# Kernel Multi-task Learning using Task-specific Features

**Edwin V. Bonilla, Felix V. Agakov, Christopher K. I. Williams**
School of Informatics, University of Edinburgh, Edinburgh EH1 2QL, UK
edwin.bonilla@ed.ac.uk, felixa@inf.ed.ac.uk, c.k.i.williams@ed.ac.uk

## Abstract

In this paper we are concerned with multi-task learning when task-specific features are available. We describe two ways of achieving this using Gaussian process predictors: in the first method, the data from all tasks is combined into one dataset, making use of the task-specific features. In the second method we train specific predictors for each reference task, and then combine their predictions using a gating network. We demonstrate these methods on a compiler performance prediction problem, where a task is defined as predicting the speed-up obtained when applying a sequence of code transformations to a given program.

## 1 INTRODUCTION

In this paper we are concerned with multi-task learning when task-specific features are available. Thus we consider $M$ tasks and for each we wish to learn the mapping $g_i(\boldsymbol{x})$ for $i = 1, \ldots, M$, where $\boldsymbol{x}$ is a vector of input features. For each task $i$ we also have a task descriptor (or task-specific feature vector) $\boldsymbol{f}_i$. Thus we can also write $g_i(\boldsymbol{x}) = g(\boldsymbol{f}_i, \boldsymbol{x})$ for some function $g$. This problem was considered by Bakker and Heskes (2003) using neural network predictors. In this paper we discuss how to address this problem using kernel machines (specifically, Gaussian process predictors), where we can directly model correlations between tasks.

In our experiments we will focus on the problem of compiler performance prediction. There are a large number of possible code transformations that maintain the correctness of the program but which can affect its runtime, e.g. loop unrolling, common subexpression elimination. These transformations can be combined into sequences yielding a very large space to explore. In general, we will have a suite of programs (or benchmarks) which we wish to evaluate. We will extract task-specific features from each program, and thus exploit the multi-task paradigm with task-specific features. Our goal will be to have an accurate predictive model of the speed-up under transformation $\boldsymbol{x}$ for a new task (program) based on very few runs on this new task, due to *transference* across tasks.

Although we will focus on the specific example of compiler performance prediction, we note that multi-task learning is a generic problem. For example, it arises in collaborative filtering, multi-level modelling in statistics, text categorization with multiple topics, and speaker-dependent speech recognition, etc.

The structure of this paper is as follows: In section 2 we describe our methods for transfer learning, and discuss related work. Section 3 covers the details of the compiler setup, and the extraction of input and task-specific features. In section 4 we describe the experimental framework, and the results are presented in section 5. We conclude with a discussion.

## 2 THEORY

We give two methods for using kernel machines to perform the multi-task problem. In the first we simply concatenate the input features $\boldsymbol{x}$ and the task-specific features $\boldsymbol{f}$ to give a vector $\boldsymbol{z}$ (so $\boldsymbol{z}^T = (\boldsymbol{x}^T, \boldsymbol{f}^T)$). Now all the training data from the individual tasks can be combined into one large training set, and predictions made using this machine. We call this the *combined* method.

If the kernel function $k(\boldsymbol{z}, \boldsymbol{z}')$ decomposes as $k(\boldsymbol{z}, \boldsymbol{z}') = k_x(\boldsymbol{x}, \boldsymbol{x}')k_f(\boldsymbol{f}, \boldsymbol{f}')$ then there is a decomposition of the problem into learning task similarity (as measured by $k_f$) and input similarity (as measured by $k_x$). The commonly used Gaussian (or squared exponential) kernel is an example where such a decomposition occurs. Note the similarity of this combined approach to co-

kriging as used in geostatistics (see e.g. Cressie (1993, sec. 3.2.3)); the use of task-specific features is one way to define a kernel which incorporates correlations between tasks.

Our second approach is based on training a separate kernel predictor for each task, and then combining these using a gating network. Specifically, let the prediction for input $\boldsymbol{x}$ on the $i$th task be denoted $p_i(s|\boldsymbol{x})$ (where $s$ is the prediction for the target variable). Then for a new task with task descriptor $\boldsymbol{f}$, we have

$$p(s|\boldsymbol{f}, \boldsymbol{x}) = \sum_{i=1}^{M} p(i|\boldsymbol{f}, \boldsymbol{x}) p_i(s|\boldsymbol{x}), \tag{1}$$

where $p(i|\boldsymbol{f}, \boldsymbol{x})$ is a gating network outputting mixing proportions that sum to 1. We call this the *gating* method. It is a specific kind of mixture of experts architecture (Jacobs et al., 1991) where the individual experts are trained independently. Details of the training of the gating network are given in section 4.1.

One difficulty with neural network models for multi-task learning (Bakker and Heskes, 2003) is that neural networks can be quite tricky to train, due to local optima in weight space, choices for the number of hidden units, etc. In contrast, Gaussian process regression (and many other kernel prediction methods) are underpinned by convex optimization problems (given the kernel). This is our motivation to produce a kernel-based solution to the problem.

### 2.1 RELATED WORK

There has been a lot of work over in recent years on multi-task learning (or inductive transfer), see e.g. Caruana (1997); Thrun (1996) for early references. A common setup is that there are multiple, related supervised learning problems and the goal is to avoid tabula rasa learning for a new problem by extracting information from the problems seen before. We particularly focus on the case when task-specific features are available. As mentioned above, this case is discussed in Bakker and Heskes (2003), but using neural network predictors. They propose two ways to use the task-specific features: One is to define a task-specific prior in weight space (section 4.1 in their paper). The second is to use a gating network (although in their case this only depended on $\boldsymbol{f}$ and not on $\boldsymbol{x}$). Note that neither of these methods introduces inter-task correlations in the prior. Yu et al. (2007) have recently investigated the combined method discussed above under the assumption of factorization of the kernel wrt $\boldsymbol{x}$ and $\boldsymbol{f}$ features. In their case the setup was as a relational model, e.g. for predicting movie ratings based on user and movie features.

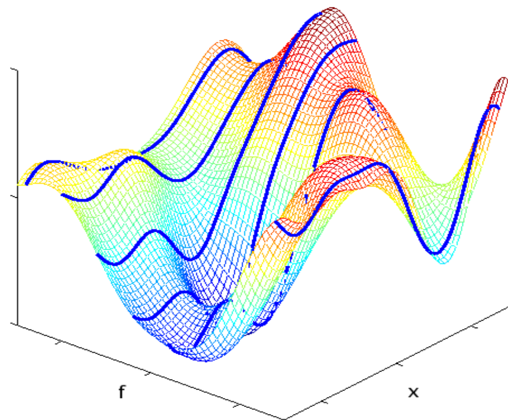


Figure 1: A schematic illustration of drawing sample functions in joint $\boldsymbol{x}$ and $\boldsymbol{f}$ space. For fixed values of $\boldsymbol{f}$ we obtain sample functions (bold lines) as functions of $\boldsymbol{x}$. Note that these sample functions for different values of $\boldsymbol{f}$ are correlated; this is in contrast to *independent* draws over sample functions if the $\boldsymbol{f}$-dimension is omitted.

There is also a lot of work on multi-task learning when there are no task-specific features. In this case one can make various assumptions about how to induce transfer between tasks. For example Minka and Picard (1999) assumed that a number of related tasks shared the same kernel parameters, and these were optimized on the set of tasks available. In a similar vein, Yu et al. (2005) induced transfer between tasks by assuming a common covariance for the tasks, with a Normal-Inverse-Wishart prior. However, note that in these cases the different tasks are conditionally independent given the kernel; in contrast our methods discussed above are stronger in that they directly induce correlations between the tasks. This is illustrated in Figure 1, where both $\boldsymbol{x}$ and $\boldsymbol{f}$ are shown schematically as one-dimensional variables. For fixed values of $\boldsymbol{f}$ we obtain sample functions (bold lines) as functions of $\boldsymbol{x}$. Note that these sample functions for different values of $\boldsymbol{f}$ are correlated; this is in contrast to *independent* draws over sample functions if the $\boldsymbol{f}$-dimension is omitted.

There are also other assumptions one can make about ways to share information between tasks; for example, one can consider mixture models for task clustering (Bakker and Heskes, 2003). Evgeniou et al. (2005) consider methods for inducing correlations between tasks based on a correlated prior over linear regression parameters; this is in fact a special case of co-kriging. Multi-task modelling is also very prevalent in statistics, where it goes under the names of multilevel modelling or hierarchical modelling, see e.g. Goldstein

(2003).

# 3 THE COMPILER PERFORMANCE PREDICTION PROBLEM

Our measure of performance is the *speed-up* of a program under a given sequence of compiler transformations. This is obtained by computing the ratio of the execution time of the original program (baseline) over the execution time of the transformed program. Therefore, a speed-up greater than one is an improvement in performance and a speed-up between zero and one indicates that the transformation sequence applied harmed the performance of the original program by increasing its execution time. The performance prediction task is to predict the speed-up of a given program under transformation $\boldsymbol{x}$.

In the remainder of this section we describe the programs, optimizations and processor architecture used in our experiments, the input and task-specific features used, and related compilers work.

## 3.1 PROGRAMS, OPTIMIZATIONS AND ARCHITECTURE

Eleven different **benchmarks** from the UTDSP suite (Lee, 1997) have been used for the experiments. This set of C programs contains small kernels and larger applications. These are regarded as compute-intensive programs by the DSP community, and are continuously used in stream-processing applications. We have considered source-to-source **transformations** applicable to C programs by using the restructuring compiler framework SUIF 1 (Hall et al., 1996). Using these transformations, we have investigated a space composed by 13 transformations (selected by compiler experts) combined into sequences of up to length 5 including loop unrolling with unroll factors $1, 2, 3$ and 4. We have not considered sequences that include repeated transformations, and also restricted loop unrolling to be applied only once, generating a total of 88214 transformation sequences, which we have exhaustively enumerated. It takes around 3 days to run one benchmark over all these sequences.

The experiments were executed on a **Texas Instruments C6713 board**, a high end floating point DSP running at 225MHz with 256kB of internal memory. The programs were compiled using the Texas Instruments' Code Composer Studio Tools Version 2.21 with the highest -O3 optimization level. Table 1 shows some statistics of the speed-ups obtained with the experiments described above for each benchmark. These results are important as they show that good improvements were obtained with the experiments and that the data presents opportunities for learning. Further, the speed-ups obtained are very encouraging in the compiler community, as the compiler used on this board is believed to produce high quality code for these types of applications. Finally, it is important to remark that there are transformation sequences that significantly degrade performance (see for example *lmsfir* and *histogram*). This motivates building a proxy that can accurately predict performance speed-ups.

Table 1: Speed-ups statistics for the benchmarks used: minimum, maximum, mean and standard deviation. Those above the horizontal line are kernels, below applications.

| Program | Min | Max | Mean | Std |
|---|---|---|---|---|
| fft | 0.98 | 1.04 | 1.00 | 0.01 |
| fir | 0.84 | 1.84 | 1.31 | 0.31 |
| iir | 1.00 | 1.19 | 1.14 | 0.07 |
| latnrm | 0.80 | 1.00 | 0.93 | 0.06 |
| lmsfir | 0.29 | 1.00 | 0.72 | 0.21 |
| adpcm | 0.77 | 1.32 | 1.03 | 0.12 |
| compress | 1.00 | 1.64 | 1.28 | 0.25 |
| edge | 1.00 | 1.05 | 1.05 | 0.01 |
| histogram | 0.55 | 1.00 | 0.85 | 0.19 |
| lpc | 0.94 | 1.12 | 1.00 | 0.04 |
| spectral | 0.96 | 1.08 | 1.00 | 0.02 |

## 3.2 INPUT FEATURES

We use two different codings of the sequence of transformations into the $\boldsymbol{x}$-vector of features: a code-features representation (C) and a transformation-based representation (T). The code-features representation is obtained by computing features believed by compiler experts to be informative about program performance. These are based on three distinct metrics: *code size*, the total number of *instructions executed* and the *parallelism* existing among those instructions. For each of these metrics, a vector of high-level machine-independent instructions is derived by using the SUIF compiler infrastructure (Hall et al., 1996). Our high-level instructions correspond to the ones used by the SUIF Intermediate Representation (IR). A total of 83 features were extracted per program. These were reduced to 18 features using PCA, retaining 95% of the variance. One advantage of the code-features representation is that these features can be extracted even if new code transformations are applied that have not been seen before.

In the training data we consider sequences of up to length $L = 5$ using $T = 13$ possible transformations. One possible representation of transformation

sequences is to use a $L \times T$-dimensional binary vector with bits set for the appropriate transformations applied. However, given the constraints imposed in our dataset, this will be a very sparse representation of the input that may require a large amount of data for training. Therefore, we have preferred a bag-of-characters representation of length $T$ where we simply record if each transformation occurs in the sequence or not. Clearly, this throws away the ordering information within a sequence, and one might lose predictive performance due to this loss of information.

### 3.3 TASK-SPECIFIC FEATURES

We define task-specific features by selecting a small number of sequences and recording the corresponding speed-ups on the given task. We will refer to this set of sequences as *canonical sequences* and to their corresponding speed-ups as *canonical responses*. We select the set of canonical transformation sequences using the technique of *principal variables* (McCabe, 1984).

Here (as in PCA) the dimensionality reduction problem is formulated as the linear mapping from a high-dimensional vector (the speedups for all sequences) to a lower dimensional one. However in this case the linear mapping simply copies some of the variables and discards the rest, hence the term "principal variables". McCabe considers a number of different criteria for selecting a subset of variables. Here we choose the set of included variables $S_{(1)}$ so as to maximize $|\Sigma_{S_{(1)}}|$. This may be interpreted as maximization of a Gaussian approximation to the mutual information $I(S_{(1)}; \mathcal{T})$ which the retained variables $S_{(1)}$ contain about the identity of the task $\mathcal{T}$. (An alternative criterion is to minimize $\mathrm{tr}(\Sigma_{S_{(2)}|S_{(1)}})$ where $S_{(2)}$ denotes the set of variables discarded.) As searching for the optimal partition is computationally expensive, we follow the suggestion in McCabe (1984) and use a greedy forward selection strategy to select the subset.

As specified above, the canonical responses would be extracted using all 88214 sequences on each of the training problems. However, we have found that using canonical responses extracted from only 2048 randomly selected sequences yields almost as good performance as extraction from the larger set. In our experiments we use 8 canonical variables.

The response-based approach is not the only one that we can consider for defining task-specific features. For instance we might describe each untransformed program with code features (see section 3.2). However, experimentally we have found that the response-based method is superior.

### 3.4 RELATED COMPILERS WORK

Although performance predictors have been previously proposed in the compilers literature, they are generally constructed to be program-specific and do not generalize across different benchmarks. (see e.g. Triantafyllis et al. (2005) as a recent reference.) Further, even if they can potentially be used for predicting performance across programs, they rely heavily on expert-knowledge of specific architectures (such as models of cache behaviour) that are difficult to adapt to increasingly complex architectures (see e.g. the work in Karkhanis and Smith (2004)).

Recently, Bonilla et al. (2006) have proposed the construction of predictive search distributions that facilitate the transfer of knowledge across different problem instances. They have applied this framework to the problem of compiler optimization by modelling good optimization sequences, i.e. sequences that lead good speed-ups. However, they only used code features from the untransformed program to define program features. In this paper we are interested in the more general task of modelling the actual performance speed-ups, although our models can also be used for finding transformation sequences that improve performance.

## 4 EXPERIMENTS

Below we present results for both the combined and gating methods, using either code features (C) or transformations (T) as the representation for $\boldsymbol{x}$.

We have used leave one out *cross-validation* (LOO-CV) for evaluating the performance of our models, so for each LOO-CV experiment there are $M = 11 - 1 = 10$ reference tasks. Obviously it would be impractical to have a system that relies on exhaustive training data for making predictions. Therefore, we have sampled the space of each benchmark and investigated different sample sizes; below we report results for $n_r = 256$ training points per benchmark.

Note also that in this set up the canonical sequences were obtained in the LOO-CV framework, so that for a given test task, the canonicals were extracted from only the 10 reference tasks.

The measure of performance we have used is the mean absolute error (MAE) computed over all 88214 examples in the test problem.

In addition to the data from the reference problems, we also consider access to varying amounts of data from the test problem. These points are chosen according to the ordered list of the canonical sequences. A minimum of 8 canonicals are required so as to define the

task-specific features, but more can also be used, as described in section 4.1 below. We have considered sizes of $n_{te} = 8,\ 16,\ 32,\ 64$ and 128 points from the test problem. As our goal is to evaluate inter-task transfer, we must also consider what performance can be obtained using only these small amounts of test data. To do this we use GP regression, and also consider a simple baseline predictor based on the median[1] of the speed-ups on the $n_{te}$ canonical sequences. However, as the choice of canonical sequences depends on the reference problems, we also compare with the median speed-up of all the sequences on each test problem.

### 4.1 REGRESSION MODELS

We have used Gaussian process regression (see, e.g. Rasmussen and Williams (2006)) to model the speed-ups, i.e. $s(\boldsymbol{v}) \sim \mathcal{GP}(\mathbf{0}, C(\boldsymbol{v}, \boldsymbol{v}'))$, where $\boldsymbol{v}$ denotes the input representation used for each scenario. A squared exponential covariance function with Automatic Relevance Determination (ARD) and a noise term was used:

$$k(\boldsymbol{v}_p, \boldsymbol{v}_q) = \sigma_s^2 \exp\left(-\frac{1}{2}\sum_i \frac{(v_i^p - v_i^q)^2}{l_i^2}\right) + \sigma_n^2 \delta_{pq}, \quad (2)$$

where $\delta_{pq}$ is the Kronecker delta and the hyperparameters, $\sigma_s^2$, $\sigma_n^2$ and the length-scales $l_i$ have been learnt by maximizing the marginal likelihood of the data from the reference problems and the $n_{te}$ test points. Prediction with linear regression models was also tried, but this gave inferior performance to the GPs.

For the "no transfer" case using only $n_{te}$ points from the test problem there is a danger that the full ARD model would involve too many parameters and thus be in danger of overfitting. In this case for $n_{te} \leq 64$, an isotropic covariance function (with all $l_i$'s tied) was used instead of the ARD model. Experiments showed that the performance of the isotropic-no-transfer GP predictor was better than or equal to the ARD-no-transfer model.

For the gating network we set

$$p(i|\boldsymbol{x}, \boldsymbol{f}) \propto \exp\left\{-(\alpha_x \|\boldsymbol{\mu}_i - \boldsymbol{x}\|^2 + \alpha_f \|\boldsymbol{f} - \boldsymbol{f}_i\|^2\right\}, \quad (3)$$

where $\boldsymbol{\mu}_i \in \mathbb{R}^{|\boldsymbol{x}|}$, $\alpha_x > 0$ and $\alpha_f > 0$ are parameters, and $\boldsymbol{f}_i$ is the task-specific feature vector for task $i$. Normalization is obtained by summing over all reference tasks. (Alternatively one could use a softmax network with input vector $\boldsymbol{z}$.)

The gating network is trained to maximize the conditional likelihood of the training data (eq. 1) using gradient-based methods. In addition to the data from the $M$ reference datasets, we also include the speed-ups obtained on the canonical sequences of the new task to train this network; here we have given the new task equal weight to the other tasks, even though there is less training data, so as to emphasize the importance of the task-specific data.

In contrast to the conditional likelihood training of mixture-of-experts models (Jacobs et al., 1991; Rasmussen and Ghahramani, 2002), our assumption is that the experts $p_i(s|\boldsymbol{x})$ have been trained separately from the gating network $p(i|\boldsymbol{x}, \boldsymbol{f})$. The intuition is that by learning the parameters of the gating network we can utilize the information about previously solved related problems for solving new tasks, even when the training data for the new task might be quite limited. Note that our approach is also attractive computationally, as it allows us to quickly mix heterogeneous experts without a need of expensive approximations.

## 5 RESULTS

Figure 2 shows results for the case when $n_{te} = 8$, i.e. we use a very small amount of data from the test problem. The four methods T-combined, C-combined, T-gating and C-gating are shown, along with the two baselines median (of the 8 canonical responses) and median (of all test data). Note that there are error bars on the four transfer methods due to the random selection of the $n_r = 256$ training points from each of the reference problems; 10 repetitions were used to assess this variability. The median of the test data gives the best possible MAE for a given test problem without looking at the $\boldsymbol{x}$ data; it defines a reference point but requires all 88214 speed-ups to compute it, rather than the 8 which are available to the transfer methods (and median canonicals).

In Figure 2 we see the general trend that those problems with higher variability (as reflected in the standard deviation column in Table 1) generally have a larger error. T-gating generally performs worse than T-combined. For the code features representation, we observe that C-gating generally performs better than C-combined. The best performing transfer method on average is T-combined. This gives some significant improvements over the median predictors, particularly on problems compress, fir, histogram, latnrm, and lmsfir, and gives similar performance to the medians for the other problems. Its average MAE performance is 0.0576 compared to 0.1162 for the average of the median canonicals. Note that Bakker and Heskes (2003) only present results which are aggregated over all tasks, rather than a more detailed decomposition like the one given here.

[1] The median is the optimal value to minimize mean absolute error; the mean is optimal for mean squared error.

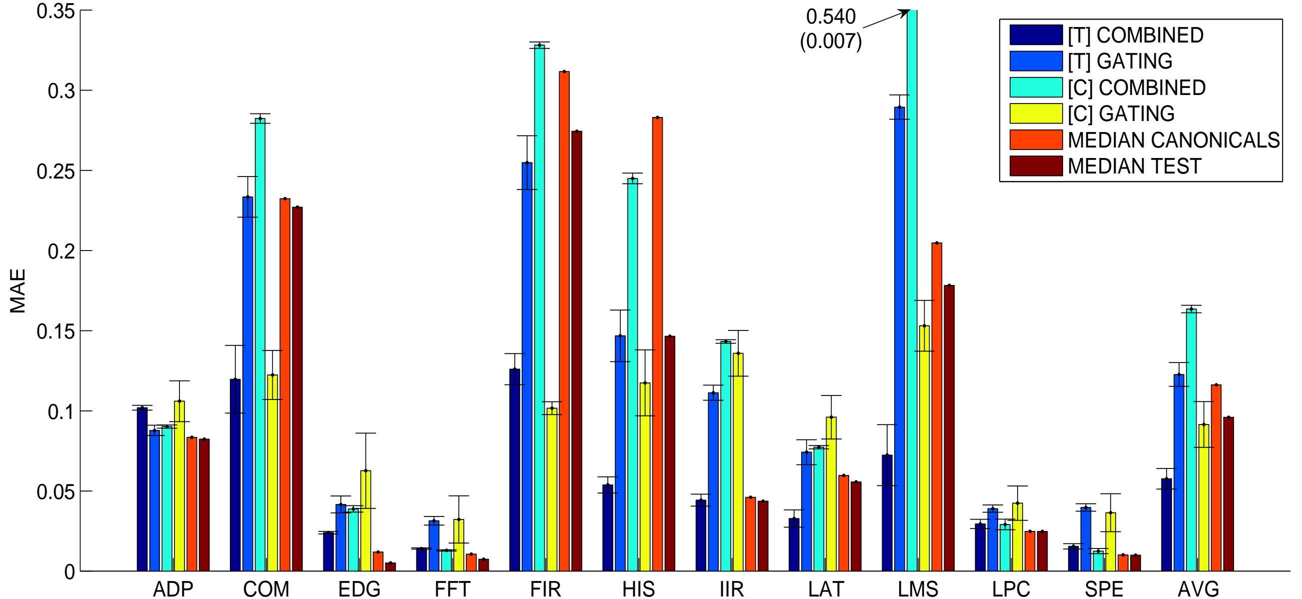


Figure 2: Mean absolute error (MAE) plotted on each of the 11 problems and the average, for the 4 methods T-combined, C-combined, T-gating and C-gating and the two baselines median (canonical) and median (of all test data). The error bars denote one standard deviation. The error bars for the averages are computed as the average of the standard deviations over all 11 problems. Recall that C denotes the code-features representation and T the transformations representation. For C-combined on LMS the MAE is 0.54, with standard deviation of 0.007.

Figure 3 shows in more detail the performance of various methods as a function of $n_{te}$. It shows the performance of the T-combined and two T-no-transfer methods, along with the median (canonicals) baseline for each of the 11 problems; the bottom right-hand panel shows the averages. The T-no-transfer-canonicals method uses a GP predictor trained on only the $n_{te}$ canonical sequences; consequently there are no error bars for these curves. In contrast T-no-transfer-random is trained on a set randomly selected datapoints of size $n_{te}$. Generally the performance of T-no-transfer-canonicals is superior to T-no-transfer-random. The plots in Figure 3 reinforce the message of Figure 2. They also show that for those problems where transfer is significant, this advantage tends to disappear when $n_{te}$ reaches higher values of around 128. However, note that for the compilers application it is desirable to make as few runs as possible on the test problem, so it is definitely the small $n_{te}$ values that are most relevant in this case. Thus we can conclude that for the T-combined method transfer learning generally either improves performance or leaves it about the same in comparison to the T-no-transfer-canonicals method.

One reason why the gating network approach may be limited is that as the component predictors are trained on the individual reference tasks, it may not be expected to generalize well if the pattern of speed-ups on the test problem is very different from those of the reference problems. This might be overcome by joint training of the component predictors and the gating network, as in the mixture of experts architecture.

It is possible to quantify the amount of inter-task transfer that is taking place when making predictions for the T-combined setup. For a Gaussian process predictor the mean prediction for a test point $\boldsymbol{z}_*$ is given by $\bar{s}(\boldsymbol{z}_*) = \boldsymbol{h}^T(\boldsymbol{z}_*)\boldsymbol{s}$, where $\boldsymbol{s}$ is the vector of speedups on the training tasks and $\boldsymbol{h}(\boldsymbol{z}_*)$ is the weight function (see e.g. Rasmussen and Williams 2006, sec. 2.6). If we order the training points according to the task they belong to, then $\boldsymbol{h}$ can be partitioned as $\boldsymbol{h}^T(\boldsymbol{z}) = (h_1^1, \ldots, h_{n_r}^1, \ldots, h_1^M, \ldots, h_{n_r}^M, h_1^{M+1}, \ldots, h_{n_{te}}^{M+1})$ and similarly for $\boldsymbol{s}$, where the superscript identifies the task, and task $M+1$ is the test task, for which we have only $n_{te}$ points. We can now measure the contribution of task $i$ for test point $\boldsymbol{z}_*$ by computing $r^i(\boldsymbol{z}_*) = |\boldsymbol{h}^i(\boldsymbol{z}_*)|/|\boldsymbol{h}(\boldsymbol{z}_*)|$. Averaging $r^i(\boldsymbol{z}_*)$ over test points gives a summary measure $\bar{r}^i$ for the contribution of task $i$ to the test problem. We prefer this measure as compared to looking directly at $K_f$ (whose entries are $[K_f]_{ij} = k_f(\boldsymbol{f}_i, \boldsymbol{f}_j)$), as the interpretation of $K_f$ is complicated by the inversion of the kernel matrix in GP prediction. Figure 4 shows the $\bar{r}$ values for each of the test tasks, for

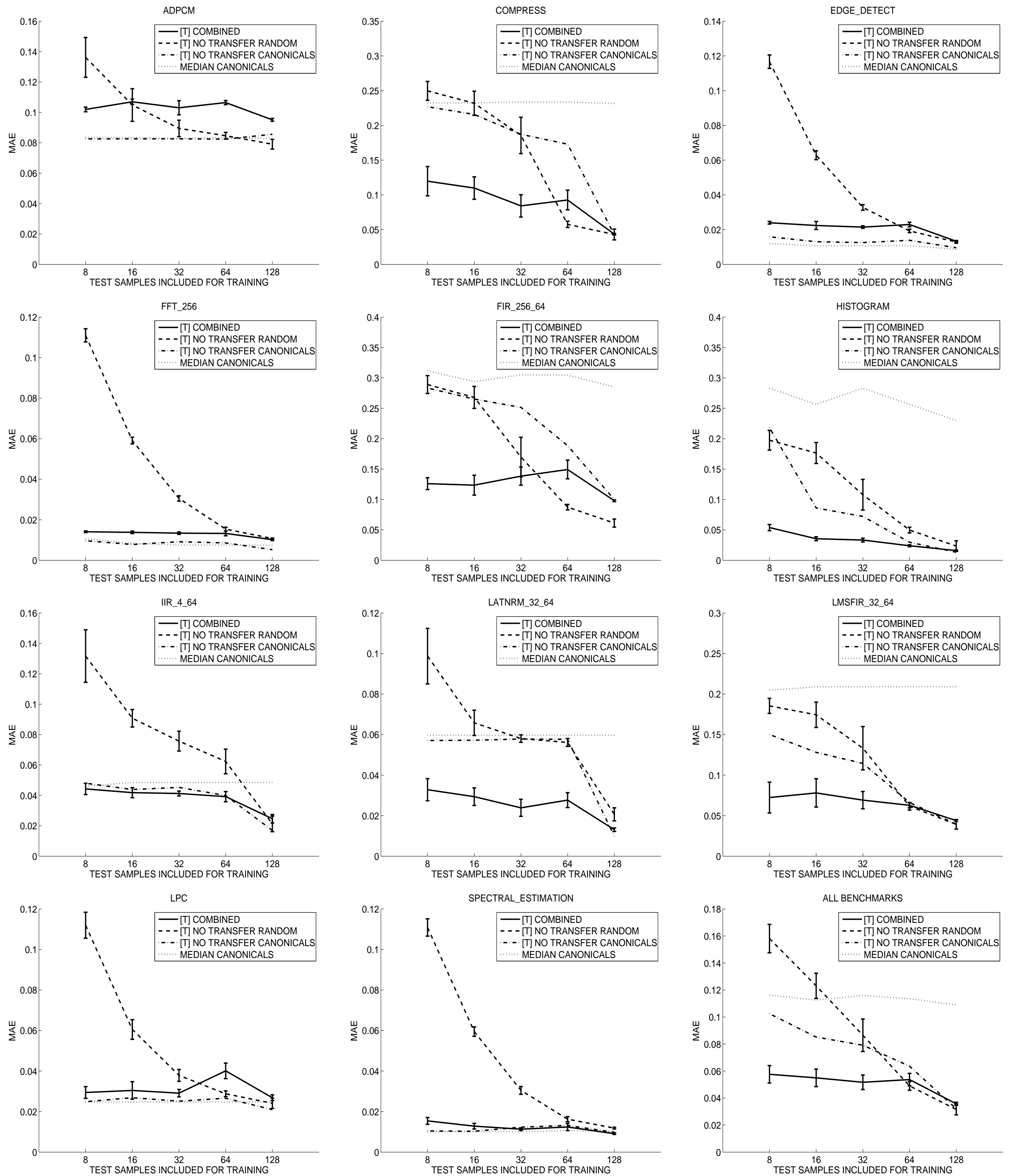


Figure 3: Plots showing the performance of the T-combined and T-no-transfer methods and median (canonicals) as a function of $n_{te}$ on all 11 problems. The bottom right panel shows the average performances. The error bars denote one standard deviation.

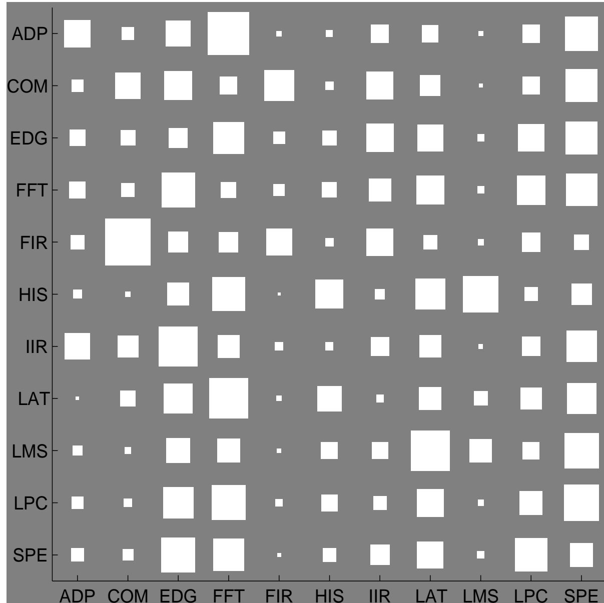


Figure 4: Hinton diagram indicating the $\bar{r}$ values for each task. Each row corresponds to the $\bar{r}$ values for the test tasks labelled on the left. The order of the tasks is *adp*, *com*, *edg*, *fft*, *fir*, *his*, *iir*, *lat*, *lms*, *lpc*, *spe*.

$n_r = 256$ and $n_{te} = 8$. We see that the contribution from the test benchmarks is in general significant for making predictions, considering that we have used only $n_{te} = 8$ test points. However, predictions are also helped by the contribution of other tasks. For example, predictions on *fir* rely heavily on training data from benchmark *compress*. Similarly, predictions on all benchmarks except *histogram* are only weakly related to training data from benchmark *lmsfir*.

## 6 DISCUSSION

In this paper we have introduced two approaches for using kernel machines for transfer learning with task-specific features, the combined and gating methods. We believe that it is useful to use kernel methods instead of neural networks due to the difficulties associated in training neural networks.

We have evaluated the performance of these methods on a compiler performance prediction problem, using two kinds of $\boldsymbol{x}$-representation. The results have shown the T-combined method to be the most effective, and that it outperforms the same method without transference. We are currently investigating if the method can be improved by incorporating ideas such as task-specific noise levels, or more radically by considering combined covariance functions that do not factor as the product of $k_f$ and $k_x$.


### Acknowledgements

We thank Christophe Dubach and John Cavazos for help in defining and extracting the code features, and Kai Yu for helpful discussions. This work is supported under EPSRC grant GR/S71118/01 and in part by the IST Programme of the European Community, under the PASCAL Network of Excellence, IST-2002-506778. This publication only reflects the authors' views.


## References


Bakker, B. and Heskes, T. (2003). Task Clustering and Gating for Bayesian Multitask Learning. *Journal of Machine Learning Research*, 4:83–99.

Bonilla, E. V., Williams, C. K., Agakov, F., Cavazos, J., Thomson, J., and O'Boyle, M. F. P. (2006). Predictive Search Distributions. In *ICML*.

Caruana, R. (1997). Multitask Learning. *Machine Learning*, 28(1):41–75.

Cressie, N. A. C. (1993). *Statistics for Spatial Data*. Wiley, New York.

Evgeniou, T., Micchelli, C. A., and Pontil, M. (2005). Learning Multiple Tasks with Kernel Methods. *Journal of Machine Learning Research*, 6:615–637.

Goldstein, H. (2003). *Multilevel Statistical Models*. Hodder Arnold.

Hall, M. W., Anderson, J.-A. M., Amarasinghe, S. P., Murphy, B. R., Liao, S.-W., Bugnion, E., and Lam, M. S. (1996). Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12).

Jacobs, R. A., Jordan, M. I., Nowlan, S. J., and Hinton, G. E. (1991). Adaptive Mixtures of Local Experts. *Neural Computation*, 3(1).

Karkhanis, T. S. and Smith, J. E. (2004). A first-order superscalar processor model. In *ISCA*.

Lee, C. (1997). UTDSP benchmark suite. `http://www.eecg.toronto.edu/~corinna/`.

McCabe, G. P. (1984). Principal variables. *Technometrics*, 26(2):137–144.

Minka, T. P. and Picard, R. W. (1999). Learning How to Learn is Learning With Point Sets. `http://research.microsoft.com/~minka/papers/point-sets.html`.

Rasmussen, C. E. and Ghahramani, Z. (2002). Infinite Mixtures of Gaussian Process Experts. In *Advances in Neural Information Processing Systems 14*.

Rasmussen, C. E. and Williams, C. K. I. (2006). *Gaussian Processes for Machine Learning*. MIT Press.

Thrun, S. (1996). Is Learning the $n$-th Thing Any Easier Than Learning the First? In *NIPS 8*.

Triantafyllis, S., Vachharajani, M., and August, D. I. (2005). Compiler optimization-space exploration. *The Journal of Instruction-Level Parallelism*, 7.

Yu, K., Chu, W., Yu, S., Tresp, V., and Xu, Z. (2007). Stochastic Relational Models for Discriminative Link Prediction. In *Advances in Neural Information Processing Systems 19*.

Yu, K., Tresp, V., and Schwaighofer, A. (2005). Learning Gaussian Processes from Multiple Tasks. In *ICML*.