Bayesian Optimization for Machine Learning A Practical Guidebook

Ian Dewancker **Michael McCourt** **Scott Clark**

SigOpt
San Francisco, CA 94108
{ian, mike, scott}@sigopt.com

Abstract

The engineering of machine learning systems is still a nascent field; relying on a seemingly daunting collection of quickly evolving tools and best practices. It is our hope that this guidebook will serve as a useful resource for machine learning practitioners looking to take advantage of Bayesian optimization techniques. We outline four example machine learning problems that can be solved using open source machine learning libraries, and highlight the benefits of using Bayesian optimization in the context of these common machine learning applications.

1 Introduction

Recently, there has been interest in applying Bayesian black-box optimization strategies to better conduct optimization over hyperparameter configurations of machine learning models and systems [19] [21] [11]. Most of these techniques require that the objective be a scalar value depending on the hyperparamter configuration $\mathbf{x}$.

$$\mathbf{x}_{opt} = \arg\max_{\mathbf{x}\in\mathcal{X}} f(\mathbf{x})$$

A more detailed introduction to Bayesian optimization and related techniques is provided in [8]. The focus of this guidebook is on demonstrating several example problems where Bayesian optimization provides a noted benefit. Our hope is to clearly show how Bayesian optimization can assist in better designing and optimizing real-world machine learning systems. All of the examples in this guidebook have corresponding code available on SigOpt's example github repo.

2 Tuning Text Classification Pipelines with scikit-learn

Text classification problems appear quite often in modern information systems, and you might imagine building a small document / tweet / blogpost classifier for any number of purposes. In this example, the classification task is to label Amazon product reviews [5] as either favorable or not. The objective is to find a classifier that is accurate in its predictions, but also one that gives us confidence it will generalize to data it has not been trained on. We employ the Swiss army knife of machine learning, logistic regression (LR), as our model in this experiment. While the LR model might be conceptually simple [16] and implemented in many statistics and machine learning software packages, valuable engineering time and resources are often wasted experimenting with feature representation and parameter tuning via trial and error.

2.1 Objective Metric : $f(\boldsymbol{\lambda})$

SigOpt finds parameter configurations that maximize any metric, so we need to pick one that is appropriate for this classification task. We'll use $f(\boldsymbol{\lambda})$ to denote our objective metric function and $\boldsymbol{\lambda}$

to represent the set of tunable parameters, which we discuss in the following section. In designing our objective metric, accuracy, the number of correctly classified reviews, is obviously important, but we also want assurance that our model generalizes and can perform well on data on which it was not trained. This is where the idea of cross-validation comes into play.

Cross-validation requires us to split up our entire labeled dataset $\mathcal{D}$ into two distinct sets: one to train on $\mathcal{D}_{\text{train}}$ and one to validate our trained classifier on $\mathcal{D}_{\text{valid}}$. We then consider metrics like accuracy on only the validation set. Taking this further and considering not one, but many possible splits of the labeled data is the idea of k-fold cross-validation where multiple training, validation sets are generated and validation metrics can be aggregated in several ways (e.g., mean, min, max) to give a single estimation of performance.

In this case, we'll use the mean of the k-folded cross-validation accuracies [10]. In our case, $k = 5$ folds are used and the train and validation sets are split randomly using 70% and 30% of the entire dataset, respectively.

$$\mathcal{L}(\boldsymbol{\lambda}, \mathcal{D}_{\text{t}}, \mathcal{D}_{\text{v}}) = \text{acc. of LR}(\boldsymbol{\lambda}, \mathcal{D}_{\text{t}}) \text{ on } \mathcal{D}_{\text{v}}$$

$$f(\boldsymbol{\lambda}) = \frac{1}{k} \sum_{i=1}^{k} \mathcal{L}(\boldsymbol{\lambda}, \mathcal{D}_{\text{train}}^{(i)}, \mathcal{D}_{\text{valid}}^{(i)})$$

This objective metric $f(\boldsymbol{\lambda})$ takes on values in the range [0, 1.0], where 0 represents a mis-classification of every example in all validation folds and 1.0 represents perfect classification on all validation folds. The higher the cross-validation metric, the better our classifier is doing. Using many folds might not be practical if training takes an very long time (you might have to settle for 1 or 2 folds only).

2.2 Tunable Parameters : $\boldsymbol{\lambda}$

The objective metric, $f(\boldsymbol{\lambda})$, is controlled by a set of parameters, $\boldsymbol{\lambda}$, that potentially influence its performance. Parameters can be defined on continuous, integer or categorical domains. The parameters used in this experiment can be split into two groups: those governing the feature representation of the review text and those governing the cost function of logistic regression. We explain these sets of parameters in the following sections.

2.2.1 Feature Representation Parameters

The CountVectorizer class in scikit-learn is a convenient mechanism for transforming a corpus of text documents into vectors using bag of words representations (BOW). scikit-learn offers quite a bit of control in determining which n-grams make up the vocabulary for your BOW vectors. As a quick refresher, n-grams are sequences of text tokens as shown below:

Original Text	"SigOpt optimizes any complicated system"
1-grams	{"SigOpt", "optimizes", "any", "complicated", "system" }
2-grams	{"SigOpt_optimizes", "optimizes_any", "any_complicated" ... }
3-grams	{ "SigOpt_optimizes_any", "optimizes_any_complicated" ... }

Table 1: Example n-grams for a sample piece of text

The number of times each n-gram appears in a given piece of text is then encoded in the BOW vector describing that text. CountVectorizer allows you to control the range of n-grams that are included in the vocabulary ($min\_n\_gram, ngram\_offset$ in our experiment), as well as filtering n-grams outside a specified document-frequency range (log_min_df, df_offset in our experiment). For example, if a rare 3-gram like "hi_diddly_ho" doesn't appear with at least min-df frequency in the corpus, it is not included in the vocabulary. Similarly, n-grams that occur in nearly every document (1-grams like "the", "a" etc) can also be filtered using the max-df parameter. Often when the range of the parameter is very large or very small, it makes sense to look at the parameter on the log scale, as we do with the log_min_df parameter.

2.2.2 Logistic Regression Error Cost Parameters

Using the SGDClassifier class in scikit-learn, we can succinctly formulate and solve the logistic regression learning problem. The error function for logistic regression, two-class classification is defined in the following way:

$$E(\boldsymbol{\theta}) = \frac{1}{M}\sum_{i=1}^{M}\log\left(1.0 + e^{-y_i(\boldsymbol{\theta}^T\mathbf{x_i})}\right) + \alpha\left(\frac{1-\rho}{2}\|\boldsymbol{\theta}\|_2^2 + \rho\|\boldsymbol{\theta}\|_1\right)$$

M = number of training examples
$\boldsymbol{\theta}$ = vector of weights the algorithm will learn for each n-gram in vocabulary
y_i = training data label : {-1, 1} for our two class problem
$\mathbf{x}_i$ = training data input vector: BOW vectors described in previous section
α = weight of regularization term
ρ = weight of L1 norm term

The first term of the cost function penalizes weights that do not fit the training data while the second term penalizes model complexity (how far are the feature weights away from zero). scikit-learn performs stochastic gradient descent on this error function with respect to the weights in an attempt to find those that minimize this function.

Should we use L1 or L2 regularization, or perhaps a weighted mixture? How much should the entire regularization term be weighted? With this error formulation, and the α and ρ parameters exposed in our experiment, SigOpt can quickly find these answers to these important questions.

2.3 Experimental Results

SigOpt offers one solution to the hyperparameter optimization problem, however there are other existing techniques. In particular, random search and grid search are two commonly employed strategies. Random search, as you might guess, simply selects parameter configurations at random, while grid search sweeps through a selected subset of the parameter space.

How should we evaluate the performance of these alternative optimization strategies? One criterion that makes sense is to consider the best found (max) value of the objective metric after optimization is complete. Better performing strategies will find better configurations over the duration of their search. Due to the stochastic nature of these systems however, we must consider the variation in our best found measurements over several runs to make fair comparisons.

To ground our discussion, we also report the performance when no hyperparameter optimization is performed, and we simply take the default values for CountVectorizer and SGDClassifier as provided by scikit-learn. For grid search, we consider 64 evenly spaced parameter configurations (order shuffled randomly) across our domain and analyze the best seen after 60 evaluations to be consistent with our limit on the total number of evaluations for this experiment. Exhaustive grid search is usually prohibitive because the number of possible configurations grows exponentially.

	SigOpt	Rnd. Search	Grid Search	No Tuning (Baseline)
Best Found ACC	**0.8760** (+5.72%)	0.8673	0.8680	0.8286

Table 2: Best found accuracy results averaged over 20 optimization runs, each run consisting of 60 function evaluations

SigOpt finds the best configuration with statistical significance over the other two approaches (p = 0.0001, using the unpaired Mann-Whitney U test) and improves the performance as compared to the baseline by 5.72%.

3 Unsupervised Feature Learning with scikit-image and xgboost

As the previous section discussed, fully supervised learning algorithms require each data point to have an associated class or output. In practice, however, it is often the case that relatively few labels are available during training time and labels are costly or time consuming to acquire. For example, it might be a very slow and expensive process for a group of experts to manually investigate and classify thousands of credit card transaction records as fraudulent or legitimate. A better strategy might be to study the large collection of transaction data without labels, building a representation that better captures the variations in the transaction data automatically.

3.1 Unsupervised Learning

Unsupervised learning algorithms are designed with the hope of capturing some useful latent structure in data. These techniques can often enable dramatic gains in performance on subsequent supervised learning task, without requiring more labels from experts. In this post we will use an unsupervised method on an image recognition task posed by researchers at Stanford [6] where we try to recognize house numbers from images collected using Google street view (SVHN). This is a more challenging problem than MNIST (another popular digit recognition data set) as the appearance of each house number varies quite a bit and the images are often cluttered with neighboring digits:

Figure 1: 32×32 cropped samples from the classification task of the SVHN dataset. Each sample is assigned only a single digit label (0 to 9) corresponding to the center digit. (Sermanet [18])

In this example, we assume access to a large collection of unlabelled images X_u, where the correct answer is not known, and a relatively small amount of labelled data $(\mathsf{X}_s, \mathbf{y})$ for which the true digit in each image is known (often requiring a non-trivial amount of time and money to collect). Our hope is to find a suitable unsupervised model, built using our large collection of unlabelled images, that transforms images into a more useful representation for our classification task.

Unsupervised and supervised learning algorithms are typically governed by small sets of hyperparameters $(\boldsymbol{\lambda}_u, \boldsymbol{\lambda}_s)$, that control algorithm behavior. In our example pipeline below, X_u is used to build the unsupervised model f_u which is then used to transform the labelled data $(\mathsf{X}_s, \mathbf{y})$ before the supervised model f_s is trained. Our task is to efficiently search for good hyperparameter configurations $(\boldsymbol{\lambda}_u, \boldsymbol{\lambda}_s)$ for both the unsupervised and supervised algorithms. SigOpt minimizes the classification error $E(\boldsymbol{\lambda}_u, \boldsymbol{\lambda}_s)$ by sequentially generating suggestions for the hyperparameters of the model $(\boldsymbol{\lambda}_u, \boldsymbol{\lambda}_s)$. For each suggested hyperparameter configuration a new unsupervised data representation is formed and fed into the supervised model. The observed classification error is reported and the process repeats, converging on the set of hyperparameters that minimizes the classification error.

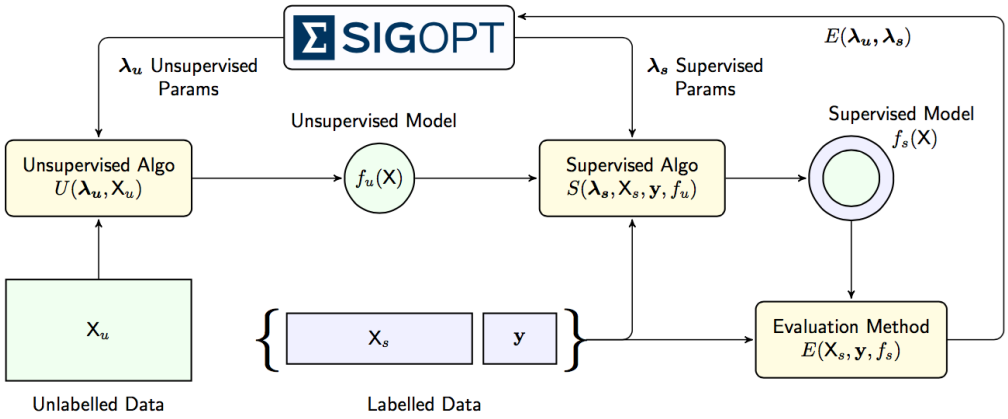

Figure 2: Process for coupled unsupervised and supervised model tuning.

SigOpt offers Bayesian optimization as a service, capable of efficiently searching through the joint variations $(\boldsymbol{\lambda}_u, \boldsymbol{\lambda}_s)$ of both the supervised and unsupervised aspects of machine learning systems, as depicted in Figure 2. This allows experts to unlock the power of unsupervised strategies with the assurance that each model is reaching its full potential automatically.

3.2 Unsupervised Model

We start with the initial features describing the data: raw pixel intensities for each image. The goal of the unsupervised model is to transform the data from its original representation to a new (more useful) learned representation without using labeled data. Specifically, you can think of this unsupervised model as a function $f : \mathbb{R}^N \rightarrow \mathbb{R}^J$. Where N is the number of features in our original representation and J is the number of features in the learned representation. In practice, expanded representations (sometimes referred to as a feature map) where J is much larger than N often work well for improving performance on classification tasks [2].

3.2.1 Image Transform Parameters (s, w, K)

A simple but surprisingly effective transformation for small images was proposed in a paper by Coates [6] where image patches are transformed into distances to K learned centroids (average patches) using the k-means algorithm, and then pooled together to form a final feature representation as outlined in Figure 3 below:

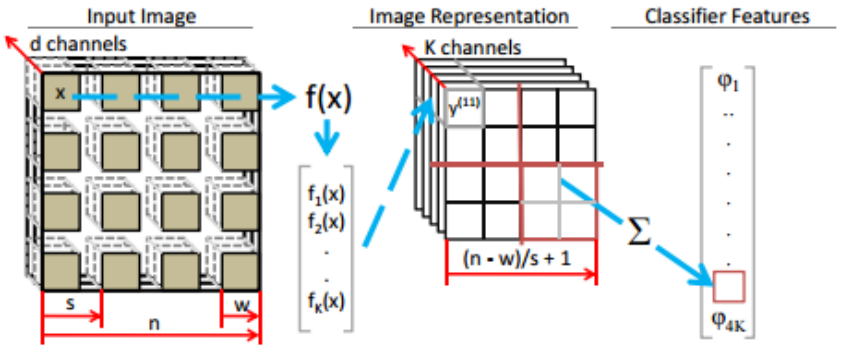

Figure 3: Feature extraction using a $w \times w$ receptive field and stride s. $w \times w$ patches separated by s pixels each, then map them to K-dimensional feature vectors to form a new image representation. The vectors are then pooled over the image quadrants to form the classifier feature vector. Coates [6]

In this example we are working with the 32x32 (n=32) converted gray-scale (d=1) images of the SVHN dataset. We allow SigOpt to vary the stride length (s) and patch width (w) parameters. The figure above illustrates a pooling strategy that considers quadrants in the 2x2 grid of the transformed image representation, summing them to get the final transformed vector. We used the suggested resolution in [6] and kept $pool_r$ fixed at 2. $\mathsf{f}(x)$ represents a K dimensional vector that encodes the distances to the K learned centroids, and $\mathsf{f}_i(x)$ refers to the distance of image patch instance x to centroid i. In this experiment, K is also a tunable parameter. The final feature representation of each image will have $J = K \cdot pool_r^2$ features.

3.2.2 Whitening Transform Parameter (ϵ_{zca})

Before generating the image patch centroids and any subsequent patch comparisons to these centroids, we apply a whitening transform to each patch. When dealing with image data, whitening is a common preprocessing transform which removes the correlation between all pairs of individual pixels [14]. Intuitively, it can be thought of as a transformation that highlights contrast in images. It has been shown to be helpful in image recognition tasks, and may also be useful for other feature data. The figure below shows several example image patches before and after the whitening transform.

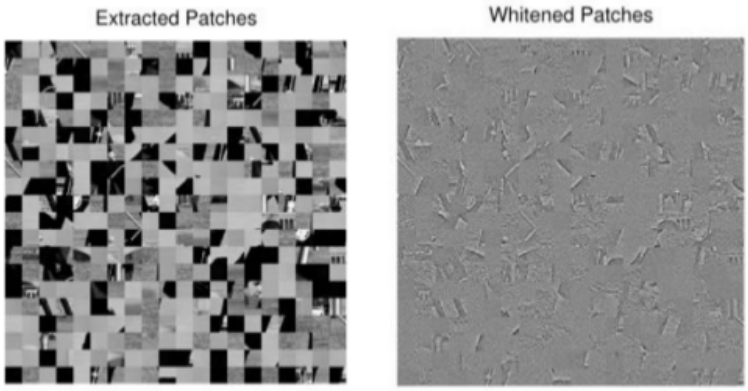

Figure 4: Comparison of image patches before and after whitening (Stansbury [20])

The whitening transformation we use is known as ZCA whitening [7]. This transform is achieved by cleverly applying the eigendecomposition of the covariance matrix estimate to a mean adjusted version of the data matrix, so that the expected covariance of the data matrix becomes the identity. A regularization term ϵ_{zca} is added to the diagonal eigenvalue matrix, and ϵ_{zca} is exposed as a tunable parameter to SigOpt.

$$
\begin{aligned}
\text{cov}(\mathsf{X}) &= \mathsf{U}\Lambda\mathsf{U}^\mathsf{T} \\
\Lambda^{-\frac{1}{2}} &= \text{diag}(1/\sqrt{\Lambda_{ii}}) \\
\mathsf{X}_{\text{zca}} &= (\mathsf{X} - \mathbf{1}\boldsymbol{\mu}^\mathsf{T})\mathsf{U}(\Lambda + \epsilon_{\text{zca}}\mathsf{I})^{-\frac{1}{2}}\mathsf{U}^\mathsf{T}
\end{aligned}
$$

3.2.3 Centroid Distance Sparsity Parameter ($sparse_p$)

Each whitened patch in the image is transformed by considering the distances to the learned K centroids. To control this sparsity of the representation we report only distances that are below a certain percentile, $sparse_p$, when considering the pairwise distances between the current patch and the centroids. Intuitively this acts as a threshold which allows for only the "close" centroids to be active in our representation.

Figure 5 below illustrates the idea with a simplified example. A whitened image patch (in the upper right) is compared against the 4 learned centroids after k-means clustering. Here, let's imagine we have set the percentile threshold to 50, so only the distances in the lower half of all centroid distances persist in the final representation, the others are zeroed out

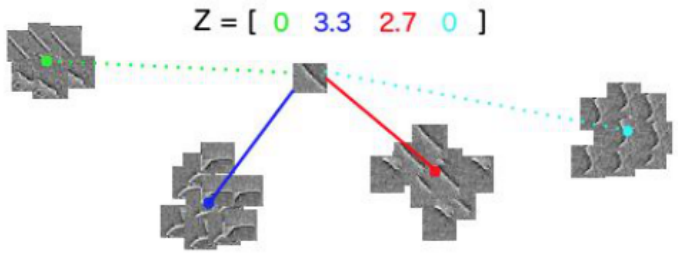

Figure 5: Sparsity transform; distances from a test patch to centroids $>$ 50th percentile are set to 0

While the convolutional aspects of this unsupervised model are tailored to image data, the general approach of transforming feature data into a representation that reflects distances to learned archetypes seems suitable for other data sets and feature spaces [9].

3.3 Supervised Model

With the learned representation of our data, we now seek to maximize performance on our classification task using a smaller labelled dataset. While random forests are an excellent, and simple, classification tool, better performance can typically be achieved by using carefully tuned ensembles of boosted classification trees.

3.3.1 Gradient Boosting Parameters (γ, θ, M)

We consider the popular library XGBoost as our gradient boosting implementation. Gradient boosting is a generic boosting algorithm that incrementally builds an additive model of base learners, which are themselves simpler classification or regression models. Gradient boosting works by building a new model at each iteration that best reconstructs the gradient of the loss function with respect to the previous ensemble model. In this way it can be seen as a sort of functional gradient descent, and is outlined in more detail below. In the pseudocode below we outline building an ensemble of regression trees, but the same method can be used with a classification loss function L

Algorithm 1 Gradient Boost

Input: $\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)\}, \theta, \gamma$
Output: $F(\mathbf{x}) = \sum_{i=0}^{M} F_i(\mathbf{x})$
$F_0(\mathbf{x}) \leftarrow \arg\min_\beta \sum_{i=1}^{N} L(y_i, \beta)$
for $m \leftarrow 1$ **to** M **do**
  $d_i = -\left[\frac{\partial L(y_i, F(\mathbf{x}_i))}{\partial F(\mathbf{x}_i)}\right]_{F(\mathbf{x}_i)=F_{m-1}(\mathbf{x}_i)}$
  $\mathcal{G} \leftarrow \{(\mathbf{x}_i, d_i)\}, i = 1, N$
  $g(\mathbf{x}) \leftarrow$ FitRegrTree$(\mathcal{G}, \theta)$
  $\rho_m \leftarrow \arg\min_\rho \sum_{i=1}^{N} L(y_i, F_{m-1}(\mathbf{x}) + \rho g(\mathbf{x}))$
  $F_m(\mathbf{x}) \leftarrow F_{m-1}(\mathbf{x}) + \gamma\, \rho_m g(\mathbf{x})$
end for

3.4 Experimental Results

We compare the ability of SigOpt to find the best hyperparameter configuration to random search, which usually outperforms grid search and manual search (Bergstra [3]) and a baseline of using an untuned model.

Because the underlying methods used are inherently stochastic we performed 10 independent hyperparameter optimizations using both SigOpt and random search for both the purely supervised and combined models. Hyperparameter optimization was performed on the accuracy estimate from a 80/20 cross validation fold of the training data (73k examples). The 'extra' set associated with the

SVHN dataset (530K examples) was used to simulate the unlabelled data X_u in the unsupervised parts of this example.

For the unsupervised model 90 sequential configuration evaluations (50 CPU hrs) were used for both SigOpt and random search. For the purely supervised model 40 sequential configuration evaluations (8 CPU hrs) were used for both SigOpt and random search. In practice, SigOpt is usually able to find good hyperparameter configurations with a number of evaluations equal to 10 times the number of parameters being tuned (9 for the combined model, 4 for the purely supervised model). The same parameters and domains were used for XGBoost in both the unsupervised and purely supervised settings. As a baseline, the hold out accuracy of an untuned scikit-learn random forest using the raw pixel intensity features.

After hyperparameter optimization was completed for each method we compared accuracy using a completely held out data set (SHVN test set, 26k examples) using the best configuration found in the tuning phase. The hold out dataset was run 10 times for each best hyperparameter configuration for each method, the mean of these runs is reported in the table below. SigOpt outperforms random search with a p-value of 0.0008 using the unpaired Mann-Whitney U test.

	SigOpt (xgboost + Unsup. Feats)	Rnd Search (xgboost + Unsup. Feats)	SigOpt (xgboost + Raw Feats)	Rnd Search (xgboost + Raw Feats)	No Tuning (sklearn RF + Raw Feats)
Hold out ACC	**0.8601** (**+49.2%**)	0.8190	0.7483	0.7386	0.5756

Table 3: Comparison of model accuracy on held out (test) dataset after different tuning strategies

The chart below in Figure 6 shows the optimization traces of SigOpt versus random search optimization strategies when tuning the unsupervised model (Unsup Feats) and only the supervised model (Raw Feats). We plot the interquartile range of the best seen cross validated accuracy score on the training set at each objective evaluation during the optimization. As mentioned above, 90 objective evaluations were used in the optimization of the unsupervised model and 40 in the supervised setting. SigOpt outperforms random search in both settings on this training data (p-value 0.005 using the same Mann-Whitney U test as before).

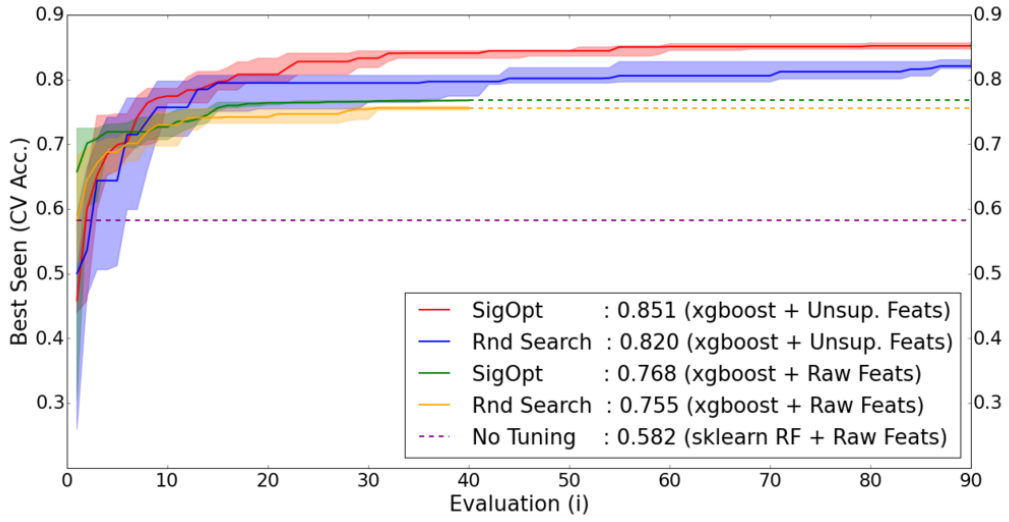

Figure 6: Optimization traces of CV accuracy using SigOpt and random search.

4 Deep Learning with TensorFlow

There are a large number of tunable parameters associated with defining and training deep neural networks [1] [4] and SigOpt accelerates searching through these settings to find optimal configurations. This search is typically a slow and expensive process, especially when using standard techniques like grid or random search, as evaluating each configuration can take multiple hours. SigOpt finds good combinations far more efficiently than these standard methods by employing an ensemble of Bayesian optimization techniques.

In this example, we consider the same optical character recognition task of the SVHN dataset as discussed in the previous section. Our goal is to build a model capable of recognizing digits (0-9) in small, real-world images of house numbers. We use SigOpt to efficiently find a good structure and training configuration for a convolutional neural net.

4.1 Convolutional Neural Net Structure

The structure and topology of a deep neural network can have dramatic implications for performance on a given task [1]. Many small decisions go into the connectivity and aggregation strategies for each of the layers that make up a deep neural net. These parameters can be non-intuitive to choose in an optimal, or even acceptable, fashion. In this experiment we used a TensorFlow CNN example designed for the MNIST dataset as a starting point. Figure 7 represents a typical CNN structure, highlighting the parameters we chose to vary in this experiment. A more complete discussion of these architectural decisions can be found in an online course from Stanford (Li [15]). It should be noted that Figure 7 is an approximation of the architecture used in this example, and the code in the SigOpt examples repository serves as a more complete reference.

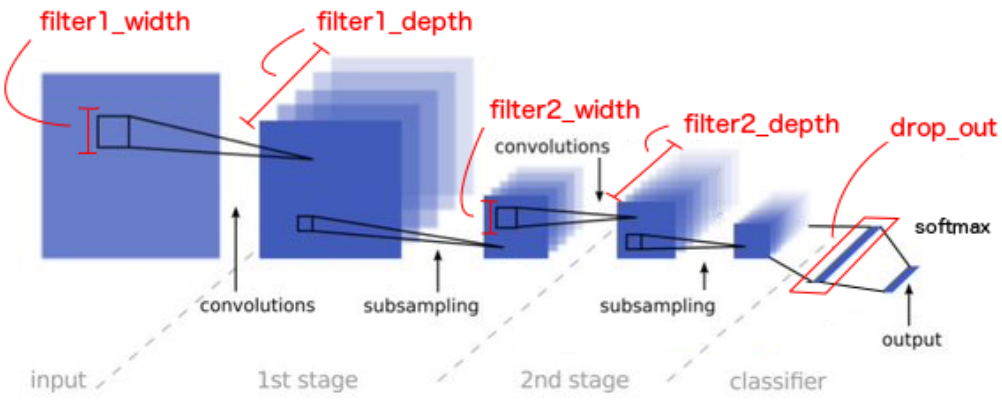

Figure 7: Representative convolutional neural net topology. Important parameters include the width and depth of the convolutional filters, as well as dropout probability [18]

TensorFlow has greatly simplified the effort required to build and experiment with deep neural network (DNN) designs. Tuning these networks, however, is still an incredibly important part of creating a successful model. The optimal structural parameters often highly depend on the dataset under consideration.

4.2 Stochastic Gradient Descent Parameters (α, β, γ)

Once the structure of the neural net has been selected, an optimization strategy based on stochastic gradient descent (SGD) is used to fit the weight parameters of the convolutional neural net. There is no shortage of SGD algorithm variations implemented in TensorFlow. To demonstrate how drastically their behavior can vary under different parameterizations, Figure 8 compares several configurations of RMSProp, a particular SGD variation on a simple 2D objective.

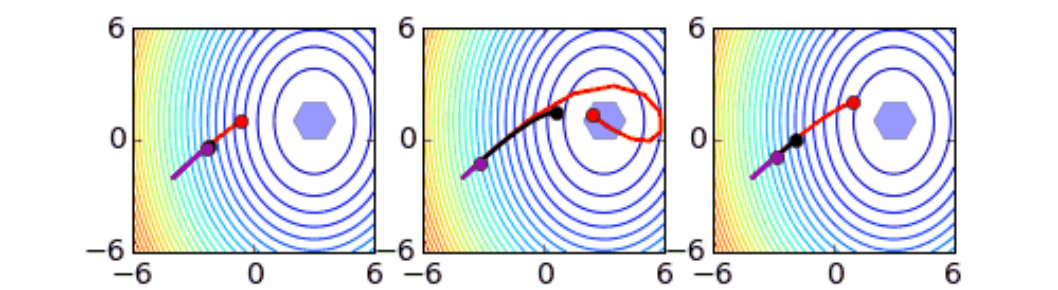

Figure 8: Progression of RMSProp gradient descent after 12 update steps under different parametrizations. left: Various decay rates with other parameters fixed: purple = .01, black = .5, red = .93. center: Various learning rates with other parameters fixed: purple = .016, black = .1, red = .6. right: Various momentums with other parameters fixed: purple = .2, black = .6, red = .93.

It can be a counterintuitive and time consuming task to optimally configure a particular SGD algorithm for a given model and dataset. To simplify this tedious process, we expose to SigOpt the parameters that govern the RMSProp optimization algorithm. Important parameters governing its behavior are the learning rate α , momentum β and decay γ terms. These parameters define the RMSProp gradient update step, outlined in the pseudo code below:

Algorithm 2 RMSProp Stochastic Gradient Descent

Input: $\nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}), \boldsymbol{\theta}_0, \alpha, \beta, \gamma, \epsilon$
$\mathbf{m}_0 \leftarrow \mathbf{0}$
$\mathbf{b}_0 \leftarrow \mathbf{0}$
for $t \leftarrow 1$ **to** T **do**
  $\mathbf{g} \leftarrow \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}_{t-1})$ stochastic gradient
  $\mathbf{m}_t[i] \leftarrow \gamma \mathbf{m}_{t-1}[i] + (1-\gamma)\mathbf{g}[i]^2 \qquad i = 1 \ldots N$
  $\mathbf{b}_t[i] \leftarrow \beta \mathbf{b}_{t-1}[i] + \alpha \left(\frac{\mathbf{g}[i]}{\sqrt{(\mathbf{m}_t[i] + \epsilon)}} \right) \qquad i = 1 \ldots N$
  $\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \mathbf{b}$
end for

For this example, we used only a single epoch of the training data, where one epoch refers to a complete presentation of the entire training data (500K images in our example). Batch size refers to the number of training examples used in the computation of each stochastic gradient (10K images in our example). One epoch is made up of several batch sized updates, so as to minimize the in-memory resources associated required for the optimization (Hinton [12]). Using only a single epoch can be detrimental to performance, but this was done in the interest of time for this example.

4.3 Experimental Results

To compare tuning the CNNs hyperparameters when using random search versus SigOpt, we ran 5 experiments using each method and compared the median best seen trace. The objective was the classification accuracy on a single 80 / 20 fold of the training and "extra" set of the SVHN dataset (71K + 500K images respectively). The median best seen trace for each optimization strategy is shown below in Figure 9.

In our experiment we allowed SigOpt and random search to perform 80 function evaluations (each representing a different proposed configuration of the CNN). A progression of the best seen objective at each evaluation for both methods is shown below in Figure 9. We include, as a baseline, the accuracy of an untuned TensorFlow CNN using the default parameters suggested in the official TensorFlow example. We also include the performance of a random forest classifier using sklearn defaults.

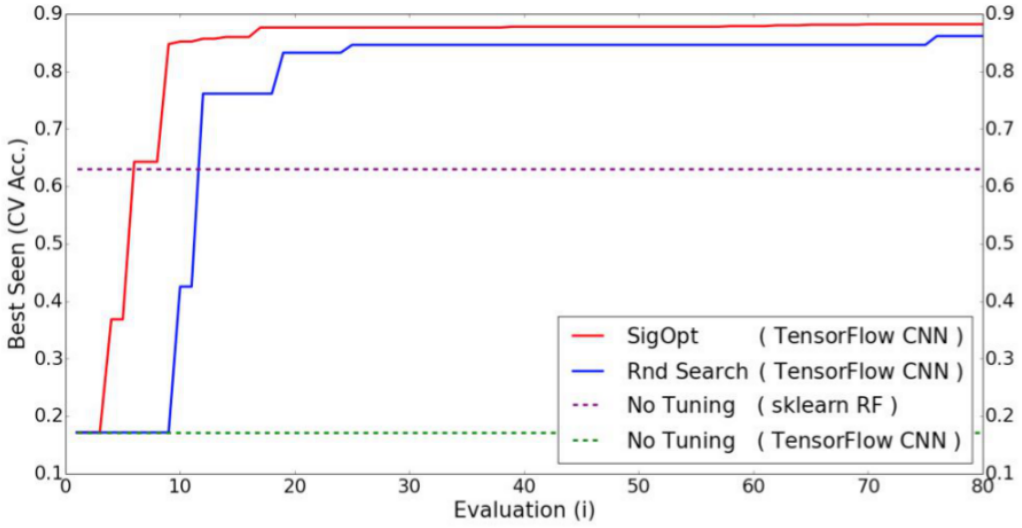

Figure 9: Median best seen trace of CV accuracy over 5 independent optimization runs using SigOpt, random search as well as two baselines where no tuning was performed.

After hyperparameter optimization was completed for each method, we compared accuracy using a completely held out data set (SHVN test set, 26K images) using the best configuration found in the tuning phase. The best hyperparameter configurations for each method in each of the 5 optimization runs was used for evaluation. The mean of these accuracies is reported in the table below. We also include the same baseline models described above and report their performance on the held out evaluation set.

	SigOpt (TensorFlow CNN)	Random Search (TensorFlow CNN)	No Tuning (sklearn RF)	No Tuning (TensorFlow CNN)
Hold out ACC	**0.8130** (**+315.2%**)	0.5690	0.5278	0.1958

Table 4: Comparison of model accuracy on the held out (test) dataset after different tuning strategies

5 Recommendation Systems with MLlib

A popular approach for building the basis of a recommendation system is to learn a model capable of predicting users' product preferences or ratings. With an effective predictive model, and enough contextual information about users, online systems can better suggest content or products, helping to promote sales, subscriptions or conversions.

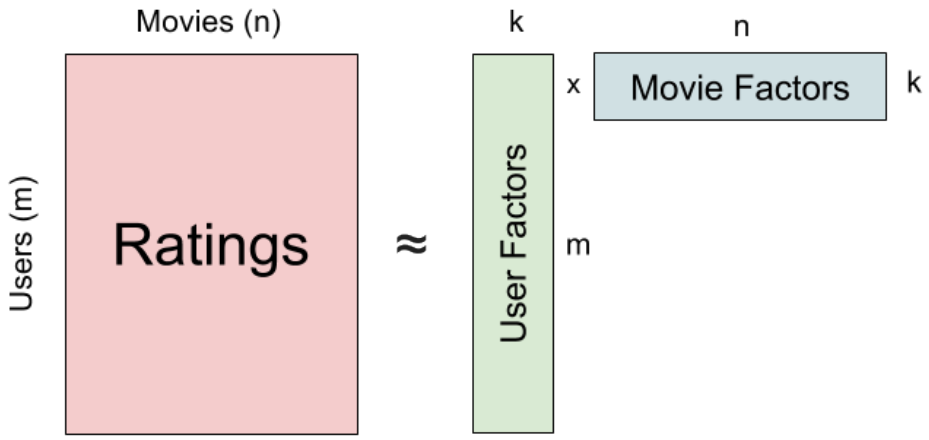

Figure 10: Collaborative Filtering via Low-Rank Matrix Factorization

A common recommender systems model involves using a low-rank factorization of a user-product ratings matrix to predict the ratings of other products for each user [13]. In general, algorithms related to collaborative filtering and recommendation systems will have tunable parameters similar to ones we have discussed in previous sections. In this problem, for example, the regularization term on the user and product factors can be difficult to choose a priori without some trial and error.

In this example we consider the MovieLens dataset and use the MLlib package within Apache Spark. The code for this example is available in the SigOpt examples github repository. We use the largest MovieLens dataset ratings matrix which has approximately 22 million user ratings for 33,000 movies by 240,000 users. To run this example, we recommend creating a small spark cluster in ec2 using the spark-ec2 tool provided in the spark library. We ran this experiment using a 3 machine cluster (1 master, 2 workers) in AWS using the m1.large instance for all nodes.

5.1 Alternating Least Squares

To solve for the latent user and movie factors, MLlib implements a variant of what is known as quadratically regularized PCA [22]. Intuitively, this optimization problem aims to learn latent factors X, Y that best recreate the ratings matrix A, with a regularization penalty coefficient λ on the learned factors. Here $\mathbf{x}_i$ represents the ith row of the X factor matrix and $\mathbf{y}_j$ represents the jth column of the Y factor matrix.

$$\arg\min_{\mathbf{x}_i, \mathbf{y}_j} \sum_{i=1}^{m} \sum_{j=1}^{n} (A_{ij} - \mathbf{x}_i \mathbf{y}_j)^2 + \lambda \sum_{i=1}^{m} ||\mathbf{x}_i||_2^2 + \lambda \sum_{j=1}^{n} ||\mathbf{y}_j||_2^2$$

This minimization problem can be solved using a technique known as alternating least squares [22] . A distinct advantage of using this formulation is that it can be easily parallelized into many independent least square problems as outlined in the pseudocode below. Each factor matrix X, Y is randomly initialized and the algorithm alternates between solving for the user factors X, holding the movie factors Y constant, then solving for the Y factors, holding X constant. The algorithm takes as input A the ratings matrix, λ the regularization term, k the desired rank of the factorization, and T the number of iterations of each alternating step in the minimization. We expose λ, k and T as tunable parameters to SigOpt.

Algorithm 3 Parallel Alternating Least Squares

Input: $A \in \mathbb{R}^{m \times n}$, λ, k, T
$X \leftarrow \text{RANDINIT}(m, k)$ ▷ Initialize factors
$Y \leftarrow \text{RANDINIT}(k, n)$
for $iter \leftarrow 1$ **to** T **do**
  par for $i \leftarrow 1$ **to** m ▷ Executed in parallel
    $\mathbf{x}_i \leftarrow \arg\min_{\mathbf{x}_i} ||\mathbf{x}_i Y - A_{i,*}||_2^2 + \lambda ||\mathbf{x}_i||_2^2$
  par for $j \leftarrow 1$ **to** n ▷ Executed in parallel
    $\mathbf{y}_j \leftarrow \arg\min_{\mathbf{y}_j} ||X\mathbf{y}_j - A_{*,j}||_2^2 + \lambda ||\mathbf{y}_j||_2^2$
end for

The regularization term λ is particularly difficult to select optimally as it can drastically change the generalization performance of the algorithm. Previous work has attempted to use a Bayesian formulation of this problem to avoid optimizing for this regularization term explicitly [17]

5.2 Experimental Results

As an error metric for this example, we used the standard measurement of the root mean square error [13] of the reconstructions on a random subset of nonzero entries from the ratings matrix.

$$\text{RMSE} = \sqrt{\sum_{(i,j) \in TestSet} \frac{(A_{ij} - \mathbf{x}_i \mathbf{y}_j)^2}{|TestSet|}}$$

Defining an appropriate error measurement for a recommendation task is critical for achieving success. Many other metrics have been proposed for evaluating recommendation systems and careful selection is required to tune for models that are best for the application at hand. Bayesian optimization methods like SigOpt can be used to tune any underlying metric, or a composite metric of many metrics (like accuracy and training time). In this example the training, validation and holdout rating entries are randomly sampled non-zero entries from the full ratings matrix A, summarized in the diagram below:

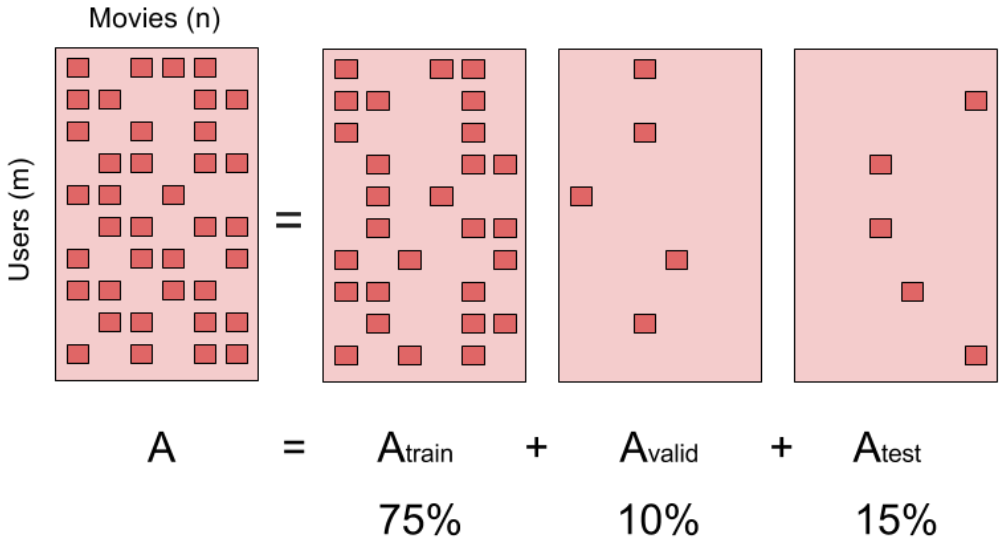

Figure 11: Train, validation and test sets for user movie ratings prediction

SigOpt tunes the alternating least square algorithm parameters with respect to the root mean squared error of the validation set. We also report the performance on the hold out set as a measure of how well the algorithm generalizes to data it has not seen. We compare parameters tuned using SigOpt against leaving the alternating least square parameters untuned. While the ratings entries for the train, valid and test sets were randomly sampled, they were identical sets in the SigOpt and the untuned comparisons.

	SigOpt	Random Search	No Tuning (Default MLlib ALS)
Hold out RMSE	**0.7864** (**-40.7%**)	0.7901	1.3263

Table 5: Comparison of RMSE on the hold out (test) ratings after tuning ALS algorithm

References

[1] Yoshua Bengio. Learning deep architectures for ai. *Foundations and Trends in Machine Learning*, 2(1):1–127, 2009.

[2] Yoshua Bengio et al. Deep learning of representations for unsupervised and transfer learning. *ICML Unsupervised and Transfer Learning*, 27:17–36, 2012.

[3] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

[4] James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in Neural Information Processing Systems*, pages 2546–2554, 2011.

[5] John Blitzer, Mark Dredze, Fernando Pereira, et al. Biographies, bollywood, boom-boxes and blenders: Domain adaptation for sentiment classification. In *ACL*, volume 7, pages 440–447, 2007.

[6] Adam Coates, Honglak Lee, and Andrew Y Ng. An analysis of single-layer networks in unsupervised feature learning. *Ann Arbor*, 1001(48109):2, 2010.

[7] Adam Coates and Andrew Y Ng. Learning feature representations with k-means. In *Neural Networks: Tricks of the Trade*, pages 561–580. Springer, 2012.

[8] Ian Dewancker, Michael McCourt, and Scott Clark. Bayesian optimization primer. `https://sigopt.com/static/pdf/SigOpt_Bayesian_Optimization_Primer.pdf`, 2015.

[9] Sander Dieleman and Benjamin Schrauwen. Multiscale approaches to music audio feature learning. In *14th International Society for Music Information Retrieval Conference (ISMIR-2013)*, pages 116–121. Pontifícia Universidade Católica do Paraná, 2013.

[10] Katharina Eggensperger, Matthias Feurer, Frank Hutter, James Bergstra, Jasper Snoek, Holger Hoos, and Kevin Leyton-Brown. Towards an empirical foundation for assessing bayesian optimization of hyperparameters. In *NIPS workshop on Bayesian Optimization in Theory and Practice*, pages 1–5, 2013.

[11] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter. Efficient and robust automated machine leraning. In *Advances in Neural Information Processing Systems 28*, pages 2944–2952, December 2015.

[12] Nitish Srivastav Geoffrey Hinton and Kevin Swersky. *Neural Networks for Machine Learning*, 2015.

[13] Yehuda Koren. Factorization meets the neighborhood: a multifaceted collaborative filtering model. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 426–434. ACM, 2008.

[14] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. 2009.

[15] Fei-Fei Li, Andrej Karpathy, and Justin Johnson. *Convolutional Neural Networks for Visual Recognition*, 2015.

[16] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[17] Ruslan Salakhutdinov and Andriy Mnih. Probabilistic matrix factorization. In *Neural Information Processing Systems*, volume 21, 2007.

[18] Pierre Sermanet, Soumith Chintala, and Yann LeCun. Convolutional neural networks applied to house numbers digit classification. In *Pattern Recognition (ICPR), 2012 21st International Conference on*, pages 3288–3291. IEEE, 2012.

[19] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

[20] Dustin Stansbury. *The Statistical Whitening Transform*, 2014 (accessed March, 2015).

[21] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 847–855. ACM, 2013.

[22] Madeleine Udell, Corinne Horn, Reza Zadeh, and Stephen Boyd. Generalized low rank models. *Foundations and Trends in Machine Learning*, 9(1), 2016.