4. Redes Neuronales Artificiales del tipo feed-forward [*]

Facundo Bromberg Ph.D.
Laboratorio DHARMa Dept. de Sistemas de Información
U. Tecnológica Nacional - Facultad Regional Mendoza - Argentina
fbromberg@frm.utn.edu.ar
http://dharma.frm.utn.edu.ar

Agosto 2012

Índice

[*]Los contenidos del presente capítulo han sido desarrollados tomando como referencia principal el capítulo 6 de *Pattern Recognition and Machine Learning* de Christopher M. Bishop [1].

Las redes neuronales feed-forward generalizan al perceptrón y los discriminadores lineales resolviendo muchas de sus limitaciones. Por un lado, al igual que el perceptrón, generalizan a los discriminadores lineales permitiendo discriminar datasets que no son linealmente separables. Por otro, generalizan al perceptrón a un perceptrón multi-capa, garantizando la separabilidad de cualquier función binaria, y permitiendo la clasificación multi-clase. Por último, proveen un mecanismo automático para el mapeo al espacio de features ϕ, mejorando drasticamente las chances de encontrar un mapeo que separe el dataset inicial. A pesar de estas ventajas, las redes feed-forward presentan aún una importante limitación y es que la función de error no solo no es analíticamente minimizable, sino que no es convexa, resultando en espacio de búsqueda con mínimos locales. Esto resulta no solo en importantes tiempos de entrenamiento, sino que no garantiza encontrar la superficie de separación. Mas aún, al igual que discriminante lineales y perceptrón, el algoritmo de entrenamiento de las redes feed-forward no provee garantias sobre el error de generalización, dando lugar a un posible sobre ajuste. Existen sin embargo técnicas ad-hoc para evitar este problema como ser regularización o la terminación temprana por minimización del error de validación. En contraste, como veremos en el próximo capítulo, existe el formalismo de las máquinas de vectores soporte (SVM por sus siglas en inglés) que no solo garantiza una minimización del error de generalización, sino que su entrenamiento consiste en una minimización numérica sobre un espacio convexo, i.e., un solo mínimo: el mínimo global.

A pesar de las ventajas de las SVMs, las redes neuronales gozan de una importantisima popularidad. Esto es resultado de la simpleza de su formalismo, su importante presencia histórica, y por la simpleza para implementarlas tanto en programas de software como en sistemas electrónicos.

1. Preliminares

Al igual que en el caso de discriminadores lineales y Perceptrón, las redes feed-forward resuelven el problema de aprendizaje inductivo, tanto *clasificación binaria*, *clasificación multi-clase*, como así también *regresión multivariada*. En su formulación mas general, el objetivo del aprendizaje inductivo es estimar una función subyacente desconocida $\widehat{\mathbf{y}}^\star(\mathbf{x})$, potencialmente multivariada, a partir de un conjunto de entrenamiento $\mathcal{T} = \{(\mathbf{x}_n, \mathbf{t}_n)\}_{n=1}^N$ de N ejemplos de comportamiento de $\widehat{\mathbf{y}}^\star$ dados por un conjunto de entradas $\mathbf{x}_n$ y sus correspondientes salidas $\mathbf{t}_n = \widehat{\mathbf{y}}^\star(\mathbf{x}_n)$. El aprendizaje consiste en determinar una función *hipótesis* $\mathbf{y}$ que aproxime correctamente $\widehat{\mathbf{y}}^\star$. Para que el

aprendizaje sea efectivo, la hipótesis **y** debe ser correcta (i.e., coincidir con $\widehat{\mathbf{y}}^{\star}$) en todo el dominio de valores de $\mathbf{x}$. Al error cometido por $\mathbf{y}$ en entradas con salida desconocida, i.e., $\mathbf{x} \notin \mathcal{T}$, s e le denomina *error de generalización*. A primera vista parecería que para lograr una buena estimación, lo ideal es encontrar una hipótesis *consistente*, i.e., una hipótesis $\mathbf{y}$ que coincida con $\widehat{\mathbf{y}}^{\star}$ en todos los ejemplos de entrenamiento. Esto sin embargo funciona correctamente (minimiza el error de generalización) solamente en casos donde las entradas no son ruidosas, es decir, cuando $\mathbf{t}_n = \widehat{\mathbf{y}}^{\star}(\mathbf{x}_n)$ para todo n. Cuando los datos son ruidosos, i.e., cuando $t_n = \mathbf{y}(\mathbf{x}_n) + ruido$, se corre el peligro de *sobre ajustar* (*overfit*) la hipótesis a la ideosincracia del ruido en los datos de entrenamiento, resultando en malas estimaciones para nuevos inputs. Una hipótesis consistente sería entonces un caso extremo de sobre ajuste.

Ya hemos estudiado un caso concreto de aprendizaje inductivo de funciones binarias para un espacio de hipótesis compuesto por separadores lineales y no lineales (perceptrón) de la forma:

$$y(\mathbf{x}) = f(\mathbf{w}^T \phi(\mathbf{x}) + b)$$

que divide el espacio en dos regiones correspondientes cada una a una clase que satisfacen $y(\mathbf{x}) \geq 0$ y $y(\mathbf{x}) < 0$ respectivamente, y por lo tanto estan separadas por el hiper-plano $\mathbf{w}^T \phi(\mathbf{x}) + b = 0$. También, hemos generalizado esto al caso multi-clase, considerando los casos OvO (one-vs-one), y OvA (one-vs-all).

Para el caso especial del perceptrón (función de activación no lineal y mapeo al espacio de features), se propuso una función de error novedosa (*perceptron criteria*), resultando en un algoritmo de aprendizaje de los pesos que minimizan este error:

$$\mathbf{w}^{t+1} \longleftarrow \mathbf{w}^t + \eta \phi_n(\mathbf{x}_n) t_n$$

donde η es la *taza de aprendizaje*, un parámetro libre. Se demostró también que la aplicación secuencial de esta regla de aprendizaje resulta en una convergencia de los pesos a una hipótesis consistente.

2. Representación de redes feed-forward

Las redes feed-forward son una especialización muy sencilla del perceptrón que propone parametrizar el mapeo al espacio de features y modificar la función de activación a una función continua, permitiendoles generalizar

al perceptrón al caso multi-clase (lo que no es posible con el Perceptrón de Rosenblat).

Comencemos re-expresando la función de separación del perceptrón para el caso de K salidas de la siguiente manera

$$\begin{aligned} y_k(\mathbf{x}) &= f(\mathbf{w}_k^T \phi(\mathbf{x}) + b_k) \\ &= f\left(\sum_{j=1}^{M} w_{kj}\phi_j(\mathbf{x}) + w_{j0}\right) \end{aligned} \tag{1}$$

donde M es la dimensión del espacio de features ϕ, w_{j0} es el k-esimo bias b_k renombrado por una conveniencia notacional mas adelante, y donde f es usualmente la *identidad* para el caso de regresión, la función *logística* para clasificación binaria, o la función *softmax* para clasificación multi-clase (definida formalmente mas abajo). Interesantemente, la parametrización propuesta para el mapeo $\mathbf{x} \longrightarrow \phi$ consiste también en una función lineal de la misma forma funcional que el perceptrón:

Viene de los perceptrones →

$$\phi_j(\mathbf{x}) = h\left(\sum_{i=1}^{D} w_{ji}x_i + w_{j0}\right) \tag{2}$$

donde h es una *función de activación* no-lineal, e.g., *logística* o *tanh*, y D es la dimensión del espacio de entrada $\mathbf{x}$.

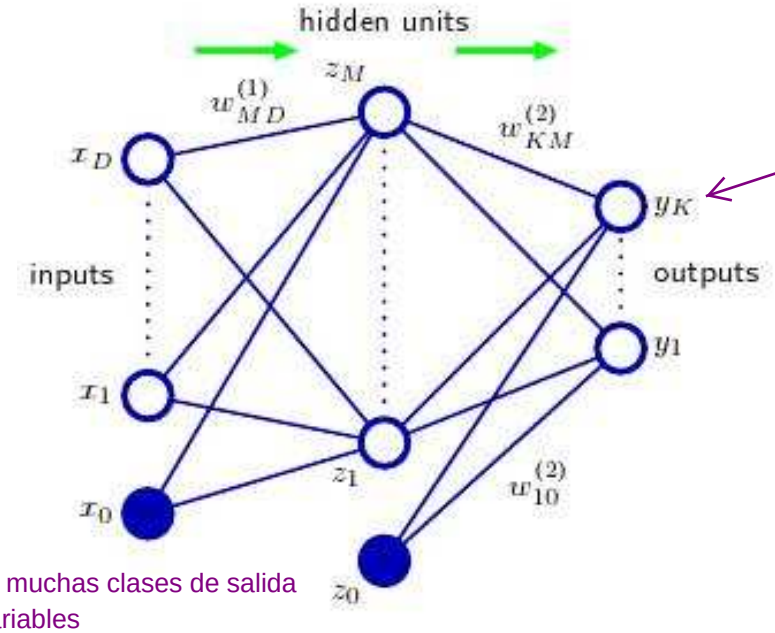
Combinando las Eqs. (1) y (2) obtenemos

Para problemas binarios o de regresión →

$$y_k(\mathbf{x}) = f\left[\sum_{j=1}^{M} w_{kj}^{(2)} h\left(\sum_{i=1}^{D} w_{ji}^{(1)}x_i + w_{j0}^{(1)}\right) + w_{k0}^{(2)}\right]. \tag{3}$$

Al igual que el perceptrón, el computo de esta fórmula puede representarse graficamente en forma de una red de unidades neuronales del tipo feed-forward, tal como se muestra en la Fig. 1. Como vemos, la red presenta tres capas con la información fluyendo desde la capa de la izquierda también llamada *capa de entrada*, pasando por la *capa intermedia* o *oculta*, hasta la capa de la derecha llamada *capa de salida*. Debido a esta capa intermedia a estas redes se las suele llamar *perceptrón multi-capa*.

La capa de entrada recibe los inputs $\mathbf{x} = \{x_1, \ldots, x_i, \ldots, x_D\}$ mas el input 1 correspondiente al bias w_{j0} (marcado con un circulo relleno azul). La información de cada uno de los inputs x_i se propaga a cada una de las neuronas z_j de la capa intermedia, la cual representa a ϕ_j, pesada por el



Tantas yk de salida como clases de clasificación posibles

Multiclase -> una varible y muchas clases de salida
Multivariado -> muchas variables

Figura 1: Representación gráfica de una red neuronal de tres capas: la capa de entrada (izquierda), la capa intermedia ú oculta, y la capa de salida (derecha).

peso w_{ji}. Cada neurona z_j recibe información de cada uno de los inputs x_i, los cuales suma en la activación

Activación:
Cuánta información
llega a dicha neurona

$$a_j = \sum_{i=1}^{D} w_{ji} x_i + w_{j0},$$

y produce como salida el resultado de aplicarle a a_j la función de activación h, i.e.,

$$z_j = h(a_j). \tag{4}$$

El flujo de información desde la capa oculta a la capa de salida es similar. Como puede verse en Eq. (3), a_j es la cantidad en parentesis, y z_j, la salida de la capa oculta, hace las veces de entrada a la primer sumatoria sobre j de la misma manera que los x_i hacian de entrada a la segunda sumatoria. Así, la información de cada una de las M neuronas ocultas z_j se propaga hacia cada una de las neuronas de salida y_k, pesadas por los pesos w_{kj}. Los

M inputs a las neuronas de salida se suman en la activación a_k, i.e.,

$$a_k = \sum_{j=1}^{M} w_{kj} z_j + w_{k0},$$

y se produce la salida

$$y_k = f(a_k)$$

luego de aplicarle la función de activación f. Es sencillo ver que reemplazando z_j de la Eq. (4) en estas dos últimas ecuaciones recuperamos la expresión completa del computo de y_k de la Eq.(3).

Para simplificar la notación hemos obviado los superindices (1) y (2) que distinguían los pesos de la primera y segunda capa. Esto no es un problema, sin embargo, ya que el nombre de los subindices será suficiente para distinguir los pesos de la primer capa (w_{ji}) con los de la segunda capa (w_{kj}).

Una de las ventajas que provee la esquematización gráfica de estas redes, es que permite entenderlas como una red de unidades de computo que toman como entrada una activación consistente en la suma pesada de las neuronas de entrada, y produce una salida que es alguna función no-lineal de la activación. De esta manera, es sencillo generalizar estas redes a estructuras mas complejas, siempre y cuando la información fluya en un solo sentido, i.e., que la red sea *feed-forward*. Es es para asegurarse que la salida de la unidad es una función determinista de sus entradas. Por ejemplo, puede considerarse conexiones directas entre la capa de entrada y la de salida. También, es posible agregar mas capas intermedias. Por último, puede omitirse algunas de las conexiones.

2.1. Redes feed-forward son proximadores universales

El poder de aproximación de estas redes ha sido ampliamente estudiado y se ha encontrado que es muy general. Es por ello que se dice que las redes neuronales son *aproximadores universales*. Por ejemplo, una red con una capa oculta con salidas lineales puede aproximar con cualquier grado de aproximación cualquier función continua sobre un dominio de entrada compacto, siempre que tenga un número suficiente de neuronas ocultas. Este resultado funciona para un número amplio de funciones de activación de la capa oculta (h), excepto polinomios.

Sin duda esta información es muy tranquilizadora ya que nos asegura que cualquier función subyacente puede ser aproximada, es decir, existen pesos que la aproximan con precisión arbitraria. Sin embargo, queda aún la

importante tarea de asegurarse que el algoritmo de entrenamiento sea capaz de encontrar estos pesos.

Las capacidad para aproximar diferentes funciones se ejemplifica en la Fig.2. La figura también muestra como las neuronas ocultas trabajan colaborativamente para aproximar la función final.

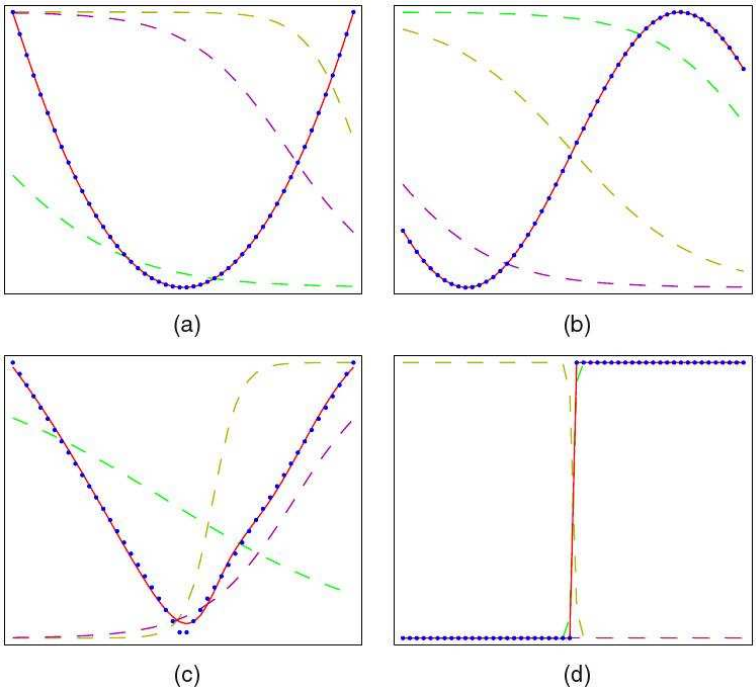


Figura 2: Ilustración de la capacidad del perceptrón multi-capa para aproximar cuatro diferentes funciones: (a) $f(x) = x^2$, (b) $f(x) = sin(x)$, (c) $f(x) = |x|$, y (d) $f(x) = H(x)$, donde $H(x)$ es la función escalón. En cada caso, $N = 50$ datapoints, mostrados en azul, fueron muestreado uniformemente en x sobre el intervalo $(-1, 1)$ y su etiqueta de entrenamiento t_n computada evaluando la $f(x)$ correspondiente. Las funciones aprendidas por un algoritmo de entrenamiento sobre una red con 3 neuronas ocultas del tipo *tanh* se muestran en rojo, y la salida de cada una de las neuronas ocultas es muestran en lineas puntueadas.

3. Aprendizaje: error back-propagation

Al igual que en el caso del Perceptrón y discriminantes lineales, propondremos una función error $E(\mathbf{w})$ para las distintas hipotesis, y buscaremos aquella hipotesis $\mathbf{w}$ que la minimiza. Esta minimización no podrá resolverse analíticamente tal como fue posible para discriminantes lineales, con lo que usaremos *gradient-descent* al igual que con el Perceptrón, resultando en:

$$\mathbf{w}^{\tau+1} = \mathbf{w}^{\tau} - \eta \nabla E(\mathbf{w}^{\tau})$$

donde η es la *taza de aprendizaje*. A esta caso se le llama *batch learning* o *aprendizaje por lotes* ya que el cálculo del gradiente requiere procesar todo el dataset en cada paso τ. Existe otro caso llamado *online learning* que en la práctica ha demostrado escapar mas facilmente de los mínimos locales de la función error. En esta modalidad, los pesos se actualizan luego de la observación de un solo data point. Aún no conocemos la función error, y esta dependerá de si el problema de aprendizaje es de *regresión*, *clasificación* binaria o *clasificación* multi-clase, pero como veremos mas adelante (siguiente sección), todas las funciones error que propondremos pueden descomponerse sobre cada *data-point*:

$$E(\mathbf{w}) = \sum_{n=1}^{N} E_n(\mathbf{w})$$

y, por ser el gradiente lineal, podemos actualizar los pesos individualmente para cada data-point $\mathbf{x}_n$:

$$\mathbf{w}^{\tau+1} = \mathbf{w}^{tau} - \eta \nabla E_n(\mathbf{w}^{\tau}). \tag{5}$$

Existen dos modalidades para el aprendizaje online, una donde los datapoints se visitan en el orden en que aparecen en el dataset, y otros donde se visitan aleatoriamente.

Entonces, para terminar de definir el aprendizaje, debemos por un lado definir las funciones de error para los distintos casos de regresión, clasificación binaria y multiclase, y luego calcular el gradiente de este error. Veremos esto en las siguientes dos secciones, respectivamente.

3.1. Funciones de error

Consideraremos tres casos: *regresión*, *clasificación binaria*, y *clasificación multiclase*.

Para el caso de regresión se contempla una función *identidad* para la función de activación de salida f, y una función error del tipo error cuadrático medio

$$E(\mathbf{w}) = \sum_{n=1}^{N} E_n(\mathbf{w}),$$

donde $E_n(\mathbf{w})$ es el error cuadrático medio del n-esimo data-point $\mathbf{x}_n$ cometido por la red con pesos $\mathbf{w}$:

Mínimos Cuadrados

$$E_n(\mathbf{w}) = \frac{1}{2} \sum_{k=1}^{K} \left[y_k(\mathbf{x}_n, \mathbf{w}) - t_{nk} \right]^2 .$$

Es facil ver que este error se minimiza cuando la hipótesis $\mathbf{y}(\mathbf{x})$ es consistente, es decir, cuando $y_k(\mathbf{x}_n) = t_n k$ para todo n.

Para el caso de clasificación binaria en clases $\mathcal{C}_1$ y $\mathcal{C}_2$, consideramos etiquetas binarias $t \in 0, 1$, codificadas con $t = 1$ para la clase $\mathcal{C}_1$ y $t = 0$ para $\mathcal{C}_2$ (a diferencia de la codificación $+/-1$ del Perceptrón). La única variable de salida y, a diferencia de la función escalón considerada en el Perceptrón, tiene una función de activación sigmoidea

$$y = \sigma(a) = \frac{1}{1 + exp(-a)},$$

de tal manera que $0 \leq y(\mathbf{x}) \leq 1$, y por lo tanto decide $\mathcal{C}_1$ siempre que $y(\mathbf{x}) \geq 1/2$, y $\mathcal{C}_2$ siempre que $y(\mathbf{x}) < 1/2$.

En este caso consideramos como función error la *cross-entropy* de la forma

$$E(\mathbf{w}) = -\sum_{n=1}^{N} \left[t_n \log y(\mathbf{x}_n) + (1 - t_n) \log(1 - y(\mathbf{x}_n)) \right].$$

la cual también descompone facilmente en funciones error E_n individuales para cada datapoint.

Puede verse que esta función efectivamente penaliza a los errores de clasificación. Consideremos primero el caso de $t_n = 1$ (clase $\mathcal{C}_1$). En este caso el segundo término se anula, y el primero crece cuando $y(\mathbf{x}_n)$ se acerca a su máximo que es 1, es decir, cuanto mas confianza tiene el clasificador en que la clase es $\mathcal{C}_1$, o lo que es lo mismo, cuanto mas se acerca $y(\mathbf{x}_n)$ a decidir igual que t_n. El caso de $t_n = 0$ es similar ya que en este caso se anula el primer término, y el segundo crece cuando $\mathbf{y}(\mathbf{x}_n)$ se acerca a 0. Aquí entonces, la contribución al error del n-esimo ejemplo decrece cuanto mas confianza tiene el clasificador de que la clase es $\mathcal{C}_2$, la clase de la etiqueta.

Para el caso de clasificación binaria es también posible utilizar el error cuadrático medio, pero [2] encontró que la cross-entropy, para problemas de clasificación, resulta en entrenamientos mas rapidos, y con resultados que mejoran el error de generalización.

Para concluir consideramos el caso de clasificación multi-clase. Para ello, la red debe tener K salidas y_k, $k = 1, \ldots, K$, para las que consideramos una activación de la forma *softmax*

$$y_k(\mathbf{x}) = \frac{exp(a_k}{\sum_j exp(a_j)}$$

con $0 \leq y(\mathbf{x}) \leq 1$ y $\sum\limits_k y_k = 1$. Para las etiquetas del conjunto de entrenamiento usamos la codificación 1-en-K, con lo que cada etiqueta será un vector $\mathbf{t}_n$ de K dimensiones.

La función error considerada para este caso multiclase es también la *cross-entropy*

$$E_n(\mathbf{w}) = -\sum_{k=1}^{K} t_{kn} \log(y_k(\mathbf{x}_n, \mathbf{w})).$$

la cual también descompone en errores E_n para cada ejemplo de entrenamiento. También, es posible hacer un análisis parecido al caso binario para verificar que este error efectivamente penaliza los errores de clasificación, pero será omitido por su similitud con el caso binario.

3.2. Cálculo del gradiente

El gradiente es un vector compuesto por la derivada parcial respecto a todas las variables independientes. Así, el gradiente del error $E(\mathbf{w})$ es la derivada respecto a cada uno de los pesos. Es claro que es suficiente considerar los siguientes dos casos: la derivada $\frac{\partial E_n(\mathbf{w})}{\partial w_{ji}}$ respecto de algún peso arbitrario w_{ji} de la capa oculta, y la derivada $\frac{\partial E_n(\mathbf{w})}{\partial w_{kj}}$ respecto de algún peso arbitrario w_{kj} de la capa de salida.

Comenzaremos con el cálculo de las derivadas del error respecto a los pesos de la capa de salida, y continuamos con las derivadas respecto a la capa oculta. Como veremos, resultados parciales de las derivadas de salida se utilizarán como parte del cálculo de las derivadas de la capa oculta. Como estos resultados parciales pueden interpretarse como errores de cada neurona de salida, al algoritmo resultante se le llama *error back-propagation*.

3.2.1. Gradiente para las unidades de salida

El calculo de $\frac{\partial E_n(\mathbf{w})}{\partial w_{kj}}$ procede aplicando la regla de la cadena, teniendo en cuenta que E_n depende de w_{kj} solo a través de a_k tal como lo muestra la Figura 3, vemos que

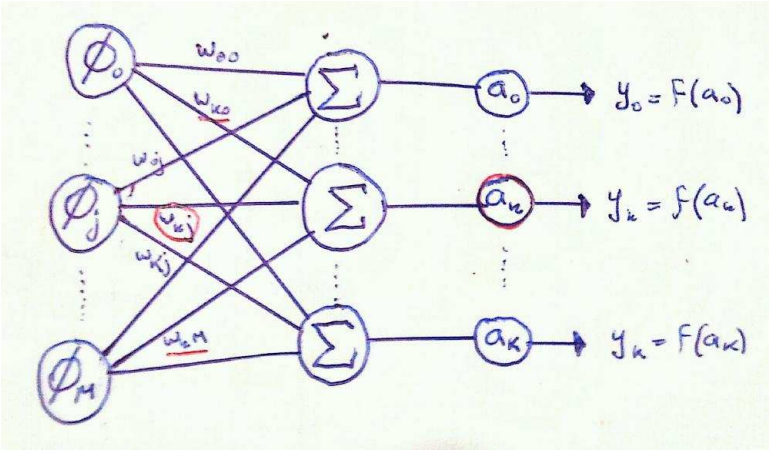


Figura 3: Detalle de la última capa de la red.

$$\frac{\partial E_n}{\partial w_k j} = \frac{\partial E_n}{\partial a_k}\frac{\partial a_k}{\partial w_k j}.$$

Dado que $a_k = \sum_{j=1}^{M} w_{kj}\phi_j$, es facil ver que

$$\frac{\partial a_k}{\partial w_{kj}} = \phi_j.$$

Además, como lo requiere demostrar el Ejercicio 2,

$$\frac{\partial E_n}{\partial a_k} = y_k - t_k.$$

Combinando ambos resultados, y denotando $\delta_k = y_k - t_k$ al error de la k-esima salida, obtenemos una expresión computable del error de la k-esima componente del gradiente de la capa de salida:

$$\frac{\partial E_n}{\partial w_{kj}} = \delta_k \phi_j.$$

3.2.2. Gradiente para las unidades ocultas

Para computar $\frac{\partial E_n(\mathbf{w})}{\partial w_{ji}}$ también comenzamos aplicando la regla de la cadena respecto a sus activaciones a_j ya que E_n depende de w_{ji} sólo a través de a_j, tal como puede confirmarse en la Figura 4:

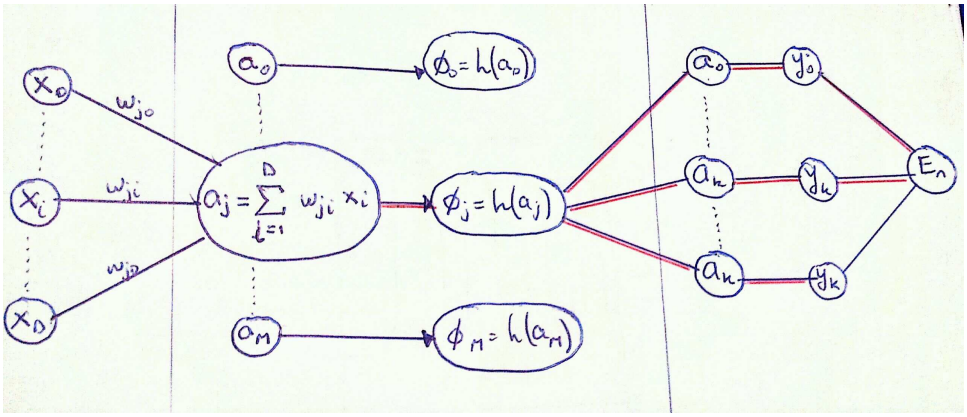


Figura 4: Detalle de dependencias entre E_n y las activaciones de las unidades ocultas a_j (en rojo).

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}}$$

donde $a_j = \sum_{i=1}^{D} w_{ji} x_i$, y por lo tanto el segundo factor $\frac{\partial a_j}{\partial w_{ji}} = x_i$.

Antes de calcular el primer factor notamos que cierta concordancia con las unidades de salida, con lo que a $\frac{\partial E_n}{\partial a_j}$ los llamamos δ_j, obteniendo

$$\frac{\partial E_n}{\partial w_{ji}} = \delta_j x_i.$$

Para calcular $\delta_j = \frac{\partial E_n}{\partial a_j}$ aplicamos también la regla de la cadena. Para ello, notamos que E_n depende de a_j a través de todas las salidas y_k (y por lo tanto a través de todas las activaciones a_k), tal como claramente lo ilustra la Figura 4 en lineas rojas. Es por ello que

$$\delta_j = \frac{\partial E_n}{\partial a_j} = \sum_k \frac{\partial E_n}{\partial a_k} \frac{\partial a_k}{\partial a_j}.$$

Como $\frac{\partial E_n}{\partial a_k} = \delta_k$, nos queda calcular $\frac{\partial a_k}{\partial a_j}$.

Dado que $a_k = \sum_{j=1}^{M} w_{kj}\phi_j = \sum_{j=1}^{M} w_{kj}h(a_j)$, tenemos que

$$\frac{\partial a_k}{\partial a_j} = w_{kj}h`(a_j)$$

Y por lo tanto, combinando estos resultados obtenemos

$$\delta_j = \frac{\partial E_n}{\partial a_j} = h`(a_j)\sum_k \delta_k w_{kj}$$

Esta última formula nos demuestra primero que debemos calcular los δ_k, para luego "propagarlos hacia atrás" en la red para obtener los δ_j.

3.3. Algoritmo de *error back-propagation*

A esta altura hemos establecido una manera de computar el gradiente, y por lo tanto de computar la actualización de los pesos en la Eq.(5). Resumimos el procedimiento completo en el algoritmo de *error back-propagation* detallado en el Algoritmo 1. el cual recibe como entrada un conjunto de ejemplos de entrenamiento $\mathcal{T} = \{(\mathbf{x}_n, \mathbf{t}_n)\}_{n=1}^{N}$, la taza de aprendizaje η, y la cantidad de unidades ocultas M:

Algorithm 1 ERROR-BACK-PROPAGATION($\mathcal{T}$, η, M)

1: Crear la red *feed-forward* con arquitectura D, M, K.
2: $\mathbf{w} \longleftarrow$ Inicializar los pesos en valores aleatorios pequeños (e.g., $[-0{,}05, 0{,}05]$)
3: Hasta que la condición de terminación se satisfaga hacer:
4: para cada $(\mathbf{x}_n, \mathbf{t}_n)$ hacer:
5: $\mathbf{w} \longleftarrow$ WEIGHT-UPDATE$(\mathbf{x}_n, \mathbf{t}_n, \eta, \mathbf{w})$

Donde la subrutina WEIGHT-UPDATE es la encargada de actualizar los pesos y se presenta en detalle en el Algoritmo 2.

Para concluir, debemos determinar la condición de terminación. En el caso del perceptrón teníamos garantía de que el algoritmo convergería en un número de pasos finito (siempre que el input fuera linealmente separable). Aquí sin embargo no tenemos tal garantía, y por lo tanto debemos determinar cuando cortarlo.

3.4. Condición de terminación

Para el algoritmo de error back-propagation no existe garantía de convergencia. Sin embargo, debido a que procede en la dirección contraria al

Algorithm 2 WEIGHT-UPDATE($\mathbf{x}_n, \mathbf{t}_n, \eta, \mathbf{w}$))

1: Propagar input $\mathbf{x}_n$ a través de la red usando

$$\mathbf{z}_j(\mathbf{x}_n) = h\left(\sum_{i=1}^{D} w_{ji}x_i + w_{j0}\right); \qquad \mathbf{y}_k(\phi) = f\left(\sum_{j=1}^{M} w_{kj}\phi_j + w_{k0}\right)$$

2: Evaluar δ_k para cada unidad de salida usando: $\delta_k = y_k - t_{nk}$

3: Evaluar los δ_j propagando hacia atrás los δ_k: $\delta_j = h`(a_j)\sum_k w_{kj}\delta_k$

4: Calcular las componentes del gradiente del error (derivadas con respecto a los pesos) usando:

$$\frac{\partial E_n}{\partial w_{kj}} = \delta_k\phi_j; \qquad \frac{\partial E_n}{\partial w_{ji}} = \delta_j x_i$$

5: Usando estas derivadas, actualizar los pesos de acuerdo a gradiente descendiente:

$$\mathbf{w}^{\tau+1} \longleftarrow \mathbf{w}^{\tau} - \eta\nabla E_n(\mathbf{w}^{\tau}).$$

6: Devolver $\mathbf{w}^{\tau+1}$

gradiente, el error siempre se verá reducido. Entonces, una condición de terminación bien sencilla sería cortar el algoritmo cuando este error haya caído por debajo de cierta cota. Este error, sin embargo, es el error sobre el conjunto de entrenamiento, y como ya hemos discutido anteriormente, la reducción excesiva del error de entrenamiento hasta incluso llegar a hipotesis consistentes con error nulo, no hace mas que sobre ajustar los parámetros del modelo (i.e., los pesos) a las idiosincracias del ruido del conjunto de entrenamiento. Como ya discutimos para el caso de regresión polinomial en el primer capítulo, existen dos alternativas para evitar el sobre-fiteo. La primera es *regularización*, que penaliza modelos complejos, i.e., modelos con parámetros excesivamente grandes. Para redes neuronales, una regularización de este tipo muy conocida es la *weight-decay* que propone elegir un número M bien grande de neuronas ocultas (con potencial sobre-ajuste), pero penalizar pesos muy grandes en la función error de la siguiente manera:

$$\widetilde{E}_n(\mathbf{w}) = E_n(\mathbf{w}) + \frac{\lambda}{2}\mathbf{w}^T\mathbf{w}.$$

La segunda alternativa para evitar sobre-fiteo es la *terminación temprana por validación*, que termina el algoritmo cuando se alcanza el mínimo en el

error sobre un conjunto de *validación*, que se separa del de entrenamiento previo al aprendizaje. Recordamos que en la práctica, cuando el conjunto de entrenamiento es pequeño, puede recurrirse a la alternativa de *validación cruzada* k, que divide al conjunto de entrenamiento $\mathcal{T}$ en k particiones $\mathcal{T}_k$ de tamaño N/k, se entrena sobre $\mathcal{T} - \mathcal{T}_k$, y se valida sobre $\mathcal{T}_k$.

4. Alternativas a gradiente descendiente

Sin dudas, gradiente descendiente, si bien el mas simple, no es el único método de minimización numérica. Una discución completa sobre estas alternativas escapa al alcance de estas notas, pero nombramos aquí algunas para guiar al interesado en el estudio de estas alternativas. Muchas de estas alternativas requieren la evaluación del Hessiano, i.e., $\frac{\partial^2 E}{\partial w_{ji} \partial w_{lk}}$. Interesantemente, el truco de propagar hacia atrás el error también aparece en estas derivadas segundas. El desarrollo completo es complejo, y puede leerse en su totalidad en la sección 5.4 de [1].

Otra alternativa muy útil en la práctica, y de facil implementación, es la de agregar un término de *momento* al gradiente descendiente. Si denotamos por $\Delta \mathbf{w}^\tau = \mathbf{w}^{\tau+1} - \mathbf{w}^\tau$, el término de momento consiste en $\alpha \Delta \mathbf{w}^{\tau-1}$, $0 \leq \alpha \leq 1$, quedando la siguiente forma para la actualización por gradiente descendiente

$$\Delta \mathbf{w}^\tau = -\eta \nabla E_n + \alpha \Delta \mathbf{w}^{\tau-1}.$$

El efecto del término de momento es contrarestar el efecto del gradiente, tendiento a que que el vector de pesos mantenga su dirección. Quizás un mejor nombre hubiera sido término de *inercia*. Esta inercia tiene una doble ventaja. Por un lado, en situaciones de mínimos locales, el término de inercia puede hacer que los pesos "sigan de largo" haciendo caso omiso al gradiente, pudiendo así escapar del mínimo local. Además, en situaciones donde el gradiente es constante, vá incrementando el tamaño del cambio en cada paso. Por ejemplo, si $k = \nabla E_n$, tenemos que

$$\Delta \mathbf{w}^{\tau+1} = -\eta k + \alpha \Delta \mathbf{w}^\tau = -\eta k - \eta k + \alpha \Delta \mathbf{w}^{\tau-1} = \ldots = -m\eta k + \alpha \Delta \mathbf{w}^{\tau-1}.$$

luego de m iteraciones.

En particular, esto es muy útil para situaciones donde el gradiente es sumamente pequeño como en una meseta, ya que de no existir momento, el cambio se mantendría constante y muy pequeño, demorando muchas iteraciones para poder salir.

4.1. Temas pendientes

Invariancias y derivación de las funciones de error por medio de métodos Bayesianos

5. Ejercicios

1. Considere una red de una capa oculta cuya formulación matemática esta dada por la Eq. (3), donde la función de activación h de la capa oculta esta dada por un función logistica de la forma

$$h(a) = \sigma(a) = \{1 + exp(-a)\}^{-1}.$$

   Demuestre que existe una red equivalente, que computa exactamente la misma función, pero con funciones de activación h de la capa oculta dadas por $tanh(a) = \frac{e^a - e^{-1}}{e^a + e^{-a}}$. (Ayuda: Primero encuentre la relación entre $\sigma(a)$ y $tanh(a)$, y luego demuestre que los parametros de las dos redes difieren tan solo por transformaciones lineales.)

2. Demuestre que la derivada de la función error respecto de a_k es igual a $y_k - t_k$, tanto para cuando las unidades de salida tienen función de activación logística, como para cuando tienen función de activación softmax.

3. Considere una red multicapa con una capa oculta, que además de las conexiones entre cada unidad de entrada y cada unidad oculta, y las conexiones entre cada unidad oculta y cada unidad de salida, posee conexiones que saltean la capa oculta y conectan directamente cada unidad de *entrada* con cada unidad de *salida*. Extienda el desarrollo presentado en la Sección 3 para encontrar las derivadas de la función error respecto a los pesos de estas nuevas conexiones.

4. Considere una red multicapa con dos capas ocultas. Extienda el desarrollo presentado en la Sección 3 para redes con una sola capa oculta, para encontrar las derivadas de la función error de todos los pesos de la red.

Referencias

[1] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer Verlag, 2006.

[2] P. Y. Simard, D. Steinkraus, and J. Platt. Best practice for convolutional neural networks applied to visual document analysis. In *Proceedings International Conference on Document Analysis and Recognition (ICDAR)*, pages 958–962. IEEE Computer Sociey, 2003.