Aprenentatge Automàtic 2

Carlos Arbonés and Juan P. Zaldivar

GCED, UPC.

Apunts.

Contents

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** $(x_1, x_2, \ldots, x_M)$ 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:

- **Regressión**: la variable y es **continua**. Por ejemplo si queremos predecir el precio de una casa.
- **Classificación**: la variable y es **discreta**. Por ejemplo si queremos predecir si el animal en una fotografia 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.

$$\hat{y} = \mathbf{w}^T\mathbf{x} + b = w_1x_1 + w_2x_2 + \cdots + w_Mx_M + b$$

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 activicaión.

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.

$$\hat{y} = g(a) = g(\mathbf{w}^T\mathbf{x} + b) = g(w_1x_1 + w_2x_2 + \cdots + w_Mx_M + b)$$

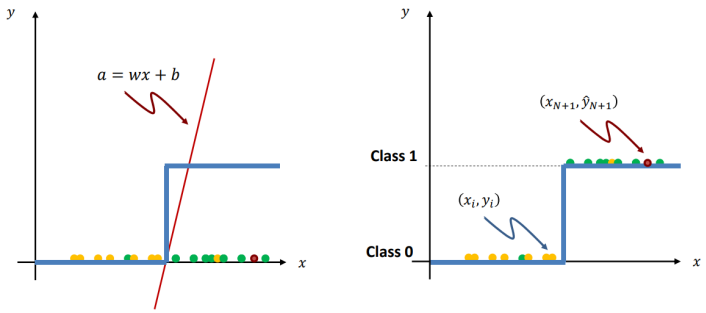


Fig. 1.1: *Ejemplo clasificaión 1-D*

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.

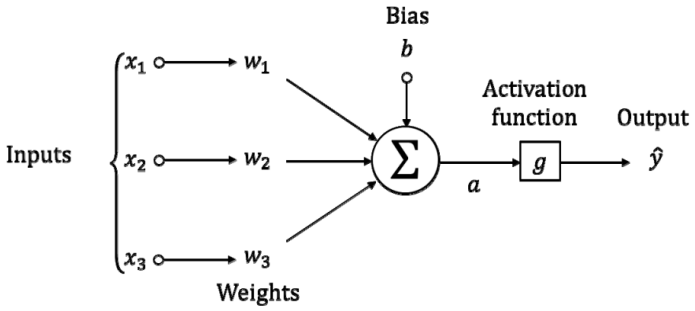


Fig. 1.2: *Esquema de Perceptron*

$$\hat{y} = g(a) = g(\mathbf{w}^T\mathbf{x} + b) = g(w_1x_1 + w_2x_2 + \cdots + w_Mx_M + b)$$

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 **explotar 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.

Preceptron 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.

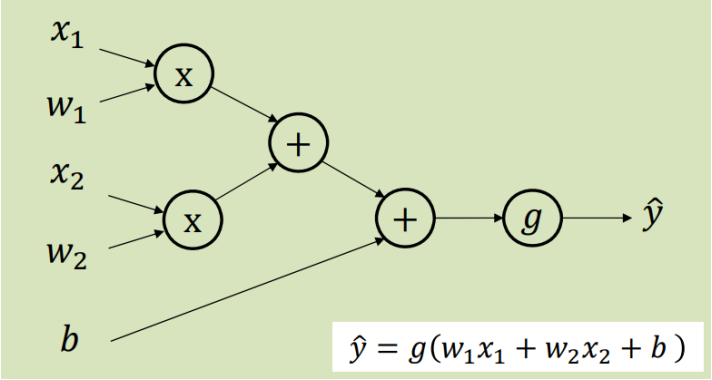


Fig. 1.3: *Ejemplo de grafo computacional de un perceptron con 2 inputs ($M = 2$)*

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.

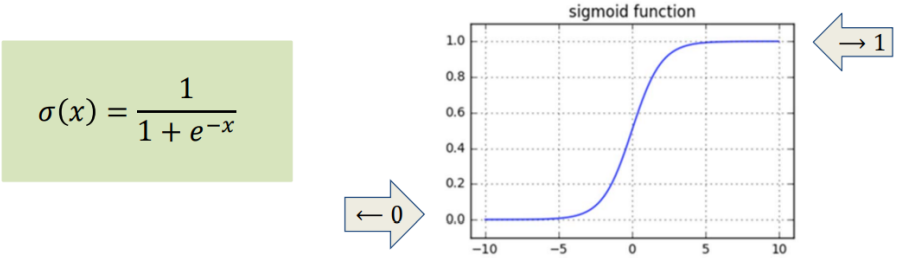


Fig. 1.4: *Definición y plot de la función sigmoide*

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.

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

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

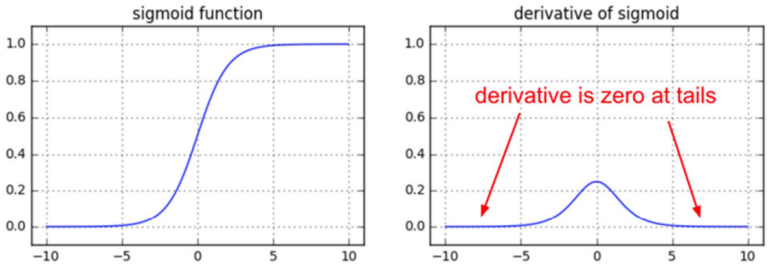


Fig. 1.5: *Derivada de la función sigmoide*

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. A si 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.

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

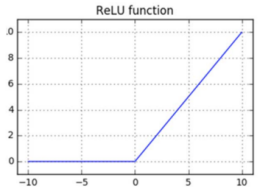
También hay que tener en cuenta que se dibuja en 1-D pero esta idea se puede extrapolar en múltiples dimensiones.

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

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 hyperbolica**. 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 mas se usa es la **ReLU(x)**

$$ReLU(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases} \qquad \frac{\partial ReLU(x)}{\partial x} = u(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x < 0 \end{cases}$$



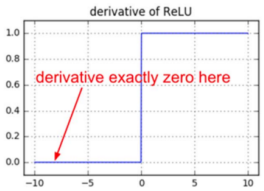


Fig. 1.6: *Definición de la función de activación ReLU*

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 probelmas 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.

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

Neural Network

Si hemos decidido concatenar perceptrones lo podemos hacer con la cantidad que queremos y aparecen redes mucho mas 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.

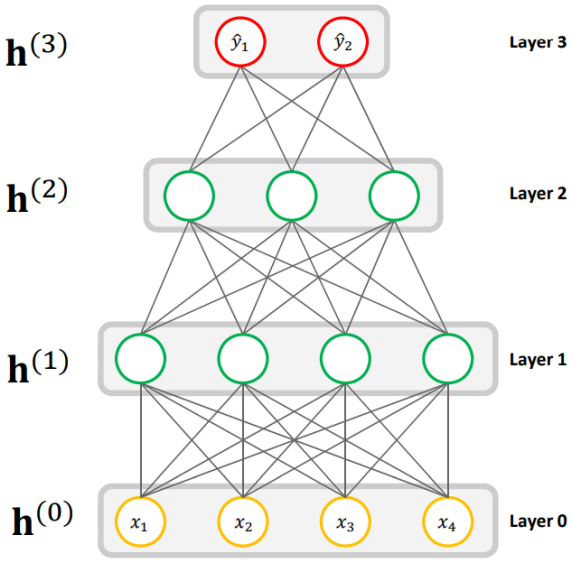


Fig. 1.7: *Neural Network example*

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**: normalemnte 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

[1] Una 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 **reglada 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 debría 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** $\frac{\partial \mathcal{L}_{\mathcal{SE}}}{\partial \theta}$. 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 $\mathcal{L}_{\mathcal{SE}}$.

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

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.

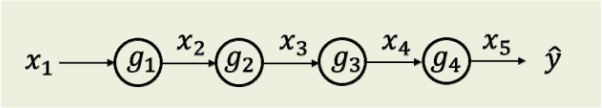


Fig. 1.8: *Input/Output relations*

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

Se tiene que ir iterando el coeficiente de la cadena de derivación. *Nota:* $g(.)$ *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

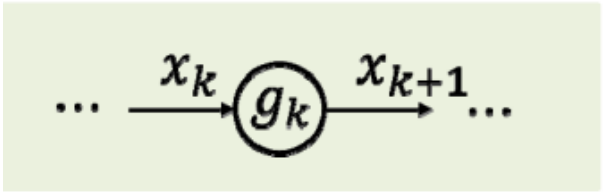


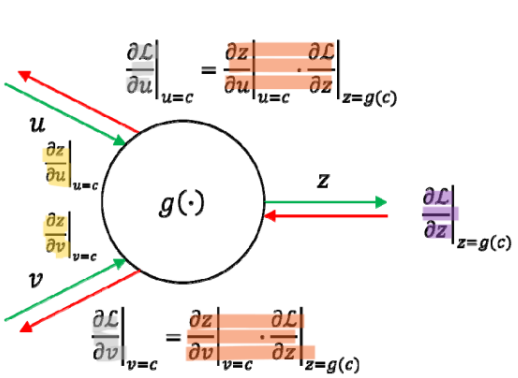
Fig. 1.9: *Node Input/Output*

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.
$$g_k^{'}(c) = \frac{\partial x_{k+1}}{\partial x_k}\Big|_c$$
3. Multiplicar $g_k^{'}(c)$ por el gradiente propagado.

$$\boxed{\frac{\partial \hat{y}}{\partial x_k}\Big|_{x_k=c} = g_k^{'}(c) \cdot \frac{\partial \hat{y}}{\partial x_{k+1}}\Big|_{x_{k+1}=g_k(c)}}$$

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.



(a) Node type

```
Class MultiplyGate(object):
    def forward (u,v):
        z = u * v
        self.dz_du = v
        self.dz_dv = u
        return z
    def backward (dz):
        du = self.dz_du * dz
        dv = self.dz_dv * dz
        return [du, dv]
```

(b) Pseudocode:

Fig. 1.10

Cuando se propaga sobre una red, se guarda toda la información de las derivadas en memoria. Hay un limite 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 *minibatchs* (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 perdida disminuirá y viceversa para un gradiente negativo.

Some considreations

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 terminos, el gradiente de uno es el termino del otro, con lo que se puede entender como un **switcher**.
- **Max node**: Como solo se selecciona un input a traves de la fucnion max, el nodo actua 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 que se enfoca la red para tomar una decisión, así podemos determinar partes de la red y de sus parámetros.

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

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.

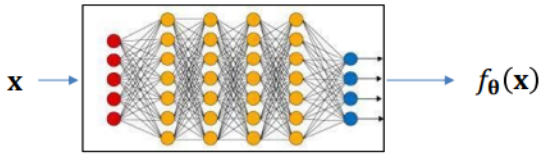


Fig. 1.11: 242 parámetros. $(7 \cdot 6 + 7 \cdot 8 + 7 \cdot 8 + 7 \cdot 8 + 4 \cdot 8)$

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 decremento hasta 0. Esto combinado con la técnica de *backpropagation*, nos permitiría conseguir este decremento 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 foward 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 **riesgo empírico**.

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

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

Cuando se procesan valores pequeños de pérdida (cercanos 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).

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

1.3.6 Losses in Classification Problems

En problemas de clasificación, la red predice variables categoricas.

- 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 aqui 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).

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

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

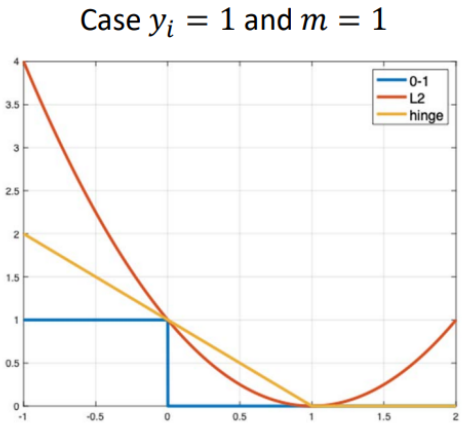


Fig. 1.12: Loss Comparison

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 parecernos (**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 clasification

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

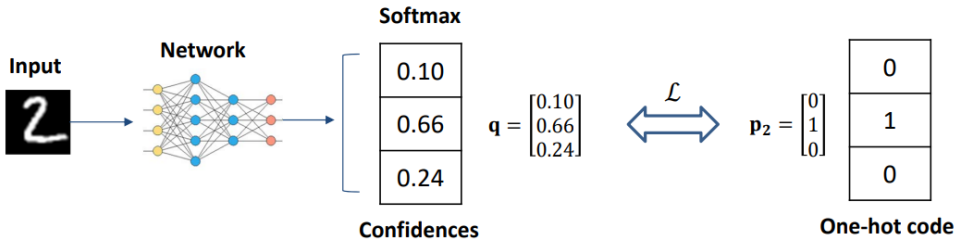


Fig. 1.13: Cross-entropy

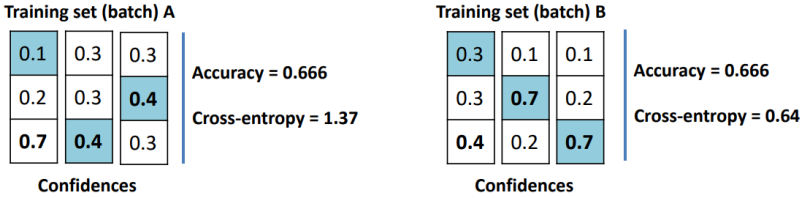
Para el caso de **Cross Entropy:** La distribución del output corresponde a la aplicación del SoftMax, mientras que la distribución del modelo de referencia se realiza con One-hot encoding, donde la longitud del vector es el número de clases diferentes y únicamente es diferente de 0 en la posición de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

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

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.

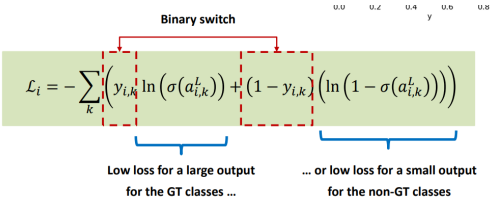
1.3.7 Multi-label

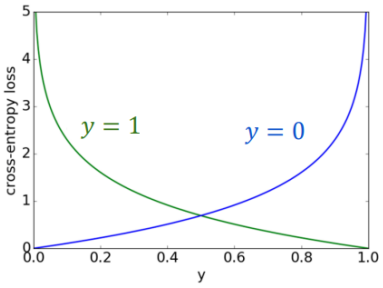
El enfoque multi-label se utiliza cuando se deben asignar múltiples etiquetas a una muestra, lo que es común en el contexto de imágenes, donde una imagen puede contener varios objetos o características de interés.

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.

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

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.

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

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:

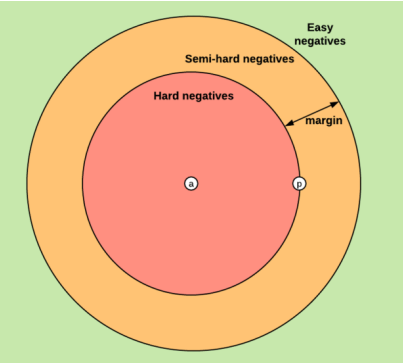
$$d^2(f_\theta(x_a), f_\theta(x_p)) - d^2(f_\theta(x_a), f_\theta(x_n)) > 0$$

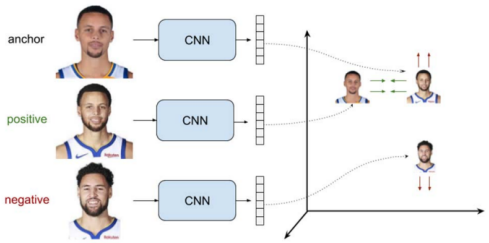
- Las parejas relativas están más cerca que un cierto margen:

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

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

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 tripetes sean de tipo *easy negatives*.

A medida que avanza el entrenamiento, más parejas/tripletes 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/tripletes a medida que progresa 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 lo datos (**batch**) de tamaño N y se obtienen las funciones de perdida para cada paso de la observación del **batch**. Los gradientes se van almacenando en la propagación hacia delante. **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 mas lenta.

1.4.2 Stochastic Gradient Descent

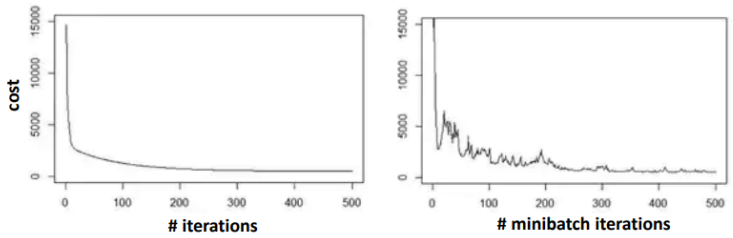
Stochastic Gradient Descent: En contraste, Stochastic Gradient Descent (**SGD**) opera de manera diferente. En SGD, se optimiza el modelo para cada punto de datos

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

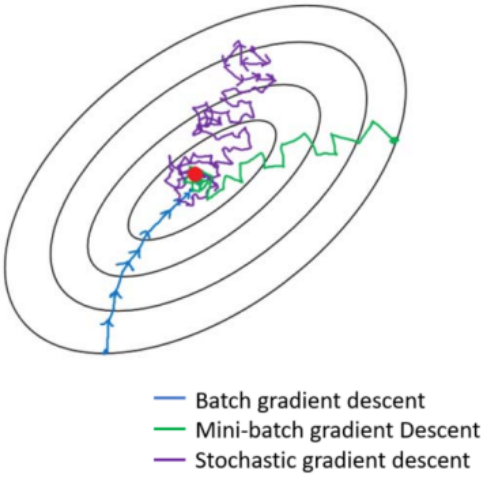
Caso intermedio. No cogemos un bloque de datos tan grande como coger todos (N). Sino que se coge un conjunto de datos que nos quepa en memoria y que nos permite confiar en la robustez de las aproximaciones para avanzar con confianza. Normalmente el tamaño del **minibatch** es una potencia de 2.



Batch vs mini-batch gradient descent

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).

$$\hat{y} = f_\theta(x) = \tanh(w_2 \tanh(w_1 x))$$

1.4.5 Local minima and saddle points

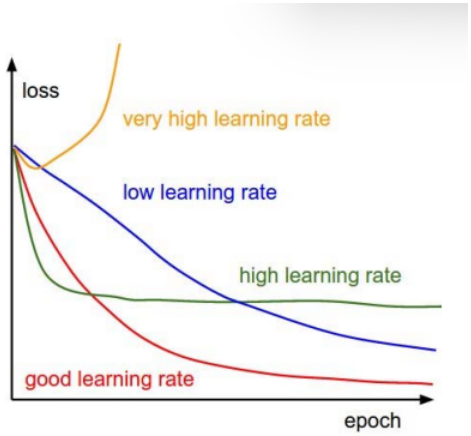
Hay muchas configuraciones que conllevan a diferentes mínimos locales. Lo que se tiene que intentar es que la red no tenga tendencia a soluciones problemáticas.

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 Inizialization

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 **sadlle 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 backpropagaton, los gradientes para los pesos serán todos iguales, lo que impide que la red pueda aprender patrones distintivos o representaciones utiles 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 = randn(n)/sqrt(n)`
    - **He**: para activaciones de tipo *ReLu*. `w = randn(n)·sqrt(2/n)`

1.4.8 Algorithms

Usando el gardient 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*.

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

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.

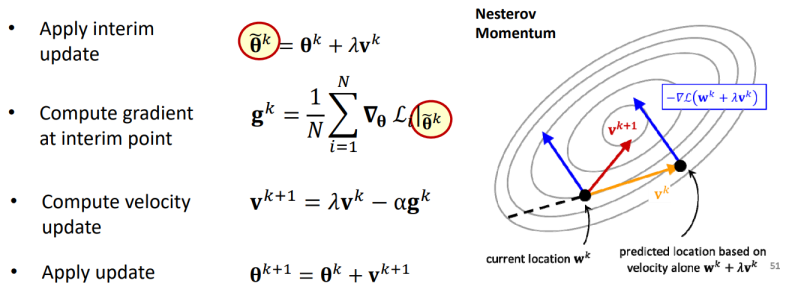
$$v^k = \alpha(-\lambda^{k-1}\nabla_\theta \mathcal{L}_0 - \lambda^{k-2}\nabla_\theta \mathcal{L}_1 - ... - \lambda \nabla_\theta \mathcal{L}_{k-1} - \nabla_\theta \mathcal{L}_k) = -\alpha \sum_{\tau=k-1}^{0} \lambda^\tau \nabla_\theta \mathcal{L}_{k-\tau-1}$$

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.

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

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.



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 mas 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.

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

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.

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

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.

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

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

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

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

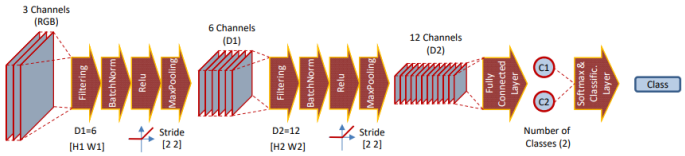
$$\boxed{\theta^{k+1} = \theta^k - \alpha \left(\epsilon \mathrm{I} + \mathrm{diag}(G^k)\right)^{\frac{1}{2}} \nabla_\theta \mathcal{L}_i}$$

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.

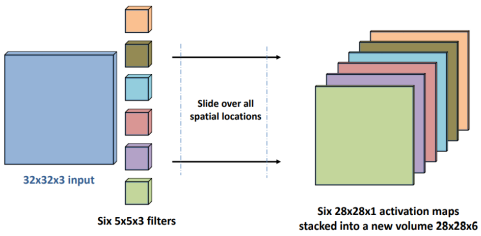
Conectividad Local

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.

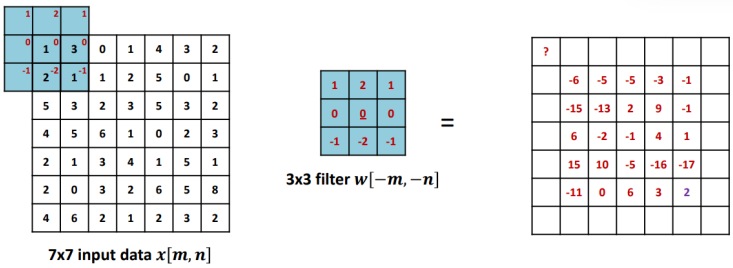
Aprendizaje de extremo a extremo (**End-to-End Learning**): La definición de la arquitectura permite que el sistema seleccione los pesos de cada una de las capas de filtrado. El propio proceso de filtrado realiza un **entrenamiento de extremo a extremo**.

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"**.



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:

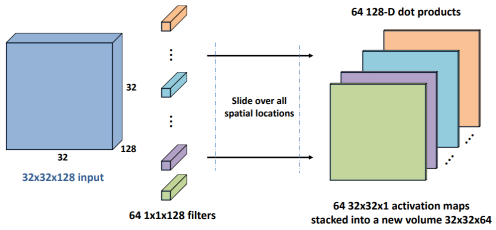
$$\text{Sin relleno: } \frac{N - F}{S} + 1$$

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

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 gausiana. 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 dados, 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 tecniacas similares a las de codificación (en la convolucion) pero incrementanto la resolucion de los resultados, lo cual requerirá tambien un aprendizaje de optimizacion.

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

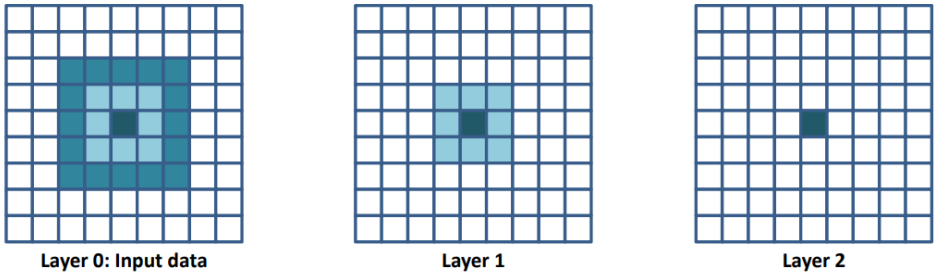
Necesitamos tener un conujnto de *ground turth images* para entrenar la red. Es decir, necesitamos imágenes ya segmentadas para poder comparar.

Receptive field

El **receptieve 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 calculo 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,



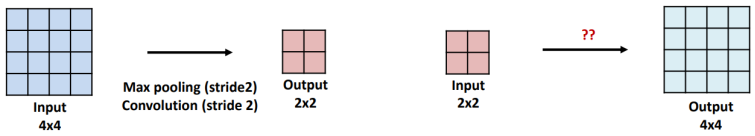
si utilizamos dos capas de convolución con filtros de 3x3, tendremos 20 parámetros a optimizar en total (9+9) y los dos sesgos (uno en cada capa) y mantendremos el mismo campo visual (25 píxeles).

Esta manera de reducir el tamaño del filtro y aumentar las capas de convoluciñon han resultado ofreccer 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

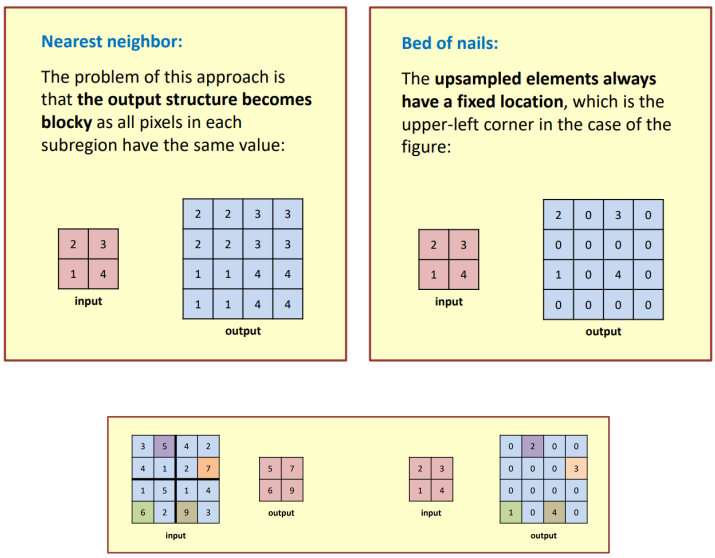
Mientras que en las capas de convoluciñon y Max-Pooling la resolución se puede ver afectada, en la descodificación se busca contrarestar esta afectacón.



Unpooling

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

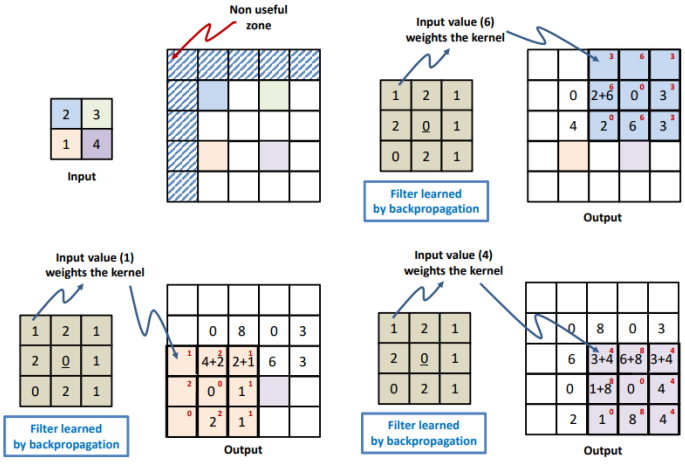
- **Nearest Neighboor**: Repetir el valor para rehacer el Max-Pooling. El problema de esta implementación es que la estructura del output se vuelve bloqueada (*blocky*), ya que todos los elementos en cada subregion tienen el mismo valor.
- **Bed of nails:** Rellenar con 0 y posicionar el valor diferente 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 encontrada el máximo anteriormente (mediante el almacenamiento de los indices de posicion). Nos ayuda a preservar un poco la información que teníamos en la imgen original.



Transpose Convolution

Los acercamientos anteriores proponen una operacion fija, minetras que las convolucion transpuesta propone aprovechar el aprendizaje. ¿Como pasamos de elementos 2x2 a elementos 4x4 interpolados?

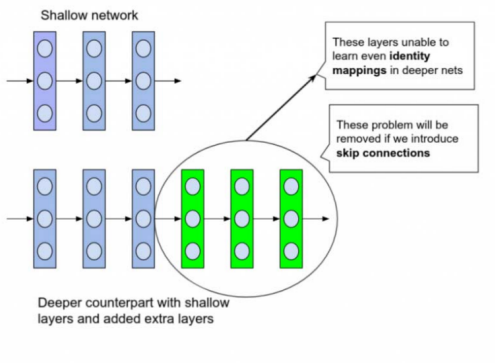
Se agrega un margen que no se utiliza pero facilita la implementación (la implementación se puede modificar). El filtro optimizado se ubica en la posición que corresponde a cada valor del input y se multiplica por ese valor. Esto se hace para las cuatro posiciones y los resultados que se sobreponen se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



Dependiendo del tipo de Stride que se aplique y el tmaño 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.

1.7 Skip connections

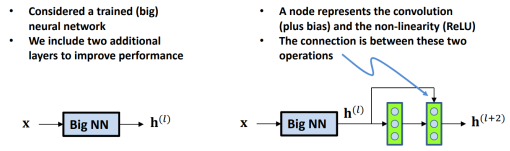
Si incrementamos el tamaño de la red (número de capas $\longrightarrow$ parámetros) debería dar mejores resultados aunque se requiera más datos de entrenamiento y más tiempo por época. Pero **no**. Hay casos donde incrementamos el número de capas pero obtenemos un error mayor, se conoce como **degradation 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).



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

Si lo pesos y los sesgos se ponen a 0 (gradiente colapsado). Entonces los **residual Block** pueden aprender la funcion 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 matrizes coincidan, para ello se realizan convoluiones 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).

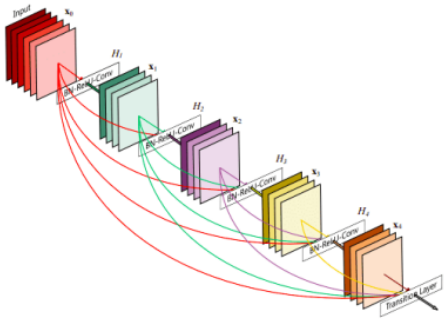
$$\mathbf{h}^{(l+2)} = g\big(\mathbf{a}^{(l+2)} + \mathbf{W}_s\mathbf{h}^{(l)}\big) = g\big(\mathbf{W}^{(l+2)}\mathbf{h}^{(l+1)} + \mathbf{b}^{(l+2)} + \mathbf{W}_s\mathbf{h}^{(l)}\big)$$

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

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



2 Practical Aspects in Neuronal Networks

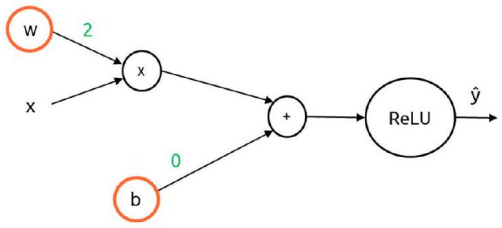
3 Architectures

Exercises: Topic 1 - Basic Elements in Neural Networks

Basic Exercises

1.1. Consider the system described in the figure, governed by parameters w and b, already initialized. The system must be trained, using an L1 loss function, to learn the identity function $f(x) = x \quad \forall x > 0$.

$$(L_1)_1 = |y_i - f(x_i)|$$



a) Complete the computational graph of the system during training, and compute the forward & backward pass for a sample $(x, y) = (1, 1)$

Foward step:

$$w \times 1 = 2 \longrightarrow 2 + b = 2 \longrightarrow ReLu(2) = 2 \longrightarrow \hat{y} = 2$$
$$\mathcal{L}(x, y) = |2 - 1| = 1$$

Backward step: $x_3 = ReLu(x_2),\ x_2 = x_1 + b,\ x_1 = wx$

$$\frac{\partial \mathcal{L}}{\partial x_3} = 1$$
$$\frac{\partial \mathcal{L}}{\partial x_2} = \frac{\partial x_3}{\partial x_2}(2)\frac{\partial \mathcal{L}}{\partial x_3} = 1$$
$$\frac{\partial \mathcal{L}}{\partial x_1} = \frac{\partial x_2}{\partial x_1}(2)\frac{\partial \mathcal{L}}{\partial x_2} = 1$$
$$\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial x_2}{\partial x_b}(0)\frac{\partial \mathcal{L}}{\partial x_2} = 1$$
$$\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial x_1}{\partial x_1}(2)\frac{\partial \mathcal{L}}{\partial x_2} = 1$$

b) Update the parameters with a single step of SGD with a learning rate of $\alpha = 0.1$

$$\theta^{k+1} = \theta^k - \alpha \nabla \mathcal{L}$$
$$w = w - 0.1\nabla_w \mathcal{L} = 2 - 0.1 = 1.9$$
$$b = b - 0.1\nabla_b \mathcal{L} = -0.1$$

c) Compute another forward & backward pass. Compare the obtained error with that of the first pass

Foward step:

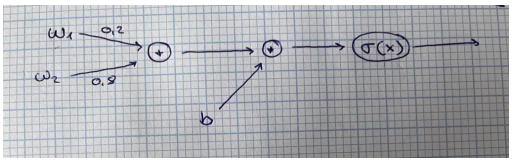
$$w \times 1 = 1.9 \longrightarrow 1.9 + b = 1.8 \longrightarrow ReLu(1.8) = 1.8 \longrightarrow \hat{y} = 1.8$$
$$\mathcal{L}(x, y) = |1.8 - 1| = 0.8$$

Backward step: $x_3 = ReLu(x_2),\ x_2 = x_1 + b,\ x_1 = wx$

$$\frac{\partial \mathcal{L}}{\partial x_3} = 1$$
$$\frac{\partial \mathcal{L}}{\partial x_2} = \frac{\partial x_3}{\partial x_2}(1.8)\frac{\partial \mathcal{L}}{\partial x_3} = 1$$
$$\frac{\partial \mathcal{L}}{\partial x_1} = \frac{\partial x_2}{\partial x_1}(1.9)\frac{\partial \mathcal{L}}{\partial x_2} = 1$$
$$\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial x_2}{\partial x_b}(0)\frac{\partial \mathcal{L}}{\partial x_2} = 1$$
$$\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial x_1}{\partial x_1}(1.9)\frac{\partial \mathcal{L}}{\partial x_2} = 1$$

1.2. Consider a binary classifier implemented with a single neuron modelled by two weights $w_1 = 0.2\mathrm{w}_1 = 0.2$ and $w_2 = 0.8$ and a bias $b = -1$. Consider the activation function to be a sigmoid $f(x) = 1/\left(1 + e^{-x}\right)$.

a) Draw a scheme of the model.



b) Compute the output of the logistic regressor for a given input $\mathbf{x}^T = \begin{bmatrix} 1 & 1 \end{bmatrix}$.

$$w_1 + w_2 = 1 \longrightarrow 1 + b = 0 \longrightarrow \sigma(0) = \frac{1}{2}$$

c) Considering a classification threshold of $\gamma = 0.9 (\gamma > 0.9$ for class A, and $\gamma < 0.9$ for class B), which class would be predicted for the considered input $\mathbf{x}^T = \begin{bmatrix} 1 & 1 \end{bmatrix}$?

$0.5 < \gamma$ por lo que x se considera de la clase B.

1.3. We want to train a neural network capable of distinguishing between images of cats (or dogs), in a multi-class classification setting. Below are the network predictions for a batch of 5 training samples, as well as their annotated labels.

a) Calculate the accuracy of this batch.

b) Calculate the precision of the "Dog" class for a decision threshold $\gamma = 0.5$

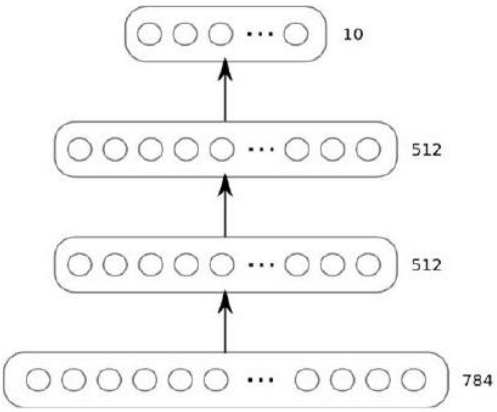
c) Calculate the recall of the "Dog" class for a decision threshold $\gamma = 0.5$

Cat	0.8	0.3	0.4	0.2	0.7	1	0	1	0	1	
Dog	0.2	0.7	0.6	0.8	0.3	0	1	0	1	0	

1.4. How many parameters does the following fully connected MLP contain? The network is built as:

- 3-layer neural network (2 hidden layers)
- 784 inputs

- 512-512-10
- tanh units (activation function)
- Softmax on top layer



1.5. Consider this neural network, given the parameters of its layers:

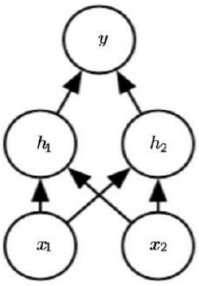
***Note**: The notation and definitions in this exercise are not the same as in the course notes but you have to get used to interpret such variations.*

- Parameters of the first layer:

$$\mathbf{W}^{(1)} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \text{ and } \mathbf{c}^{(1)} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

- Parameters of the second layer:

$$\mathbf{w}^{(2)} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}$$



a) How many hidden layers are there?

Only one hidden layer, whith h_1 and h_2

b) Compute h_1 and h_2 given as inputs $x_1 = 0$ and $x_2 = 0$.

$$\mathbf{h} = \begin{bmatrix} h_1 \\ h_2 \end{bmatrix} \mathbf{W}\mathbf{x} + \mathbf{c} = \mathbf{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

c) Consider an activation ReLU, $g(h_i) = \max(0, h_i)$, between the first and second layer. Compute $g(h_1)$ and $g(h_2)$.

$$g(h_1) = ReLu(0) = 0$$
$$g(h_2) = ReLu(-1) = 0$$

d) Compute the output value y.

$y = g(\mathbf{h})\mathbf{w} = 0$

e) Compute the output of all possible combinations of the binary inputs (truth table). What is the implemented logic gate?

The XOR-logit gate.

$(0,0)$	0
$(1,0)$	1
$(0,1)$	1
$(1,1)$	0

1.6. If we have three classes on our dataset where the first class corresponds to 80% of the total dataset, and the second and third classes to the 10% respectively, propose the factor values that could be used to solve the problem.

Como las clases no tienen proporciones similares dentro de las observaciones del dataset, no se puede asumir que la función de perdida deba ser ponderada por el mismo factor $\frac{1}{N}$. Sino que se debe adecuar en función de los porcentajes de cada clase, de modo que los factores serían los siguientes:

- Para la primera clase: $a_1 = \frac{1}{0{,}8}$
- Para la segunda clase: $a_2 = \frac{1}{0.1}$
- Para la tercera clase: $a_3 = \frac{1}{0.1}$

1.7. We are using a network to generate encodings (embeddings) of images to recognize faces in a metric learning framework. Suppose that we are using 1-dimensional embeddings and that the embedding value for the anchor image is $f_{\boldsymbol{\theta}}(\mathbf{x}_A) = 1$, for a positive sample is $f_{\boldsymbol{\theta}}(\mathbf{x}_P) = 2$ and for a negative sample $f_{\boldsymbol{\theta}}(\mathbf{x}_N) = 3$.

a) Compute the contrastive loss between the pairs $(\mathbf{x}_A, \mathbf{x}_P)$ and $(\mathbf{x}_A, \mathbf{x}_N)$ using as distance function $L1$ and a margin $m = 4$. Use the following expression for the contrastive loss:

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

b) In the previous scenario, compute the triplet loss of the triplet $(\mathbf{x}_A, \mathbf{x}_P, \mathbf{x}_N)$.

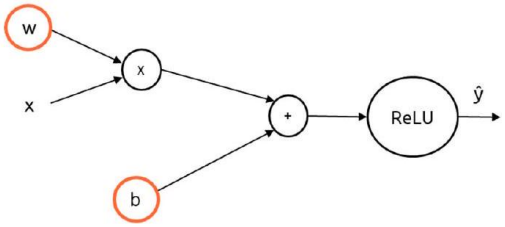
Exam Exercises

Mid Term Exam Fall 2022. Exercise 1:

We would like to train the following system to learn an unknown function for which we have some pairs of training data.

In order to analyse a simple problem, we optimize the system using a single training pair $(x, y) = (2, 2)$ and a learning rate $\alpha = 0.3$.

The last iterations of the optimization process result in the following values:



1.

Iteration	**n**	**n + 1**	**n + 2**	**n + 3**	**n + 4**	**n + 5**
w	2.7	2.1	1.5	0.9	1.5	0.9
b	1	0.7	0.4	0.1	0.4	0.1
$\mathcal{L}$	4.4	2.9	1.4	0.1	1.4	0.1
$\nabla_w \mathcal{L}\|_{w^n}$	2	2	2	-2	2	-2
$\nabla_b \mathcal{L}\|_{b^n}$	1	1	1	-1	1	-1

Note: $\frac{\partial |y_i - f(x_i)|}{\partial x_i} = \begin{cases} -f'(x_i) \text{ if } y_i - f(x_i) > 0 \\ f'(x_i) \text{ if } y_i - f(x_i) < 0 \end{cases}$

2. Complete the computational graph using the loss function L1: $\mathcal{L} = |y_i - f_{\boldsymbol{\theta}}(\mathbf{x}_i)|$
   Se debe agregar un nodo mas que multiplique por -1, otro que sume el valor real y_i, otro que haga el valor absoluto $|.|$ y al final la salida es la función de perdida $\mathcal{L}$
3. Using the gradient descent algorithm and the L1 loss function, find the missing values on the table above for iterations $k = n + 3, k = n + 4$ i $k = n + 5$. Note: Use a different figure for the computational graphs for each forward / backward propagation.

$$\text{Gradient descent algorithm:} \quad \boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k - \alpha \boldsymbol{\nabla}_{\boldsymbol{\theta}} \mathcal{L}|_{\boldsymbol{\theta}^k}$$

4. Draw the sequence of solutions of the previous optimization on the plane (w, b), discuss its behavior in all iteration steps and how it will evolve in the subsequent iterations $(k > n + 5)$

A partir de la iteración $n+4$ se observa que el valor de la función de perdida vuelve a ser la misma que en la iteración $n+2$ y en la iteración $n+5$ vuelve repetirse la iteración $n+3$. Con lo que se puede deducir que el learning rate es muy grande y las iteraciones se quedan saltando de un punto a otro alrededor del valor óptimo para las próximas iteraciones.

To improve the convergence of the algorithm, two solutions are proposed that are studied starting at iteration $\boldsymbol{n} + 3$:

The first option is to use, instead of the L1 loss function, the Huber loss function (Huber loss or delta loss) which is given by the following expression:

$$\mathcal{L}_\delta = \begin{cases} \frac{1}{2}\left(y_i - f_{\boldsymbol{\theta}}(\mathbf{x}_i)\right)^2 & \text{for } |y_i - f_{\boldsymbol{\theta}}(\mathbf{x}_i)| < \delta \\ \delta\,|y_i - f_{\boldsymbol{\theta}}(\mathbf{x}_i)| - \frac{1}{2}\delta^2 & \text{otherwise} \end{cases}$$

5. Explain the rational of this proposal. What do you expect to achieve? What delta value do you suggest? At what iteration would this delta value cause the Huber function to change its behavior? Calculate an iteration of the optimization algorithm with the Huber loss function from the chosen iteration and comment on the result.

La idea de la función de perdida Huber es conectar las dos variantes, L1 y L2 para aprovechar las ventajas que ambas proporcionan. Pues cerca del 0 es mejor que la función de perdida se comporte como una función cuadrada y lejos del intervalo $\{-1, 1\}$ queremos aprovechar las ventajas de la función de perdida lineal.

El valor de δ ha de contener al valor 0.1 para ajustar la función de perdida a una función cuadrática y no quedar atrapados alrededor del óptimo. Podemos adoptar un valor ligeramente superior a 0.1, con lo que este valor de δ haría cambiar el comportamiento de la función de Huber en la iteración $n + 3$.

Con esto el resultado de la función de perdida sería:

$$L_\delta = \frac{1}{2}(2-1.9)^2 = 0.005, \quad \frac{\partial L_\delta}{\partial \hat{y}} = (2-\hat{y})(-1) = -0.1$$

Y por lo tanto, tras hacer el backpropagation, la actualización de los parámetros se notaría en un ligero aumento, pues la función cuadrática permite este descenso más controlado hacia el óptimo que no la función lineal:

$$w = 0.9 - 0.3(-0.2) = 0.96, \quad b = 0.1 - 0.3(-0.1) = 0.13,$$
$$\hat{y} = 2(w) + b = 2.05, \quad L_\delta = \frac{1}{2}(2-2.05)^2 = 0.00125$$

Another option to improve convergence is to use an adaptive learning step, using the AdaGrad (Adaptive Gradient Algorithm) algorithm:

$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k - \alpha \left(\varepsilon \mathbf{I} + \operatorname{diag}\left(\mathbf{G}^k\right)\right)^{-1/2} \boldsymbol{\nabla}_{\boldsymbol{\theta}} \mathcal{L}\Big|_{\boldsymbol{\theta}^k}$$

$$\text{where} \quad \mathbf{G}^{\mathrm{k}} = \sum_{\boldsymbol{\tau}=1}^{\mathrm{k}} \mathbf{g}^{\boldsymbol{\tau}} \left(\mathbf{g}^{\boldsymbol{\tau}}\right)^{\mathrm{T}} \quad \text{and, in this case} \quad \mathbf{g}^{\mathrm{k}} = \boldsymbol{\nabla}_{\boldsymbol{\theta}} \mathcal{L}|_{\boldsymbol{\theta}^{\mathrm{k}}}$$

6. Explain the basis of this proposal. What do you expect to achieve? How does the gradient behave in the case of the previous table $\left(\mathbf{g}^{\mathrm{k}} = \boldsymbol{\nabla}_{\boldsymbol{\theta}} \mathcal{L}|_{\boldsymbol{\theta}^{\mathrm{k}}}\right)$? Compute the matrix $\mathbf{G}^{\mathrm{n+3}}$. How will the normalization matrix affect the process $\left(\varepsilon \mathbf{I} + \operatorname{diag}\left(\mathbf{G}^{n+3}\right)\right)$

Mid Term Exam Fall 2022. Exercise 2: In 1998, the research group led by Yann LeCun presented the network called LeNet which, in summary, had the following characteristics:

The network architecture was [CONV-POOL-CONV-POOL-FC-FC]:

- INPUT: $[32 \times 32 \times 1]$
- CONV: $5x5$ convolutional filters applied at stride 1 $[28 \times 28 \times 6]$
  - tanh non-linearity after the convolution layer
- POOL: Average 2×2 pooling at stride 2 $[14 \times 14 \times 6]$
  - tanh non-linearity after the pooling layer
- CONV: $5x5$ convolutional filters applied at stride 1 $[10 \times 10 \times 16]$
  - tanh non-linearity after the convolution layer
- POOL: Average $2x2$ pooling at stride 2 $[5 \times 5 \times 16]$
  - tanh non-linearity after the pooling layer
- FC: Fully connected layer with 120 neurons (including a non-linearity) [120]
- FC: Fully connected layer with 84 neurons (including a non-linearity) [84]
- FC: Radial Basis Function unit to obtain the final probabilities [10]

1. Justify the padding that has been used on each layer that needs it in order to obtain the dimensions of the volumes specified in the characteristics above.

No se ha utilizado padding en ninguna de las capas de filtrado. Esta afirmación se respalda a base de comprobar las dimensiones de las salidas después de cada capa.

En la primera convolución se realiza un filtrado 5×5 con lo que en los bordes de la imagen, el filtro sobresale siempre por dos píxeles de longitud. Contando para los cuatro bordes esto significa un total de 4 columnas y 4 filas menos, con lo que la dimensión de salida $28 \times 28 \times 6$ corresponde a no utilizar padding.

En la segunda capa de convolución, ocurre la misma reducción que en la primera capa comentada, por lo que en esta capa tampoco se ha utilizado padding.

2. Compute the number of parameters for each layer and the total for the entire network.

Asumimos de la pregunta que se refieren a parámetros optimizables por la red. No se incluyen los hyperparámetros.

Para cada capa de convolución l sabemos que tiene un total de $(F_l \cdot F_l \cdot D_l + 1) \cdot K_l$. Para las dos capas de filtrado tenemos:

- $F_1 = 5,\ D_1 = 1,\ K_1 = 6$
- $F_2 = 5,\ D_2 = 6,\ K_2 = 16$

En cuanto a la red FCNN, la entrada consiste en los $5 \times 5 \times 16$ píxeles que conectan con cada una de las 120 neuronas, más sus 120 sesgos (asumiendo sesgos no nulos) tenemos un total de $400 * 120 + 120$. En la capa oculta se tienen que optimizar 120 pesos para cada una de las 84 neuronas más sus 84 sesgos, con lo que suman $400 * 120 + 120 + 120 * 84 + 84$. Para finalizar con 84 pesos para cada una de las 10 neuronas de salida, cada una con sus sesgo correspondiente. Con lo que el total queda:

$$(5 \cdot 5 \cdot 1 + 1) \cdot 6 + (5 \cdot 5 \cdot 6 + 1) \cdot 16 + 400 * 120 + 120 + 120 * 84 + 84 + 84 * 10 + 10 = 61706$$

3. With the knowledge you currently have about the elements that bring better performance to a network, comment on what changes you would make to the architecture of the LeNet network, justifying each proposal. Changes to the elements that make up the LeNet network are requested, not global architectural changes.

   Las neuronas ReLU están activas (emiten un valor distinto de cero) o inactivas (emiten 0) para una entrada dada. Esta sparseness puede ayudar con la eficiencia y la regularización del modelo, ya que fomenta que la red se enfoque en un conjunto más pequeño de características, reduciendo el sobreajuste. Además que ReLU es computacionalmente más simple que tanh.

   La agrupación por máximo (max pooling) retiene las características más importantes dentro de una región de la entrada. Al tomar el valor máximo en cada grupo, preserva las características dominantes y descarta información menos relevante. Además, la agrupación por máximo (max pooling) ayuda a capturar bordes y límites dentro de los datos. Los bordes suelen ser características destacadas en las imágenes, y la agrupación por máximo ayuda a resaltar estos fuertes gradientes al tomar el valor máximo en el campo receptivo.

   El relleno (padding) ayuda a mantener las dimensiones espaciales de los mapas de características, lo cual puede ser crucial para preservar la estructura e información posicional en los datos. Sin relleno (padding), los píxeles en los bordes de la entrada pueden estar subrepresentados en los mapas de características de salida. Esto puede llevar a una reducción de la información en los bordes de la imagen.