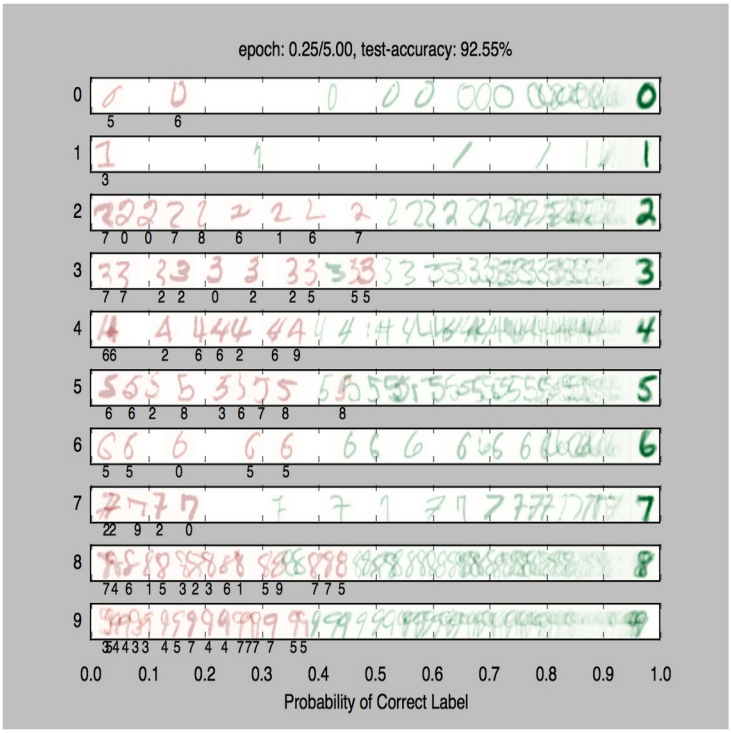
Proyecto 3b: Aprendizaje automático

Tabla de contenido

En este proyecto, construirás una red neuronal para clasificar dígitos, ¡y más!

Introducción

Este proyecto será una introducción al aprendizaje automático. El código de este proyecto contiene los siguientes archivos, disponibles como nn_student.tar.gz.



Archivos que editarás:	
models.py	Modelos de perceptrón y redes neuronales para una variedad de aplicaciones
Archivos que debe leer pero NO editar:	
nn.py	Minibiblioteca de redes neuronales donde encontrarás funciones para inicializar tu red, realizar el productoDot, actualizar los gradientes, calcular el loss (el error), etc. Necesitarás instalar numpy. Para ello emplea pip install numpy.
matplotlib	También necesitarás instalar esta librería para poder graficar tu función del seno. Para ello pip install matplotlib
Archivos que no editarás:	
autograder.py	Autocalificador de proyectos
backend.py	Código de backend para varias tareas de aprendizaje automático
data	Conjuntos de datos para clasificación de dígitos e identificación de idiomas

Archivos para editar y enviar: completará partes models.pydurante la tarea. Por favor, no cambiar los otros archivos en esta distribución.

Evaluación: Por favor, no cambiar los nombres de las funciones o clases establecidas en el código, o se le causar estragos en el autograder.

Instalación

Para este proyecto, necesitarás instalar las siguientes dos bibliotecas:

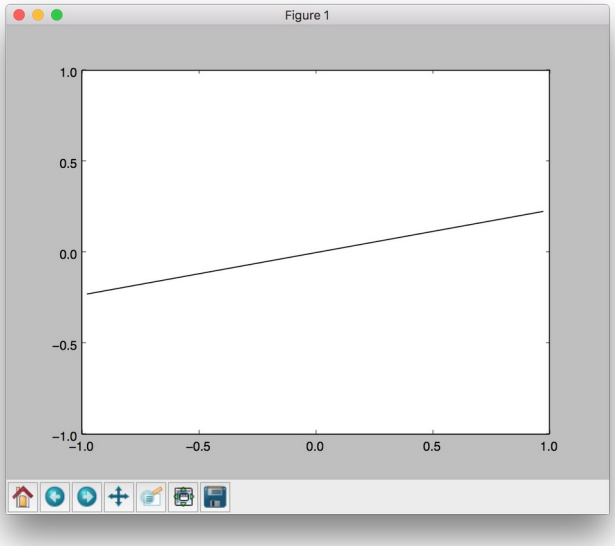
- numpy, que proporciona soporte para grandes matrices multidimensionales: instrucciones de instalación
- matplotlib, una biblioteca de trazado 2D - instrucciones de instalación

No utilizarás estas bibliotecas directamente, pero son necesarias para ejecutar el código proporcionado y el autograder.

Para probar que todo se ha instalado, ejecute:

```
python autograder.py --check-dependencies
```

Si `numpy` y `matplotlib` están instalados correctamente, debería ver una ventana emergente donde un segmento de línea gira en un círculo:



Código proporcionado (Parte I)

Para este proyecto, se te ha proporcionado una minibiblioteca de redes neuronales (`nn.py`) y una colección de conjuntos de datos (`backend.py`).

La biblioteca en `nn.py` define una colección de objetos de nodo. Cada nodo representa un número real o una matriz de números reales. Las operaciones en los objetos Node están optimizadas para funcionar más rápido que con los tipos integrados de Python (como las listas).

A continuación, se muestran algunos de los tipos de nodos proporcionados:

- `nn.Constant` representa una matriz (matriz 2D) de números float. Por lo general, se usa para representar características de entrada o salidas / etiquetas. Otras funciones de la API le proporcionarán instancias de este tipo; no necesitarás construirlos directamente
- `nn.Parameter` representa un parámetro entrenable de un perceptrón o red neuronal
- `nn.DotProduct` calcula un producto escalar entre sus entradas

Funciones adicionales proporcionadas:

- `nn.as_scalar` puede extraer un float de Python de un nodo (convertir el número almacenado en el nodo en un float, para realizar comparaciones).

Al entrenar un perceptrón o una red neuronal, se le pasará un objeto `dataset`. Deberás realizar un entrenamiento por lotes de ejemplos `dataset.iterate_once(batch_size)`:

```
para x, y in dataset.iterate_once (batch_size):

...
```

Por ejemplo, extraigamos un lote o batch **X de tamaño 4** (es decir, un lote formado por 4 ejemplos de entrenamiento x^1 .. x^4) de los datos de entrenamiento del perceptrón donde cada ejemplo x^i viene descrito por 3 rasgos = (x_1 , x_2 , x_3)

```
>>> batch_size = 4

>>> for X, Y in dataset.iterate_once (batch_size):  ← ESTE BUCLE DARÁ TANTAS VUELTAS COMO TAMAÑO DEL CORPUS DE ENTRENAMIENTO dividido por el BATCH_SIZE

... print (X)

... print (Y)

...

<Constant shape = 4x3 en 0x11a8856a0>

<Constant shape = 4x1 en 0x11a89efd0>
```

Las características de las entradas `X` y sus etiquetas correctas asociadas `Y` se proporcionan en forma de nodos `nn.Constant`. La forma de `X` será `batch_size` x `num_features`, y la forma de `Y` es `batch_size` x `num_outputs (número de clases posibles)`. Aquí hay un ejemplo de `Y` cómo calcular un producto escalar consigo mismo, primero como un nodo y luego como un número de Python.

```
>>> nn.DotProduct (X, Y)  este es el producto escalar o punto que es ~ la distancia del coseno

<DotProduct shape = 1x1 en 0x11a89edd8>

>>> nn.as_scalar (nn.DotProduct (X, Y))

1.9756581717465536
```

Antes de iniciar esta parte, asegúrate de que tienes `numpy` y `matplotlib` ya instalado!

Consejos sobre redes neuronales

En este laboratorio implementará los siguientes modelos:

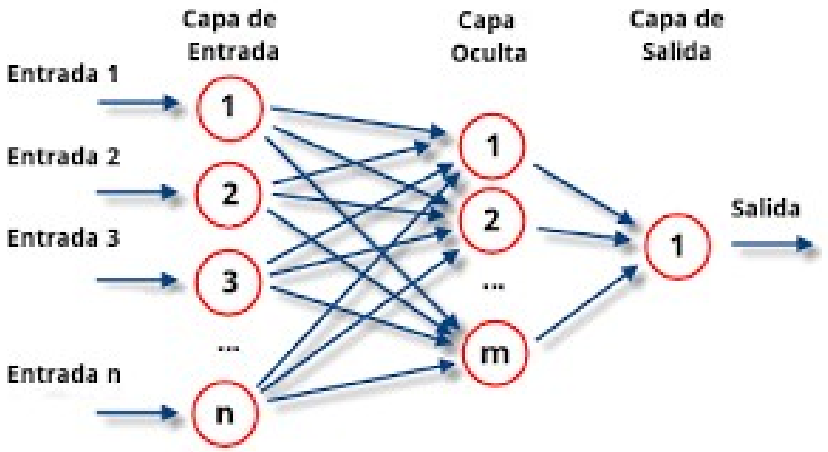
- Q1: Regresión
- Q2: Clasificación de dígitos escritos a mano

Construyendo redes neuronales

A lo largo de la parte de aplicaciones del proyecto, utilizará el marco proporcionado `nn.py` para crear redes neuronales para resolver una variedad de problemas de aprendizaje automático. Una red neuronal simple tiene capas, donde cada capa realiza una operación lineal (como el perceptrón). Primero deberás definir la configuración de la red que quieres emplear y el tamaño del batch, por ejemplo:

```
def __init__(self):
        # Initialize your model parameters here
        self.batch_size = 20
        self.w0 = nn.Parameter(1, 5)
        self.b0 = nn.Parameter(1, 5)
        self.w1 = nn.Parameter(5, 1)
        self.b1 = nn.Parameter(1, 1)
        self.lr = -0.01
```

Esta configuración se correspondería con la siguiente red donde m será 5 y n es el tamaño del batch y aunque no se vea, el número de rasgos según la configuración será 1 (porque en este ejemplo se intentará aprender la función del seno de x, así pues la entrada x vendrá representada por un solo valor, el valor de la propia x:



Las capas están separadas por una ReLU que aporta no linealidad, lo que permite que la red se aproxime a funciones generales (encontraréis una explicación de la ReLu al final de este documento). Usaremos la operación ReLU para nuestra no linealidad, definida como $\text{relu}(X) = \max(X, 0)$. Por ejemplo, una red neuronal simple de dos capas para mapear un vector de fila de entrada X a un vector de salida $S(X)$ estaría dada por la función:

$$S(X) = \text{relu}(X \cdot W_1 + b_1) \cdot W_2 + b_2$$

donde tenemos matrices de parámetros W_1 y W_2 y vectores de parámetros b_1 y b_2 para aprender durante el descenso del gradiente (gradient descent). W_1 será una matriz $i \times h$, donde i es la dimensión de nuestros vectores de entrada X (recordad que para el segundo ejercicio Q2, X vendrá descrito por un solo rasgo o feature que es el propio valor de X) y h es el tamaño de la capa oculta (primera capa). El vector b_1 tendrá un tamaño h. Somos libres de elegir cualquier valor que queramos para el tamaño oculto (solo necesitaremos asegurarnos de que las dimensiones de las otras matrices y vectores concuerden para que podamos realizar las operaciones). El uso de un tamaño oculto más grande generalmente hará que la red sea más potente (capaz de adaptarse a más datos de entrenamiento), pero puede hacer que la red sea más difícil de entrenar (ya que agrega más parámetros a todas las matrices y vectores que necesitamos aprender), o puede conducir al sobreajuste en los datos de entrenamiento.

También podemos crear redes más profundas agregando más capas, por ejemplo, una red de tres capas:

$$S(X) = \text{relu}(\text{relu}(X \cdot W_1 + b_1) \cdot W_2 + b_2) \cdot W_3 + b_3$$

Nota sobre la aleatoriedad

Los parámetros de su red neuronal se inicializarán aleatoriamente y los datos de algunas tareas se presentarán en orden aleatorio. Debido a esta aleatoriedad, es posible que de vez en cuando falle algunas tareas incluso con una arquitectura sólida: ¡este es el problema de los óptimos locales! Sin embargo, esto debería suceder muy raramente: si al probar tu código falla el autograder dos veces seguidas en una pregunta, debes explorar otras arquitecturas.

Consejos prácticos

El diseño de redes neuronales puede requerir algo de prueba y error. Aquí hay algunos consejos que le ayudarán en el camino:

- Se sistemático. Manten un registro de cada arquitectura (tamaño de la capa oculta, numero de capas, que haya probado, cuáles fueron los hiperparámetros (tamaños de capa, tasa de aprendizaje, etc.) y cuál fue el rendimiento resultante. A medida que pruebes más cosas, podrá empezar a ver patrones sobre qué parámetros son importantes. Si encuentra un error en su código, asegúrate de tachar los resultados anteriores que no sean válidos debido al error.
- Comienza con una red poco profunda (solo dos capas, es decir, una no lineal). Las redes más profundas tienen exponencialmente más combinaciones de hiperparámetros, y equivocarse

incluso una sola puede arruinar su rendimiento. Utilice la red pequeña para encontrar una buena tasa de aprendizaje y tamaño de capa; luego, puede considerar agregar más capas de tamaño similar.

- Si tu tasa de aprendizaje es incorrecta, ninguna de sus otras opciones de hiperparámetros importa. Puede tomar un modelo de vanguardia de un trabajo de investigación y cambiar la tasa de aprendizaje de manera que no funcione mejor que al azar. Una tasa de aprendizaje demasiado baja dará como resultado que el modelo aprenda demasiado lento, y una tasa de aprendizaje demasiado alta puede hacer que la pérdida diverja hasta el infinito. Comience probando diferentes ritmos de aprendizaje mientras observa cómo la pérdida disminuye con el tiempo.
- Los lotes más pequeños requieren tasas de aprendizaje más bajas. Cuando experimente con diferentes tamaños de lote, tenga en cuenta que la mejor tasa de aprendizaje puede ser diferente según el tamaño del lote.
- Abstenerse de hacer que la red sea demasiado amplia (los tamaños de las capas ocultas son demasiado grandes) Si sigue ampliando la red, la precisión disminuirá gradualmente y el tiempo de cálculo aumentará cuadráticamente en el tamaño de la capa; es probable que se rinda debido a la lentitud excesiva durante mucho tiempo. antes de que la precisión caiga demasiado. El autocalificador completo para todas las partes del proyecto tarda de 2 a 12 minutos en ejecutarse con las soluciones del personal; si su código está tardando mucho más, debe comprobar su eficacia.
- Si su modelo devuelve Infinity o NaN, su tasa de aprendizaje probablemente sea demasiado alta para su arquitectura actual.
- Valores recomendados para sus hiperparámetros:
    - Tamaños de capa oculta: entre 10 y 400
    - Tamaño del lote: entre 1 y el tamaño del conjunto de datos. Para Q2 y Q3, requerimos que el tamaño total del conjunto de datos sea divisible de manera uniforme por el tamaño del lote.
    - Tasa de aprendizaje: entre 0,001 y 1,0
    - Número de capas ocultas: entre 1 y 3

Código proporcionado (Parte II)

Aquí hay una lista completa de nodos disponibles en `nn.py`. Los utilizará en las partes restantes de la tarea:

- `nn.Constant` representa una matriz (matriz 2D) de números de punto flotante. Por lo general, se usa para representar características de entrada o salidas / etiquetas de destino. Otras funciones de la API le proporcionarán instancias de este tipo; no necesitarás construirlos directamente
- `nn.Parameter`representa un parámetro entrenable de un perceptrón o red neuronal. Todos los parámetros deben ser bidimensionales.
    - Uso: `nn.Parameter(n, m)`construye un parámetro con forma$norte \times metro$
- `nn.Add` agrega matrices por elementos
    - Uso: `nn.Add(x, y)`acepta dos nodos de forma `batch_size` x y `num_features` construye un nodo que también tiene forma `batch_size` x `num_features`

- `nn.AddBias` agrega un vector de sesgo a cada vector de característica
    - Uso: `nn.AddBias(features, bias)` acepta `features`de forma `batch_size` x `num_features` y `bias`de la forma de `1`x `num_features`, y construye un nodo que tiene la forma `batch_size` x `num_features`.

- `nn.Linear` aplica una transformación lineal (multiplicación de matrices) a la entrada
    - Uso: `nn.Linear(features, weights)` acepta `features` de forma `batch_size` x `num_input_features` y `weights` de la forma de `num_input_features` x `num_output_features`, y construye un nodo que tiene la forma `batch_size` x `num_output_features`.

- `nn.ReLU` aplica la no linealidad de la unidad lineal rectificada por elementos $\text{relu}(X)=\max(X,0)$. Esta no linealidad reemplaza todas las entradas negativas en su entrada con ceros.
    - Uso : `nn.ReLU(features)`, que devuelve un nodo con la misma forma que `input`.
- `nn.SquareLoss` calcula una pérdida cuadrada por lotes (error al cuadrado para que de igual si el error es positivo o negativo), que se utiliza para problemas de regresión
    - Uso:`nn.SquareLoss(a, b)` donde `a` y `b` ambos tienen forma `batch_size` x `num_outputs`.

- `nn.SoftmaxLoss` calcula una pérdida softmax por lotes, utilizada para problemas de clasificación
    - Uso:`nn.SoftmaxLoss(logits, labels)`donde `logits` y `labels`ambos tienen forma `batch_size` x `num_classes`. El término "logits" se refiere a las puntuaciones producidas por un modelo, donde cada entrada puede ser un número real arbitrario. Sin embargo, las etiquetas deben ser no negativas y cada fila debe sumar 1. ¡Asegúrate de no cambiar el orden de los argumentos!

- ***No emplees `nn.DotProduct` para ningún modelo, en su lugar emplearás nn.Linear para el resto de los ejercicios, que realizará la multiplicación de matrices.***

Los siguientes métodos están disponibles en `nn.py`:

- `nn.gradients` calcula los gradientes de una pérdida con respecto a los parámetros proporcionados.
    - Uso: `nn.gradients(loss, [parameter_1, parameter_2, ..., parameter_n])` devolverá una lista `[gradient_1, gradient_2, ..., gradient_n]`, donde cada elemento es un que `nn.Constant`contiene el gradiente de la pérdida con respecto a un parámetro.

- `nn.as_scalar` puede extraer un número float de Python de un nodo de error (loss). En muchos casos se suele emplear para determinar cuándo dejar de entrenar, cuando el error sea menor que un determinado valor de corte.
    - Uso : `nn.as_scalar(node)`, donde `node` es un nodo de error que tiene forma1×1.

Los conjuntos de datos proporcionados también tienen dos métodos adicionales:

- `dataset.iterate_forever(batch_size)` produce una secuencia infinita de lotes de ejemplos.

- `dataset.get_validation_accuracy()`devuelve la precisión de su modelo en el conjunto de validación. Esto puede resultar útil para determinar cuándo dejar de entrenar. Se recomienda emplear este método.

Ejemplo: regresión lineal

Como ejemplo de cómo funciona el marco de la red neuronal, ajustemos una línea a un conjunto de puntos de datos. Comenzaremos empleando un batch de tamaño cuatro para el entrenamiento construidos usando la función $y=7X_0+8X_1+3$. En forma de batches (lotes), nuestros datos son:

$$X=\begin{vmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{vmatrix} \qquad Y=\begin{vmatrix} 3 \\ 11 \\ 10 \\ 18 \end{vmatrix}$$

Supongamos que los datos se nos proporcionan en forma de `nn.Constant` nodos:

```
>>> x

<Constant shape = 4x2 a 0x10a30fe80>

>>> y

<Constant shape = 4x1 a 0x10a30fef0>
```

Construyamos y entrenemos un modelo de la forma $S(X)=x_0 \cdot w_0+x_1 \cdot w_1+b$. Si se hace correctamente, deberíamos poder aprender $w_0=7$, $w_1=8$ y $b=3$.

Primero, creamos nuestros parámetros entrenables. En forma de matriz, estos son:

$$W=\begin{vmatrix} w_0 \\ w_1 \end{vmatrix} \qquad B=\begin{vmatrix} b \end{vmatrix}$$

Que corresponde al siguiente código:

```
W = nn.Parameter (2, 1)

B = nn.Parameter (1, 1)
```

Imprimirlos da:

```
>>> W

<Parameter shape = 2x1 en 0x112b8b208>

>>> B

<Parameter shape = 1x1 en 0x112b8beb8>
```

A continuación, calculamos las predicciones de nuestro modelo para y:

```
xW = nn.Linear (x, W)

predicted_y = nn.AddBias (xW, B)
```

Nuestro objetivo es que los valores de y predichos coincidan con los datos proporcionados por el humano (el gold standard o partrón de oro). En la regresión lineal hacemos esto minimizando la pérdida al cuadrado (error al cuadrado):

$$Error = \frac{1}{2N} \sum_{(x,y)} (y - f(x))^2$$

Construimos un nodo de errores:

```
error = nn.SquareLoss(predicted_y, y)
```

En nuestro marco, proporcionamos un método que devolverá los gradientes de la pérdida con respecto a los parámetros:

```
grad_wrt_m, grad_wrt_b = nn.gradients (error, [W, B])
```

La impresión de los nodos utilizados da:

```
>>> xW

<Linear shape = 4x1 en 0x11a869588>

>>>

predicted_y <Forma AddBias = 4x1 en 0x11c23aa90>

>>> error

<SquareLoss shape= () en 0x11c23a240>

>>> grad_wrt_W
```

```
<Constant shape = 2x1 en 0x11a8cb160

>> >> grad_wrt_B

<Constant shape = 1x1 en 0x11a8cb588>
```

Luego podemos usar el método `update` para actualizar nuestros parámetros. Aquí hay una actualización para `W`, asumiendo que ya hemos inicializado una variable `multiplier` basada en una tasa de aprendizaje adecuada de nuestra elección:

```
m.update (grad_wrt_W, multiplicador)
```

Si también incluimos una actualización `B` y agregamos un bucle para realizar actualizaciones de gradiente repetidamente, tendremos el procedimiento de entrenamiento completo para la regresión lineal.

Pregunta 1 (6 puntos): Regresión no lineal

Para esta pregunta, entrenarás una red neuronal para aproximar la función del sen(x) determinado sobre $[-2\pi,2\pi]$.

Deberás completar la implementación de la clase `RegressionModel` en `models.py`. Para este problema, una arquitectura relativamente simple debería ser suficiente (consulta Consejos de redes neuronales para obtener sugerencias de arquitecturas). Emplea `nn.SquareLoss` como tu error cuadrático (SquareLoss).

Tus tareas son:

- Implementar `RegressionModel.__init__` con cualquier inicialización necesaria
- Implementar `RegressionModel.run` para devolver un nodo de dimensión `batch_size x 1` que represente la predicción de su modelo.
- Implementar `RegressionModel.get_loss` para devolver una pérdida para determinadas entradas y salidas objetivo.
- Implementar `RegressionModel.train`, que debería entrenar su modelo usando actualizaciones basadas en gradientes.

Solo hay una única división del conjunto de datos para esta tarea, es decir, solo hay datos de entrenamiento y no hay datos de validación o conjunto de prueba. Tu implementación recibirá todos los puntos si obtiene un error de 0.02 o menor, como promedio en todos los ejemplos del conjunto de datos. Puede usar la pérdida de entrenamiento para determinar cuándo detener el entrenamiento (emplea `nn.as_scalar` para convertir un nodo error (lo que te devuelva SquareLoss)

en un número de Python). Ten en cuenta que el modelo debería tardar unos minutos en entrenarse.

```
python autograder.py -q q2
```

Pregunta 2 (6 puntos): Clasificación de dígitos

Para esta pregunta, entrenarás una red para clasificar los dígitos escritos a mano del conjunto de datos MNIST (como hicistes en el anterior laboratorio, pero esta vez para actualizar los pesos emplearás en descenso del gradiente).

Cada dígito es de tamaño 28×28 píxeles, cuyos valores se almacenan en un vector de dimensión $1 \text{x} 784$ compuesto por floats. Cada salida que proporcionamos es un vector de 10 dimensiones que tiene ceros en todas las posiciones, excepto un uno en la posición correspondiente a la clase correcta del dígito.

Completa la implementación de la `DigitClassificationModel`clase en `models.py`. El valor de retorno de `DigitClassificationModel.run()`debe ser un nodo `batch_size` x `10` que contenga puntuaciones, donde las puntuaciones más altas indican una mayor probabilidad de que un dígito pertenezca a una clase en particular (0-9). Deberías usar `nn.SoftmaxLoss` como tu error. No emplees una activación de ReLU después de la última capa de la red.

Para esta pregunta, además de los datos de entrenamiento, también hay datos de validación y un conjunto de pruebas. Puedes utilizar `dataset.get_validation_accuracy()` para calcular la precisión de validación de su modelo, lo que puede resultar útil a la hora de decidir si detener el entrenamiento. El autograder utilizará el conjunto de test.

Para recibir puntos por esta pregunta, tu modelo debe alcanzar una precisión de al menos el 97% en el equipo de prueba. Como referencia, nuestra implementación logra consistentemente una precisión del 98% en los datos de validación después de aproximadamente 5 epochs. Ten en cuenta que la prueba le califica según la **precisión de la prueba**, mientras que solo tiene acceso a la **precisión de la validación**, por lo que si su precisión de validación alcanza el umbral del 97%, aún puedes noobtener la máxima puntuación si la precisión sobre el test no alcanza el umbral. Por lo tanto, puede ser útil establecer un umbral de detención ligeramente más alto en la precisión de la validación, como 97,5% o 98%.

Para probar tu implementación, ejecuta el autograder:

```
python autograder.py -q q3
```

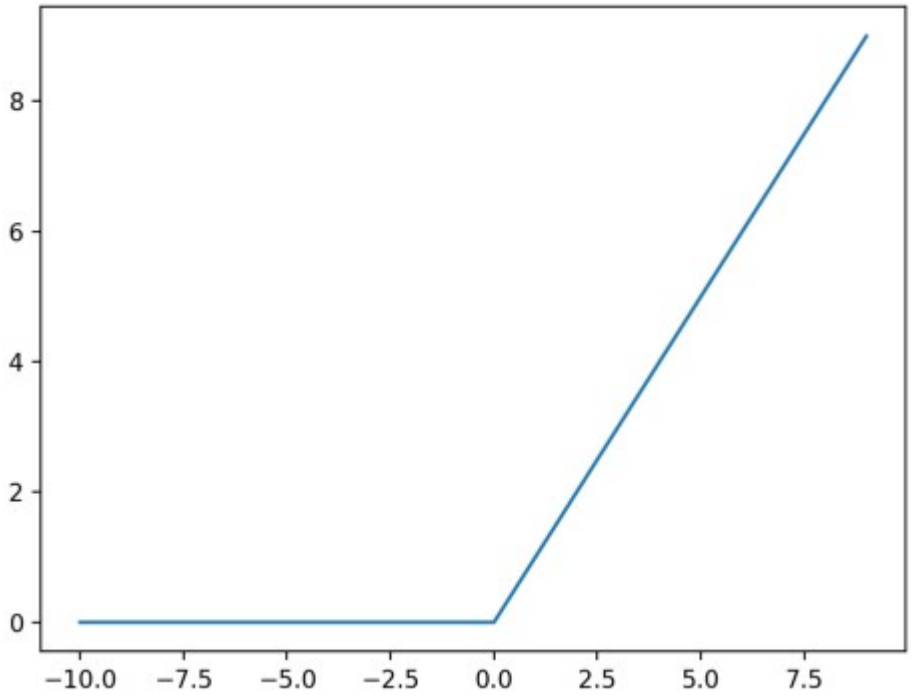
Breve explicación sobre las ReLUs

Rectified Linear Unit es una función de activación que se emplea habitualmente en las arquitecturas de redes neuronales modernas para aportar no-linearidad. Se define como max (0, x).

Inicialmente parece difícil imaginar que aporte no linealidad, porque su forma es bastante lineal… pero son muy poderosas y empleando varias se puede aproximar prácticamente cualquier función con ellas.

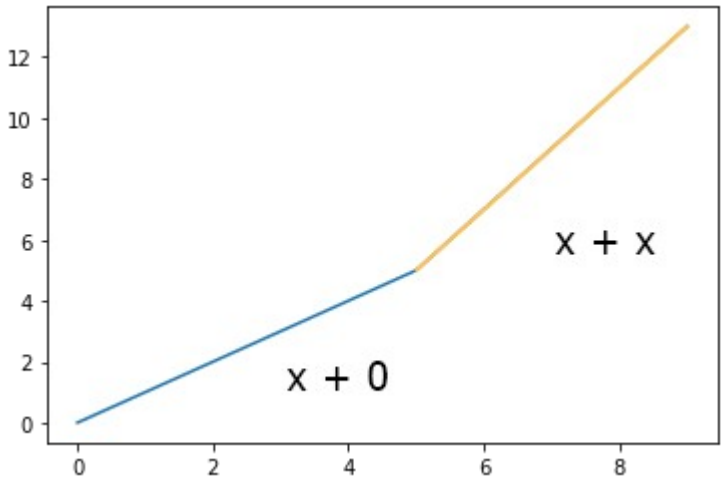
Primero, hay una propiedad importante de ReLU:

$$ReLU(x - c) = 0, \text{ for } x \leq c$$



Así, teniendo f(x) = x

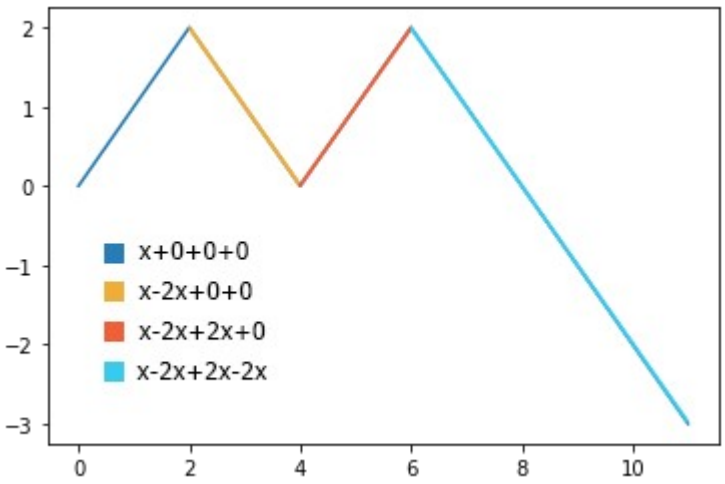
$$f(x) + ReLU(x - c) = f(x), \text{ for } x \leq c$$



Suponiendo **c** = **5** obtendríamos la siguiente representación no lineal

Incluso de podría generar un angulo mayor multiplicando el término **ReLU**(**x** - **c**) por una constante, mientras los outputs para valores de **x** ≤ **c** no se verán afectados. Así añadiendo muchos términos **ReLU**(**x** - **c**) y contante * **ReLU**(**x** - **c**) podríamos incluso conseguir la siguiente representación.

Ejemplo con mas términos :



ReLU(x) + (-2)*ReLU(x — 2) + 2*ReLU(x — 4) + (-2)*ReLU(x — 6)

La formula general para aproximar una función no lineal sería para **0** ≤ **x** ≤ **n**:

$$f(x) \approx b + \sum_{k=0}^{n} a_k * ReLU(x - k)$$

En teoría, la precisión de la aproximación puede llegar a ser perfecta añadiendo una ReLu por cada posible valor de x.

Conclusion

> ReLU **es una función no lineal** que nos permite **aproximar funciones no lineales** cuando la añadimos a cualquier función aunque esta sea lineal**.**