Homework 2 (by December 12, 2017) – MLPs

Note: use only standard tensorflow components for the implementation of the homework, such as `tf.matmul`, `tf.maximum`, `tf.reduce_sum`, etc. Do not use specific neural networks packages, such as `tf.layers.dense`, `tf.contrib.learn.DNNClassifier`, `tf.nn.softmax`, except where specifically asked to.

1) continue to practice building and running tensorflow graphs

. . . also try using python loops, functions and (if you know how) classes to build graphs, in particular repetitive elements.

2) training data

Download the data file

`https://cvml.ist.ac.at/courses/DLWT_W17/data/hw2-train-data.npy`

(beware, 170MB!) and the labels file

`https://cvml.ist.ac.at/courses/DLWT_W17/data/hw2-train-labels.npy`

You can load these into python using numpy's `load` command. Each row of the data matrix contains one vector of dimension $d = 768$. The label vector contains the corresponding labels in $\{0, 1, \dots, 9\}$. In total, there are 57500 examples. Use the first $n = 50000$ for training and the rest as a validation set.

3) Multilayer Percepton (MLP)

Implement a multilayer perceptron with architecture:

- input size 768
- two hidden layers of sizes 100 and 200:
  - hidden layers should be *affine* function of all inputs $f(x) = Wx + b$, followed by *ReLU* activations, $\mathrm{ReLU}(x) = \max(0, x)$
- output layer with 10 outputs:
  - output layer should also be *affine*, and followed by a *softmax*

$$\mathrm{softmax}(x)[j] = \frac{\exp(x[j])}{\sum_{i=1}^{dim} \exp(x[i])}$$

Hint: to make dimensions match for the softmax, look up the `keep_dims` keyword of `tf.reduce_sum`.

4) cross-entropy loss function

Implement a *cross-entropy* loss function

$$\mathrm{loss}(f) = -\frac{1}{n} \sum_{i=1}^{n} \sum_{c=0}^{9} \delta(y_i = c) \log(g(x_i)[c]),$$

where $g(x)[0], ..., g(x)[9]$ are the network outputs, and a term that computes the *accuracy*

$$\mathrm{accuracy}(f) = \frac{1}{n} \sum_{i=1}^{n} \delta\big(y_i = \mathrm{argmax}_c(g(x_i)[c])\big)$$

Hint: in the loss, implement the δ-function using `tf.one_hot`. In the accuracy, use `tf.equal` to implement the δ-function. Beware that before averaging, you might use `tf.cast` to convert its boolean output to a floating point number.

5) train the network: batch

Train the network using ordinary gradient descent:

- implement a **batch gradient descent** using standard components, such as `tf.gradients` etc. ,
- apply it to the `loss` for 10 steps with learning rate $\eta = 0.1$
- after each step, print the `loss` on the training set as well as the `accuracy` both on the training and validation data
- (optional) can you train by directly minimizing `accuracy`? Why not?

Hint: if you get NaNs anywhere during training:

- make sure you avoid overflows when using `tf.exp` (e.g. search online for "softmax overflow")
- make sure you avoid computing `tf.log(0)`, even if it's multiplied with `0` afterwards (e.g. using `tf.clip_by_value` or `tf.maximum` to replace 0s by very small values)

6) train the network: stochastic

Train the network using stochastic gradient descent:

- implement **stochastic gradient descent** using mini-batches
- train the network by repeating for 10 *epochs*:
  - shuffle the training data randomly (e.g. using a numpy permutation)
  - train with mini-batches of size 25 and $\eta = 0.01$ until all training data has been used once
- after each such *epoch*, print the `loss` on the training set as well as the `accuracy` both on the training and validation data

Hint:

- you should achieve at least: loss ≤ 0.05, acc$_{train} \geq 99\%$, acc$_{val} \geq 95\%$)
- if your loss does not converge well, try initializing the weights with noise of smaller variance

7) model selection

Try different learning rates and batch sizes to find a combination that works even better than the above

8) optimization routines

Instead of your self-written optimizer, use some provided by tensorflow:

- `tf.train.GradientDescentOptimizer`
- `tf.train.AdamOptimizer`

9) (optional) architecture selection

Try different architectures (number of layers, number of neurons, activation functions, initialization of weights, batch sizes..) to find one that

1. achieves the best validation results, regardless of number of iterations, runtime etc.
2. reaches training accuracy above 99% as fast as possible (in terms of real runtime)
3. can be evaluated as fast as possible while keeping a validation accuracy of at least 95%

10) (optional) convenience functions

Reimplement the Multilayer Perceptron using tensorflow's convenience functions from `tf.layers` etc.

Hand-in requirements

1. upload your code with a reasonable learning rate to the IST *git* server
2. pick exactly **one** of your trained models and use it to predict labels for the data available at:

`https://cvml.ist.ac.at/courses/DLWT_W17/data/hw2-test-data.npy`

Hint: you will need to either store the network parameters, or perform the prediction in the same session that you used for training.

3. write the resulting class predictions to a file "hw2-test-labels.txt" (one number 0-9 per line in the same order as the test examples) and upload it. The contribution with the smallest number of errors will win a prize.