Machine Learning in the Cloud

Eric Meissner and Carl Henrik Ek

November 29, 2020

Abstract

In this worksheet we will focus on the task of how we can run machine learning models in the cloud. Many of the methods that have seen so far are quite expensive to run. A simple Bayesian Optimisation loop is OK to run for low-dimensional problems but as soon as the dimensionality increases the computational cost goes up very quickly. To circumvent this problem we will show how we can use external computational resources and run our computations in the cloud. We will also introduce a couple of software packages and additional infrastructure that you will need.

Comment

This worksheet will be slightly different compared to the previous ones. You have now seen material in this course and others and it is up to you to reflect how well they fit your research goals. We will provide an initial path towards how you can think about doing machine learning in the cloud motivated by some of the methods that we have discussed. We hope that this will work as a starting point for you and then you can test out your knowledge on something specific that is more in-line with your specific research interest.

1 Infrastructure

The idea of this worksheet is to teach you the infrastructure that is needed to run cloud computations. The key components that we will use is `Amazon Web Services` (AWS) as the service for the computation. There are many others one could use as well but I believe that AWS is both the easiest to use and the most commonly adopted among machine learning practitioners. The second thing we are going to need is some form of virtualisation that specifies the computational instance that we want to run. Again there are many ways of doing this but by far the most adopted one is `Docker`. One of the real benefits of this virtualisation is that it is light-weight and importantly it has a fantastic documentation.

Rather than going through these structural aspects in this document we have prepared a `git` repository that includes a `Jupyter Notebook` that goes through all the steps that is involved with setting up the computational instance. So first thing that you should do is to `pull` this repository `https://github.com/meissnereric/cirrocumulus`. Browse around this repository as it should include most of the structural code that you are going to need to test everything.

2 Bayesian Optimisation

With the creation of large open data-sets machine learning methods with very little regularisation have become applicable. These databases have reframed the machine learning question from being how to input knowledge into our model to how we can use all the availible data. So we can trade-off our ignorance of a problem with massive amounts of data. The downside to this is that with an increase in data also comes increased computational demands. These computational demands do not come for free and come with a significant environmental impact. In a recent paper Strubell et al., 2019 it was estimated that training some

of the biggest natural language models releases the equivalent of five times the CO2 as what a car would do during its entire lifetime. This is of course highly problematic and we need to come up with more efficient ways of training these models.

Earlier in the course we looked at what we called the Pareto-front of machine learning, how we can trade-off knowledge with data. The neural network models that underlies these computationally expensive models are far on the data side of that graph, not being able to exploit knowledge they rely nearly completely on data to regularise the solution space. However, for the problems they are able to address the volume of data is still not nearly sufficient to make the problem well posed[1]. So how can they still learn something sensible? For neural networks this comes into how the training procedure is done. These methods are optimised using gradient based methods such as Kingma et al., 2014. These methods define specific heuristics to explore a parameter space, using different methods we will choose different solutions. So in effect this is where we aim to try and circumvent the "No Free Lunch" Shalev-Shwartz et al., 2014 by introducing knowledge in how we search for the minima rather than regulate the function directly. This might sound counter intuitive, and it is in many ways hard to understand why we are doing certain things but in practice this has been shown to work really well. However, this being said what is often used in practice to make this approach less computationally expensive is to build a surrogate model over the objective function and use a Bayesian optimisation loop to search for some of the parameters that defines the search procedure Snoek et al., 2012.

Before we move to neural networks let us take a look at the traditional learning loop in gradient based optimisation. We have an objective function $f(\theta, \mathcal{D})$ that takes a set of parameters θ that we wish to learn and the training data $\mathcal{D}$. Now in order to find the parameters $\hat{\theta}$ that maximise the objective function we search using a gradient based strategy. This means that given the current estimate θ_{t-1} we take a step in the direction of the steepest gradient to get the next estimate θ_t. Now the question is how large step along the gradient direction should we take? This is controlled by a parameter referred to as the *learning rate* η. This leads to the following update rule,

$$\theta_t = \theta_{t-1} + \eta \frac{\partial}{\partial \theta} f(\theta, \mathcal{D}). \tag{1}$$

Now it turns out that setting the learning rate is very challenging, choosing it too small we will waste computational resources as our update rule becomes too restrictive while setting it too high mean that we might miss important structures and simply "overshoot" parts of the optimisation surface. Now this is something we could use Bayesian Optimisation to address, we could try to come up with a structure so that we search for the best learning rate for a specific problem. This means that each evaluation of the objective function means training the neural network, this being very expensive we really want to do as few evaluations as possible and this we can do by using a Bayesian Optimisation loop.

Before we take on the task of learning the learning rate for a full blown neural network lets make sure that we get the principles right by taking a simpler model where each evaluation is less expensive. We will now define a simple neural network binary classifier, a so called *Multi Layered Perceptron*[2]Rumelhart et al., 1986. We will use the implementation from the `scikit learn`[3] toolbox where you can find lots of well documented classic machine learning algorithms. There is a nice evaluation of different learning rates here that you can have a look at.

We will now first build up the model and create a data-set for classification. We will use the built in data-set generator in `scikit learn` called `make_classification`. We will from this generate a training and a test data-set using `train_test_split`. When we have generated the data the important part is building the algorithm and importantly the setting for the gradient based optimiser. We can build the Multi Layered Perceptron using `MLPClassifier`. This is the important part as when we create the object we also set the training procedure. We are going to use the Adam optimiser Kingma et al., 2014 in this example and as you can see from the documentation there are several parameters that we could tune. In this case we will use the standard settings for everything but the learning rate.

[1] this would require infinite volumes of data.
[2] Medium Post on MLP
[3] `https://scikit-learn.org/stable/index.html`

Code

```
clf = MLPClassifier(random_state=1, max_iter=1000,
                    activation='logistic',
                    hidden_layer_sizes=(50,),
                    solver='adam', learning_rate='constant',
                    momentum=0,
                    warm_start=False,
                    learning_rate_init=learning_rate_init
                    )
```

The code above creates a neural network with 50 nodes in the hidden layer and a single output node corresponding to the class variable. The activation function is chosen to be the `logisitc` function. There are many heuristics to control the learning rate and make it adaptive. By setting `learning_rate='constant'` we fix it so that it remains at the initial value through the iterations. The code below will perform a grid-search over the learning rate which generates Figure 1.

Code

```
from sklearn.neural_network import MLPClassifier
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import numpy as np


def f(learning_rate_init, X_train, y_train,X_test,y_test):

    # Create Classifier
    clf = MLPClassifier(random_state=1, max_iter=1000,
                        activation='logistic',
                        hidden_layer_sizes=(50,),
                        solver='adam', learning_rate='constant',
                        momentum=0,
                        warm_start=False,
                        learning_rate_init=learning_rate_init
                        )

    # Train Classifier
    clf.fit(X_train, y_train)

    # Return error
    return -clf.score(X_train, y_train), -clf.score(X_test, y_test)

# Create Data-set
X, y = make_classification(n_samples=1000, random_state=1,
                           n_classes=2, n_clusters_per_class=2)
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y,
                                                    random_state=1)

eta = np.logspace(5, -8, base=10, num=40)

e_train = []
e_test = []
for i in range(0,eta.shape[0]):
    a, b = f(eta[i], X_train, y_train, X_test, y_test);
    e_train.append(a)
    e_test.append(b);

fig = plt.figure(figsize=(10,5))
ax = fig.add_subplot(111)
train, = ax.semilogx(eta,e_train,color='blue',linewidth=3.0)
test, = ax.semilogx(eta,e_test,color='red',linewidth=3.0)
ax.set_ylabel('Negative Accuracy')
ax.set_xlabel('Learning Rate')
train.set_label('Training')
test.set_label('Test')
ax.legend(fancybox=True, framealpha=0.0)

# IGNORE THIS
fig.tight_layout()
plt.savefig(path,transparent=True)
return path
```

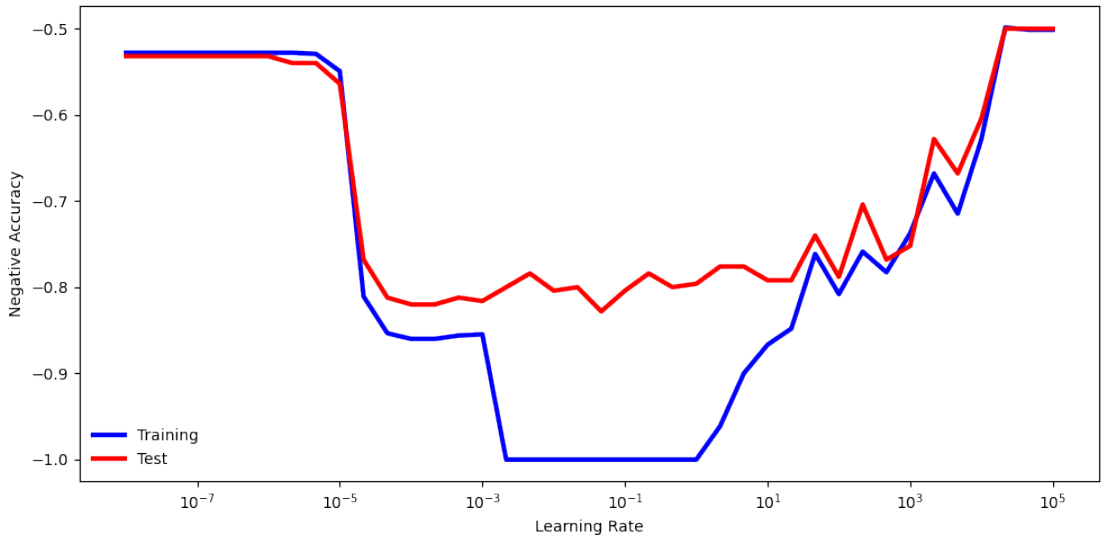

Figure 1: *The plot above shows the negative accuracy for different learning rates, the blue curve is the accuracy on the training data while the red is for the test data. As you can see, if we choose a too small learning rate (left side) we are not able to efficiently explore the space, while if the learning rate is too large (right side) we are not able to find the local structures in the objective function. The second important aspect is to look at the scaling of the axes. The* `x-axis` *is on a* `log-scale` *meaning that we need to take this into account when we formulate our prior for the search. There is a clear region where the learning rate leads to perfect performance on the training data.*

Now we have an objective function and it is time to implement a Bayesian optimisation loop to infer the learning rate. You are free to use whatever software you like for your Bayesian optimisation loop, you can use your own implementation, GPyOpt, EmuKit or any other toolbox. The important thing is to use the knowledge that we got from the plot above when we design our surrogate model. Below I will use GPyOpt to define the loop.

Code

```
import GPy
from GPyOpt.methods import BayesianOptimization
import numpy as np
from sklearn.neural_network import MLPClassifier
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

def f(learning_rate_init):
    learning_rate_init = np.atleast_2d(np.power(10,learning_rate_init))

    for i in range(0, learning_rate_init.shape[0]):
        # Create Classifier
        clf = MLPClassifier(random_state=1, max_iter=100,
                            activation='logistic',
                            hidden_layer_sizes=(50,),
                            solver='adam', learning_rate='constant',
                            momentum=0,
                            warm_start=False,
                            learning_rate_init=learning_rate_init[i,0]
                            )
        # Train Classifier
        clf.fit(X_train, y_train)

    # Return accuracy
    return np.atleast_2d(-clf.score(X_test, y_test))

np.random.seed(42)
X, y = make_classification(n_samples=1000, random_state=1,
                           n_classes=2, n_clusters_per_class=2)
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y,
                                                    random_state=1)


kernel_rbf = (GPy.kern.RBF(input_dim=1, variance=0.2, lengthscale=2.0) +
              GPy.kern.White(input_dim=1, variance=0.05))

domain = [{'name': 'Learning Rate', 'type': 'continuous', 'domain': (-5,5)}]
opt = BayesianOptimization(f=f, domain=domain, model_type='GP',
                           kernel=kernel_rbf,
                           acquisition_type = 'EI',
                           initial_design_numdata = 3,
                           )

opt.run_optimization(max_iter=30)
```

Looking at the results we can see that we have found a learning rate close to 10^{-2} which is about where we would expect from the grid-search in Figure 1. The nice thing is that we found this after 15 evaluations of the objective and looking at the exploration of the space it looks like it has quickly figured out the relevant region of the space. Now, if you would time the two loops that we did you will most likely find that the Bayesian optimisation procedure was if not more expensive at least not particularly much quicker. The

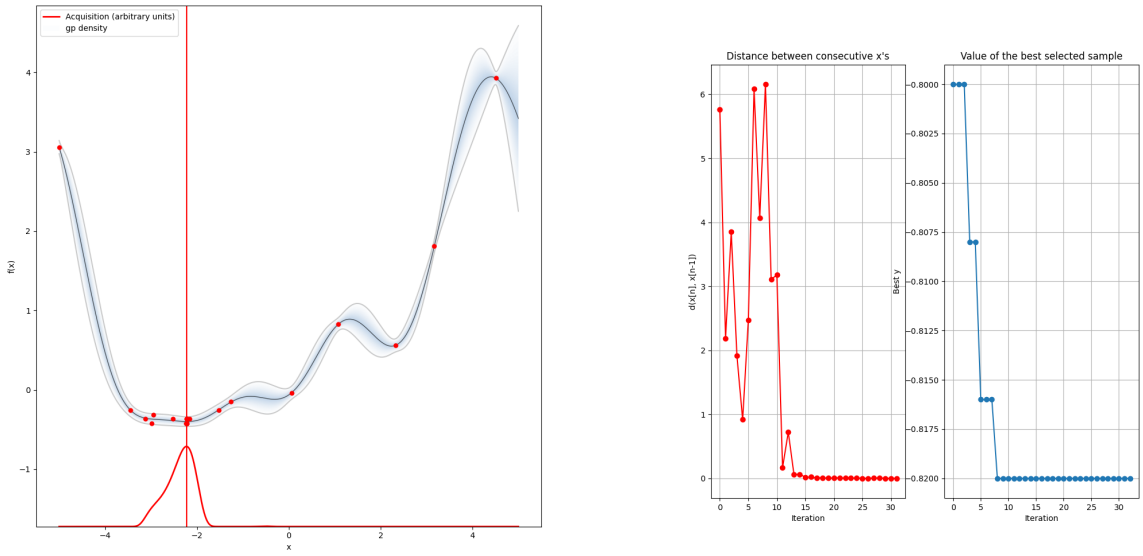

Table 1: *The figure above shows the results of applying Bayesian optimisation to learn the learning rate for the MLP classifier. The plot to the right shows the model and the acquisition function for the current iteration while the plot on the right shows how large the difference is between consequtive points that are evaluated and how the optimal value converges. As you can see the loop has converged after less than 15 iterations. Notice that the plot here is not on a log-scale as I used the internal plotting of GPyOpt.*

important thing to remember here is that the cost of the outer-loop will be equivalent even if you change the cost of the inner-loop. So if you pick a very expensive model in the inner-loop saving just a couple of iterations of the outer-loop will make a massive difference.

Now you have the loop set-up there are additional things you can do as well, rather than optimising the accuracy you can optimise the `loss` you can get by accessing the attribute `clf.loss_`. In this case we have used a very simple data-set try to pick something more complicated which allows you to do a bit more qualitative evaluations of the results. Other things that you can try is to see if you can run less iterations, maybe you do not have to run the algorithm to convergence at each step, this will save you a lot of computations in the inner-loop. There are plenty of methods inside `scikit learn` that you can use a very similar loop for so feel free to test other algorithms as well. When you feel like you have an understanding of how this works you can now move onto to picking a more complicated model that might be more tedious to run on your own machine.

2.1 Neural Network

Now when we have the infrastructure set-up for running the Bayesian optimisation loop it is time to look at a more computationally expensive algorithm to train rather than the simple classifier we used. To do so we will train a Convolutional Neural Network LeCun et al., 1995 and train an image classifier on the MNIST data-set LeCun et al., 2010. Now one of the real benefits of the deep-learning revolution is that it has lead to the development of really mature software packages that allows us to efficiently implement these methods. In this example we will use PyTorch but as it is only the inner-loop that will be effected you can use other packages as well. If you do not have `PyTorch` installed go to the website and follow the install instructions based on the Python installation you have. I am using `pip` and I do not have `CUDA` on my machine which means I get the following install command,

Code

```
pip install torch==1.7.0+cpu torchvision==0.8.1+cpu torchaudio==0.7.0 -f\
    https://download.pytorch.org/whl/torch_stable.html
```

Once we have this set-up we are going to build up the network and load the data-set. We will follow the excellent tutorial on Kaggle where you can download a notebook with the code. You will also need the data which can be downloaded from the following repository. First try and make the code run and see if you can regenerate the plots that is shown in Figure 2.

The first thing that you will probably notice is that it took quite some time to get the results. On my rather old and slow laptop it took about 20 minutes to train the network 25 epochs[4]. So now our inner-loop is much more expensive compared to the previous example and we are in a situation where we really can see the benefit of the Bayesian optimisation loop. We will now briefly go through the code in the tutorial to explain a couple of points and then we will set-up the objective function so that it is suitable for Bayesian optimisation.

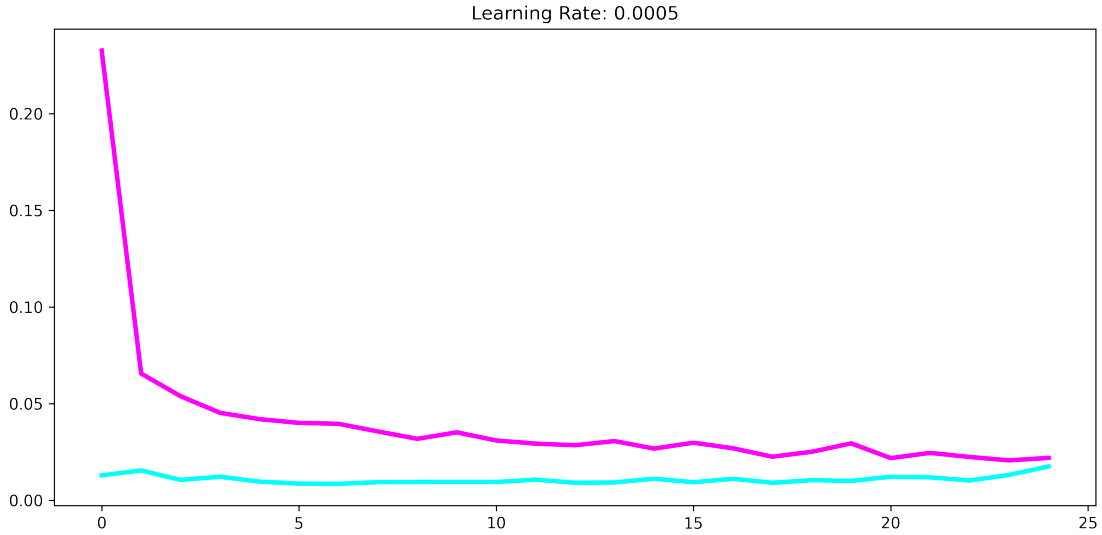

Figure 2: *The figure above shows the performance on the training (magenta) and the validation (cyan) set for the neural network for a learning rate of* 0.0005 *using the Adam optimiser. Note how the validation loss starts getting worse towards the end of the optimisation while the training loss is still reducing. This shows that eventually the algorithm starts over-fitting to the data.*

We are not going to go through the code in detail, just the things we need to understand to convert this to a Bayesian optimisation loop. In recent years there has been an explosion in different software packages for learning neural networks. Fundamentally they all have the same goal, *to provide a framework that allows propagation of gradients through functions*. This is quite a challenging thing as it means that we cannot compartmentalise our code to the same extent. The most common approach to achieve this is to build up a computational graph that clearly specifies how each part of the code is executed. Now what this means is that the code that you write is not actually the code that is executed, the code that you write is the *specification* of the graph. Different packages have different approaches to achieving this each with their own pros and cons. The important part of this structure is that as long as you have specified the derivatives of each function you can back-propagate through the graph Rumelhart et al., 1986. Its beyond the scope and the intention of this worksheet to go through the inner-workings of `PyTorch` and how this is done but there

[4] an epoch is the time it takes to update the network based on all the training data.

exists many tutorials online, a good example is directly from the PyTorch Website and another article that explains Linear Regression and how you go from `Numpy` to `PyTorch`.

The first part of the code is focused on loading the data and preparing it for the algorithm. The second part creates the neural network with `class Net(nn.Module)`. To understand this part of the code you can look at the documentation for the base-class `torch.nn.Module`. The part that is important for us is how the training of the model is set-up.

Code

```
import torch.optim as optim

# specify loss function
criterion = nn.CrossEntropyLoss()

# specify optimizer
optimizer = optim.Adam(model.parameters(), lr=0.0005)
```

First the objective function is specified, in this case to be the *cross entropy loss*. The cross entropy of a distribution q relative to p is defined as,

$$H(p, q) = -\mathbb{E}_p[\log\, q].$$

A simple way to see why this makes sense as a loss function is to expand the expectation as,

$$H(p, q) = -\mathbb{E}_p[\log\, q] = H(p) + \mathrm{KL}(p\|q),$$

this means that if we are minimising the cross entropy between the true distribution p and the output of the model q we are minimising the kullback-leibler divergence between the two distributions. Now we will consider the labels of the training data y as the "true" distribution of the data p and the output of the model q.

The next step is to specify the optimiser, similarly to our previous example we will use the Adam optimiser Kingma et al., 2014. When the optimiser is created we connect it to the parameters of the model by `model.parameters()` and set the learning rate `lr`. Our Bayesian optimisation loop will be for the learning rate, it is therefore around this call we need to write our own objective function. Now we have the model, an objective function and an optimiser and we are ready to train the model.

Removing the house-keeping code in the notebook the key parts of the training is the following,

Code

```
# Set the model in training state
model.train(True)
# Set the gradients to zero
optimizer.zero_grad()
# Compute the prediction of the model
output=model(data)
# Compute the loss of the current prediction
loss=criterion(output, target)
# Compute the gradients of the loss wrt the model parameters
loss.backward()
# Update parameters of model according to gradients
optimizer.step()
```

We hope that the above code makes sense, one of the things that looks a little bit strange is the `optimizer.zero_grad()` call. This is a `PyTorch` peculiarity in that for each call of `loss.backward()` the framework will accumulate

the gradients, this can be very convenient for certain models[5] but as we do not want this in this specific case we need to specifically set the gradients to zero before each update. The reminder of the code simply evaluates the model on the validation set.

This was a very quick explanation of what is going on in the code hopefully it makes a rough sense. The important thing is not to understand how the actual inner-workings of the neural network functions but how the network is trained. This should now be sufficient for us to alter the code and make a Bayesian optimisation loop just as for the previous model to find the best learning rate. As you have noticed the training is very slow, what I would recommend is to reduce the number of epochs and choose a slightly smaller training data-set to make sure that everything works before you crank everything up for your final experiment.

1. alter the code so that you get an objective function suitable for Bayesian optimisation
2. try the loop on your own machine with very low settings to make sure the loop works
3. write the docker image that you need to get the required packages
4. try the docker image locally
5. run the experiment on AWS

3 Parallel Computations

One of the benefits of using the cloud for computation is that you are no longer bound to run your computations on a single machine, you can spawn more than a single instance in the cloud. However, certain tasks are not easy to distribute as the computations of each task depends on each other. The first thing you need to have is some form of algorithm that we can make parallel, secondly we need to think if we need to run all the computations *synchronous* or if we can allow for *asynchronous* computation. The first implies that we have a method where we can split things up into smaller independent jobs, we now farm these out of our computational *nodes* from a *master* Figure~3. The master then waits for each of the nodes to finish before it repeat the same process. This we can often do with many algorithms, however, it will only be efficient if **all** the nodes take the same time to finish. Another way of saying this is that the time of the *master loop* is dependent on the *worst* node computation. To avoid this we can instead try and create a *asynchronous* algorithm where the *master loop* can continue as soon as something is returned.

The key components we need to add in order to run parallel computation in the cloud is some form of messaging system that allows different nodes to communicate with each other. For AWS we will use something called `Amazon Simple Que Service` (SQS). This framework will allow us to pass messages from each of the nodes to the master. In the Bayesian optimisation case the master will keep track of the surrogate model and compute the acquisition function while each node will evaluate the objective function and pass back the result of the evaluation to the master. In the next section we will go through how we can perform parallel Bayesian optimisation using quite a simple method that simply alters the acquisition function from what we have been used to so far.

3.1 Parallel Bayesian Optimisation

In the last part of this worksheet we will look at how we can use the parallellisations that we discussed previously in a Bayesian optimisation loop. We will take the approach outlined in Kandasamy et al., 2018 which describe both a synchronous and asynchronous method to perform Bayesian optimisation using Thompson sampling. There have been other works on parallel versions of the loop such as Alvi et al., 2019 however due to the Thompson sampling loop in Kandasamy et al., 2018 I think this paper is easier to implement. You are of course free to choose another paper that you want to try, and there are more than the two I mentioned here. At this stage we hope that you should be familiar with how AWS and Docker

[5] for example sequential models such as RNNs

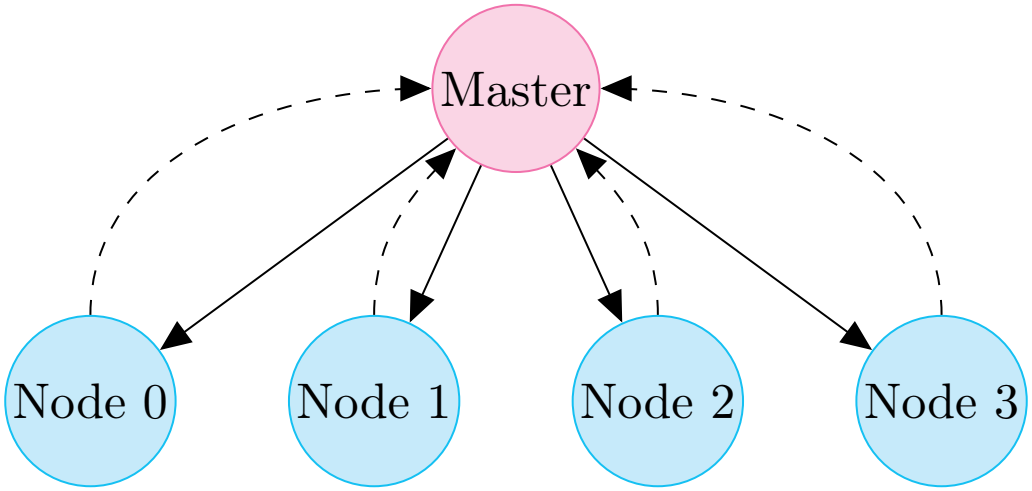

Figure 3: *The above figure shows the naming of the computational nodes. The Master starts the each of the Nodes (filled arrows) by spinning up each of the computational instances. As soon as the nodes are finished the communicate back the results (dashed arrow) to the master.*

works and also how the parallel computation can be done. We will therefore now only discuss the method and not the specific implementation of the method.

First have a brief look at the paper and see if it makes sense. Its quite algorithmic in the way it is written which suits us well as we are going to implement it. The first part of the paper goes through Bayesian optimisation and then provides a background to Gaussian processes as surrogate models. The main difference in this material to what you have seen before is the use of Thompson sampling as the acquisition function. Thompson sampling is a simple acquisition function that is easy to implement. In effect what it does is to draw a sample from the surrogate model and then use the negative sample[6] as the acquisition. An example is shown in Figure 4. Intuitively it makes sense as it naturally balances exploration and exploitation through the stochasticity in the model.

The important part of the paper starts in Section 3 where two different algorithms are presented. It is a little bit confusingly written as when you look at `Algorithm 1` it just looks like a standard Bayesian optimisation loop. What you have to do is to read a bit further where the algorithm is adapted to the parallel setting. The algorithm below assumes that you have run the algorithm in the paper to start-up all M workers using the `Algorithm 1` in the paper.

Now this method should be fairly straight-forward to implement, as you can see it basically exploits the stochastic structure of Thompson sampling such that it draws a unique acquisition function for each worker. This gives each worker a unique location to evaluate. After each worker is done we accumulate all the data, update the surrogate model and move to the next iteration. As you probably notice this feels like a rather inefficient way to do parallelisation as the speed of the algorithm is based on the slowest of the workers. The authors of the paper therefore proceeds to propose an algorithm that removes this step and simply updates the Gaussian process as soon as a new function value is available. This is shown in `Algorithm 2`. Hopefully the two algorithms makes sense at a level where you should now be able to implement these in a cloud infrastructure.

The reminder of the paper is focused on the theoretical guarantees and the experimental evaluation of the

[6] if we are minimising the function

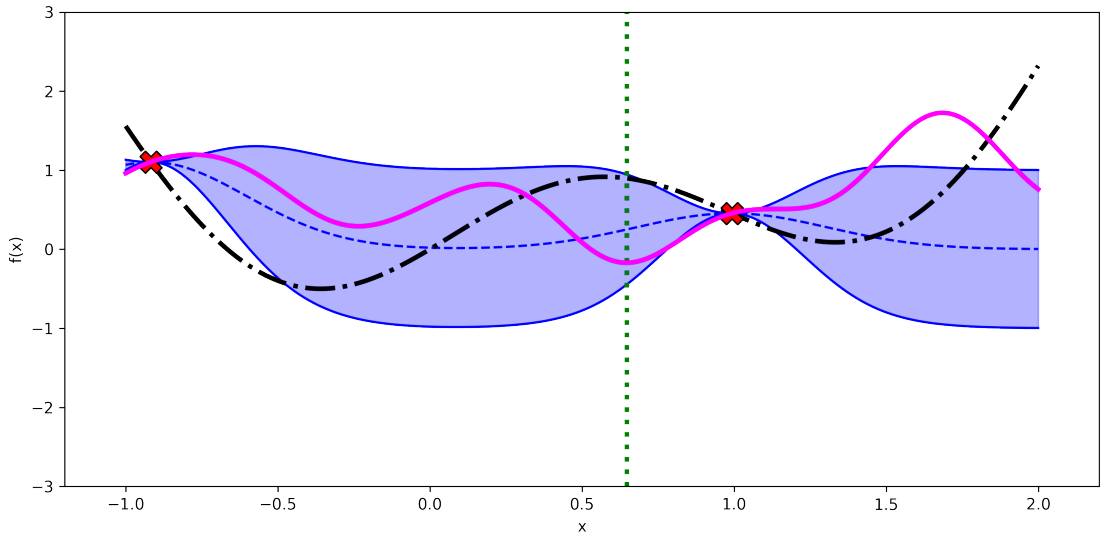

Figure 4: *The plot shows one iteration of thompson sampling. The current posterior prediction is shown in blue, the true unknown function in black and a sample from the posterior in magenta. We will now pick the minima of the magenta shown in green as a location to evaluate the function.*

Algorithm 1 seqTS

Require: Posterior GP $\mathcal{GP}(\mu_{D_1}, \kappa_{D_1})$

$\mathcal{D}_1 \leftarrow \{(x_{1i}, y_{1i}\}_{i=1}^{M}$, $\mathcal{GP}_2 \leftarrow \mathcal{GP}(\mu_{D_1}, \kappa_{D_1})$

for j=2,3,... **do** ▷ The outer Bayesian Optimisation loop

  for i=1,2,...,M **do** ▷ The loop evaluating a different aquisition for each woker

    Sample $g_i \sim \mathcal{GP}_j$

    $x_{ji} \leftarrow \text{argmax}_{x \in X}\, g_i(x)$

  end for

  Start worker i to evaluate y_{ji} at location x_{ji} ▷ Start each worker with their designated evaluation

  Wait for **all** workers to finish

  $\mathcal{D}_{j+1} \leftarrow \mathcal{D}_j \cup \{(x_{ij}, y_{ij})\}_{i=1}^{M}$ ▷ When all workers are done update the data

  Compute posterior $\mathcal{GP}_{j+1} = \mathcal{GP}(\mu_{D_{j+1}}, \kappa_{D_{j+1}})$ ▷ Update the $\mathcal{GP}$ using all the aquired data from all workers

end for

work. While this is interesting, and shows the benefit of this rather "hacky" approach empirically its not directly important for the work we are doing. If you are interested do read the rest of the paper and maybe you could see if you could replicate the experimental findings on a simple task.

4 Summary

This ends this worksheet and the course in Machine Learning for the Physical World for you. We hope that you have gotten a slightly different view on some of machine learning and seen how we can build surrogate models that allows us to exploit the knowledge that we have about a system. With this worksheet we aimed to give you on the CDT course an opportunity to see how we can leverage computational structures in the cloud. Hopefully you can now take this material and think about how this will relate to your specific research interest.

References

Alvi, Ahsan S. et al. (2019). "Asynchronous Batch Bayesian Optimisation With Improved Local Penalisation". In: *CoRR*. arXiv: `1901.10452 [stat.ML]`.

Kandasamy, Kirthevasan, Akshay Krishnamurthy, Jeff Schneider, and Barnabas Poczos (2018). "Parallelised Bayesian Optimisation via Thompson Sampling". In: ed. by Amos Storkey and Fernando Perez-Cruz. Vol. 84. Proceedings of Machine Learning Research. Playa Blanca, Lanzarote, Canary Islands: PMLR, pp. 133–142.

Kingma, Diederik P. and Jimmy Ba (2014). "Adam: a Method for Stochastic Optimization". In: *CoRR*. arXiv: `1412.6980 [cs.LG]`.

LeCun, Yann, Yoshua Bengio, et al. (1995). "Convolutional networks for images, speech, and time series". In: *The handbook of brain theory and neural networks* 3361.10, p. 1995.

LeCun, Yann and Corinna Cortes (2010). "MNIST handwritten digit database". In:

Rumelhart, D E, G E Hintont, and R J Williams (Oct. 1986). "Learning representations by back-propagating errors". In: *Nature* 323.9, pp. 533–536.

Shalev-Shwartz, Shai and Shai Ben-David (2014). *Understanding Machine Learning: From Theory to Algorithms*. New York, NY, USA: Cambridge University Press. ISBN: 1107057132, 9781107057135.

Snoek, Jasper, Hugo Larochelle, and Ryan P Adams (2012). "Practical Bayesian Optimization of Machine Learning Algorithms". In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger. Curran Associates, Inc., pp. 2951–2959.

Strubell, Emma, Ananya Ganesh, and Andrew McCallum (2019). "Energy and Policy Considerations for Deep Learning in Nlp". In: *CoRR*. arXiv: `1906.02243 [cs.CL]`.