Assignment 4

Submission deadline: classes on **25.11-1.12.2015**

Note: this time Tuesday groups have an extended deadline and submit the last.

Points: 10 + 1 bonus

The goal of this assignment is to implement a small 2-layer neural network and evaluate its performance on the Iris and MNIST datasets.

1 SoftMax Regression

Samples in the Iris dataset belong to one of three classes, while in CIFAR10 and MNIST they belong to one of 10 classes. Thus, linear regression cannot be applied, because it distinguishes between two classes only. We will use SoftMax regression instead.

Let $x \in \mathbb{R}^n$ be a sample vector and $y \in \{1, 2, \ldots, K\}$ its class label. Similarly to what has been done during the lecture, we extend vector x with the bias term $x_0 = 1$ to simplify the calculations (so now $x \in \mathbb{R}^{n+1}$).

In SoftMax regression, we model conditional probability, that a given sample x belongs to class k. Such model is parametrized with a matrix $\Theta \in \mathbb{R}^{K \times n+1}$. Note that in SoftMax regression, a separate linear model is build for each class. First we compute the vector a of total inputs:

$$a_k = \sum_{j=0}^{n} \Theta_{k,j} x_j, \tag{1}$$

or using matrix notation $a = \Theta x$. Then we compute conditional probabilities using SoftMax regression:

$$p(\text{class} = k | x, \Theta) = o_k = \frac{\exp a_k}{\sum_{j=1}^{K} \exp a_j}. \tag{2}$$

Function SoftMax transforms a K-element vector of real numbers to a vector of non-negative numbers which sum to 1. Thus they can be treated as probabilities assigned to K separate classes.

As it is the case with linear regression, we use cross-entropy as the loss function in SoftMax regression:

$$\begin{aligned} J^{(i)}(\Theta) &= -\sum_{k=1}^{K} [y^{(i)} = k] \log o_k^{(i)} \\ J(\Theta) &= \frac{1}{m} \sum_{i=1}^{m} J^{(i)}(\Theta) = -\frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{K} [y^{(i)} = k] \log o_k^{(i)} \end{aligned} \tag{3}$$

where $[y^{(i)} = k]$ equals 1 when the $i-$th sample belongs to class k, and 0 otherwise. Value $[y^{(i)} = k]$ might be interpreted as the correct value of the k-th output of the model on sample i. In addition, the total loss is expressed as a mean loss of particular samples, to make it independent of the size of the training set.

Loss function gradient with respect to total inputs a is simple:

$$\frac{\partial J^{(i)}}{\partial a_k^{(i)}} = o_k^{(i)} - [y^{(i)} = k]. \tag{4}$$

Using the chain rule, the gradient of the loss function with respect to model parameters becomes:

$$\frac{\partial J}{\partial \Theta_{kj}} = \sum_{i=1}^{m} \frac{\partial J}{\partial J^{(i)}} \frac{\partial J^{(i)}}{\partial \Theta_{kj}} = \sum_{i=1}^{m} \frac{1}{m} \cdot \frac{\partial J^{(i)}}{\partial a_k^{(i)}} \frac{\partial a_k^{(i)}}{\partial \Theta_{kj}} = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial J^{(i)}}{\partial a_k^{(i)}} x_j^{(i)}. \tag{5}$$

Problem 1. **[1p]** Show that: $\frac{\partial J^{(i)}}{\partial a_k^{(i)}} = o_k^{(i)} - [y^{(i)} = k]$.

Problem 2. **[1p]** How will the value of SoftMax function change, if we will add the same constant term to each element of a? Often, before computing SoftMax, the largest value can be subtracted to mitigate large exponents and associated numerical errors. Is it a good practice?

Are two-class SoftMax regression and logistic regression equivalent (do they yield the same outputs for the same inputs)?

Problem 3. **[2p]** Implement SoftMax regression and apply it to the Iris dataset. During training, use L-BFGS from `scipy.optimize`. You can initialize the algorithm with a null matrix (all entries being zeros). Obtained accuracy should be comparable with that of k-NN (roughly 3% of errors). If your model doesn't work, check the gradient using the `check_gradient` routine from Starter Code of Assignment 2, which computes the gradient numerically.

2 2-layer Neural Network

The task is to extend the SoftMax regression model to a 2-layer neural net. The network will transform an input vector to an activation vector of hidden neurons and finally, using the SoftMax function, to a vector of probabilities of the sample's belonging to one of 10 classes.

To train the network, we'll need the loss function J and its gradient with respect to network's parameters (weights and biases). For a 2-layer net, this can be achieved using the following relationships:

Data The training set has m samples of n dimensions, belonging to one of K classes, is given as a set of matrices: $X \in \mathbb{R}^{n \times m}$ and $Y \in \{1, 2, \ldots, K\}^{1 \times m}$.

Parameters The net will have 2 layers: 1) a hidden one, having L neurons, and 2) an output one, having K neurons (one for each of K classes). The layers are defined through:

1. parameters of the hidden layer, which maps n-dimensional input vectors into activations of L neurons: weight matrix $W^h \in \mathbb{R}^{L \times n}$ and bias vector $b^h \in \mathbb{R}^{L \times 1}$,
2. parameters of the output layer, which maps L-dimensional vector of activations of the hidden layer to K activations of output neurons: weight matrix $W^o \in K \times L$ and bias vector $b^o \in \mathbb{R}^{K \times 1}$.

Signal forward propagation (fprop) Each hidden neuron computes its total input as a sum of product of its inputs, weight matrix and bias. For an i-th sample, the total input ${a^h}_l^{(i)}$ of an I-th neuron is thus:

$$ {a^h}_l^{(i)} = \sum_{j=1}^{n} {W^h}_{l,j} x_j^{(i)} + {b^h}_l \tag{6} $$

The total input of neurons might also be expressed via matrices, using matrix multiplication and broadcasting (which allows to add a column vector to all column vectors of a matrix):

$$ a^h = W^h \cdot x + b^h \tag{7} $$

This can be implemented as:

```
ah = W.dot(x) + b           # In Python.
ah = bsxfun(@plus, W*x,b)   % In Matlab.
```

Next, we compute activation h^h of hidden neurons with hyperbolic tangent $\tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}}$:

$$ {h^h}_l^{(i)} = \tanh({a^h}_l^{(i)}) \tag{8} $$

Thanks to vectorization in Python, h^h might be computed with a single expression:

```
hh = numpy.tanh(ah)
```

Total input of the output layer can be computed using activations of the hidden layer (with the help of broadcasting) as:

$$a^o = W^o \cdot h^h + b^o \tag{9}$$

Finally, probabilities of a sample's belonging to particular classes have to be computed. This can be achieved with SoftMax:

$$p(y^{(i)} = k|x^{(i)}) = o_k^{(i)} = \frac{\exp({a^o}_k^{(i)})}{\sum_{k'=1}^{K} \exp({a^o}_{k'}^{(i)})}. \tag{10}$$

Like with SoftMax regression, we will use cross-entropy as the loss function:

$$\begin{aligned} J^{(i)}(\Theta) &= -\sum_{k=1}^{K} [y^{(i)} = k] \log o_k^{(i)}, \\ J(\Theta) &= \frac{1}{m} \sum_{i=1}^{m} J^{(i)}(\Theta) = -\frac{1}{m} \sum_{i=1}^{n} \sum_{k=1}^{K} [y^{(i)} = k] \log o_k^{(i)}. \end{aligned} \tag{11}$$

Error backpropagation (bprop) Using the chain rule one can derive the gradient of the loss function in respect to neurons' activations and network parameters.

First we compute the gradient with respect to the output layer's total inputs:

$$\frac{\partial J}{\partial {a^o}_k^{(i)}} = \frac{1}{m}(o_k^{(i)} - [y^{(i)} = k]), \tag{12}$$

then we compute the gradient with respect to activations of hidden units:

$$\frac{\partial J}{\partial {h^h}_l^{(i)}} = \sum_{k=1}^{K} \frac{\partial J}{\partial {a^o}_k^{(i)}} \frac{\partial {a^o}_k^{(i)}}{\partial {h^h}_l^{(i)}} = \sum_{k=1}^{K} \frac{\partial J}{\partial {a^o}_k^{(i)}} {W^o}_{kl}, \tag{13}$$

then we compute the gradient with respect to the total activations of hidden units:

$$\frac{\partial J}{\partial {a^h}_l^{(i)}} = \frac{\partial J}{\partial {h^h}_l^{(i)}} \frac{\partial {h^h}_l^{(i)}}{\partial {a^h}_l^{(i)}} = \frac{\partial J}{\partial {h^h}_l^{(i)}} (1 - ({h^h}_l^{(i)})^2), \tag{14}$$

where we have used the relationship $\frac{\partial \tanh(x)}{\partial x} = 1 - \tanh(x)^2$. Finally we can use the gradients with respect to the total inputs to compute the gradients with respect to network parameters, eg. for the input layer:

$$\frac{\partial J}{\partial {W^o}_{kl}} = \sum_i \frac{\partial J}{\partial {a^o}_k^{(i)}} \frac{\partial {a^o}_k^{(i)}}{\partial {W^o}_{kl}} = \sum_i \frac{\partial J}{\partial {a^o}_k^{(i)}} {h^h}_l^{(i)}, \tag{15}$$

$$\frac{\partial J}{\partial {b^o}_k} = \sum_i \frac{\partial J}{\partial {a^o}_k^{(i)}} \frac{\partial {a^o}_k^{(i)}}{\partial {b^o}_k} = \sum_i \frac{\partial J}{\partial {a^o}_k^{(i)}}. \tag{16}$$

Problem 4. [2p] Implement a 2-layer neural network as a function `TwoLayerNet(Θ,X,Y)`, which computes the loss and gradient of loss with respect to the weights and bias terms (encoded as Θ) on data given as X and Y. Refer to Starter Code for Assignment 4 for the details. Try to express as much as possible with matrix calculus.

The following problems will require to train the network. Use the L-BFGS optimizer from `scipy.optimize` to minimize your function (particularly: the loss) and find the right Θ.

Problem 5. **[1p]** Choosing the initial vector. In the case of linear and logistic regression, we could start the optimization with a vector of zeros. Such initialization will be troublesome for neural networks.

You can use the following initialization methods for network parameters: a) initialize weight matrices with small random numbers (eg. drawn from $\mathcal{N}(0, 0.2)$), b) initialize bias vectors with zeros. Train the network on the Iris dataset and report classification accuracy.

Problem 6. **[1p + 1p bonus]** Test your network on a 2-dimensional and a 3-dimensional XOR problem. How many hidden neurons the network requires to express the XOR function?

Bonus Plot samples in hidden neurons' activation space (similarly to http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/).

Problem 7. **[1p]** Normalize the Iris dataset, so that each of the 4 attributes would fall into [-1,1] interval. Train the network and check classification accuracy.

Now make a deliberate mistake in backpropagation (eg. change the sign somewhere in the expression). Try to train the network - how does it work, even though it's faulty?

Problem 8. **[1p]** Final questions:

- Are neural networks parametric or non-parametric models? Why is it so?
- What will happen, if for each layer (hidden and output) all weights will be initialized to the same values before the training?