A Additional Details on Modeling

In this section, we provide additional details on our modeling techniques.

A.1 Soft Nearest Neighbor Loss

The soft nearest neighbor loss applied to the learned encodings z_j^i looks like:

$$l_{snn} = -\frac{1}{N} \sum_{i,j} \log \left(\frac{\sum_{k \neq i} e^{-\frac{||z_j^i - z_j^k||^2}{T}}}{\sum_{\substack{k,l \\ (k,l) \neq (i,j)}} e^{-\frac{||z_j^i - z_l^k||^2}{T}}} \right)$$

where T is a temperature parameter.

In our experiments, we combine this loss function with the reconstruction loss from the autoencoder so that the final loss becomes:

$$\mathcal{J} = \frac{1}{N} \sum_{i,j} \left(||\tilde{x}_j^i - x_j^i||^2 - \lambda \log \left(\frac{\sum_{k \neq i} e^{-\frac{||z_j^i - z_j^k||^2}{T}}}{\sum_{\substack{k,l \\ (k,l) \neq (i,j)}} e^{-\frac{||z_j^i - z_l^k||^2}{T}}} \right) \right)$$

where λ is a regularization coefficient.

A.2 Hybrid Architectures

Finally, we propose a hybrid architecture between siamese neural networks and autoencoders. The idea is to augment the siamese neural network by adding a decoder so that we can have add a reconstruction term to the loss function. We also add the other terms related to our customized disentanglement. The complete loss function looks like:

$$\mathcal{J} = \sum_{a=1}^{n} \sum_{i=1}^{I_s} \sum_{j \neq a} L_T(x_a^i, x_a^k, x_j^i) + \gamma L_R(x_a^i, x_a^k, x_j^i) + \lambda L_C(v_a^i, v_a^k, v_j^i)$$

with L_T as provided in Section 4.4, L_R is the reconstruction of the 3 terms (anchor, positive, and negative) and L_C corresponds to the configuration approximation for the 3 terms as well.

$$L_R(x_a^i, x_a^k, x_j^i) = ||\tilde{x}_a^i - x_a^i||^2 + ||\tilde{x}_a^k - x_a^k||^2 + ||\tilde{x}_j^i - x_j^i||^2$$

$$L_C(v_a^i, v_a^k, v_j^i) = ||\tilde{v}_a^i - v_a^i||^2 + ||\tilde{v}_a^k - v_a^k||^2 + ||\tilde{v}_j^i - v_j^i||^2$$

A.3 Regression on end target objective

For the encoder/decoder based architectures (namely the auto-encoders, siamese neural networks and hybrid aproaches) we have introduced in earlier sections losses that correspond to learning the representations. The full architecture however has a regression module that takes learned workload encodings z_j^i and fits a regression function on runtime latency. The regressor takes as input the job configuration v_j^i and z_j (the centroid of $\{z_j^i\}_i$ for a particular workload j) and tries to approximate at its output the runtime latency y_j^i.

The loss function for the regression is:

$$L = \frac{1}{N}\sum_{i,j}(f(v_j^i, z_j) - y_j^i)^2$$

With our best performing representation learning technique (siamese neural network), we tried both: (1) training separately (a) the siamese neural network and (b) the regression architecture (2) training the encoder first, then finetuning its layers when minimizing the loss function of the regressor. Fine tuning the encoder while training the regressor didn't give improvements on the test set errors for the end regression task.

B Additional Experimental Details

B.1 More Details regarding the traces

All of the traces have been collected from Spark clusters deployed on homogeneous hardware. Each Spark cluster spans 3 nodes (with 1 node reserved for the driver and 2 nodes left for executors). The traces contain two types of metrics: (1) Spark related metrics collected using Spark listener and (2) OS related metrics collected using the unix command *nmon*.

Preprocessing The traces are first averaged across the period of execution of the running Spark workloads. Then, we drop metrics for which the values are either constant or NaN across different traces. Then, we do minmaxscaling of both the knobs v_j^i as well as the retained runtime metrics x_j^i.

Within the streaming trace, we vary 10 knobs, and thus the dimension of v_j^i is 10. The total number of metrics in raw traces is 931 metrics. This number becomes 561 after preprocessing. Thus, the dimension of x_j^i is 561.

On the other hand, within the TPCx-BB trace we vary 12 knobs. Thus, the dimension of v_j^i is 12. The total number of metrics in raw traces is however 572 metrics. This number becomes 286 after applying the preprocessing. Thus, the dimension of x_j^i is 286.

B.2 Hyper-parameter tuning

For the encoder/decoder based architectures as well as the neural networks we tuned:

(1) topology related hyper-parameters (number of layers, number of hidden units per layer, activations) (2) learning related hyper-parameters (learning rate, number of epochs, patience (for early stopping) (3) loss related hyper-parameters (regularization coefficients introduced in losses having more than one term, temperature, etc...)

We tuned the hyper-parameters by random sampling from a pool of hyper-parameters. We use a 5-fold cross validation scheme for tuning our workloads. We try to simulate the same conditions on the test set when we do cross validation, thus we consider observing only few (1 or 5 traces) for training workloads within the left out fold during the cross validation procedure.

It may appear that we are casting tuning Spark workloads to tuning hyper-parameters of machine learning models. While this is partially correct, it is important to emphasize that the machine learning solution to modeling performances of spark workloads we are proposing can be tuned overnight (and not at the time of the execution of the Spark workload). Having a robust global model that allows us to predict performances of a new submitted Spark job from a unique (or few) trace, makes the tuning of Spark workload non-invasive to the user, and much faster.

C Implementation and Hardware Details

We have implemented all encoder/decoder based architectures as well as the neural network regressor using Tensorflow [1]. For the comparison with baseline representation learning techniques (PCA and KPCA), we used open source implementations from scikit-learn [16].

We have a dedicated server of 20 nodes for training our models. Each node has 2 x Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz processors (with 16 cores per processor) and 754 GB of RAM.