**Homework 1 Report** **Ricardo Rei 78047**

Deep Structured Learning (IST, Fall 2018)

Prof. Andre Martins

TAs: Vlad Niculae and Erick Fonseca Due Date: 10/10/18

Question 1

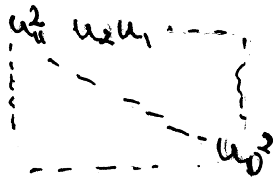
1. $$g(Wu) = \begin{bmatrix} \left(\sum_i^D w_{1i}u_i\right)^2 \\ \vdots \\ \left(\sum_i^D w_{Ki}u_i\right)^2 \end{bmatrix}$$

Selecionando uma linha j de $g(Wu)$:

$$\left(\sum_i^D w_{ji}u_i\right)^2 = \sum_i^D (w_{ji}u_i)^2 + 2\sum_{K=2}^{D}\sum_{i=1}^{K-1} w_{jK}u_K$$

$$\overset{*}{=} u_1^2 w_{j1}^2 + 2\, w_{j2}u_2 \times w_{j1}u_1 + u_2^2 w_{j2}^2 + \dots$$

Considere-se agora: $\phi(u)$ = diagonal superior do produto externo de $u$ com $u$. Vectorizado!




$$\phi(u) = \begin{bmatrix} u_1^2 \\ u_1u_2 \\ u_2^2 \\ u_3u_2 \\ \\ u_D^2 \end{bmatrix}$$

e considere-se uma Matriz $A_\Theta \in \mathbb{R}^{K\times \frac{D(D+1)}{2}}$ cujos valores podem ser escritos à custa de $W$:

$$A_\Theta = \begin{bmatrix} w_{11}^2, & 2w_{11}\times w_{12} & \dots & w_{1D}^2 \\ \\ w_{K}^2, & \dots & & , w_{KD}^2 \end{bmatrix}$$

* $A_\Theta[j,:] \times \phi(u) =$
$w_{j1}^2 u_1^2 + 2w_{j1}w_{j2}u_2u_1 \dots$

Note-se que $A_\Theta \times \phi(u) \overset{*}{=} g(Wu) = h$
e portanto $h$ pode ser escrito como uma única transformação linear se $u$ for transformado em $\phi(u)$ sem que se perca expressividade.

2. $V^T \times A_\Theta = \begin{bmatrix} w_{11}^2 v_1 + 2w_{21} w_{11} \times v_1 \ldots \\ \\ w_K^2 \times v_K + \ldots \quad \ldots \end{bmatrix} = C_\Theta^T$

Ora se $A_\Theta$ é uma transformação linear em relação a $\phi(u)$ então $V^T \times A_\Theta = C_\Theta^T$ é também uma transformação linear em relação a $\phi(u)$ (composição de transformações lineares é também uma transformação linear.

3. b. Ora cada linha $j$ de $C_\Theta$ é na verdade o produto externo vectorizado da linha $w_j \in W$ com ela mesma multiplicada por $v_j$

$\hat{C}_\Theta$ vai ser um conjunto de D matrizes vectorizadas

$$\hat{C}_\Theta = \sum_i^D v_i \, W^T e_i \times e_i^T W$$

(note-se que $W$ e $W^T$ não dependem de $i$

$$= W^T \underbrace{\left( \sum_i^D v_i \, e_i \, e_i^T \right)}_{\text{diagonalização de } v_i \rightarrow S} W$$

Para matrizes simetricas podemos escrever a sua ortogonalização através de: $\hat{C}_\Theta = U^T S \, U$ onde $U$ são as eigenvectors dos eigenvalues da ~~ma~~ diagonal de $S$

Note-se que $W \in \mathbb{R}^{D \times K}$ e $\hat{C}_\Theta \in \mathbb{R}^{D \times D}$

Mas dado que $K \geq D$ então $W$ vai ter no mínimo

D vectores proprios e valores proprios.

Esses D vectores proprios e valores proprios permitem

escrever $\hat{C}_\Theta$ à custa de $W$ na forma:

$$\hat{C}_\Theta = U^T S U$$

sem que com isso se perca expressividade.

Este modelo resume-se portanto a uma transformação linear

aplicada por $C_\Theta$. $\Rightarrow$ linear model

5. Como é uma transformação linear então sabemos que

o modelo se trata de uma Regressão linear e portanto

$L(\Theta, D)$ é uma função convexa com uma única solução

$= 0$ (mínimo global)

↳ Isto não aconteceria se por exemplo fosse usada uma

Sigmoid dado que passaríamos a ter uma regressão

Logistica e portanto introdução de não linearidade.

4. Como vimos anteriormente (alínea 2) $\hat{y}(u; C_\Theta)$ resu-

me-se a um modelo Linear e como tal a sum of



squares do erro é uma função convexa com um único

mínimo global.

# Question 2

## 1. a)

I think this type of representation it's only suitable for simple experimentation of linear methods but it does not exploit relations between adjacent pixels or other type of patterns from the data. Yet, I think it is a good choice for a baseline method.

## 1. b)

In figure 1 we can see that right after the first 2 epochs the train and validation accuracy does not change much. After 20 epochs the accuracy achieved in the training set, validation set, and test set was 54.82%, 54.89%, and 53.33% respectively.

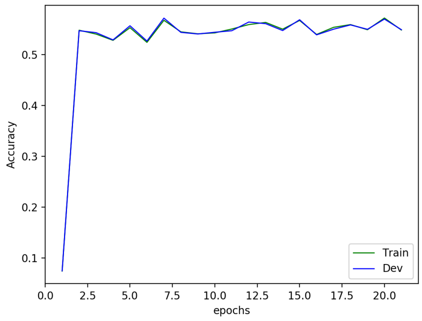



Figure 1: Multi-class Perceptron training.

## 2. a)

From figure 2 we can see that by using a pairwise pixel combination we can achieve a higher accuracy both in the train and validation sets. These features correspond to a Linear Kernel.

After 20 epochs the accuracy achieved in the training set, validation set, and the test set was 94.81%, 85.63%, and 84.09% respectively.

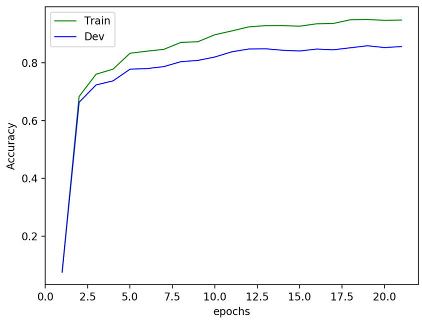



Figure 2: Multi-class Perceptron with Linear Kernel training.

## 2. b)

From figure 3 we see that a Multinomial Logistic Regression with stochastic gradient descent achieves better results than the Perceptron algorithm. After 20 epochs with a simple binary pixel representation, the accuracy achieved in the training set, validation set, and test set was 75.52%, 75.33%, and 72.74% respectively. We can also see from figure 4 that applying a linear kernel improves the results significantly. The accuracy achieved in the training set, validation set and test set with the linear kernel was 95.99%, 88.05%, and 86.62% respectively.

In terms of code, the main difference between the perceptron and the multinomial logistic regression is the way we update weights. Another small difference is that logistic regression also applies a softmax function before the argmax at prediction time. These differences hold between other classifiers such as SVMs. Figure 5 and figure 6 highlight the update difference between the perceptron and logistic regression.

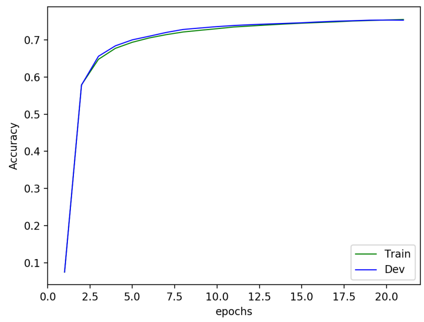



Figure 3: Multinomial Logistic Regression training.

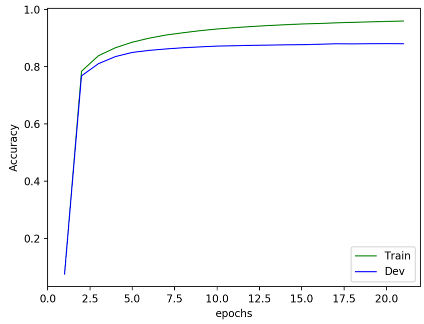



Figure 4: Multinomial Logistic Regression with Linear Kernel training.

```
def update_weights(self, x, y):
        y_pred = self.predict(x)
        if y != y_pred:
            self.parameters[:, y] += np.append(x, [1])
            self.parameters[:, y_pred] -=  np.append(x, [1])

```

Figure 5: Perceptron update rule.

```
def update_weights(self, x, y):
        linear = np.dot(np.append(x, [1]), self.parameters)
        predictions = self.softmax(linear)

        if self.l2 is not None:
            self.parameters = self.parameters - self.lr*(np.outer(predictions, np.append(x, [1])).T - self.l2*self.parameters)
        else:
            self.parameters = self.parameters - self.lr*(np.outer(predictions, np.append(x, [1])).T)
        self.parameters[:, y] = self.parameters[:, y] + self.lr*np.append(x, [1])
```

Figure 6: Multinomial Logistic Regression update rule (with l2-regularization).

### 1. c)

Adding regularization does not seem to improve the results and I ended up not using it. Yet you can see the implementation inside the update_weights function of the MultinomialLR class.

## Question 3

### 1.

A multi-layer perceptron consists of a network of multiple layers of interconnected nodes. Each node is connected to all the nodes of the previous layer through a set of edges that have weights associated. These weights are learned parameters that allow a given node to learn a meaningful combination of features, and thus, learn a new representation based on the input coming from the previous layer. A depth analysis of the expressiveness power of the internally learned representations can be found in the paper presented by Rumelhart et al. (1986).

### 2. and 3.

With a simple binary pixel representation, the multi-layer perceptron with gradient backpropagation algorithm is able to achieve an accuracy of 81.41%, 79.82% and 78.31% in the training, validation and test sets respectively (7). Adding one more hidden layer did not improve the results even after although the accuracy is still improving. This can be the result of a poor initialization.

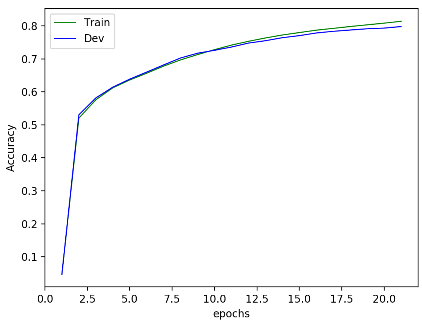



Figure 7: Multi-Layer Perceptron (1 hidden layer) training.

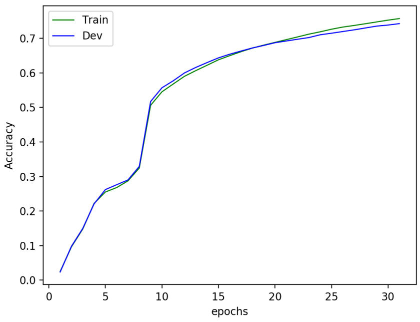



Figure 8: Multi-Layer Perceptron (2 hidden layers) training.

## References

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1. chapter Learning Internal Representations by Error Propagation, pages 318–362. MIT Press, Cambridge, MA, USA.