# Online learning using Multi-Layer Perceptrons with Importance Sampling

**Christina Bogdan, Manoj Kumar**
{ceb545, mks542}@nyu.edu

## 1 Introduction

Online learning is a subset of machine learning where the entire set of training data is not available at one specific instance of time but arrives over a number of time steps in batches. When using a parametric model to solve a problem in the online learning setting, it is possible that the distribution of the data will shift over time. Thus, it is ideal to use a model that can adjust to this shift. In this project, we apply Sequential Monte Carlo (SMC) methods to solve this problem. More specifically, we apply SMC techniques to estimate the parameters of a multi-layer perceptron (a type of neural network). We use the framework described in **Sequential Monte Carlo Methods in Practice** [AD01], but make some changes in the algorithm to overcome issues that stem from the large number of parameters in this type of model. We then compare the model obtained using this algorithm to one trained through stochastic gradient descent on a toy dataset, and the iris dataset.

## 2 Framework

In this section, we describe in brief the various key concepts associated with the project.

### 2.1 Supervised Learning

Supervised learning is a machine learning task where every sample has a set of features and a label. In this setting, the features correspond to attributes that describe an instance, and the label is an attribute that the user wants to predict. The goal of the task is then to predict the labels of samples for which the features are known but labels are unknown. For example, in a house-price prediction task the features can be the location and size of the house. The task is to predict the pricing of a house for which its features are known.

Mathematically each sample can be represented as a data point $(X_i, y_i)$ where $X_i$ is a vector of numerical values of size $N$ corresponding to the features of sample $i$. The features can now be represented as a matrix $X$ of size $M \times N$ and the labels can be represented by a vector of size $M$. Given these $M$ data points, the task is to predict the labels $y_{new}$ given new data points $X_{new}$.

### 2.2 Logistic Regression

Logistic Regression belongs to the family of linear classifiers which assume the data to be linearly separable. A classifier assigns to each data point a likelihood of belonging to one of a pre-defined number of classes. Given a matrix of weights $W$ of size $L \times N$, for an unlabelled sample $x$, a linear classifier predicts the label to be $\underset{c \in 1 \ldots L}{\operatorname{argmax}}(w_c^T x)$

For an unknown sample, sometimes it is beneficial to not have just the predicted output label but also the probabilities for every class. Every linear mapping can be converted into probabilities using the softmax function

$$p_c = \frac{e^{w_c^x}}{\sum_{l=1}^{L} e^{w_l^x}}$$

The loss function used in logistic regression is the negative log-likelihood of the data

$$\ell = \sum_{i=1}^{N} -\log p_{y_i}$$

Where $p_{y_i}$ is the probability of the true label under the model. In practice regularization is added to the loss function to prevent the weights from becoming large. For this project, we use L2 regularization which is the squared norm of the weights. Thus, the regularized loss function is given by $\sum_{i=1}^{N} -\log p_{y_i} + \alpha||W||_F^2$ where $\alpha$ is a parameter that controls the degree of regularization, and $W$ is the parameters of the model.

Typically, the weights are set to some random value initially and the optimal weights can be obtained by minimizing this loss function using convex optimization techniques such as gradient descent.

### 2.3 Multi-Layer Perceptron

In some cases, certain higher order or non-linear combination of existing features might be required to model the problem. For example, an indicator feature representing if 0.5 times the size of the house plus the number of bedrooms is greater than zero can be a higher order feature. However, such higher order non-linear combinations require domain-specific knowledge. A multi-layer perceptron solves this problem by learning such higher order feature combinations on its own.

Let $x$ be the input as before, but now we get a higher-order representation of $x$ by passing it through a non-linear function $x_2 = f_1(w_1^T x))$. Then $x_2$ can be used as an input instead of $x$ to the logistic regression model described above. We fix the number of hidden-layers to 1 over here but it can be anything in general depending upon the complexity of the features required. Some popular choices of non-linearity in $f$ include the relu which zeros out negative values and the tanh function which squishes the input values to lie in between the range (-1, 1).

Now, instead of a single weight matrix mapping from input to the classes, we have two weight matrices. $W_1$ of size $N \times N_1$ mapping the input to the hidden layer and $W_2$ of size $N_1 \times N_L$ mapping the hidden layer to the output. Though the problem is no longer convex, gradient-based techniques can still be used to give a local solution.

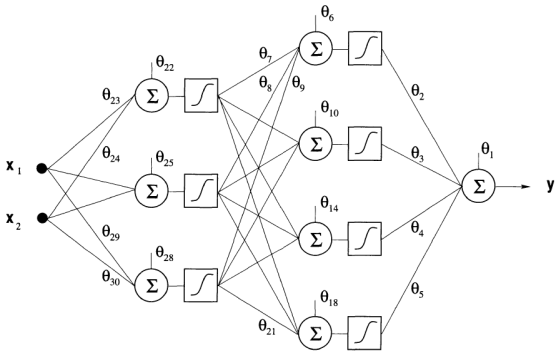



Figure 1: An example of a two-layer MLP [AD01]

### 2.4 Sequential Monte Carlo

In online learning applications, we might not have access to all of $X, y$ at once but in a streaming or online fashion across multiple time-periods $t = 1$ to $T$. In addition, we might need samples from the posterior distribution of the weights $P(w_t|w_{t-1}, (X_{1:(t-1)}, y_{1:(t-1)}))$ rather than a point estimate of $w_t$.

In most of the cases, the distribution $P(w_t|w_{t-1}, (X_{1:(t-1)}, y_{1:(t-1)}))$ is intractable. A generic sequential Monte Carlo algorithm samples $N$ times from a proposal distribution $q(w_t|w_{t-1}, (X_{1:(t-1)}, y_{1:(t-1)}))$, provides weights to each of these $N$ samples and then redistributes them by resampling according to the weights.

## 3 Algorithm

The algorithm involves several stages. We propose a large number of parameters through importance sampling, and we weigh the parameters according to the regularized log-loss. These parameter weights are locally refined using particle swarm optimization [Ken11]. To suppress parameters with low probability, we then resample the parameters according to their normalized weights. Finally, k-means clustering is used to group weights that are similar to one-another, and these clusters are averaged to create a cluster weight. We then select the final weights as the one whose cluster average incurs the smallest log-loss on the training set. At the next timestep, new weights are proposed using the resampled weights of the previous timestep. This gives a 'warm start' based on the previous state of the parameters. Below, we outline the full algorithm and then discuss the individual steps. Our implementation is also on GitHub[1].

**Data:** dataset $d_{1:T}$ where $d_t$ is data acquired at timestep $t$
**Result:** MLP parameters $\theta_T^* \in \mathbb{R}^p$, where p is the number of parameters in the model

Initialize $\theta_0 \sim \mathcal{N}([0]^p, \sigma^2 \cdot I^{p \times p})$

**for** $t = 1 \cdots T$ **do**
  **for** $i = 1 \cdots N$ **do**
    *(Importance sampling step)*
    Sample $\hat{\theta}_t^i \sim q(\theta_t | \theta_{0:t-1}^{(i)}, d_{1:t})$
    **for** $j = 1 \cdots M$ **do**
      *(Swarm Optimization Step)*
      Propose $k$ new candidates $\hat{\theta}_t^{(i,k)} \sim \mathcal{N}(\hat{\theta}_t^{(i)}, \sigma'^2 \cdot I)$
      Acquire $\hat{y}_t^{(i,k)} = \text{MLP}(\hat{\theta}_t^{(i,k)})$ Acquire $\hat{y}_t^{(i)} = \text{MLP}(\hat{\theta}_t^{(i)})$
      Calculate loss $l_t^{(i,k)} = \log(y_t | \hat{y}_t^{(i,k)}) - \alpha ||\hat{\theta}_t^{(i,k)}||_F^2$
      **if** $\min_k l_t^{(i,k)} < l_t^{(i)}$ **then**
        Set $\hat{\theta}_t^{(i)} = \hat{\theta}_t^{(i,k)}$
      **end**
    **end**
    Compute importance weight
    $w_t^{(i)} = \log(y_t | \hat{y}_t^{(i)}) - \alpha ||\theta_t^{(i)}||_F = -y_t \log(\hat{y}_t^{(i)}) + (1 - y_t) \log(1 - \hat{y}_t) - \alpha ||\theta_t^{(i)}||_F^2$
  **end**
  Normalize each weight
  $$\tilde{w}_t^{(i)} = \sigma(w_t^{(i)}) = e^{w_t^{(i)}} [e^{\sum_{j=1}^N w_t^{(j)}}]^{-1}$$
  Resample parameters according to importance weights using a multinomial distribution
  $$\theta_t^{(1:N)} \sim \text{multi}(\hat{\theta}_t^{(1:N)}, \tilde{w}_t^{(1:N)}))$$
**end**
Cluster final weights $\theta_T^{(1:N)}$ using k-means clustering
**for** $\mathcal{C}_k$ *in* $\mathcal{C}$ **do**
  $\theta_T^k = |\mathcal{C}_k|^{-1} \sum_{\theta_T^{(i)} \in \mathcal{C}_k} \theta_T^{(i)}$
  Compute cluster loss $l_T^k = \log(y_t | \hat{y}_t^k) - \alpha ||\hat{\theta}_T^k||_F^2$
**end**
Select final parameters $\theta_T^* = \theta_T^{k^*}$ where $k^* = \arg\min_k l_T^k$

**Algorithm 1:** Sequential Monte-Carlo technique to find weights

### 3.1 Importance Sampling Step

At each timestep we use an importance distribution based on the weights at the previous timestep. In our implementation, we take the importance distribution to be a Gaussian with the mean being the value of the parameters at the previous timestep. The covariance matrix of this Gaussian is isotropic

[1] `https://github.com/christaina/MCMCMC/blob/master/mlp_sequential.py`

with the scale being a hyperparameter in the model. Hence, the $q(w_t|w_{t-1}, (X_{1:(t-1)}, y_{1:(t-1)})$ described above becomes $\mathcal{N}(\theta_{t-1}, \sigma^2 I)$. The importance weight of each of these samples is given by the negated regularized log loss.

There is a tradeoff between having a large scale and the number of importance samples taken. As the scale grows, more of the parameter space is explored. However, at the same time, more samples are needed if one wants to generate samples that are close to the ideal set. This is especially notable in the case of multi-layer perceptions, which have a large number of parameters. The ability to define an importance function makes the model flexible in allowing the user to consider what knowledge they want to use when they propose a new set of parameters.

### 3.2 Particle Swarm Step

We use particle swarm [Ken11] on the samples produced by the importance function to locally optimize them. In our experiments, we use a scale of 0.1 for our proposal Gaussian, and produce 10 candidates for each $\theta_t^{(i)}$ over 50 iterations. This improved issues when a few samples had a large weight associated with them, and resampling these during the selection step collapsed all the proposed parameters into one sample. In this setting, we also see better results when fewer importance samples are taken.

### 3.3 Selection Step

After performing the particle swarm, we transform the importance weights into probabilities using a softmax function. We assume these probabilities to be parameters of a multinomial distribution and we resample from this distribution. The purpose of this is to discard unlikely samples and encourage the algorithm to explore parameter spaces that are likely to produce the data. At the same time, we hope to distribute the resampled parameters so that we maintain several high-probability potential sets.

### 3.4 Clustering Step

To form our final coefficients, we cluster the final set of parameters using k-means clustering with the number of clusters set to 8 and then average the parameters within each cluster. We return the parameter that has the lowest loss on the training data. We included this step because the large parameter space would result in several potential parameters that were likely, but distant from one-another.

## 4 Experiments

We tested our model on both real and toy datasets, and evaluated its performance in comparison to a multi-layer perceptron trained through stochastic gradient descent. We outline the results below and highlight its successes.

### 4.1 Iris Dataset

First, to evaluate the method's stand alone performance, we tested it on the iris dataset[2]. Here, the task is to classify flowers into one of three classes: Setosa, Versicolour, and Virginica. There are 150 examples in the dataset, and four features: sepal length, sepal width, petal length, and petal width. The iris dataset is a popular baseline for machine learning problems, which is why we test on it. We did a train test split of 70-30 and test independently on the test dataset. We set the regularization to 1e-3, vary the scale of the covariance matrix of the proposal distribution and report the accuracy on the train and test splits.

[2] `http://scikit-learn.org/stable/auto_examples/datasets/plot_iris_dataset.html`

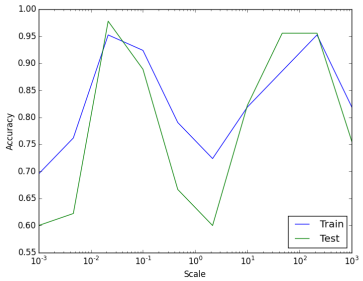
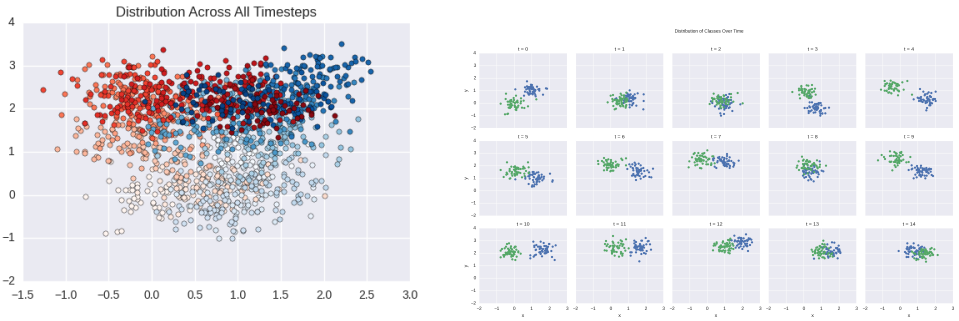



Figure 2: Train - test accuracy on iris for different importance function scales

### 4.2 Shifting Gaussians

As a toy example, we generate two classes from separate two-dimensional Gaussian distributions. At each time-step, the mean of each Gaussian is shifted randomly by a factor between -1 and 1. The dataset spans 15 timesteps.




(a) Full plot of data. Color represents class and shade represents timestep

(b) Plot of data at individual timesteps

Examining the plot of the data set across all timesteps, it is clear that it cannot be classified by a single model. At the same time, when there are only a small number of data points available at every timestep ($n = 50$), it is inefficient to train a separate model. Further, the timestep data may not be neatly batched.

We evaluate the performance of our method compared to stochastic gradient descent by training the SMC MLP across all timesteps, and also train individual SGD MLPs for each time step. Each MLP has the same number of parameters and same amount of regularization (hidden size of 8, $\alpha = 1e-4$). In the case when each time step has 50 data points, the MLP trained with SMC outperforms the one trained with SGD by a large margin. On the test set, which consists of additional data from the same distribution at the final timestep, the SMC algorithm achieved a test accuracy of **0.829** with a standard deviation of 0.12, while the SGD-trained MLP had an accuracy of, on average, **0.635** with a standard deviation of 0.22 across 50 runs.

## 5 Conclusion

We introduced and tested and algorithm to obtain parameters of a multi-layer perceptron using sequential Monte Carlo techniques. The algorithm that we used was introduced in **Sequential Monte Carlo Methods in Practice** [AD01], but we modified it to account for issues that stemmed from the large number of parameters. Our modifications were using particle swarm to locally optimize parameters generated during the importance sampling step, and using k-means clustering to cluster the proposed parameters before averaging them to acquire the final set of coefficients.

With these modifications, we tested the SMC-trained MLP on the iris dataset, as well as an artificially generated dataset of Gaussians that shifted over time. The algorithm performed well on the iris

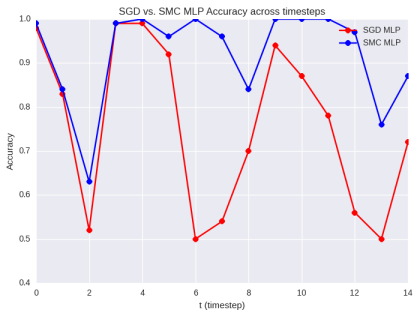



Figure 3: Train accuracies at each time step

dataset, and algorithm outperformed an MLP with the same number of parameters trained using stochastic gradient descent. At the same time, the SMC MLP algorithm is slow, and struggles to scale when the number of parameters increases.

## 6 Future Work

In the future, a next step would be to try different distributions as importance functions, and vary their dependency both on the data and on the previous timesteps (for example, the importance distribution that we used only depended on the parameter values at the previous timestep). We could also attempt to model error in the dataset in the loss function, or use a different loss function to weight samples other than log-loss, because with this algorithm we do not have the constraint of the loss having to be differentiable. We could also try different clustering methods. For example–clustering using the loss rather than in the parameter space [HGH$^+$12].

We would also like to find more sequential or time series datasets to test this algorithm on, to see under which circumstances it performs best. Finally, we would like to be able to speed up the algorithm so that it would be more efficient with larger models.

## References

[AD01] Nando de Freitas & Neil Gordon Arnaud Doucet, editor. *Sequential Monte Carlo Methods in Practice*. Springer, 2001.

[HGH$^+$12] Fengji Hou, Jonathan Goodman, David W Hogg, Jonathan Weare, and Christian Schwab. An affine-invariant sampler for exoplanet fitting and discovery in radial velocity data. *The Astrophysical Journal*, 745(2):198, 2012.

[Ken11] James Kennedy. Particle swarm optimization. In *Encyclopedia of machine learning*, pages 760–766. Springer, 2011.