AI principle HW1 第七組

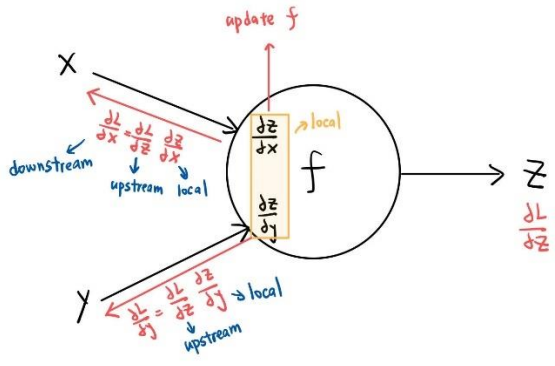
110511068 蔡雅婷 110511164 張語楹 109261007 李紹穎

Code: https://github.com/Benchangatrul284/AI_principle/tree/main/HW1_late

Gradient and Gradient descent (GD)

在實作 SGD 之前，我們要先能夠計算梯度(gradient)。我們將利用 back propagation 來有效率的計算對於每個 input 和 weight 對於 loss function 的微分(也就是梯度)，才能利用 gradient descent 來更新參數，使 loss 降低。

首先，我們先假設有一個 function $f(x,y)=z$，也就是 input 是 x 和 y，output 是 z。我們的目標是找到每個 input 對於 loss 的偏微分，並將它傳到下一層。如下圖所示:



我們的目標是計算$\frac{\partial L}{\partial x}$和$\frac{\partial L}{\partial y}$。

這裡$\frac{\partial L}{\partial z}$為已知，我們稱呼他叫 upstream gradient，我們因為知道$f(x,y)$，所以

我們也能計算$\frac{\partial f(x,y)}{\partial x}$和$\frac{\partial f(x,y)}{\partial y}$，因為這些偏微分僅和該節點有關，因此我們稱呼他為 local gradient, 也是已知。最後，我們根據 chain rule，就能寫成:

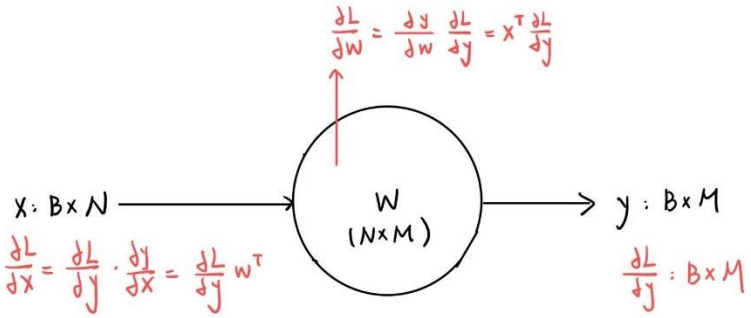
$$\frac{\partial L}{\partial x}=\frac{\partial f(x,y)}{\partial x}\frac{\partial L}{\partial z}=\frac{\partial z}{\partial x}\frac{\partial L}{\partial z}=local\ gradient\times upstream\ gradient$$

再利用不斷地向傳遞直到傳到 input 端，這就是利用 backpropagation 計算梯度的核心。

上述僅提到 scaler 的計算，在神經網路當中，我們操作的都是矩陣，因此我們要將 back propagation 擴展到 matrix operation。

我們假設 input dimension 為 $B\times N$，output dimension 為 $B\times M$。這裡的 f 就會變成$f(\boldsymbol{x})=\boldsymbol{xw}$，因此 W 的維度為$N\times M$。這裡的 B 代表的是 batch size 的大小，N 是指單一個 sample 輸入的維度，M是指單一個 sample 輸出的維度。

如下圖所示:



以矩陣的方法表示:

$$x = \begin{pmatrix} x_{11} & \cdots & x_{1N} \\ \vdots & \ddots & \vdots \\ x_{B1} & \cdots & x_{BN} \end{pmatrix}, w = \begin{pmatrix} w_{11} & \cdots & w_{1M} \\ \vdots & \ddots & \vdots \\ w_{N1} & \cdots & w_{NM} \end{pmatrix}, y = \begin{pmatrix} y_{11} & \cdots & y_{1M} \\ \vdots & \ddots & \vdots \\ y_{B1} & \cdots & y_{BM} \end{pmatrix}$$

我們的目標和前面相同，就是算出 local gradient $\frac{\partial y}{\partial x}$。因為矩陣相當複雜，我們先求$\frac{\partial y}{\partial x_{11}}$。

$$\frac{\partial y}{\partial x_{11}} = \begin{pmatrix} \frac{\partial y_{11}}{\partial x_{11}} & \cdots & \frac{\partial y_{1M}}{\partial x_{11}} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_{B1}}{\partial x_{11}} & \cdots & \frac{\partial y_{BM}}{\partial x_{11}} \end{pmatrix} = \begin{pmatrix} w_{11} & \cdots & w_{1M} \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 0 \end{pmatrix}$$

之後可以用相同的方法來求出$\frac{\partial y}{\partial x_{BN}}$就可以算出 local gradient $\frac{\partial y}{\partial x}$。

從上面可以發現若是直接做矩陣相乘會相當沒有效率(畢竟矩陣相當的 sparse)。但根據上面的式子，我們發現其實 local gradient 只和 weight 有關。也就是說，當計算的 x 在第幾個 column，我們就取 weight matrix 的第幾個 row。

因此我們簡化問題成為 weight matrix 和 upstream gradient 做 inner product。

因此最後的式子可以寫成:

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} w^T$$

用相同的方法，我們也可以求出 weight 對 loss 的偏微分:

$$\frac{\partial y}{\partial w_{11}} = \begin{pmatrix} \frac{\partial y_{11}}{\partial w_{11}} & \cdots & \frac{\partial y_{1M}}{\partial w_{11}} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_{B1}}{\partial w_{11}} & \cdots & \frac{\partial y_{BM}}{\partial w_{11}} \end{pmatrix} = \begin{pmatrix} x_{11} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ x_{B1} & \cdots & 0 \end{pmatrix}$$

當計算的 w 在第幾個 row，我們就取 input matrix 的第幾個 column。

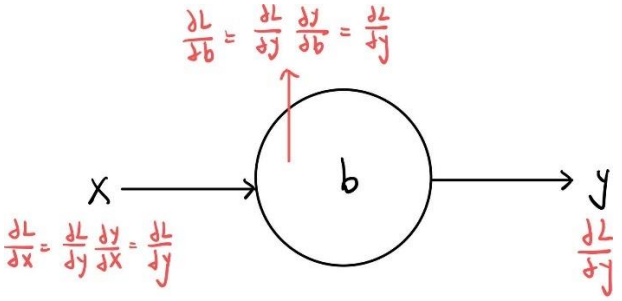
因此我們簡化問題成為 input matrix 和 upstream gradient 做 inner product。

$$\frac{\partial L}{\partial w} = x^T \frac{\partial L}{\partial y}$$

以下是程式碼的實現:

```
def backward_propagation(self, upstream, learning_rate):
        downstream = np.dot(upstream, self.weights.T)
        weights_error = np.dot(self.input.T, upstream)
        return downstream
```

上述是討論對於 weight 矩陣，對於 bias 其實挺簡單的，考慮一個 function $f(x) = x + b$，如下圖:



$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial b} = \frac{\partial L}{\partial y}, \qquad \frac{\partial L}{\partial x} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial x} = \frac{\partial L}{\partial y}$$

一個線性層可以想成先做 weight 矩陣相乘，在加上 bias。因為加上 bias 這個操作不會影響到 downstream gradient。因此在考慮 back propagation 時不需考慮 bias。上述是對一個線性層做 back propagation，對於 activation function，操作會更直觀。因為對於 sigmoid 或 ReLU 等 activation function，沒有可更新的參數，所以我們只要找出 downstream gradient $\frac{\partial L}{\partial x}$就好。

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial x} = \frac{\partial L}{\partial y} f'(x)$$

```
def backward_propagation(self, output_error, learning_rate):
        return self.activation_prime(self.input) * output_error
```

gradient descent 其實就是沿著梯度的方向去更新參數。

```
self.weights -= learning_rate * weights_error
```

其中 learning rate 是一個 hyperparameter，決定一次更新的步長為多少。
有了 linear 和 activation layer 的 forward pass 和 backward pass，我們就能計算所有 layer 對於 loss 的偏微分了。

```
for layer in reversed(self.layers):
    error = layer.backward_propagation(error, learning_rate)
```

可以注意這裡的 reversed 代表著 back propagation。

Stochastic gradient descent (SGD)

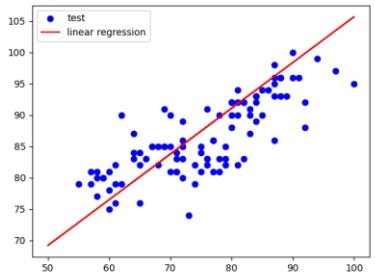
在實際應用層面，一次更新若需先看過所有 training data 計算梯度再更新參數會非常耗費記憶體。一個改善的方法就是每看過一個 training data 就計算梯度並做 gradient descent。因為樣本是隨機取的，因此這個方法叫做 SGD 。此方法的缺點很明顯，就是每次所計算的 loss 只代表模型再一個 training data 的 loss，並不能代表所有 training data 的 loss。因此，折衷的方法就是不用單個 training data，而是用一個 batch，而 batch size 就是一次要送進 model 的 sample 數。

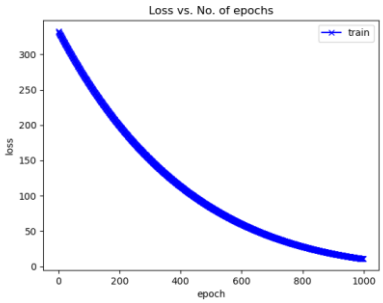
Problem 1: linear regression:

本題因為是 linear regression，因此 loss function 選為 MSE。Model 採用一個矩陣相乘加上 bias。

```
net = Network([Linear(1,1)])
```

此外， learning rate = 0.01 並不會收斂，因此，我將 learning rate 整為 1e-4。



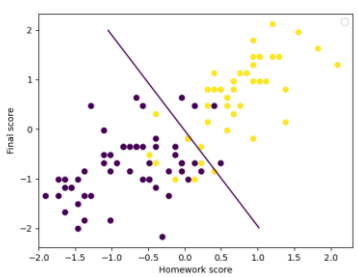


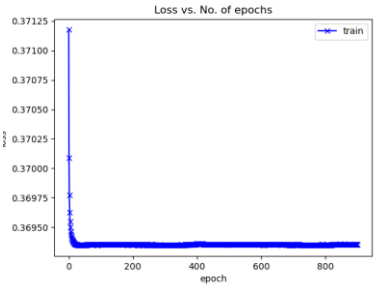
Loss on test set: 96.15

Problem 2: logistic regression

和上一題不同的是 input 改為 2 為所以 linear 層的維度要改，此外，因為問題為一個分類問題，答案僅會介在 0、1 之間，因此輸出前要過 sigmoid。

```
net = Network([Linear(2, 1),ActivationLayer(sigmoid, sigmoid_prime)])
```





loss on test set: 1.25

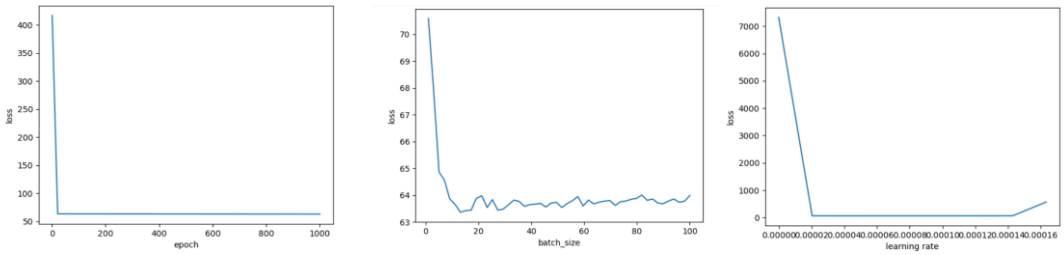
Part 3: hyperparameter selection

最後一題是去跑跑看不同超參數對訓練有甚麼影響。調整 batch size 是裡面較為困難的。在前面實作 SGD 時，因為僅考慮一個 sample，算出 gradient 後可以直接更新。在這題，因為我們考慮的為一個 batch，算出 gradient 後還要先做平均再更新。程式碼如下:

```
def backward_propagation(self, upstream, learning_rate):
        downstream = np.dot(upstream, self.weights.T)
        weights_error = np.dot(self.input.T, upstream)
        upstream = np.mean(upstream,axis=0)
        # breakpoint()
        # update parameters
        self.weights -= learning_rate * weights_error
        self.bias -= learning_rate * upstream
        return downstream
```

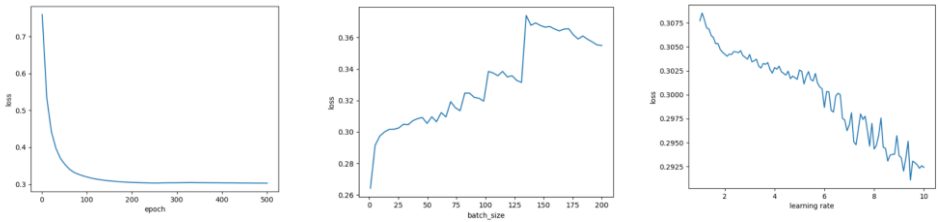
也可以看的出來當 batch size 為 1 就是第一題所做的 SGD 接下來就可以來跑跑看對於不同 hyperparameter 的結果了。

Problem 1:



左上圖可以看到隨著 epoch 數增加，loss 並沒有下降，是因為我們已經走到 loss function 的 minimum。上圖可以看到隨著 batchsize 增加，loss 先下降後上升。loss 下降是因為增加 batch size 有助於 model 往更正確的方向做 gradient descent，而當 batch size 達到一定數量後，model 做 gradient descent 的次數會減少，造成 loss 上升。而左上圖表示當 learning rate 太高時，model 反而會跳過 minimum。

Problem 2:



再 logistic loss 這題可以發現 batch size 越小，learning rate 越高，造成的 loss 越低。

Conclusion:

本次作業探討如何利用 back propagation 來計算 gradient，在本作業中，我們學習到許多使用深度學習套件所體驗不到的知識，受益良多。