Bar-Ilan University

Undergraduate Project

Kernel-based Phoneme Distributed Recognizer

Felix Kreuk (306558040) & Yanai Elazar (305215980)

supervised by
Dr. Joseph Keshet

September 10, 2016

Contents

1 Overview

Currently the best results on phoneme classification and phoneme recognition are obtained using deep neural networks (DNNs). However, the combinatorial difficulty of performing exhaustive model selection in the discrete space of DNN architectures and the potential to get trapped in local minima are well recognized as valid concerns. DNNs also lack the theoretical motivation.

Kernel methods have theoretical appeal and grounding and found to be very successful in small scale applications. Nevertheless, kernel methods are often not the first choice for large-scale speech applications due to their significant memory requirements and computational expense. In recent years, randomized approximate feature maps have emerged as an elegant mechanism to scale-up kernel methods. Still, in practice, a large number of random features is required to obtain acceptable accuracy in predictive tasks. This project will show that kernel methods can match the performance of state of the art DNNs for phoneme classification and phoneme recognition.

2 Project Description

2.1 Learning Algorithm

The goal of our project was identifying phonemes from the TIMIT dataset, that is, given an unknown phoneme, classifying it as one of 39 possible phonemes (reduced set used instead of 44). Therefore, we had to use a classification algorithm for a multi-class problem. We have used the SVM algorithm since it solves a large-margin problem:

$$w_r = argmin_{w_r} \frac{1}{N} \sum_{i=1}^{N} max_{\widehat{y}}[\delta(\widehat{y} \neq y_i) - w^{y_i} \cdot x + w^{\widehat{y}} \cdot x] + \gamma ||\omega||^2 \tag{1}$$

The implementation of the SVM multi-class algorithm is done using stochastic-gradient-descent (a method of minimizing the loss function, specifically, in SGD we update the hypothesis after each instance of the training set):

Algorithm 1 Multi-class SVM

procedure MULTI-CLASS SVM
  for for t = 1...T **do**
    choose $(x_i, y_i) \in S$
    $y' = argmax_{y' \in \{1,..,k\}}[\delta(y' \neq y_i) + w^{y'} x_i]$
    if $[\delta(y' \neq y_i) - w^{y_i} x_i + w^{y'} x_i] > 0$ **then**
      $w^{y_i} = w^{y_i} - \eta_t \nabla_{w^{y_i}} = w^{y_i}(1 - \eta_t \lambda) + x_i \eta_t$
      $w^{y'} = w^{y'} - \eta_t \nabla_{w^{y_i}} = w^{y'}(1 - \eta_t \lambda) + x_i \eta_t$
      $w^r = w^r - \eta_t \nabla_{w^r} = w^r(1 - \eta_t \lambda)$ for $y \neq y_i, y'$
    end if
  end for
end procedure

And since it is easy to incorporate the use of kernel functions into it, Specifically an approximation of the RBF kernel as described in [**?**]. The RBF (Radial Basis Function)

kernel's feature space has an infinite amount of dimensions. This kernel is commonly used in SVM and is calculated by:

$$k(x, x') = exp\Big(- \frac{||x - x'||^2}{2\sigma^2} \Big) \tag{2}$$

But due to bad scaling with the size of the dataset this kernel is many times approximated to get better performance:

$$z(x)z(x') \approx \phi(x)\phi(x') = k(x, x') \tag{3}$$

2.2 Kernel Approximation

Kernel functions are used in machine learning to help solve problems otherwise not linearly-separable, this is achieved by mapping the original problem into a higher dimension (possibly infinite) where it will be linearly-separable.

The down-side of such functions is that they scale poorly with the training dataset. Meaning, for each x_i the inference rule is of time complexity $O(Nd)$ (N being the number of instances and d the dimension), the overall complexity is $(N^2 d)$, which is just not feasible on large training sets.

We use the kernel approximation suggested in [**?**]. The approximation maps the input explicitly to a low dimension (relative to original kernel), there we can solve the problem using standard linear SVM. By using this kernel the inference rule is much more efficient, with time complexity of $(D + d)$ (D being the new dimension and d is the old one). Thus, the overall complexity is $O(N(D + d))$. The mapping is given by a function $z(x) : R^d \to R^D$:

$$z(x) = \sqrt{\frac{2}{D}}[cos(\omega_1 x + b_1) + \ldots + cos(\omega_D x + b_D)]' \tag{4}$$

where w are D iid samples from a normal distribution $\omega_1, \ldots, \omega_D \in R^d$ and d are D iid samples from a uniform distribution $b_1, \ldots, b_D \in R$ on $[0, 2\pi]$.

Algorithm 2 Random Fourier Features

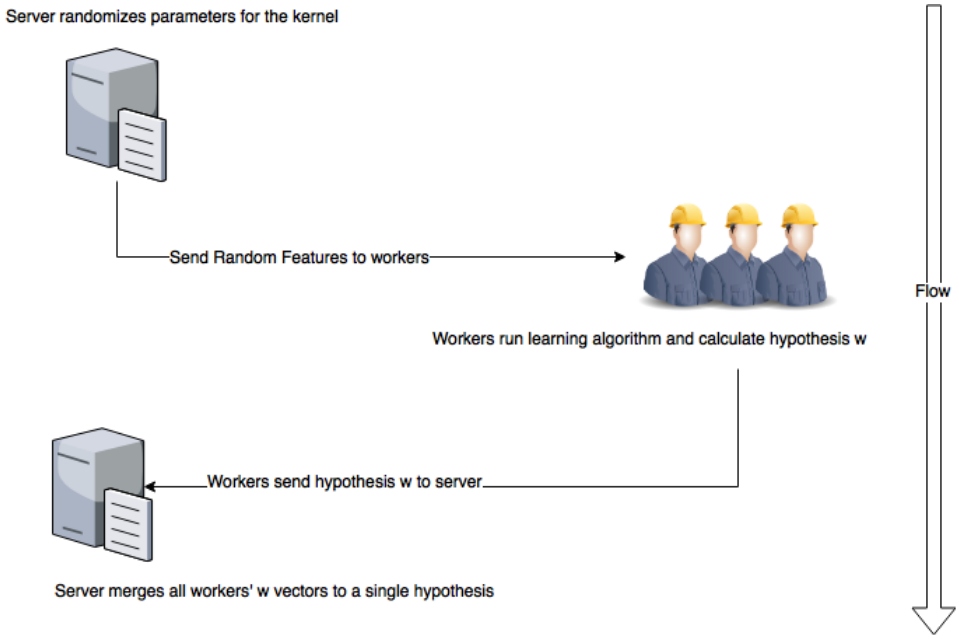
1: **procedure** Random Fourier Features
2: Compute the Fourier transform p of the kernel $k : p(w) = \frac{1}{2\pi} \int e^{-jw'\delta} k(\delta) d\Delta$
3: Draw D iid samples $w_1, w_2, ..., w_D \in R^d$ from p and D iid samples $b_1, b_2, ...b_D \in R$ from the uniform distribution on $[0, 2\pi]$
4: $Let z(x) = \sqrt{\frac{2}{D}}[cos(\omega_1 x + b_1) + \ldots + cos(\omega_D x + b_D)]'$
5: **end procedure**

2.3 Distributed Gradients

To further speed up the learning process we have implemented it in a distributed manner. The initial dataset is first divided into chunks, and spread across numerous worker machines. Each machine runs the kernel-based SVM algorithm using gradient descent. Finally, after the batch is finished, the weight vector learned ω is passed to a central

parameter server, which aggregates the vectors for all worker machines. The aggregation is done by:

$$\omega = \frac{1}{N} \sum_{i=1}^{k} \omega_i \tag{5}$$



Algorithm 3 Distributed Stochastic Gradient Descent

procedure WORKER
    load a part of the training set
    pull random features parameters from server
    run Multi-class SVM with SGD and get w hypothesis
    send w to server
end procedure
procedure SERVER
    randomize parameters for Random Fourier Features
    send parameters to workers
    get w_i from each i worker
    calculate $\omega = \frac{1}{N} \sum_{i=1}^{k} \omega_i$
end procedure

3 Implementation Details

Please read our GitHub documentation - it contains all classes descriptions along with running instructions and environment configuration.

4 Experiments

As seen in the chart above, there was a slight spike in accuracy (from 80% to 83%) when increasing amount of workers from 1 to 2. In terms of time we have seen a decrease when increasing amount of workers from 2 to 4 as time we nearly cut in half. When using 16 workers time slightly increased due to, what we believe, is overhead of managing multiple workers. The above chart was generated on the Adult dataset.

5 Problems & Solutions

5.1 Hyper Parameters

Problem - During the initial runs of our implementation we have encountered performance (in terms of accuracy) considerably different than the one presented in [**?**]. in order to solve it we have run the algorithm for numerous runs, in each run, finding the best hyper-parameters so far. Eventually we have converged to an accuracy rate with a deviation of only 2%.

5.2 Parallelization

The random features kernel presented in [**?**] obviously used random elements, this leads to no complications on one machine. But when dealing with numerous machines, the random features must match across all machines, only then will the algorithm work. The initial random elements (W matrix and b bias vector) were created once on the parameter server, transferred to all working machines to keep them synchronised.

5.3 Closed-form Solution

In the initials stages of the project we have implemented a close-form solution of SVM very similar to Rahimi's implementation in MATLAB. This worked as expected on one machine, but turned out to be impossible to parallelise. We have switched the implementation to the traditional iterative gradient-descent algorithm.

5.4 Normalisation

Different methods of normalising the dataset led to very different accuracy rates. We have researched the methods used in normalisation across several packages, and decided to use the method used in $scikit - learn$ random features algorithm (normalising to z-score according to columns), as it lead to good accuracy rates.

5.5 Algebraic c++ libraries

There are many libraries available for different algebraic operations (armadillo, eigen, opencv). We have researched several benchmarks testing the most widely-used algebra packages. Finally, we have decided to use *armadillo* due to its good performance in matrix multiplication in dimensions relevant to our problem.

5.6 Varying Types

Armadillo and open-mpi both use different types, armadillo uses *mat* type, while open-mpi uses *double* arrays. We have created wrapper function converting all types in both directions.

6 Struggles

- This project required the installation of several packages: armadillo, open-mpi, easy-logging, htk infrastructure, etc. All of the above required considerable time to setup, run and learn. Some of which had to be edited for our needs.
- Finding a solid IDE was a hassle, we started out with eclipse only to find that configuring it to work was a real pain. Later we have switched to CLion, which was a big improvement interface-wise and managed to link our external libraries well, but had its own faults, as many these libraries were badly recognised by CLion's syntax colouring engine.
- Working on the same project on the same repository, often on different computers. We used Git as our VCS environment. Like anything else, it required a learning curve, which we learned a lot from it. from local commit, 'push' to the repository and merging code whenever there is a conflict, which we had, quite a lot.
- Relative low language to work on - C++. As this is relatively a low level language, and quit an old one, everything is hard. There are not a lot of built in utilities, the debugging tools are very advanced and more... Therefor, all the coding process takes much more time than high level languages. On the other hand, as it is much closer to the computer core, we handle all it's memory, so the running time is faster.

7 Future Thoughts

- Big Data - some datasets may not fit in memory, thus normalising the dataset as a whole will not be possible. In the future it should support the division of the dataset into chunks and normalising it in a distributed manner.

- Parameter tunning. As for today, as time enabled us, we made the parameter tuning on the dataset we tested on, and on one local machine. In the future, one should add the possibility to make it distributed. All the tools are ready, just need to add some functionallity to manage it correctly.