Astroestadística

Optimización de hiperparámetros de redes neuronales profundas con PSO

Gonzalo Vodanovic

Problema

Objetivos:

Crear una red neuronal profunda que clasifique correctamente las imágenes del set de datos MNIST.

Obtener los hiperparámetros de la red neuronal mediante *Particle Swarm Optimization*.

Lenguaje: Python

Bibliotecas:

NumPy

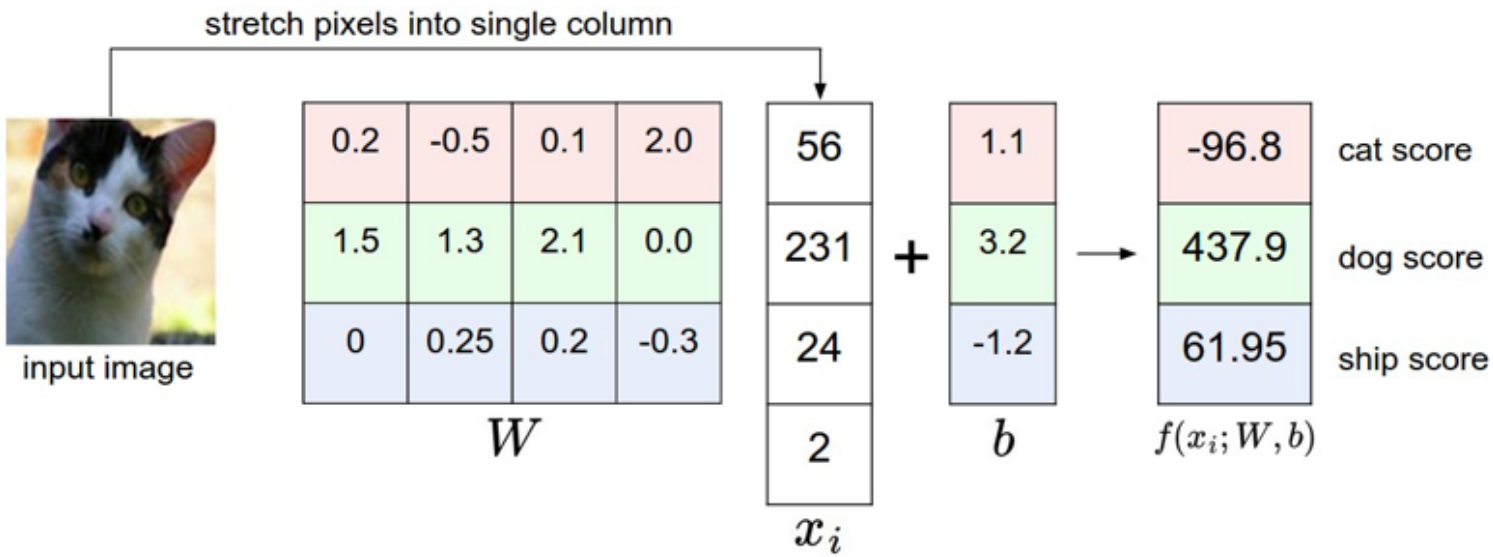
TensorFlow

Keras

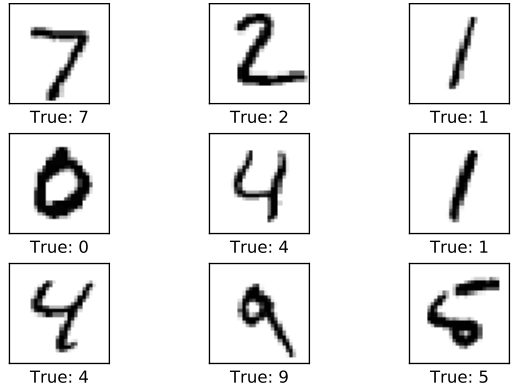
Optunity

Clasificador lineal [1/2]

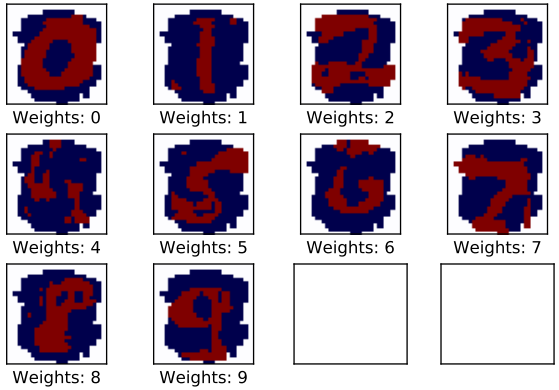
$$f(x_i, W, b) = W\,x_i \;+\; b$$



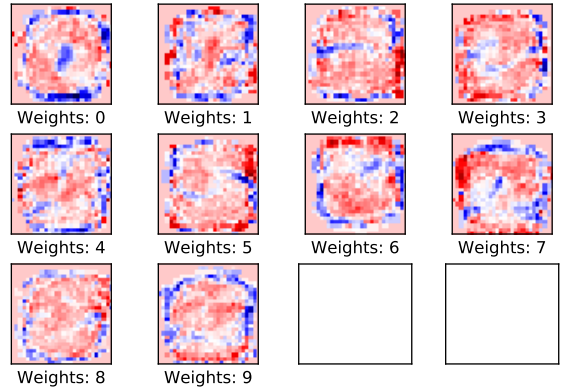
Clasificador lineal [2/2]



100 imágenes



100.000 imágenes



Softmax

Scores: probabilidades logarítmicas no normalizadas de las clases.

$$P(y_i \mid x_i; W) = (\frac{e^{s_k}}{\sum_j e^{s_j}}) \qquad s = f(x_i; W)$$

La entropía cruzada entre la distribución "verdadera" p y la predicha mediante *softmax* q se define como:

$$H(p, q) = -log(\sum_x p(x)\ log\ q(x))$$

$$q = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

La función de pérdida cuantifica las diferencias entre las categorías estimadas y las categorías verdaderas.

$$L_i = -log(\frac{e^{s_k}}{\sum_j e^{s_j}})$$

$$L = \frac{1}{N} \sum_i (-log(\frac{e^{s_k}}{\sum_j e^{s_j}}) + \lambda R(W))$$

SVM

Este clasificador intenta que el score de la clase verdadera sea superior al de las demás clases por un determinado margen Δ.

$$L_i = \sum_{j \neq y_i} max(0, s_j - s_{y_i} + \Delta) \qquad s = f(x_i; W)$$

Por lo general $\Delta = 1.0$

Este limite en cero $max(0, -)$ suele llamarse **hinge loss**

Optimización [1/4]

Objetivo: Encontrar el valor de los parámetros (w) que minimicen la función de pérdida.

Gradient Decent: Algoritmo de optimización iterativo de primer orden para encontrar el mínimo de una función. En cada iteración se mueve un paso en el sentido opuesto del gradiente de la función a minimizar en un punto dado.

```
# Gradient Descent
while True:
        weights_grad = evaluate_gradient(loss_fun, data, weights)
        weights += - step_size * weights_grad
```

```
# Minibatch Gradient Descent (SGD)
while True:
  data_batch = sample_training_data(data, 256)
  weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
  weights += - step_size * weights_grad
```

Optimización [2/4]

Vanilla update

```
x += -learning_rate * dx
```

Momentum

```
v = mu * v - learning_rate * dx # intagrate velocity
x += v # integrate position
```

Optimización [3/4]

Adagrad

```
cache += dx**2
x += -learning_rate * dx / (np.sqrt(cache) + eps)
```

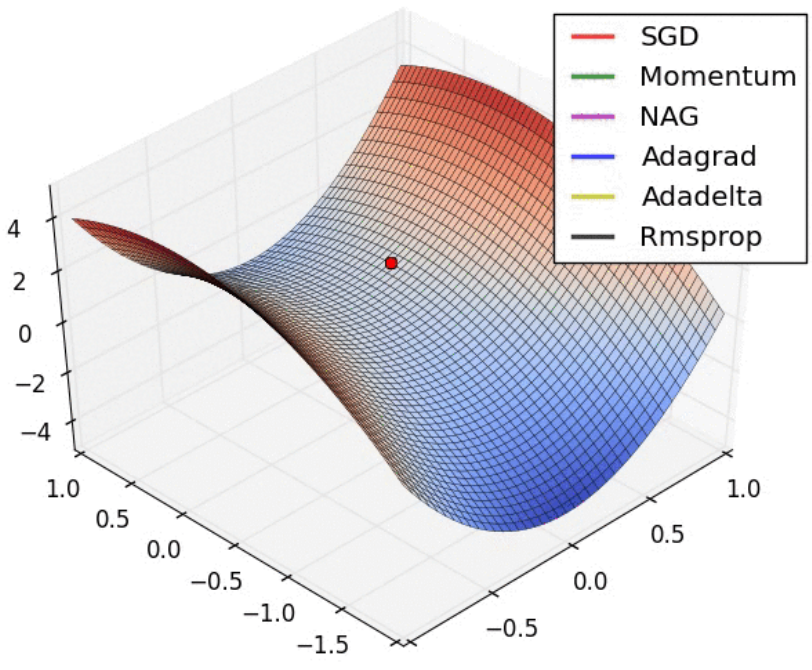
RMSprop

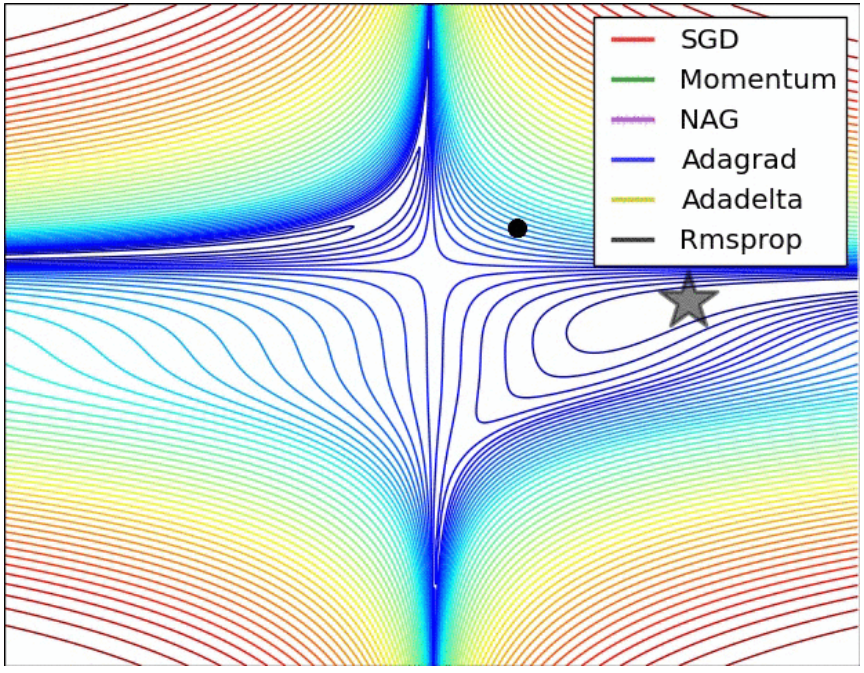
```
cache = decay_rate * cache (1 - decay_rate) * dx**2
x += -learning_rate * dx / (np.sqqrt(cache) + eps)
```

Adam

```
m = beta1*m + (1 - beta1) * dx
v = beta2*v + (1 - beta2) * dx**2
x += -learning_rate * m /(np.sqrt(v) + eps)
```

Optimización [4/4]

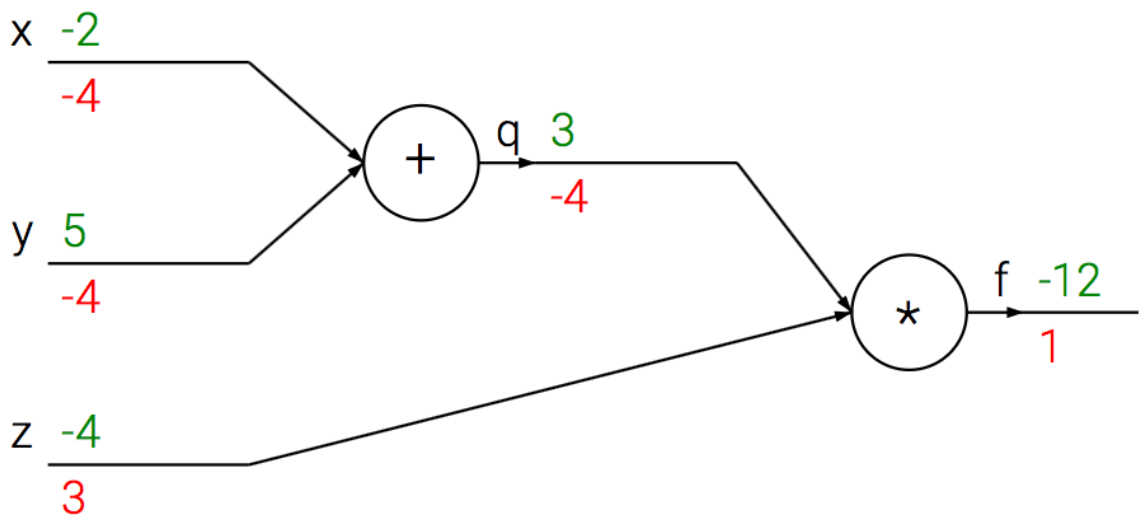




Backpropagation [1/2]

Método para encontrar el gradiente de una función en un punto aplicando la regla de la cadena de las derivadas.

$$f(x, y, z) = (x + y)z \qquad q = x + y \qquad f = q\,z$$



$$\frac{\partial f}{\partial q} = z \qquad \frac{\partial f}{\partial z} = q \qquad \frac{\partial q}{\partial x} = 1 \qquad \frac{\partial q}{\partial y} = 1$$

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x} \qquad \frac{\partial f}{\partial y} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial y} \qquad \frac{\partial f}{\partial z} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial z}$$

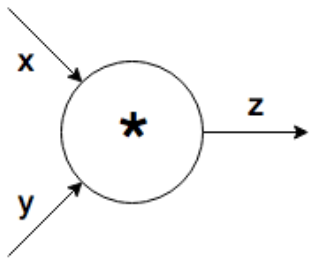
Backpropagation [2/2]

Cada nodo en el diagrama a partir de las entradas computa dos cosas:

1. Resultado operación (suma, multiplicación, etc) (*forward*)
2. El gradiente local (*backward*)

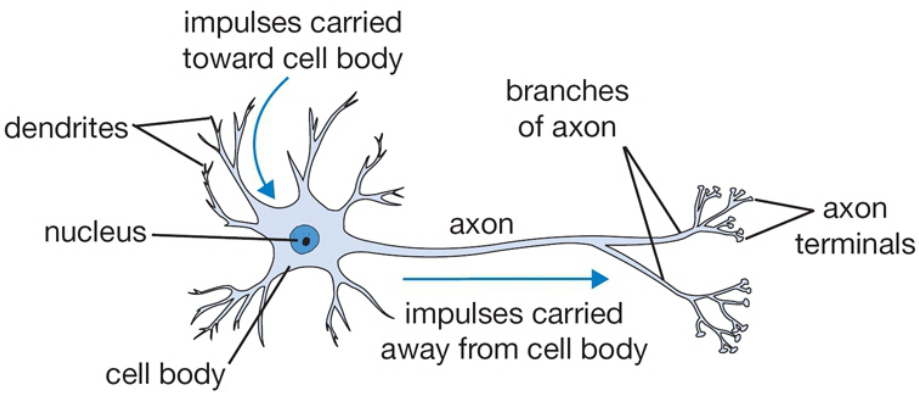
Los nodos realizan ambos cálculos completamente independientemente del resto del circuito (Modularización).

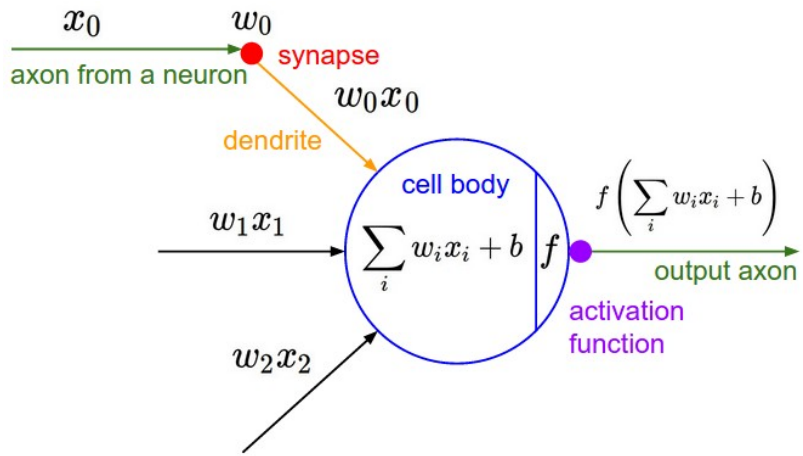
Una vez que finalizó la pasada hacia adelante (*forward*), se calcula el gradiente local a partir del resultado obtenido.




```
class MultiplyGate(Object):
        def forward(x,y):
                z = x * y
                self.x = x
                self.y = y
                return z
        def backward(dz):
                dx = self.y * dz
                dy = self.x * dz
                return [dx,dy]
```

Modelado de una neurona

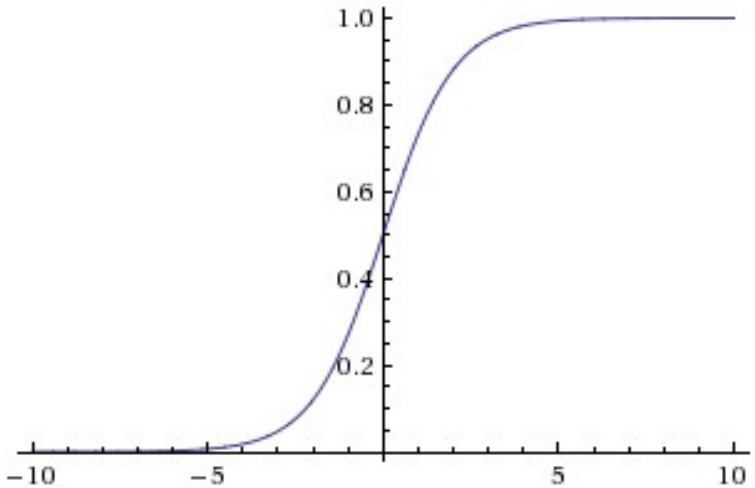




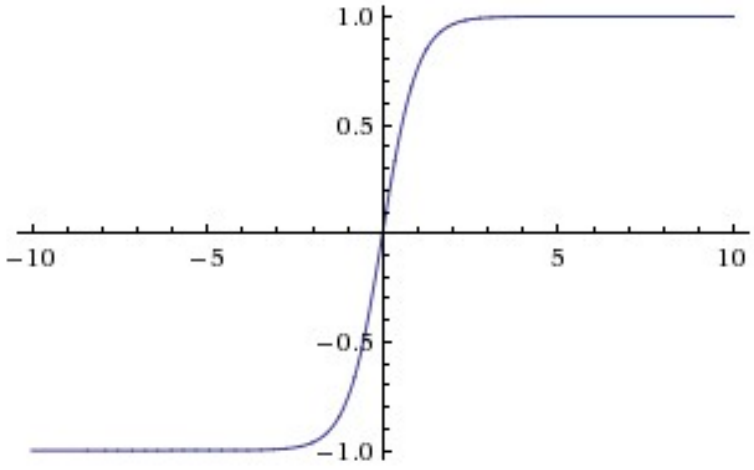

```
class Neuron(object):
  # ...
  def forward(self, inputs):
  # assume inputs and weights are 1-D numpy arrays and bias is a number
  cell_body_sum = np.sum(inputs * self.weights) + self.bias
  # sigmoid activation function
  firing_rate = 1.0 / (1.0 + math.exp(-cell_body_sum))
  return firing_rate
```

Funciones de Activación [1/2]

Sigmoid



tanh

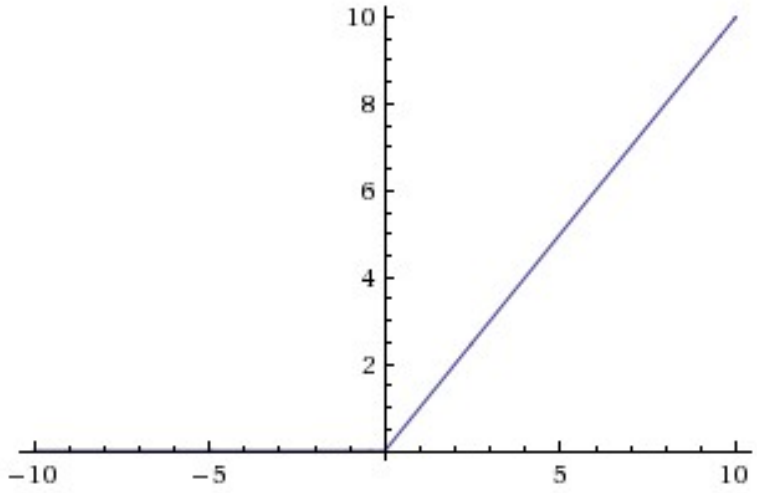


Desventajas Sigmoid:

- La saturación para grandes magnitudes (positivas y negativas).
- La salida no esta centrada en cero.

Funciones de Activación [2/2]

ReLU:



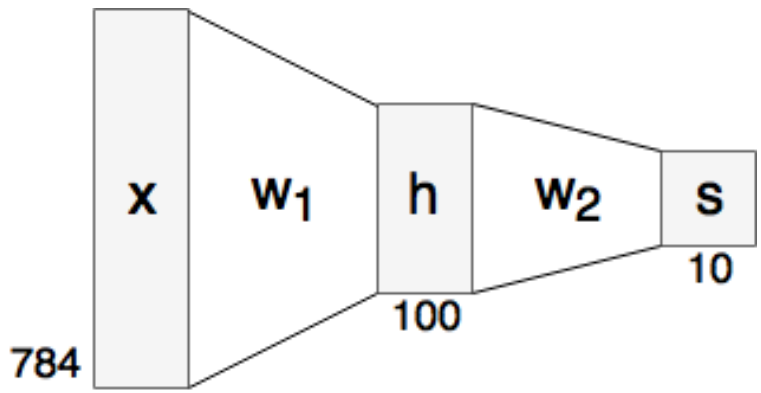
Características:

- Converge mas rápido que las funciones sigmoid y tanh.
- La implementación es sencilla, no requiere funciones computacionalmente caras (ej: exponenciales, tanh, etc)
- Satura en la región negativa. Leaky ReLU mejora esto: $f(x) = max(0.01x, x)$

Redes Neuronales

Linear score function: $f = W\,x$

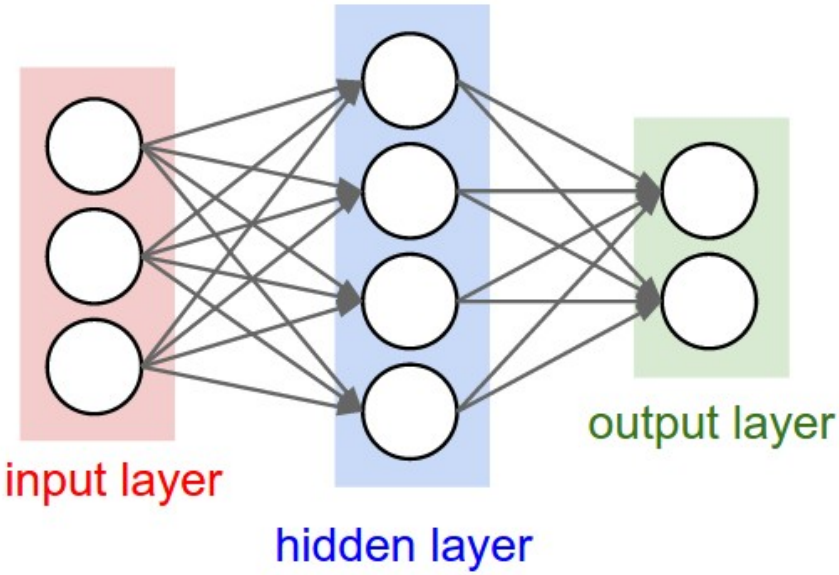
Red neuronal de dos capas $f = W_2\,max(0, W_1\,x)$

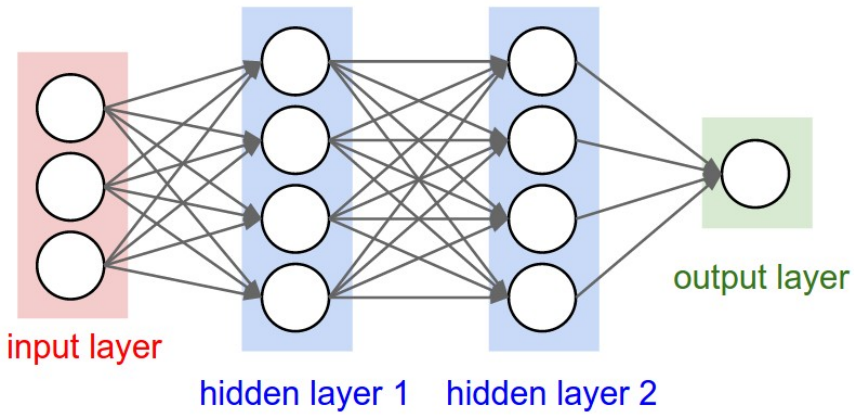


Red neuronal de tres capas $f = W_3\,max(0, W_2\,max(0, W_1\,x)$

Arquitectura de Redes Neuronales

Capa: Fully-connected






```
# forward-pass of a 3-layer neural network:
f = lambda x: 1.0/(1.0 + np.exp(-x)) # activation function (use sigmoid)
h1 = f(np.dot(W1, x) + b1) # calculate first hidden layer activations (4x1)
h2 = f(np.dot(W2, h1) + b2) # calculate second hidden layer activations (4x1)
out = np.dot(W3, h2) + b3 # output neuron (1x1)
```

Clasificador Lineal en TensorFlow

```
# Placeholder variables
x = tf.placeholder(tf.float32, [None, img_size_flat])
y_true = tf.placeholder(tf.float32, [None, num_classes])
y_true_cls = tf.placeholder(tf.int64, [None])
# Model Variables
weights = tf.Variable(tf.zeros([img_size_flat, num_classes]))
biases = tf.Variable(tf.zeros([num_classes]))
# Model
logits = tf.matmul(x, weights) + biases
y_pred = tf.nn.softmax(logits)
y_pred_cls = tf.argmax(y_pred, axis=1)
# Cost function
cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits=logits,
                                                        labels=y_true)
cost = tf.reduce_mean(cross_entropy)
# Optimization
optimizer = tf.train.AdamOptimizer(learning_rate=0.5).minimize(cost)
def optimize(num_iterations):
for i in range(num_iterations):
    x_batch, y_true_batch = data.train.next_batch(batch_size)
    feed_dict_train = {x: x_batch, y_true: y_true_batch}
    session.run(optimizer, feed_dict=feed_dict_train)
# TensorFlow Session
session = tf.Session()
session.run(tf.global_variables_initializer())
optimize(num_iterations=N)
```

Red Neuronal de dos capas en Keras

```
# Start construction of a Keras Sequential model.
model = Sequential()

# Build 2 Layers model
model.add(InputLayer(input_shape=(img_size_flat,)))
model.add(Dense(128,activation='relu',name='layer1'))
model.add(Dense(num_classes, activation='softmax'))

# Adam Optimizer
optimizer = Adam(lr=learning_rate)

# In Keras we need to compile the model so it can be trained.
model.compile(optimizer=optimizer,
                          loss='categorical_crossentropy',
                          metrics=['accuracy'])

# Training
model.fit(x=X_train,y=y_train,epochs=3,batch_size=batch_size)

# Evaluation
result = model.evaluate(x=X_test,y=y_test)
```

Red Neuronal Convolucional en Keras

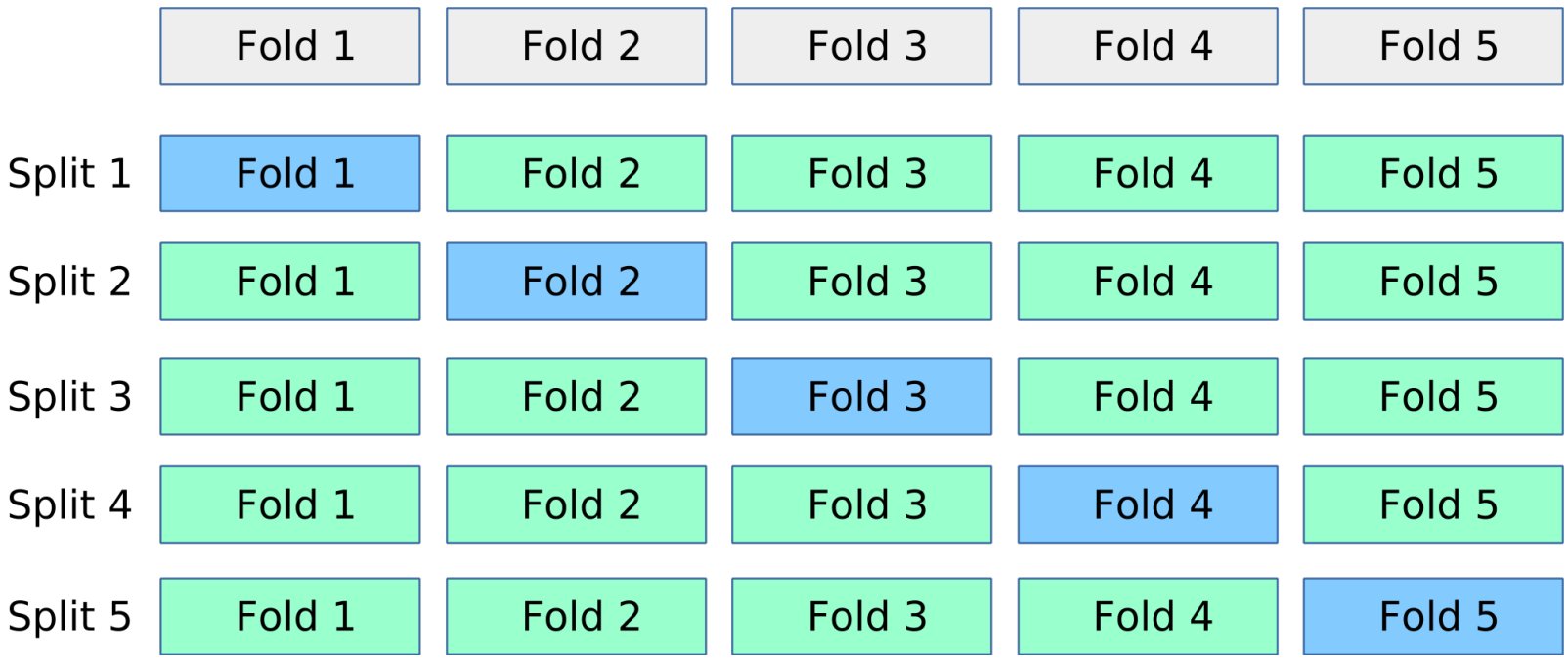
```
model = Sequential()

# Build 2 Layers Convolutional model
model.add(InputLayer(input_shape=(img_size_flat,)))
model.add(Reshape(img_shape_full))
model.add(Conv2D(kernel_size=5, strides=1, filters=16, padding='same',
                 activation='relu', name='layer_conv1'))
model.add(MaxPooling2D(pool_size=2, strides=2))
model.add(Conv2D(kernel_size=5, strides=1, filters=36, padding='same',
                 activation='relu', name='layer_conv2'))
model.add(MaxPooling2D(pool_size=2, strides=2))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(num_classes, activation='softmax'))

# Adam Optimizer
optimizer = Adam(lr=1e-3)

# Compile
model.compile(optimizer=optimizer,
      loss='categorical_crossentropy',
      metrics=['accuracy'])
# Training
model.fit(x=data.train.images,
          y=data.train.labels,
          epochs=1, batch_size=128)
# Evaluation
result = model.evaluate(x=data.test.images,=data.test.labels)
```

Hiperparámetros

Hiperparámetros a optimizar:

- Número de capas
- Cantidad de neuronas en cada capa
- Función de activación
- *learning rate*

K-fold cross validation



Particle Swarm optimization [1/3]

Si x es un punto en un espacio de búsqueda de D dimensiones y $F(x)$ es una función que computa la calidad del punto x, el algoritmo PSO computa $F(x)$ en varios puntos simultaneamente mediante partículas que comparten información. Se determinan nuevos puntos de exploración gracias a su experiencia individual y la colectiva.

Partícula: Agente computacional que explora el espacio de búsqueda. Cada una tiene una posición $X_i(t)$ y una velocidad $V_i(t)$.

Función de fitness: Función que evalúa la calidad del punto x como solución.

Mejor valor individual: Cada partícula guarda la posición que obtuvo el máximo valor de fitness hasta el momento.

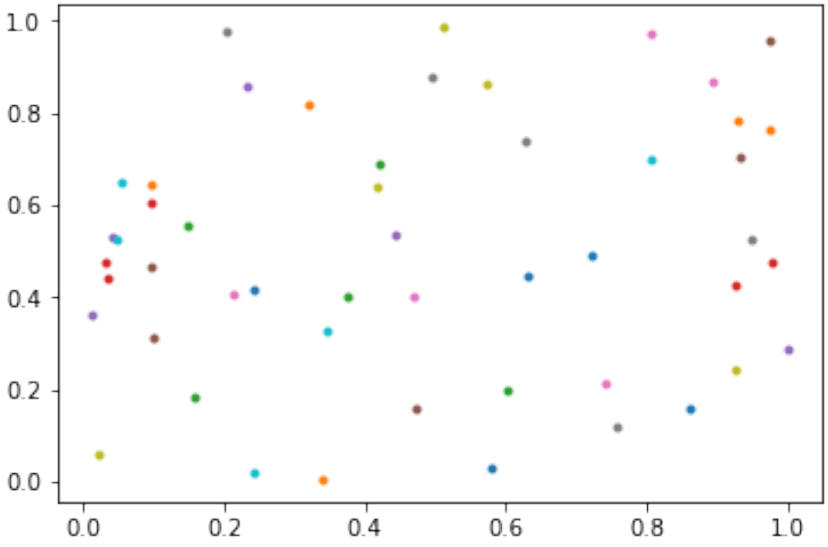
Mejor valor global: Es la posición que obtuvo el máximo valor de fitness entre todas las partículas.

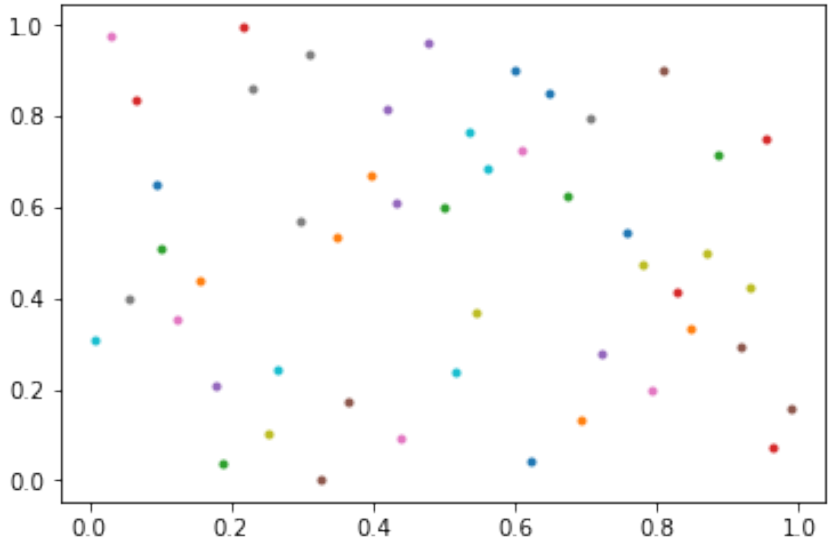
En cada nueva iteración la posición y la velocidad de cada partícula se actualiza mediante la siguiente ecuación:

$$X_i(t+1) = X_t + V_i(t+1) \quad V_{i+1} = wV_i(t) + c_1\psi_1[B_i(t) - X_i(t)] + c_2\psi_2[G(t) - X_i(t)]$$

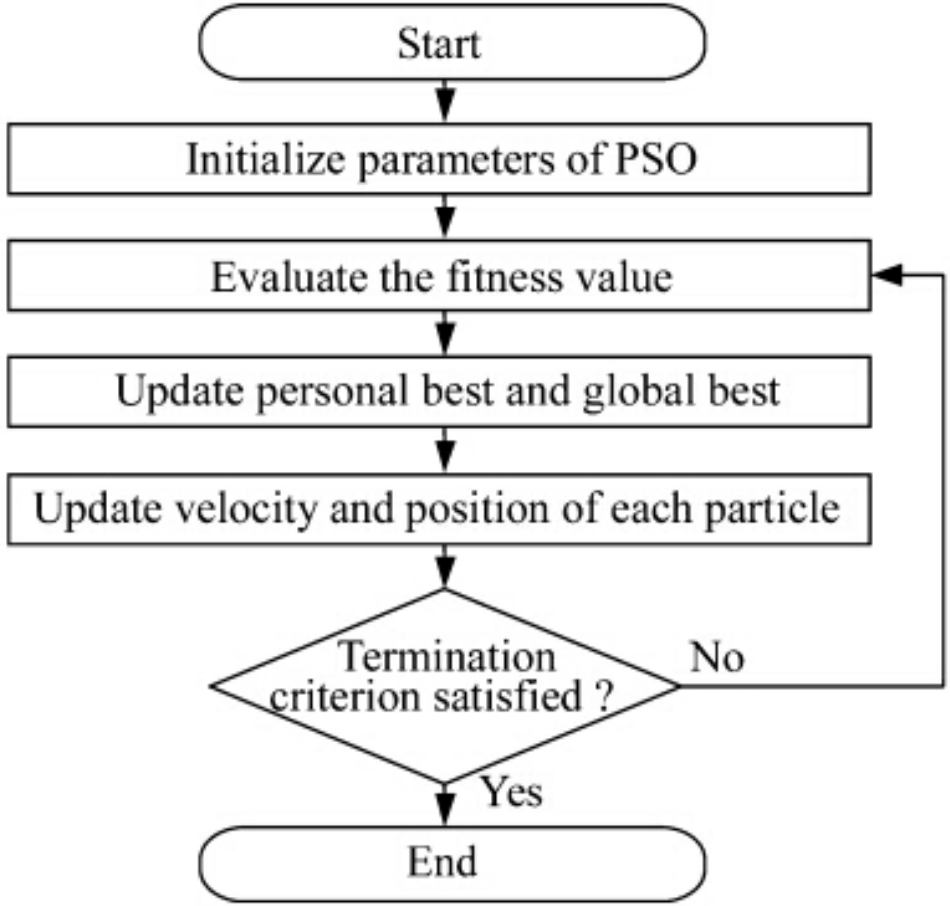
Particle Swarm optimization [2/3]

La posición inicial de las partículas se asigna aleatoriamente utilizando *Maximin Latin Hipercube*, una técnica que explora el espacio multidimensional eficientemente.





Particle Swarm optimization [3/3]



Resultados

Jupyter Notebook