Regularización

- Idea general: dar preferencia a algunas funciones sobre otras dentro de todas las que son posibles de aprender a partir de los parámetros
- *Regularización: cualquier modificación que le hacemos a un algoritmo de aprendizaje para reducir su error de generalización pero no el error de entrenamiento ==> la capacidad se mantiene intacta*
- Generalmente la regularización se logra poniendo penalizaciones o restricciones sobre los parámetros.
    - que codifican algún conocimiento previo acerca de los posibles modelos que debieran fucionar bien, o
    - que hacen elegir soluciones más "simples" sobre soluciones más complejas (Navaja de Occam)
    - que combinen varias hipótesis (funciones) complementarias para hacer una predicción

- En DL la mayoría de los métodos de regularización son regularizaciones a los estimadores de parámetros (por ejemplo al estimador de máxima verosimilitud)
- Idea:
    - disminuir variance (disminuir overfitting) intentando no aumentar significativamente el bias (no aumentar el underfitting)

La más típica estrategia hoy en día para DL:
- Construir un modelo con mucha capacidad ==> poca probabilidad de underfitting
- Regularizar el modelo agresivamente ==> poca probabilidad de overfitting

==========

1 Regularización por penalizaciones al tamaño de los parámetros

- Función de error pérdida no regularizada:

    L = 1/N * sum(error(y_pred_i, y_i))

- Función de error regularizada:

    Lreg = 1/N * (sum(error(y_pred_i, y_i)) + alpha * penalty(Theta))

- penalty(Theta) en general es una función de alguna norma de los parámetros mientras alpha es un (hiper)parámetro que gobierna la importancia que le damos a la penalización

- La más común de las penalizaciones es l2-norm-penalty. Si Theta = W1,b1,...,WL,bL,U,c, l2-norm-penalty(Theta) es:

    l2-norm-penalty(Theta) = 1/2 * (sum(W1 * W1) + ... + sum(WL * WL) + sum(U * U))

- Note que solo penalizamos los pesos y no los biases!
- Idea:

* El nuevo descenso de gradiente tiende a hacer decaer el valor de los pesos. Simplemente note que

$$dLreg/dWi = dL/dWi + alpha/N * Wi$$

* Esto implica que el nuevo paso en el descenso de gradiente será:

$$Wi := Wi - lr * dL/dWi - lr * alpha/N * Wi$$

o equivalentemente

$$Wi := lambda * Wi - lr * dL/dWi$$

con lambda = (1-alpha*lr/N)

- Por esto a la regularización con l2-norm-penalty se le llama también "weight decay"
- Por qué podría funcionar para aliviar el overfitting? principalmente porque tendrá un sesgo por modelos más simples (más parecido a un modelo lineal).
- Note que, como toda regularización, no impide a la red aprender funciones altamente complejas (con pesos muy altos), pero el impacto de la complejidad debe ser suficiente como para contrarrestar la penalización de pesos.

- Otra norma que se usa para penalizar es la norma l1, usando l1-norm-penalty:

$$l1\text{-}norm\text{-}penalty(Theta) = sum(W1) + ... + sum(WL) + sum(U)$$

- En este caso el gradiente de Lreg cambiará (adicionalmente al gradiente de L) con el signo de cada parámetro elemento a elemento.

2 Regularización agregando más datos:

- La manera más efectiva de disminuir el error de generalización es agregando más datos al train set
- Generalmente agregar más datos es el paso más costoso (necesito humanos!)
- Dos técnicas:
  - ir a buscar etiquetas que estén naturalmente presentes en los datos (ejemplo idioma)
  - generar datos sintéticos, por ejemplo:
    * rotar, trasladar una imagen
    * añadir ruido a un audio
    * agregar pequeños cambios a texto

3 Regularización por Early Stopping:

- Comparamos el error de train y dev durante entrenamiento
- Elegimos el modelo con menor dev error sin importanr que no se de en la última iteración

[DIBUJO DE EARLY STOPPING]

- Podemos ver la cantidad de iteraciones (epochs) como otro hiperparámetro
- Early stopping es muy barato, simple, casi 0 overhead, siempre se debería hacer
- Una vez decidido el tiempo de entrenamiento óptimo T, podemos volver a entrenar el modelo por T iteracionoes con todos los datos de train+dev y elegir ese modelo.
- **Cuidado con Early stopping**: usándolo estamos simultáneamente afectando el train y dev error (ya no funciona nuestra receta general para ML)

4 Regularización por agrupación y bootstrap (Ensemble + bagging):

- Idea: muchos modelos distintos entrenados sobre el mismo conjunto cometerán errores complementarios
- Si considero como clasificador la votación de mayoría de varios métodos, en esperanza el error disminuirá comparado con cada uno por separado.
- Los clasificadores que usan esta estrategia se llaman métodos de agrupación (ensemble methods)
- En la práctica se puede usar no sólo votación si no el promedio, el mínimo o cualquier función que haga sentido en la clasificación
- La técnica se puede generalizar a un "meta entrenamiento" en donde se aprende la forma de combinar los resultados con otro clasificador (una red neuronal de redes neuronales que han sido entrenadas de manera independiente con diferentes arquitecturas). Esta técnica se llama **stacking**

- Si entrenar, y encontrar hiperparámetros para redes neuronales (profundas) es costoso, será mucho más costoso tratar de entrenar redes complementarias! :-(
- Bootstrap (o bagging por bootstranp averaging) es una técnica para alcanzar las buenas propiedades de Ensembles sin tener que usar muchos modelos distintos.
- Idea: modificar los datos de entrenamiento de manera que ciertos modelos sean buenos en ciertos aspectos de los datos y otros modelos sean buenos en otros aspectos.
    - Se construyen K conjuntos de datos del mismo tamaño que el train set, pero se construyen con un sampleo con remplazo, asegurando que con alta probabilidad los conjuntos de datos serán distintos.
    - Luego se entrena K veces el mismo modelo sobre cada uno de los conjuntos distintos, resultando en K estrategias diferentes para predecir que pueden combinarse en un ensemble de las predicciones de cada uno.

- Las competencias de machine learning en general son ganadas por modelos que construyen agrupaciones de varios métodos entrenados

5 Regularización por Dropout:

- Idea: aproximarse a una agrupación de una cantidad exponencial de redes neuronales
- Dropout consiste en entrenar una agrupación de todas las posibles sub-redes que se pueden formar eliminando neuronas de una arquitectura base
- Es muy simple, barata y fácil de implementar: solo tenemos que multiplicar por 0 la salida de las neuronas que queremos dejar de considerar en la sub-red que estamos entrenando en ese momento
- Importante: debemos escalar por la porción de neuronas que estamos dejando fuera para que los valores esperados se mantengan estables (a la salida de las capas)

- Implementación (inverted dropout):
    * para la capa de input y cada capa escondida, elige una probabilidad pi (probabilidad de mantener una neurona)
    * cada vez que se va a pasar un minibatch de ejemplos a la red, se elige una "máscara de bits" del tamaño de cada capa, usando la probabilidad pi correspondiente
    * sea Mi la máscara de bits elegida para la capa i, las nuevas capas se calculan como:

        - h0 := x * M0/p0
        - hi := fi(h(i-1)Wi + bi) * Mi/pi

    * se hace toda la pasada forward y luego el backpropagation considerando el factor Mi/pi
    * se actualizan los parámetros y se repite el procedimiento eligiendo una nueva máscara en cada iteración (para cada minibatch)
- En test time, simplemente se olvida todo el dropout y se hace la predicción
- Los valores usuales para dropout son: mayores o iguales a 0.8 para la capa de input y mayores o iguales 0.5 para la capas escondidas, pero estos son hiperparámetros a encontrar

- Por qué funciona?
    * es como promediar varios modelos independientes (casi, porque se están compartiendo parámetros!)
    * otra interpretación: la red no puede adaptarse a un conjunto fijo de features para hacer las predicciones, debe ser robusta a "apagar algunas neuronas".

- Hoy en día el default para DL (con muchos ejemplos) es:
    * gran capacidad (para evitar underfitting)
    * dropout agresivo (para evitar overfitting)