

Aprenentatge Automàtic 2

Carlos Arbonés and Juan P. Zaldivar

GCED, UPC.

Apunts.

Contents

| | |
|---|-----------|
| 1 Basic Elements in Neuronal Networks | 4 |
| 1.1 Perceptron and MLP | 4 |
| 1.1.1 Regression vs Classification | 4 |
| 1.1.2 Perceptron | 5 |
| 1.1.3 Multi-Layer Perceptron (MLP) | 8 |
| 1.2 Backpropagation | 11 |
| 1.2.1 Forward Pass | 11 |
| 1.2.2 Gradients from composition: Chain rule | 11 |
| 1.2.3 Backward pass | 13 |
| 1.2.4 Gradient Descent | 13 |
| 1.3 Losses | 14 |
| 1.3.1 Loss Function | 14 |
| 1.3.2 Properties of the loss function | 15 |
| 1.3.3 Losses with multiples inputs | 15 |
| 1.3.4 Loss function in regression problems L1 vs L2 | 15 |
| 1.3.5 Huber Loss | 16 |
| 1.3.6 Losses in Classification Problems | 16 |
| 1.3.7 Multi-label | 19 |
| 1.3.8 Aprendizaje de Métricas | 20 |
| 1.4 Optimizers | 22 |
| 1.4.1 Batch Gradient Descent | 22 |
| 1.4.2 Stochastic Gradient Descent | 22 |
| 1.4.3 Mini-Batch Gradient Descent | 23 |
| 1.4.4 Potential Problems | 24 |
| 1.4.5 Local minima and saddle points | 24 |
| 1.4.6 Learning Rate | 25 |
| 1.4.7 Parameter Initialization | 25 |
| 1.4.8 Algorithms | 25 |
| 1.5 CNNs: Convolutional and Pooling Layers | 27 |
| 1.5.1 Convolutional Layers | 29 |
| 1.5.2 Batch normalization and Pooling Layers | 30 |
| 1.6 CNN: Upsampling and Skip connections | 31 |
| 1.6.1 Introduction | 31 |
| 1.6.2 Upsampling | 32 |
| 1.6.3 Skip connections | 34 |
| 1.7 Architectures and Interpretability | 35 |
| 1.7.1 VGGNet-16 | 35 |
| 1.7.2 Inception Module | 36 |
| 1.7.3 GoogLeNet | 37 |
| 1.7.4 ResNet | 37 |
| 1.7.5 Interpretability | 37 |
| 2 Practical Aspects in Neuronal Networks | 38 |
| 2.1 Capacity of the Network | 38 |
| 2.1.1 Generalization | 39 |
| 2.1.2 Overfitting | 39 |
| 2.1.3 Underfitting | 39 |
| 2.2 Data Partition | 39 |
| 2.3 Prevent Overfitting | 39 |
| 2.3.1 Early Stopping | 40 |

| | | |
|----------|--|-----------|
| 2.3.2 | Weight Regularization | 41 |
| 2.3.3 | Activation Regularization | 41 |
| 2.3.4 | Batch Normalization | 42 |
| 2.3.5 | Dropout | 42 |
| 2.3.6 | Data Augmentation | 43 |
| 2.3.7 | Label Smoothing | 43 |
| 2.3.8 | Parameter Sharing | 43 |
| 2.4 | Strategy | 43 |
| 2.5 | Transfer Learning | 43 |
| 2.5.1 | General techniques | 44 |
| 2.5.2 | Domain Adaptation | 45 |
| 2.5.3 | Task transfer | 46 |
| 3 | Advanced Architectures | 48 |
| 3.1 | Recurrent Neural Networks | 48 |
| 3.1.1 | Forward and Backward passes thought time | 50 |
| 3.1.2 | Long Short Term Memory (LSTM) | 51 |
| 3.1.3 | Neural Machine Translation NMT | 55 |
| 3.2 | Attention | 55 |
| 3.2.1 | Key, Query, Value | 56 |
| 3.2.2 | Attention Mechanisms | 58 |
| 3.3 | Generative Networks | 60 |
| 3.3.1 | Unsupervised Learning | 60 |
| 3.3.2 | Generative Models | 60 |
| 3.3.3 | Explicit Density Models | 60 |
| 3.3.4 | Implicit Density Models | 64 |

1 Basic Elements in Neuronal Networks

1.1 Perceptron and MLP

1.1.1 Regression vs Classification

Queremos **predecir** una variable dependiente y a partir de un conjunto de variables (x_1, x_2, \dots, x_M) o *input sample* $x \in \mathbb{R}^M$. Dada la naturaleza del *target* (y) que queremos predecir podemos estar delante de 2 tipos de problemas:

- **Regresión:** la variable y es **continua**. Por ejemplo si queremos predecir el precio de una casa.
- **Clasificación:** la variable y es **discreta**. Por ejemplo si queremos predecir si el animal en una fotografía es un gato, un perro o un caballo.

Linear Regression (1D input)

Tenemos una serie de puntos, necesitamos **aproximar los datos linealmente** y encontramos un cierta pendiente w . Hay veces que necesitamos también un sesgo b ya que la recta no pasa por el origen. Tenemos una **aproximación de nuestros puntos** que son datos reales de tal modo que si alguien nos da un valor **podemos devolver el valor aproximado**.

$$\hat{y} = wx + b$$

Linear Regression (MD input)

Nos pueden dar toda una serie de datos para combinar linealmente.

$$\hat{y} = \mathbf{w}^T \mathbf{x} + b = w_1 x_1 + w_2 x_2 + \dots + w_M x_M + b$$

Binary Classification (1D input)

También se puede tener el caso en el que se desea **predecir una etiqueta** $y \in \{0, 1\}$. En base a esta información, se quisiera tener una diferenciación entre las clases.

$$\hat{y} = g(wx + b)$$

Si lo que queremos es separar a estas dos poblaciones parece evidente que no se podrá hacer de forma perfecta. Lo que podemos hacer es lo que se muestra en la Figura 1.1, encontramos una recta que separa de mejor forma posible los datos y ajustar una función sobre esta recta. Todo lo que da positivo asignamos al amarillo y lo negativo al verde. (*Heaviside (or step) function*).

$$g(a) = \begin{cases} 1 & \text{if } a \geq 0 \\ 0 & \text{if } a < 0 \end{cases}$$

Donde $a = wx + b$ se dice que es el input de la función de activación.

Binary Classification (M-D input)

Todo esto se puede escalar a múltiples dimensiones con funciones que ponderen múltiples entradas con funciones que dividen en 2 el espacio M-dimensional.

$$\hat{y} = g(a) = g(\mathbf{w}^T \mathbf{x} + b) = g(w_1 x_1 + w_2 x_2 + \dots + w_M x_M + b)$$

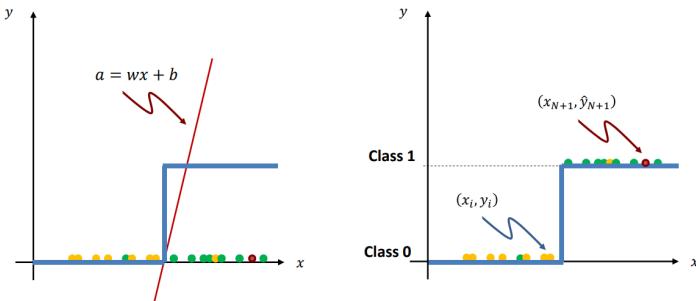


Fig. 1.1: Ejemplo clasificación 1-D

1.1.2 Perceptron

Algoritmo de aprendizaje supervisado capaz de generar un **criterio para la clasificación de observaciones**. El conjunto de datos de entrada (secuencia de datos temporales/píxeles, etc) se **ponderan linealmente** y se suman además de un sesgo. Entran en una **función de activación** no lineal y obtenemos una aproximación no lineal.

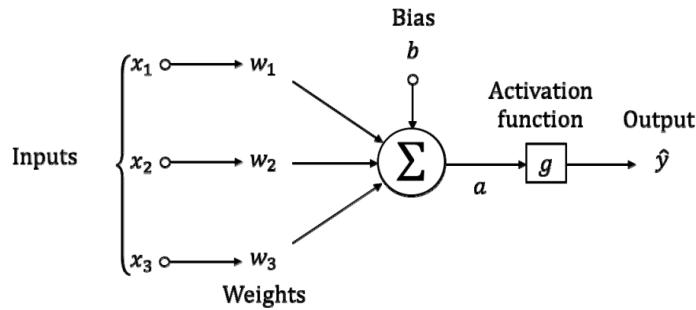


Fig. 1.2: Esquema de Perceptron

$$\hat{y} = g(a) = g(\mathbf{w}^T \mathbf{x} + b) = g(w_1 x_1 + w_2 x_2 + \dots + w_M x_M + b)$$

En este sistema tenemos variables a optimizar siguiendo alguna métrica que compara el resultado de la estimación con el valor real.

Single Neuron

Tenemos los **pesos** y el sesgo que se le suma a la combinación lineal. Tenemos también una función de activación que normalmente es no lineal, esto nos permite **explorar mucho mas el conjunto de posibilidades** para aproximar los datos.

- **Peso w** y **sesgo b** son parámetros que definen el comportamiento de la neurona.
Son estimados durante el *training*.
- **Input data x** se combinan linealmente con los pesos y el sesgo.
- **Activation function g(.)** introduce un comportamiento no lineal.

Perceptron as a computational graph

Para poder estudiar estos sistemas utilizaremos un **grafo computacional**, en un diagrama de este tipo tenemos representadas los datos y las operaciones que hacemos hasta

obtener el resultado que queremos. Se usa para saber como actualizar los parámetros para aproximar correctamente los valores deseados.

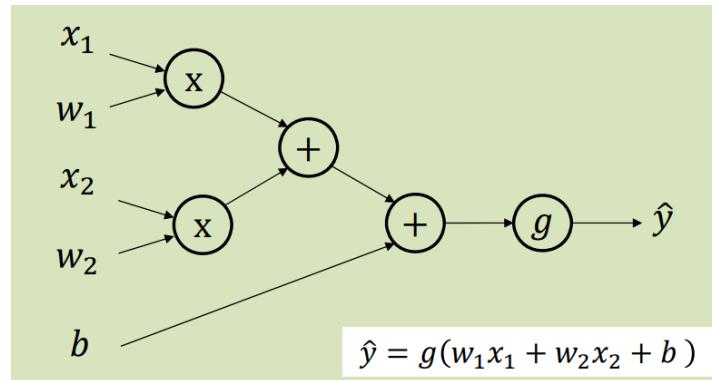


Fig. 1.3: Ejemplo de grafo computacional de un perceptrón con 2 inputs ($M = 2$)

Activation Function

La frontera de decisión que se obtiene con la función escalón es poco flexible y obvia muchos errores alrededor de la frontera de separación. Además de que su derivada conlleva a la **función de Dirac**.

Intentamos hacer una decisión mas suave, en vez de la función escalón usamos la función **Sigmoid** $\sigma(x)$. Si esta muy alejado del origen no tenemos dudas sobre la clasificación, si esta por el medio es donde tenemos mas dudas y la probabilidad de error es mas grande. Cuando x toma valores negativos muy grandes la función tiende a 0, y cuando toma valores positivos muy grandes tiende a 1.

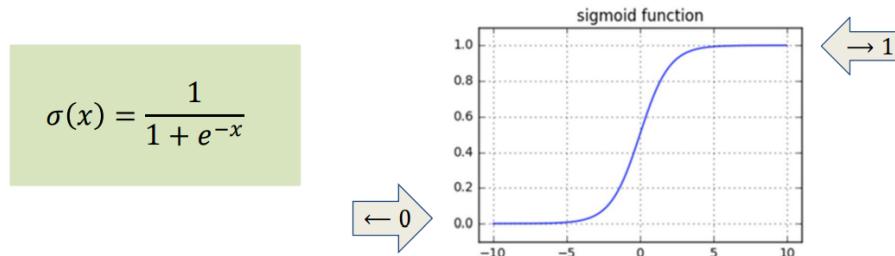


Fig. 1.4: Definición y plot de la función sigmoid

Con esta suavidad se obtiene retrasar la toma de decisión sobre la clasificación de una observación mientras se continua avanzando a través de la red y obteniendo más información.

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

$$\frac{\partial \sigma(x)}{\partial x} = (1 - \sigma(x))\sigma(x)$$

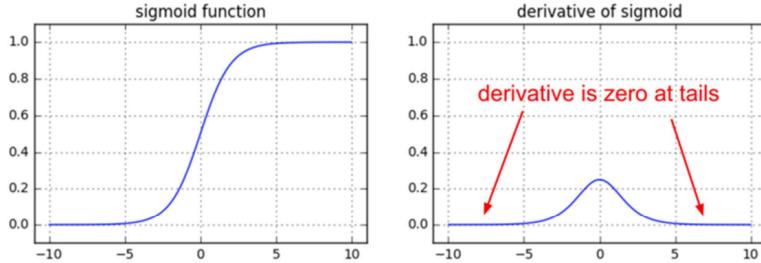


Fig. 1.5: Derivada de la función sigmoide

Igualmente, se obtiene una derivada que construye una zona desincerteza sobre la clasificación, diferenciando bien los valores que son claramente clasificables en los extremos de la función, con valores casi nulos. Así mismo los parámetros de las neuronas pueden ser estimados mediante la optimización de esta nueva función de activación.

Logistic Regression

Podemos relacionar la salida de la función de activación **Sigmoid** como una probabilidad de que la observación pertenezca a alguna de las dos clases. Si estamos con valores muy elevados de x es muy probable que sea de la clase 1 y al revés.

$$\sigma(x) \rightarrow p(\hat{y}_i = 1 | x_i; \theta)$$

También hay que tener en cuenta que se dibuja en 1-D pero esta idea se puede extrapolar en múltiples dimensiones.

$$\sigma(x) = \frac{1}{1 + e^{-a}}, \quad a = \mathbf{w}^T \mathbf{x} + b$$

Other activation functions

Lo que le pedimos a la no linealidad es que sea suave, continua y diferenciable. Otras funciones no lineales que se usan normalmente son la **tangente hiperbólica**. La derivada se parece mucho a la Sigmoide así como la función. Pero la Sigmoide da valores entre 0 y 1 que se pueden asociar a probabilidad y esta no (tienen otras funciones). Hoy en día una de las que más se usa es la **ReLU(\mathbf{x})**

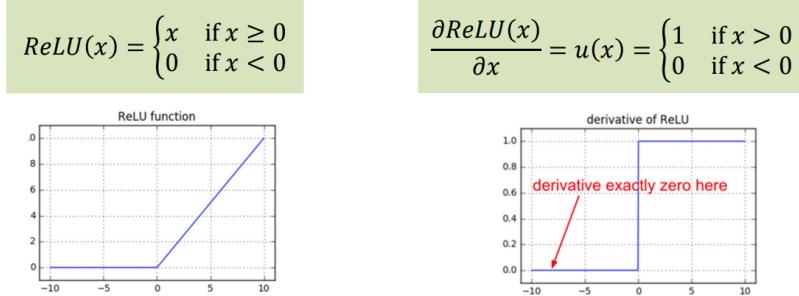


Fig. 1.6: Definición de la función de activación ReLU

Lo que nos ofrece esta función es una **mejor propagación del gradiente**, una activación esparsa (en una inicialización *random* solo un 50% de las unidades son activadas) y una **computación eficiente** ya que incluye solo una comparación.

1.1.3 Multi-Layer Perceptron (MLP)

Motivation

Perceptron sirve para problemas sencillos como funciones del tipo AND/OR, pero no puede implementar **X-OR** ya que es un problema no linealmente separable. Lo que se hizo es juntar múltiples perceptrones para **combinar y concatenar** los outputs. Los pesos y los sesgos ahora están distinguidos por la capa en la que se encuentran.

$$w_{ij}^l, b_i^l, \quad \begin{cases} l : & \text{capa de la red en la que está el perceptrón.} \\ i : & \text{posición del perceptrón dentro de la capa correspondiente.} \\ j : & \text{índice de la observación a la que corresponde el peso.} \end{cases}$$

Neural Network

Si hemos decidido concatenar perceptrones lo podemos hacer con la cantidad que queremos y aparecen redes mucho más complicadas con combinaciones. Las **redes neuronales** son una composición de neuronas simples (perceptrones).

Tenemos una capa \mathbf{h}^0 que son las entradas, una capa final que es la salida y las capas intermedias (*capas ocultas*) h^l que procesan los datos de entrada (combinaciones lineales). Cuando las neuronas usan los outputs de todas las neuronas de la capa anterior, se denomina una **red completamente conectada**.

En forma matricial,

$$\mathbf{h}^{(l)} = g(\mathbf{W}^{(l)} \mathbf{h}^{(l-1)} + \mathbf{b}^{(l)})$$

La matriz $W^{(l)}$ contiene por filas, los pesos de todos los perceptrones de la capa l . La salida es una función muy complicada de unos parámetros que vamos optimizando.

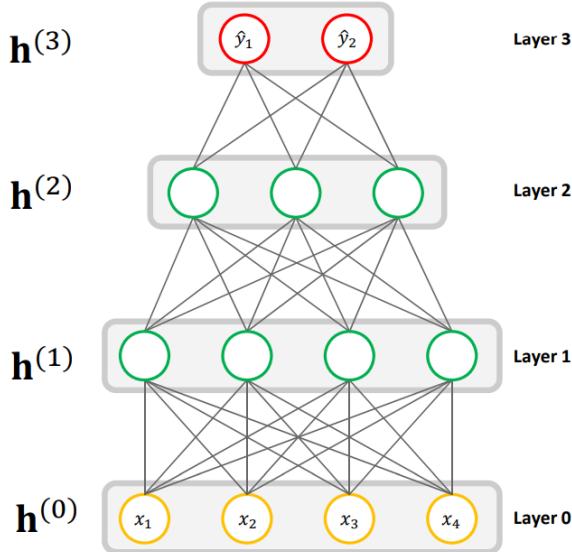


Fig. 1.7: Neural Network example

Neural Network dimensions

Podemos incrementar y buscar maneras de representar nuestros datos, pero lo que intentamos finalmente es intentar disminuir el numero de neuronas e internar representar el resultado en dimensiones menores.

- **Profundidad:** numero de capas hacia los nodos de salida exceptuando la primera capa ¹. La primera solo se considera para la entrada de datos.
- **Amplitud:** se trata de la capa mas amplia, la capa con mas neuronas.
- **Output:** normalmente se trata de un vector $\hat{\mathbf{y}} = f_{\theta}(\mathbf{x})$

Multiple class classification: Softmax

Podemos coger y hacer una clasificación con múltiples clases teniendo una salida separada para cada una de los datos de entrada. Lo que se propuso es combinar todas las salidas y combinarlas para tener para cada uno de esos valores una **probabilidad**. La salida de la ultima capa es un conjunto de valores h_i^L .

$$S(h_i^L) = \frac{e^{(h_i^L)}}{\sum_{i=1}^M e^{(h_i^L)}}$$

Se usa una exponencial para empujar a los valores positivos a que tengan mayor importancia. Tenemos una función que genera probabilidades pero hace que tengamos más en cuenta los valores más positivos. Los valores negativos tienden a 0. Se puede hacer el Softmax directamente sin aplicar la función Sigmoid previamente.

SoftMax function with temperature

La temperatura sirve para controlar la suavidad de la distribución de probabilidad final. Si la temperatura es grande los valores son todos mas cercanos a 0 y por lo tanto tendrán una "probabilidad" parecida. En cambio si la temperatura es muy baja estamos priorizando otro tipo de configuración y estamos exagerando aun mas los valores mas

¹Una capa se define como una columna de nodos dentro de una red neuronal

grandes. El valor correcto de la temperatura no existe, hay que jugar con la T ya que es un hyperparametro.

$$S(h_i^L) = \frac{e^{(h_i^L)/T}}{\sum_{i=1}^M e^{(h_i^L)/T}}$$

1.2 Backpropagation

Lo que vimos es que uniendo perceptrones podemos construir arquitecturas que resuelven problemas muy complejos. Pero cuando la complejidad crece mucho también lo hacen los parámetros.

De manera automática y adaptativa conseguir un conjunto de parámetros de la red neuronal. Los parámetros se acomodan para representar de la mejor manera (mejor calidad posible). Es decir, minimizando una **función de perdida**. Dada la complejidad de las redes neuronales, la optimización tiene que ser mejor que un simple **Gradient Descent**.

Todo el proceso de optimización tiene los siguientes pasos:

1. **Paso hacia delante:** Cogemos la red que está inicializada con unos parámetros. Lo que hacemos es alimentar la red con los valores de la entrada. Vamos propagando los valores hasta llegar a ver para este conjunto de parámetros como se parece la salida a los valores de referencia. Esto se hace utilizando una cierta función de pérdida.
2. **Paso hacia atrás:** Usando la **regla de la cadena** para derivadas, se calcula la **función de perdida** con respecto a los parámetros, mientras se propaga la derivación hacia los nodos anteriores. Con esto se puede decidir que cambios hay que aplicar para minimizar el valor de la función de perdida.
3. **Paso de actualización:** Actualizar los valores de los parámetros mediante **Gradient descent**.

1.2.1 Forward Pass

La estimación que estamos generando (\hat{y}) viene dada por la función

$$\hat{y} = wx + b$$

Estamos intentando optimizar estos parámetros respecto a la función de error (*squared error*). Comparamos la aproximación con el valor real que debería ser.

$$\mathcal{L}_{SE} = (y - \hat{y})^2$$

Si queremos reducir el valor de la función de pérdida como debes cambiar los valores del peso y del sesgo? Esta información nos la da el **gradiente** $\frac{\partial \mathcal{L}_{SE}}{\partial \theta}$. Mediante la variación de los parámetros $\theta = [w, b]$ de forma iterativa se escoge de una forma conveniente los valores que conlleven a que la función de perdida sea 0.

1.2.2 Gradients from composition: Chain rule

Hemos visto que debemos calcular la derivada parcial para ver como afecta el cambio de w y b a la función de pérdida \mathcal{L}_{SE} .

$$h(x) = f(g(x)) \rightarrow h'(x) = f'(g(x))g'(x)$$

Sin embargo, las funciones que estudiamos presentan relaciones complejas entre las variables de entrada y salida. Podemos sino imaginar que $f(x)$ se puede describir como una composición de funciones más sencillas, de manera que permiten aplicar más fácilmente la regla de la cadena.

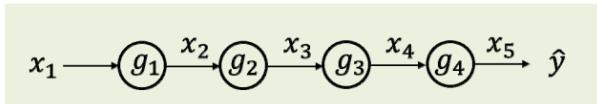


Fig. 1.8: Input/Output relations

$$\hat{y} = x_5 = f(x) = g_4(g_3(g_2(g_1(x_1)))), \quad x_{i+1} = g(x_i)$$

Se tiene que ir iterando el coeficiente de la cadena de derivación. *Nota: $g(\cdot)$ no representa una función no-lineal, sino que es cualquier tipo de función.*

$$\frac{\partial \hat{y}}{\partial x_1} = \frac{\partial \hat{y}}{\partial x_5} \cdot \frac{\partial x_5}{\partial x_4} \cdot \frac{\partial x_4}{\partial x_3} \cdot \frac{\partial x_3}{\partial x_2} \cdot \frac{\partial x_2}{\partial x_1}$$

Mediante esta **backpropagation**, se va obteniendo como se relaciona x_k con \hat{y} y asimismo con $g(x_k)$.

Propagation of the derivate

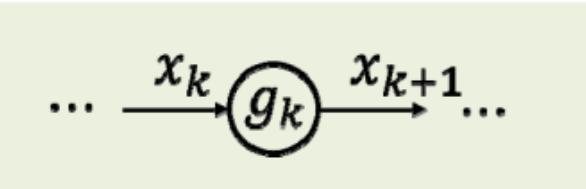


Fig. 1.9: Node Input/Output

El procedimiento para **propagar la derivada** a la variable x_k y obtener su influencia en el output \hat{y} es el siguiente:

1. Calcular la derivada de la función usada en el nodo k . Este cálculo se realiza en el forward pass.

$$g'_k(\cdot) = \frac{\partial x_{k+1}}{\partial x_k}$$

2. Evaluar la derivada en el punto actual c .

$$g'_k(c) = \left. \frac{\partial x_{k+1}}{\partial x_k} \right|_c$$

3. Multiplicar $g'_k(c)$ por el gradiente propagado.

$$\left. \frac{\partial \hat{y}}{\partial x_k} \right|_{x_k=c} = g'_k(c) \cdot \left. \frac{\partial \hat{y}}{\partial x_{k+1}} \right|_{x_{k+1}=g_k(c)}$$

Durante la navegación de la red, en cada neurona se puede guardar las derivadas parciales de las entradas y evaluarlas en los puntos correspondientes con los que se trabaje en la backpropagation.

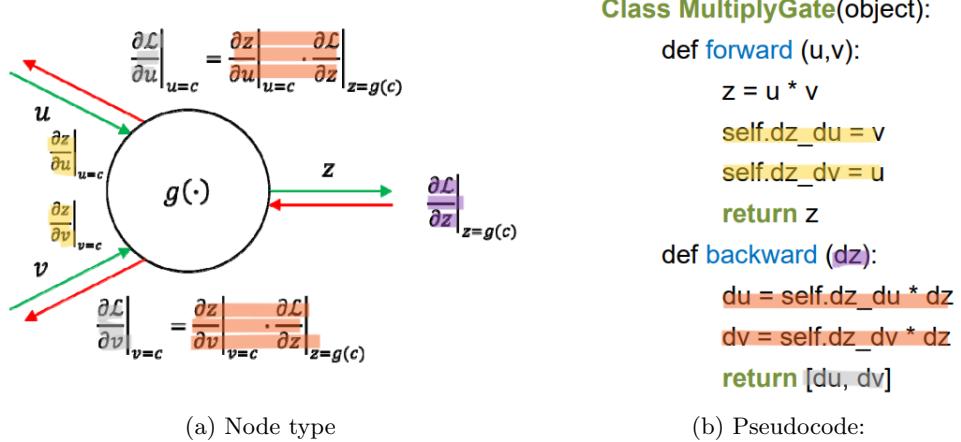


Fig. 1.10

Cuando se propaga sobre una red, se guarda toda la información de las derivadas en memoria. Hay un límite de datos que se pueden guardar, por lo que se tiene que optimizar para aprovechar la limitación de memoria. En la práctica no se entrena el dataset entero, sino que se van usando *minibatches* (particiones de los datos enteros) para que sea realizable.

1.2.3 Backward pass

Al concluir con la propagación hacia atrás se concluye con los gradientes de los parámetros. Un valor positivo del gradiente significa que si se aumenta el valor de aquel parámetro, el valor de la función de pérdida disminuirá y viceversa para un gradiente negativo.

Some considerations

Un conjunto de nodos que se pueden representar en un único nodo con derivada sencilla, se puede implementar un único nodo (*nodo sigmoid p.e.*). Se hace una reducción de nodos y derivadas.

Node types

- **Addition node:** gradientes locales unitarios con lo que el gradiente de la backpropagation se distribuye a las diferentes ramas con el mismo valor.
- **Product node:** En el producto de dos términos, el gradiente de uno es el término del otro, con lo que se puede entender como un **switcher**.
- **Max node:** Como solo se selecciona un input a través de la función max, el nodo actúa como **router**, pues solo propaga el gradiente de una rama seleccionada.

1.2.4 Gradient Descent

Los parámetros de las capas ocultas son muy importantes en la backpropagation, que dan interpretabilidad de la red. Podemos entender en qué se enfoca la red para tomar una decisión, así podemos determinar partes de la red y de sus parámetros.

$$\theta^{k+1} = \theta^k - \alpha \nabla_{\theta} \mathcal{L}_{SE}$$

De uso para la actualización de los parámetros y la reducción de la función de pérdida.

Con una única observación como referencia de entrenamiento, se hace **overfitting** de la red neuronal. Se tienen que facilitar muchos más observaciones etiquetadas.

Normalmente se subdivide el conjunto de datos en N subconjuntos para entrenar la red neuronal con los pares de observaciones. A su vez que se guardan en memoria todas las derivadas y parámetros intermedios, lo que limita el tamaño de los subconjuntos a las limitaciones físicas de almacenamiento.

En el paso backpropagation, se computa la media de los gradientes de los parámetros.

$$\boxed{\frac{\partial \hat{y}}{\partial x_k} = \frac{1}{N} \sum_{i=1}^N g'_k(c_i) \cdot \frac{\partial \hat{y}}{\partial x_{k+1}}}$$

1.3 Losses

La función de perdidas \mathcal{L} la evaluamos en un cierto momento con los parámetros que tenemos. Miramos el resultado que da cuando verificamos toda la red con nuestros datos etiquetados.

$$\mathcal{L} = \text{dist}(f_\theta(\mathbf{x}), y)$$

Note: *Hay veces que se dice función de coste, objetivo o error.*

Queremos conseguir que la función de perdida se minimice. Este concepto no deja de ser el de **minimizar la distancia** entre el procesado de las observaciones con unos ciertos parámetros y el *ground truth* y_i (el valor que queremos que devuelva nuestra red).

1.3.1 Loss Function

La función de perdida sirve para guiar a nuestro entrenamiento, en el sentido que mide la calidad de la red. Normalmente la función de pérdida real no es derivable, por lo que se propone una más sencilla (**surrogate loss**) para optimizar la red y optimizar los parámetros.

Optimizaremos parámetros en bloque. No haremos solo un *gradient descent*, haremos cosas mas desarrolladas pero son técnicas dentro de esta familia.

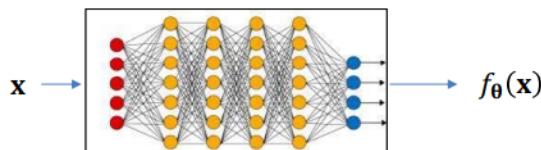


Fig. 1.11: 242 parámetros. $(7 \cdot 6 + 7 \cdot 8 + 7 \cdot 8 + 7 \cdot 8 + 4 \cdot 8)$

La función de pérdida se usa para guiar el proceso de entrenamiento. A menudo, minimizar la función real de error es inviable. La pérdida real puede involucrar cálculos complicados o costosos computacionalmente, o que no se conoce una función de pérdida adecuada para el problema en cuestión. En tales casos, se elige una pérdida sustituta que sea más fácil de calcular o que se adapte mejor a las características del problema, y se minimiza esa pérdida sustituta en su lugar.

1.3.2 Properties of the loss function

Nos gustaría que la función de pérdida sea **suave**, **convexa**, etc. Normalmente no es el caso (porque depende de muchos parámetros) y eso incrementa la complejidad (**funciones multi-modales**). Hay arquitecturas que nos permiten conseguir funciones de pérdida con mejores condiciones (más suaves).

Lo que nos gustaría es que esta función tenga un decrecimiento hasta 0. Esto combinado con la técnica de *backpropagation*, nos permitiría conseguir este decrecimiento a medida que se va iterando.

Si hemos generado una cierta salida $f(x)$ lo que nos gustaría es que una pequeña variación en esta salida nos diera una pequeña variación en la función de perdida.

$$\hat{y} + \epsilon_1 \rightarrow \mathcal{L}(\hat{y}, y) + \epsilon_2$$

1.3.3 Losses with multiples inputs

Durante el entrenamiento, lo ideal es poder usar todo el conjunto de datos de entrada, pero por motivos de almacenaje (debido a los gradientes en el forward pass) se selecciona un **minibatch** de datos para trabajar. Lo que hacemos es procesar todos estos datos y ponderamos la pérdida que nos da cada una de los datos anotados. La **función total de pérdida** (\mathcal{L}) es también el **rriesgo empírico**.

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^N \mathcal{L}_i = \frac{1}{N} \sum_{i=1}^N \text{dist}(f_\theta(x_i), y_i)$$

1.3.4 Loss function in regression problems L1 vs L2

Para la predicción de variables continuas, numéricas.

L1, $\mathcal{L}_i = |y_i - f_\theta(x_i)|$

- No es derivable.
- Más fácil de calcular.
- Más penalización a errores < 1 (con respecto a L2)
- Menos penalización a errores > 1 (con respecto a L2)
- Menos sensible a *outliers*, no son tan importantes.
- Tendencia a no converger porque genera saltos (valores del gradiente) mas grandes al no ser tan suave como en L2.

L2, $\mathcal{L}_i = (y_i - f_\theta(x_i))^2$

- Más difícil de calcular.
- Más sensible a *outliers*.

Cuando se procesan valores pequeños de pérdida (cercaos a cero), la función de pérdida L1 devuelve valores de gradiente grandes, lo que conduce a iteraciones ineficientes para encontrar el mínimo. Por otro lado, la función de pérdida L2 devuelve gradientes pequeños cuando está cerca del mínimo.

Es importante notar que, cuando se considera un conjunto de pares de entrenamiento, la función de pérdida L1 se convierte en el Error Absoluto Medio (MAE) y la función de pérdida L2 se convierte en el Error Cuadrático Medio (MSE).

1.3.5 Huber Loss

Vamos a combinarlas de manera que la forma que conecten. Cerca del 0 queremos que se comporte como una función cuadrática y lejos de $|1|$ queremos que se comporte como una función lineal.

- Es **menos sensible a los valores atípicos** en los datos que la pérdida MSE.
- Es **diferenciable** en 0 (a diferencia del MAE).
- Básicamente, **representa el error absoluto**, que se vuelve cuadrático cuando el error es pequeño.
- Se define **en función de δ** , que se convierte en un **hiper parámetro** (generalmente $\delta = 1$).

Múltiples posibilidades de encajar para diferentes valores de δ . Asegura conexión entre las dos bandas. δ se puede definir a priori a mano, pero es difícilmente derivable. Este parámetro pasa a ser tratado como **hiper parámetro** (probar una serie de valores).

$$L_\delta(y, f(x)) = \begin{cases} \frac{1}{2}(y - f(x))^2, & \text{if } |y - f(x)| < \delta \\ \delta(|y - f(x)| - \frac{1}{2}\delta), & \text{otherwise} \end{cases}$$

1.3.6 Losses in Classification Problems

En problemas de clasificación, la red predice variables categóricas.

- Clasificación binaria
- Clasificación de una sola clase
- Clasificación de múltiples clases

Binary Classification

Una primera aproximación puede ser coger una de las funciones de perdidas para regresión y adaptarla para clasificación.

Hinge or margin-based loss

Sin embargo, si por ejemplo si consideramos dos clases como $y_i = \{-1, 1\}$. Si obtenemos errores de clasificación fuera de este rango, estas observaciones son sin duda correspondientes a alguna de las dos clases. Entonces para estas clasificaciones que sin lugar a duda pertenecen a una clase, no se debería contar su penalización. De aquí la idea principal de **Hinge**. No tiene en cuenta términos de probabilidades.

- Se trata de aquellas **predicciones que son correctas y que creemos en ellas** esas **no** tienen que ser **penalizadas**.
- Las **predicciones incorrectas** se tienen que **penalizar**.
- Las que son **correctas pero no estamos seguros** (entre -1 y 1) también se tienen que **penalizar** (aunque en menor grado).

$$\mathcal{L}_i = \max(0, m - y_i f_\theta(x_i)), \quad y_i = \pm 1$$

La m es el concepto de seguridad, se llama parámetro de margen (normalmente $m = 1$).

Case $y_i = 1$ and $m = 1$

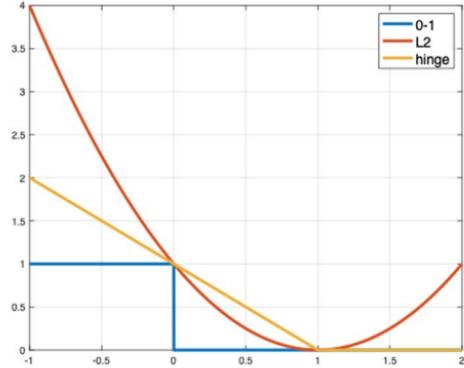


Fig. 1.12: Loss Comparison

Para $y_i = 1$, valores de \mathcal{L}_i mayores que 1 no se tendrán en cuenta porque estamos seguros de que han sido clasificados a la clase correcta. Mientras que a medida que no se este 100% seguro o se haya clasificado la observación como incorrecta, se asignará una penalización en función de que tan inseguro/erroneo se haya clasificado.

Cross-entropy loss

Si queremos tener en cuenta probabilidades lo que tenemos que hacer es comparar p.e. las estadísticas de primer orden (de la pdf de los outputs y de las referencias). Entran los datos y queremos generar no solo un valor sino una pdf. Que puede ser p.e. si queremos clasificar entre 10 clases cual es la probabilidad de que sea cualquiera de esas clases.

Para poder usar estas estadísticas tenemos que tener una estimación del histograma del output (**SoftMax**). Necesitamos una definición del modelo el cual queremos parecer-nos (**distribución de referencia como One-hot encoding**). Y mediante un calculo de distancia (**Divergence, Cross-Entropy**) comparar estas dos pdf.

Ambas métricas (Cross Entropy y Divergence) unidas están relacionadas. Optimizar ambos términos es lo mismo.

$$H(p, q) = - \sum_{x \in \mathcal{X}} p(x) \ln q(x)$$

$$D(p||q) = \sum_{x \in \mathcal{X}} \ln \left(\frac{p(x)}{q(x)} \right) = H(p, q) - H(p)$$

Donde consideraremos que q es la distribución output y p es la distribución del modelo.

Single-label multiclass classification

Tanto **Hinge**, como **Cross Entropy** se pueden aplicar para Binary classification como para Single-label multiclass classification.

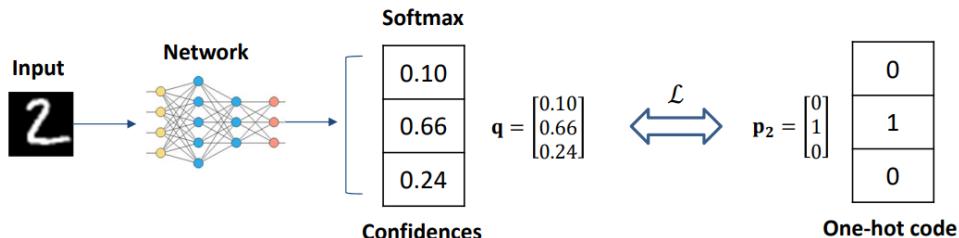


Fig. 1.13: Cross-entropy

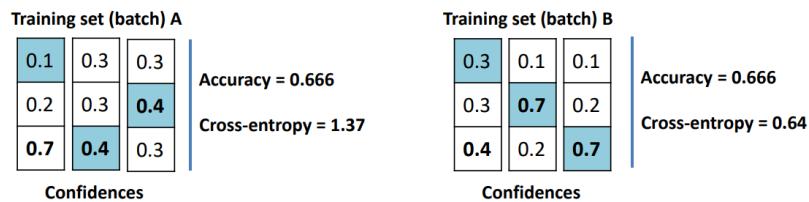
Para el caso de **Cross Entropy**: La distribución del output corresponde a la aplicación del SoftMax, mientras que la distribución del modelo de referencia se realiza con One-hot encoding, donde la longitud del vector es el número de clases diferentes y únicamente es diferente de 0 en la posición de la *single label*. Lo que hace que en la función de Cross Entropy, solo un término sea diferente de 0.

$$\mathcal{L}_i = - \sum_k y_{i,k} \ln S(a_{i,k}^L) = - \ln S(a_{i,k_0}^L)$$

Cross Entropy vs Accuracy

La Cross Entropy (\mathcal{L}) nos puede dar más información con respecto al rendimiento o desempeño de la red en comparación con la Accuracy. Esto se debe a que, mediante la Cross Entropy se tiene en cuenta la diferencia/comparación entre el valor de la distribución del output en comparación con el de la distribución del modelo referencia. En cambio, el Accuracy solo se enfoca en el número de aciertos con respecto al total, sin tener en cuenta por cuanto ha sido el acierto/error con respecto a las otras clases.

Además, la Cross Entropy proporciona una medida más sensible y detallada del rendimiento del modelo en problemas de clasificación con múltiples clases. Al considerar la discrepancia entre las distribuciones de probabilidad de las clases reales y las predichas, es capaz de captar sutilezas en la calidad de las predicciones. Por ejemplo, puede indicar si el modelo tiende a estar seguro en sus predicciones, incluso cuando está equivocado, o si está indeciso y distribuye probabilidades similares a varias clases.



Weighted cross entropy loss

Puede ser que haya una clase predominante y que por lo tanto la red se especialice solo en esa clase (al haber desbalance de clases). En la suposición actual, las perdidas se ponderan por $1/N$ (consideramos todas igual de importantes). En cambio, ahora lo que hacemos es ponderarlo por el inverso de probabilidad que salga esa clase a_k . Si clase sale poco esa clase tendrá un peso grande.

$$\mathcal{L}_i = - \sum_k \alpha_k y_{i,k} \ln S(a_{i,k}^L) = -\alpha_{k_0} \ln S(a_{i,k_0}^L)$$

Focal Loss

El concepto de Focal Loss se utiliza para abordar situaciones en las que tenemos pocas representaciones de un objeto en comparación con otros que tienen muchas representaciones, como ocurre en el contexto de la distinción entre primer plano (foreground) y fondo (background).

La Focal Loss introduce un factor $(1 - S(a_{i,k}))$ que está relacionado con la probabilidad estimada de la entrada. Aquí, $S(a_{i,k})$ representa la probabilidad estimada de que el parche x_i pertenezca a la clase k .

En esta fórmula, es importante notar que las entradas que son más frecuentes, como las correspondientes al fondo, tienen una probabilidad estimada más alta de aparecer, lo que resulta en una contribución menor a la pérdida. Por otro lado, las entradas menos frecuentes, como las relacionadas con el primer plano, tienen una probabilidad estimada más baja y, por lo tanto, contribuyen más significativamente a la pérdida.

$$\mathcal{L}_i = - \sum_k (1 - S(a_{i,k}))^\gamma y_{i,k} \ln S(a_{i,k})$$

Donde \mathcal{L}_i es la pérdida para la muestra i , γ es un hiperparámetro que ajusta el grado de enfoque en ejemplos difíciles, $y_{i,k}$ es la etiqueta real de la clase k para la muestra i , y $S(a_{i,k})$ es la probabilidad estimada para la clase k en la muestra i .

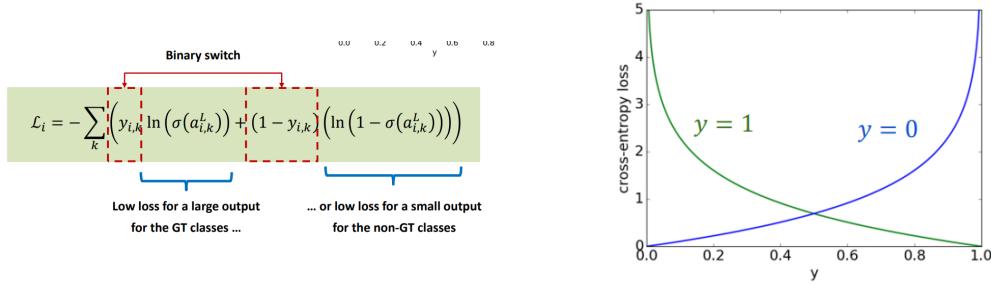
1.3.7 Multi-label

El enfoque multi-label se utiliza cuando se deben asignar múltiples etiquetas a una muestra, lo que es común en el contexto de imágenes, donde una imagen puede contener varios objetos o características de interés.

A diferencia del enfoque de clasificación tradicional, en el que se busca asignar una sola clase a una muestra (clasificación única), en problemas multi-label se aceptan múltiples clases para una sola muestra. Esto significa que, por ejemplo, una imagen puede estar etiquetada con "perro" y "gato" al mismo tiempo.

En el caso de problemas multi-label, no utilizamos una función Softmax como en la clasificación única. En lugar de eso, el objetivo es determinar si existe una alta probabilidad de que una muestra tenga una determinada etiqueta, y es posible que varias etiquetas sean aplicables simultáneamente. Para lograr esto, aseguramos que las salidas para cada clase estén en un rango entre 0 y 1.

Esto se logra mediante la aplicación de una función Sigmoid a cada salida $a_{i,k}$ de manera independiente. La función Sigmoid asigna valores en el intervalo [0, 1] a cada clase, lo que permite representar la probabilidad de que una etiqueta particular esté presente en la muestra. En otras palabras, cada valor de salida se interpreta como la probabilidad de que la etiqueta correspondiente esté presente en la muestra, y múltiples etiquetas pueden tener valores cercanos a 1 simultáneamente.



Nota: No se usa la función tanh() porque queremos representar una "probabilidad" y tanh() va de -1 a 1.

1.3.8 Aprendizaje de Métricas

El aprendizaje de métricas se enfoca en la reducción del espacio dimensional. En este contexto, partimos de un conjunto de datos representado en un espacio N -dimensional, y nuestro objetivo es reducir este espacio a uno más pequeño, donde la información retenida sea más o menos equivalente.

El proceso de aprendizaje de métricas se basa en encontrar una representación más compacta de los datos que conserve las relaciones de similitud y distancia entre las muestras originales, lo que facilita tareas posteriores de análisis y procesamiento.

Red Siamesa

Una **red siamesa** se refiere a cualquier arquitectura de modelo que incluye al menos dos redes paralelas e idénticas, lo que significa que comparten los mismos parámetros. Cada una de estas redes forma parte de una red siamesa, que a menudo se utiliza en tareas de aprendizaje profundo, como el procesamiento de lenguaje natural y la visión por computadora, con el propósito de generar un **embedding**.

Embedding es una representación en un espacio N -dimensional de la entrada a una M -dimensión de salida ($N > M$). Se espera que esta representación capture de manera efectiva las diferencias entre diferentes entradas al mapearlas a salidas distintas en términos de distancia.

En el proceso de entrenamiento, alimentamos la red con parejas de entrada "buenas" x_p y parejas de entrada "malas" x_n . En este contexto, siempre tenemos un elemento llamado **anchor** (x_a) que se utiliza para comparar con el elemento positivo (x_p) y el elemento negativo (x_n). El objetivo es que las parejas positivas tengan una distancia mínima entre sí, mientras que las distancias entre las parejas negativas no son de importancia, ya que no aportan información útil. Sin embargo, si la pareja negativa (x_a, x_n) se encuentra a una distancia menor de la permitida según un margen m , es necesario penalizar esta situación para ajustar los parámetros de manera adecuada.

Contrastive Loss

Este escenario se alinea con el modelo de pérdida **Hinge**, ya que se requiere aplicar una penalización cuando la pareja negativa está a una distancia menor de la permitida según el margen m , al igual que se debe considerar la distancia entre la pareja positiva.

$$\mathcal{L} = \begin{cases} \mathcal{L}_p = d^2(f_{\theta}(\mathbf{x}_a), f_{\theta}(\mathbf{x}_p)) & \text{if Positive pair} \\ \mathcal{L}_n = \max^2(0, m - d(f_{\theta}(\mathbf{x}_a), f_{\theta}(\mathbf{x}_n))) & \text{if Negative pair} \end{cases}$$

Cuando es una pareja positiva $y = 1$ penalizamos las muestras entre las muestras positivas y cuando es negativa $y = 0$ penalizamos las muestras negativas que estén más cerca que un umbral m .

$$\mathcal{L} = y \cdot d^2(f_{\theta}(\mathbf{x}_a), f_{\theta}(\mathbf{x}_p)) + (1 - y) \cdot \max^2(0, m - d(f_{\theta}(\mathbf{x}_a), f_{\theta}(\mathbf{x}_n)))$$

Triplet ranking loss

Los inputs también se pueden considerar como tripletes, lo cual mejora los resultados la función de perdida. Estos contribuirán al entrenamiento en el caso de que:

- Si el **embedding** de la observación negativa está más cerca al anchor que la observación positiva:

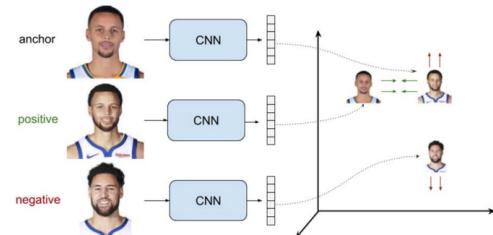
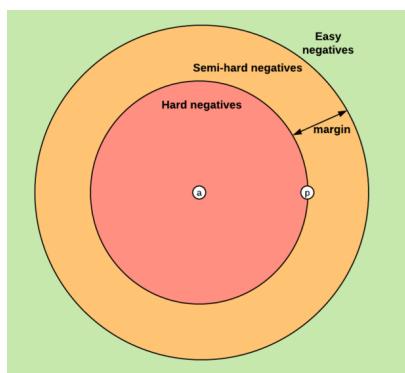
$$d^2(f_{\theta}(x_a), f_{\theta}(x_p)) - d^2(f_{\theta}(x_a), f_{\theta}(x_n)) > 0$$

- Las parejas relativas están más cerca que un cierto margen:

$$d^2(f_{\theta}(x_a), f_{\theta}(x_n)) - d^2(f_{\theta}(x_a), f_{\theta}(x_p)) < m$$

$$\mathcal{L}(\mathbf{x}_a, \mathbf{x}_p, \mathbf{x}_n) = \max(0, m + d^2(f_{\theta}(\mathbf{x}_a), f_{\theta}(\mathbf{x}_p)) - d^2(f_{\theta}(\mathbf{x}_a), f_{\theta}(\mathbf{x}_n)))$$

De manera que tenemos nuevamente tres escenarios:



- **Hard negatives:** Cuando la observación negativa está más cerca al anchor que la observación positiva. $d^2(f_{\theta}(x_a), f_{\theta}(x_p)) > d^2(f_{\theta}(x_a), f_{\theta}(x_n))$.

- **Easy negatives:** Cuando la observación negativa está más lejos de la observación positiva más un cierto margen m : $d^2(f_\theta(x_a), f_\theta(x_n)) > m + d^2(f_\theta(x_a), f_\theta(x_p))$. La pérdida es 0.
- **Semi-hard negatives:** Cuando la observación negativa está tan cerca del anchor como la observación positiva dentro de un cierto margen: $d^2(f_\theta(x_a), f_\theta(x_p)) < d^2(f_\theta(x_a), f_\theta(x_n)) < d^2(f_\theta(x_a), f_\theta(x_p)) + m$. La pérdida sigue siendo positiva para mejorar los parámetros.

Triplet mining

El número de tripletes (parejas) crece de manera cúbica (cuadrática). Existe la necesidad de desarrollar una estrategia para seleccionar ejemplos valiosos.

Como encontramos/definimos los tríos de entrenamiento? Si se seleccionan de manera aleatoria, es muy probable que la mayoría de tripes sean de tipo *easy negatives*.

A medida que avanza el entrenamiento, más parejas/triplets se vuelven fáciles (con pérdida igual a 0), lo que impide que la red aprenda de manera efectiva. El proceso de aprendizaje requiere ejemplos relevantes. Se han desarrollado estrategias para seleccionar ejemplos "semi-duros" o "duros", pues así la red tiene que trabajar duro para poder separarlos durante el aprendizaje y aumenta la eficiencia computacional del aprendizaje.

Estrategia fuera de línea:

- Calcular todos los embeddings en el conjunto de entrenamiento y seleccionar solo los tripletes semi-duros/duros.
- Esta estrategia tiene una alta complejidad computacional ya que implica encontrar negativos en todo el conjunto de entrenamiento.

Estrategia en línea

- Emplear diferentes estrategias para encontrar tripletes semi-duros/duros en cada lote (batch) de datos.
- Una solución posible para un conjunto de entradas (lote) con K clases diferentes y N ejemplos por clase es seleccionar, para cada ancla, el positivo más difícil (mayor distancia) y el negativo más difícil en el lote.
- Esto genera NK tripletes (los más difíciles del lote).

Ambas estrategias tienen como objetivo abordar el crecimiento exponencial de las parejas/triplets a medida que progresó el entrenamiento y garantizar que el proceso de aprendizaje se base en ejemplos relevantes.

1.4 Optimizers

1.4.1 Batch Gradient Descent

Se entrena la red con todos los datos (**batch**) de tamaño N y se obtienen las funciones de pérdida para cada paso de la observación del **batch**. Los gradientes se van almacenando en la propagación hacia adelante. **Se realiza una única actualización de los parámetros del modelo**

Con N visiones de diferentes se tiene una aproximación mejor a los *targets*. Se usa toda la información del **batch** en cada uno de los parámetros de la red. Una N más grande nos dará una mejor aproximación pero implica un coste superior e incluso una ejecución más lenta.

1.4.2 Stochastic Gradient Descent

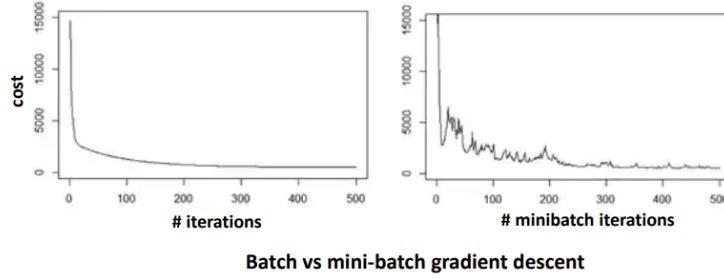
Stochastic Gradient Descent: En contraste, Stochastic Gradient Descent (**SGD**) opera de manera diferente. En SGD, se optimiza el modelo para cada punto de datos

individualmente. Se estima la pérdida con el único par de entrada para tomar un paso mediante el gradient descent. Por ejemplo, se toma una imagen, se realiza una propagación hacia adelante y se optimizan los parámetros, y luego se repite este proceso para todas las imágenes en el conjunto de datos.

SGD introduce una cantidad significativa de ruido en el proceso de optimización, lo que da como resultado trazos de actualizaciones de parámetros muy erráticos. Esto se debe a que la visión que tiene el algoritmo en todo momento es local, lo que puede llevar a una convergencia más lenta en comparación con Batch Gradient Descent. Sin embargo, tiene la ventaja de no requerir el almacenamiento en **memoria** de todo el conjunto de datos, lo que lo hace adecuado para procesar datos en línea (**online method**), donde los datos llegan uno por uno y los parámetros del modelo deben ajustarse a medida que llegan los datos.

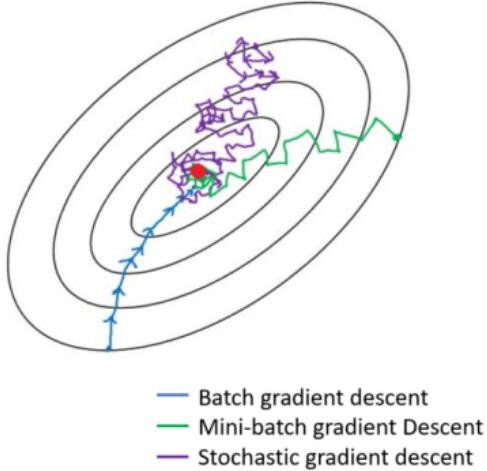
1.4.3 Mini-Batch Gradient Descent

Caso intermedio. No cogemos un bloque de datos tan grande como coger todos (N). Sino que se coge un conjunto de datos que nos quepa en memoria y que nos permite confiar en la robustez de las aproximaciones para avanzar con confianza. Normalmente el tamaño del **minibatch** es una potencia de 2.



En cada iteración, normalmente hay una cierta tendencia a bajar, pero hay oscilaciones como comportamiento parcial. Influye mucho como es la representación representación estadística del **minibatch** (si representa bien las estadísticas de todo el conjunto de datos). Intentamos trabajar con **minibatch** con medida que puedan caber en memoria sin problemas y sean representativos.

| Method | Accuracy | Time | Memory usage | Online learning |
|------------|----------|--------|--------------|-----------------|
| Batch | + | slow | high | no |
| Mini-batch | + | medium | medium | yes |
| SGD | - | fast | low | yes |



1.4.4 Potential Problems

Varios problemas en las funciones de perdida que son comunes en la practica. En el caso de el perceptron multilayers es con solo dos capas, un nodo por capa y solo un peso en cada nodo. Este esquema ya tiene una función de perdida tan compleja con puntos de sella, simetrías, zonas planas (acantilados).

| |
|--|
| $\hat{y} = f_{\theta}(x) = \tanh(w_2 \tanh(w_1 x))$ |
| $x \xrightarrow{w_1} \tanh(\cdot) \xrightarrow{w_2} \tanh(\cdot) \rightarrow \mathcal{L}_{L2} = (y - \hat{y})^2$ |

1.4.5 Local minima and saddle points

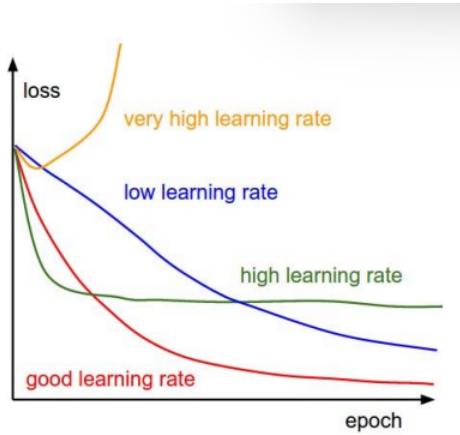
Hay muchas configuraciones que conllevan a diferentes mínimos locales. Lo que se tiene que intentar es que la red no tenga tendencia a soluciones problemáticas.

Nota: *No siempre que se añaden mas capas a la red, la función de perdida disminuye.*

Lo que mas aparece son puntos de ceja cuando las funciones de pérdida son de dimensiones altas. Necesitamos que la red tenga la capacidad de poder escapar de estos puntos pues el gradient descent se ve atraído a estos puntos. Para tenerlos en cuenta se pueden mirar a través de la Hessiana de la función de perdida.

$$\mathbf{H}_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}$$

La probabilidad de que salgan todos los autovalores positivos o negativos en matrices tan grandes como las que tratamos es muy difícil.



1.4.6 Learning Rate

Para diagnosticar los problemas de optimización, es útil mirar las curvas de entrenamiento. Normalmente se hace después de que acabe una **epoch**.

Learning Rate alto es el desajuste, nos quedamos saltando alrededor del mínimo. Lo que se habría de hacer es dar pasos mas seguros (learning rate menor).

Cuando tenemos oscilaciones, normalmente son causadas por la superficie de la función. Se tunea el learning rate mediante el **momentum**.

Una variación del learning rate a lo largo de las **epoch** para decrecer (por igual en todas las dimensiones) el learning rate. Hay varias estrategias que se han implementado.

1.4.7 Parameter Initialization

Muchos algoritmos están muy afectados por la inicialización de los parámetros.

- **Biases:** inicializamos todo a 0. Asumimos que no hay necesidad de sesgar las combinaciones lineales.
- **Weights:**
 - Si los inicializamos en 0 la red no prosperara. Se quedara estancado en un **saddle point**. Cuando los gradientes son pequeños y se propagan hacia atrás a través de múltiples capas, se produce una multiplicación de gradientes pequeños. Esto significa que a medida que los gradientes se retropropagan hacia las capas anteriores, su magnitud disminuye aún más. Como resultado, los gradientes pueden volverse extremadamente pequeños o incluso indistinguibles de cero.
 - Si ponemos todo el mismo valor, las posibles simetrías que haya en la red serán más fuertes, porque todas las neuronas actuarán de la misma manera. En el proceso de backpropagation, los gradientes para los pesos serán todos iguales, lo que impide que la red pueda aprender patrones distintivos o representaciones útiles en los datos.
 - Valores diferentes pero grandes, se nos desenfocará.
 - Valores aleatorios pequeños. De una distribución gaussiana o similar.
 - * **Xavier:** para activaciones tipo tanh. $w = \text{randn}(n) / \text{sqrt}(n)$
 - * **He:** para activaciones de tipo *ReLU*. $w = \text{randn}(n) \cdot \text{sqrt}(2/n)$

1.4.8 Algorithms

Usando el gradient descent hay varios problemas que se nos presentan. Uno de ellos es la dispersión de los VAP.

Dispersion of eigenvalues

Lo que puede suceder es que nuestra curva de función de pérdidas tenga autovalores de alta dispersión.

Adaptive learning rates: Para solventar la gran dispersión de VAP, se puede fijar el learning rates diferente para las diferentes dimensiones. Esto permite avanzar de forma segura.

Momentum: Mirar la traza de los gradientes previos (*la trayectoria*) y mirar la tendencia del recorrido y se puede calcular su *velocidad* v de la solución. Se hace Una combinación del gradiente y la velocidad que traía la solución ponderada. Se ve un mejor enfoque hacia el *punto óptimo*.

$$v^{k+1} = \lambda v^k - \alpha \nabla_{\theta} \mathcal{L}_{\theta} |_{\theta^k}$$

Esta nueva velocidad se incluye en la actualización de los parámetros. La idea detrás del momentum es que si el gradiente apunta en la misma dirección en múltiples iteraciones consecutivas, la velocidad de convergencia aumentará. También ayuda a superar obstáculos locales en la función de pérdida y acelerar el proceso de aprendizaje. Disfrutan del beneficio adicional de ser mucho más efectivos en casos en los que el problema de optimización está mal condicionado (es decir, en los que existen direcciones en las que el progreso es mucho más lento que en otras, asemejando un cañón estrecho). La definimos como un promedio con decrecimiento exponencial de los gradientes previos.

$$v^k = \alpha(-\lambda^{k-1} \nabla_{\theta} \mathcal{L}_0 - \lambda^{k-2} \nabla_{\theta} \mathcal{L}_1 - \dots - \lambda \nabla_{\theta} \mathcal{L}_{k-1} - \nabla_{\theta} \mathcal{L}_k) = -\alpha \sum_{\tau=k-1}^0 \lambda^{\tau} \nabla_{\theta} \mathcal{L}_{k-\tau-1}$$

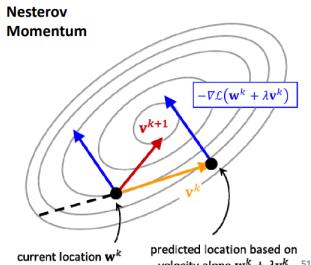
A medida que el numero de iteraciones aumenta, el hyper-parámetro $\lambda \in \{0, 1\}$ tiende a 0, eliminando los gradientes iniciales del promedio ponderado. Valores grandes de λ proporciona una media de gran rango, mientras que valores pequeños corresponde a una ligera corrección relativa al método del gradiente.

$$\theta^{k+1} = \theta^k + v^{k+1}$$

Incorporación de un nuevo parámetro λ como "pago" para mejorar la convergencia. Pasa a ser un hiper-parámetro para el sistema.

Nesterov momentum: Es una variante del momento, donde los gradientes se evalúan después de que se aplique la velocidad.

- Apply interim update $\tilde{\theta}^k = \theta^k + \lambda v^k$
- Compute gradient at interim point $\mathbf{g}^k = \frac{1}{N} \sum_{i=1}^N \nabla_{\theta} \mathcal{L}(\tilde{\theta}^k)$
- Compute velocity update $v^{k+1} = \lambda v^k - \alpha \mathbf{g}^k$
- Apply update $\theta^{k+1} = \theta^k + v^{k+1}$



El conjunto de parámetros de la solución actual la perturbo y me la llevo donde la velocidad media dice que debería estar (en la siguiente iteración) la solución. Es allí donde calculo el gradiente y actualizamos los parámetros.

- Facilita una convergencia más rápida dado a que anticipa la dirección óptima.

Adaptive Learning Rates

En lugar de usar momentum, que añade un hyperparámetro más, se puede adaptar el learning rate adecuandolo por separado a las diferentes dimensiones a lo largo del proceso de aprendizaje.

AdaGrad: Adaptación del gradiente basado en cada uno de los valores de los parámetros en base a la información histórica de los gradientes. Allí donde el gradiente sea mas grande avanzaremos de forma más despacio. Esto ayuda al algoritmo a adaptarse a las características individuales del parámetro.

La actualización se lleva a cabo guardando la suma del cuadrado de los gradientes en la matriz diagonal G^k y escalamos los gradientes con respecto a su inverso.

$$g^k = \frac{1}{M} \sum_{i=1}^M \nabla_{\theta} \mathcal{L}_i \quad G^k = \sum_{\tau=1}^k g^{\tau} (g^{\tau})^T$$

Tenemos los gradientes promediados sobre el **mini-batch**. La matriz G^k aumenta de dimensión a medida que se aumentan las iteraciones k .

De manera que el learning rate se pondera por el inverso del gradiente más un hyper-parámetro ϵ para evitar que se anule.

$$\theta^{k+1} = \theta^k - \alpha (\epsilon I + \text{diag}(G^k))^{\frac{1}{2}} \nabla_{\theta} \mathcal{L}_i$$

RMSprop: (Root Mean Square propagation) es una modificación de AdaGrad para corregir learning rates que decaen de manera muy *agresiva*. Pues AdaGrad puede resultar en un decremento excesivo/prematuro.

En lugar de guardar la suma de los gradientes al cuadrado sobre todas las iteraciones, se usa una media móvil. γ es el hyperparametro que controla el peso de la matriz de gradientes previos con el gradiente actual.

$$G^k = \gamma G^{k-1} + (1 - \gamma) g^k (g^k)^T$$

$$\theta^{k+1} = \theta^k - \alpha (\epsilon I + \text{diag}(G^k))^{\frac{1}{2}} \nabla_{\theta} \mathcal{L}_i$$

Adam: es una combinación de AdaGard y RMSProp. Introduce una media móvil tanto en la velocidad de momentum como en el gradiente.

$$v^{k+1} = \gamma_1 v^k - (1 - \gamma_1) \nabla_{\theta} \mathcal{L}_{\theta} |_{\theta^k} \quad G^k = \gamma_2 G^{k-1} + (1 - \gamma_2) g^k (g^k)^T$$

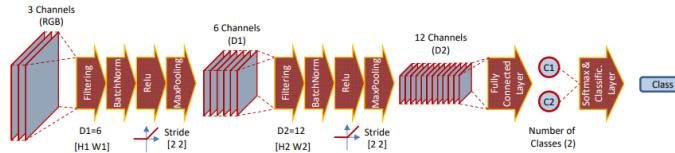
$$\theta^{k+1} = \theta^k - \alpha (\epsilon I + \text{diag}(G^k))^{\frac{1}{2}} \nabla_{\theta} \mathcal{L}_i$$

1.5 CNNs: Convolutional and Pooling Layers

Cuando tratamos con imágenes que tienen un gran número de píxeles, esto se traduce en una alta cantidad de parámetros en una red neuronal. En tales casos, una red completamente conectada se vuelve poco práctica, incluso:

- Llega a ser una solución del tipo **brute force**.
- La red resultante será difícil de entrenar.
- Se requiere una gran cantidad de datos.
- El rendimiento puede incluso decaer.

Las **redes neuronales convolucionales (CNN)** son un tipo de redes especializadas para una cierta estructura conocida, como lo son las imágenes. Para abordar este problema, podemos reducir la cantidad de parámetros que cada neurona recibe considerando un vecindario de conexiones. Otra solución posible es compartir parámetros entre neuronas.



Las partes principales de las CNNs son el filtrado, la normalización y la no-linealidad *ReLU* normalmente.

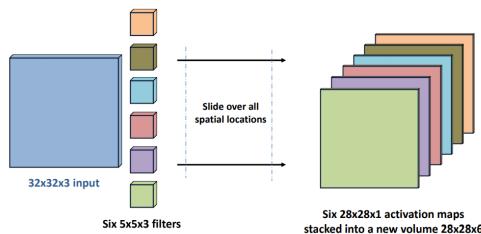
Conejividad Local

Bajo este esquema, cada neurona oculta recibe una subregión (filtro de la imagen). Consiste en filtrar la imagen con un filtro que no está definido de antemano. La ventaja de esto es que los pesos del filtro se optimizan durante la propagación hacia adelante y hacia atrás de la red neuronal. Los pesos del filtro se pueden inicializar de una manera específica (por ejemplo, utilizando un filtro promedio).

Típicamente, comenzamos con imágenes a color, lo que requiere filtros tridimensionales (por ejemplo, un filtro 3x3x3 para una imagen a color RGB).

Parameter sharing

Otra técnica implica tomar una imagen y decidir extraer diferentes características de la misma imagen. Utilizando un filtro, lo convolucionamos en toda la imagen, generando una versión filtrada de la imagen (también conocida como **mapa de características**). Las unidades dentro del mismo para de características comparten los mismos parámetros: los parámetros del filtro. Este proceso se repite para otros filtros.



Para cada conjunto de características, utilizamos un filtro **único** pero no **fijo**. Los parámetros del filtro se optimizan a lo largo del proceso. El objetivo es encontrar filtros que capturen cierta información en toda la imagen.

Aprendizaje de extremo a extremo (**End-to-End Learning**): La definición de la arquitectura permite que el sistema seleccione los pesos de cada una de las capas de filtrado. El propio proceso de filtrado realiza un **entrenamiento de extremo a extremo**.

Nos gustaría pensar que las capas de filtrado tienen un cierto significado. Extraen características que son similares a lo que el humano interpretaría. No siempre ocurre así; a menudo encuentra características que no se pueden interpretar. Como dicen, "La red encuentra cosas que yo no sabría encontrar."

1.5.1 Convolutional Layers

Padding

Si no se aplica **zero padding** (u otra solución de relleno), la dimensión de la imagen se reduciría cada vez más después de pasar por el filtro, lo que podría resultar en la pérdida de información importante. La dimensión de salida en ese caso sería $(N - F + 1) \times (N - F + 1)$, donde N es el tamaño de la imagen de entrada cuadrada y F es el tamaño del filtro cuadrado. Si se mantiene la dimensión de entrada igual a la salida del filtrado, se llama **convolución "same"**.

7x7 input data $x[m, n]$

3×3 filter $w[-m, -n]$

El padding P requerido se calcula a partir de $P = \frac{(F-1)}{2}$ en cada lado de la imagen.

Stride

¿Por qué realizar todo el filtrado de la imagen si al final se realizará un **muestreo**? El **paso (stride)** nos dice que no es necesario calcular todas las posiciones si se van a descartar, sirve para controlar cómo se desplaza la ventana de operación. En su lugar, colocamos el filtro en posiciones separadas geométricamente.

- Al aumentar el valor del stride, se reducen el número de operaciones de convolución o pooling necesarias para procesar los datos de entrada.
- Un stride mayor que 1 reduce la dimensión espacial de la salida de una capa. Esto puede ser útil si deseas reducir el tamaño de la representación resultante de manera específica y sabes que la pérdida de detalles en los bordes de la entrada no es crítica. Esto es especialmente útil si el núcleo de convolución es grande, ya que captura una amplia área de la imagen subyacente.

Las fórmulas para calcular el tamaño resultante de la imagen son las siguientes:

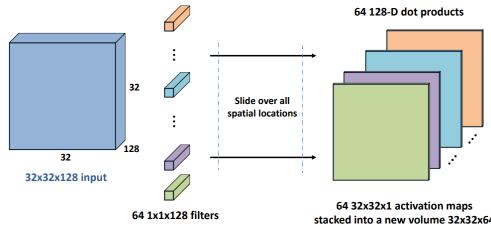
$$\text{Sin relleno: } \frac{N - F}{S} + 1$$

$$\text{Con relleno: } \frac{N - F + 2P}{S} + 1$$

No todos los valores de **stride** son válidos; dependen de los valores de N y F . $N + F$ debe ser un múltiplo de la medida del **stride**. Se reduce la dimensión de la salida del filtro, esta reducción se logra mediante un *muestreo* de la imagen.

1x1 Convolution

Compacta la profundidad/características para poder pasar a tomar una decisión. Reducción de la dimensionalidad combinando todos los mapas de características.



Parameters in Convolutional Layers

Hyper-parametros fijos para cada capa l :

- M_l , N_l dimensiones del volumen de entrada
- D_l características de entrada
- K_l Filtros
- S_l Stride
- P_l Padding

The number of parameters to optimize is:

$$(F_l \cdot F_l \cdot D_l + 1) \cdot K_l$$

1.5.2 Batch normalization and Pooling Layers

Sobre las capas de normalización y reducción de dimensionalidad (**pooling**).

Batch normalization Layers

Se intenta que los valores input queden normalizados a una distribución gaussiana. Que los valores de las características y las salidas de las funciones de activación dentro de cada capa sean similares y no haya una predominante en la combinación.

$$X_{\text{norm}} = \frac{X - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

No solo se normaliza a nivel de características, sino que también se hace una normalización a nivel de todo el **min-batch**.

- La Batch Normalization ayuda a acelerar y estabilizar el proceso de entrenamiento de redes neuronales. Al normalizar las activaciones intermedias en cada capa, la propagación hacia atrás se vuelve más eficiente, lo que permite entrenar redes más profundas y en menos tiempo.
- La Batch Normalization hace que la red sea menos sensible a la inicialización de pesos. Esto significa que es menos probable que la elección de pesos iniciales cause problemas como gradientes que desaparecen o explotan.
- La Batch Normalization puede llevar a una convergencia más rápida del entrenamiento, pues se previene de tener una dispersión de VAPs que afecten a la optimización de parámetros, lo que significa que la red alcanza un rendimiento deseado en menos épocas de entrenamiento.

Si la red quiere potenciar alguna característica, se le puede dar la oportunidad mediante dos parámetros a optimizar (para cada neurona) para que la red potencie la característica por su cuenta.

$$BN(x) = \gamma \cdot x_{\text{norm}} + \beta$$

Se aplica una no-linealidad **ReLU** para poner a 0 aproximadamente la mitad de los pesos (pues estadísticamente la mitad de los pesos son negativos).

Pooling layers

En Stride6 no se tiene en cuenta las propiedades de los datos, sino que es una selección en base a un concepto geométrico de saltos. Las capas de pooling se centran en la reducción de dimensión en función de la combinación de los datos para extraer un "representante" de los datos.

Permite actuar en cada mapa de características de forma diferente. No agrega parámetros extras a optimizar. Es una actuación fija.

1.6 CNN: Upsampling and Skip connections

1.6.1 Introduction

Cuando creamos la etapa *fully connected* tenemos que hacer que coincida con las dimensiones previas de convolución. Esto **no** nos permite procesar imágenes de cualquier dimensión.

Si lo que queremos es no solo detectar elementos pero también definir el segmento en el cual esta representado en la imagen, se necesita recuperar la dimensión de la imagen original (que ha sido reducida durante las capas de convolución).

Semantic Segmentation

Queremos que cada píxel de entrada tenga un valor de píxel de salida. Una idea ingenua sería aplicar una red neuronal para cada píxel.

Lo que haremos será construir una arquitectura que llegue a dar descriptores pertinentes (*embedding*), que representen muy bien los datos iniciales. A partir de estos datos crearemos etapas para ir recuperando lo que hayamos hecho en los pasos previos. Con técnicas similares a las de codificación (en la convolución) pero incrementando la resolución de los resultados, lo cual requerirá también un aprendizaje de optimización.

No nos interesa recuperar todo. Pero para recuperar las dimensiones, se necesitan compensar las convoluciones (**convolucion transpuesta**) y compensar el Max-Pooling (**Max-Unpooling**).

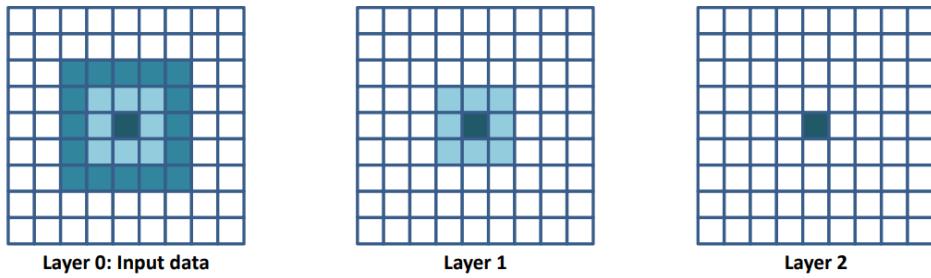
Necesitamos tener un conjunto de *ground truth images* para entrenar la red. Es decir, necesitamos imágenes ya segmentadas para poder comparar.

Receptive field

El **receptive field** se define como, para un elemento en concreto de una capa l de la CNN en particular, la región en el espacio del input que afecta a dicho elemento.

En mi capa 2 (azul oscuro), tengo un elemento que ha sido calculado a partir de un cálculo con los elementos vecinos mediante el filtro de la capa anterior y así sucesivamente hasta los elementos de la imagen de entrada. En el ejemplo de la imagen, se han utilizado un filtro de (3x3) y dos capas de convolución.

Si queremos trabajar con imágenes que tengan un **receptive field** de tamaño 25 (5x5), podemos plantearnos utilizar una única etapa convolucional con un filtro de 5x5. En este caso, tendríamos 25 parámetros a optimizar más un sesgo. En lugar de esto,



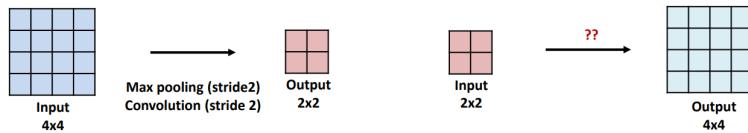
si utilizamos dos capas de convolución con filtros de 3×3 , tendremos 20 parámetros a optimizar en total ($9+9$) y los dos sesgos (uno en cada capa) y mantendremos el mismo campo visual (25 píxeles).

Esta manera de reducir el tamaño del filtro y aumentar las capas de convolución han resultado ofrecer mejores resultados. Algunas razones de esto pueden ser:

- Reducir el tamaño de los filtros permite capturar características más locales y específicas en una imagen. Al agregar más capas de convolución, se pueden combinar estas características locales para formar representaciones más complejas y abstractas de la imagen, lo que aumenta la capacidad de representación de la red.
- Usar filtros más pequeños reduce la cantidad de parámetros en la red en comparación con filtros más grandes. Esto es beneficioso en términos de eficiencia computacional y evita problemas de sobreajuste en conjuntos de datos pequeños.

1.6.2 Upsampling

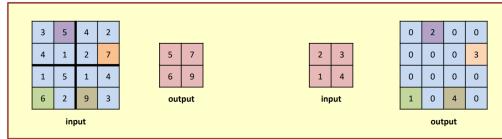
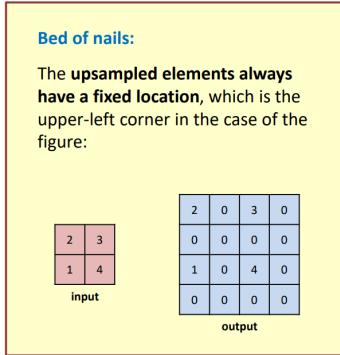
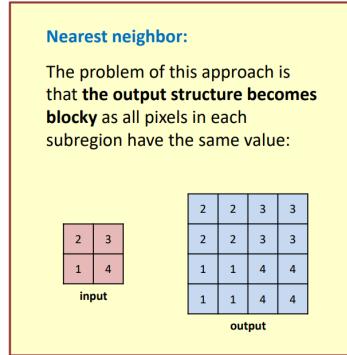
Mientras que en las capas de convolución y Max-Pooling la resolución se puede ver afectada, en la descodificación se busca contrarestar esta afectación.



Unpooling

Se busca realizar la operación contraria a Pooling: recuperar el tamaño original del mapa de características de entrada. Las técnicas principales son:

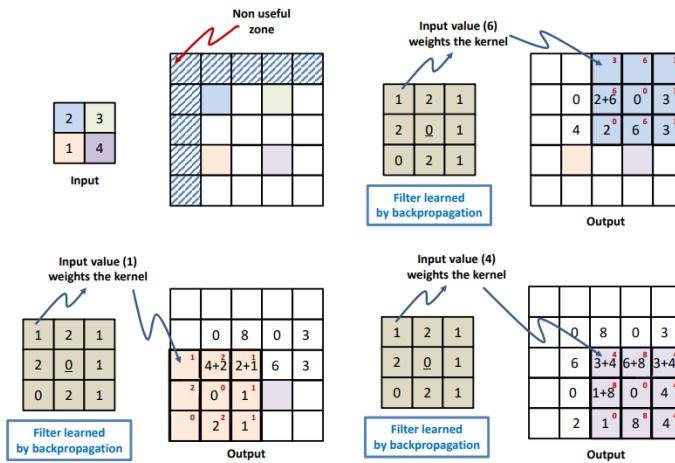
- **Nearest Neighbor:** Repetir el valor para rehacer el Max-Pooling. El problema de esta implementación es que la estructura del output se vuelve bloqueada (*blocky*), ya que todos los elementos en cada subregion tienen el mismo valor.
- **Bed of nails:** Rellenar con 0 y posicionar el valor diferente de cero en una misma posición. Los ceros quedaran interpolados y se llenan con los valores interpolados con un sistema que hace algo parecido a la convolución. A diferencia del método del vecino más cercano, no produce el efecto de bloqueo, ya que solo se copian los valores originales. Mantiene los detalles de la imagen original y evita la suavización excesiva que se encuentra en el método del vecino más cercano.
- **Max unpooling:** Hacemos un link con el input-output del paso Max-Pooling. Ubicamos el valor del input en el lugar donde habíamos encontrado el máximo anteriormente (mediante el almacenamiento de los índices de posición). Nos ayuda a preservar un poco la información que teníamos en la imagen original.



Transpose Convolution

Los acercamientos anteriores proponen una operación fija, mientras que las convolución transpuesta propone aprovechar el aprendizaje. ¿Cómo pasamos de elementos 2x2 a elementos 4x4 interpolados?

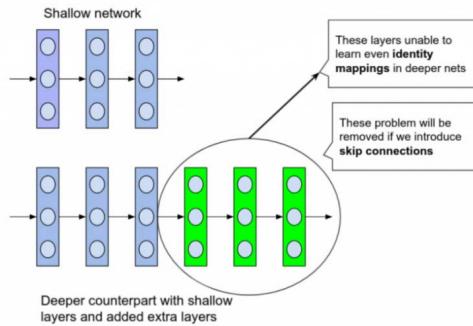
Se agrega un margen que no se utiliza pero facilita la implementación (la implementación se puede modificar). El filtro optimizado se ubica en la posición que corresponde a cada valor del input y se multiplica por ese valor. Esto se hace para las cuatro posiciones y los resultados que se sobreponen se suman (pixeles que reciben más de una contribución de varios elementos).



Dependiendo del tipo de Stride que se aplique y el tamaño del filtro a la interpolación, aparecen patrones visibles en las imágenes resultantes. Se solapan filtros y tenga elementos que reciban más de un valor. Esto proviene en parte también por el Max-Unpooling.

1.6.3 Skip connections

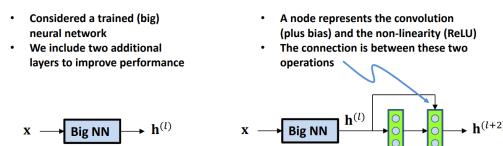
Si incrementamos el tamaño de la red (número de capas → parámetros) debería dar mejores resultados aunque se requiera más datos de entrenamiento y más tiempo por época. Pero **no**. Hay casos donde incrementamos el número de capas pero obtenemos un error mayor, se conoce como **degradation problem**. Hay un problema de agregación, hacen falta muchos más datos para mejorar una red muy grande hasta que se introdujo **Skip connections**.



Tenemos una red que tiene unas capas y que funciona aceptable. Pero añadimos unas capas (verde) esperando que mejore. Pero estas capas lo único que hacen es degradar los resultados (no son capaces ni de reproducir la identidad, es decir, dar el mismo resultado).

Residual Block

Para resolver este problema se introducen dos capas (verdes) para mejorar el rendimiento a nuestra red de gran tamaño (azul).



$$\mathbf{h}^{(l+2)} = g(\mathbf{W}^{(l+2)}\mathbf{h}^{(l+1)} + \mathbf{b}^{(l+2)} + \mathbf{h}^l)$$

Si los pesos y los sesgos se ponen a 0 (gradiente colapsado). Entonces los **residual Block** pueden aprender la función identidad. Lo que en el peor de los casos, el hecho de añadir estas dos capas tiene mucha probabilidad de no empeorar.

Se tiene que asegurar que las dimensiones de las matrices coincidan, para ello se realizan convoluciones de tipo same (con padding y stride=1) y no pooling.

En el caso de que las dimensiones no se cumplan, se agrega una matriz de pesos (que se aprende en el proceso de aprendizaje).

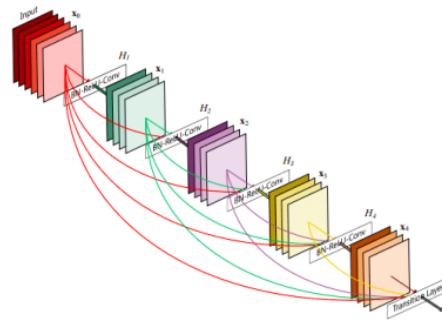
$$\mathbf{h}^{(l+2)} = g(\mathbf{a}^{(l+2)} + \mathbf{W}_s \mathbf{h}^{(l)}) = g(\mathbf{W}^{(l+2)} \mathbf{h}^{(l+1)} + \mathbf{b}^{(l+2)} + \mathbf{W}_s \mathbf{h}^{(l)})$$

Including residual blocks in CNN

Hay muchos filtros pero de dimensiones pequeñas porque así optimizamos el número de parámetros. Tenemos el mismo campo de visión pero a un coste muy menor.

Addition and concatenation

Coneectar todos los canales con todos (Redes densas). Aumento de complejidad y de desempeño.

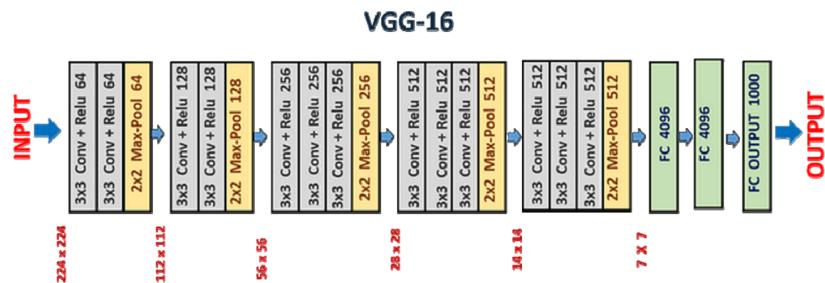


1.7 Architectures and Interpretability

1.7.1 VGGNet-16

Crea un entorno mucho más conceptual que lo que había antes. Se trata de un arquitectura donde tenemos un conjunto de capas convolucionales de tamaño muy pequeño. Tenemos concatenamos convoluciones con una etapa no lineal 3x3 que consigue hacer la reducción del tamaño de la imagen.

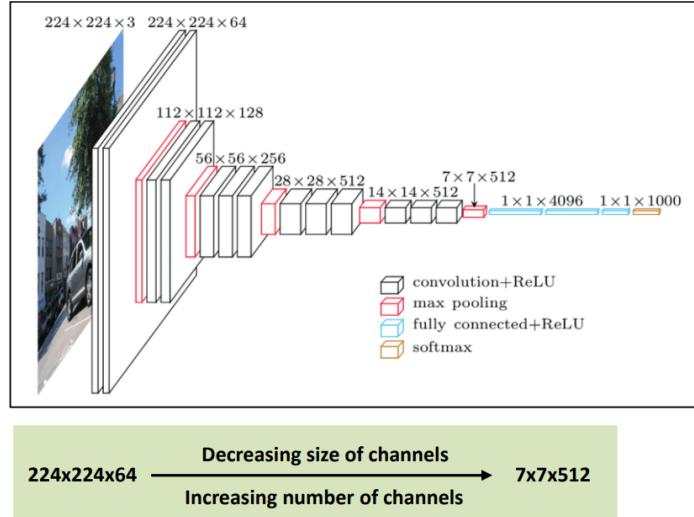
Hacemos convoluciones pequeñas porque obtenemos un campo de visión amplio reduciendo la complejidad.



Tenemos cinco etapas con sus no-linealidades respectivas. Tenemos una etapa de extracción de características que permiten estudiar mi problema y luego tenemos la red con FC layers que lo que hacen es tomar la decisión. En este caso como tenemos que predecir una clase la función de activación final es una Softmax.

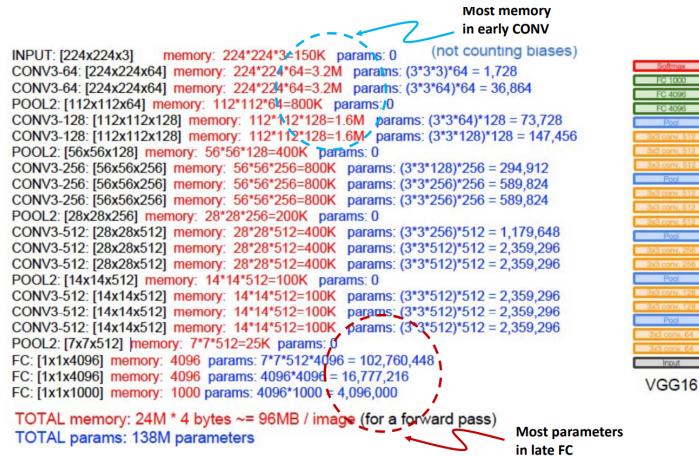
Como tenemos cada vez más canales tenemos cada vez más features.

Tenemos una secuencia de capas que se van entrenando secuencialmente mediante **same convolution**. La red tiene alrededor de 138M de parámetros. Son volúmenes de



datos muy relevantes.

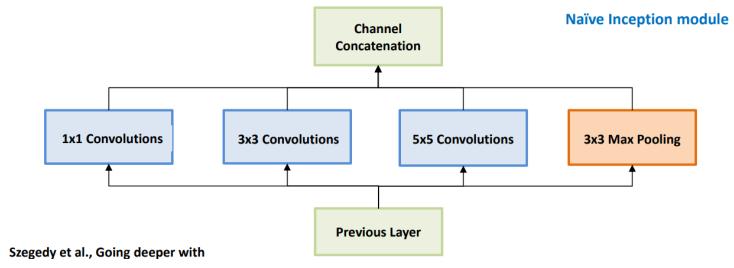
La medida de los canales disminuye pero el numero de canales aumenta. Esto implica que tenemos una progresión de mucha memoria requerida al principio contra muchos parámetros necesarios al final. Las primeras capas ocupan muchas mas memoria que las siguientes pero el numero de parámetros aumenta.



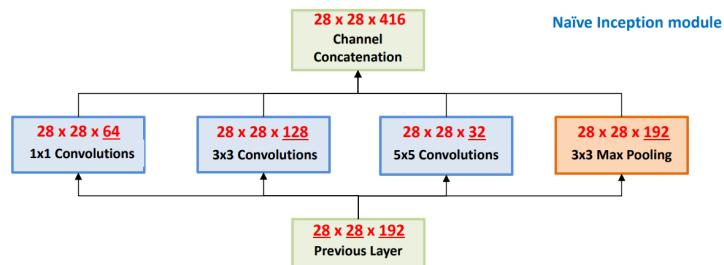
La AlexNet tenía una estructura en la cual las capas trabajan con un filtro de 11x11, esto implica tener 121 elementos y lo que es mejor opción es concatenar 5 etapas de 3x3 para tener este campo de visión. Lo que propuso esta red es un esquema de trabajo para la toma de decisiones mediante la red neuronal.

1.7.2 Inception Module

Modelo que se basa en la **inception** que lo que hace es no escoger el orden de las etapas de convolución, no linealidad. Sino que se pone en paralelo. Esto explota el número de parámetros.



La salida de los elementos en paralelo tiene que resultar en una concatenación con las mismas dimensiones de entrada. Incluyendo el MaxPooling (stride=1).



Conlleva en un gran numero de cálculos y operaciones de convolución.

Bottleneck

En cambio se propone un paso cuello de botella, que evite cruzar los dos conjunto de dimensión alta. Se ha hecho una reducción de 8, en función de este parámetro se hacen mas o menos operaciones.

1.7.3 GoogLeNet

La idea es ir cada vez más profundo, cada vez analizar mas en detalle. Introducen la estructura de antes. Lo que tenemos es una extensión de lo que habíamos visto hasta ahora, introduciendo los módulos de **inception**. Además introducen salidas laterales de la red. Esta red ya no esta siendo entrenada para 1 finalidad sino para 3 (en este caso). Tiene clasificadores auxiliares que ayudan al clasificador final o hacer aplicaciones importantes para el problema.

1.7.4 ResNet

Como tenemos cada vez más y más capas nos encontramos con el problema de que las capas que añadimos no mejoren nuestros resultados. Para este se propuso los bloques residuales. La manera en la cual se propuso es evitar que colapse la red. No solo evitan el crecimiento erróneo de la red pero guía la red hacia una mejor optimización de los parámetros.

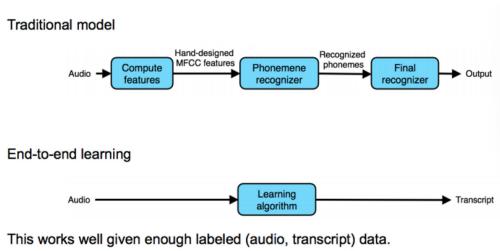
1.7.5 Interpretability

Un red neuronal suele esconder como toma la decisión, esto puede generar errores, sesgos, etc. Y estos problemas condicionan los resultados ante la sociedad.

2 Practical Aspects in Neuronal Networks

Si la red es grande, su potencial de realizar overfitting incrementa. tenemos el compromiso entre la capacidad de la red y su performance (overfitting).

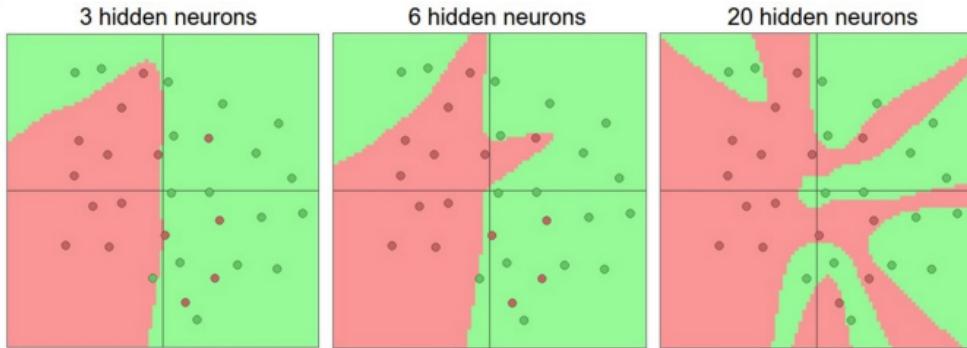
La capacidad de calculo ha evolucionado mucho a lo largo de los años (GPU). Ha hecho que el uso de las redes neuronales sea mas viable. También las arquitecturas para potenciar las redes neuronales.



Antes, los métodos tradicionales calculaban manualmente las características más importantes del input. A partir de ahí, era posible reconocer patrones, características, etc. En cambio, con las redes neuronales, se puede aprender de los datos sin necesidad de calcular manualmente las características (End-to-End). Esto permite montar un sistema que aprenda a traducir automáticamente, por ejemplo.

2.1 Capacity of the Network

La capacidad de la red esta asociada al potencial que tiene la red en representar diferentes funciones de entrada/salida. Una capacidad pequeña implica que la red puede mapear un numero limitado de funciones. Una capacidad grande implica que la red puede mapear un numero grande de funciones.



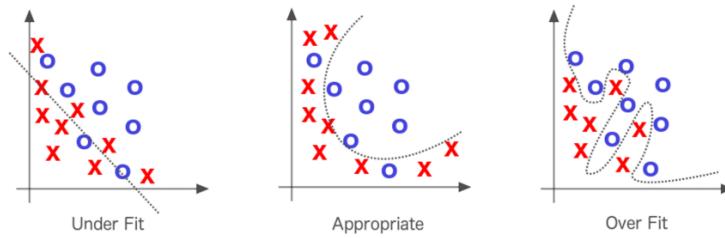
Si entrenamos esto, la red hará una división entre los puntos rojos y verdes. Si la capacidad es pequeña, las divisiones serán simples. Si la capacidad es grande, las divisiones serán mas complejas. Si la capacidad es muy grande, la red puede hacer overfitting.

Parece lógico que entre mas capas y mas nodos tienan mas capacidad. Existe una relación no analítica pero podemos entender que a mas parámetros, la capacidad es mayor.

2.1.1 Generalization

No queremos que las redes aprendan los datos de entrenamiento, sino que generalicen para que funcionen en otros datos. Si la red aprende los datos de entrenamiento, no generaliza. Si la red generaliza, no aprende los datos de entrenamiento.

La idea es que no queremos estar ni en el caso de overfitting ni en el caso de underfitting. Queremos estar en el medio.



El "generalization gap" (brecha de generalización) se refiere a la diferencia en el rendimiento de un modelo de aprendizaje automático entre su desempeño en los datos de entrenamiento y su desempeño en datos que no ha visto durante el entrenamiento, como el conjunto de datos de validación o prueba.

2.1.2 Overfitting

La red es capaz de aprender los datos de entrenamiento tan bien que luego no es capaz de generalizar. Una causa puede ser que tenemos una red demasiado compleja, la capacidad de la red es demasiado grande para aprender los datos de entrenamiento.

2.1.3 Underfitting

Un caso en el que la red no puede aprender los datos de entrenamiento. Una causa puede ser que tenemos una red demasiado simple, la capacidad de la red no es lo suficiente para aprender los datos de entrenamiento.

2.2 Data Partition

División del conjunto de datos en dos subconjuntos disjuntos: conjunto de entrenamiento y conjunto de test. El conjunto de entrenamiento es el que se usa para entrenar la red. El conjunto de test es el que se usa para evaluar la red.

Se tiene que añadir un set de validación. Se usa para evaluar la red durante el entrenamiento. Se usa para evitar el overfitting. Se usa para seleccionar el modelo que mejor generaliza. Se usa para seleccionar los hiperparámetros de la red.

Cuando escogemos una red de baja capacidad, su loss suelen tener mínimos locales que no ayudan. La mayoría de los mínimos locales tienen una diferencia significativa con respecto a los mínimos globales. Cuando aumentamos la capacidad, la función de loss tienen muchos mínimos locales pero tienden a uniformizarse. Los mínimos locales tienden a ser más parecidos a los mínimos globales.

Es mejor usar redes de más capacidad e intentar reducir el overfitting (desde un punto de vista técnico).

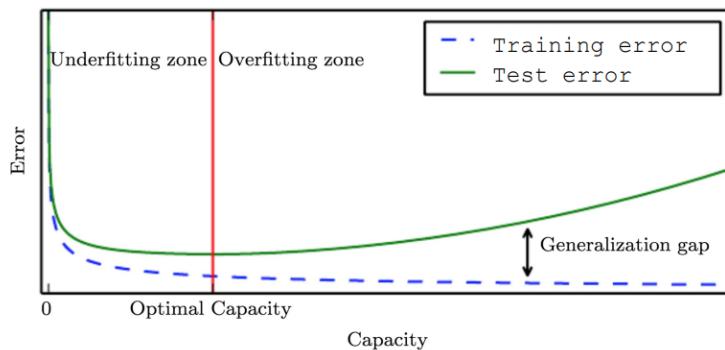
2.3 Prevent Overfitting

Buscamos que la loss de test tenga una estructura similar a la loss de entrenamiento.

2.3.1 Early Stopping

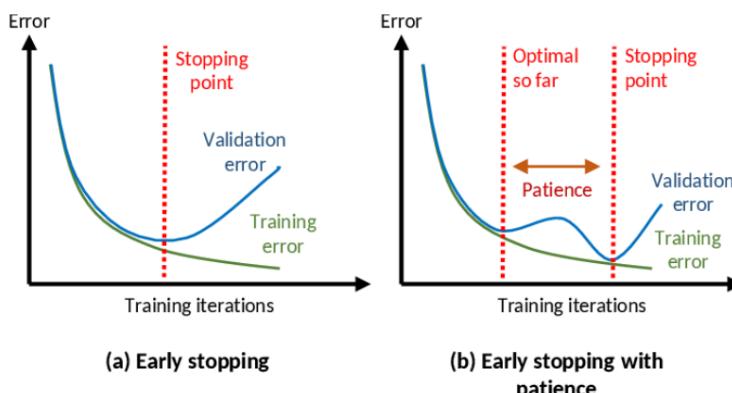
Consiste en detener el proceso de entrenamiento del modelo antes de que este comience a sobre ajustarse a los datos de entrenamiento, lo que podría conducir a un rendimiento deficiente en datos nuevos o de prueba. Cuando se emplea el early stopping, se detiene el entrenamiento del modelo cuando la métrica de validación deja de mejorar o comienza a empeorar, lo que indica que el modelo podría haber alcanzado su capacidad máxima de generalización y podría estar empezando a sobre ajustarse a los datos de entrenamiento. Esto ayuda a evitar que el modelo se ajuste demasiado a los datos específicos de entrenamiento y mejora su capacidad para generalizar bien a datos nuevos.

Por lo tanto, a medida que la red va aprendiendo, casi siempre el error de training va disminuyendo con las iteraciones. En cambio, el error de validación va disminuyendo hasta un punto y luego empieza a aumentar y no va tan bien como el training y consigue overfitting.



Se va midiendo el mínimo de validación y cuando se llega a un mínimo, se para el entrenamiento. Los pesos de la validación mínima se guardan y se usan para evaluar la red como la que mejor generaliza.

Con **paciencia** significa que hay una cierta paciencia de iteraciones que se mira a partir del primer mínimo y hay un aumento para buscar la posibilidad de que el mínimo se pueda mejorar. Si no se mejora, se para el entrenamiento. Es una extensión o variante del método de early stopping estándar que introduce un grado de tolerancia antes de detener el entrenamiento del modelo. En lugar de detenerse inmediatamente cuando la métrica de validación deja de mejorar, el early stopping con paciencia espera un cierto



número de épocas o iteraciones adicionales después de que la métrica de validación deje de mejorar antes de tomar la decisión de detener el entrenamiento.

Evita detener prematuramente el entrenamiento del modelo debido a fluctuaciones menores en la métrica de validación. Si bien la métrica puede empeorar temporalmente debido al ruido o variaciones aleatorias en los datos, la paciencia permite que el modelo continúe su entrenamiento en caso de que el desempeño mejore nuevamente después de un pequeño deterioro.

2.3.2 Weight Regularization

No queremos que la red haga cambios bruscos en la salida para los cambios pequeños en la entrada. Queremos que la red sea suave. Para esto se usa esta intuición en los que los pesos grandes no interesan (parece que contribuyan al overfitting). Como puedo hacer para que a red no aprenda pesos grandes en el training?

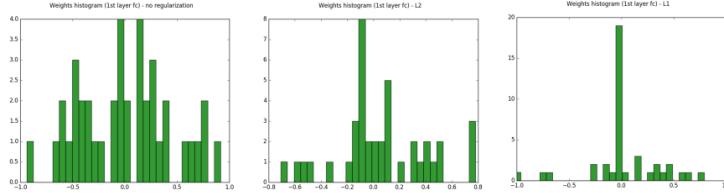
Le añado a la loss una regularización en función de los pesos. Se le suma un parámetro que depende de los pesos. L2 o L1.

$$\mathcal{L}_{L2} = \mathcal{L} + \frac{\lambda}{2} W^2, \quad \mathcal{L}_{L1} = \mathcal{L} + \frac{\lambda}{2} |W|$$

Cuando se añade la regularización, hay más pesos con valores pequeños que valores grandes. Poner la regularización hace que los pesos además se distribuyan en valores pequeños y esto evita el overfitting. Estas técnicas agregan términos de penalización a la función de pérdida durante el entrenamiento, lo que afecta los coeficientes de los parámetros del modelo.

La penalización tiene el efecto de controlar el crecimiento excesivo de los parámetros del modelo, lo que ayuda a evitar que el modelo se ajuste demasiado a los datos de entrenamiento y, en cambio, mejore su capacidad de generalización a datos no vistos.

$$\mathcal{L}_{new} = \mathcal{L} + \frac{\lambda}{2} W^2 \quad \mathcal{L}_{new} = \mathcal{L} + \frac{\lambda}{2} |W|$$



2.3.3 Activation Regularization

Similar a la idea de anterior, en lugar de actuar sobre los pesos, actuar sobre las salidas de las activaciones. No se usa tanto en la práctica.

Cuando las activaciones se vuelven muy grandes, esto puede conducir al sobreajuste, donde la red neuronal se adapta demasiado a los datos de entrenamiento y tiene dificultades para generalizar bien a datos nuevos.

La "activation regularization" busca evitar este problema al agregar términos de penalización en la función de pérdida del modelo que castigan las activaciones grandes. Al agregar esta penalización, se desalienta a las neuronas de generar activaciones excesivamente altas, lo que puede ayudar a reducir el sobreajuste.

2.3.4 Batch Normalization

Generalmente las capas son muy dependientes de la salida de la capa anterior, lo que puede provocar que los valores de algunas neuronas sean muy grandes o muy pequeñas y se propaguen a otras neuronas. Las capas se coadaptan unas a otras, esto facilita el overfitting. Una manera de evitarlo es cuando pasamos de una capa a otra se normalizan los datos. **Batch Normalization** es una técnica utilizada en redes neuronales profundas para normalizar y estabilizar las activaciones a lo largo de una red, específicamente normalizando las entradas de cada capa antes de la función de activación.

Se hace para todo el dataset si se quiere hacer bien. Esto no se hace, se reduce para cada *batch*. Se calcula la media y la varianza por cada canal de un *batch*.

Si se aplica tal que así, tiene una contrapartida, pues en algunas activaciones se desaprovecha la etapa no lineal. Se escala y se mueve con un sesgo para tener la libertad de modificar y aprender como normalizar.

$$x_{i+1} = \gamma \dots$$

- Mejora el tiempo de entrenamiento y la precisión de los resultados
- Reduce el efecto de la inicialización de los datos
- Dependiente del tamaño del *batch*
- No funciona bien con redes recurrentes
- No se puede aplicar *batch normalization* en el test pero la red espera que el input este normalizado. (Se usa la media y la varianza del set de training y es la que se aplica en el test).

Una modificación es normalizar por datos de entrada. **Layer normalization**. Ahora normalizamos para todos los canales fijando el dato de entrada.

- La ventaja es que no depende de todo el *batch*
- Se puede aplicar a redes recursivas.
- No funciona bien para redes convolucionales.

2.3.5 Dropout

En cada iteración, cada backpropagation hacemos que algunos de la red se quiten. Tenemos una probabilidad p en la que cada iteración el nodo se "apaga", estas neuronas eliminadas no contribuyen ni reciben señales durante el pase hacia adelante (forward pass) ni el pase hacia atrás (backpropagation) en esa iteración de entrenamiento. En cada iteración consideramos la red un poco distinta. Las capas no se adaptan a los nodos de la capa anterior y aprende a depender de mas nodos de entrada, lo que hace que generalice mejor.

Al aplicar dropout, la red neuronal se ve obligada a aprender características más robustas y útiles, ya que no puede depender excesivamente de un conjunto específico de neuronas en cada iteración. Esto evita que las neuronas se adapten demasiado a las características particulares de los datos de entrenamiento y promueve la creación de representaciones más generalizables.

Al final con el **Dropout** tenemos un conjunto de redes mas pequeñas que todas contribuyen al resultado. Esto es mas robusto que solo una única red. **En test no se hace dropout**. La salida se escala por un factor durante el training para adaptar la escala de la salida.

Como se quitan nodos aleatorios, se necesitan mas iteraciones para que la red aprenda. La red tarda mas en aprender.

2.3.6 Data Augmentation

La idea principal detrás del data augmentation es aumentar la cantidad y variedad de datos de entrenamiento sin necesidad de recopilar más datos reales. Al aplicar transformaciones como rotaciones, traslaciones, zoom, volteos horizontales o verticales, cambios en el brillo y otros cambios geométricos o de apariencia, se generan nuevas versiones de las imágenes originales. Si tengo mas ejemplos, reduce el overfitting. Una manera de aumentar los datos artificialmente, es modificar los datos de entrada para hacer que parezcan un poco distintos y conseguir mas datos.

Altamente dependiente de los datos de entrada. Suele ser un proceso online, mientras entrenamos introducimos diferencias a los datos de entrada. Algunas transformaciones necesitan transformar el ground truth también.

2.3.7 Label Smoothing

Su objetivo es mejorar la generalización del modelo y prevenir el sobre ajuste, suavizando las etiquetas (labels) o las salidas deseadas de un modelo durante el proceso de entrenamiento. En lugar de definir el ground truth en base a one-hot encoding, se pueden suavizar la distribución para bajar la capacidad de overfitting.

El impacto de este suavizado es que el modelo no está tan seguro sobre las etiquetas; en lugar de estar completamente seguro (probabilidad 1) de la clase correcta y completamente seguro de que todas las demás clases son incorrectas (probabilidad 0), el modelo se entrena para ser menos confiado y tener una distribución de probabilidad más suave.

2.3.8 Parameter Sharing

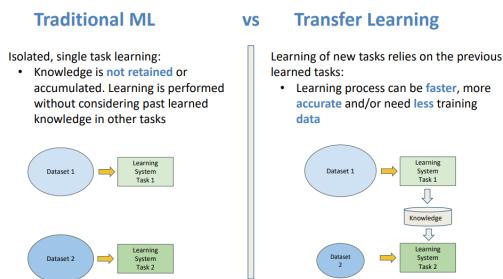
A veces es bueno reutilizar la misma red para hacer varias tareas. Esto hace que se reparta la capacidad y tienda a reducir el overfitting. Menos pesos que hagan mas trabajo.

2.4 Strategy

Necesitamos un nivel adecuado/mínimo de precisión. Puede ser basándose en un experto en el tema. Después se realiza el split de los datos.

2.5 Transfer Learning

Aprovechar la habilidad de transferir conocimiento de una tarea ya realizada a nuevas tareas. Poder adaptarse a nuevas tareas transfiriendo conocimiento. Tengo un dataset que entreno y aprovecho el conocimiento para otro dataset.



- Permite obtener un modelo mejor inicializado, ya que se aprovecha el conocimiento previamente adquirido del modelo pre-entrenado, lo que puede conducir a un rendimiento superior en la tarea de destino, especialmente en situaciones donde el conjunto de datos de destino es pequeño.
- Al aprovechar los conocimientos del modelo pre-entrenado, se puede lograr un buen rendimiento con menos datos de entrenamiento, lo que resulta útil en casos donde se dispone de recursos limitados.

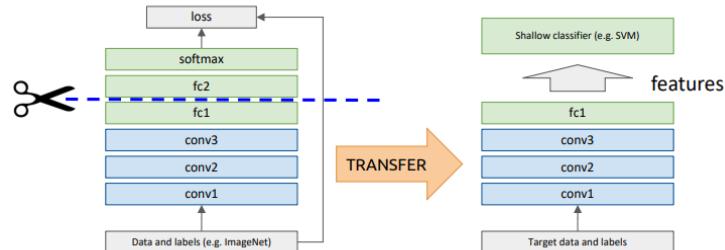
Domain: Consiste en el espacio de características de los datos y por supuesto pueden tener diferente distribución. Un dominio se caracteriza por las propiedades de los datos, que pueden incluir aspectos como la distribución estadística, el tipo de características, la representación de los datos y las características específicas de la tarea en cuestión. En el contexto del aprendizaje automático, diferentes conjuntos de datos pueden pertenecer a diferentes dominios si tienen propiedades o características distintas.

- **Domain bias:** tenemos que tener en cuenta que siempre esta presente. No es posible tener una red que se pueda adaptar a todos los dominios que existen.

Task: Consiste en la tarea que se quiere realizar, donde figura el label space (espacio de agrupaciones) y la función predictiva. Podemos tener tareas diferentes, reconocer personas (políticos, deportistas, etc.). también se pueden tener tareas diferentes, no en el label space, sino en la función predictiva (clasificación, regresión, etc.). La task en transfer learning es la tarea que se está intentando resolver utilizando el conocimiento aprendido por el modelo pre-entrenado en una tarea relacionada o diferente. Esta task de destino puede variar ampliamente y puede incluir problemas de clasificación, detección de objetos, segmentación semántica, traducción de idiomas, generación de texto, entre otros.

2.5.1 General techniques

Como podemos transferir el conocimiento de un dominio a otro? En una CNN tenemos diferentes capas y las primeras capas generalmente se especializan en características de bajo nivel, y las últimas capas se especializan en características de alto nivel. Es lógico pensar que para entrenar una tarea y dominio específico, las primeras capas tienen características más genéricas.



Off-the-shelf: Si queremos transferir conocimiento, las primeras capas se pueden adaptar fácilmente, no como el caso de las últimas capas. Re-aprovechamos las primeras capas y re-entrenamos las últimas capas en el nuevo dominio. Ahora necesitamos menos datos para entrenar la red y es menos propensa a overfitting.

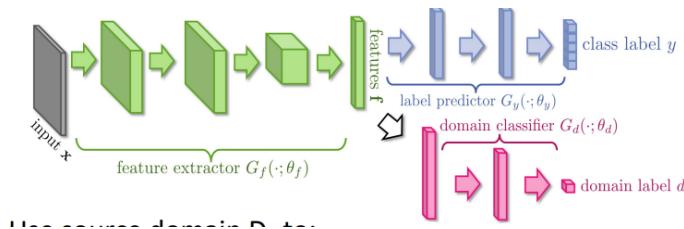
Fine-tuning: Otra opción es re-entrenar todas las capas. Se toman las primeras las capas como inicialización en lugar de random para mi entrenamiento (se necesitan menos muestras para el entrenamiento). Hay un juego de hiperparametros porque no todas se tiene que entrenar con el mismo learning rate (normalmente se usa un learning rate pequeño en las primeras capas y uno mas grande en las ultimas).

2.5.2 Domain Adaptation

Solo queremos adaptarnos al nuevo dominio, la tarea continua siendo la misma. Suponemos que no tenemos labels del nuevo dominio (**unsupervised domain adaptation**), lo que no nos permite simplemente re-entrenar la red. Buscamos que los dos espacios de características sean el mismo y la función predictiva funcione. Como sabemos si los dos espacios de características son el mismo? No lo sabemos, pero podemos hacer que se parezcan.

El **Unsupervised Domain Adaptation** (UDA), o adaptación de dominio no supervisada, es una rama del aprendizaje automático que aborda el problema de adaptar un modelo entrenado en un dominio fuente, donde se tienen datos etiquetados, a un dominio objetivo diferente pero relacionado, donde no se tienen etiquetas o se tienen pocas etiquetas disponibles. El objetivo es aprender representaciones o características que sean útiles y transferibles entre ambos dominios, de manera que el modelo pueda generalizar mejor a datos no etiquetados del dominio objetivo.

Necesitamos un clasificador que nos indique si dos imágenes son del mismo dominio o no. Si el clasificador no puede distinguir entre los dos dominios (su error de calorificación es alto), entonces los dos dominios no son separables.



Podemos separar la red en tres bloques y entrenarlas con el dominio de origen:

- **Feature Extractor:** Extracción de características que deben ser similares entre los dos dominios. Entrenarlo para que el clasificador binario tenga un error alto. Este extractor puede ser una red neuronal convolucional (CNN) en el caso de imágenes, o una red neuronal recurrente (RNN) en el caso de secuencias de texto. El objetivo del extractor de características en UDA es aprender representaciones robustas y transferibles que capturen información útil de ambos dominios.
- **Clasificador:** Es un componente del modelo que se entrena en el dominio fuente con datos etiquetados para realizar la tarea específica de clasificación. Por ejemplo, en un problema de clasificación de imágenes, este clasificador intentaría predecir las clases de las imágenes utilizando las características extraídas por el extractor de características.
- **Clasificador binario:** Este clasificador se utiliza en algunos enfoques de UDA como parte de una estrategia para alinear las representaciones de los dos dominios. Su función es distinguir entre muestras del dominio fuente y muestras del dominio objetivo utilizando las características extraídas por el extractor de características.

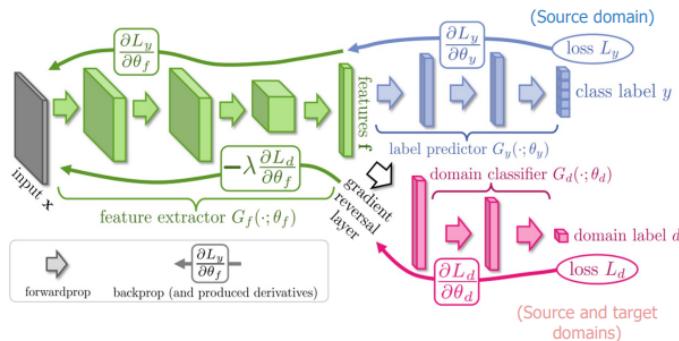
A través de este proceso, se busca minimizar la discrepancia entre las distribuciones de los dos dominios para mejorar la adaptabilidad de las características extraídas.

Con el **source domain** se pueden entrenar los tres bloques, con el segundo solo se pueden entrenar el **feature extractor** y el **classificador binario**. En el entrenamiento queremos minimizar el error de la clase label y que las features sean invariantes al dominio, para eso tenemos que maximizar el error del clasificador binario.

En el entrenamiento del modelo en el dominio fuente (source domain) y utilizando los tres bloques mencionados previamente, se busca aprender representaciones útiles y transferibles que puedan generalizar bien a la tarea específica. Durante este entrenamiento, se minimiza el error de la clasificación de las etiquetas (el error del clasificador de clases) para lograr una buena predicción de las clases en el dominio fuente.

Por otro lado, en el dominio objetivo, el clasificador de clases no se entrena debido a la falta de etiquetas en el dominio objetivo. El objetivo aquí es que el extractor de características aprenda a extraer características invariantes al dominio, es decir, características que sean relevantes y útiles para la tarea de clasificación pero que no dependan específicamente del dominio en el que se encuentren.

Para lograr esta invarianza al dominio, se busca maximizar el error del clasificador binario en el dominio objetivo. Al maximizar el error del clasificador binario (que intenta distinguir entre el dominio fuente y el dominio objetivo utilizando las características extraídas por el extractor), se fuerza al extractor de características a aprender representaciones que sean similares entre ambos dominios, lo que se traduce en características que capturan información relevante para la tarea de clasificación pero que no dependen específicamente de las diferencias entre los dominios.



$$\mathcal{L} = L_y - \lambda L_d$$

λ determina el peso del clasificador binario. Cuando se entrena el clasificador binario se busca minimizar el error, porque sino simplemente bastaría con iniciar random los valores en el entrenamiento conjunto para maximizar su error.

2.5.3 Task transfer

Tenemos el mismo dominio pero queremos cambiar la tarea (cambiar la función predictiva). Una posible manera es usando **distillation**. Utilizar las predicciones equivocadas

del modelo incial, es decir en la classificacion, centrarse en la infromacion que proporcionan las clases con probabilidades de classificacion menor. Y en lugar de que el ground truth sea una distribucion one-hot-encoding, entrenar el nuevo modelo con la distribucion de prediccion del modelo incial (destilar el conocimiento del modelo incial al nuevo modelo).

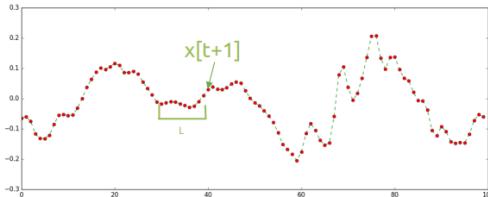
La idea principal de la destilación es transferir el conocimiento y las habilidades aprendidas por el modelo maestro al modelo alumno de manera que el modelo alumno pueda aproximarse lo más posible al rendimiento del modelo maestro en la tarea objetivo.

Se recomienda utilizar un hiperparametro de temperatura para suavizar mucho mas la distribución de predicción de salida del modelo inicial.

3 Advanced Architectures

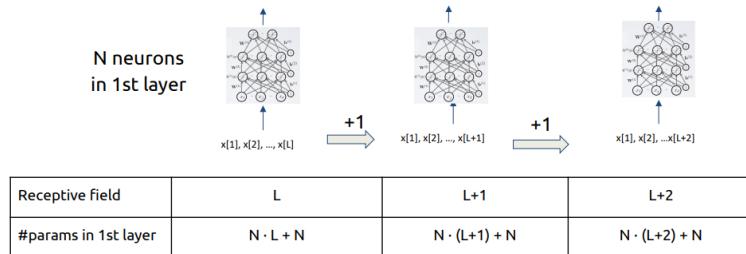
Usar una red neuronal para predecir un valor en el futuro de una secuencia. Un acercamiento sencillo es crear un MLP con la entrada de las muestras anteriores para generar una salida de predicción.

- Predict sample $x[t+1]$ knowing previous values $\{x[t], x[t-2], x[t-3], \dots, x[t-L]\}$



La ventana de los datos de entrada es siempre de un tamaño fijo en la MLP. La ventana se traslada a los L valores de muestras anteriores a $x[t + 1]$. Los problemas que surgen son:

- Cuando nos quedamos cortos de muestras al inicio de la serie temporal (generalmente se hace cero padding).
- El incremento del numero de parámetros al aumentar por una unidad el numero de muestras de input.

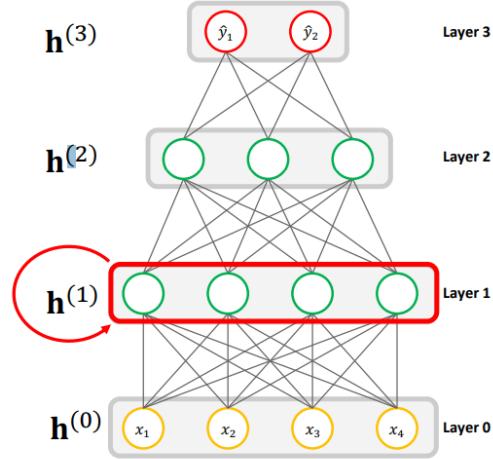


- Al incorporar las predicciones a nuevas predicciones en el tiempo (se puede hacer un modelo auto-regresivo en el que las salidas previas de la red se conviertan en inputs de la futura red).

3.1 Recurrent Neural Networks

Se puede hacer que una salida de una capa sea también usada para ser auto-regresiva. Usar la salida del instante anterior como entrada para la misma capa. Las RNN son un tipo de modelo de aprendizaje automático diseñado para trabajar con secuencias de datos de longitud variable sin causar un incremento en el numero de parametros de la red. A diferencia de las redes neuronales convencionales, las RNN tienen conexiones que permiten el procesamiento de información secuencial y temporal.

Las RNN son especialmente útiles para datos donde la relación entre los elementos depende del contexto o la secuencia en la que aparecen. Tienen la capacidad de recordar información pasada y utilizarla en la toma de decisiones presentes. Lo que las distingue de otras arquitecturas de redes neuronales es su capacidad para mantener y utilizar información previa sobre la secuencia de datos mientras procesan elementos posteriores



en la secuencia.

La principal característica de las RNN es su capacidad para mantener y utilizar información previa utilizando conexiones recurrentes dentro de la red. Esto significa que la salida en un paso de tiempo dado no solo depende de la entrada actual, sino también de la información que la red ha acumulado hasta ese momento.

Permiten adaptar al numero de elementos de input sin necesidad de hacer padding. No incrementa la complejidad del **receptive field**. En la practica si que se requiere añadir padding por motivos de implementación, para hacer que las secuencias de input tengan la misma longitud. El padding se coloca al inicio de la secuencia para asegurar que lo que importa de la secuencia (el contenido actual) este en las ultimas posiciones y el modelo pueda aprender a ignorar los elementos del padding.

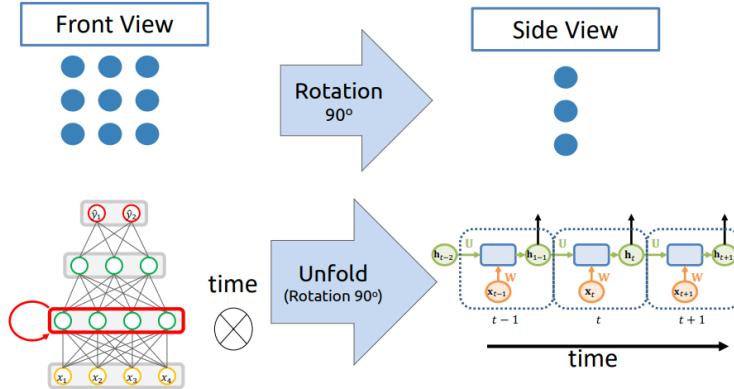
$$\mathbf{h}_t^{(l)} = g(\mathbf{W}^{(l)} \mathbf{x}_t^{(l)} + \mathbf{U}^{(l)} \mathbf{h}_{t-1}^{(l)} + \mathbf{b}^{(l)})$$

Aparece el subíndice t , con un aumento de dimensión en sentido de un índice temporal. Se aumenta el numero de parámetros de la capa. Se crean dos dimensiones en el forward pass; el paso por la red y la recurrencia en el tiempo de los resultados de los instantes.

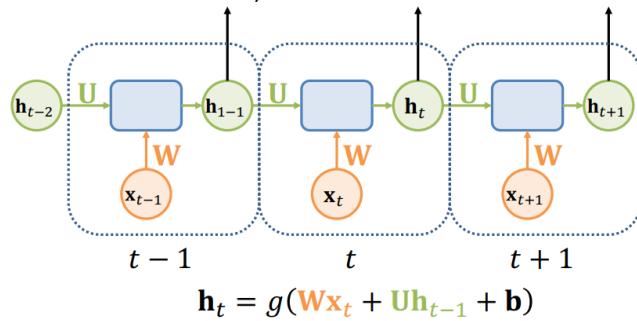
Se introduce igualmente una matriz de pesos $\mathbf{U}^{(l)}$ que corresponden a la salida de la misma capa en el instante anterior.

3.1.1 Forward and Backward passes thought time

La entrada es temporal, y cada instancia entra a través de la red. La \mathbf{U} y \mathbf{W} son constantes en el tiempo. La recurrencia es la misma para cualquier t . El número de las instancias es un hiperparámetro a concretar para determinar el nivel de recurrencia.



Como es recursivo ahora puedo tener una secuencia de datos de entrada y una secuencia de datos de salida. La longitud de entrada y la de salida no tienen porque ser iguales.



Una consecuencia de la recurrencia es que la memoria (recurrencia) a largo plazo de las RNN es difícil de mantener. Hay una recurrencia que al cabo de un tiempo, la contribución de $t - h$ de tiempos muy atrás $h \gg$ es casi mínima. Una solución a eso es usar **skip connections**.

Backpropagation: Para cada salida en el instante de tiempo t tenemos una \mathcal{L} y el error total es la suma de cada uno de los errores en cada instante $t = 0 \dots T$.

$$\mathcal{L} = \sum_{t=1}^T \mathcal{L}_t$$

y por lo tanto su derivada es la suma de las derivadas de cada instante t .

$$\frac{\partial \mathcal{L}}{\partial \theta} = \sum_{t=1}^T \frac{\partial \mathcal{L}_t}{\partial \theta}$$

donde θ es el conjunto de parámetros de la red ($\mathbf{W}, \mathbf{U}, \mathbf{b}$).

❖ In general (for \mathbf{U}, \mathbf{W} and \mathbf{b}), we can rewrite the gradient of each loss as:

$$\frac{\partial \mathcal{L}_t}{\partial \theta} = \frac{\partial \mathcal{L}_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial h_t} \sum_{k=1}^t \left(\prod_{j=k}^{t-1} \frac{\partial h_{j+1}}{\partial h_j} \right) \frac{\partial h_k}{\partial \theta}$$

❖ And the global gradient for a generic number of steps T as:

$$\frac{\partial \mathcal{L}}{\partial \theta} = \sum_{t=1}^T \frac{\partial \mathcal{L}_t}{\partial \theta} = \sum_{t=1}^T \frac{\partial \mathcal{L}_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial h_t} \sum_{k=1}^t \left(\prod_{j=k}^{t-1} \frac{\partial h_{j+1}}{\partial h_j} \right) \frac{\partial h_k}{\partial \theta}$$

El gradiente que tenemos que aplicar para actualizar los parámetros es el producto de estos gradientes. Lo que puede pasar es que tengamos problemas de **vanishing** o **exploding gradients**. El resultado es que casi no tengo actualización de los parámetros o que se actualicen demasiado.

Para solventar este problema, existen diferentes soluciones:

- **Truncated** fijar que el numero de instancias temporales que hacemos en el back-propagation sea menor a las que hacemos en el forward pass. Esto hace que el gradiente sea mas estable.
- **Gradient clipping** limitar el valor del gradiente. Si el gradiente es mayor que un valor, se limita a ese valor.

3.1.2 Long Short Term Memory (LSTM)

Las Long Short-Term Memory Networks (LSTM) son un tipo especializado de redes neuronales recurrentes (RNN) diseñadas para superar algunos de los desafíos de las RNNs tradicionales, como el problema de los gradientes que desaparecen o explotan durante el entrenamiento con secuencias largas.

La idea es que para intentar mejorar la memoria a largo plazo se introduce una nueva variable que es la **cell state** (\mathbf{c}_t). Estas variables tienen la capacidad de retener y olvidar información durante largos períodos de tiempo. Las LSTM logran esto mediante el uso de tres "puertas", las cuales cambian como se conserva la información pasada:

- Puerta de olvido (Forget Gate): Decide qué porcentaje de información almacenada en la celda de memoria debe ser olvidada o mantenida (mecanismo de limpieza optimizado). Utiliza una función sigmoid para generar valores entre 0 y 1 que multiplican la información actual en la celda de memoria para determinar qué información debe ser olvidada y qué información se mantendrá.

$$\mathbf{f}_t = \sigma(\mathbf{Wx}_t + \mathbf{Uh}_{t-1} + \mathbf{b}_f)$$


 $\mathbf{f}_t = \sigma(\mathbf{W}_f \cdot [\mathbf{x}_t; \mathbf{h}_{t-1}] + \mathbf{b}_f)$
↓
 Concatenate
(vstack)

Se concatena la entrada y la salida de la capa en el instante anterior para poder referirnos a una única matriz de pesos.

- Puerta de entrada (Input Gate): Decide qué porcentaje de la nueva información (la información de la instancia actual de tiempo \tilde{c}_t) se debe agregar a la celda de memoria. Esta puerta utiliza una combinación de funciones sigmoid y tangente hiperbólica para actualizar la celda de memoria con nueva información relevante.

Input Gate Layer:

$$\mathbf{i}_t = \sigma(\mathbf{Wi} \cdot [\mathbf{x}_t; \mathbf{h}_{t-1}] + \mathbf{bi})$$

New contribution to cell state:

$$\tilde{\mathbf{c}}_t = \tanh(\mathbf{W}_c \cdot [\mathbf{x}_t; \mathbf{h}_{t-1}] + \mathbf{b}_c)$$

Se crea una nueva contribución (\tilde{c}_t) a la **state cell**. \tilde{c}_t es la candidata a nuevo valor del estado de la celda. En cada paso de tiempo t , este valor se calcula considerando la información de entrada actual y el estado oculto previo, filtrado a través de la puerta de entrada (input gate).

- Puerta de salida (Output Gate): Decide qué porcentaje de información en la celda de memoria a corto plazo (h_{t-1}) y la entrada se utilizará para generar la salida en ese paso de tiempo. Esta puerta filtra la información de la celda de memoria utilizando una función sigmoid y la combina con la función tangente hiperbólica para producir la salida final.

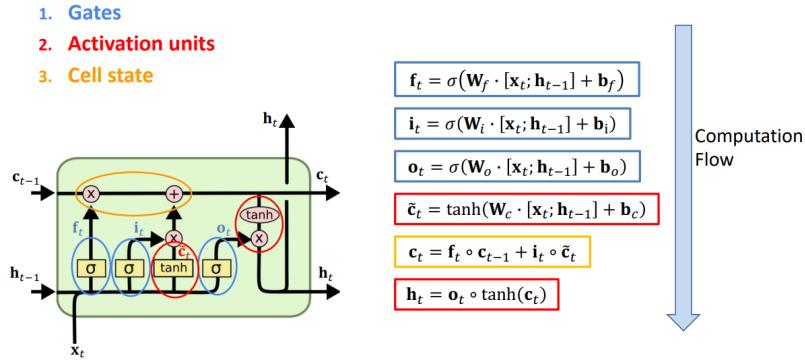
Output Gate Layer:

$$\mathbf{o}_t = \sigma(\mathbf{Wo} \cdot [\mathbf{x}_t; \mathbf{h}_{t-1}] + \mathbf{bo})$$

Output to next layer & timestep:

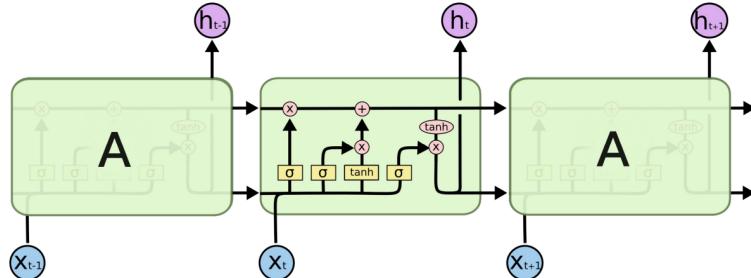
$$\mathbf{h}_t = \mathbf{o}_t \circ \tanh(\mathbf{c}_t)$$

Utilizar puertas para olvidar las cosas que queramos olvidar. Otra puerta para que los datos que entran actualicen el estado de la celda. Y otra puerta para que los datos que entran se usen para generar la salida.



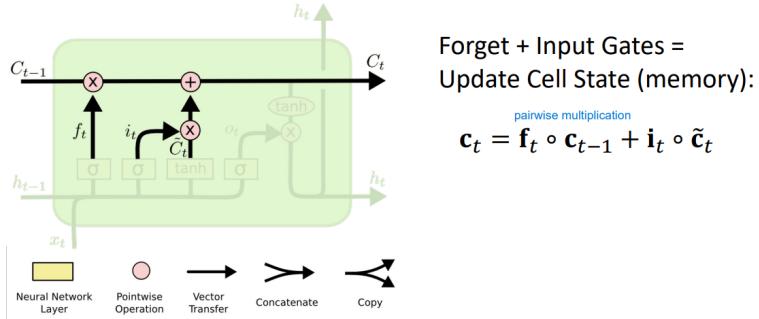
La falta de pesos o sesgos en c_{t-1} (sin contar las modificaciones causadas por la multiplicación y la suma), permiten a la memoria a largo plazo de la cell state circular a lo largo de las instancias sin causar que haya problemas con el gradiente como pasaba en las RNN simples.

Mientras que la memoria a corto plazo h_{t-1} si que sufre modificaciones mediante los parámetros en las puertas.



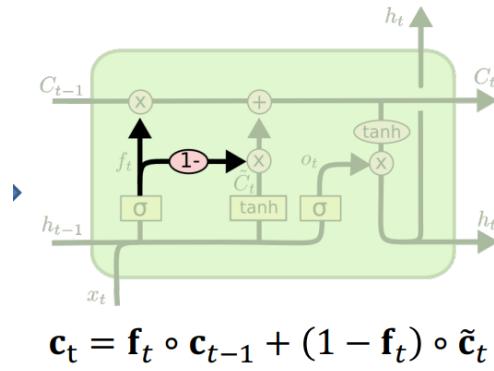
Todas las gates vienen con una etapa no lineal (sigmoid) a la salida para que los valores estén entre 0 y 1. Se multiplican a los datos de entrada para hacer este olvidar/recordar matemáticamente. Si la salida es 0, se olvida. Si la salida es 1, se recuerda. En comparación con las RNNs, las LSTM tienen 4 veces mas parámetros, que se deben a las tres puertas y a la activación de la cell state candidata.

La cell gate se va actualizando mediante las gates. **Forget gate** para olvidar, **input gate** para actualizar y **output gate** para generar la salida. Se tiene que aprender a olvidar a partir de las muestras de entrenamiento y la salida de la instancia anterior.



Variations

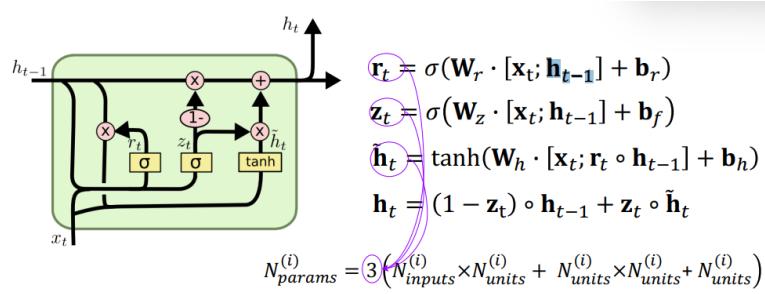
- Relacionar lo que se olvida y lo que se recuerda. Utilizar la puerta de olvidar y de input de manera conjunta.



- **Peephole connections** conectar la cell state con las gates. Implica tener mas parámetros.

$$\begin{aligned}\mathbf{f}_t &= \sigma(\mathbf{W}_f \cdot [\mathbf{x}_t; \mathbf{h}_{t-1}; \mathbf{c}_{t-1}] + \mathbf{b}_f) \\ \mathbf{i}_t &= \sigma(\mathbf{W}_i \cdot [\mathbf{x}_t; \mathbf{h}_{t-1}; \mathbf{c}_{t-1}] + \mathbf{b}_i) \\ \mathbf{o}_t &= \sigma(\mathbf{W}_o \cdot [\mathbf{x}_t; \mathbf{h}_{t-1}; \mathbf{c}_t] + \mathbf{b}_o)\end{aligned}$$

- **GRU (Gated Recurrent Unit)**: Las GRU son una versión simplificada de las LSTM que también abordan los problemas de desvanecimiento del gradiente y la memoria a largo plazo, pero con menos parámetros y estructura más simple. Tienen dos puertas principales: la puerta de reinicio (reset gate) y la puerta de actualización (update gate). Se simplifica el numero de gates. Se fusionan la forget y la input gate en una sola. Se fusionan la cell state y la hidden state en una sola.
 - **Reset Gate**: decide que información se va a olvidar de la hidden state anterior. Una simplificación es no utilizar el reset gate y olvidar toda la hidden state anterior.
 - **Update gate**: Decide que tanto se debe actualizar el hidden state anterior con el state candidato y controla que tanto del hidden state se debe olvidar.



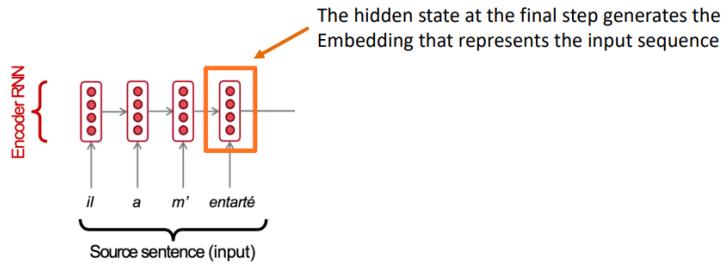
- **Tentative Hidden State:** se calcula la hidden state provisional.

Tanto las LSTM como las GRU no aseguran solventar el problema de vanishing/exploding gradients.

3.1.3 Neural Machine Translation NMT

Es una manera de hacer Machine Translation con una unica end-to-end red neuronal. La arquitectura de la red se llama **seq2seq** al requerir dos RNN.

Encoder para generar un embedding de la frase de entrada. El **decoder** genera la frase de salida a partir del embedding generado por el encoder.



La idea de usar esta estructura es que:

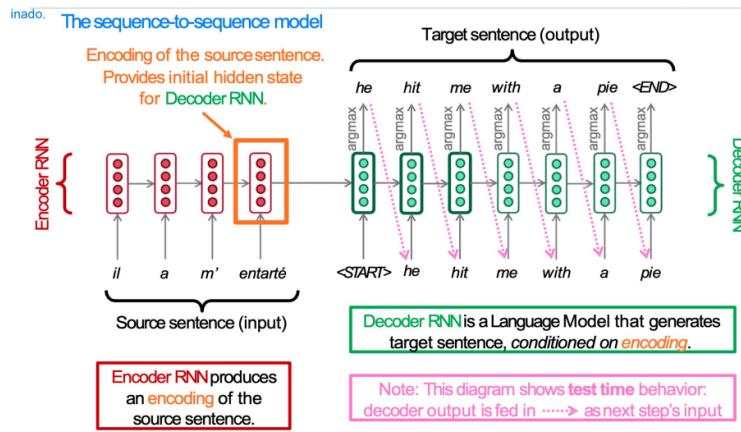
- Funciona mejor que los métodos anteriores al usar el contexto.
- Para entrenar solo se necesitan las parejas de las dos frases. End-to-end completo.
- Es poco interpretable como es común en las redes neuronales.
- Como es end-to-end, es difícil de controlar. No se puede controlar el proceso de traducción.

En estos modelos se pueden mezclar modalidades. Se puede insertar una imagen como input y obtener una frase de salida por ejemplo. En el caso de la imagen de entrada, el **encoder** no es necesario que sea RNN al ser solo una muestra de entrada.

3.2 Attention

En una red normal utilizas toda la entrada para estimar la salida, esto generalmente no es eficiente. No todos los píxeles importan igual para la salida. Lo mismo ocurre tanto en las imágenes como en los textos.

En la inferencia del test (no supervisada), la primera entrada del **decoder** es $iStart_\delta$ (token de identificación de inicio de la palabra). La palabra de salida del **decoder** se



genera mediante una maximización de qué palabra es la más probable en ese instante arg max. Cuando se que ha terminado (las longitudes de las frases no tiene porque coincidir y por lo tanto no son referencia de final) es cuando encuentro el token de finalización */End/*.

Cuando entreno, las flechas rosas no existen pues se tienen los labels.

Como afecta el tamaño de la secuencia de palabras de la frase a la longitud del embedding? No depende, pues el embedding es un vector de longitud fija que resulta de ser el input completo comprimido por el **encoder**. A veces no tiene sentido reducir toda la frase a una longitud fija (embedding) y puede ocasionar un cuello de botella al no dejar pasar toda la información necesaria de la frase al **decoder**. De aquí viene la idea de **attention**.

No se trata de que el **decoder** tenga que aprender a usar la información del instante final del **encoder**. Se trata de que el **decoder** aprenda a usar la información del **encoder** de la manera más eficiente posible. Escogiendo el encoding de la hidden state que más le interese y no obligatoriamente la del final.

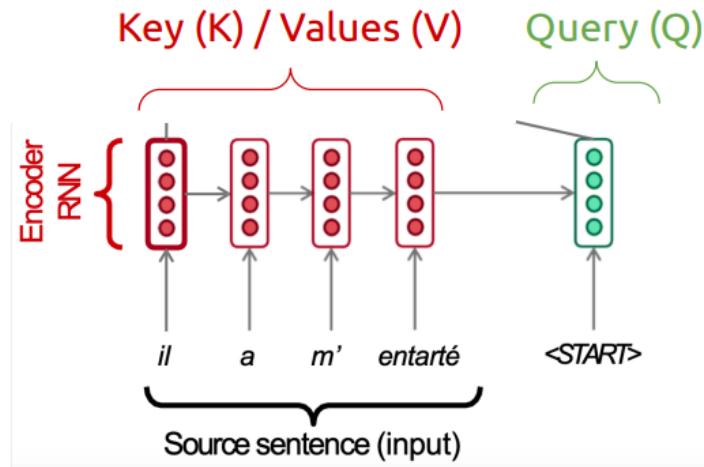
La **attention** es un mecanismo que ayuda al modelo a enfocarse en partes específicas de una secuencia de entrada durante el proceso de generación de una secuencia de salida. Este mecanismo es fundamental para mejorar la capacidad del modelo para manejar secuencias largas y capturar relaciones temporales complejas entre los elementos de las secuencias.

3.2.1 Key, Query, Value

En nuestro caso, los tres componentes son vectores. En el esquema seq2seq, las **keys** y las **values** corresponden ambas a los instantes del encoder, son los embedding en cada instante del encoder. La **query** es el hidden state del decoder en cada instante (los embeddings del decoder).

A cada instante de la salida, enfocamos al decoder a ciertas partes del encoder. En el decoder tiene un embedding de la palabra en el instante actual y se puede calcular un score entre la query y cada una de las keys para que nos diga que parecidas son.

1. **Query (Consulta):** La consulta es un vector que representa la posición o el elemento actual en la secuencia de salida. Durante el proceso de atención, la consulta se utiliza para calcular la similitud entre la posición actual en la secuencia de salida

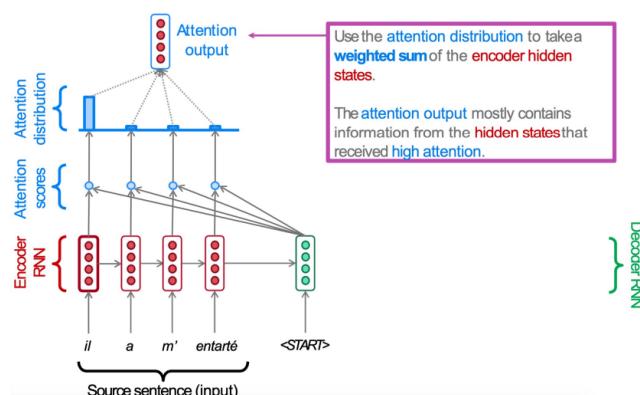


y cada una de las claves (keys) en la secuencia de entrada. Esta consulta se usa para determinar qué elementos de la secuencia de entrada son más relevantes para generar el siguiente elemento en la secuencia de salida.

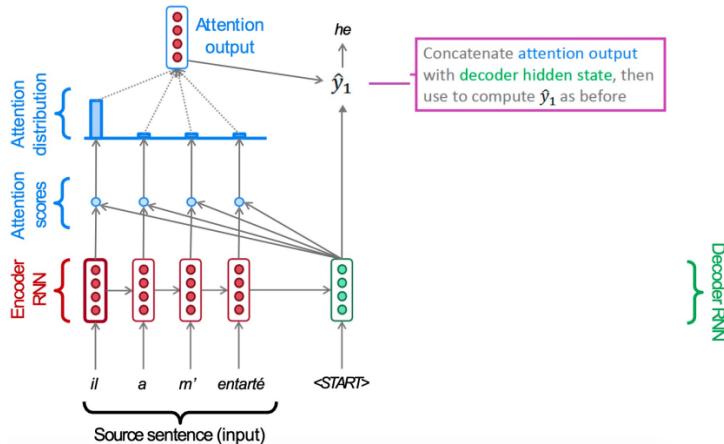
2. Key (Clave): La clave es un vector que representa cada posición o elemento en la secuencia de entrada. Las claves se utilizan para calcular la **similitud** con la consulta y determinar cuánta atención debe asignarse a cada elemento de la secuencia de entrada en relación con la consulta actual.
3. Value (Valor): El valor es un vector que contiene información específica asociada con cada clave en la secuencia de entrada. Estos valores se utilizan junto con los pesos de atención para calcular el vector de **contexto**, que es una combinación ponderada de los valores correspondientes a las claves relevantes.

La idea a alto nivel es que el encoder produzca una representación de la misma longitud que la entrada. Que se realiza mediante aplicar una softmax a los scores de atención.

Entonces el decoder, puede (mediante algun tipo de mecanismo de control) recibir como input este **contexto (attention output)** que consiste en una suma ponderada de las representaciones de entrada en cada instante.



Se concatena el **contexto** con la hidden state del decoder y se calcula la salida. Que se ocupa como entrada para el siguiente instante del decoder. Y así sucesivamente para cada instante del decoder.



En términos de ecuaciones:

- Las keys son los embedding del encoder. $\mathbf{h}_1 \dots \mathbf{h}_n$.
 - La query es el embedding del decoder. \mathbf{s}_t en el instante t .
- La atención implica calcular un score entre la query y cada una de las keys.
- **Attention score:** $\mathbf{e}_t = [e_{t,1}, \dots, e_{t,n}]$ donde $e_{t,i} = f(\mathbf{s}_t, \mathbf{h}_i)$.
 - **Attention distribution:** $\mathbf{a}_t = \text{softmax}(\mathbf{e}_t)$.
 - **Context:** $\mathbf{c}_t = \sum_{i=0}^{n-1} a_{t,i} \mathbf{h}_i$.
 - **Concateno con la query:** $\begin{bmatrix} \mathbf{s}_t \\ \mathbf{c}_t \end{bmatrix}$.

Me permite trabajar con distintas secuencias de longitud variable como entrada. Además:

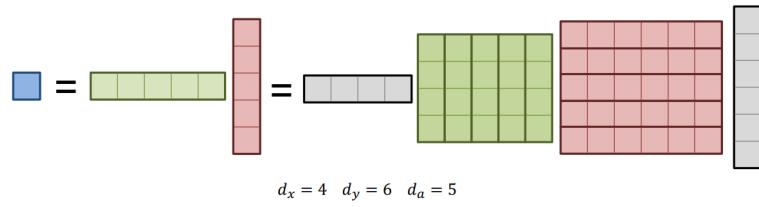
- Soluciona el problema del **bottle neck**, utiliza el contexto para fijarse en las cosas importantes.
- Tiene una interpretación mas clara (la distribución de contexto nos indica en que se fija la red en cada instante).
- También mejora el vanishing gradient porque en cada instante de la salida, se fija en una salida. Aquí como nos podemos fijar, mejora el long term memory y por tanto el vanishing gradient.

3.2.2 Attention Mechanisms

- Si ya nos podemos fijar en lo que queremos, las RNN ya no son necesarias. Podemos utilizar cualquier tipo de red.
- **Self-attention:** en vez de utilizar el encoder para fijarnos en el decoder, nos fijamos en el encoder para fijarnos en el encoder. Si aplicamos atención entre las palabras de una frase, podemos agregar contexto a cada palabra. Tiene sentido en lugar de usar los embedding de la entrada independientemente, utilizar la información de las otras palabras de la frase.

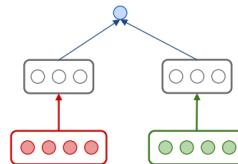
Dot product para calcular el score de atención. El problema es que la dimensión de las keys tienen que ser la misma que la de las queries. Generalmente lo que se hace es utilizar matrices para igualar las longitudes; $\mathbf{W}_k \in \mathbb{R}^{d_a \times d_x}$ y $\mathbf{W}_q \in \mathbb{R}^{d_a \times d_y}$. Son matrices de proyección que se entrenan junto con el resto de parámetros. Son iguales para todas las \mathbf{x} y \mathbf{y} . d_a es la dimensión de la atención (hyperparametro).

$$e_{ij} = \alpha(\mathbf{x}_i, \mathbf{y}_j) = \mathbf{k}_i^T \mathbf{q}_j = \mathbf{x}_i^T \mathbf{W}_k^T \mathbf{W}_q \mathbf{y}_j$$



Additive(concat attention) es una forma de calcular el score de atención. Se concatenan las keys y las queries y se aplica una red neuronal. Normalmente se utiliza una red neuronal de una capa y sin sesgo.

$$e_{ij} = \alpha(\mathbf{x}_i, \mathbf{y}_j) = \mathbf{w}_a^T \tanh([\mathbf{W}_k \mathbf{x}_i + \mathbf{W}_q \mathbf{y}_j]) = \mathbf{w}_a^T \tanh([\mathbf{W}_k \quad \mathbf{W}_q] [\mathbf{x}_i \quad \mathbf{y}_j])$$



$$\mathbf{k}_i = \mathbf{W}_k \mathbf{h}_i$$

$$\mathbf{q}_t = \mathbf{W}_q \mathbf{s}_t$$

y el score de atención es:

$$e_{t,i} = \mathbf{q}_t^T \mathbf{k}_i$$

$$\mathbf{e}_t = \mathbf{K} \mathbf{q}_t$$

donde $\mathbf{K} = [\mathbf{k}_1, \dots, \mathbf{k}_n] \in \mathbb{R}^{d_a \times n}$.

$$\mathbf{E} = \mathbf{K}^T \mathbf{Q}_t$$

donde $\mathbf{Q}_t = [\mathbf{q}_1, \dots, \mathbf{q}_n] = \mathbf{W}_q \mathbf{Y} \in \mathbb{R}^{d_a \times n}$.

$$\mathbf{A}_t = \text{softmax}_{\text{col}}(\mathbf{E})$$

$$\mathbf{C}_t = \mathbf{V} \mathbf{A} = \mathbf{W}_v \mathbf{X} \mathbf{A}$$

Multi-head: Una única etapa de atención es mejorable. La idea es que la atención se puede aplicar en varias etapas. Varias cabezas de atención, cada una con sus matrices de protección y distribuciones y luego se combinan todas.

En la práctica, la combinación de la serie de queries, keys y values puede que sea útil para extraer y combinar conocimiento de diferente comportamiento del mismo mecanismo de atención. Por ejemplo, una cabeza de atención puede ser buena para capturar la dependencia de largo alcance, mientras que otra puede ser útil para capturar la dependencia local. Por lo que puede ser beneficioso permitir que el mecanismo de atención use conjuntamente diferentes subespacios de representación de queries, keys y values.

Para esto, se pueden utilizar múltiples cabezas de atención independientemente.

3.3 Generative Networks

3.3.1 Unsupervised Learning

Solo tenemos los datos, sin los labels. La aplicación es aprender alguna estructura de esos datos, su función de distribución de probabilidad (para por ejemplo hacer clustering, generar datos, etc). También podemos aprender representaciones de los datos que tengan algún sentido físico/geométrico que los datos originales.

3.3.2 Generative Models

Los modelos generativos son un tipo de modelo en el campo del aprendizaje automático que tienen como objetivo principal capturar la distribución de los datos subyacentes para poder generar muestras nuevas que sean similares a los datos de entrenamiento. Estos modelos son capaces de crear datos nuevos que imitan las características estadísticas de los datos originales.

Aprenden la distribución de probabilidad de los datos. A diferencia del mapeo entre los datos y los labels de los modelos discriminativos, aquí se aprende la distribución de probabilidad de los datos.

Ya no aprendemos una probabilidad condicionada $p(y|x)$, sino que directamente se aprende la probabilidad de los datos $p(\mathbf{x})$.

Se tiene que aprender la distribución de probabilidad de los datos de entrada sin condicionar el aprendizaje a los datos de la base de datos.

3.3.3 Explicit Density Models

Tractable Density

En lugar que sea una función de densidad de probabilidad a todo el espacio que tiene probabilidades altas en pocas zonas (por ejemplo, de todas las imágenes posibles, hay pocas que sean de caras), se trata de que la función de densidad de probabilidad sea fácil de calcular. En su lugar se simplifica con la probabilidad condicional. Usar la regla de la cadena para descomponer la likelihood de una muestra en un producto de distribuciones unidimensionales.

Calcular estas probabilidades condicionales tampoco es fácil. Se puede hacer con una red neuronal para estimar esta probabilidad (seq2seq aunque es lento).

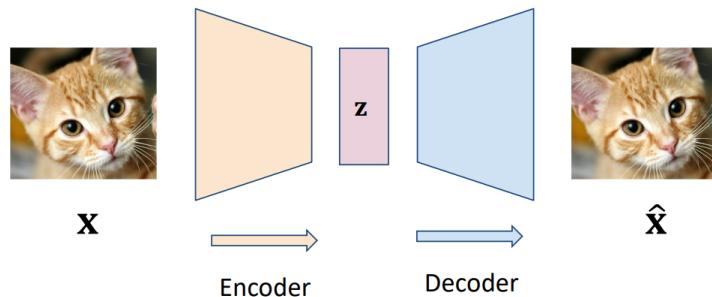
$$p(\mathbf{x}) = \prod_{i=1}^n p(\mathbf{x}_i | \mathbf{x}_1, \dots, \mathbf{x}_{i-1})$$

Likelihood of sample \mathbf{x}
(sentence, image) Probability of i^{th} (word,char,pixel) value
given all previous (words,chars,pixels)

Variational Autoencoders

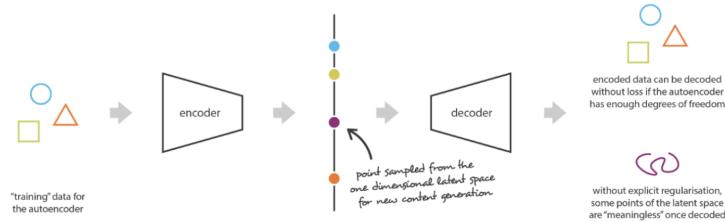
Los VAEs se utilizan para modelar distribuciones de datos complejas y de alta dimensión. Estos modelos constan de dos componentes principales: el codificador (encoder) y el decodificador (decoder). Es auto porque la entrada y la salida son de las mismas dimensiones. Queremos encontrar un embedding \mathbf{z} para que la entrada y la salida sean iguales. *La ventaja es que no se necesitan labels, pues la muestra de entrada hace de ground truth.*

- Codificador (Encoder): Transforma los datos de entrada en una representación en un espacio latente (o espacio de variables latentes) de dimensionalidad reducida \mathbf{z} .
- Decodificador (Decoder): Toma muestras del espacio latente \mathbf{z} y las reconstruye en el espacio original de los datos. El decodificador trata de generar datos que se asemejen a los datos de entrada.

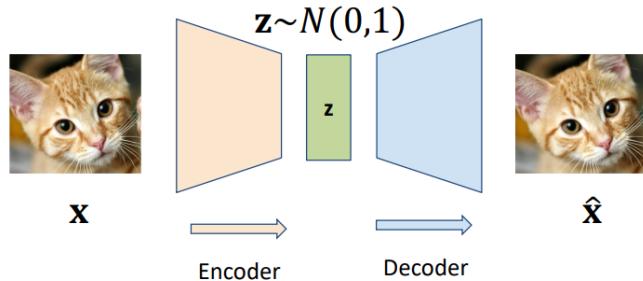


\mathbf{z} es un espacio (espacio de features) que tiene una dimensión bastante menor que a la de la entrada. Se busca que el embedding sea lo suficientemente rico como para poder volver a generar la imagen de entrada a través del decodificador.

A la hora de generar nuevos datos, lo que se busca es generar (imágenes de gatos, por ejemplo) que no hayan estado presentes en el entrenamiento. Para ello no se utiliza el encoder, sino que se selecciona una nueva \mathbf{z} (que no sea igual a las del entrenamiento). Esto no funciona, el autoencoder aprende a memorizar las \mathbf{z} (no tenemos restricciones en el espacio latente). No se ha regularizado para obtener imágenes nuevas al hacer variaciones de las \mathbf{z} . Se tiene que aplicar algún tipo de regularización en el espacio latente de las \mathbf{z} .



Regularización del espacio latente para que las \mathbf{z} se distribuyan de alguna manera que tengan sentido y que el espacio latente tenga buenas propiedades que permitan el proceso generativo. Distribución gaussiana $N(0, 1)$, porque es simple y se puede mapear a cualquier función de densidad más compleja (mediante redes neuronales). Tendremos un espacio latente de distribución gaussiana y el decoder mapeara la distribución gaussiana a la distribución de los datos.



El problema es que estimar la distribución de los datos para todas las posibles combinaciones de \mathbf{z} es muy difícil. Computacionalmente muy costoso. El mapeo entre imágenes de entrenamiento a \mathbf{z} es también costoso.

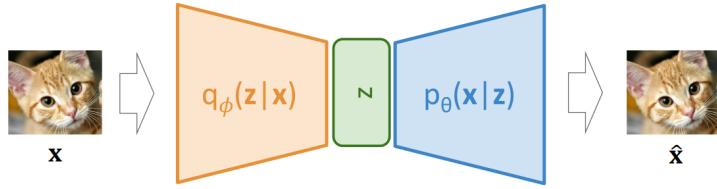
$$p_{\theta}(\mathbf{x}) = \int p_{\theta}(\mathbf{z}) p_{\theta}(\mathbf{x}|\mathbf{z}) d\mathbf{z}$$

Simple Gaussian prior

Decoder Neural Network

Intractable to compute $p(\mathbf{x}|\mathbf{z})$ for every \mathbf{z}

Se usan las redes para estimarlas. Decoder nos estima la función de probabilidad de densidad $p(\mathbf{x}|\mathbf{z})$ y el encoder nos estima la función de densidad $q(\mathbf{z}|\mathbf{x})$.



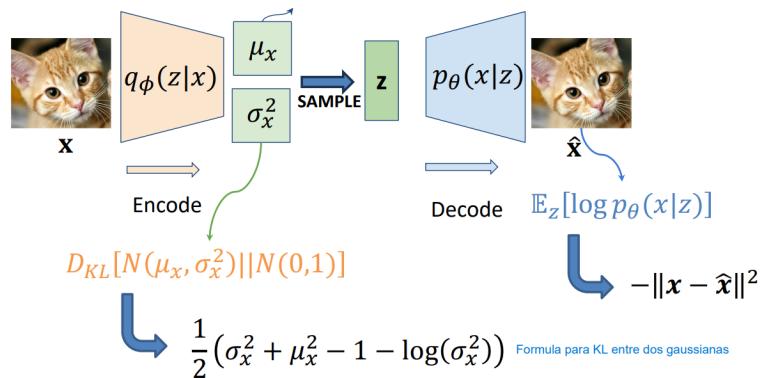
La regularización permite entrenar el encoder de manera que el mapeo quede como gaussiana.

$$\mathcal{L}(x, \theta, \phi) = D_{KL} (q_\phi(z|x) || p_\theta(z)) - \mathbb{E}_z [\log p_\theta(x|z)]$$

**Regularization of our latent representation
→ NEURAL ENCODER projects over prior.**

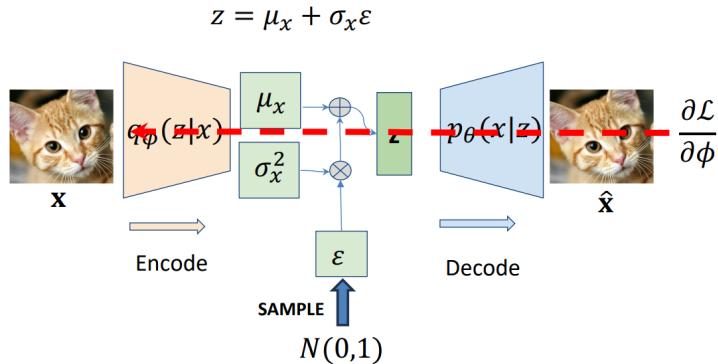
Reconstruction loss of our data given latent space → NEURAL DECODER reconstruction loss!

Con lo que a la salida del encoder, tenemos la media y la varianza de la imagen de entrada. Esta media y varianza queremos que sea una distribución gaussiana con media 0 y varianza 1.



A partir de estos parámetros, se realiza un muestreo estocástico para obtener puntos en el espacio latente. En el backpropagation, se rompe el gradiente en el punto del **sample**.

Se utiliza la llamada "reparametrización" donde se toman muestras aleatorias de una distribución gaussiana estándar y se transforman utilizando la media y la desviación estándar generadas por el encoder. Esto permite que el modelo sea diferenciable y entrenable mediante técnicas de gradiente descendente.



En la generación de imágenes, no se utiliza el encoder, sino que \mathbf{z} es ahora una distribución gaussiana y el decoder genera nuevas imágenes de salida.

3.3.4 Implicit Density Models

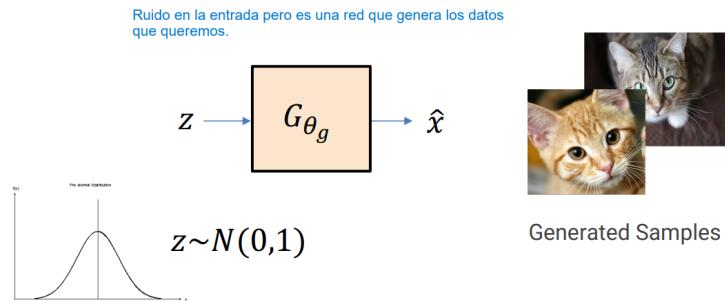
Los "Implicit Density Models" (Modelos de Densidad Implícita) son un tipo de modelo en el campo del aprendizaje automático que buscan aprender y representar la distribución de los datos sin necesariamente modelar explícitamente la función de densidad de probabilidad.

GANs (Generative Adversarial Networks)

Las GANs, o Generative Adversarial Networks en inglés, son un tipo de modelo de aprendizaje automático que se utiliza para generar datos nuevos que sean similares a un conjunto de datos de entrenamiento dado.

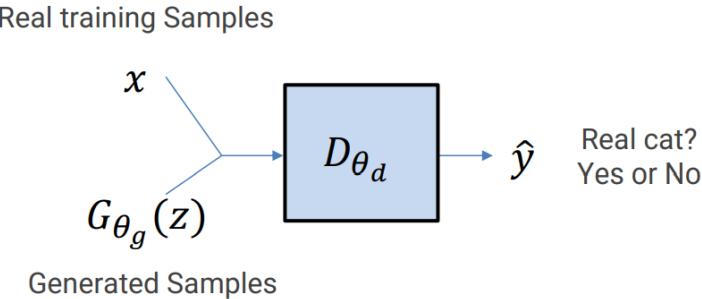
La estructura de una GAN consiste en dos redes neuronales que se entrenan simultáneamente en un proceso de competencia, creando una dinámica de "juego de adversario". Estas dos redes son:

1. **Generador:** Esta red tiene como objetivo crear datos sintéticos que sean lo más cercanos posible a los datos reales del conjunto de entrenamiento (que sigan la distribución de probabilidad de los datos de entrenamiento). Toma muestras de ruido aleatorio como entrada y las transforma en ejemplos que se asemejan a los datos reales. Busca minimizar la probabilidad de que la red discriminativa acierte.



2. **Discriminador:** Esta red tiene como objetivo distinguir entre datos reales y datos generados por el generador. Se entrena para clasificar correctamente si una

muestra de datos proviene del conjunto real de entrenamiento o si fue creada por el generador. Busca maximizar la probabilidad de acertar si el dato es real o no.



El proceso de entrenamiento de una GAN implica que el generador y el discriminador se mejoren mutuamente a través de la competencia. La red generativa intenta engañar a la red discriminativa y la red discriminativa intenta distinguir entre los datos reales y los generados.

Discriminator outputs likelihood in (0,1)

$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log \underbrace{D_{\theta_d}(x)}_{\text{Discriminator output for real data}} + \mathbb{E}_{z \sim p(z)} \log \underbrace{(1 - D_{\theta_d}(G_{\theta_g}(z)))}_{\text{Discriminator output for generated fake data}} \right]$$

- Discriminator wants to **maximize the objective** such that $D_{\theta_d}(x)$ is close to 1 (real) and $D_{\theta_d}(G_{\theta_g}(z))$ is close to 0 (fake).
- Generator wants to **minimize the objective** such that $D_{\theta_d}(G_{\theta_g}(z))$ is close to 1 (discriminator is fooled into thinking generated $G_{\theta_g}(z)$ is real)

Durante el entrenamiento, el generador busca mejorar su capacidad para generar datos que engañen al discriminador, mientras que el discriminador busca mejorar su capacidad para distinguir entre datos reales y generados. Esta competencia continúa iterativamente hasta que el generador produce datos que son indistinguibles de los datos reales para el discriminador.

