Tarea_2_CC6204_2020_[PUBLICADA]

October 2, 2020

1 Tarea 2: Backpropagation, descenso de gradiente y entrenamiento CC6204 Deep Learning, Universidad de Chile

Fecha de entrega: 16 de octubre de 2020 (Hoja de respuestas)

En esta tarea programarás backpropagation para el caso específico de las redes que construiste en la Tarea 1 (si tuviste problemas resolviendo la Tarea 1 puedes usar esta solución tipo que preparó el cuerpo docente). Además comenzarás a entrenar la red con descenso de gradiente, que también tendrás que programar.

Te recomendamos que repases la materia de las clases de backpropagation. En particular para resolver esta tarea te puedes apoyar en el siguiente material: * Clase 04-2020: Descenso de Gradiente para encontrar los parámetros de una red. * Clase 05-2020: Introducción a Backpropagation * Clase 06-2020: Continuación Backpropagation

Algunos videos de versiones pasadas del curso que te pueden servir también de apoyo son los siguientes (los actualizaremos con la versión 2020 cuando los tengamos disponibles): * Entropia Cruzada, funcion de pérdida y tensores (2018) * Derivando tensores y backpropagation a mano (2018)

IMPORTANTE: A menos que se exprese lo contrario, sólo podrás utilizar las clases y funciones en el módulo `torch`.

(por Jorge Pérez, https://github.com/jorgeperezrojas, [@perez](https://twitter.com/perez))

[ ]:
```
# Este notebook está pensado para correr en CoLaboratory.
# Lo único imprescindible por importar es torch
import torch

# Posiblemenete quieras instalar e importar ipdb para debuggear.
# Si es así, descomenta lo siguiente:
# !pip install -q ipdb
# import ipdb
```

2 Parte 1: Preliminares: funciones de activación y función de error

2.1 1a) Derivando las funciones de activación

En esta parte debes calcular a mano las derivadas de las funciones `relu`, `swish` y `celu` que usamos en la Tarea 1. Recuerda que `swish` y `celu` tienen ambas parámetros adicionales así que debes calcular las derivadas (parciales) con respecto a ellos también. Intenta expresar las derivadas en términos de aplicaciones de la misma función (o sub expresiones de esta). Por ejemplo, si derivas

la función sigmoid(x) (hazlo! es un buen ejercicio) encontrarás que su derivada se puede expresar como:

$$\frac{\partial \,\text{sigmoid}(x)}{\partial x} \;=\; \text{sigmoid}(x)\big(1-\text{sigmoid}(x)\big) \tag{1}$$

Usa la Hoja de respuesta para incluir tus expresiones.

2.2 1b) Entropía Cruzada

Comenzaremos haciendo una función para computar la pérdida de nuestra red. Recuerda que para dos distribuciones de probabilidad discreta $p(x)$ y $q(x)$ la entropía cruzada (cross entropy) entre p y q está dada por

$$CE(p,q) = \sum_x p(x)\log\left(\frac{1}{q(x)}\right) = -\sum_x p(x)\log q(x) \tag{2}$$

donde x varía sobre todos los posibles valores para los cuales la distribución está definida.

En esta parte debes programar la función `CELoss` que recibe tensores Q_{ij} y P_{ij} (de las mismas dimensiones) y calcula el promedio de las entropías cruzadas de las distribuciones p_i y q_i de la siguiente forma

$$CELoss(Q,P) = \frac{1}{N}\sum_i CE(p_i,q_i) \tag{3}$$

donde $p_i(x) = P_{ix}$, $q_i(x) = Q_{ix}$ y N es el tamaño de la primera dimension de los tensores (dimension `0`). Nota que el resultado es un escalar. Nota también el orden de Q y P en $CELoss(Q,P)$. Esto no es un error, sino es la forma standard de usar la entropía cruzada como una función de error, en donde el primer argumento (Q) es la aproximación (distribución de probabilidad erronea) y el segundo argumento (P) es el valor al que nos queremos acercar (distribución de probabilidad real, o más percisamente en nuestro caso, distribución de probabilidad empírica).

En nuestra implementación debemos evitar cualquier ocurrencia de `NaN` debido a valores en nuestras distribuciones de probabilidad excesivamente pequeños al calcular `torch.log`. Estos valores deberían devolver números negativos demasiado pequeños para procesar y dan como resultado `NaN`. El valor épsilon limitará el valor mínimo del valor original cuando `estable=True`.

[ ]:
```
# Tu código acá
def CELoss(Q, P, estable=True, epsilon=1e-8)
  pass
```

2.3 1c) Opcional: Entropía Cruzada Categórica

La entropía cruzada es un concepto muy general pero en el caso de entrenamiento la usaremos como una función de error entre una distribución de probabilidad general Q sobre clases (la distribución que genera nuestra red) y una distribución P que siempre tiene toda la probabilidad asignada a una única clase (la etiqueta de entrenamiento). Esta observación permite calcular una función de error menos general pero más eficiente para nuestro caso. En esta parte programarás una versión alternativa de `CELoss` que llamaremos `CategoricalCELoss` y que recibirá dos parámetros `(Q,Target)` donde `Q` es como en el caso anterior y `Target` es un tensor con los índices de las etiquetas correctas. Tu implementación de esta parte debiera entregar los mismos

resultados que la anterior si es que cambias `Target` por una matriz que tiene `1`s exactamente en los índices de las clases y `0`s en las otras posiciones.

[ ]:
```
# Tu código acá
def CategoricalCELoss(Q, Target, estable=True, epsilon=1e-8)
  pass
```

Esta parte (opcional), te servirá para entender y practicar dos formas de calcular la entropía cruzada lo que es importante pues distintas librerías de Deep Learning usan distintas estrategias para este cálculo en la práctica.

Hay otras observaciones de cómo las librerías implementan esta función de error en la práctica y es importante entender cómo se deben utilizar en cada caso por eso te incentivamos a que completes esta parte opcional y que también investigues cómo lo hacen algunas librerías en la práctica. Por ejemplo, otro cambio que hace `pytorch` para hacer el proceso mas eficiente es combinar la capa final de clasificación de la red (la de softmax) con el cálculo que describimos arriba, en una única función torch.nn.functional.cross_entropy y lo integran también de esta forma dentro de la clase torch.nn.CrossEntropyLoss

3 Parte 2: Más derivadas y back propagation

En esta parte comenzaremos a usar el algoritmo de back propagation para poder actualizar los parámetros de nuestra red neuronal (la que empezaste a construir en la Tarea 1). Nuestra red está dada por las ecuaciones

$$\begin{aligned} h^{(\ell)} &= f^{(\ell)}(h^{(\ell-1)}W^{(\ell)} + b^{(\ell)}) \\ \hat{y} &= \text{softmax}(h^{(L)}U + c). \end{aligned}$$

Recuerda que en estas ecuaciones consideramos que el $h^{(0)}$ es el tensor de input, digamos x, y típicamente llamamos a $\hat{y}$ como $\hat{y} = \text{forward}(x)$.

Para optimizar los parámetros de nuestra red usaremos la función de pérdida/error de entropía cruzada (ver la parte anterior). Dado un conjunto (mini batch) de ejemplos $\{(x_1, y_1), \ldots, (x_B, y_B)\}$, llamemos x al tensor que contiene a todos los x_i's *apilados* en su dimensión 0. Nota que x tendrá una dimensión más que los x_i's. Similarmente llamemos y al tensor que contiene a todos los y_i's. La función de pérdida de la red se puede entonces escribir como

$$\mathcal{L} = CELoss(\hat{y}, y) \tag{4}$$

donde $\hat{y} = \text{forward}(x)$ y $CELoss(\hat{y}, y)$ es la función de entropía cruzada aplicada a $\hat{y}$ e y. En esta parte computaremos las derivadas parciales

$$\frac{\partial \mathcal{L}}{\partial \theta} \tag{5}$$

para cada parámetro θ de nuestra red.

3.1 2a) Derivando la última capa

Recuerda que $\hat{y} = \text{softmax}(h^{(L)}U + c)$. Nuestro objetivo en esta parte es calcular la derivada de $\mathcal{L}$ con respcto a U, $h^{(L)}$ y c. Para esto llamemos primero

$$u^{(L+1)} = h^{(L)}U + c. \tag{6}$$

Nota que con esto, nuestra predicción es simplemente $\hat{y} = \text{softmax}(u^{(L+1)})$. Calcula la derivada (el *gradiente*) de $\mathcal{L}$ respecto a $u^{(L+1)}$, y escribe un trozo de código usando las funcionalidades de `torch` que calcule el valor y lo almacene en una variable `dL_duLm1`, suponiendo que cuentas con los tensores `y` e `y_pred` (que representa a $\hat{y}$).

Puedes escribir tu cálculo acá (usa la Hoja de respuesta para la entrega)

$$\frac{\partial \mathcal{L}}{\partial u^{(L+1)}} = \tag{7}$$

[ ]:
```
# Para ir chequeando que al menos las dimensiones de los tensores son
# consistentes usaremos las varibles *dummy* a continuación.
B, C = 5, 10
y = torch.ones(B,C)
y_pred = torch.ones(B,C)
```

[ ]:
```
# Acá tu trozo de código.
# Primero agregamos algunas variables dummy para chequear
# que al menos las dimensiones están correctas
dimL = 40
hL = torch.ones(B,dimL)
U = torch.ones(dimL,C)
c = torch.ones(C)
uLm1 = hL @ U + c

# Ahora tu fórmula para el gradiente
dL_duLm1 = None

# El gradiente debe coincidir en dimensiones con la variable
assert dL_duLm1.size() == uLm1.size()
```

3.2 2b) Derivando la última capa (continuación)

Usa la derivada de $\mathcal{L}$ con respecto a $u^{(L+1)}$ y la regla de la cadena para encontrar las derivadas (*gradientes*) de $\mathcal{L}$ con respecto a U, c y $h^{(L)}$. Recuerda tener cuidado con los índices de los tensores, chequear que las dimensiones sean las correctas y cuando sea necesario usa la notación de Einstein para simplificar tu vida. Escribe también un trozo de código para calcular estas derivadas y almacenarlas en `dL_dU`, `dL_dc` y `dL_dhL`.

Puedes escribir tu cálculo acá (usa la Hoja de respuesta para la entrega)

$$\frac{\partial \mathcal{L}}{\partial U} = \dots \tag{8}$$

$$\frac{\partial \mathcal{L}}{\partial c} = \dots \tag{9}$$

$$\frac{\partial \mathcal{L}}{\partial h^{(L)}} = \dots \tag{10}$$

[ ]:
```
# Acá puedes probar tus cálculos usando código.
dL_dU = None
dL_dc = None
dL_dhL = None

# El gradiente debe coincidir en dimensiones con las variables.
assert dL_dU.size() == U.size()
assert dL_dc.size() == c.size()
assert dL_dhL.size() == hL.size()
```

3.3 2c) Derivando desde las capas escondidas

Ahora derivaremos las capas escondidas en general para todas las funciones de activación que consideramos en esta tarea. **Importante: esta parte es larga no la empieces a hacer tarde**. Consideremos la capa k, en este caso tenemos

$$h^{(k)} = f(h^{(k-1)}W^{(k)} + b^{(k)}) \tag{11}$$

done f es una de las funciones de activación sig, tanh, relu, celu, swish. Lo que queremos es computar las derivadas parciarles de $\mathcal{L}$ con respecto a $W^{(k)}$, $b^{(k)}$ y $h^{(k-1)}$. Para esto consideremos

$$u^{(k)} = h^{(k-1)}W^{(k)} + b^{(k)}. \tag{12}$$

Supondremos que ya tenemos computado (antes) el gradiente de $\mathcal{L}$ con respecto a $h^{(k)}$ ($\partial\mathcal{L}/\partial\langle^{(||)}$). Para cada función de activación de entre relu, celu, swish, calcula primero

$$\frac{\partial\mathcal{L}}{\partial u^{(k)}} \tag{13}$$

usando $\partial\mathcal{L}/\partial\langle^{(||)}$ y luego usa $\partial\mathcal{L}/\partial\sqcap^{(||)}$ y la regla de la cadena para calcular

$$\frac{\partial\mathcal{L}}{\partial W^{(k)}}, \frac{\partial\mathcal{L}}{\partial b^{(k)}}, \frac{\partial\mathcal{L}}{\partial h^{(k-1)}}. \tag{14}$$

Crea trozos de código para cada uno de los cálculos de los gradientes. Este código lo usaremos luego en la función `backward` de tu red.

Usa la Hoja de respuesta para tus cálculos

[ ]:
```
# Acá puedes probar tus cálculos usando código.
# Primero agregamos algunas variables dummy para chequear
# que al menos las dimensiones están correctas.
dimk = 20
dimkm1 = 30
hk = torch.ones(B,dimk)
Wk = torch.ones(dimk,dimkm1)
bk = torch.ones(dimkm1)
uk = hk @ Wk + bk
dL_dhkm1 = torch.ones(B,dimkm1)

# Ahora tu fórmula para el gradiente.
# Esto puedes repetirlo con tus expresiones para relu, celu, y swish.
```

```
dL_duk = None
dL_dWk = None
dL_dbk = None
dL_dhk = None

# El gradiente debe coincidir en dimensiones con las variables.
assert dL_dWk.size() == Wk.size()
assert dL_dbk.size() == bk.size()
assert dL_dhk.size() == hk.size()
```

4 Parte 3: Backpropagation en nuestra red

En esta parte programaremos todos nuestros cálculos anteriores dentro del método `backward` de nuestra red.

4.1 3a) Método `backward`

Programa un método `backward` dentro de la clase FFNN que hiciste para la Tarea 1. El método debiera recibir como entrada tres tensores `x`, `y`, `y_pred`, y debiera computar todos los gradientes para cada uno de los parámetros de la red (con todas las suposiciones que hicimos en la Parte 3, incluyendo el uso de entropía cruzada como función de pérdida). Recuerda computar los gradientes también para capas escondidas con activaciones sig y tanh.

Podemos aprovecharnos de las funcionalidades de la clase `torch.nn.Parameter` para almacenar los resultados de cada gradiente. De hecho, cada objeto de la clase `torch.nn.Parameter` tiene un atributo `grad` que está pensado específicamente para almacenar los valores computados a medida que se hace backpropagation. Utiliza este atributo para almacenar el gradiente del parámetro correspondiente.

[ ]:
```
# Tu código debiera continuar aquí

class FFNN(torch.nn.Module):
  def __init__(self, F, l_h, l_a, C):
    pass

  def forward(self, x):
    # ya lo creaste en la parte anterior
    pass

  def backward(self, x, y, y_pred):
    # computar acá todos los gradientes
    pass
```

4.2 3b) Opcional: Incluyendo los parámetros de `celu` y `swish`

Si lo deseas, puedes agregar los parámetros de todas las activaciones `celu` y `swish` de tu red como parámetros. Para esto tendrás que agregar parámetros en el inicializador de la clase y computar las derivadas correspondientes en la función `backward`.

4.3 3c) Opcional: Chequeo de gradiente

Determinar si calculaste bien o no las derivadas puede ser un verdadero dolor de cabeza. Si no confías plenamente en tus capacidades para computar derivadas a mano (nadie debería), puedes usar una técnica muy útil para determinar si cometiste un error. La técnica se llama **chequeo de gradiente** y consiste en computar el gradiente de forma numérica y compararlo con la evaluación del gradiente computado de forma manual. Es una de las técnicas más útiles para *debuggear* redes neuronales. En esta parte porgrmarás el chequeo de gradiente. Si bien esta parte es opcional, puede que te ahorre demasiados problemas tenerla programada para ir comprobando que no cometiste errores.

La idea general de cómo implementar chequeo de gradiente es la siguiente: - Supongamos que todos los posilbes parámetros de tu red son $\Theta = [\theta^{(1)}, \theta^{(2)}, \ldots, \theta^{(M)}]$, es decir pusimos los parámetros como un vector gigante. Entonces podemos pensar en la función de error como una función $\mathcal{L}(\Theta) = \mathcal{L}([\theta^{(1)}, \theta^{(2)}, \ldots, \theta^{(M)}])$. - Elegimos al azar valores para cada parámetro, digamos $\Theta_{ch} = [\theta_{ch}^{(1)}, \theta_{ch}^{(2)}, \ldots, \theta_{ch}^{(M)}]$. - Elegimos un valor fijo muy pequeño, digamos ε. Típicamente ε puede ser cercano a 10^{-7} o 10^{-8}. - Elegimos un índice i que presentará el parámetro para el que queremos chequear el gradiente. - Computamos los siguientes valores

$$\mathcal{L}_{+\varepsilon}^{(i)} = \mathcal{L}([\theta_{ch}^{(1)}, \theta_{ch}^{(2)}, \ldots, \theta_{ch}^{(i)} + \varepsilon, \ldots, \theta_{ch}^{(M)}]) \tag{15}$$

$$\mathcal{L}_{-\varepsilon}^{(i)} = \mathcal{L}([\theta_{ch}^{(1)}, \theta_{ch}^{(2)}, \ldots, \theta_{ch}^{(i)} - \varepsilon, \ldots, \theta_{ch}^{(M)}]) \tag{16}$$

Nota para calcular estos valores necesitamos hacer dos pasadas hacia adelante de la red. - Computamos

$$\text{AppGrad}(\theta_{ch}^{(i)}) = \frac{\mathcal{L}_{+\varepsilon}^{(i)} - \mathcal{L}_{-\varepsilon}^{(i)}}{2\varepsilon} \tag{17}$$

- Finalmente comparamos este valor con lo que nuestra función de backpropagation entrega para el mismo parámetro cuando hacemos la pasada hacia atrás (backward) evaluada para los mismos valores específicos, es decir comparamos $\text{AppGrad}(\theta_{ch}^{(i)})$ con

$$\left.\frac{\partial \mathcal{L}}{\partial \theta^{(i)}}\right|_{\Theta_{ch}} \tag{18}$$

- Repetimos el proceso para todos los parámetros de la red.

Si nuestras derivadas están bien calculadas, esperaríamos que la diferencia entre $\text{AppGrad}(\theta^{(i)})$ y el gradiente que calcula nuestra función `backward` sea muy (muy) pequeña, típicamente cercana a ε. Si es mucho más grande que ε tenemos que preocuparnos porque muy posiblemente tendremos un bug.

Una forma alternativa de hacer el chequeo del gradiente es meter todos los $\partial \mathcal{L}/\partial \theta^{(i)}|_{\Theta_{ch}}$ en un vector, digamos Grad, hacer lo mismo con todos los $\text{AppGrad}(\theta^{(i)})$ y meterlos todos en un vector AppGrad y después chequear que los dos vectores sean relativamente cercanos haciendo

$$\frac{\|\text{Grad} - \text{AppGrad}\|^2}{\|\text{Grad}\|^2 + \|\text{AppGrad}\|^2} \tag{19}$$

Acá también esperamos que el valor resultante sea muy cercano a ε.

[ ]:
```
# Tu código de chequeo de gradiente acá
```

5 Parte 4: Descenso de gradiente y entrenamiento

En esta parte programaras el algoritmo de descenso de gradiente más común y entrenarás finalmente tu red para que encuentre parámetros que le permitan clasificar datos aleatorios (mas abajo podrás hacerlo opcionalmente también para MNIST).

5.1 4a) Descenso de gradiente (estocástico)

Construye una clase `SGD` que implemente el algoritmo de descenso de gradiente. El inicializador de la clase debe recibir al menos dos argumentos: un "iterable" de parámetros a los cuales aplicarles el descenso de gradiente, y un valor real `lr` correspondiente a la taza de aprendizaje para el descenso de gradiente. El único método que debes implementar es el método `step` que debe actualizar todos los parámetros. En este caso asumiremos que a cada parámetro ya se le han computado los gradientes (todos almacenados en el atributo `.grad` de cada parámetro). El uso de esta clase debiera ser como sigue:

```
# datos = iterador sobre pares de tensores x, y
# red = objeto FFNN previamente inicializado

optimizador = SGD(red.parameters(), 0.001)
for x,y in datos:
  y_pred = red.forward(x)
  l = CELoss(y_pred, y)
  red.backward(x, y, y_pred)
  optimizador.step()
```

[ ]:
```
# Tu código debiera comenzar así

class SGD():
  def __init__(self, parameters, lr):
    # lo que sea necesario inicializar
    pass

  def step(self):
    # actualiza acá los parámetros a partir del gradiente de cada uno
    pass
```

5.2 4b) Datos para carga

En esta parte crearás un conjunto de datos de prueba aleatorios para probar con tu red. La idea de partir con datos al azar es para que te puedas concentrar en encontrar posibles bugs en tu implementación antes de probar tu red con cosas más complicadas.

Para esta parte debes crear una clase `RandomDataset` como subclase de `Dataset` (que se encuentra en el módulo `torch.utils.data`). Tu clase debe recibir en su inicializador la cantidad de ejemplos a crear, la cantidad de características de cada ejemplo, y la cantidad de clases en la función objetivo. Debes definir la función `__len__` que retorna el largo del dataset y la función `__getitem__` que permite acceder a un item específico de los datos. Cada elemento entregado por `__getitem__` debe ser un par (x, y) con un único ejemplo, donde x es un tensor que representa a

los datos de entrada (características) e y representa al valor esperado de la clasificación para esa entrada.

Lo positivo de definir un conjunto de datos como `Dataset` es que luego puedes usar un `DataLoader` para iterar por paquetes sobre el dataset y entregarlos a una red (tal como lo hiciste en la Tarea 1 para MNIST). El siguiente trozo de código de ejemplo muestra cómo debieras usar tu clase en conjunto con un `DataLoader`.

```
dataset = RandomDataset(1000,200,10)
data = DataLoader(dataset, batch_size=4)
for x,y in data:
  # x,y son paquetes de 4 ejemplos del dataset.
```

[ ]:
```
from torch.utils.data import Dataset, DataLoader

# Aquí tu código.
# Tu clase debiera verse así
class RandomDataset(Dataset):
  def __init__(self, N, F, C):
    pass

  def __len__(self):
    pass

  def __getitem__(self, i):
    pass
```

5.3 4c) Entrenando la red con datos al azar

Por fin podrás crear un ciclo de entrenamiento. Para esto crea la función `entrenar_FFNN` que recibe una red, un dataset, un optimizador, la cantidad de épocas por las que se quiere entrenar, el tamaño de los paquetes de ejemplos usados en el entrenamiento, y el dispositivo donde se correrá el loop. Puedes definir todos los argumentos adicionales que quieras para la función.

Dentro de la función debes hacer tantas iteraciones sobre el dataset como la cantidad de épocas indicadas utilizando el optimizador para actualizar los parámetros de la red para cada paquete de ejemplos. Procura que todo el trabajo (forward y backward) se haga en el dispositivo indicado. Al finalizar, la función debe retornar una lista con el valor de la pérdida en cada iteración. Asegúrate de que la función muestre información relevante durante el entrenamiento (piensa en toda la información que te gustaría tener mientras la red entrena y muéstrala en pantalla). Si quieres, puedes agregar un parámetro `verbose` para indicar el nivel de información mostrada durante el entrenamiento.

El uso de la función, y de todo tu código hasta ahora, debiera verse como sigue:

```
F, C = 300, 10
red = FFNN(F, [50,30], [relu,sig], C)
optimizador = SGD(red.parameters(), 0.001)
N = 1000
dataset = RandomDataset(N, F, C)
perdida = entrenar_FFNN(red, dataset, optimizador, epochs=20, batch_size=8, device='cpu')
```

[ ]:
```
# Tu código acá
def entrenar_FFNN(red, dataset, optimizador, epochs=1, batch_size=1,␣
↪device='cuda'):
 pass
```

4d) Graficando la pérdida/error en el tiempo

Usa la librería `matplotlib` para mostrar cómo varía la pérdida a medida que entrena tu red. Muestra al menos tres redes distintas entrenadas con el mismo conjunto de datos y compara la forma de aprendizaje. Intenta mostrar un caso que diferencie a las redes, cambiando su tamaño y también el tamaño de los datos de entrada.

[ ]:
```
# Tu código acá
```

5.4 4e) Opcional: Entrenando tu red con MNIST

Usa tu red para entrenar con los datos de MNIST. Mira cómo usamos este conjunto de datos en la Tarea 1, pero esta vez usa el argumento `train=True` cuando descargues el dataset. Grafica la pérdida para distintas opciones de redes (más o menos capas, más o menos neuronas por capas, distintos tipos de funciones de activación, etc.) y compara un par de opciones diferentes. Trata de explicar las diferencias (o la ausencia de diferencias).

[ ]:
```
# Tu código de carga de datos, creación de la red,
# entrenamiento y reportes acá
```