# ASSIGNMENT 3: [IFT6390]

LÉA RICARD & JOSEPH D. VIVIANO

## 1. Derivatives and relationships between basic functions

1.1.

$$sigmoid(x) = \frac{1}{2}\left(tanh(\frac{1}{2}x) + 1\right) \tag{1.1}$$

$$= \frac{1}{2}\left(\frac{e^{\frac{1}{2}x} - e^{\frac{-1}{2}x}}{e^{\frac{1}{2}x} + e^{\frac{-1}{2}x}} + 1\right) \tag{1.2}$$

$$\tag{1.3}$$

We substitue 1 for $\frac{e^{\frac{1}{2}x}+e^{\frac{-1}{2}x}}{e^{\frac{1}{2}x}+e^{\frac{-1}{2}x}}$ to get:

$$sigmoid(x) = \frac{1}{2}\left(\frac{e^{\frac{1}{2}x} - e^{\frac{-1}{2}x}}{e^{\frac{1}{2}x} + e^{\frac{-1}{2}x}} + \frac{e^{\frac{1}{2}x} + e^{\frac{-1}{2}x}}{e^{\frac{1}{2}x} + e^{\frac{-1}{2}x}}\right) \tag{1.4}$$

$$= \frac{1}{2}\left(\frac{e^{\frac{1}{2}x} - e^{\frac{-1}{2}x} + e^{\frac{1}{2}x} + e^{\frac{-1}{2}x}}{e^{\frac{1}{2}x} + e^{\frac{-1}{2}x}}\right) \tag{1.5}$$

$$= \frac{1}{2}\left(\frac{e^{\frac{1}{2}x} + e^{\frac{1}{2}x}}{e^{\frac{1}{2}x} + e^{\frac{-1}{2}x}}\right) \tag{1.6}$$

$$= \frac{1}{2}\left(\frac{2e^{\frac{1}{2}x}}{e^{\frac{1}{2}x} + e^{\frac{-1}{2}x}}\right) \tag{1.7}$$

$$= \frac{1}{1+e^{-x}} \tag{1.8}$$

1.2.

$$ln(sigmoid(x)) = ln(\frac{1}{1+e^{-x}}) \tag{1.9}$$

$$= ln(1) - ln(1+e^{-x}) \tag{1.10}$$

$$= -ln(1+e^{-x}) \tag{1.11}$$

$$= -softmax(-x) \tag{1.12}$$

*Date*: Oct 2018.

1.3.

$$
\begin{aligned}
\frac{dsigmoid}{dx}(x) &= \frac{d}{dx}(\frac{1}{1+e^{-x}}) && (1.13)\\
&= \frac{d}{dx}(1-e^{-x})^{-1} && (1.14)\\
&= [-1][1+e^{-x}]^{-2}\frac{d}{dx}[1+e^{-x}] && (1.15)\\
&= [-1][1+e^{-x}]^{-2}[e^{-x}]\frac{d}{dx}[-x] && (1.16)\\
&= [-1][1+e^{-x}]^{-2}[e^{-x}][-1] && (1.17)\\
&= \frac{e^{-x}}{(1+e^{-x})^2} && (1.18)
\end{aligned}
$$

We can rewrite the last result as:

$$
\begin{aligned}
\frac{e^{-x}}{(1+e^{-x})^2} &= \frac{e^{-x}+1-1}{(1+e^{-x})^2} && (1.19)\\
&= \frac{1+e^{-x}}{(1+e^{-x})^2} - \frac{1}{(1+e^{-x})^2} && (1.20)\\
&= \frac{1}{1+e^{-x}} - \frac{1}{(1+e^{-x})^2} && (1.21)\\
&= \frac{1}{1+e^{-x}} - \frac{1}{(1+e^{-x})}\frac{1}{(1+e^{-x})} && (1.22)\\
&= \frac{1}{1+e^{-x}}(1-\frac{1}{1+e^{-x}}) && (1.23)\\
&= sigmoid(x)(1-sigmoid(x)) && (1.24)
\end{aligned}
$$

1.4.

$$
\begin{aligned}
\frac{dtanh}{dx}(x) &= \frac{d}{dx}(\frac{e^x-e^{-x}}{e^x+e^{-x}}) && (1.25)\\
&= \frac{\frac{d}{dx}(e^x-e^{-x})(e^x+e^{-x}) - \frac{d}{dx}(e^x+e^{-x})(e^x-e^{-x})}{(e^x+e^{-x})^2} && (1.26)\\
&= \frac{(e^x+e^{-x})(e^x+e^{-x}) - (e^x-e^{-x})(e^x-e^{-x})}{(e^x+e^{-x})^2} && (1.27)\\
&= \frac{(e^x+e^{-x})^2}{(e^x+e^{-x})^2} - \frac{(e^x-e^{-x})^2}{(e^x+e^{-x})^2} && (1.28)\\
&= 1-tanh^2(x) && (1.29)
\end{aligned}
$$

1.5.

$$
sign(x)\begin{cases} 1 & x>0\\ 0 & x=0\\ -1 & x<0 \end{cases} \qquad (1.30)
$$

Can be written as athe sum of two indicator functions:

$$sign(x) = \mathbf{1}_{\{x>0\}} - \mathbf{1}_{\{x<0\}} \tag{1.31}$$

$x = 0$ is handled implicitly by this notation, because it matches neither condition.

1.6.

$$abs'(x) = sign(x) = \frac{\partial}{\partial x}|x| = \mathbf{1}_{\{x>0\}} - \mathbf{1}_{\{x<0\}} \tag{1.32}$$

Where $abs'(x)$ is technically undefined at zero, but is simply set to 0 using the above notation.

1.7. The rectifier function is 0 for all $x \leq 0$, so:

$$rect'(x) = \mathbf{1}_{\{x>0\}} \tag{1.33}$$

1.8. If $\|x\|_2^2 = \sum_i x_i^2$, then the gradient is:

$$\begin{aligned} \frac{\partial \|x\|_2^2}{\partial x} &= \frac{\partial}{\partial x} \sum_i x_i^2 & (1.34) \\ &= \sum_i \frac{\partial}{\partial x} x_i^2 & (1.35) \\ &= \sum_i 2x_i & (1.36) \end{aligned}$$

1.9. If $\|x\|_1 = \sum_i |x_i|$, then the gradient is:

$$\begin{aligned} \frac{\partial \|x\|_1}{\partial x} &= \frac{\partial}{\partial x} \sum_i |x_i| & (1.37) \\ &= \sum_i \frac{\partial}{\partial x} |x_i| & (1.38) \\ &= \sum_i sign(x_i) & (1.39) \\ &= \sum_i \mathbf{1}_{\{x>0\}} - \mathbf{1}_{\{x<0\}} & (1.40) \end{aligned}$$

## 2. Gradient computation for parameters optimization in a neural net for multiclass classification

2.1. The dimension of $\mathbf{b}^{(1)}$ is: $\mathbf{b}^{(\mathbf{1})} \in \mathbb{R}^{d_h}$.

$$\mathbf{h}^a = \mathbf{W}^{(1)T} \cdot \mathbf{x} + \mathbf{b}^{(1)} \tag{2.1}$$

$$\mathbf{h}^a_j = \mathbf{W}^{(1)T}_j \cdot \mathbf{x} + b^{(1)}_j \tag{2.2}$$

$$\mathbf{h}^s = g(\mathbf{h}^a) \tag{2.3}$$

Where $g(x)$ is the activation function (i.e., ReLU nonlinearity) applied element wise to the hidden layer. $g(x) = max(0, x)$.

2.2. The dimensions of $\mathbf{W}^{(2)}$ and $\mathbf{b}^{(2)}$ are: $\mathbf{W}^{(2)} \in \mathbb{R}^{m \times d_h}$, and $\mathbf{b}^{(2)} \in \mathbb{R}^m$.

$$\mathbf{o}^a = \mathbf{W}^{(2)T} \cdot \mathbf{h}^{\mathbf{s}} + \mathbf{b}^{(2)} \tag{2.4}$$

$$\mathbf{o}^a_k = \mathbf{W}^{(2)T}_k \cdot \mathbf{h}^{\mathbf{s}} + b^{(2)}_k \tag{2.5}$$

2.3. Let's define an m-class softmax: $softmax(x) = \frac{e^{x_i}}{\sum_{i=1}^m e^{x_i}}$. Therefore,

$$\mathbf{o}^s_k = \frac{e^{\mathbf{o}^a_k}}{\sum_{i=1}^m e^{\mathbf{o}^a_k}} \tag{2.6}$$

$$\begin{aligned}\sum_{i=1}^m \mathbf{o}^s_k &= \sum_{i=1}^m \frac{e^{\mathbf{o}^a_k}}{\sum_{i=1}^m e^{\mathbf{o}^a_k}} && (2.7)\\ &= \frac{\sum_{i=1}^m e^{\mathbf{o}^a_k}}{\sum_{i=1}^m e^{\mathbf{o}^a_k}} && (2.8)\\ &= 1. && (2.9)\end{aligned}$$

Since $e^x > 0$, the result will always be positive. This is crucial because we need the softmax to produce a probability distribution over our $m$ output classes.

2.4.

$$L(\mathbf{o}^a, y) = -\sum_{k=1}^M y_k \log\left(\frac{e^{\mathbf{o}^a_k}}{\sum_{i=1}^m e^{\mathbf{o}^a_k}}\right) \tag{2.10}$$

2.5. The set of parameters is:

$$\theta = \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \mathbf{W}^{(2)}, \mathbf{W}^{(2)}\} \tag{2.11}$$

Since $\mathbf{W}^{(1)} \in \mathbb{R}^{d_h \times d}$, $\mathbf{b}^{(1)} \in \mathbb{R}^{d_h}$, $\mathbf{W}^{(2)} \in \mathbb{R}^{m \times d_h}$, and $\mathbf{b}^{(2)} \in \mathbb{R}^m$, there is a total of $n_\theta = d_h \times (d+1) + m \times (d_h + 1)$ scalar parameters.

The empirical risk $\hat{R}$ associated with the lost function is:

$$\hat{R} = \frac{1}{n}\sum_{i=1}^{n} L(f_\theta(\mathbf{x}^{(i)}), y) \tag{2.12}$$

The optimization problem of training the network in order to find the optimal values of the parameters is:

$$\theta^* = argmin\hat{R}_\lambda(f_\theta, D_n) \tag{2.13}$$

Where we can add a regularization term, $\lambda\Omega(\theta)$, to the empirical risk.

2.6.

$$\theta \leftarrow \theta - \eta\frac{\partial\hat{R}\lambda}{\partial\theta} \tag{2.14}$$

$$\leftarrow \theta - \eta\frac{\partial}{\partial\theta}\left(\frac{1}{n}\sum_{i=1}^{n} L(f_\theta(\mathbf{x}^{(i)}), y^{(1)})\right) \tag{2.15}$$

Again, we haven't, but we could add a regularization term at the end of this equation: $+\eta\frac{\partial}{\partial\theta}\Omega(\theta)$.

2.7. Cross Entropy Loss with Softmax function are used as the output layer extensively. Now we use the derivative of softmax [1] that we derived earlier to find the derivative of the cross entropy loss function.

First, we re-arrange our equation to allow for easy differentiation:

$$L(\mathbf{o}_k^a, y) = -\sum_{k=1}^{M} y_k \log\left(\frac{e^{\mathbf{o}_k^a}}{\sum_{i=1}^{m} e^{\mathbf{o}_i^a}}\right) \tag{2.16}$$

$$= \sum_{k=1}^{M} y_k - \log y_k * \left(\frac{e^{\mathbf{o}_k^a}}{\sum_{i=1}^{m} e^{\mathbf{o}_i^a}}\right) \tag{2.17}$$

$$= -\log y_k * \left(\frac{e^{\mathbf{o}_k^a}}{\sum_{i=1}^{m} e^{\mathbf{o}_i^a}}\right) - \sum_{j\neq y}^{M-1} y_j \log y_j * \left(\frac{e^{\mathbf{o}o_k^a}}{\sum_{i=1}^{m} e^{\mathbf{o}_i^a}}\right) \tag{2.18}$$

$$\tag{2.19}$$

Note that the right-hand term where $j \neq y$ is all zero because $y_k$ is all 0, and our target $y_k$ is 1, so

$$\begin{align}
L(\mathbf{o}^a_k, y) &= -\log\left(\frac{e^{\mathbf{o}^a_k}}{\sum_{i=1}^m e^{\mathbf{o}^a_i}}\right) \tag{2.20}\\
&= -\log(e^{\mathbf{o}^a_k}) + \log(\sum_{i=1}^m e^{\mathbf{o}^a_i}) \tag{2.21}\\
&= -\mathbf{o}^a_k + \log(\sum_{i=1}^m e^{\mathbf{o}^a_i}) \tag{2.22}\\
&= -\mathbf{o}^a_k + \log(\sum_{i\neq k}^m e^{\mathbf{o}^a_i} + e^{\mathbf{o}^a_k}) \tag{2.23}
\end{align}$$

Now we are ready to take the derivative with respect to the output layer when we have the correct class k:

$$\begin{align}
\frac{\partial L}{\partial \mathbf{o}^a_k} &= -1 + \frac{e^{\mathbf{o}^a_k}}{\sum_{i\neq k}^m e^{\mathbf{o}^a_i} + e^{\mathbf{o}^a_k}} \tag{2.24}\\
&= \frac{e^{\mathbf{o}^a_k}}{\sum_{k=1}^m e^{\mathbf{o}^a_k}} - 1 \tag{2.25}\\
&= \mathbf{o}^s_k - 1 \tag{2.26}
\end{align}$$

Since $onehot_m(y) = 1$ when $m$ is the target and is 0 otherwise, we see that the above is true. We can similarly take the derivative with respect to the output layer when we have the incorrect class i:

$$\begin{align}
\frac{\partial L}{\partial \mathbf{o}^a_i} &= \frac{e^{\mathbf{o}^a_i}}{\sum_{i\neq k}^m e^{\mathbf{o}^a_i} + e^{\mathbf{o}^a_k}} \tag{2.27}\\
&= \mathbf{o}^s_i - 0 \tag{2.28}
\end{align}$$

2.8. The numpy equivalent of the expression is given by:

```
onehot = OneHotEncoder(sparse=False)

def softmax_backward(y_hat, y):
    y_one = onehot.fit_transform(y.reshape(-1,1))
    return(y_hat - y_one)

def _softmax(self, X):
    """numerically stable softmax"""
    exps = np.exp(X - np.max(X))
    return(exps / np.sum(exps))
```

```
os = self._softmax(oa)
grad_oa = softmax_backward(os, y)
```

2.9.

$$\frac{\partial L}{\partial W^{(2)}_{kj}} = \sum_{i=1}^{m} \frac{\partial L}{\partial o^a_i} \frac{\partial o^a_i}{\partial W^{(2)}_{kj}} \tag{2.29}$$

$$\frac{\partial L}{\partial b^{(2)}_{k}} = \sum_{i=1}^{m} \frac{\partial L}{\partial o^a_i} \frac{\partial \mathbf{o}^a_i}{\partial b^{(2)}_{k}} \tag{2.30}$$

We have already defined $\frac{\partial L}{\partial o^a_k}$. $\frac{\partial o^a_k}{\partial W^{(2)}_{kj}}$ and $\frac{\partial \mathbf{o}^a_k}{\partial b^{(2)}_{k}}$ are given by:

$$\frac{\partial o^a_i}{\partial W^{(2)}_{kj}} = \frac{\partial}{\partial W^{(2)}_{kj}} (W^{(2)}_{kj} h^s_j + b^{(2)}_k) \tag{2.31}$$

$$= \frac{\partial}{\partial W^{(2)}_{kj}} (h^s_j W^{(2)}_{kj} + b^{(2)}_k) \tag{2.32}$$

$$= h^s_j \tag{2.33}$$

$$\frac{\partial \mathbf{o}^a_i}{\partial b^{(2)}_{k}} = \frac{\partial}{\partial b^{(2)}_{k}} (W_k j^{(2)} \mathbf{h}^s_j + b^{(2)}_k) \tag{2.34}$$

$$= 1 \tag{2.35}$$

2.10.

$$\frac{\partial L}{\partial \mathbf{W}^{(2)}} = \frac{\partial L}{\partial \mathbf{o}^a} \frac{\partial \mathbf{o}^a}{\partial \mathbf{W}^{(2)}} \tag{2.36}$$

$$\frac{\partial \mathbf{o}^a}{\partial \mathbf{W}^{(2)}} = \frac{\partial}{\partial \mathbf{W}^{(2)}} (\mathbf{W}^{(2)T} \mathbf{h}^s + \mathbf{b}^{(2)}) \tag{2.37}$$

$$= \mathbf{h}^{sT} \tag{2.38}$$

and

$$\frac{\partial L}{\partial \mathbf{b}^{(2)}} = \frac{\partial L}{\partial \mathbf{o}^a} \frac{\partial \mathbf{o}^a}{\partial \mathbf{b}^{(2)}} \tag{2.39}$$

$$\frac{\partial \mathbf{o}^a}{\partial \mathbf{b}^{(2)}} = \frac{\partial}{\partial \mathbf{b}^{(2)}} (\mathbf{W}^{(2)T} \mathbf{h}^s + \mathbf{b}^{(2)}) \tag{2.40}$$

$$= \mathbf{1} \tag{2.41}$$

Where $\mathbf{W}^{(2)} \in \mathbb{R}^{m \times d_h}$, $\mathbf{b}^{(2)} \in \mathbb{R}^m$, $\mathbf{o}^a \in \mathbb{R}^m$, $\mathbf{h}^s \in \mathbb{R}^{d_h}$, $\mathbf{1} \in 1^m$ and $\frac{\partial L}{\partial o^a} \in \mathbb{R}^m$.

The numpy form is:

```
grad_b2 = np.sum(grad_oa, axis=0)
grad_W2 = hs.T.dot(grad_oa)
```

2.11.

$$\frac{\partial L}{\partial h_j^s} = \sum_{k=1}^{m} \frac{\partial L}{\partial \mathbf{o}_k^a} \frac{\partial \mathbf{o}_k^a}{\partial h_j^s} \tag{2.42}$$

We have already defined $\frac{\partial L}{\partial \mathbf{o}_k^a}$ and we can calculate $\frac{\partial \mathbf{o}_k^a}{\partial h_j^s}$ by:

$$\frac{\partial \mathbf{o}_k^a}{\partial h_j^s} = \frac{\partial}{\partial h_j^s}(W_{kj}^{(2)} h_j^s + b^{(2)}) \tag{2.43}$$

$$= W_{kj}^{(2)} \tag{2.44}$$

2.12.

$$\frac{\partial L}{\partial \mathbf{h}^s} = \frac{\partial L}{\partial \mathbf{o}^a} \frac{\partial \mathbf{o}^a}{\partial \mathbf{h}^s} \tag{2.45}$$

We have aleady defined $\frac{\partial L}{\partial \mathbf{o}^a}$. The gradient of $\frac{\partial \mathbf{o}^a}{\partial \mathbf{h}^s}$ is given by:

$$\frac{\partial \mathbf{o}^a}{\partial \mathbf{h}^s} = \frac{\partial}{\partial \mathbf{h}^s}(\mathbf{W}^{(2)T}\mathbf{h}^s + \mathbf{b}^{(2)}) \tag{2.46}$$

$$= \mathbf{W}^{(2)T} \tag{2.47}$$

Where $\mathbf{W}^{(2)} \in \mathbb{R}^{mxd_h}$, $\mathbf{h}^s \in \mathbb{R}^{d_h}$, $\mathbf{b}^{(2)} \in \mathbb{R}^m$, $\mathbf{o}^s \in \mathbb{R}^m$ and $\mathbf{y} \in \mathbb{R}^m$.

The numpy form is:

```
grad_hs = self.W2.dot(grad_oa.T)
```

2.13.

$$\frac{\partial L}{\partial h_j^a} = \frac{\partial L}{\partial h_j^s} \frac{\partial h_j^s}{\partial h_j^a} \tag{2.48}$$

Where:

$$\frac{\partial h_j^s}{\partial h_j^a} = \begin{cases} 0 & \text{if } h_j^a < 0 \\ 1 & \text{if } h_j^a > 0 \end{cases} \tag{2.49}$$

And is undefined if $h_j^a = 0$

2.14.

$$\frac{\partial L}{\partial \mathbf{h}^a} = \frac{\partial L}{\partial \mathbf{h}^s} \frac{\partial \mathbf{h}^s}{\partial \mathbf{h}_j} \tag{2.50}$$

Where:

$$\frac{\partial \mathbf{h}^s}{\partial \mathbf{h}^a} = \mathbf{I}_{\{h_j^a > 0\}} \tag{2.51}$$

Where $\mathbf{I} \in \mathbb{R}^{d_h}$.

The numpy form is:

```
def relu_backward(self, z):
    z[z <= 0] = 0
    z[z > 0] = 1
    return(z)

dhs_ha = relu_backward(ha)
grad_ha =  dhs_ha * grad_hs.T
```

2.15.
$$\frac{\partial L}{\partial W_{ji}^{(1)}} = \sum_{k=1}^{d_h} \frac{\partial L}{\partial h_k^a} \frac{\partial h_k^a}{W_{ji}^{(1)}} \tag{2.52}$$

We have already defined $\frac{\partial L}{\partial h_j^a}$. The gradient $\frac{\partial h_k^a}{W_{ji}^{(1)}}$ is given by:

$$\frac{\partial h_k^a}{W_{ji}^{(1)}} = \frac{\partial}{W_{ji}^{(1)}}(W_{ji}^{(1)} x_i + b_j^{(1)}) \tag{2.53}$$
$$= \frac{\partial}{\partial W_{ji}^{(1)}}(x_i W_{ji}^{(1)} + b_j^{(1)}) \tag{2.54}$$
$$= x_i \tag{2.55}$$

and

$$\frac{\partial L}{\partial b_j^{(1)}} = \sum_{k=1}^{d_h} \frac{\partial L}{\partial h_k^a} \frac{\partial h_k^a}{b_j^{(1)}} \tag{2.56}$$

Again, we have already defined $\frac{\partial L}{\partial h_k^a}$. The gradient $\frac{\partial h_k^a}{b_j^{(1)}}$ is given by:

$$\frac{\partial h_k^a}{b_j^{(1)}} = \frac{\partial}{b_j^{(1)}}(W_{ji}^{(1)T} x_i + b_j^{(1)}) \tag{2.57}$$
$$= 1 \tag{2.58}$$

2.16.
$$\frac{\partial L}{\partial \mathbf{W}^{(1)}} = \frac{\partial L}{\partial \mathbf{h}^a} \frac{\partial \mathbf{h}^a}{\mathbf{W}^{(1)}} \tag{2.59}$$

$$\frac{\partial \mathbf{h}^a}{\mathbf{W}^{(1)}} = \frac{\partial}{\mathbf{W}^{(1)}}(\mathbf{W}^{(1)} \mathbf{x}^T + \mathbf{b}^{(1)}) \tag{2.60}$$
$$= \mathbf{x}^T \tag{2.61}$$

and

$$\frac{\partial L}{\partial \mathbf{b}^{(1)}} = \frac{\partial L}{\partial \mathbf{h}^a} \frac{\partial \mathbf{h}^a}{\mathbf{b}^{(1)}} \tag{2.62}$$

$$\frac{\partial \mathbf{h}^a}{\partial \mathbf{b}^{(1)}} = \frac{\partial}{\mathbf{b}^{(1)}}(\mathbf{W}^{(1)}\mathbf{x}^T + \mathbf{b}^{(1)}) \tag{2.63}$$
$$= \mathbf{1} \tag{2.64}$$

Where $\mathbf{h}^a \in \mathbb{R}^{d_h}$, $\mathbf{W}^{(1)} \in \mathbb{R}^{d_h \times d}$, $\mathbf{b}^{(1)} \in \mathbb{R}^{d_h}$, $\mathbf{x} \in \mathbb{R}^d$ and $\mathbf{1} \in \mathbb{R}^{d_h}$

The numpy form is:

```
grad_W1 = X.T.dot(grad_ha)
grad_b1 = np.sum(grad_ha, axis=0)
```

2.17.

$$\frac{\partial L}{\partial \mathbf{x}} = \frac{\partial L}{\partial \mathbf{h}^a}\frac{\partial \mathbf{h}^a}{\mathbf{x}} \tag{2.65}$$

We have already defined $\frac{\partial L}{\partial \mathbf{h}^a}$. The gradient $\frac{\partial \mathbf{h}^a}{\mathbf{x}}$ is given by:

$$\frac{\partial \mathbf{h}^a}{\mathbf{x}} = \frac{\partial}{\mathbf{x}}(\mathbf{W}^{(1)}\mathbf{x}^T + \mathbf{b}^{(1)}) \tag{2.66}$$
$$= \mathbf{W}^{(1)T} \tag{2.67}$$

2.18. Given an elestic net regularization:

$$L(\theta) = L(\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \mathbf{W}^{(2)}, \mathbf{b}^{(2)}) \tag{2.68}$$
$$= \lambda_{11}||\mathbf{W}^{(1)}||_1 + \lambda_{12}||\mathbf{W}^{(1)}||_2^2 + \lambda_{21}||\mathbf{W}^{(2)}||_1 + \lambda_{21}||\mathbf{W}^{(2)}||_2^2 \tag{2.69}$$

The gradient of the regularizers are added to the unregularized gradient of the parameters to update. The regularizers affect only the parameters $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$.

The gradient of the $L_1$ and $L_2$ regularization is given by:

$$\Delta_\theta L(\theta) = \lambda_{11} sign(w_j^{(1)}) + \lambda_{12}[2\mathbf{W}^{(1)}] + \lambda_{21} sign(w_j^{(2)}) + \lambda_{22}[2\mathbf{W}^{(2)}] \tag{2.70}$$

Where $sign(w_j)$ gives a vector of the the sign of each element of $\mathbf{W}$ undefined at zero. To recap, the gradient of the parameters $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$ are, with regularization, given by:

$$\Delta_{W^{(1)}} = -\frac{\partial L}{\partial \mathbf{W}^{(1)}} - \lambda_{11} sign(w_j^{(1)}) - \lambda_{12}[2\mathbf{W}^{(1)}] \tag{2.71}$$

$$\Delta_{W^{(2)}} = -\frac{\partial L}{\partial \mathbf{W}^{(2)}} - \lambda_{21} sign(w_j^{(2)}) - \lambda_{22}[2\mathbf{W}^{(2)}] \tag{2.72}$$

## 3. Neural network implementation and experiments

**3.1. Details on the Implementation.** All experiments were conducted using a single python file, attached. Below we detail some key subclasses of the implementation.

3.1.1. *Numerically Stable Softmax.* The numerically stable softmax was implemented this way in order to deal with single samples, or minibatches, interchangably:

```
def _softmax(self, X):
    """numerically stable softmax"""
    exps = np.exp(X - np.max(X, axis=1).reshape(-1, 1))
    return(exps / np.sum(exps, axis=1).reshape(-1, 1))
```

3.1.2. *Parameter Initialization.* We implemented Glorot-style initialization this way:

```
def _init_W(self, n_in, n_out):
    """initializes weight matrix using glorot initialization"""
    # Sample weights uniformly from [-1/sqrt(in), 1/np.sqrt(in)].
    bound = 1/np.sqrt(n_in)
    return(np.random.uniform(low=-bound, high=bound, size=(n_in, n_out)))
```

3.1.3. *Gradient Checking.* The crucial elements of the gradient checker are as follows:

```
    for param, grad in self._get_params():
        for theta_i in range(param.shape[0]):
            for theta_j in range(param.shape[1]):

                eps = np.random.uniform(10e-6, 11e-4)
                param[theta_i, theta_j] += eps
                loss_mod = self._nll_loss(X, y)
                param[theta_i, theta_j] -= eps

                # grad_bprop normalized by batch size in bprop function
                grad_fd = (loss_mod - loss_raw) / (eps)
                grad_bprop = grad[theta_i, theta_j]

                ratio = grad_fd / grad_bprop

                if ratio < 0.99 or ratio > 1.01:
                    raise Exception('check ur grads bra')
```

3.1.4. *Forward Propogation.* Our implementation of fprop loops over subjects, or processes them using matrix math, using the same bones:

```
def fprop(self, X, train=True):
    """take inputs, and push them through to produce a prediction y"""

    if self.stupid_loop:

        all_ha, all_hs, all_oa, all_os = [], [], [], []

        for x in X:
            x = x.reshape(1,-1)  # reshape so that dimensions are kosher

            ha_sing = x.dot(self.W1) + self.b1;       all_ha.append(ha_sing)
            hs_sing= self._relu(ha_sing);             all_hs.append(hs_sing)
            oa_sing= hs_sing.dot(self.W2) + self.b2;  all_oa.append(oa_sing)
            os_sing = self._softmax(oa_sing);         all_os.append(os_sing)

        ha = np.vstack(all_ha) # stack for compatibility with non-loop
        hs = np.vstack(all_hs) #
        oa = np.vstack(all_oa) #
        os = np.vstack(all_os) #

    else:
        ha = X.dot(self.W1) + self.b1
        hs = self._relu(ha)
        oa = hs.dot(self.W2) + self.b2
        os = self._softmax(oa) # this is y_hat

    if train == True:
        self.ha, self.hs, self.oa, self.os = ha, hs, oa, os

    return(os)
```

3.1.5. *Backpropagation.* A similar strategy was employed for backpropagation:

```
def bprop(self, X, y_hat, y, train=True):
    """
    backpropogate error between y_hat and y to update all parameters
    dL_b and dL_W are both normalized by batch size
    """
    if self.stupid_loop:

        all_dL_W1, all_dL_b1, all_dL_W2, all_dL_b2 = [], [], [], []

        y = y.reshape(-1,1) # used for k=1 case
        for i, (x, y_hat_1, y_1) in enumerate(zip(X, y_hat, y)):
            x = x.reshape(1,-1)  # reshape so that dimensions are kosher

            # gradients for output --> hidden layer
            dL_oa = self._softmax_backward(y_hat_1, y_1)
            dL_hs = self.W2.dot(dL_oa.T)
            hs_sing = self.hs[i, :].reshape(1, -1)
            dL_W2_sing = hs_sing.T.dot(dL_oa)
            dL_b2_sing = dL_oa

            all_dL_W2.append(dL_W2_sing)
            all_dL_b2.append(dL_b2_sing)

            # gradients for hidden --> input layer
            ha_sing = self.ha[i, :].reshape(1, -1)
            dhs_ha = self._relu_backward(ha_sing)
            dL_ha  = dhs_ha * dL_hs.T
            dL_W1_sing = x.T.dot(dL_ha)
            dL_b1_sing = dL_ha

            all_dL_W1.append(dL_W1_sing)
            all_dL_b1.append(dL_b1_sing)

        # NB: instead of dividing by self.this_k, just take the mean
        self.dL_W2 = np.mean(np.stack(all_dL_W2, axis=2), axis=2)
        self.dL_b2 = np.mean(np.stack(all_dL_b2, axis=2), axis=2).ravel()
        self.dL_W1 = np.mean(np.stack(all_dL_W1, axis=2), axis=2)
        self.dL_b1 = np.mean(np.stack(all_dL_b1, axis=2), axis=2).ravel()

    else:
        # NB: divide gradients by self.this_k here!
        # gradients for output --> hidden layer
        dL_oa = self._softmax_backward(y_hat, y)
        dL_hs = self.W2.dot(dL_oa.T)
        self.dL_W2 = self.hs.T.dot(dL_oa) / self.this_k
        self.dL_b2 = np.sum(dL_oa, axis=0) / self.this_k

        # gradients for hidden --> input layer
        dhs_ha = self._relu_backward(self.ha)
        dL_ha  = dhs_ha * dL_hs.T
        self.dL_W1  = X.T.dot(dL_ha) / self.this_k
        self.dL_b1  = np.sum(dL_ha, axis=0) / self.this_k

    # calculate regularization
    reg_l11 = self.l1 * np.sign(self.W1)
```

```
        reg_l21 = self.l1 * np.sign(self.W2)
        reg_l12 = self.l2 * 2 * self.W1
        reg_l22 = self.l2 * 2 * self.W2

        # update all parameters via gradient descent with regularization
        self.W1 -= self.lr * (self.dL_W1 + reg_l11 + reg_l12)
        self.W2 -= self.lr * (self.dL_W2 + reg_l21 + reg_l22)
        self.b1 -= self.lr *  self.dL_b1
        self.b2 -= self.lr *  self.dL_b2
```

3.1.6. *Size of the Mini Batches.* This parameter was controlled by a passable hyperparameter, K:

```
    def _get_minibatches(self, X, shuffle=True):
        """use the number of samples in X and k to determine the batch size"""
        idx = np.arange(X.shape[0])
        if shuffle:
            np.random.shuffle(idx)

        # if the final batch size is smaller than k, append -1s for reshape
        if self.k > 1 and self.k != X.shape[0]:
            rem = len(idx) % self.k
            idx = np.hstack((idx, np.repeat(-1, (self.k-rem))))

        idx = idx.reshape(int(len(idx) / self.k), self.k)

        # load minibatches as a list of numpy arrays
        mbs = []
        for mb in np.arange(idx.shape[0]):
            mbs.append(idx[mb, :])

        # if -1's are in the final batch, remove them
        mbs[-1] = np.delete(mbs[-1], np.where(mbs[-1] == -1)[0])
        if len(mbs[-1]) == 0:
            mbs = mbs[:-1]

        return(mbs)
```

So ends the tour of our code.

3.2. **Gradient Checks With Loops.** Using the circles dataset, we built a small network with two hidden layer neurons, and a minibatch size of 1, and our loop implementation, to do a gradient check:

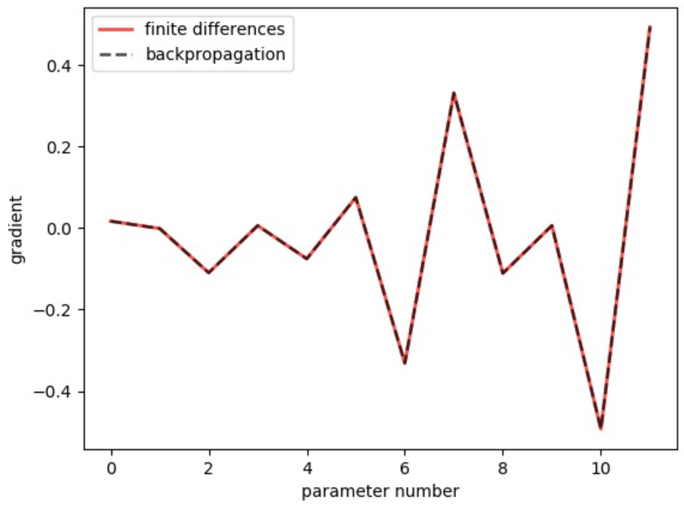

FIGURE 1. Finite Differences vs Backprop Gradients with K=1, Loop Implementation, 2 Hidden Layer Neurons

And followed up with the same analysis using loops and a minibatch size of 10:

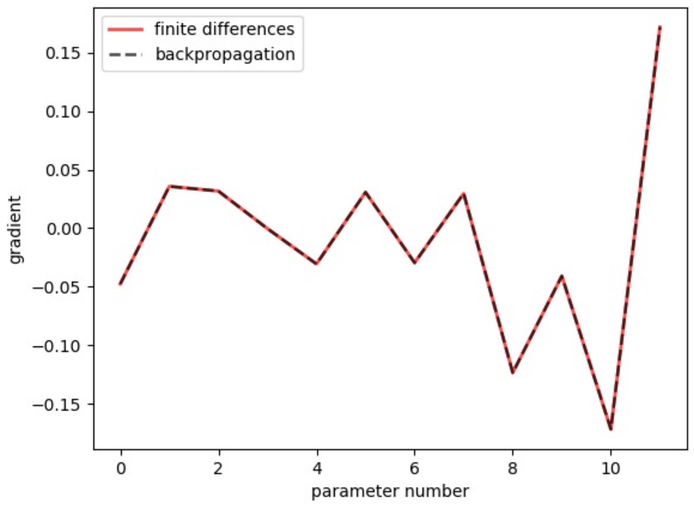

FIGURE 2. Finite Differences vs Backprop Gradients with K=10, Loop Implementation, 2 Hidden Layer Neurons

We hope you agree that they both look good.

3.3. **Hyperparameter Choice and Decision Boundaries.** To evaluate the effect of the hyperparameters on our decision boundaries, we did a grid search over the following:

- **Number of hidden units**: 5, 100
- **Learning rate**: $5 \times 10^{-2}$, $5 \times 10^{-3}$
- **L1 regularization lambda**: 0, 0.01
- **L2 regularization lambda**: 0, 0.01
- **Epochs**: 25, 100

We've attached all of the resulting images in a zip, and show only two here (one bad example, and one good example) for clarity. The number of hidden units has a large effect on capacity. All models trained with 5 hidden units were unable to learn a circular decision boundary:

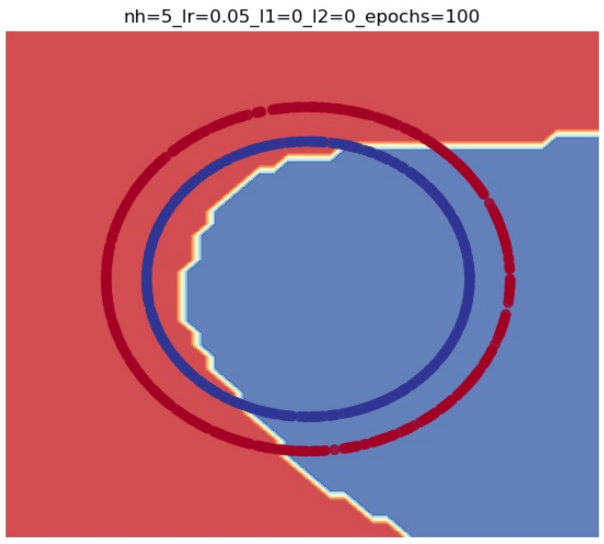

Figure 3. Decision boundary with 5 hidden units.

We found that training the model for longer effectively increased it's capacity, as none of the 25 epoch models were able to perfectly learn the training set. Furthermore, larger values of regularization prevented the model from learning a perfect decision boundary. In some examples in the attached zip, you can see that too much regularization leads to poor gradients and no decision boundry at all.

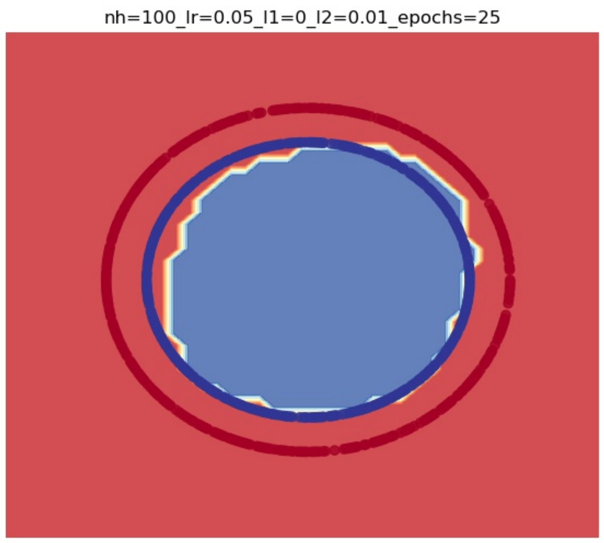

Figure 4. Decision boundary with 100 hidden units and regularization.

A model trained for the full 100 epochs with no regularization and 100 hidden units was able to perfectly memorize the training data.

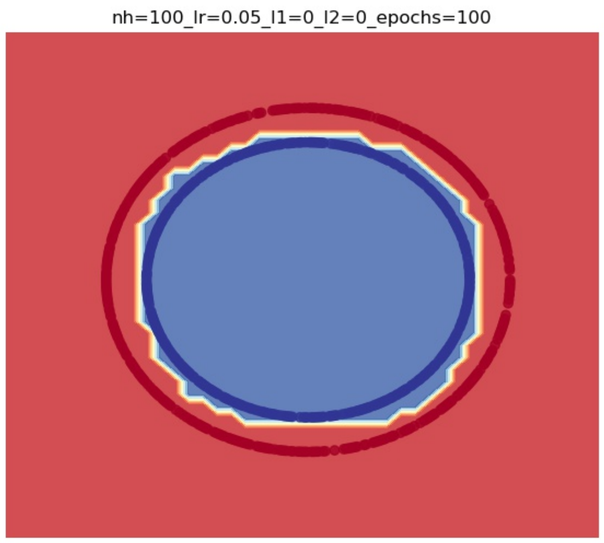

FIGURE 5. Decision boundary with 100 hidden units and no regularization.

3.4. **Matrix Implementation.** See the previous **fprop** and **bprop** methods for the two implementations (loop and vector, see the if self.stupid_loop statement). The key difference between the two methods is that the dimension of the input and output matricies have a singluar dimension, necessitating the use of resaping some vectors to act like matrices: (e.g., y = y.reshape(-1,1) used for k=1 case). When this is done, all other matrix operations (dot products etc.) will work for all $k = n$, since the other matrix dimension will always batch the other parameters when doing dot products.

One key element is the averaging that happens at the end of the loop implementation. In the matrix version of backprop, there is an implicit sum of the gradients over the minibatch that happens right after the softmax, as part of a dot product. When using a loop, no such sum takes place. Therefore, the resulting vectors calculated (and stacked) in the vector implementation are averaged. In the matrix formulation, they are simply divided by the current batch size self.this_k.

We then checked the gradients of this implementation with K=1 and K=10.

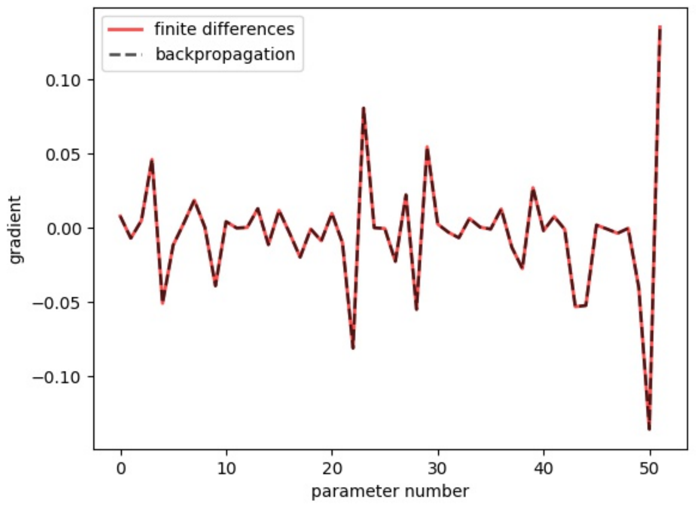

FIGURE 7. Finite Differences vs Backprop Gradients with K=10, Loop Implementation, 2 Hidden Layer Neurons

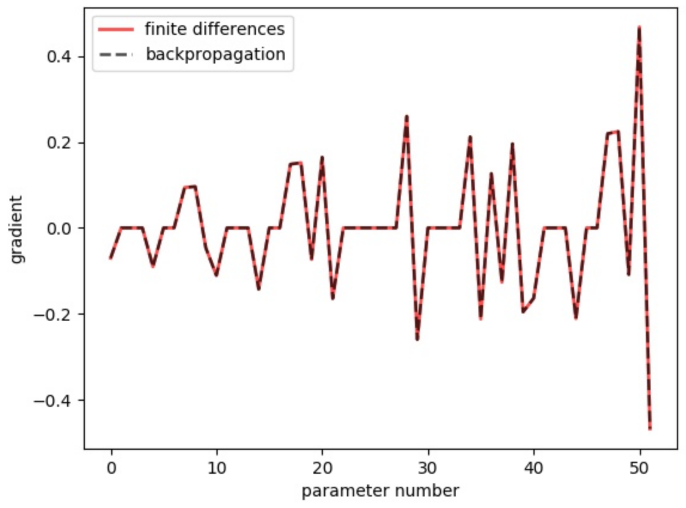

FIGURE 6. Finite Differences vs Backprop Gradients with K=1, Loop Implementation, 2 Hidden Layer Neurons

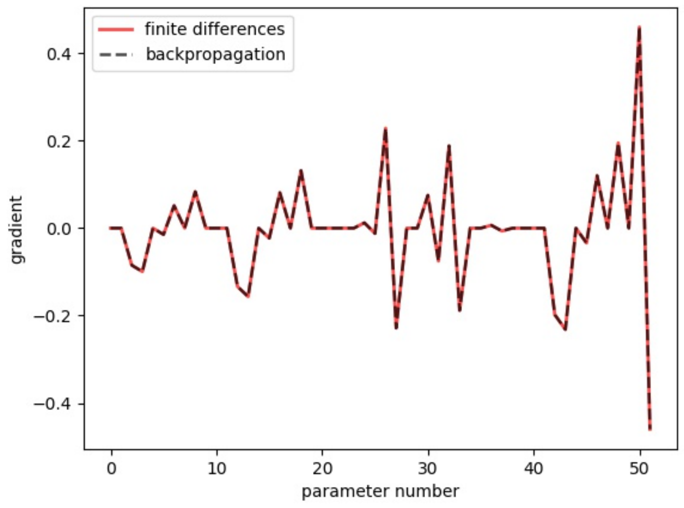

FIGURE 8. Finite Differences vs Backprop Gradients with K=1, Matrix Implementation, 2 Hidden Layer Neurons

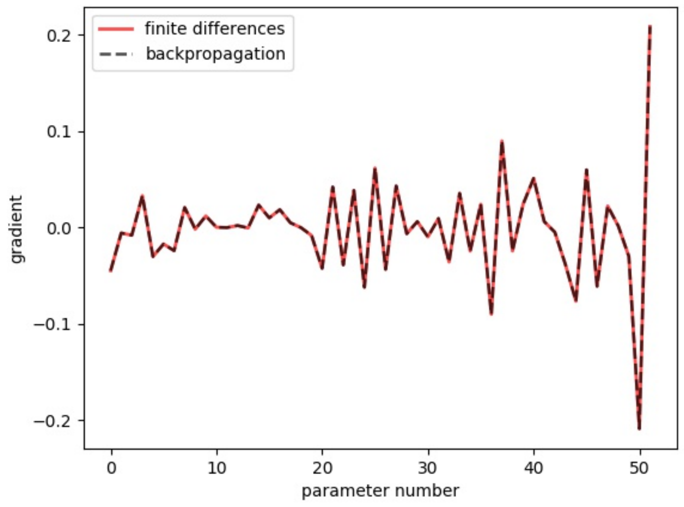

FIGURE 9. Finite Differences vs Backprop Gradients with K=10, Matrix Implementation, 2 Hidden Layer Neurons

3.5. **Fashion MNIST.** The Loop implementation when run on minibatch of 100 for one epoch took approximately 10× as long as the matrix implemntation: FASHION MNIST VECTOR time=218.50477123260498 vs LOOP time=2190.9738194942474 (see mlp_log.txt, attached).

We trained a final model for 100 Epochs using the following hyperparameters:

- k=256
- l1=0
- l2=0.001
- hidden units=50,
- learning rate=0.001

The following are the train, validation, and test loss / accuracy curves:

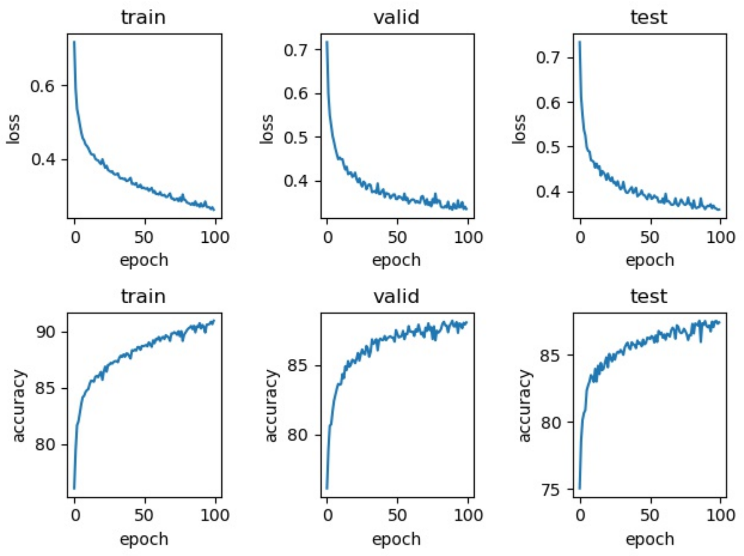

FIGURE 10. Train, Validation, and Test Learning Curves for Fashion MNIST, plotting Negative Log Liklihood and Accuracy

See the attached mlp_log.txt for the full training log.

UNIVERSITÉ DE MONTRÉAL
*E-mail address*: joseph@viviano.ca, lea.ricard@umontreal.ca