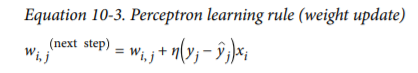
**Perceptron:** Es una idea de neurona muy básica que consta de una capa de input, una capa de pesos y la capa de outputs. Usualmente tiene un **step function** para que pueda predecir un caso binario. Cuando los inputs y outputs son números, se les llama LTU (linear threshold units), luego de esto se suman todos los pesos con las variables y se le pasa la step function para predecir.

**Multi-Layer Perceptron:** Es una red de perceptrones LTU que consta de diferentes capas donde cada una de ellas tiene un nodo como bias. La necesariedad del bias se puede entender como cuando en una función linear se le pone una constante para que pueda moverse a lo largo del eje y. En este caso, el nodo bias hace que podamos movernos a través de las predicciones. La idea principal de este tipo de arquitectura es que va a **iterar** por sobre los datos, y en cada iteración hará predicciones. Estas predicciones se miden y se ven **si son correctas**, en caso lo sean se refuerza la conexión con las neuronas que también nos dicen que son correctas y que han contribuido a que me digan que son correctas.

La regla de aprendizaje del perceptrón tiene la forma: 

**OJO:** Hasta ahora se está viendo una relación lineal dentro de los perceptrones. Eso quiere decir que son imposibles de ver patrones complejos (así como la regresión logística) Esto no siempre es así, porque si las instancias de entrenamiento son linealmente separables Rossenblatt ha demostrado que este algoritmo puede converger a una solución.

**MLP y BackPropagation:** Básicamente la idea de backpropagation es ver junto con derivadas parciales (autodiff en tensorflow) qué tanto afectan las variables a los errores de los outputs. Estas gradientes se propagan hacia todas las capas anteriores y se puede ver y corregir los pesos de la variable que más ha influido en el error. En son de que este algoritmo funcione bien, se pueden cambiar las funciones de conversión. En líneas generales y hasta este momento tenemos 4 funciones conocidas: step function, logistic function, hyperbolic tangent function y ReLu function. Cada capa de la red neuronal tiene una función de activación.

**Importante:** Es importante inicializar los pesos de las conexiones de manera random, en caso contrario (en el que todos sean cero) el algoritmo corregirá de manera igual a todos los pesos y así sucesivamente. Esto sería equivalente a tener una red neuronal de una sola neurona.

Si vas a desarrollar un modelo y tu target tiene la forma de una matriz one hot encodeada, solo tendrás que usar **categorical\_crossentropy** como loss function. Pero si es una matriz de números enteros, entonces usaremos **sparse\_categorical\_crossentropy**.

**Regression MLP:** Con redes neuronales también puedes hacer regresiones. En este caso se tiene que pensar que toda la red neuronal está hecha para predecir en base a un solo registro. Por ejemplo, si quiero predecir el precio de las casas, esa red solo tendrá un nodo en el output. Además, en casos de regresión **no se usan** funciones de activación convencionales, sólo cuando se quiere garantizar que los outputs siempre sean positivos.

Lo que sí se debe de tener en cuenta en estos casos es el loss function, que puede ser una métrica como mean\_squared\_error, sin embargo se debe tener cuidado en no tener muchos outliers en la data. Existe otra función que puede ser útil como **loss\_function**: Huber loss, que si el error es muy grande solo usa valor absoluto, y si es pequeño usa el cuadrático. Así se evita darle mucho peso a los outliers.

**Redes neuronales secuenciales:** Existen redes neuronales secuenciales y no secuenciales. Las secuenciales son las más básicas de estas y se pueden usar en clasificación y en regresión.

**Redes neuronales no-secuenciales:** Podemos hacer diferentes arquitecturas de redes neuronales y hacer que la data pase por las capas que queremos. Por ejemplo, podemos hacer una red que pase data por dos layers y a su vez haya data que pase directamente a la capa que sigue a los dos layers iniciales. Esto también nos puede servir para **predecir dos output** dentro de una red neuronal. Por ejemplo, un output de clasificación y otro de regresión en la misma red neuronal. Como los dos outputs tienen diferentes lógicas, es intuitivo pensar que la información pase por diferentes neuronas.

**Callbacks:** Te permite llamar objetos mientras la red esté entrenando. Esto puede ser útil porque una RNN puede demorar bastante en entrenar. Imagínate que demora 6 horas pero en las últimas 4 horas no mejora tanto el accuracy. Podemos establecer código para que guarde los parámetros de la red y guardar los resultados antes de que termine todo el entrenamiento. En Keras existe el objeto keras.callbacks donde puede hacer tu función específica y meter los callbacks dentro. Uno de las métodos de este objeto es **TensorBoard** que nos permite ver los learning rates de manera muy dinámica, además crea una carpeta fuera donde almacena todos los resultados durante las epochs.

**Cómo los DNN aprovechan la información entre capas:** La mayoría de información es ordenada de forma jerárquica, esto significa que pequeñas abstracciones construyen a las grandes. Esto se traslada a las redes neuronales en la forma en que cada capa abstrae algún concepto y/o característica de la data analizada. Por ejemplo, la primera capa luego del input podría abstraer líneas curvas y rectas, luego la segunda capa extrae círculos y cuadrados y la última capa ya abstraería patrones de reconocimiento facial.

**Transfer learning:** Este último concepto de las capas puede usarse a nuestro favor. Como al final las capas son solo combinaciones de parámetros, bastaría con estos parámetros para poder replicar toda la red en cualquier lugar. Esto nos sirve para poder construir por encima de una red neuronal otra red neuronal. Me explico, envés de entrenar denuevo una Rnn que identifique caras de humanos, podría agarrar una red ya entrenada, trabajar sobre ella y mejorarla.

**Optimizando el número de neuronas por Layer:** Usualmente se usaba el método pirámide, pero se ha visto que no es tan eficiente hacerlo. Ahora se recomienda usar el método de pantalones grandes, es decir, entrenar una red con un número de neuronas mucho mayor que el necesario para eventualmente bajar el número de neuronas un poco y quedarnos con él. Esta red con un número mucho mayor de neuronas hará que nuestro algoritmo no converja, y eventualmente bajaremos hasta que lo haga. Desafortunadamente, un número óptimo de neuronas por ahora es una ciencia oscura…

**Tuneando otros hiperparámetros de la red neuronal (learning rate, Batch Size, etc):**

Acá también podemos usar un Randomized Search para establecer los parámetros óptimos. Aunque se recomiende un learning rate alto que haga diverger la función y en cada iteración dividirlo entre 3 para quedarnos con el que converja en sí. Por otro lado, para optimizar el Batch Size se menciona que **debería estar entre 20 y 32** porque un número muy bajo haría que nuestras gradientes no apunten a la dirección correcta ¿Igualmente con 32?

**El problema de las gradientes (Vanishing/Exploiting):** Dentro del cálculo del backpropagation desafortunadamente las gradientes se vuelven más y más pequeñas. En este caso y, antes del 2010, se observaban dos problemas en general. Primero, que las gradientes en cierto punto del entrenamiento eran muy pequeñas y no ayudaban a mejorar el modelo (vanishing problem) o que las gradientes eran muy grandes y el algoritmo divergía en este sentido (Exploiting gradientes problem) Generalmente las Deep Neural Networks sufren de gradiente inestables en general.

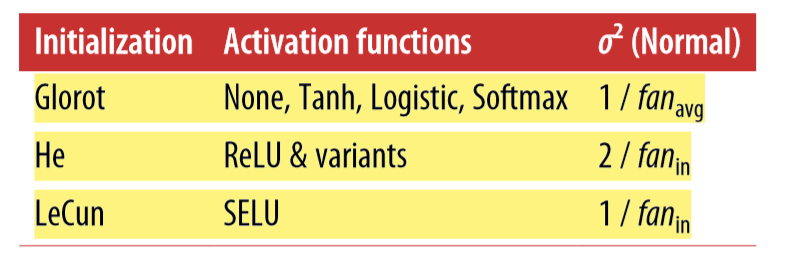
En este sentido Glorot y Bengio sacaron un paper indicando que el problema en general era la **inicialización de los pesos** y la función de activación logística. Además, el método que se usaba en ese entonces para inicializar los pesos era randomizar alrededor de una distribución normal. Esto llevaba a que **la varianza de los outputs sea mucho mayor que la de los inputs.** Además, la función de activación logística hace que a mayores valores de los inputs se obtenga o 0 o 1 dentro de la función, cuya derivada es 0. Entonces como la derivada es 0 no hay gradiente y el algoritmo se queda estancado…

**Glorot, He y LeCun initialization:** El principal objetivo acá es que las gradientes no desaparezcan y que se pueda hacer el ida y vuelta dentro del backpropagation. Para esto, se establece que **la varianza de los ouputs de cada layer tiene que ser igual a la de los inputs,** e igualmente para las gradientes en dirección de ida y de vuelta. Para esto se introduce el concepto **fan-in** y **fan-out** que no es más que el número de variables de entrada y salida.

**Glorot initialization:** Establece una inicialización random con una media de Fan-avg que es (fan-in + fan-out)/2 y una desviación estándar de sqrt(r/fan-avg).

**Le-Cunn initialization:** Se reemplaza en la inicialización de Glorot el fan-avg por el fan-in.

**He intialization:** Usa el sqrt(4/fan-in)



Si quieres usar las diferente initializaciones en Keras, tendrías que usar el parámetro **kernel\_intializer** para poder editarlo de manera fácil.

**Funciones de activación no saturadas:** Hasta ahora solo hemos visto como función de activación la función logística (underperform until 2010) y la función ReLu (la más conocida por todos). Pero nos damos cuenta que la misma función ReLu tiene un problema al lidiar con valores negativos porque siempre retornará 0 como output. Es así que se han creado variantes y derivados de esta función que mejoran el rendimiento de las redes neuronales en ciertos casos:

* **leaky ReLu:** Es un tipo de ReLu que tiene un parámetro α (alpha) acompañado. Este parámetro ayuda a medir qué tanto se va a desviar nuestra función de la ReLu original para que el problema del retorno del 0 sea corregido. Por eso, este parámetro se puede tunear y, usualmente, un parámetro más alto hace que la red neuronal tenga un mucho mejor performance.
* **PReLU(Parametric leaky ReLu):** Este tipo de función se entrena a medida que todo el algoritmo lo hace. Es decir, se buscan valores óptimos para el α durante el entrenamiento.
* **ELU (Exponential Linear Unit):** Como su nombre lo dice, depende de una función exponencial. La fórmula está hecha para que tome valores negativos y no sea cero en ellos haciendo que esto último ayude a lidiar con el problema del vanishing gradient. Pero son difícil de procesar.
* **Self-normalized Elu:** Este tipo de función requiere que se tenga un LeCun initialization y, a su vez, que la red neuronal sea densa y secuencial. Se ha descubierto que este tipo de función de activación hace que se mejor mucho el performance, pero no se garantiza que lo haga dentro de arquitecturas complejas.

**Batch Normalization:** A pesar de que tengamos una función de activación como ElU, los problemas de vanishing y exploits de las gradientes no están solucionados. Por eso, se ha propuesto un método de entrenamiento que parece ser decisivo para solucionar del todo estos problemas: Batch Normalization. A grandes rasgos, esto sirve para poder dar una escala a nuestros inputs (escalar) y meterles un shift a los valores (desviarlo un poco). Esto le permite al modelo establecer el valor **óptimo** de la escala y de la media para los inputs dentro del tiempo de entrenamiento. Pero sucede un problema cuando tratamos de predecir los valores, es decir, ¿vamos a sacar los valores óptimos también para los inputs que usaremos para predecir?. Pues los parámetros óptimos de los inputs se sacarán dentro del training.

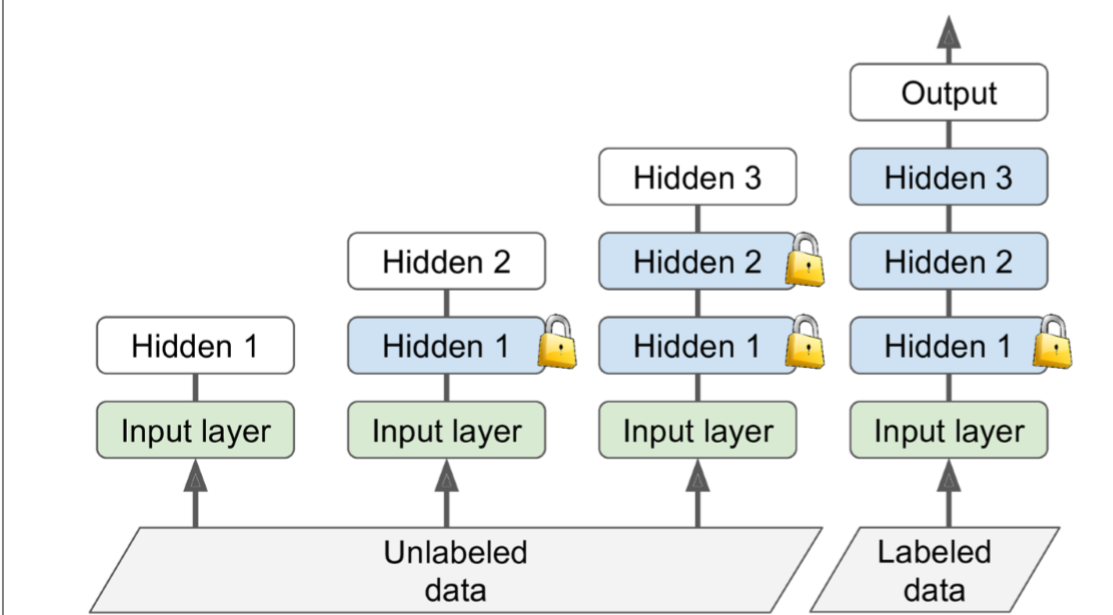
Este método mejora mucho dentro de deep neural networks y es recomendable usarlo necesariamente. Se recomienda añadir layers de Batch Normalization antes que layers de funciones de activación. Esto Keras te lo permite fácilmente. BN se ha convertido en el **algoritmo más usado dentro de Deep Learning,** tanto así que se sobreentiende que se usa en cada arquitectura. Sin embargo, Hongyi Zhang, ha sacado un nuevo método que overperformea al BN: **Fixup normalization,** que genera un buen performance en DNN’s (10 mil layers).

**Gradient Clipping:** Clipear las gradientes es un método para evitar que exploten. Básicamente es establecer un threshold para que se redondeen una vez pasen de cierto número. Por ejemplo, si mi gradiente es (0.9,1000) y mi threshold es 1, entonces esa gradiente ahora será (0.9,1). Nota que hay una transformación importante en el espacio para este tipo de gradiente. Dentro del optimizador en Keras, puedo especificar el clipvalue. Y en realidad existen dos tipos de clippers: clipvalue & clipnorm. Aunque el método de clipvalue puede que no altere mucho tus resultados, puede que no quieras que se cambie la dirección del vector. Es ahí donde usas “clipnorm”, que básicamente no solo transforma el eje faltante sino que transforma todo el vector para que siga apuntando a la dirección que quieras. Recuerda que puedes trackear las gradientes usando TensorBoard.

**Re-usando Layers Pre-entrenados:** Usualmente cuando se encuentra un modelo ya entrenado, podemos re-usarlo para entrenar un modelo que solucione un problema parecido. Recuerda que los layers de arriba almacenan las capas más específicas de abstracción y los primeros layers las más generales, por eso es intuitivo que usemos las capas iniciales y que las **freezeemos,** es decir, que nos las entrenemos más sino que simplemente las usemos. Pero acá surgen algunas preguntas: ¿Cómo sé cuántas capas del otro modelo usaré?

1. Congela primero las capas finales (freeze) y evalúa el performance del modelo.
2. Descongela capa por capa y ve evaluando qué modelo es mejor. Recuerda que es útil reducir el ratio de aprendizaje para que no te pases en corregir los valores iniciales del layer descongelado.

Pero, cuidado! Iterar e iterar hasta que la data confiese significa también forzar al modelo a tener un mejor performance, y eso puede caerse en cualquier momento (overfitting) Estos métodos de transfer learning funcionan mejor cuando son capas profundas o **deep convolutionar neural networks**.

**Pre-Entrenamiento No-Supervisado:** Imagina que no tenemos la suficiente data categorizada como para entrenar nuestro algoritmo. Existen modelos de entrenamiento semi-supervisado que nos pueden solucionar el problema (Restricted Boltzmann Machines y los autoencoders) donde cada layer es entrenado con los outputs que los layers previamente entrenados. Los **autoencoders** en sí son las mejores opciones para que solucionar que no haya data categorizada, sin modelos similares o pequeñas cantidades de datos categorizados y varios datos sin categorizar. La idea principal es que cada layer es entrenado con el output de los layers anteriormente entrenados.

La imagen muestra que secuencialmente se entrenan los inputs con los layers de la data que no está categorizada, luego se le va agregando layers hasta tener una red con la que se pueda entrenar los layers con solo la data labelizada.

**Pre-training en una tarea auxiliar:**  Otra vez, si el problema es la falta de data con labels, entonces podemos entrenar un modelo para una tarea pequeña en la que sí tengamos la información necesaria. Luego de eso, entrenar el modelo reusando los layers anteriores, generalmente los layers de más abajo son los que funcionarán.

**Optimizadores más rápidos:** Hasta ahora hemos visto que elegir buenos inicializadores para los pesos de conexión, usar una función de activación, Batch Normalization y reusar partes de RNNs son métodos para optimizar y sacar mejor performance. Pero elegir bien tu optimizador (y no solamente SGD) también sirve para esto. Por ejemplo, unos optimizadores son:

1. Momentum Optimization: Obtiene las gradientes del vector de momento, y actualiza los pesos simplemente añadiendolo a la ecuación. En otras palabras, este optimizador es para acelerar y no para hacer más rápido el cálculo. El parámetro beta es un símil de la fricción en la aceleración.
2. Nesterov Accelerated Gradient: Es una pequeña variante del método anterior, pero la idea general es no medir la gradiente de la función de costo en la posición local, sino un poco al lado de la dirección del momento. Esto resulta en apuntar un poco mejor al óptimo. En Keras, basta con un parámetro nesterov=True para aplicarlo.
3. AdaGrad: Viene de adaptative gradiente. Le agrega y quita pesos a las ecuaciones de cálculo de las gradientes haciendo que sean más rápidos. Lamentablemente, no funciona muy bien en DNN’s y no se recomienda usar este optimizador. Sin embargo, puede funcionar bien para problemas de regresión lineal. Una de las razones es que para hacer el cálculo tiene en consideración todas las gradientes anteriores.
4. RMSProp: Arregla el problema de la gradiente acumulando solo las gradientes de las más recientes iteraciones. Hace esto agregando un parámetro beta típicamente seteado en 0.9. Este optimizador usualmente tiene mejor performance que el AdaGrad.
5. **Adam & Nadam Optimization:** Combina las ideas de RMSProp y de Momentum optimization. En este caso tiene un b1 que proviene del algoritmo de momentum y tiene b2 que proviene de Rmsprop.
   1. alpha tiene que ser tuneado:
   2. **B1**: 0.9
   3. **B2:** 0.999
   4. **E:** 10^-8

Existen dos variantes de estos algoritmos:

* **Adamax:** reemplaza la escala de parámetros echa por l2 por un l-infinito (Por eso el max) En la práctica, esto hace a Adamax mucho más estable que Adam solamente.
* **Nadam Optimization:** Es una fusión de Adam + Nesterov Optimization.

Nota: Cuando te decepciones de los optimizadores de tu red, prueba solamente con Nesterov Accelerated Gradient. Así podrías mejorar el performance dado que tu data podría ser alérgica a los otros optimizadores.

**Learning Rate Scheduling:** El **learning\_rate** es uno de los más importantes parámetros. Una manera de optimizarlo es reducirlo a medida que vayamos entrenando, este procedimiento puede ser de varias formas. Estas formas se les llama **learning schedules** y a grandes rasgos presentamos tres:

* **Power scheduling:**
* **Exponential scheduling:**
* **Piecewise constant scheduling:** Usa un determinado l\_r por un determinado número de iteraciones, luego redúcelo y así sucesivamente.
* **Performance scheduling:** Mide el error cada ciertas iteraciones y va ajustando los parámetros cada cierto tiempo.

Implementarlo en Keras es muy fácil, ya que solo tienes que setear el parámetro de decay para implementar la función en específico que quieres hacer.

**Técnicas de regularización para evitar el Overfitting:**

**l1 y l2 regularization:**

Keras tiene las funciones de regularización l1 y l2 ya hechas. Solo basta con llamarlas en cada uno de los layers. Pero llamarlos a su vez en cada uno de los layers no es acaso un poco engorroso? Para eso podemos usar functools y especificar un nuevo objeto para usarlo. Dentro de esta especificiación diremos que lo que queremos será un layer con las regulaciones respectivas que quisiera en todos los demás layers.

**DROPOUT:** ojo(Es aplicado tanto a los inputs como a los hidden layers. pero no a outputs)

Es una de las técnicas que más funcionan dentro de una red neuronal y consiste en dropear nodos temporalmente en cada iteración al momento de entrenar. Estos nodos se dropearán con una probabilidad p, que usualmente es en 50%. Pero cuidado, el que se dropeen nodos temporalmente hará también que los pesos se ajusten doblemente así que tenemos dos opciones que funcionan casi igual: dividir entre (1-p) en cada corrección e iteración o multiplicar por (1-p) cada weigth que derive del entrenamiento. Así tendremos una red neuronal con la arquitectura que deseábamos y con los weights entrenados de la manera en que queríamos.

Como el drop-out solo está activo durante el entrenamiento, el training loss es penalizado comparado con el de validación. Así que comparar los dos no sería apropiado. Lo mejor sería evaluar el training loss sin el dropout. Sin embargo, poniendo keras.backend.learning\_phase\_scope(1) fuerza a que el dropout esté activo durante el entrenamiento y la validación.

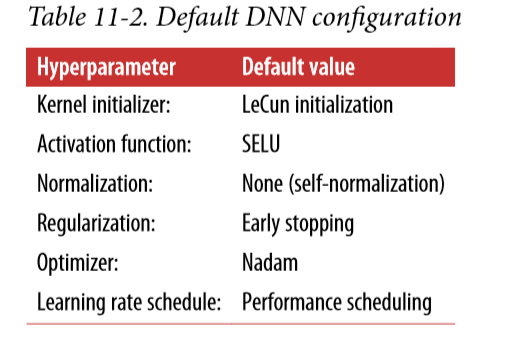
¿Qué signfica que el dropout esté activo? ¿Si es que ya no va a entrenar, por qué lo mantendría activo?

(Recuerda que AlphaDropout por si quiero hacer regularizaciones)

**Monte-carlo dropout**:

Esta técnica consiste en hacer predecir a cada una de nuestras redes creadas por el drop el valor que queramos. Como el dropout está activo (keras.backend.learning\_phase\_scope(1)) todas las predicciones serán diferentes. Por eso, haremos que todas estas redes predigan y sacaremos su promedio. Eso será lo que nuestra red pueda sacar como predicción final.

**Caveat con Batch normalization**: Si tu modelo contiene otros layers que funcionan de manera diferente, como layers con batch normalization, deberías reemplazar los layers de dropout por layers con la clase MCDropout.

**Max-norm regularization:** Es una forma de regularizar los pesos en sí y no su proceso, se usa de manera muy fácil en Keras también para reducir el problema de la gradiente (claro, si no estás usando ya Batch Norm) ¿Se podría combinar con Dropout para excluir Batch Norm del proceso? 

**Resumen del Capítulo:**

Lo que funciona usualmente es:

**Guidelines:**

* Si tu modelo no self-normaliza (Chequear eso luego):
  + Si overfitea el training set, deberías usar **alpha dropout** (y siempre usa early stopping también) No uses otros métodos de regularización, sino podría romper el self-normalization.
* Si tu modelo no self-normaliza:
  + Puedes probar **ELU,** envés de SELU pq tendría mejor performance. Asegúrate de cambiar el método de inicialización acorde al ELU.
  + Si es una DNN deberías usar **Batch normalization** después de cada hidden layer. Si sigue overfitteando, podrías usar max-norm o l2 regularization.
* Si necesitas un sparse model (un modelo que prediga diferentes clases) puedes usar **l1 regularization.** Si necesitas un modelo aún más sparser, tendrías que usar FTRL envés de Nadam optimization, junto con l1 claro. En este caso, vendrían a romper el self-normalization así que tendrías que cambiar a batch normalization.
* Si necesitas un modelo con poca latencia (que demore poco en hacer cada una de las predicciones) no uses batch norm. puedes reemplazarlo con un Selu y con un leaky relu.
* Si estás construyendo un risk-sensitive app, o una app donde la altencia no es muy importante, puedes usar MC Dropout para mejorar el performance y obtener mejores estimaciones.