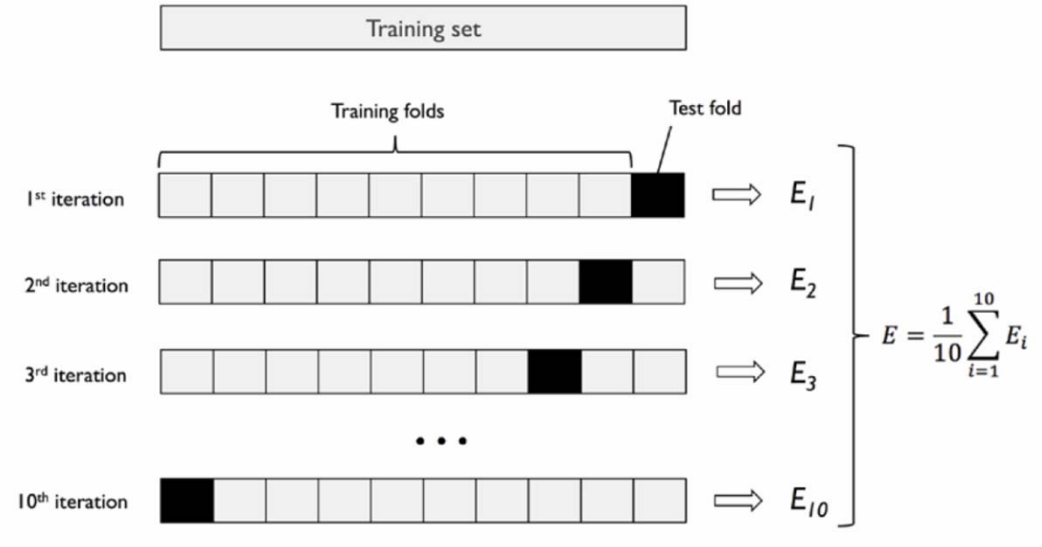
La tasa de aciertos sobre el conjunto de test es de 97.72% y el número medio de iteraciones en cada simulación es de 600.

CLASIFICACIÓN DE DÍGITOS (MNIST)

Ejemplo: nistnngrid.py

El paquete scikit-learn nos ofrece utilidades para facilitarnos la búsqueda de los parámetros que optimizan los resultados de clasificación.

Además para que los resultados sean menos dependientes de la partición realizada sobre el conjunto de entrenamiento se da la opción de realizar un k-cross-validation.



CLASIFICACIÓN DE DÍGITOS (MNIST)

Ejemplo: nistnngrid.py

```
from sklearn import datasets
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import GridSearchCV
import numpy as np
import pickle
from keras.datasets import mnist

import tensorflow as tf
config=tf.ConfigProto()
config.gpu_options.allow_growth=True

(xtrain,ytrain),(xtest,ytest)=mnist.load_data("/home/miguel/images/nist1")
xtrain=xtrain.reshape(60000,784)/255
xtest=xtest.reshape(10000,784)/255
#Exploramos el valor del numero de neuronas que mejor resuelve el problema
model = MLPClassifier(batch_size=500)
#Para definir la busqueda definimos un alista de diccionarios.
param_grid = [{'activation':['relu','logistic'],'hidden_layer_sizes':[(200,),(500,),(1000,)]}]
gs=GridSearchCV(estimator=model,param_grid=param_grid,scoring='accuracy',cv=5,refit=True,n_jobs=-1)
gs=gs.fit(xtrain,ytrain)
print(gs.best_score_)
print(gs.best_params_)
#Estimamos con la mejor red la precision sobre el conjunto de test
clf=gs.best_estimator_
print('Precision sobre el conjunto de test: %.3f' % clf.score(xtest,ytest))
```

```
0.9802833333333333
{'activation': 'relu', 'hidden_layer_sizes': (1000,)}
Precision sobre el conjunto de test: 0.983
```

GENERALIZACIÓN

Optimización vs generalización

Hasta ahora sólo hemos hablado de optimización, ajuste del modelo sobre el conjunto de entrenamiento en el que el criterio de parada ha sido el número máximo de iteraciones.

Esta forma de actuar conlleva a que la red puede sobreajustarse (overfitting) al conjunto de entrenamiento perdiendo capacidad de generalización ante un conjunto de datos nuevo.

La red, una vez entrenada, debe tener buena capacidad de generalización, es decir, debe responder de forma adecuada ante patrones de entrada distintos del conjunto de aprendizaje.

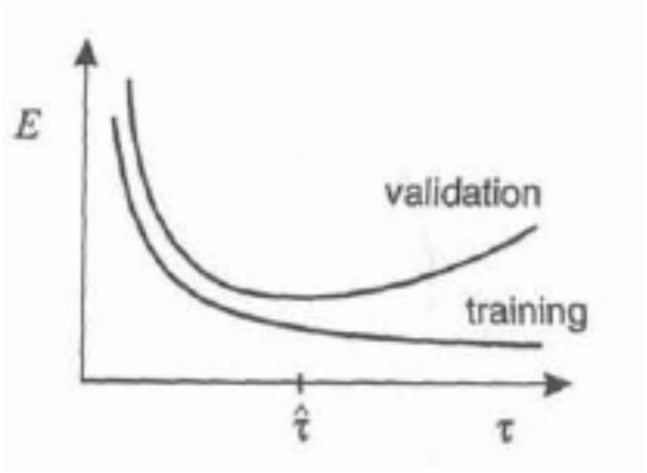
Por tanto la etapa de aprendizaje no debe limitarse a la minimización del error cometido sobre dicho conjunto y por tanto a la memorización de los patrones del conjunto de entrenamiento.

GENERALIZACIÓN

Early stooping

Se divide el conjunto de entrenamiento en: Aprendizaje, Validación.

Early stopping: La fase de entrenamiento se detiene cuando se alcanza un mínimo del error sobre el conjunto de validación.



Antes del mínimo el modelo se considera "underfit" y después "overfit" se ajusta en exceso al conjunto de entrenamiento perdiendo la capacidad de generalización.

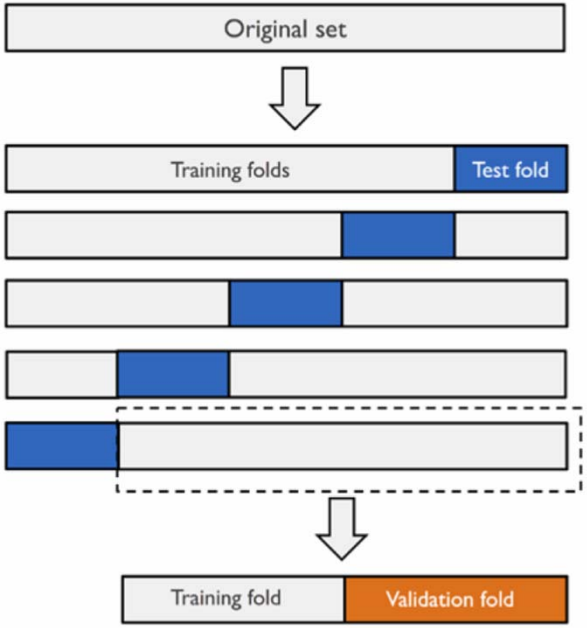
El mínimo del error sobre el conjunto de validación es la figura de mérito utilizada para elegir el modelo que mejor se ajusta a los datos. (Un solo conjunto de validación para conjuntos grandes de prototipos o K-fold en caso contrario).

CLASIFICACIÓN DE DÍGITOS (MNIST)

Ejemplo: nistnnearly.py

Utilizamos la técnica de early stopping como criterio de parada del aprendizaje (es una opción del modelo neuronal).

En este caso la figura de mérito es el coste sobre el conjunto de validación que se elige aleatoriamente como un 20% del conjunto de entrenamiento.



CLASIFICACIÓN DE DÍGITOS (MNIST)

Ejemplo: nistnnearly.py

```
from sklearn import datasets
from sklearn.neural_network import MLPClassifier
import numpy as np
import pickle
from keras.datasets import mnist

(xtrain,ytrain),(xtest,ytest)=mnist.load_data("/home/miguel/images/nist1")
xtrain=xtrain.reshape(60000,784)/255
xtest=xtest.reshape(10000,784)/255

#Exploramos el valor del numero de neuronas que mejor resuelve el problema
neus=np.arange(50,501,50)
errormin=1000.
print(neus)
for numneu in neus:
        print("Simulando con: " + str(numneu) + " neuronas")
        model = MLPClassifier(activation='relu', alpha=1e-05, batch_size='auto',
                        early_stopping=True,validation_fraction=0.2, hidden_layer_sizes=(numneu),
                        learning_rate='constant', learning_rate_init=0.001,
                        max_iter=10000, momentum=0.9, n_iter_no_change=10,
                        shuffle=True, solver='sgd', tol=0.0001,verbose=False)
        model.fit(xtrain,ytrain)
        print("Iteraciones: "+str(model.n_iter_))
        print("Loss: "+str(model.loss_))
        if model.loss_ < errormin:
                errormin=model.loss_
                pickle.dump(model,open("nistnn.net",'wb'))
                print("Bes net: "+str(model.loss_))
#Estimamos con la mejor red la precision sobre el conjunto de test
model = pickle.load(open("nistnn.net",'rb'))
acc = model.score(xtest,ytest)
print(acc)
```

CLASIFICACIÓN DE DÍGITOS (MNIST)

Ejemplo: nistnnearly.py

Los mejores resultados se obtienen con 500 neuronas y un loss=0.0517

La tasa de aciertos sobre el conjunto de test es de 97.57% y el número medio de iteraciones en cada simulación de 220.

Early stopping no supone una mejora en los resultados debido a que los conjuntos de test y training son muy extensos y estadísticamente muy similares.

CLASIFICACIÓN DE DÍGITOS (MNIST)

Ejemplo: nistnngridearly.py

```
from sklearn import datasets
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import GridSearchCV
import numpy as np
import pickle
from keras.datasets import mnist

import tensorflow as tf
config=tf.ConfigProto()
config.gpu_options.allow_growth=True

(xtrain,ytrain),(xtest,ytest)=mnist.load_data("/home/miguel/images/nist1")
xtrain=xtrain.reshape(60000,784)/255
xtest=xtest.reshape(10000,784)/255
#Exploramos el valor del numero de neuronas que mejor resuelve el problema
model = MLPClassifier(batch_size=500,early_stopping=True,validation_fraction=0.2)
#Para definir la busqueda definimos una lista de diccionarios.
param_grid = [{'activation':['relu','logistic'],'hidden_layer_sizes':[(200,),(500,),(1000,)]}]
gs=GridSearchCV(estimator=model,param_grid=param_grid,scoring='accuracy',cv=5,refit=True,n_jobs=-1)
gs=gs.fit(xtrain,ytrain)
print(gs.best_score_)
print(gs.best_params_)
#Estimamos con la mejor red la precision sobre el conjunto de test
clf=gs.best_estimator_
print('Precision sobre el conjunto de test: %.3f' % clf.score(xtest,ytest))
```

```
0.9779833333333333
{'activation': 'relu', 'hidden_layer_sizes': (1000,)}
Precision sobre el conjunto de test: 0.982
```

GENERALIZACIÓN

Prevenir overfitting

Para mejorar la generalización la mejor técnica es aumentar el conjunto de prototipos evitando la redundancia. A veces es difícil.

Si lo anterior no es posible existen otras técnicas basadas en reducir la capacidad del modelo para evitar overfitting (regularización):

- Reducir el tamaño de la red
- Añadir algoritmos para reducir los valores de los pesos (regularización L1 y L2).
- Dropout
- Data augmentation

CLASIFICACIÓN DE DÍGITOS (MNIST)

Ejemplo: nistnnpca.py

Utilizaremos PCA como extractor de características.

Utilizaremos early stopping y neuronas de 50 a 500 en saltos de 50.

Compararemos los resultados de clasificación con 20, 30, 40 y 50 Pcs.

En este caso para un posterior reconocimiento no sólo hay que guardar la red entrenada sino también el modelo PCA para extraer las componentes principales que hemos generado.

CLASIFICACIÓN DE DÍGITOS (MNIST)

Ejemplo: nistnnpca.py

```
from sklearn import datasets
from keras.datasets import mnist
from sklearn.decomposition import PCA
from sklearn.neural_network import MLPClassifier
import numpy as np
import pickle

(xtrain,ytrain),(xtest,ytest)=mnist.load_data("/home/miguel/images/nist1")
xtrain=xtrain.reshape(60000,784)/255
xtest=xtest.reshape(10000,784)/255
#hacemos el pca de train_xdata
pca=PCA(50) #Extreaemos las 50 primeras componentes
pcstrain=pca.fit_transform(xtrain)
pcstest=pca.transform(xtest)
pickle.dump(pca,open("nist.pca",'wb')) #Guardamos el modelo pca para el reconocimiento
#Exploramos el valor del numero de neuronas y de componentes que mejor resuelve el problema
neus=np.arange(50,501,50)
compo=np.arange(20,50,10)
errormin=1000.
for numcompo in compo:
        for numneu in neus:
                print("Simulando con: " + str(numneu) + " neuronas")
                model = MLPClassifier(activation='relu', alpha=1e-05, batch_size='auto',
                        early_stopping=True,validation_fraction=0.2, hidden_layer_sizes=(numneu),
                        learning_rate='constant', learning_rate_init=0.001,
                        max_iter=10000, momentum=0.9, n_iter_no_change=10,
                        shuffle=True, solver='sgd', tol=0.0001,verbose=False)
                model.fit(pcstrain[:,:numcompo],ytrain)
                if model.loss_ < errormin:
                        errormin=model.loss_
                        pickle.dump(model,open("nnpca.net",'wb'))
                        print("Bes net: "+str(model.loss_)+" Neuronas: "+str(numneu)+" Componentes: "+str(numcompo))
#Estimamos con la mejor red la precision sobre el conjunto de test
model = pickle.load(open("nnpca.net",'rb'))
numcompo=model.coefs_[0].shape[0]
acc = model.score(pcstest[:,:numcompo],ytest)
print(acc)
```

CLASIFICACIÓN DE DÍGITOS (MNIST)

Ejemplo: nistnnpca.py

Los mejores resultados se obtienen con 500 neuronas y 50 componentes y un loss=0.057

La tasa de aciertos sobre el conjunto de test es de 97.74%.

En este caso la red tiene un número de parámetros ajustables (pesos) muy inferior a los casos anteriores y por lo tanto, tanto la etapa de aprendizaje como la de reconocimiento son mucho más rápidas.

CLASIFICACIÓN DE DÍGITOS (MNIST)

Ejemplo: reconocistnnpca.py

Utilizaremos el modelo pca y la red neuronal guardadas en el ejemplo anterior para reconocer uno a uno y observar visualmente los caracteres del conjunto de test.

```
from sklearn import datasets
import matplotlib.pyplot as plt
from keras.datasets import mnist
from sklearn.decomposition import PCA
from sklearn.neural_network import MLPClassifier
import numpy as np
import pickle
import getch

(xtrain,ytrain),(xtest,ytest)=mnist.load_data("/home/miguel/images/nist1")
xtrain=xtrain.reshape(60000,784)/255
xtest=xtest.reshape(10000,784)/255
#Cargamos el PCA y lo aplicamos al conjunto de test
pca=pickle.load(open("nist.pca","rb"))
print("Model PCA cargado con "+str(pca.n_components)+" componentes")
model = pickle.load(open("nnpca.net",'rb'))
print("Red neuronal cargada con "+str(model.n_layers_)+" capas")
ninputs=model.coefs_[0].shape[0]
print("De: "+str(ninputs)+","+str(model.hidden_layer_sizes)+","+str(model.n_outputs_)+" neuronas.")
i=0
c='c'
while c == 'c':
        if c == 'c':
                pcs=pca.transform(xtest[i,:].reshape(1,-1))
                out=model.predict(pcs[:,:ninputs])
                plt.imshow(xtest[i,:].reshape(28,28))
                plt.title("Es un "+str(ytest[i])+" y creo que es un: "+str(out))
                plt.pause(0.2)
                i=i+1
        else:
                exit()
        print("c to continue otra tecla to exit")
        c=getch.getch()
```