Course: DD2424 - Assignment 3

In this assignment you will train a ConvNet to predict the language of a surname from its spelling. The inspiration for the assignment's task is the tutorial Classifying Names with a Character-Level RNN. As in this tutorial to make the task slightly harder I removed all accented letters, frequently language specific, and replaced them with their non-accented version. We are thus left with the *English spelling* of each name. As an example the character Š was replaced by S. To a native English speaker this seems perfectly reasonable, but I'm aware that this may seem odd to everybody else and that these accents are not merely decoration. Apologies!

The inspiration for applying a ConvNet to a string of characters is the paper Character-level Convolutional Networks for Text Classification and in this assignment you will implement a simplified version of the network they use.

The final version of your code should contain these major components:

- **Preparing Data**: Read in the training data, determine the maximum length of all the names in the dataset, determine the number of unique characters in the text and set up mapping functions - one mapping each character to a unique index and another mapping each index to a character.
- **Back-propagation**: The forward and the backward pass of the back-propagation algorithm for a ConvNet (with multiple convolutional layers and one fully connected layer) to efficiently compute the gradients for a mini-batch.
- **Mini-batch gradient descent with momentum** to update your ConvNet's parameters.
- **Sampling of the training data** during back-prop training to compensate for the fact that the training data contains unbalanced classes. For example one language has over $9,000$ examples while others have less than one hundred.
- **Functions to compute** the accuracy, loss and confusion matrix on the validation set given the current values of the ConvNet's parameters.

Background 1: *A two layer ConvNet*

The mathematical details of the ConvNet you will implement are now described. Each input in its original form is a fixed length sequence of n_{len}

letters. Each letter will be represented by a one-hot encoding vector of length d. Each name, therefore will be encoded as a matrix of size $d \times n_{\text{len}}$ corresponding to the concatenation of the one-hot (column) vectors for each letter in the name. (If a name has less then n_{len} letters then we will pad the input matrix with column vectors of 0's until the input matrix has n_{len} columns.) Let X represent this input matrix. The network function that you will apply to the input X is:

$$\mathbf{x}_i^{(1)} = \max(0, X * F_{1i}) \quad \text{for } i = 1, \ldots, n_1 \tag{1}$$

$$X^{(1)} = \left(\mathbf{x}_1^{(1)}, \mathbf{x}_2^{(1)}, \ldots, \mathbf{x}_{n_1}^{(1)}\right)^T \tag{2}$$

$$\mathbf{x}_i^{(2)} = \max(0, X^{(1)} * F_{2i}) \quad \text{for } i = 1, \ldots, n_2 \tag{3}$$

$$X^{(2)} = \left(\mathbf{x}_1^{(2)}, \mathbf{x}_2^{(2)}, \ldots, \mathbf{x}_{n_2}^{(2)}\right)^T \tag{4}$$

$$\mathbf{s} = W \text{vec}(X^{(2)}) \tag{5}$$

$$\mathbf{p} = \text{SoftMax}(\mathbf{s}) \tag{6}$$

where

- The $d \times n_{\text{len}}$ input matrix X is sparse and has at most one non-zero (=1) element per column.
- Each filter F_{1i} has size $d \times k_1$ and is applied with stride 1 and no zero-padding. We denote the set of layer 1 filters as $\mathbf{F}_1 = \{F_{1i}, \ldots F_{1,n_1}\}$.
- Each $\mathbf{x}_i^{(1)}$ has size $n_{\text{len}_1} \times 1$ where $n_{\text{len}_1} = n_{\text{len}} - k_1 + 1$ which $\implies$ $X^{(1)}$ has size $n_1 \times n_{\text{len}_1}$. Each row of $X^{(1)}$ corresponds to the output of convolving the input with one of the filters. We use this arrangement of the data to be consistent with Classifying Names with a Character-Level RNN.
- Each filter F_{2i} has size $n_1 \times k_2$ and is applied with stride 1 and no zero-padding. We denote the set of layer 2 filters with $\mathbf{F}_2 = \{F_{2i}, \ldots F_{2,n_2}\}$.
- Each $\mathbf{x}_i^{(2)}$ has size $n_{\text{len}_2} \times 1$ where $n_{\text{len}_2} = n_{\text{len}_1} - k_2 + 1$ which $\implies$ $X^{(2)}$ has size $n_2 \times n_{\text{len}_2}$.
- W, the weight matrix for the fully connected layer, has size $K \times n_2 * n_{\text{len}_2}$ $\implies$ $\mathbf{s}$ has size $K \times 1$.
- In the lecture notes I have the convention that the operator vec$(\cdot)$ makes a vector by traversing the elements of the matrix row by row from left to right. However, in this assignment to make things consistent with `Matlab`'s (:) operation (and how it stores data matrix in memory) we will assume that vec$(\cdot)$ makes a vector by traversing the

elements column by column from top to bottom. This simple example illustrates:

$$X = \begin{pmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{pmatrix} \quad \Longrightarrow \quad \text{vec}(X) = \begin{pmatrix} X_{11} \\ X_{21} \\ X_{12} \\ X_{22} \end{pmatrix}$$

I have omitted all bias terms in the network for the sake of clarity and simplicity of implementation.

Background 2: *Writing the convolution as a matrix multiplication I*

To make the back-propagation algorithm transparent and relatively efficient (as we will be running this on CPU not a GPU and to take computational advantage of the sparse input data) we will set up the convolutions performed as matrix multiplications. In the following we will show how you can set up the appropriate matrix based on the entries of the applied filter for a small example. Let X be a 4×4 input matrix and F be a 4×2 filter:

$$X = \begin{pmatrix} X_{11} & X_{12} & X_{13} & X_{14} \\ X_{21} & X_{22} & X_{23} & X_{24} \\ X_{31} & X_{32} & X_{33} & X_{34} \\ X_{41} & X_{42} & X_{43} & X_{44} \end{pmatrix}, \quad F = \begin{pmatrix} F_{11} & F_{12} \\ F_{21} & F_{22} \\ F_{31} & F_{32} \\ F_{41} & F_{42} \end{pmatrix}. \tag{7}$$

When we convolve X with F (stride 1 and no zero-padding), the output vector has size 3×1.

$$\mathbf{s} = X * F \quad \Longrightarrow \quad \mathbf{s} = M^{\text{filter}}_{F,4}\text{vec}(X) \tag{8}$$

where $M^{\text{filter}}_{F,4}$ is 3×16 and the subscript 4 denotes the number of rows in X. The latter is needed to specify the number of columns in $M^{\text{filter}}_{F,4}$ given the width of F. Thus

$$M^{\text{filter}}_{F,4} = \begin{pmatrix} F_{11} & F_{21} & F_{31} & F_{41} & F_{12} & F_{22} & F_{32} & F_{42} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & F_{11} & F_{21} & F_{31} & F_{41} & F_{12} & F_{22} & F_{32} & F_{42} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & F_{11} & F_{21} & F_{31} & F_{41} & F_{12} & F_{22} & F_{32} & F_{42} \end{pmatrix} \tag{9}$$

$$= \begin{pmatrix} \leftarrow \mathbf{v}^T \rightarrow & \mathbf{0}_4^T & \mathbf{0}_4^T \\ \mathbf{0}_4^T & \leftarrow \mathbf{v}^T \rightarrow & \mathbf{0}_4^T \\ \mathbf{0}_4^T & \mathbf{0}_4^T & \leftarrow \mathbf{v}^T \rightarrow \end{pmatrix} \tag{10}$$

where $\mathbf{v} = \text{vec}(F)$. Hopefully you can glean the pattern for the construction of $M^{\text{filter}}_{F,n_{\text{len}}}$ when X has size $d \times n_{\text{len}}$ and F has size $d \times k$ (where $k \leq n_{\text{len}}$).

In the assignment you will apply multiple filters at each convolutional layer. Continuing with our simple example let's say we apply 2 filters F_1, F_2:

$$\mathbf{s}_i = X * F_i \quad \text{for } i = 2 \tag{11}$$

and these are stacked to get the response map:

$$S = \big(\mathbf{s}_1, \mathbf{s}_2\big)^T. \tag{12}$$

We can apply these two convolutions with this matrix multiplication

$$\text{vec}(S) = \begin{pmatrix} \leftarrow V_{F_1,F_2} \rightarrow & 0_{2\times 4} & 0_{2\times 4} \\ 0_{2\times 4} & \leftarrow V_{F_1,F_2} \rightarrow & 0_{2\times 4} \\ 0_{2\times 4} & 0_{2\times 4} & \leftarrow V_{F_1,F_2} \rightarrow \end{pmatrix} = M^{\text{filter}}_{F_1,F_2,4}\text{vec}(X) \tag{13}$$

where V_{F_1,F_2} is 2×8 matrix with

$$V_{F_1,F_2} = \begin{pmatrix} \text{vec}(F_1)^T \\ \text{vec}(F_2)^T \end{pmatrix} \tag{14}$$

You can, of course, then apply n_f filters to vec(X) with this matrix:

$$M^{\text{filter}}_{\mathbf{F},4} = \begin{pmatrix} \longleftarrow \quad V_{\mathbf{F}} \quad \longrightarrow & 0_{n_f\times 4} & 0_{n_f\times 4} \\ 0_{n_f\times 4} & \longleftarrow \quad V_{\mathbf{F}} \quad \longrightarrow & 0_{n_f\times 4} \\ 0_{n_f\times 4} & 0_{n_f\times 4} & \longleftarrow \quad V_{\mathbf{F}} \quad \longrightarrow \end{pmatrix} \tag{15}$$

where $\mathbf{F} = \{F_1, \ldots, F_{n_f}\}$ and $V_{\mathbf{F}}$ is a $n_f \times 8$ matrix with

$$V_{\mathbf{F}} = \begin{pmatrix} \text{vec}(F_1)^T \\ \vdots \\ \text{vec}(F_{n_f})^T \end{pmatrix} \tag{16}$$

Benefits of this formulation

Efficient forward pass of back-prop During training we need to construct $M^{\text{filter}}_{\mathbf{F}_i, n_{\text{len}_{i-1}}}$ for each layer i after each update of the parameters. However, once we have this matrix it is trivial to apply it to a mini-batch of data. For a mini-batch of input training data, $X_1, \ldots, X_n$ then

$$\big(\text{vec}(S_1) \quad \ldots \quad \text{vec}(S_n)\big) = M^{\text{filter}}_{\mathbf{F}_i, n_{\text{len}_{i-1}}} \big(\text{vec}(X_1) \quad \ldots \quad \text{vec}(X_n)\big) \tag{17}$$

where each S_j is the output of convolving X_j with the filters $\mathbf{F}_i = \{F_{i1}, \ldots, F_{i,n_i}\}$.

Simple and efficient formulation to propagate gradient through a convolution As was shown in the lectures if we write the convolution as a matrix multiplication in the form just described then the gradient of the loss w.r.t. $X^{(1)}$ can be written as

$$\frac{\partial l}{\partial \text{vec}(X^{(1)})} = \frac{\partial l}{\partial \text{vec}(S^{(2)})} M^{\text{filter}}_{\mathbf{F}_2, n_{\text{len}_1}} \tag{18}$$

This matrix multiplication has to be performed for each input in the mini-batch and is done so efficiently with this formulation.

Background 3: *Writing the convolution as a matrix multiplication II*

For the back-propagation algorithm we will also need to write the convolution as a matrix multiplication where the matrix is made up of the elements of the input data. Let's go back to our example where X is the 4×4 input matrix and F is 4×2 filter (see equation (7)). When we convolve X with F (stride 1 and no zero-padding) the output vector has size 3×1. We can write this convolution as a matrix convolution:

$$\mathbf{s} = X * F \qquad \Longrightarrow \qquad \mathbf{s} = M^{\text{input}}_{X,2} \text{vec}(F) \tag{19}$$

where $M^{\text{input}}_{X,2}$ is a matrix of size 3×8 where the subscript 2 corresponds to the width of the filter F and

$$M^{\text{input}}_{X,2} = \begin{pmatrix} X_{11} & X_{21} & X_{31} & X_{41} & X_{12} & X_{22} & X_{32} & X_{42} \\ X_{12} & X_{22} & X_{32} & X_{42} & X_{13} & X_{23} & X_{33} & X_{43} \\ X_{13} & X_{23} & X_{33} & X_{43} & X_{14} & X_{24} & X_{34} & X_{44} \end{pmatrix} \tag{20}$$

$$= \begin{pmatrix} \text{vec}(X_{:,1:2})^T \\ \text{vec}(X_{:,2:3})^T \\ \text{vec}(X_{:,3:4})^T \end{pmatrix} \tag{21}$$

The notation $X_{:,1:2}$ indicates the 4×2 sub-matrix of X corresponding to the columns 1 and 2 of X. Conceptually the rows of $M^{\text{input}}_{X,2}$ correspond to the all sub-blocks of X which are involved with a dot-product with F when the convolution with stride 1 and no zero-padding is applied.

In the assignment you will apply multiple filters at each convolutional layer. As in the previous section first consider the case where 2 filters F_1 and F_2 are applied (see equation (11)):

$$\text{vec}(S) = \begin{bmatrix} \text{vec}(X_{:,1:2})^T & \mathbf{0}_8 \\ \mathbf{0}_8 & \text{vec}(X_{:,1:2})^T \\ \text{vec}(X_{:,2:3})^T & \mathbf{0}_8 \\ \mathbf{0}_8 & \text{vec}(X_{:,2:3})^T \\ \text{vec}(X_{:,3:4})^T & \mathbf{0}_8 \\ \mathbf{0}_8 & \text{vec}(X_{:,3:4})^T \end{bmatrix} \begin{pmatrix} \text{vec}(F_1) \\ \text{vec}(F_2) \end{pmatrix} = M^{\text{input}}_{X,2,2} \, \text{vec}(\mathbf{F}) \tag{22}$$

where the third subscript in $M^{\text{input}}_{X,2,2}$ refers to the number of filters we want to apply. Note we can use the *Kroneckor* product to streamline the notation:

$$M^{\text{input}}_{X,2,2} = \begin{pmatrix} I_2 \otimes \text{vec}(X_{:,1:2})^T \\ I_2 \otimes \text{vec}(X_{:,2:3})^T \\ I_2 \otimes \text{vec}(X_{:,3:4})^T \end{pmatrix} \tag{23}$$

However, at implementation time equation (23) is a very slow way to calculate $M^{\text{input}}_{X,k_i,n_i}$ as it introduces lots of unnecessary multiplications by 0. Though it is a useful formulation as you are probably more likely to write a bug free implementation with this formulation than with a more efficient calculation of $M^{\text{input}}_{X,k_i,n_i}$.

We can then apply 3 filters with this matrix:

$$M^{\text{input}}_{X,2,3} = \begin{bmatrix} \text{vec}(X_{:,1:2})^T & \mathbf{0}_8^T & \mathbf{0}_8^T \\ \mathbf{0}_8^T & \text{vec}(X_{:,1:2})^T & \mathbf{0}_8^T \\ \mathbf{0}_8^T & \mathbf{0}_8^T & \text{vec}(X_{:,1:2})^T \\ \text{vec}(X_{:,2:3})^T & \mathbf{0}_8^T & \mathbf{0}_8^T \\ \mathbf{0}_8^T & \text{vec}(X_{:,2:3})^T & \mathbf{0}_8^T \\ \mathbf{0}_8^T & \mathbf{0}_8^T & \text{vec}(X_{:,2:3})^T \\ \text{vec}(X_{:,3:4})^T & \mathbf{0}_8^T & \mathbf{0}_8^T \\ \mathbf{0}_8^T & \text{vec}(X_{:,3:4})^T & \mathbf{0}_8^T \\ \mathbf{0}_8^T & \mathbf{0}_8^T & \text{vec}(X_{:,3:4})^T \end{bmatrix} \tag{24}$$

$$= \begin{bmatrix} I_3 \otimes \text{vec}(X_{:,1:2})^T \\ I_3 \otimes \text{vec}(X_{:,2:3})^T \\ I_3 \otimes \text{vec}(X_{:,3:4})^T \end{bmatrix} \tag{25}$$

Hopefully now you begin to see the pattern and you can extrapolate what the matrix $M^{\text{input}}_{X,2,n_f}$ will be.

The purpose of this formulation is that we can write the gradient of the loss w.r.t. filters, vec($\mathbf{F}_i$), at layer i, in our two layer network simply as

$$\frac{\partial L}{\partial \text{vec}(\mathbf{F}_i)} = \frac{\partial L}{\partial \text{vec}(S^{(i)})} M^{\text{input}}_{X^{(i-1)},k_i,n_i} \tag{26}$$

Background 4: *Equations for the back-propagation algorithm*

As per usual let L represent the cost function for the mini-batch of data $\mathcal{B}$ that is

$$L(\mathcal{B}, \mathbf{F}_1, \mathbf{F}_2, W) = \frac{1}{|\mathcal{B}|} \sum_{(\mathbf{x},y)\in\mathcal{B}} l_{\text{cross}}(\mathbf{x}, y, \mathbf{F}_1, \mathbf{F}_2, W) \tag{27}$$

and to learn the parameters $\mathbf{F}_1, \mathbf{F}_2, W$ we need to compute the gradient of L w.r.t. each one.

In the description of the back-propagation algorithm we will assume that the $d \times n$ matrix X_{batch} contains the input vectors for each training example in the mini-batch. Thus each column of X_{batch} corresponds to the vectorised version of the corresponding input vector. The notation in the following is slightly inconsistent at times (but hopefully understandable). I refer to the convolution matrix $M^{\text{input}}_{X,k_2,n_2}$ w.r.t. the vectorized version of input matrices instead of the 2D array and a couple of times I use *Matlab* notation to denote the column of a matrix due to fact that I already have too many sub and superscripts to start including more!

Forward Pass

- Construct $M^{\text{filter}}_{\mathbf{F}_1,n_{\text{len}}}$ for the first convolutional layer and $M^{\text{filter}}_{\mathbf{F}_2,n_{\text{len}_1}}$ for the second convolutional layer.
- Apply the first convolutional layer to the input data

$$X^{(1)}_{\text{batch}} = \max(M^{\text{filter}}_{\mathbf{F}_1,n_{\text{len}}} X_{\text{batch}}, 0) \tag{28}$$

- Apply the second convolutional layer

$$X^{(2)}_{\text{batch}} = \max(M^{\text{filter}}_{\mathbf{F}_2,n_{\text{len}_1}} X^{(1)}, 0) \tag{29}$$

- Apply the fully connected matrix

$$S_{\text{batch}} = W X^{(2)}_{\text{batch}} \tag{30}$$

- Apply the SoftMax operator to each column of S_{batch} to get P_{batch}.

For the backward pass of the back-prop algorithm you will need to keep a record of $X^{(1)}_{\text{batch}}, X^{(2)}_{\text{batch}}, P_{\text{batch}}$.

Backward Pass

- Set all the entries in the gradient matrix and vectors, $\frac{\partial L}{\partial W}$, $\frac{\partial L}{\partial \text{vec}(\mathbf{F}_1)}$ and $\frac{\partial L}{\partial \text{vec}(\mathbf{F}_2)}$ to zero.
- Let

$$G_{\text{batch}} = -(Y_{\text{batch}} - P_{\text{batch}}) \tag{31}$$

- Then the gradient of l w.r.t. W is given by

$$\frac{\partial L}{\partial W} = \frac{1}{n}\sum_{j=1}^{n} G_{\text{batch}}(:, j) X^{(2)}_{\text{batch}}(:, j)^T = \frac{1}{n} G_{\text{batch}} X^{(2)T}_{\text{batch}} \tag{32}$$

- Propagate the gradient through the fully connected layer and the second ReLu function.

$$G_{\text{batch}} = W^T G_{\text{batch}} \tag{33}$$

$$G_{\text{batch}} = G_{\text{batch}} \odot \text{Ind}(X^{(2)}_{\text{batch}} > 0) \tag{34}$$

  where $\odot$ represents element-by-element multiplication (i.e. $.*$ in `Matlab` notation) of two matrices.

- Compute the gradient w.r.t. the second layer convolutional filters:

  for $j = 1, \ldots, n$

$$\mathbf{g}_j = G_{\text{batch}}(:, j) \tag{35}$$

$$\mathbf{x}_j = X^{(1)}(:, j) \tag{36}$$

$$\mathbf{v} = \mathbf{g}_j^T M^{\text{input}}_{\mathbf{x}_j, k_2, n_2} \tag{37}$$

$$\frac{\partial L}{\partial \text{vec}(\mathbf{F}_2)} = \frac{\partial L}{\partial \text{vec}(\mathbf{F}_2)} + \frac{1}{n}\mathbf{v} \tag{38}$$

- Propagate the gradient to the previous layer through the second convolutional layer and first ReLu operation:

$$G_{\text{batch}} = \left(M^{\text{filter}}_{\mathbf{F}_2,}\right)^T G_{\text{batch}} \tag{39}$$

$$G_{\text{batch}} = G_{\text{batch}} \odot \text{Ind}(X^{(1)}_{\text{batch}} > 0) \tag{40}$$

- Compute the gradient w.r.t. the first layer convolutional filters:

  for $j = 1, \ldots, n$

$$\mathbf{g}_j = G_{\text{batch}}(:, j) \tag{41}$$

$$\mathbf{x}_j = X_{\text{batch}}(:, j) \tag{42}$$

$$\mathbf{v} = \mathbf{g}_j^T M^{\text{input}}_{\mathbf{x}_j, k_1, n_1} \tag{43}$$

$$\frac{\partial L}{\partial \text{vec}(\mathbf{F}_1)} = \frac{\partial L}{\partial \text{vec}(\mathbf{F}_1)} + \frac{1}{n}\mathbf{v} \tag{44}$$

The description of how to implement the back-prop algorithm I have given is somewhat optimized for implementation in `Matlab`. In many of the deep learning software packages the back-propagation algorithm is frequently implemented via a formulation similar to this. Though of course there is a trade-off between speed and memory consumption. For a certain size of problem it may not be possible to perform the back-prop as described.

Background 5: *Extra measures for a more efficient implementation*

The matrix multiplications performed in equations (37) and (43) are quite computationally wasteful because each $M^{\text{input}}_{\mathbf{x}_j,k_i,n_i}$ has lots of zeros (see equation 24). Thus there are few options to make the computations more efficient

- Pre-compute for the first layer the $M^{\text{input}}_{\mathbf{x}_j,k_1,n_1}$'s and make each one sparse. This is worthwhile because each $M^{\text{input}}_{\mathbf{x}_j,k_1,n_1}$ only depends on the input data and remains fixed throughout training. Also remember each input data-point is encoded as a sparse matrix/vector so $M_{\mathbf{x}_j,k_1,n_1}$ is really very sparse and there are big performance gains to be made with this simple step.

- Unfortunately each $M^{\text{input}}_{\mathbf{x}_j,k_2,n_2}$ cannot be pre-computed, made sparse and then stored because each one is constructed from the responses after applying the first convolution layer and this will change every time the network's parameters change. Also the vectors $\mathbf{x}_j$ at layer 1 are not necessarily sparse. However, do not despair you can still make improvements. Instead of creating $M^{\text{input}}_{\mathbf{x}_j,k_2,n_2}$ you should create $M^{\text{input}}_{\mathbf{x}_j,k_2}$ (the generalization of equation (21) to filters of width k_2 and input of size $n_1 \times n_{\text{len}_1}$) and then you can write

$$\mathbf{v} = \mathbf{g}_j^T M^{\text{input}}_{\mathbf{x}_j,k_2,n_2} \tag{45}$$

equivalently as

$$V = M^T_{\mathbf{x}_j,k_2} \begin{bmatrix} \mathbf{g}_{j,1} & \mathbf{g}_{j,2} & \cdots & \mathbf{g}_{j,n_2} \\ \mathbf{g}_{j,1+n_2} & \mathbf{g}_{j,2+n_2} & \cdots & \mathbf{g}_{j,n_2+n_2} \\ \vdots & \vdots & \vdots & \vdots \\ \mathbf{g}_{j,1+(n_{\text{len}_2}-1)n_2} & \mathbf{g}_{j,2+(n_{\text{len}_2}-1)n_2} & \cdots & \mathbf{g}_{j,n_2+(n_{\text{len}_2}-1)n_2} \end{bmatrix} \tag{46}$$

s.t. $\text{vec}(V) = \mathbf{v}$. It is left as an exercise to show that this is true.

Background 6: *Compensating for unbalanced training data*

In this assignment you encounter unbalanced training data for the first time. If you do not compensate for this fact you run the risk that your ConvNet will just learn to classify the dominating classes correctly. There are several practical ways you can overcome this problem. One way is to define a new loss function for a mini-batch:

$$L_{\text{upsampled}}(\mathcal{B}, \mathbf{F}_1, \mathbf{F}_2, W) = \frac{1}{|\mathcal{B}|} \sum_{(X,y)\in\mathcal{B}} p_y \, l(X, y, \mathbf{F}_1, \mathbf{F}_2, W) \tag{47}$$

where p_y is a weight applied to the loss for each individual training example in the batch. The value p_y has a constant value for each class and is commonly set to

$$p_y = \frac{1}{K}\frac{1}{n_y} \tag{48}$$

where n_y is the total number of training examples (in the whole training set) with label y and K is the number of classes. If you use this loss function then you effectively down-weight the loss from the popular classes much more aggressively than those from the rare classes. The updated loss function then transfers to the gradient calculations in a simple way, for example

$$\frac{\partial L_{\text{upsampled}}}{\partial \mathbf{F}_1} = \frac{1}{|\mathcal{B}|} \sum_{(X,y)\in\mathcal{B}} p_y \frac{\partial l(X, y, \mathbf{F}_1, \mathbf{F}_2, W)}{\partial \mathbf{F}_1} \tag{49}$$

If you use this loss function you will still use the same back-prop equations described in equations (31) - (44) except in equations (32), (38) and (44) where you sum up the individual gradients of the examples in the batch. For these equations you will have to include the re-weighting factor associated with each class.

The other option is that for each epoch of training you randomly sample the same number (this number is usually set to the number of examples from the smallest class) of examples from each class and this becomes the effective training set for this epoch. This latter option is what I implemented for the assignment and it worked fine. You are free to choose either approach.

Exercise 1: *Implement and train a two-layer ConvNet*

In the following I will sketch the different parts you will need to write to complete the assignment. Note it is a guideline. You can, of course, have a different design, but you should read the outline to help inform how different parameters and design choices are made and how I have gone about optimizing calculations that need to be computed.

0.1 Read in the data & get it ready

First you need to read in the dataset from the text file `ascii_names.txt` available the Canvas website. To save you some time I have written `Matlab` code (available from the Canvas page) that will read in the contents of this text file and put all the names and their corresponding labels in the cell array `all_names` and a vector `ys` respectively. The code also converts all uppercase letters to lowercase.

You have access to all the characters used to generate the names. To get a vector containing the unique characters you can apply this *Matlab* code:

```
C = unique(cell2mat(all_names));
d = numel(C);
```

Once you have this list then its length `d` corresponds to the dimensionality of the one-hot vector used to encode each character (d in the **Background 1** section). You can also examine the contents in `all_names` to get the maximum length of name in the dataset and set `n_len` to this value. You can also use the `unique` function on `ys` to get the number of classes `K` we are trying to predict (it is 18).

To allow you to easily go between a character and its one-hot encoding, you should initialize a map container of the form:

```
char_to_ind = containers.Map('KeyType','char','ValueType','int32');
```

Then for `char_to_ind` you should fill in the characters in your alphabet as its keys and create an integer for its value (keep things simple and use where the character appears in the vector `C` as its value). You will use this map container to convert a name into a matrix of size `d` × `n_len`. The ith column of this matrix should correspond to the one-hot encoding of the ith letter in the name. If a name has length less then `n_len` then the last (`n_len` - `n_name`+1) columns of the encoding matrix should be set to all zeros. While if a test name has length greater than `n_len` you can choose what to do. The most obvious choices are to either use the first `n_len` or the last `n_len` characters in the name and ignore the other ones.

Encode your input names I recommend that you convert each name into its matrix encoding and then vectorize it, via the (:) operator, into a vector of length `d * n_len`. You can then store all the vectorized inputs into a matrix `X` of size `(d * n_len)` × `N` where `N` is the number of names in the dataset. If you save `X` you can then load it quickly each time you run your code.

For this assignment we will just partition the data into a training and validation set. (Not a great machine learning practice but the main purpose of this assignment is get the training of a ConvNet up and running efficiently......) You can download the indices of the inputs I used for the validation set, `Validation_Inds.txt` from Canvas webpage. I assume the same ordering of the names in the original input file.

0.2 Set hyper-parameters & initialize the ConvNet's parameters

The hyper-parameters you need to define the ConvNet's architecture are

- the number of filters applied at layer 1: n_1
- the width of the filters applied at layer 1: k_1 (Remember the filter applied will have size $d \times k_1$.)
- the number of filters applied at layer 2: n_2
- the width of the filters applied at layer 2: k_2 (Remember the filter applied will have size $n_1 \times k_2$.)

The other hyper-parameters you need to set are those associated with training and these are the learning rate `eta` and the momentum term `rho`. I used the settings `.001` and `.9`.

In my code I found it easiest to store the parameters of the model in an object called `ConvNet`. In this object I defined a cell array containing the weight/filter matrices for each layer. For example:

```
ConvNet.F{1} = randn(d, k1, n1)*sig1;
ConvNet.F{2} = randn(n1, k2, n2)*sig2;
ConvNet.W = randn(K, fsize)*sig3;
```

where `fsize` is the number of elements in $X^{(2)}$ in equation (4). I use He initialization to set `sig1`, `sig2` and `sig3`. However, you should be careful in setting `sig1` as the input vector is very sparse with the non-zero elements equalling 1 and this changes the assumptions of He initialization. How should you change the formula?

0.3 Construct the convolution matrices

Before you begin to write the code to compute the gradients you need to write one function that will allow to construct the matrices $M^{\text{filter}}_{\mathbf{F}_1,n_{\text{len}}}$, $M^{\text{filter}}_{\mathbf{F}_2,n_{\text{len}_1}}$ and another that will allow you to construct $M^{\text{input}}_{\mathbf{x},k_1,n_1}$ and $M^{\text{input}}_{\mathbf{x},k_2,n_2}$. For the later matrices we have assumed the input vector is already in its vectorized form as this is the form you will have during training and testing. (Remember in *Matlab* you can transfer between the matrix and vector encodings of an input example with

```
x_input = X_input(:)
X_input = reshape(x_input, [d, nlen]);
```

where `X_input` has size `d` × `nlen`. Note if you have `x_input` then you either need to know `d` or `nlen` to do the reshaping as you can find the other dimension of the 2D matrix by dividing the length of x_input by the known dimension.)

It is crucial that you get your implementation of these functions correct as they are central for computing the gradients. Now for some hand-holding! The first function you write (returning the matrix, based on the entries in the convolutional filter, to perform all the convolutions at a given layer) should take as input the 3D array containing the convolutional filters at one layer and the number of columns of the input to that layer. So if it is layer 1 then this latter number is nlen. The declaration of the function could look something like

```
function MF = MakeMFMatrix(F, nlen)
```

The function then outputs the matrix MF of size (nlen-k+1)*nf × nlen*dd where

```
[dd, k, nf] = size(F);
```

The second function you write (returning the matrix, based on the entries in the input vector, to perform all the convolutions at a given layer) should take as input the vector of size × 1 corresponding the vectorized version of the input to the convolutional layer, the height and width of each filter applied and the number of filters that will be applied. So if it is layer 1 and you will apply the 2D filters stored in ConvNet.F{1} then the numbers you need are

```
[d, k, nf] = size(ConvNet.F{1})
```

The declaration of the function could look something like

```
function MX = MakeMXMatrix(x_input, d, k, nf)
```

The function then outputs the matrix MX of size (nlen-k+1)*nf × k*nf*d. The first debug check you can perform is that for the first convolutional layer you compute MX for one training input x_input and MF. Then you can compute

```
s1 = MX * ConvNet.F{1}(:);
s2 = MF * x_input;
```

and check s1 and s2 are the same. To also help confirm you have a bug-free implementation, you can download from the Canvas webpage the file DebugInfo.mat. From this *Matlab* data file you can upload the following variables:

- name - the 1d array of characters containing the original name of the input.
- X_input - the matrix numerical encoding of name of size 28 × nlen
- x_input - the 28*nlen × 1 vector corresponding to the vectorised version of X_input
- F - the 28 × 5 × 4 3D array corresponding to the 4 convolutional filters applied to X_input. The first filter corresponds to F(:, :, 1), the second to F(:, :, 2), etc.
- vecF - the 28*5*4 × 1 vector corresponding to the vectorized version of F. Note in *Matlab* the operation F(:) vectorises F by first flattening F(:, :, 1) then F(:, :, 2) etc.

- S - the 4 × nlen-5+1 matrix corresponding to the output of convolving X_input with the filters stored in F.
- vecS - the 4*(nlen-4) × 1 vector corresponding to the vectorized version of S.

From x_input and F you can use your newly written functions to compute MX and MF and then check if applying these matrices results in the vector vecS. You can also check that if you reshape your calculated version of vecS you get S.

Once you have a bug-free version of MakeMXMatrix and MakeMFMatrix then you can spend some time optimizing your implementation especially (if you have used the function kron to build MX). Remember the profile command is your friend when optimizing code in Matlab.

0.4 Implement the forward & backward pass of back-prop

Now you are ready to implement the back-propagation algorithm described by the equations in the **Background 4** section. I suggest you first start with a mini-batch of size 1, then 2 and then 3 and also that you have a relatively small number of filters per layer (say 5). When I did the numerical checks of my analytical gradients, I applied 5 filters of width 5 in both layers. To perform the numerical checks of your gradient you will need to write the function to compute the loss for a mini-batch of data, see equation (27).

```
function loss = ComputeLoss(X_batch, Ys_batch, MFs, W)
```

where X_batch is matrix of size nlen*d × n that contains all the vectorised input data (each column corresponds to one vectorized datapoint), Y_batch is the one-hot encoding of the labels of each example, MFs is a cell array containing the MF matrix for each layer and W is the 2D weight matrix for the last fully connected layer. Here I suggest you use the MF matrices to carry out the convolutions as this will be the most efficient way to compute the loss on the validation set during training (if you don't explicitly exploit the sparse of X_batch).

From the course Canvas page you can download my code for computing the numerical gradients. You will, of course, have to adapt it to fit your code. As per usual you should take care to make your numerical and analytical gradients match before you move on to the next stage.

0.5 Train using mini-batch gradient descent with momentum

Once you have convinced yourself that your analytic gradient calculations are correct, you are ready to write code to start training the parameters of your network. I suggest that you begin with an implementation of ordinary mini-batch gradient descent. When you have implemented that then upgrade the optimizer to include a momentum term. Just one technical detail in the back-prop equations you compute - the gradients you compute are all vectors but you will probably have stored the convolutional filters as 3D arrays. You can use the `reshape` function to

```
grad_F1 = reshape(grad_vecF1, [d, k1, n1]);
```

I suggest you get the training algorithm working for a small sized network for example filters of size 5 or 3 at each layer and then ∼10 filters at each layer and mini-batch size 100. After every n_{update}th update step you should compute the validation loss and accuracy. I set $n_{\text{update}} = 500$. What I also found useful was to also regularly compute and print out the **confusion matrix** generated by the network when applied to the validation set. The confusion matrix is the $K \times K$ matrix M where M_{ij} is the number of examples with label i that are classified as class j by your classifier. By examining this matrix you can check whether the unbalanced data is causing a problem such as all examples are classified to a small subset of the classes.

When you do the initial experiments with your small network you should get sufficiently good results to indicate

1. the speed bottlenecks in your code when you use the `profile` command,
2. whether some form of useful learning is occurring and
3. whether the unbalanced dataset is causing a problem or not.

At this stage you can decide to implement the computational efficiencies described in the **Background 5** section especially if you want to train a much bigger network for ∼100,000 update steps and to ensure it will not take too long. Another tip is that `Matlab` makes copies of everything when you pass variables to functions (there really is no concept of passing by reference). Thus in the innermost loops it makes sense to cut-and-paste code from the functions you call. Yes, it makes things much less readable but it in many cases it saves lots of time. And as always in *Matlab* pre-allocate memory for matrices whenever you can!

Once you have a relatively efficient implementation then you can start to do some serious training and search for a good parameter setting. What I found, in general, was that the more filters I applied at each layer the better performance I got in a fewer number of update steps.

0.6 Account for the unbalanced dataset

Because the dataset is quite unbalanced it makes sense to compensate for this fact, so if at test time the proportion of names from each class is drastically different then during training your network can cope. In the section

Background 6 I describe how to do this. Please implement one of the alternatives and notice the qualitative differences between training and the intermediary values of the validation loss, accuracy and the confusion matrices when you compensate and do not compensate for the imbalanced classes.

To complete the assignment:

To pass the assignment you need to upload to Canvas:

1. The code for this assignment.

2. A brief pdf report with the following content:

   i) State how you checked your analytic gradient computations and whether you think that your gradient computations are bug free for your ConvNet.

   ii) Comment on how you compensated for the imbalanced dataset during training.

   iii) Include a graph of the validation loss, equation (47), for a longish training $\geq 20,000$ update steps for a network with $k_1 = 5, k_2 = 3$ and $n_1 = 20$, $n_2 = 20$ when you train without compensating for the unbalanced dataset and when you do. Include also the final confusion matrix on the validation set in both cases. Comment on the qualitative differences between the two different training regimes.

   iv) Include the validation loss function, equation (47), for your best performing ConvNet. State the final validation accuracy of this network and its accuracy for each class. Detail the parameter settings and how trained the network and the training parameters.

   v) State whether you implemented the efficiency gains in **Background 5** and by how much they helped speed up training. You can quote how long it took to perform a fixed number of update steps ($\sim$100) with mini-batches of size 100 before and after the upgrade.

   vi) Show the probability vector output by your *best* network when applied to your surname and those of 5 of your friends. Comment on the results!

Exercise 2: *Optional for bonus points*

We have only scratched the surface of all the tweaks one could apply to make our ConvNet a better predictor for this problem. They are many options for improvement

- Implement a max-pooling layer see the reference paper for how they applied this problem. This will allow you to apply more filters at each layer because if you reduce n_{len_i} at each layer then you will have less computational and memory requirements during training for a fixed number of filters.

- Make your code generic so you can define networks with L convolutional layers. And then perform a search for a good number of layers and the number of filters applied at each layer. I'm interested in knowing does adding more convolutional layers make things better or does training become too hard.

- Make your convolution implementation more generic so you can apply filters with arbitrary strides and zero-padding.

- Include the omitted bias terms at each layer.

- Apply dilated convolutions. You could then get long range correlations but with the same number of parameters and computational effort.

- The raw data I used for this assignment contained duplicates and I cleaned it up and made this version available to you. But interestingly in the original data some names had multiple labels. For example the name "Sullivan" appears both with the label 5 and 9. This version of the data with names potentially having multiple labels ($\geq$ 2) is available in the file `ascii_names_with_multi_label.txt`. I have not tried this but you could train a classifier which has soft labels for names with multiple labels. So for example the name Sullivan would have the label $\mathbf{y}$ where $y_i = .5$ for $i = 5, 9$ and zero otherwise. You can still use a cross-entropy loss for training but remember for the training examples not with 1-hot encoding the cross-entropy loss is defined as $-\mathbf{y}^T \log(\mathbf{p})$.

The list is endless... I will award 2 bonus for each reported significant upgrade implemented. You max out after 6 bonus points. I'm aware, of course, that an *improved* network does not necessarily correspond to improved performance, but as long as you implemented something reasonable...