Title: Meta-Learning Algorithms and their Applications to Quantum Computing

Speakers: Mat Kallada

Series: Condensed Matter

Date: January 28, 2020 - 2:30 PM

URL: http://pirsa.org/20010095

Abstract: Meta-learning involves learning mathematical devices using problem instances as training data. In this talk, we first describe recent meta-learning approaches involving the learning of objects such as: initial weights, parameterized losses, hyper-parameter search strategies, and samplers. We then discuss learned optimizers in further detail and their applications towards optimizing variational circuits. This talk also covers some lessons learned starting a spin-off from academia.

Learned Optimizers & Quantum Computing

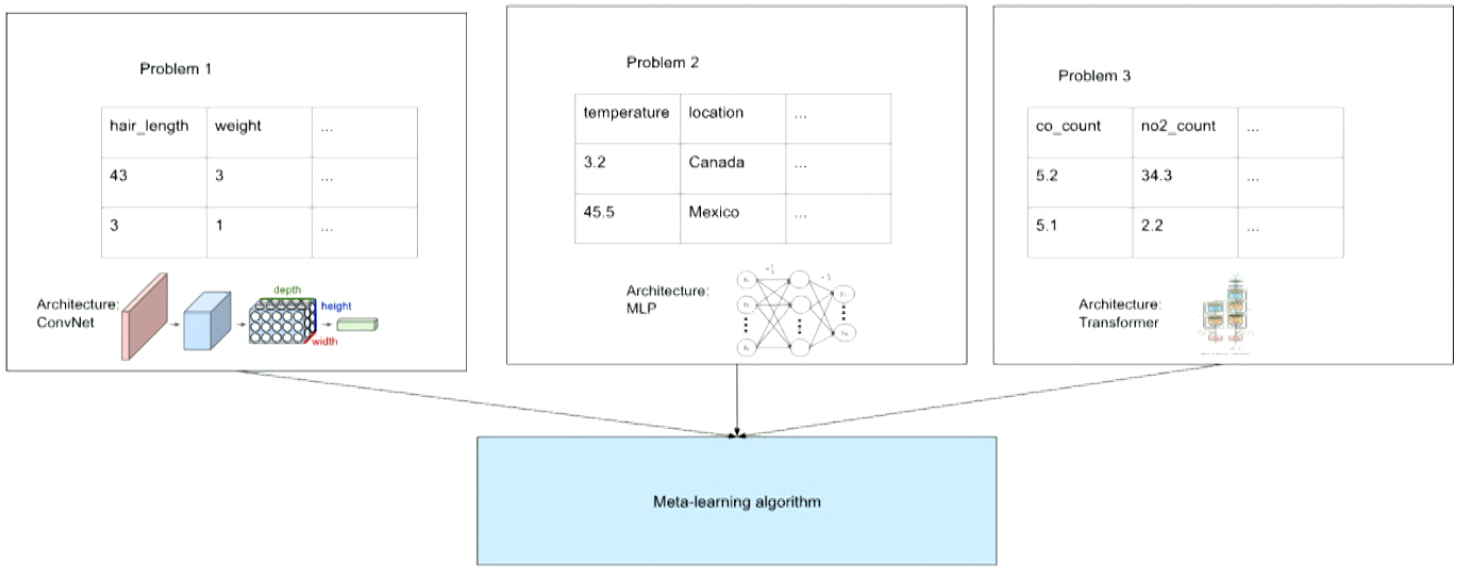
Jie Fu **Mat Kallada**

Mila

Meta-Learning

Our **training data** are machine learning problems.

Generally speaking, let's learn *something interesting* over these problems.



Meta-Learning

Definition: Automatically learning a mathematical device from **example problem instances**.

Very general framework. Under this paradigm, we can learn anything!

- An optimizer
- A loss function
- A data generation scheme

We are now at the inflection point where meta-learning is becoming practical.

The Most Basic Meta-Learning Problem: Algorithm Selection

Goal: Given a machine learning problem, predict which machine learning algorithm will perform the best.

Step 1. First, extract features from each ML problem instance.

n_features (integer feature)	**algorithm (categorical feature)**	**average_stdev_of_features (continuous)**	**accuracy (Label)**
4	“random_forests”	4.2	95%
50	“knn”	1.1	13%
2	“svm”	1.5	6%

Step 2. Build regression model to predict accuracy of the algorithm.

Predicts which algorithm gives the highest accuracy, without running them

Algorithm Selection Literature has advanced methods for this (Vanschoren et al.)

Name	Formula	Rationale	Variants
Nr instances	n	Speed, Scalability (Michie et al., 1994)	p/n, $log(n)$, log(n/p)
Nr features	p	Curse of dimensionality (Michie et al., 1994)	$log(p)$, % categorical
Nr classes	c	Complexity, imbalance (Michie et al., 1994)	ratio min/maj class
Nr missing values	m	Imputation effects (Kalousis, 2002)	% missing
Nr outliers	o	Data noisiness (Rousseeuw and Hubert, 2011)	o/n
Skewness	$\frac{E(X-\mu_X)^3}{\sigma_X^3}$	Feature normality (Michie et al., 1994)	min,max,μ,σ,q_1, q_3
Kurtosis	$\frac{E(X-\mu_X)^4}{\sigma_X^4}$	Feature normality (Michie et al., 1994)	min,max,μ,σ,q_1, q_3
Correlation	$\rho_{X_1X_2}$	Feature interdependence (Michie et al., 1994)	min,max,μ,σ,ρ_{XY}
Covariance	$cov_{X_1X_2}$	Feature interdependence (Michie et al., 1994)	min,max,μ,σ,cov_{XY}
Concentration	$\tau_{X_1X_2}$	Feature interdependence (Kalousis and Hilario, 2001)	min,max,μ,σ,τ_{XY}
Sparsity	sparsity(X)	Degree of discreteness (Salama et al., 2013)	min,max,μ,σ
Gravity	gravity(X)	Inter-class dispersion (Ali and Smith-Miles, 2006a)	
ANOVA p-value	$p_{val_{\mathbf{X}_1X_2}}$	Feature redundancy (Kalousis, 2002)	$p_{val_{XY}}$ (Soares et al., 2004)
Coeff. of variation	$\frac{\sigma_Y}{\mu_Y}$	Variation in target (Soares et al., 2004)	
PCA ρ_{λ_1}	$\sqrt{\frac{\lambda_1}{1+\lambda_1}}$	Variance in first PC (Michie et al., 1994)	$\frac{\lambda_1}{\sum_i \lambda_i}$(Michie et al., 1994)
PCA skewness		Skewness of first PC (Feurer et al., 2014)	PCA kurtosis
PCA 95%	$\frac{dim_{95\%var}}{p}$	Intrinsic dimensionality (Bardenet et al., 2013)	
Class probability	$P(\mathbf{C})$	Class distribution (Michie et al., 1994)	min,max,μ,σ
Class entropy	$H(\mathbf{C})$	Class imbalance (Michie et al., 1994)	
Norm. entropy	$\frac{H(\mathbf{X})}{log_2 n}$	Feature informativeness (Castiello et al., 2005)	min,max,μ,σ
Mutual inform.	$MI(\mathbf{C},\mathbf{X})$	Feature importance (Michie et al., 1994)	min,max,μ,σ
Uncertainty coeff.	$\frac{MI(\mathbf{C},\mathbf{X})}{H(\mathbf{C})}$	Feature importance (Agresti, 2002)	min,max,μ,σ
Equiv. nr. feats	$\frac{H(C)}{MI(C,X)}$	Intrinsic dimensionality (Michie et al., 1994)	
Noise-signal ratio	$\frac{H(X)-MI(C,X)}{MI(C,X)}$	Noisiness of data (Michie et al., 1994)	
Fisher's discrimin.	$\frac{(\mu_{c1}-\mu_{c2})^2}{\sigma_{c1}^2-\sigma_{c2}^2}$	Separability classes c_1, c_2 (Ho and Basu, 2002)	See Ho:2002
Volume of overlap		Class distribution overlap (Ho and Basu, 2002)	See Ho and Basu (2002)
Concept variation		Task complexity (Vilalta and Drissi, 2002)	See Vilalta (1999)

In Meta-Learning

Meta-Learning		**A Classical Counterpart**
Learned Distance Metric		Euclidean Distance
Learned Initializer		Glorot Initialization
Learned Optimizer		Adam
Learned Synthetic Data	**Replaces →**	Upsampling
Learned HPO		RandomSearch
Learned Sampler		MCMC
Learned Loss Functions		Cross Entropy Loss
Learned Architecture Search		A hand-designed MLP

Example Use Case. We'll learn a very good distance metric and give it to the world, so, they can benefit from it by plugging it into their K-Means clustering technique.

Behaviour is governed by input problems that you've supplied

Learned device **captures commonalities** between the supplied input problems

Device is expect to work on problems of ***similar*** nature as the ones supplied.

The more diverse the distribution of input problems that you supply, the more likely we'll learn some sort-of general purpose device that can work on basically any problem.

Like any machine learning system, the training data dictates how well it will perform. So we need to curate it well.

Outline of Talk

- A short survey of main meta-learning approaches found today
- Discussion about Learned Optimizers
- Meta-learning in Quantum Computing: Learned Optimizers for Variational Circuits

Some Meta-Learning Algorithms

Only consider meta-learning algorithms which helps us solve **a new machine learning problem**.

Some improvement to a metric below:

- Less compute
- With Less data
- Higher Accuracy

There are other meta-learning algorithms, but, we won't discuss them in detail. (E.g. Algorithm Selection)

Familiar Paradigm: Fine-Tuning

This was arguably the main meta-learning paradigm in the pre-modern meta-learning era.

Limitations:

- You can't fine-tune a music model to solve a text classification problem.
- You can't perform "dual" fine-tuning (fine tune a model with two sets of initial weights).
- Note: Some people don't consider classical fine tuning to be "meta-learning". I think it is.
- It's a acute version of MAML where the number of episodes is 1.

The embedding is the mathematical device of interest.

Allows us to perform representational transfer.

Considering these limitations, what is a good meta-learning framework

A Checklist of Good Traits (Part 1)

1. Can have multiple input problems. N>1 (Multi-source)
2. Can transfer a lot of information from known problem instances to query problem instance. (Transfer efficiency)

A Checklist of Good Traits (Part 2)

1. Generalizes to out of task distribution (Domain-General, Extrapolation)
2. Use Less Training Data to Learn Same Model (Low Data Regime)
3. Computationally Efficient during Meta-Training
4. Computationally Efficient when exposed to query problem

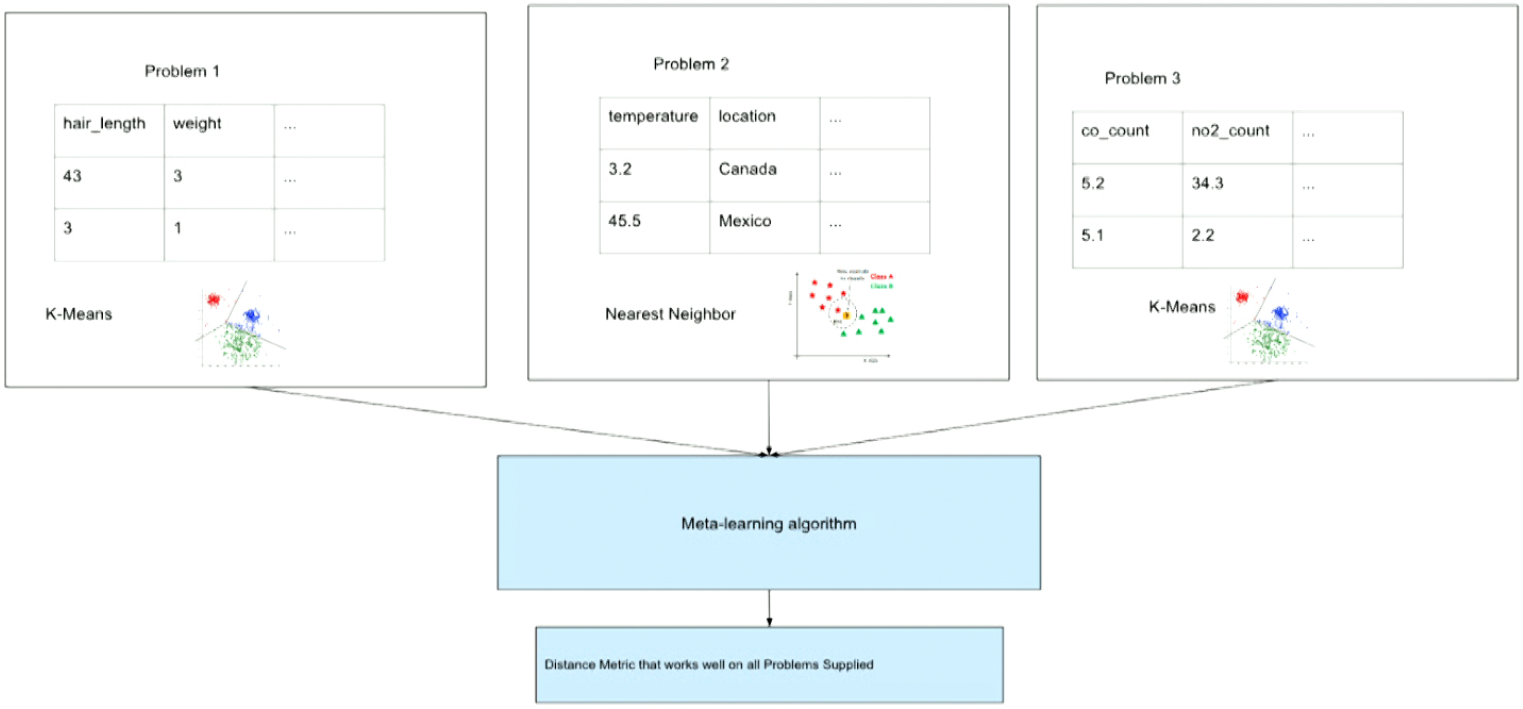
Learning a Distance Metric

Goal: Build a superior distance metric that others can use to plug into their metric-based classifiers.

E.g. If all your input problems are genomics problems, can we infer better measures of genomic similarity that can work for other genomic problems?

Reminder of Setup

Trying to infer a distance metric that works well across a variety of problems which incorporate this metric within their sub-routines.



Example Implementation: Prototypical Networks (Snell et al.)

learned_metric = random_function()

classifier = NearestNeighbors(metric=**learned_metric**, type="centroid")
classifier.fit(train_features, train_labels)
predicted = classifier.predict(test_features, activation="softmax")

loss = cross_entropy_loss(predicted, test_labels)
meta-gradients = ∇ loss wrt learned_metric
learned_metric += **learned_metric** + 0.01 * meta-gradients

How you parameterize the metric is up to you. In the original paper it looked something like:

learned_metric(x1, x2) = Euclidean(hidden_layer(x1), hidden_layer(x2))

Prototypical Networks

Introduced specifically for the application of **few-shot learning**. That is, induces priors into the metric such that it performs well on few-shot learning.

But can be extended to encode whatever priors you want depending on your meta-validation set.

In a sense, we're backproping through a distance-based procedure into the parameterized metric.

Learned Optimizers

Goal: Learn a Regression Model such that given gradients as a feature vector, predict the delta update.

Hopefully the learned optimizer will understand how to navigate a similar loss landscape.

- Navigate the loss landscape quicker
- Find a higher quality optima compared to a vanilla algorithm

Training Algorithm: Unrolled Optimization (Bengio et al.)

learned_optimizer = random_function()

# Use the optimizer just like any other optimizer! (e.g. Instead of Adam, use ours)
for _ in range(100):
    loss = cross_entropy_loss(classifier.predicted, labels)
    grads = ∇ loss wrt classifier.parameters
    classifier.parameters = classifier.parameters - **learned_optimizer(grads)**

# Depending on how it went, improve the learned optimizer
last_loss = loss
meta-gradients = ∇ last_loss wrt learned_optimizer.parameters
learned_optimizer.parameters += **learned_optimizer.parameters** + 0.01 * meta-gradients

- Issue: Differentiating through an long optimization trajectory. (Hessian)
  - This becomes very slow
- Can run multiple tasks in parallel, compute multiple meta-gradients at same time, and average them (batching).

Multi-Source Fine Tuning

Learned Optimizers are capable of “**multi-source** fine tuning”.

Multi-source fine tuning: Leveraging the problem characteristics of more than one problem when fine tuning.

Normal fine tuning only leverages one problem during fine tuning.

Important: Craft your **meta-loss** to encourage the desired behaviour of the device.

E.g. Learned optimizer that finds models that are defensible to adversarial examples.

learned_optimizer = random_function()

```
# Use the optimizer just like any other optimizer! (e.g. Instead of Adam, use ours)
for _ in range(100):
        loss = cross_entropy_loss(classifier.predicted, labels)
        grads = ∇ loss wrt classifier.parameters
        classifier.parameters = classifier.parameters - learned_optimizer(grads)

# The Optimizer does not have access to this information but we are evaluating it on this criteria
meta-loss = classifier.loss(adversarial_features, adversarial_labels)

# Depending on how it went, improve the learned optimizer
meta-gradients = ∇ meta-loss wrt learned_optimizer.parameters
learned_optimizer += learned_optimizer + 0.01 * meta-gradients
```

Bilevel Optimization Framework

We can use the following setup as a guidepost for developing meta-learning algorithms.

1. Parameterize some mathematical device of interest.
2. Observe the performance of that statistical device on a **new problem**
3. Compute the **Gradient** of the Mathematical Device's parameters

    With respect to the performance

Learning Loss Functions

A loss function is a device which computes how well a model is doing.

For example, by exposing a large quantity of problems, we may want to infer a domain-specific loss metric that properly penalizes **predicted=dogs, actual=wolf** is better than an off-the-shelf loss metric.

An ML engineer would take long to encode this type of ontological knowledge to hand build this loss function.

Potentially we could give this to people and they could train their models with this loss function.

This is capable of learning problem-specific regularization strategies.

Gradient-Based Loss Search (Bechtle et al.)

```
learned_loss = initialize_loss()

# Run a test with our learned loss.
classifier_params = glorot_initialization()
for _ in range(iters):
        loss = learned_loss(classifier.predict, labels)
        grad = ∇ classifier_params wrt loss

        classifier_params += classifier_params + 0.01 * grad

meta_loss = cross_entropy_loss(classifier.predict, labels)
meta-gradients = ∇ meta_loss wrt learned_loss.parameters
learned_loss.parameters = learned_loss.parameters - meta-gradients * 0.01
```

Evolutionary Loss Search (Houthooft et al.)

1. Parameterize loss function
2. Run a test run with gradient descent using the learned loss
3. Update the loss function parameters using evolution strategies

Only difference: Instead of using gradient descent on the **outer optimization problem**, they propose using Evolution Strategies.

In general, we can solve the outer problem (the parameterization of the device) with any algorithm we want.

Learned Initializations

Goal: The device of interest is a starting **initialization weights** such that we can give these weights to someone, so they can fine tune for a new task in the same domain.

That is, we want to create pre-trained weights over several datasets.

With one dataset, it's easy (classic fine tuning). But how do you do it with several?

There are companies that sell pre-trained weights as a service. Some use meta-learning to learn these pre-trained weights.

MAML

learned_initial_weights = random_function()

```
# Fine tune using these initial weights.
classifier.weights = learned_initial_weights
for _ in range(100):
        loss = cross_entropy_loss(classifier.predicted, labels)
        grads = ∇ loss wrt classifier.parameters
        classifier.parameters = classifier.parameters - 0.01*grads

# Depending on how it went, improve the initial weights
meta-gradients = ∇ loss wrt learned_initial_weights
learned_initial_weights += learned_initial_weights + 0.01 * meta-gradients
```

Differentiating through an optimization trajectory. (Hessian)
Can run multiple updates in parallel. (Batching)

Similarity between MAML and Learned Optimizers

In MAML, the meta-parameters are the initial weights. In learned optimizers, the meta-parameters are the weights of the optimizer.

The training algorithm is the same (unrolled optimization).

In addition to learned losses, LO and MAML are both considered **optimization-based meta-learning methods** because they backprop through an optimization process in the meta-training algorithm.

The learned optimizer can “learn” to pre-train the optimizee. It just needs to hop to the initial weights at the first iteration.

- Finn et al. proved that a learned initialization paired with vanilla SGD can approximate any learned optimizer.

Meta-Learned Samplers (Liu et al.)

Goal: Sample from some target, potentially intractable, distribution.

Approach:

- You have a fixed number of samples that approximates a target distribution.
- Keep moving those samples until they more accurately depict the posterior or target distribution.

Computing the Stein Variational Gradient is expensive.
Construct a “Neural Sampler”

We can train a sampler parameterized by a neural network which uses SVGD as a **supervision signal**.

That is, this neural network should mimic the behaviour of the SVGD sampling method by taking in the raw sample as input and predicting how to move it.

As the neural sampler is exposed to more problems, probabilistic inference becomes computationally more efficient on new problems.
Idea: Give your sampler to other people once you train it on enough problems.

Learnable Hyperparameter Search (Feurer et al., 2018)

Goal: Learn a hyperparameter search strategy and then give it away to people so they can tune models quicker.

The device we are learning is a hyper-parameter proposer.

We're given a database of existing hyper-parameter runs. These are our problem instances which we've observed.

Proposed clever way to combine GPs from existing runs. Helps define probability distribution over hyper-parameters to try next.

Learnable acquisition functions using RL were proposed by Volpp et al.

Note on Wording

Note: Fancy hyper-parameter schemes like the NEAT algorithm family is technically **not** meta-learning

There is no *learning* over input problems.

But, a scheme like Bayesian HPO or RL-based NAS **is** meta-learning.

Learnable Hyperparameter Search (Feurer et al., 2018)

Goal: Learn a hyperparameter search strategy and then give it away to people so they can tune models quicker.

The device we are learning is a hyper-parameter proposer.

We're given a database of existing hyper-parameter runs. These are our problem instances which we've observed.

Proposed clever way to combine GPs from existing runs. Helps define probability distribution over hyper-parameters to try next.

Learnable acquisition functions using RL were proposed by Volpp et al.

Related: NAS.

Comparison Matrix - General Sentiments

	Fine Tuning	**Learned Losses (Houthooft et al.)**	**Learned Optimizers (Andrychowicz et al.)**	**Learned Initializations (Finn et al.)**	**Learned Samplers (Liu et al.)**	**Learned Hyper-parameter Search (Feurer et al.)**	**Learned Metrics (Snell et al.)**
Out of Task Generalization (within reason)	Poor	Great (Houthooft et al.)	Usually Poor (Andrychowicz et al)	Good	Unknown?	Good	Unknown?
Multi-Source	X	Check	Check	Check	Check	Check	Check
Heterogenous	X	Check	Check	X	X	Check	Usually not.

Comparison Matrix - General Sentiments

	Fine Tuning	**Learned Losses (Houthooft et al.)**	**Learned Optimizers (Andrychowicz et al.)**	**Learned Initializations (Finn et al.)**	**Learned Samplers (Liu et al.)**	**Learned Hyper-parameter Search (Feurer et al.)**	**Learned Metrics (Snell et al.)**
Transfer efficiency	Great	Great	Poor in most formulations	Good	Good (transfers compute usage)	Good	Good
Fast Meta-Training	X	X	X	X	X	Check	Check
Fast Meta-Testing	Check	Check	Check	Check	Check	Check	Check
Performs in Low Data Regime	Check	Check	Check	Check	X	X	Great

You can meta-learn **anything**! Some esoteric ones

- Meta-learn when to stop an iterative search procedure (so you don't waste time).
- Meta-learn which data points are most important to learn from.
- Meta-learn an model to be robust against adversarial examples
- Meta-learn how to correct for drift in your model.

Meta-learning can replace some of the task that algorithm designers and mathematicians.

The core of most meta-learning papers can be boiled down to like ~10-30 LOC. Why not start playing around and see what you can do with it?

Formulating Meta-Learning Algorithms

In meta-learning, the real question here is: What *can't* we learn?

In the worst case, we have to use some classical program induction (e.g. Bello et al. who used a symbolic search for optimizer search).

Think:

> "We have some algorithm that does X. Can we *learn* a **domain-specific** version of that algorithm. Not a vanilla one-size fits all version."

E.g. A problem-specific version of a solver for the travelling salesman problem that works exceptionally well on railroad problems. Learns problem-level heuristics.

A learned optimizer for NNs that realizes that flat valleys lead to higher quality optima. NNs are a problem class and we can exploit unique properties of them.

Trade-offs in Meta-Learned Algorithms

Usually some trade-off between the following:

- Domain Specificity (What input problems should we support?)
- Accuracy
- Computational Resource used by learned device

Regarding the domain specificity: Is true “extrapolation” possible in meta-learning? We can’t even do that in normal machine learning. That is, is it realistic to learn universal rules with meta-learning?

“Vanilla One-Size Fits All” Algorithms

There is a performance hit for attempting to do well at all possible input cases.

Virtually classical methods today try to work well on all problems.

Sure, there’s No Free Lunch, but:

> If your algorithm will NEVER see these input cases, why are we taking a performance hit for them?

While there are inductive biases in SGD, an algorithm like SGD is too general IMO, it is close to the vanilla end of the spectrum

The Promise of Learned Optimizers

Regression Model: Given gradients as a feature vector, predict the delta update.

Algorithm 1 General structure of optimization algorithms

Require: Objective function f

$x^{(0)} \leftarrow$ random point in the domain of f

for $i = 1, 2, \ldots$ **do**

  $\Delta x \leftarrow \phi(\{x^{(j)}, f(x^{(j)}), \nabla f(x^{(j)})\}_{j=0}^{i-1})$

  if stopping condition is met **then**

    return $x^{(i-1)}$

  end if

  $x^{(i)} \leftarrow x^{(i-1)} + \Delta x$

end for

Gradient Descent	$\phi(\cdot) = -\gamma \nabla f(x^{(i-1)})$
Momentum	$\phi(\cdot) = -\gamma \left(\sum_{j=0}^{i-1} \alpha^{i-1-j} \nabla f(x^{(j)}) \right)$
Learned Algorithm	$\phi(\cdot) =$ Neural Net

Image Source: “Learning to Optimize with Reinforcement Learning” - Ke Li, UC Berkley

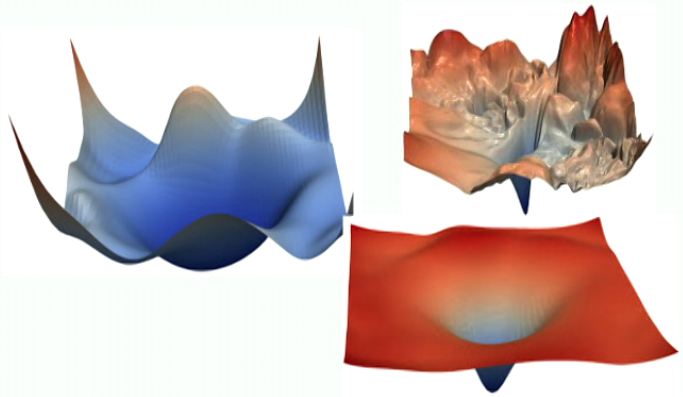
Overview of This Section

- Motivation of Learned Optimizers
- Recent Trends in Learned Optimizers
- Investigation of their applications towards Quantum

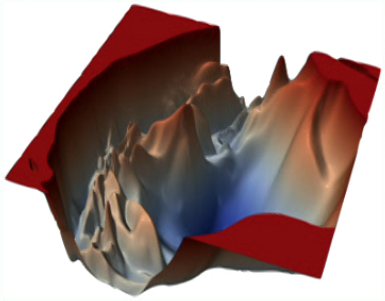
Why do this?

Optimizer will “see” a lot of loss landscapes, so it can solve new ones **faster** and with **more accuracy**.

Training Data



Solve a New Problem



Find high quality optimum in <u>minutes</u>? Opposed to weeks?

“We know how to explore these landscapes, we’ve seen them before!”

Versatility of Learned Optimizers

- Optimization is everywhere! Just plug-in a learned optimizer and it will capture correlations between problems.
- By being able to modify the weights directly, we have fine-grained control over the model, as opposed to techniques, such as Bayesian Hyper-parameter Search / Algorithm Selection, which only pick the model family.
  - This fine-grained control will allow us to get higher accuracy and faster convergence.
- Fine tuning only works on homogenous problem domains (that is, an image classification model can only be fine tuned to create another image classification model; you cannot fine tune an image classification model to create an audio recognition model).
- By encoding the meta-loss properly -- **you can implicitly re-shape the loss landscape**
  - E.g. Can learn models which are especially robust to noise
  - E.g. Can perform few shot learning
- Removes annoying hyper-parameters such as learning rate

Enabling Heterogenous Transfer

Gradient vector has size of K where K is the number of parameters in the network.

How do you apply the optimizer to a problem of different size?

Component-Wise Assumption

Each parameter gets its own learned optimizer.

Yet, all the weights of the learned optimizers are trained jointly (shared weights, like in the conv layers of a CNN)

Pretty simple, but, doesn't consider **inter-parameter correlations**.

How we normally parameterize the optimizer?

- RNNs (Either GRUs or LSTMs)
  - However, the only justification seems to be that “they look similar, so, let’s use them”.
- FNNs have been used sparingly over the past 3 years and have shown better performance.

How we normally train the optimizer?

- Previous approaches include: symbolic search, RL, evolution strategies, and of course: backprop through time (unrolled optimization).

Current Problems with Learned Optimizers

1. The RAM usage is large due to backpropagation through time.
   a. *Though, there are approximations.*
2. Empirically found that they cannot effectively solve problems outside the problem domain (compared to other meta-learning methods)
3. Transfer efficiency can be low due to the component-wise assumption.
4. Exhibits chaotic behaviour. (Small perturbations lead to drastic changes)
   a. Ke Li claims this is related to the fact that standard supervised learning assumes IID inputs which isn't the case because we have correlated inputs.
5. Short-Horizon Bias. If you learn an update rule for 10 steps, it can start acting weird after the 10th step. Other problems with horizons (exploding / vanishing gradient).

Training on small *prototypical* problems (Wichrowska et al.)

Namely: Train on a bunch of small problems but that are highly realistic in optimization scenarios.

And several other ideas such as hierarchical RNNs for the learned optimizer.

Does reasonably well in terms of performance on ImageNet. However, diverges at later stages of training.

Does not work in terms of Wall-Clock time.

Simple Idea: Use Variational Circuits as Optimizee

Again, learned optimizers are a very general framework.

Anywhere there is optimization, we can replace it with a **learned** one.

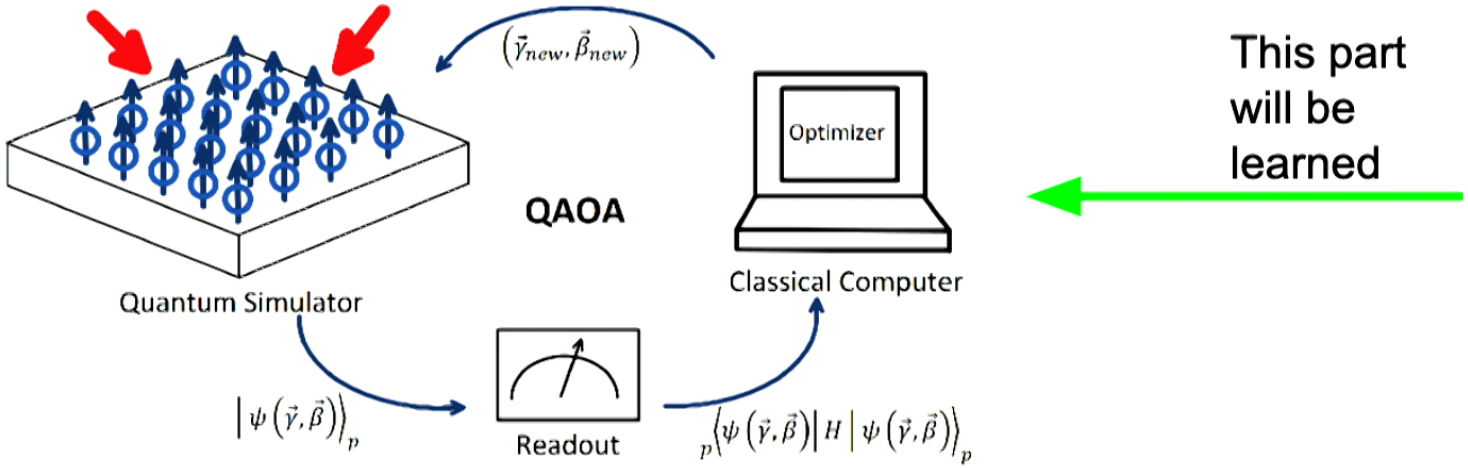


Image Source: Ho et al.

~3 Months Before Verdon et al.: A Master's Student's Thesis

Quantum machine learning

Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

Universitat de Barcelona (UB)

Universitat Rovira i Virgili (URV)

Master's Thesis

Quantum Machine Learning

Master in Artificial Intelligence

April 16, 2019

View/Open

138031.pdf (1.128Mb)

Riu I Vicente, Jordi

Tutor / director / evaluator: Cortés García, Ulises; García Sáez, Artur

Document type: Master thesis

Date: 2019-04-14

Rights access: Open Access

Abstract

We use reinforcement learning techniques to optimize the Quantum Approximate Optimization Algorithm when applied to the MaxCut problem. We explore Q-learning based techniques both for continuous and discrete action environments with regular and irregular graphs.

Subjects: Machine learning, Quantum computers, Aprenentatge automàtic, Ordinadors quàntics

URI: http://hdl.handle.net/2117/133060

~3 Months Before Verdon et al.: A Master's Student's Thesis

First application of meta-learning to variational circuits.

Suggests using RL to learn a policy for the classical loop of QAOA. (Chapter 3)

In other words, learning the optimizer with RL (Kai Li's approach to learned optimizers).

Solves the MaxCut problem with RL for the classical loop.

Prior to this work, there was also a well known (and patented) work on learning a learning rate schedule by Microsoft.

Some Additional Motivations for Learned Optimizers

- Learning to deal / navigate with Barren Plateaus in the loss landscapes
  - Where the gradients are very small in the QAOA loss landscape when #N of qubits gets larger
- Learned Optimizers can learn to be robust to the noisy measurements in variational circuits. To solve the noisy measurement issue.
  - Consider in the papr “Using learned optimizers to make models robust to input noise”.
  - Where the validation set used in meta-loss has an emphasis on measuring its performance on noise.
  - In Wichrowska, they also proposed purposely exposing optimizer to noisy inputs to the neural optimizer, so it would learn to be robust to them.

Some Additional Motivations for Learned Optimizers

- Learning to deal / navigate with Barren Plateaus in the loss landscapes
  - Where the gradients are very small in the QAOA loss landscape when #N of qubits gets larger
- Learned Optimizers can learn to be robust to the noisy measurements in variational circuits. To solve the noisy measurement issue.
  - Consider in the papr “Using learned optimizers to make models robust to input noise”.
  - Where the validation set used in meta-loss has an emphasis on measuring its performance on noise.
  - In Wichrowska, they also proposed purposely exposing optimizer to noisy inputs to the neural optimizer, so it would learn to be robust to them.

A properly trained learned optimizer will learn to excel despite all these issues.

QAOA Ansatz for VQE

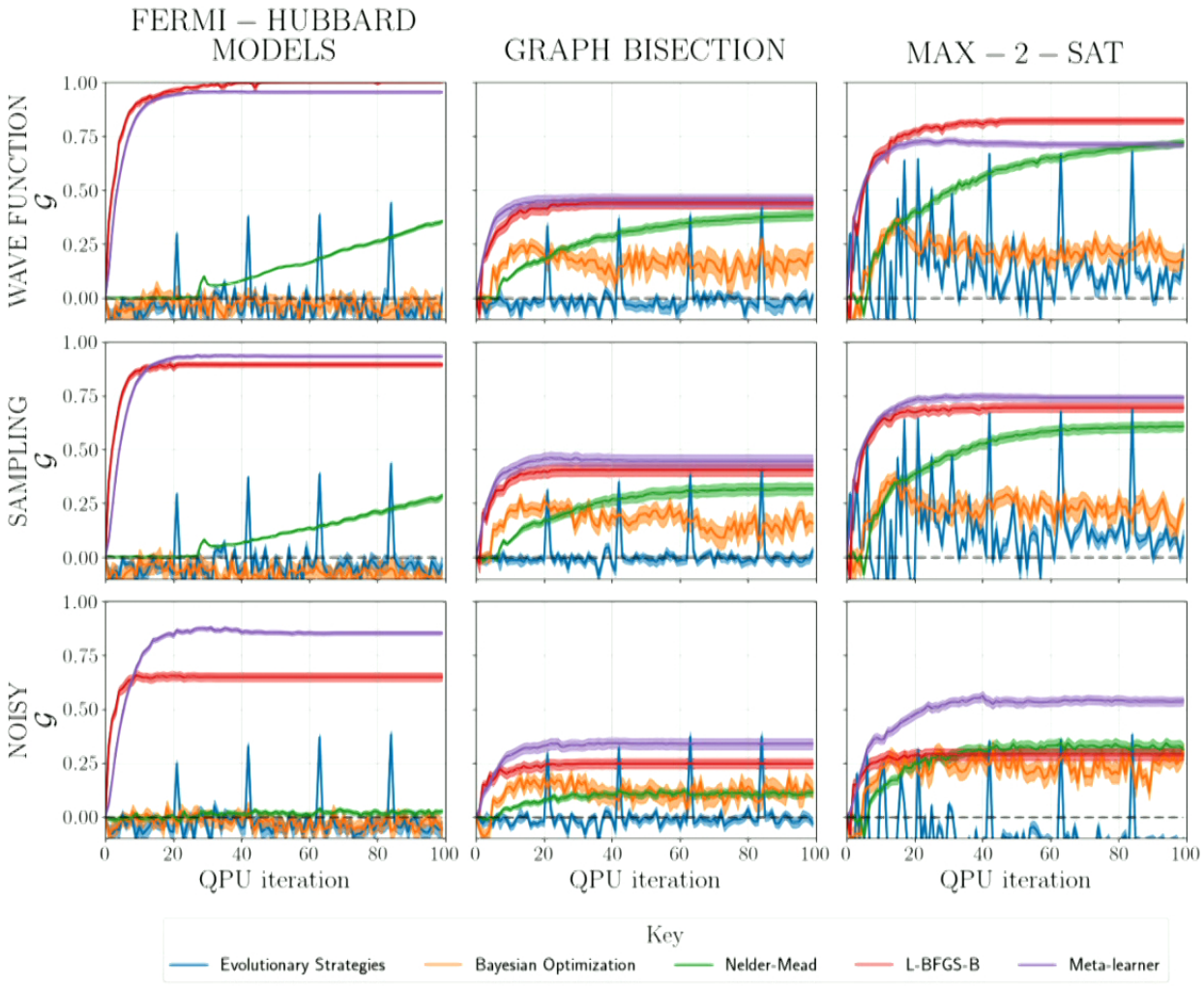
Problem setup: Figure out a parameter vector which comprises of the beta and gamma parameters to minimize the expectation of a variational circuit.

Various methods for computing the gradient of the parameter vector with respect to the expectation (e.g. parameter shift rule)

General Setup for QAOA with Learnable Optimizers (Wilson)

- Neural optimizer parameterized by LSTM
- LSTM takes Gradient as input (which Verdon does not do).
- Uses original input pre-processing scheme proposed by DeepMind
  - Not computationally efficient. Augments parameter size by ~2x.
- Meta-loss is the weighted sum of the losses of bast optimizee
  - Weights are hyper-parameters.

- Reported performance in iterations.
  - The meta-learner does well on all problems except one. Should also report wall-clock time?


FERMI – HUBBARD MODELS
GRAPH BISECTION
MAX – 2 – SAT
WAVE FUNCTION
SAMPLING
NOISY
$\mathcal{G}$
1.00
0.75
0.50
0.25
0.00
0
20
40
60
80
100
QPU iteration
Key
Evolutionary Strategies
Bayesian Optimization
Nelder-Mead
L-BFGS-B
Meta-learner

General Setup for QAOA with Learned Initializations (Verdon)

- Learns an initializer where a normal optimizer such as Nelder-Mead is subsequently used.
- Parameterized by an LSTM which takes in **previous expectation value** and last parameter values. (No gradients)
  - As they mention, similar to "Learning without Gradient Descent by Gradient Descent".
  - *"Since precise gradient estimations on NISQ devices are hampered by the large number of runs required and by the noise of the device, ….."*
- Does not use gradients as input to LSTM
- Transfers from small problems on simulator to large problems on real-world quantum hardware. (Similar to Wichrowska)
- Loss function for rewarding optimizers which make *improvements in lowest loss*
- After RNN is ran for 10 unroll steps (to avoid short-horizon bias), leave it to the normal optimizer to do the rest. **Initializes in good basin of attraction.**

Observed Improvement Loss Function

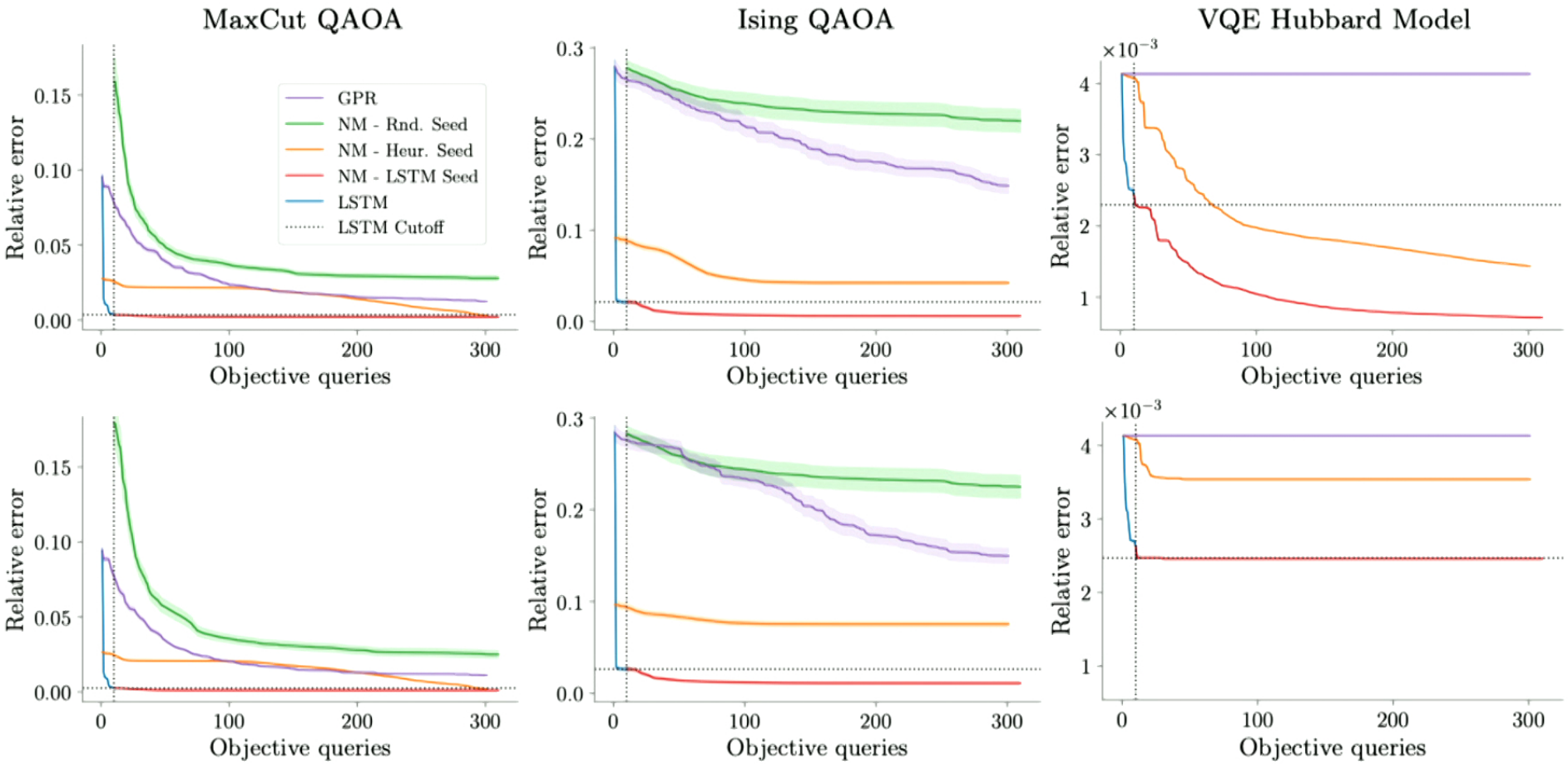
Observed Improvement Loss: **Reward optimizers that keeps make significant improvements from the last lowest loss we saw.**

Rewards optimizers to keep getting better and exploring new optimum.

New loss function that wasn't seen in existing meta-learning literature.

Looks like this:

min(current_loss - lowest_loss_so_far) summed for all losses.


MaxCut QAOA
Ising QAOA
VQE Hubbard Model
GPR
NM - Rnd. Seed
NM - Heur. Seed
NM - LSTM Seed
LSTM
LSTM Cutoff
Relative error
Objective queries
$\times 10^{-3}$

Some notes on launching a spin-off from academia

- Economic Moats for Algorithm Companies
- Other spin-off tips

Moats for Algorithm-Based Companies

In this framework, future revenue is a function of the following factors.

- Network Effect (Machine Learning Companies -- only logarithmic)
- Trade Secrets (Google Algorithm)
- Brand (Stripe)
- Patents (Qualcomm)
- Vendor Lock (Oracle)
- Scale Effects (AWS)

Company Valuation: ~10 times revenue.

Idea: It is *usually* not sufficient to create an algorithm that runs

K times faster, without complementary moats.

Usually, that is...

The Forces of Innovation

Discussion with Mike Errecart -- CEO, OldSchoolValue.

Product Quality Alone Is Not Sustainable

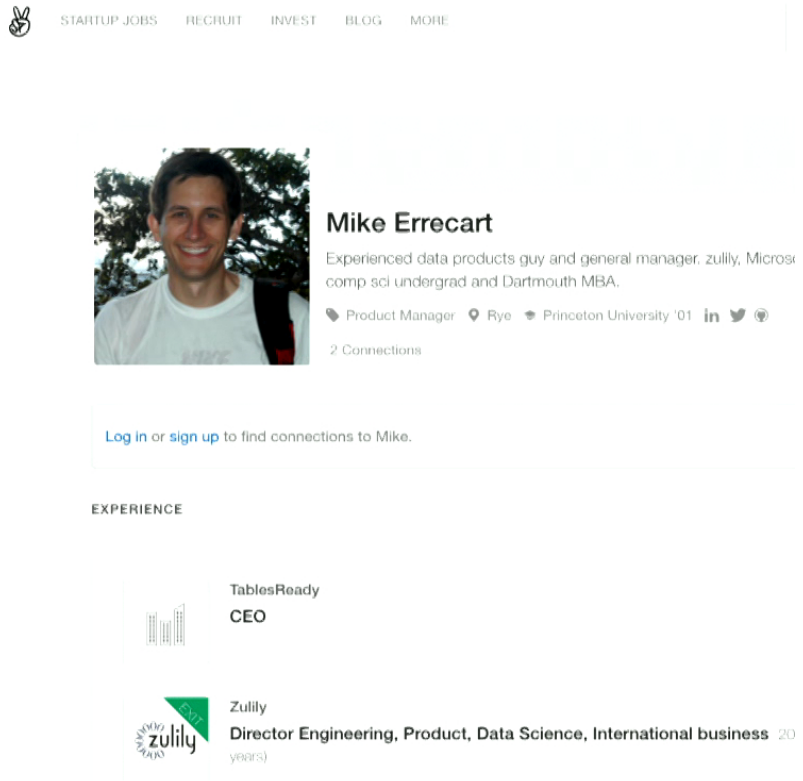
So, does a high product quality-to-cost switching cost strategy do that?

Often not.

In Dorsey's example, a superior lubricant is not going to get used in more places over time. It's possible the superior lubricating technology can be used in related products that could find new applications in a customer's business. Really, though, as soon as a better product comes along, any and all could be switched out.

There's also clearly no network effect inherent in using a better product like this.

So, is a product quality-based switching costs moat worse than an integration- and network effects-based one?



Entire Discussion available online on Mike's blog.

Let's return to Dorsey's definition of a switching cost: "The cost of switching to a competitor outweighs the cost or product benefits of a new and better product."

I think this is a good definition, and I think product quality meets this bar.

Is Product Quality a Sustainable Competitive Advantage?

Mat had an interesting take on that:

> *I find it interesting that a vanilla product quality-based switching cost does not get bigger as the user continues to use it (unlike, a traditional switching cost — e.g., the Oracle database).*
>
> *The example that Dorsey gives for the product quality-based switching cost is the*

“Unoptimizing”: Don’t be a hero.

Theory: People who over-optimize on tech stack suffer from imposter syndrome -- NOT because they love the technology.

Thanks! Questions?

Email: kallada@cs.dal.ca