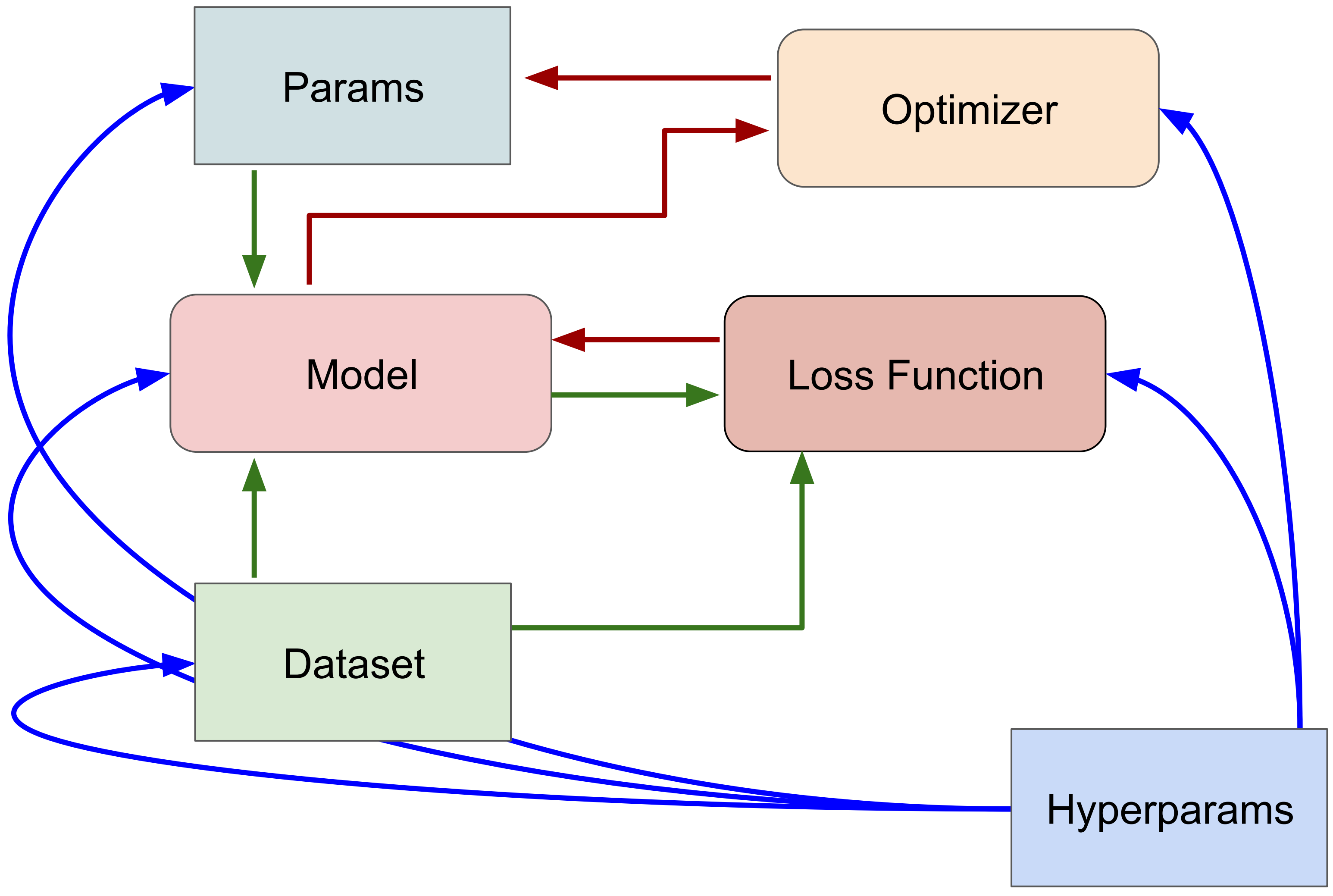
Unidad 2: Redes Neuronales Artificiales

Curso: Redes Neuronales Profundas

Visión Modular



Funciones de costo

Teorema de Bayes

$$p(h|D) = \frac{p(D|h)\,p(h)}{p(D)}$$

Probabilidad de haber visto los datos, suponiendo la hipótesis

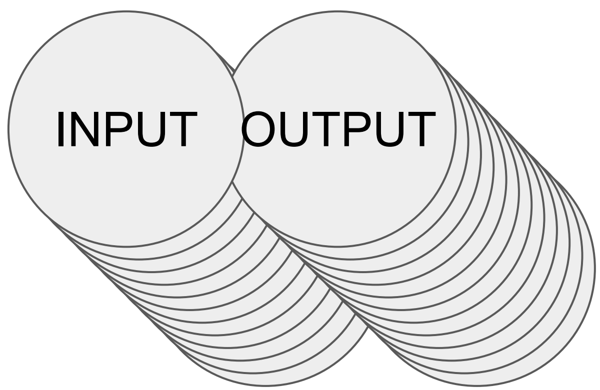
Likelihood. Si suponemos i.i.d., es un producto.

$$\sum_{d_i \in D} \log(p(d_i|h))$$

Log likelihood

Aprendizaje supervisado

SUPERVISADO

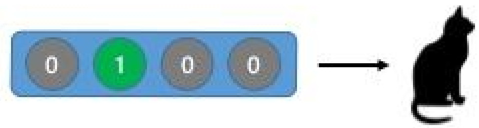


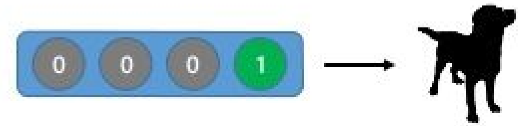
$$\sum_{d_i \in D} \log(p(d_i|h))$$

$$\sum_{(x_i, y_i) \in D} \log(p(y_i|x_i, h))$$

No se modela (explícitamente) la probabilidad de los inputs

Clasificación





$$\sum_{(x_i,y_i)\in D} \log(p(y_i|x_i,h))$$

Representación One hot: $p_{ij} \in \{0,1\}$

i-ésimo dato

j-ésima clase

El modelo es una función f tal que dada x_i devuelve la probabilidad q_{ij} para cada una de las clases

$$-\sum_{i=1}^{N}\sum_{j=1}^{C} p_{ij}\log(q_{ij})$$

Cross-entropy

`theano.tensor.nnet.categorical_crossentropy`(*coding_dist*, *true_dist*)

Return the cross-entropy between an approximating distribution and a true distribution. The cross entropy between two probability distributions measures the average number of bits needed to identify an event from a set of possibilities, if a coding scheme is used based on a given probability distribution q, rather than the “true” distribution p. Mathematically, this function computes

$$H_i(p, q) = -\sum_{j=1}^{C} p_{ij} \log(q_{ij})$$

where p=true_dist and q=coding_dist.

Parameters:	• *coding_dist* - symbolic 2D Tensor (or compatible). Each row represents a distribution. • *true_dist* - symbolic 2D Tensor **OR** symbolic vector of ints. In the case of an integer vector argument, each element represents the position of the ‘1’ in a 1-of-N encoding (aka “one-hot” encoding)
Return type:	tensor of rank one-less-than coding_dist

`theano.tensor.nnet.categorical_crossentropy`(*coding_dist*, *true_dist*)

Note:
An application of the scenario where *true_dist* has a 1-of-N representation is in classification with softmax outputs. If coding_dist is the output of the softmax and true_dist is a vector of correct labels, then the function will compute

```
y_i = - \log(coding_dist[i, one_of_n[i]])
```

which corresponds to computing the neg-log-probability of the correct class (which is typically the training criterion in classification settings).

```
y = T.nnet.softmax(T.dot(W, x) + b)
cost = T.nnet.categorical_crossentropy(y, o).mean()
# o is either the above-mentioned 1-of-N vector or 2D tensor
```

Clasificación binaria

$$-\sum_{i=1}^{N}\sum_{j=1}^{C=2} p_{ij}\log(q_{ij}) \qquad \text{Cross-entropy}$$

$$-\sum_{i=1}^{N}\left[p_{i1}\log(q_{i1}) + p_{i2}\log(q_{i2})\right]$$

$$-\sum_{i=1}^{N}\left[p_{i}\log(q_{i}) + (1-p_{i})\log(1-q_{i})\right]$$

```
p_1 = 1 / (1 + T.exp(-T.dot(x, w) - b)) # Prob. that target = 1
prediction = p_1 > 0.5
xent = -y * T.log(p_1) - (1-y) * T.log(1-p_1) # Cross-entropy
cost = xent.mean() + 0.01 * (w ** 2).sum()
```

Regresión

Modelo $$y = f(x) + \epsilon$$ Normal

$$p(y|x) = \mathcal{N}(y|f(x), \sigma)$$

$$p(y|x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{y-f(x)}{\sigma}\right)^2}$$

Regresión

$$\log p(y|x)$$

$$-\log(\sigma\sqrt{2\pi}) - \frac{1}{\sigma^2}\left[\frac{1}{2}(y - f(x))^2\right]$$

Si sumamos para todos los $(x_i, y_i) \in D$,

$$-N\log(\sigma\sqrt{2\pi}) - \frac{1}{\sigma^2}\sum_{(x_i, y_i)\in D}\left[\frac{1}{2}(y_i - f(x_i))^2\right]$$

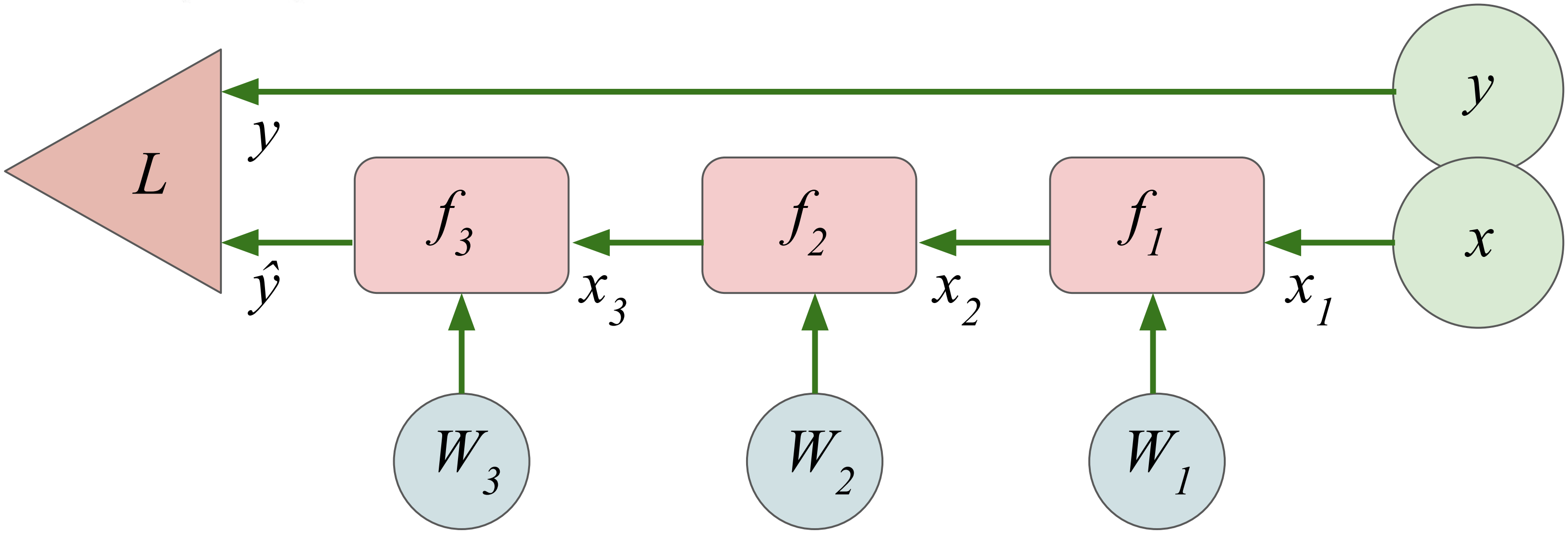
Funciones de costo en Theano

```
import theano
import theano.tensor as T

def CategoricalCrossEntropy(y_true, y_pred):
    return T.nnet.categorical_crossentropy(y_pred, y_true).mean()

def BinaryCrossEntropy(y_true, y_pred):
    return T.nnet.binary_crossentropy(y_pred, y_true).mean()

def MeanSquaredError(y_true, y_pred):
    return T.sqr(y_pred - y_true).mean()

def MeanAbsoluteError(y_true, y_pred):
    return T.abs_(y_pred - y_true).mean()
```

Descenso por gradiente

Backpropagation

$$L = L(y, \hat{y}) \qquad \hat{y} = F(x)$$



$$F(x) = f_3\Big(W_3, f_2\big(W_2, f_1(W_1, x)\big)\Big)$$

$$F(x) = f_3(W_3) \circ f_2(W_2) \circ f_1(W_1)(x)$$

Backpropagation

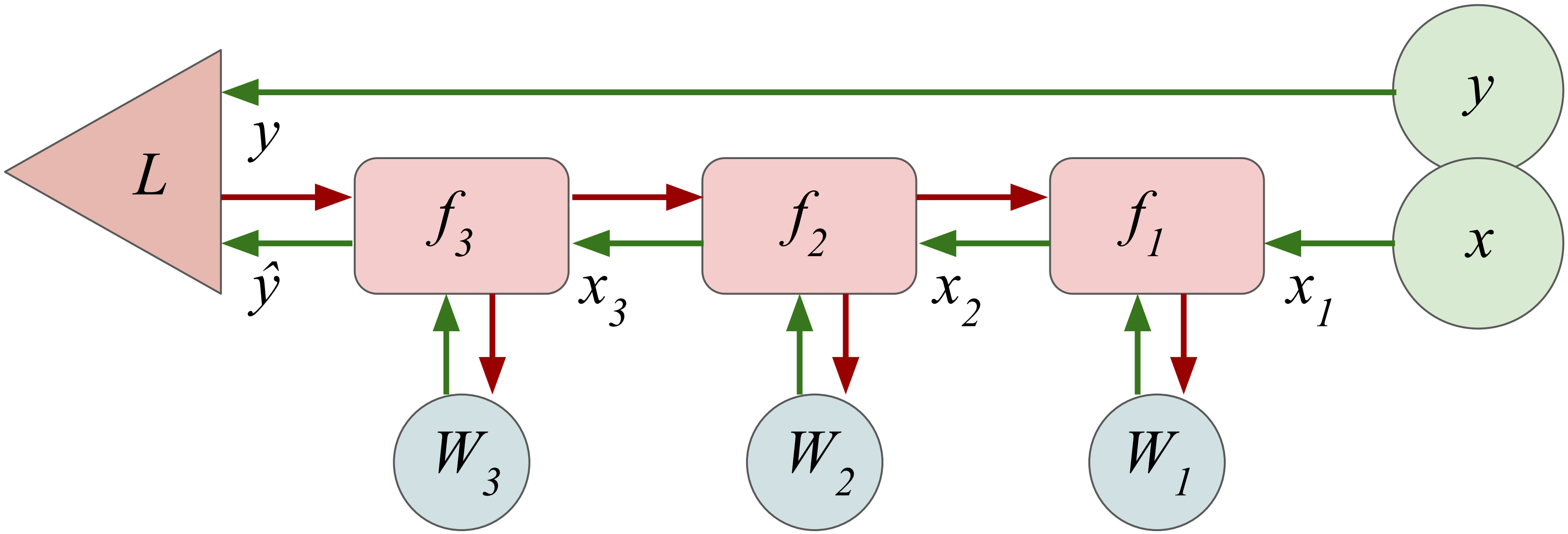
$$\begin{aligned} x_1 &= x \\ x_2 &= f_1(W_1, x_1) \\ x_3 &= f_2(W_2, x_2) \\ \hat{y} &= f_3(W_3, x_3) \end{aligned}$$

$$L = L(y, \hat{y})$$

$$\frac{\partial L}{\partial W_3} = \underbrace{\frac{\partial L}{\partial \hat{y}}}_{\delta} \cdot \frac{\partial f_3}{\partial W_3}$$

$$\frac{\partial L}{\partial W_2} = \underbrace{\frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial f_3}{\partial x_3}}_{\delta} \cdot \frac{\partial f_2}{\partial W_2}$$

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial f_3}{\partial x_3} \cdot \frac{\partial f_2}{\partial x_2} \cdot \frac{\partial f_1}{\partial W_1}$$

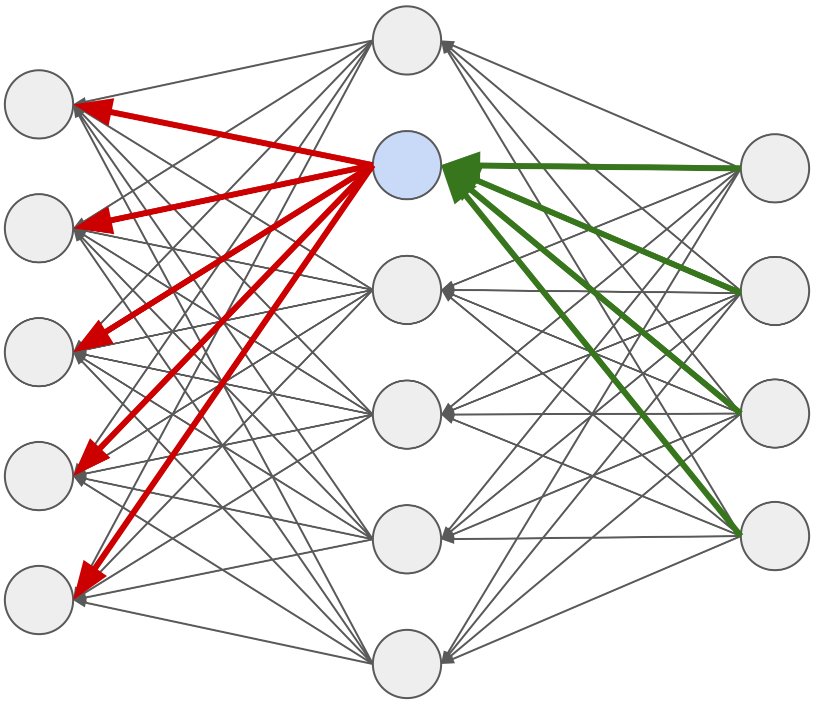


$$\frac{\partial L}{\partial W_3} = \underbrace{\frac{\partial L}{\partial \hat{y}}}_{\downarrow\, \delta_{bp}} \cdot \frac{\partial f_3}{\partial W_3}$$

$$\frac{\partial L}{\partial W_2} = \underbrace{\frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial f_3}{\partial x_3}}_{\downarrow\, \delta_{bp}} \cdot \frac{\partial f_2}{\partial W_2}$$

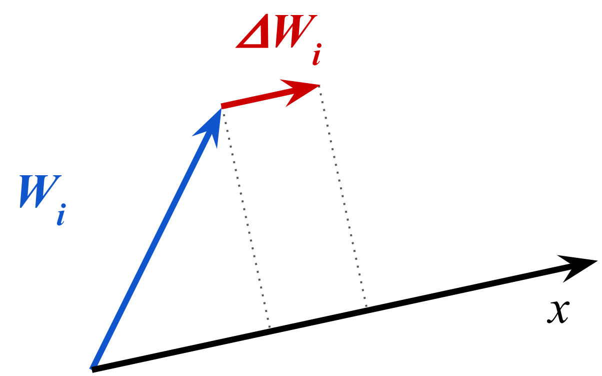
$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial f_3}{\partial x_3} \cdot \frac{\partial f_2}{\partial x_2} \cdot \frac{\partial f_1}{\partial W_1}$$

Backprop sobre una neurona



$$
\begin{aligned}
z_i &= W_i \cdot x + b_i \\
h_i &= \mathrm{relu}(z_i)
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial L}{\partial W_i} &= \delta_{bp} \frac{\partial h_i}{\partial z_i} \frac{\partial z_i}{\partial W_i} \\
\frac{\partial L}{\partial W_i} &= \delta_{bp} \, \mathrm{step}(z_i) x
\end{aligned}
$$



SGD

```
class SGD:
    def __init__(self, lr=0.01):
        self.lr = lr

    def __call__(self, params, cost):
        updates = []
        grads = T.grad(cost, params)
        for p,g in zip(params,grads):
            updated_p = p - self.lr * g
            updates.append((p, updated_p))
        return updates
```

SGD + Momentum

```
class Momentum:
    def __init__(self, lr=0.01, momentum=0.9):

    def __call__(self, params, cost):
        updates = []
        grads = T.grad(cost, params)
        for p,g in zip(params,grads):
            m = theano.shared(p.get_value() * 0.)
            v = (self.momentum * m) - (self.lr * g)
            updates.append((m, v))

            updated_p = p + v
            updates.append((p, updated_p))
        return updates
```

Nesterov

```
class NAG:
    def __init__(self, lr=0.01, momentum=0.9):

    def __call__(self, params, cost):
        updates = []
        grads = T.grad(cost, params)
        for p, g in zip(params, grads):
            m = theano.shared(p.get_value() * 0.)
            v = (self.momentum * m) - (self.lr * g)

            updated_p = p + self.momentum * v - self.lr * g
            updates.append((m,v))
            updates.append((p, updated_p))
        return updates
```

Momentum común: primero computa el gradiente, luego hace un salto grande en la dirección del gradiente acumulado.

Mejor: Primero hacer el salto grande y luego medir el gradiente.

Inconvenientes de SGD tradicional

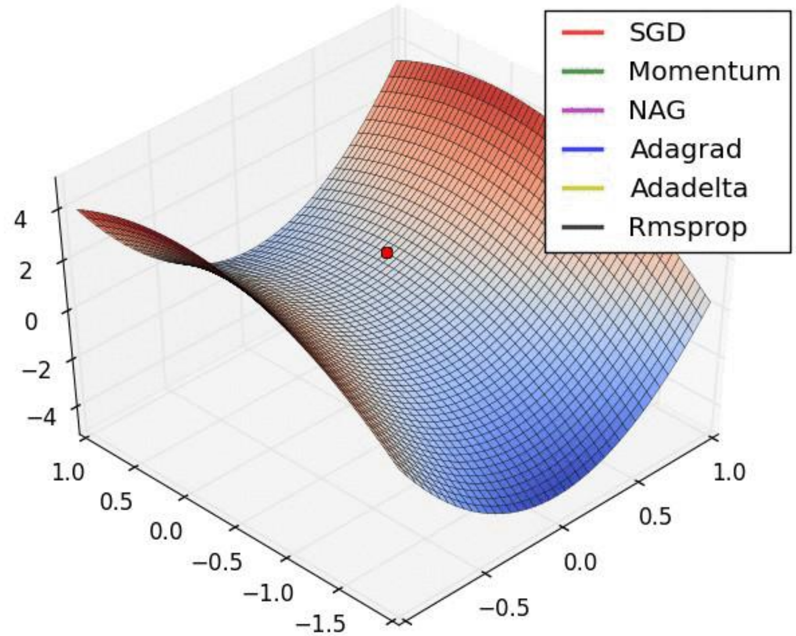
- ¿Cómo elegimos el learning rate?
- El learning rate es el mismo para todos los parámetros. Para features que se dan raramente, puede ser que convenga utilizar saltos más grandes.
- Trampas: mínimos locales subóptimos (raros) o puntos de ensilladura donde el gradiente tiende a 0.

AdaGrad

Idea: hacer updates grandes para parámetros infrecuentes y updates chicos para parámetros frecuentes. Sirve para sparse data.

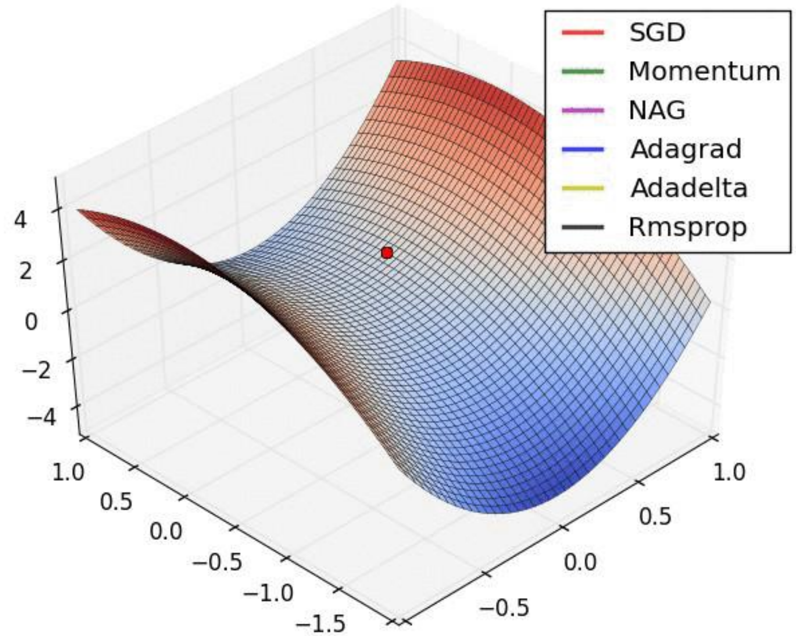
El learning rate correspondiente a un parámetro p en el tiempo T se escala con

$$\sqrt{\sum_{t=1}^{T} grad_t(p)^2}$$



AdaGrad

```
class Adagrad:
    def __init__(self, lr=0.01, epsilon=1e-6):

    def __call__(self, params, cost):
        updates = []
        grads = T.grad(cost, params)
        for p,g in zip(params,grads):
            acc = theano.shared(p.get_value() * 0.)
            acc_t = acc + g ** 2
            updates.append((acc, acc_t))

            p_t = p - (self.lr / T.sqrt(acc_t + self.epsilon)) * g
            updates.append((p, p_t))
        return updates
```



RMSprop

```
class RMSprop:
    def __init__(self, lr=0.001, rho=0.9, epsilon=1e-6):

    def __call__(self, params, cost):
        updates = []
        grads = T.grad(cost, params)
        for p,g in zip(params,grads):
            acc = theano.shared(p.get_value() * 0.)
            acc_t = self.rho * acc + (1 - self.rho) * g ** 2
            updates.append((acc, acc_t))

            p_t = p - (self.lr / T.sqrt(acc_t + self.epsilon)) * g
            updates.append((p, p_t))
        return updates
```

Sumar sobre todo el pasado trae problemas.

Solución: usar un decay exponencial

Adam

Combinación de RMSprop con Momentum

```
class Adam:
    def __init__(self, lr=0.001, b1=0.9, b2=0.999,
                    e=1e-8, l=1-1e-8):

    def __call__(self, params, cost):
        updates = []
        grads = T.grad(cost, params)
        t = theano.shared(floatX(1.))
        b1_t = self.b1*self.l**(t-1)

        for p, g in zip(params, grads):
            m = theano.shared(p.get_value() * 0.)
            v = theano.shared(p.get_value() * 0.)

            m_t = b1_t*m + (1 - b1_t)*g
            v_t = self.b2*v + (1 - self.b2)*g**2
            m_c = m_t / (1-self.b1**t)
            v_c = v_t / (1-self.b2**t)
            p_t = p - (self.lr * m_c) /
                        (T.sqrt(v_c) + self.e)
            updates.append((m, m_t))
            updates.append((v, v_t))
            updates.append((p, p_t) )
            updates.append((t, t + 1.))
        return updates
```

¿Cuál usar?

- No hay una diferencia importante entre los diferentes métodos con learning rate adaptativo. Kingma muestra una leve mejora de Adam frente a RMSprop.
- Con una buena inicialización, SGD puede llegar a mínimos similares. Pero con más tiempo.
- Para datos o representaciones internas sparse, los métodos adaptativos muestran su ventaja. ¿Recuerdan las ReLU?