Clase 4

Funciones de activación

Sigmoidea

- Las funciones sigmoideas no se utilizan como funciones de activación en las neuronas internas.
- Esto es porque, para valores de pre-activación muy grandes o muy chicos (i.e., que tienden a más/menos infinito), sus derivadas son muy cercanas a cero. Entonces, al realizar backpropagation, los gradientes de las capas más tempranas van a ser muy cercanos a cero, es decir, nos vamos a chocar con el problema del **gradiente que se desvanece**.
    - Como los gradientes son muy cercanos a cero, los parámetros no se ajustan porque "ya están en el óptimo".

ReLu

- Es muy utilizada para funciones de activación en las neuronas internas.
- Los valores negativos los hace cero, y a los valores positivos no los cambia. Esto nos asegura tener gradientes mayores a cero y que haya menos probabilidad de que desvanezcan.

Leaky ReLu

- Un problema que tienen las funciones de activación ReLu es que, al convertir todos los valores negativos en cero, puede que perdamos información y la red neuronal no se capaz de aprender los patrones en los datos.
- En vez de convertir en cero a los valores negativos, la función Leaky ReLu los multiplica por un valor cercano a cero. Esto no solo nos ayuda a no perder los valores negativos, sino que también nos permite que los valores negativos también tengan pendiente, así podemos calcular los gradientes.

Inicialización de pesos

- Si inicializamos los pesos en cero, nos va a faltar **asimetría.** Por ejemplo, si inicializamos todos los pesos en cero, todos los kernels de una capa convolucional van a ser iguales y, por lo tanto, no van a poder generar diferentes representaciones de la imágen de entrada.

- Rompemos esa simetría e inicializamos los pesos utilizando una distribución normal con media cero y varianza a elección.
  - Hay varios métodos para elegir la varianza:
    - **Método de Xavier-Glorot (utilizado para funciones tanh).** Cuantas más features tenemos (i.e., cuantos más valores de entrada tenemos), el valor de pre-activación de una neurona se vuelve más grande. Para evitar que se vuelva extremadamente grande, elegimos la varianza de la distribución normal igual a `1/n` (con `n` la cantidad de features o valores de entrada).
      Entonces, si `n` es muy grande, la varianza es muy chica, por lo que estmaos inicializando los pesos con una distribución centrada en cero y con poca varianza (i.e., los pesos de inicialización van a ser chicos). Esto nos asegura que el valor de pre-activación no sea grande.
    - **Método He (utilizado para funciones ReLu).** La varianza de la distribución es igual a `2/n` (muy parecido al método de Xavier-Glorot).

Sobre-ajuste

- Uno espera que, para los muestra de entrenamiento, la curva de aprendizaje del modelo sea estrictamente decreciente, es decir, que la función de pérdida vaya decreciendo después de cada época (epoch).
  - Esto porque en cada época ajustamos los parámetros para que minimice la función de pérdida. Siempre se puede mejorar un poco más.
- Esto no ocurre para la muestra de validación. La muestra de validación decrece hasta cierto punto, a partir del cual empieza a crecer. Ese punto, en donde se alcanza el mínimo, es muy importante porque marca el punto a partir del cual empezamos a sobre-ajustar los datos.
  - En ese punto el modelo deja de generalizar a los datos y comienza a aprender los datos de entrenamiento.

Regularización

- Llamamos regularización a cualquier método que apliquemos para poder mejorar la capacidad de generalización del modelo, es decir, mejorar el rendimiento sobre la muestra de validación.

Early Stopping:

- Monitorea el rendimiento del modelo sobre la muestra de validación. Cuando el rendimiento del modelo comienza a empeorar (i.e., el error de validación empieza a crecer), frenamos.

    - Se utiliza un parámetro de "paciencia", que básicamente define cuantas épocas vamos esperar antes de frenar el entrenamiento. Esto porque puede ocurrir que el error de validación comience a crecer pero luego de algunas iteraciones vuelva a caer.
    - Si el error de validación vuelve a decrecer, entonces no paramos el entrenamiento. Si el error sigue creciendo después de pasadas la cantidad de épocas definidas, el entrenamiento se termina y nos quedamos con el modelo que mejor logró generalizar a los datos.
- Otra técnica es entrenar el modelo una cierta cantidad de épocas (e.g., entrenarlo 50 épocas), y después de cada época guardar el modelo resultante. De esa manera podemos volver atrás y quedarnos con el mejor modelo de los que entrenamos en caso de que el modelo final sobre ajuste.

Weight Decay:

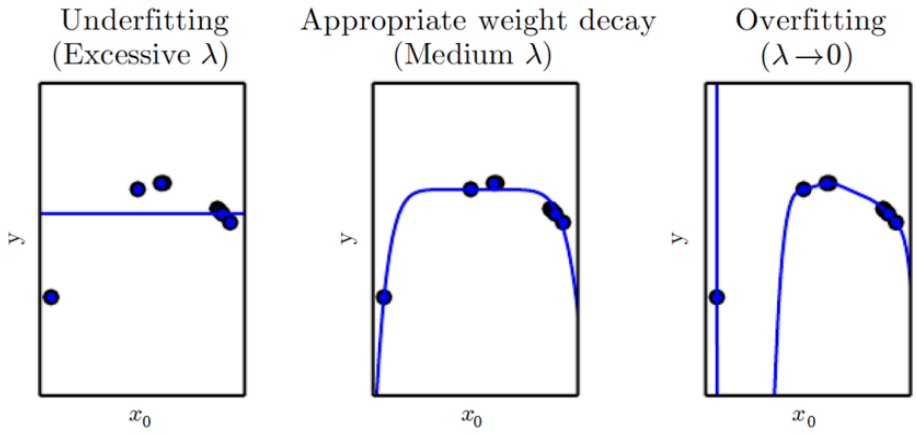
- Sabemos que cuanto más complejo es el modelo, mayor es la probabilidad de sobre-ajuste. Entonces, una manera de evitar que el modelo sobre-ajuste es reduciendo la complejidad del modelo, por ejemplo, eliminando neuronas o pesos.
    - En este método lo que se propone es agregar un término de penalización (al igual que en Lasso o Ridge) a la función de pérdida. Ahora buscamos minimizar la siguiente función:

$$\|\mathbf{w}\|_2^2 = \left(\left(\sum_{m=1}^{M} w_m^2\right)^{1/2}\right)^2 = \sum_{m=1}^{M} w_m^2$$

$$\hat{\mathbf{w}} = \arg\min_{\mathbf{w}} \frac{1}{N} \sum_{n=1}^{N} \mathcal{L}((\mathbf{x}, d)_n; \mathbf{w}) + \frac{1}{2}\lambda||\mathbf{w}||_2^2$$

N = Número de data samples en el training set **M = Número de parámetros en w**

- Al incluir el término de penalización (i.e., la norma de la matriz de pesos), estamos agregando un trade-off al problema de minimización: para minimizar la función de pérdida necesitamos pesos grandes (lo que lleva al sobre-ajuste) mientras que para minimizar el término de regularización necesitamos pesos chicos.
- El parámetro `lambda` nos permite definir cuanto penalizamos, es decir, cuanta importancia le estamos dando al término de penalización. Si `lambda` es muy grande, los pesos son muy chicos y el modelo no aprende. Si `lambda` es muy chico, entonces casi no hay regularización y el modelo sobre-ajusta



Aumentación de datos:

- La idea es generar nuevos datos a partir de los datos que tenemos. Por ejemplo, para el caso de las imágenes, podríamos generar nuevas imágenes al escalarlas, rotarlas, agregarles ruido Gaussiano, etc.
  - Para nosotros la imágen es la misma, pero para el modelo no. El modelo ve esas imágenes como nuevas observaciones a clasificar.

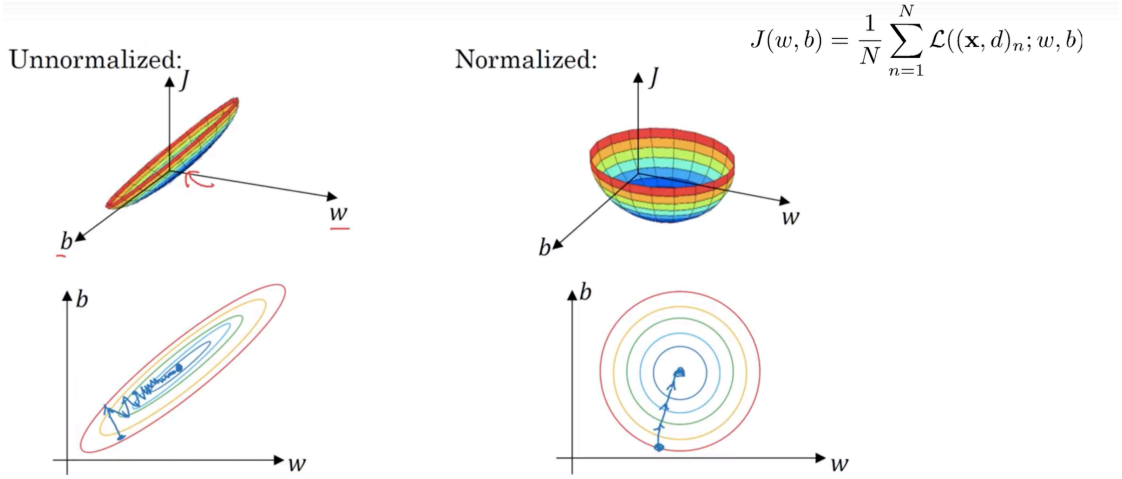
Dropout:

- La idea es, durante la pasada forward en el entrenamiento, ir apagando neuronas de manera aleatoria. De esa manera evitamos depender o "confiar" demasiado en una sola o pocas neuronas.
  - Para implementar este método lo que hacemos es samplear un vector de una distribución Bernoulli con tantos elementos como valores de entrada. Los valores que quedan en cero son los que apagamos (i.e., no utilizamos) y los que quedan en uno son los que utilizamos.
  - Durante la evaluación del modelo se utilizan todas las neuronas.
- Podemos utilizar este método para tener una idea de la incertidumbre de las predicciones de la red neuronal. Para eso, realizamos un par de predicciones en nuestra muestra de evaluación con dropout prendido.
  - Si las predicciones varían mucho, entonces quiere decir que nuestra red neuronal depende mucho de una(s) de las neuronas.

Normalización:

- **Normalización de las entradas.**

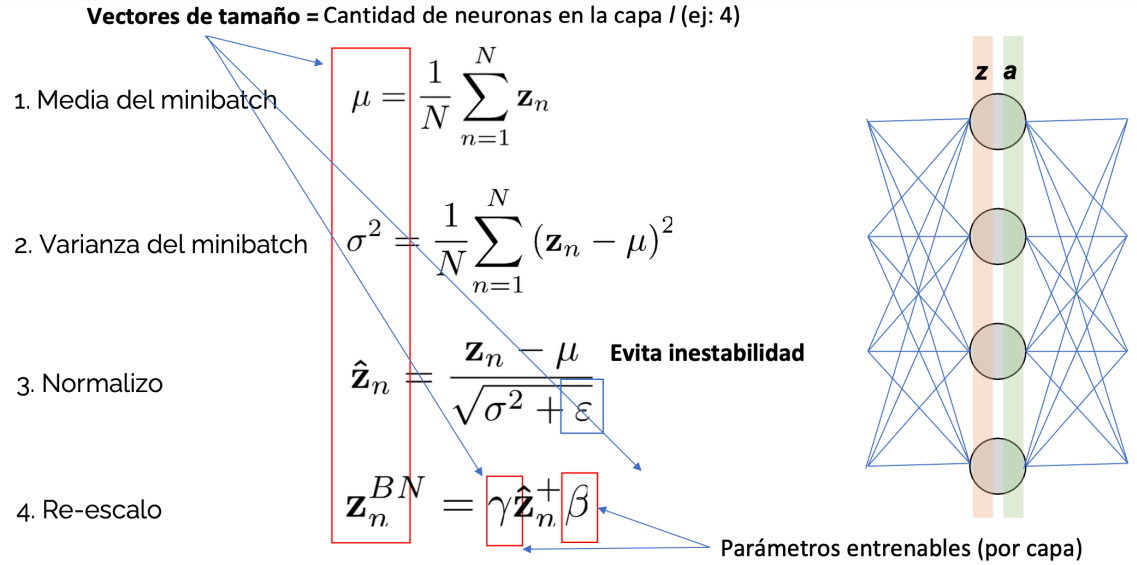
- La manera más sencilla de normalizar datos tabulares es restar la media (para centrar la distribución) y dividir por el desvío. Esto modifica el espacio en donde estamos trabajando, lo que ayuda a que nuestro algoritmo de optimización converja más rápido.



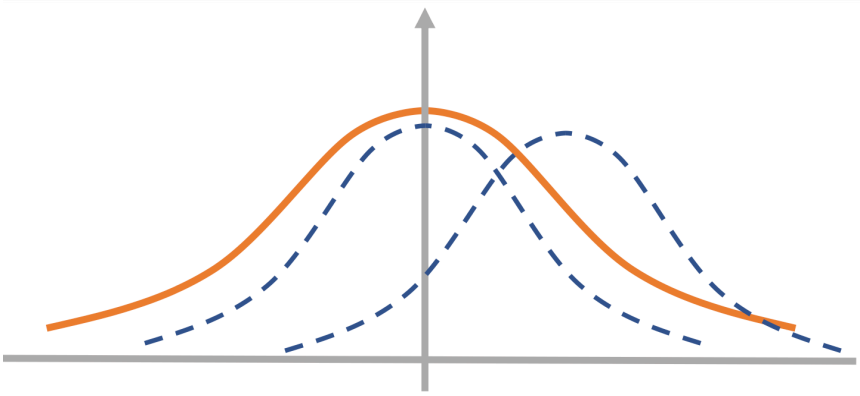
- Para normalizar imágenes no es necesario dividir por el desvío estándar. Lo que hacemos es restar únicamente por la media.

- **Batch Normalization.**
  - En este caso, el proceso de normalización ocurre dentro de cada neurona, previo a aplicar la función de activación. Es decir, lo que normalizamos son los valores de pre-activación.
  - Los estadísticos que utilizamos para la normalización (i.e., media y varianza) se calculan utilizando el batch y no toda la muestra.
    - Para cada neurona vamos a calcular una media y una varianza. La media es el promedio de los valores de pre-activación de la neurona para las observaciones de nuestro batch. La varianza se computa de la misma manera.

- Normalizamos restando la media y dividiendo por la varianza (más un término `epsilon` que evitar inestabilidad por redondeos). Finalmente, reescalamos la preactivación multiplicando por `gamma` y sumando `beta` (son parámetros que aprendemos en el entrenamiento).



- El re-escalado nos permite modificar la distribución de las pre-activaciones de cada neurona:



De esa manera los valores de pre-activación se encuentran dentro de un rango más estable (e.g., para una función sigmoidea necesitamos que los valores de preactivación esten entre -2.5 y 2.5 para que los gradientes no sean cero). Esto también agrega un poco de ruido al proceso de entrenamiento, lo que puede generar efectos de regularización.

- Para la muestra de evaluación no tenemos un batch, solo tenemos una observación. Entonces, si queremos obtener la misma normalización que utilizamos durante el entrenamiento pero ahora sobre la muestra de evaluación, no podemos.

- Para eso, durante el entrenamiento, computamos la **media móvil exponencial** después cada iteración, utilizando los medias del paso anterior. La fórmula es la siguiente:

$$\hat{\mu}_t = \alpha\hat{\mu}_{t-1} + (1-\alpha)\mu_t$$

donde `alpha` es un parámetro que determina cuanto nos importa la media pasada (es decir, cuanto queremos que influya la historia).

Variantes de Gradiente Descendiente

Gradiente decendiente con Momentum

- En vez de actualizar los pesos con el gradiente, los actualizamos con un **vector de velocidad.** Ese vector de velocidad combina el gradiente actual con la memoria de los gradientes anteriores.

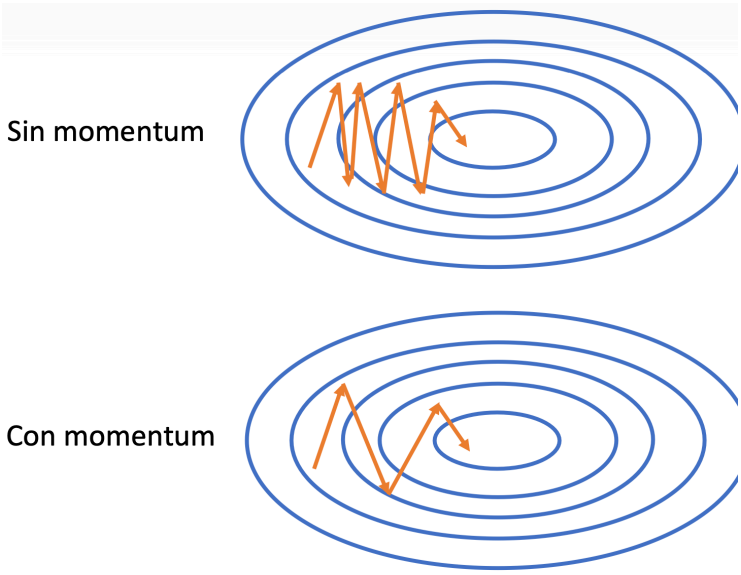
Alfa suele tomarse = 0.9 (promedio 10 últimos)

$$\mathbf{V}_t = \alpha\mathbf{V}_{t-1} + (1-\alpha)\nabla_{\mathbf{W}}\mathcal{L}(\mathbf{W})$$

Los pesos se actualizan con el vector de velocidad:

$$\mathbf{W} = \mathbf{W} - \delta\mathbf{V}_t$$

  - Eso nos permite tener memoria y poder movernos en la dirección en la que se reduce la función de pérdida. Notar que en la imágen, cuando no tenemos momentum, el camino del gradiente es más sinuoso. En particular, porque nos movemos mucho sobre un eje en el que no tenemos que movernos mucho.
  - Cuando agregamos momentum, el camino del gradiente es menos sinuoso y su dirección es mucho más directa.



- Nos permite suavisar el camino que tomar el gradiente decendiente, lo que nos permite llegar más rápido al mínimo.

Variar el learning rate

- Si la tasa de aprendizaje es muy grande, el algoritmo oscila caóticamente y puede ocurrir que no lleguemos al mínimo. Si la tasa de aprendizaje es muy chica, el algoritmo tarda mucho tiempo en llegar al mínimo (o incluso dejar de aprender).
- **Step decay.** Reducimos la tasa de aprendizaje cada cierta cantidad de épocas, o cuando el error en validación no mejora.
- **Exponential decay.** Reducir la tasa de aprendizaje de manera exponencial.
- **LR Adaptativo.** En vez de aplicar una tasa de aprendizaje global (es decir, un escalar para todos los parámetros), tenemos una tasa de aprendizaje para cada parámetro (peso), los cuales se van adaptando en cada iteración.

Transferencia de aprendizaje

- Esta técnica nos permite utilizar modelos que fueron entrenados con un conjunto de datos grande y para una tarea específica, y re-entrenarlos con un conjunto de datos más chico para una tarea distinta.
  - Por ejemplo, la red neuronal AlexNet fue entrenada con datos de ImageNet para clasificar imágenes. Pero si ahora queremos clasificar una clase que no se encuentra en los datos de ImageNet, nuestra red no estaría, en principio, entrenada para clasificar esa clase. Eso no quiere decir que no la podemos utilizar: podríamos tomar un conjunto de datos con las imágenes que queremos clasificar, y re-entrenar la red neuronal para que ahora clasifique esas imágenes.

- Para lograr esto hacemos *fine-tuning*. Usualmente no re-entrenamos todo el modelo, sino las últimas capas. Por ejemplo, para una red neuronal convolucional, la parte convolucional que extrae las características la dejamos intacta y re-entrenamos el clasificador (MLP).
  - La ventaja de hacer fine-tuning sobre el final de nuestra red es que evitamos sobre-ajustar los datos. Es decir, logramos generalizar mejor. Especialmente porque usualmente el fine-tuning se hace con pocos datos.