Efficient Dynamic Computational Offloading

Josh Veltri
Department of Electrical Engineering and Computer Science
Case School of Engineering
Case Western Reserve University
December 12, 2017

Abstract – For systems utilizing dynamic computational offloading, the problem of dynamically determining the optimal execution location of each stage of the computational task can be very computationally expensive, especially if the total number of stages is large. In this paper, we investigate whether it is possible to lower this computational overhead while still obtaining near-optimal execution location assignments by using (i) a multi-label classifier implemented using a multi-output neural network or (ii) a reinforcement learner implementing the UCB1 algorithm to make the assignments. The performance of these two approaches were evaluated on randomly-generated synthetic examples. We find that the multi-label neural-network classifier yields both high accuracy in its execution location decisions and significantly lower computation times than the standard non-machine-learning approach. However, the UCB1 reinforcement learner yields poor overall performance.

I. Introduction

Dynamic computational offloading is the idea that, in embedded or mobile systems with significant time and/or energy resource constraints, overall performance, as measured by local resource consumption, can be improved by allowing a cloud-based remote computing resource to handle some portions of the computation on behalf of the local device [1].

The key distinction between traditional cloud computing models and dynamic computational offloading is that, unlike in traditional cloud computing, dynamic computational offloading **adapts as network conditions change** to always provide the best balance of local and remote computation for the current conditions. For instance, if the local device's communication with the remote server is impeded by a drop in network throughput, a dynamic computational offloading system might perform additional computation locally if doing so reduces the amount of data it is required to send over the network.

To use computational offloading, **a computational task must be broken into a series of sequentially-executed stages that are individually assigned to execute either locally or remotely as conditions warrant**. See Table 1 for a basic visualization of this. In most of the existing literature on computational offloading, the software engineer implementing an application chooses logical breakpoints to serve as boundaries between computational stages [2, 3, 4, 5]. Execution can be migrated from one device to the other only at these stage boundaries. When this occurs, the application data output by the last stage is sent over the network and used as input data for the next stage.

	Task			
	Stage 1	Stage 2	Stage 3	Stage 4
One possible execution location assignment	Local	Remote	Remote	Local
Another possible execution location assignment	Remote	Remote	Local	Local

Table 1: Two possible offloading assignments for a task with 4 stages. In dynamic computational offloading, the first may be selected under some network conditions and the second in under others. Note that while only two example offloading assignments are shown here, there are many more possible execution location assignments for a 4-stage computational task.

Choosing which stages should be handled locally and which should be offloaded in order to minimize resource consumption on the local device requires considering dynamic factors such as the network throughput and network round-trip-time delays alongside the average local execution time, remote execution time, and data transfer requirements for each stage [1].

Unfortunately, **computing the optimal offloading behavior can be computationally expensive**. In this work, we explore two machine-learning approaches for generating dynamic offloading decisions and evaluate (i) whether they provide good enough approximations to the optimal decision to be useful in practice, and (ii) whether they substantially reduce the computational overhead compared to approaches that are guaranteed to yield the optimal offloading decision. In the first machine-learning approach, we train a fully-connected neural network to generate offloading decisions. In the second, we employ a UCB1 reinforcement learner to make offloading decisions.

II. Problem Setting

a. Computational Offloading Model

At its core, **dynamic computational offloading consists of assigning each stage of the computational task to be executed either locally or remotely in the way that provides the greatest utility to the local device**. We use the utility formulation from [6], where the utility U is defined to be a weighted-product of two factors: one related to the total time t it takes to perform a computation and one related to the total energy E expended by the local device during the computation. Specifically, we define U as:

$$U(t, E) = U_t(t)^{\lambda} * U_E(E)^{1-\lambda}$$

Here λ lies on the interval [0,1] and controls the relative importance of the time and energy factors. U_t and U_E are formulated as follows:

$$U_t(t) = \begin{cases} 1, & t \le t_{min} \\ 1 - \dfrac{t - t_{min}}{t_{max} - t_{min}}, & t_{max} > t > t_{min} \\ 0, & t \ge t_{max} \end{cases}$$

$$U_E(E) = \begin{cases} 1, & E \le E_{min} \\ 1 - \dfrac{E - E_{min}}{E_{max} - E_{min}}, & E_{max} > E > E_{min} \\ 0, & E \ge E_{max} \end{cases}$$

t_{min}, t_{max}, E_{min}, and E_{max} are application-specific parameters controlling the points at which the time and energy components of the utility reach their maximum or minimum values.

As in [6], the total time t and total energy expended E required to compute the utility can be modeled as follows:

- The time t to complete the entire computation is the sum of the local computation times for any locally executed stages, the remote execution times for any remotely executed stages, and the communication time required to setup a TCP connection and exchange application data over the network.
- The energy E consumed by the computation is the sum of the energy expended on local computation, the energy expended transmitting data to the remote device, and the energy expended idling while awaiting the result of remotely computed stages.

Throughout this paper, we let t_{min} be ¼ times of the sum of the local computation times of all stages, t_{max} be 1.1 times the sum of the local computation times of all stages, E_{min} be 0, and E_{max} be t_{max} times the CPU power consumption while under load. These choices ensure that even if the system decides to execute all stages locally, some positive utility will still be derived. We also set λ to 0.25, which reflects the reality that most low-power and embedded systems will weight energy consumption more than computation latency.

The problem of scheduling dynamic computational offloading consists of finding the set of stages L that should be executed locally and the set of stages R that should be executed remotely to provide the highest utility score. It turns out that, for the utility formulation presented here and the (very reasonable) assumptions that (i) the average execution time of a stage is lower on the remote device than on the local device and (ii) the local device expends less energy when idling than while under heavy computational load, the optimal offloading strategy will include offloading just one sequence of consecutive stages [6]. That is, L will consist of the first α stages and the last γ stages, while R is the middle β stages. The task is then to find the α, β, and γ that yield the largest utility. More formally, for a computation with n stages [6]:

$$\arg\max_{\alpha,\beta,\gamma} U(t(\alpha,\beta,\gamma), E(\alpha,\beta,\gamma))$$

$$s.t. \quad \alpha + \beta + \gamma = n$$

$$\alpha \geq 0; \ \beta \geq 0; \ \gamma \geq 0$$

where definitions for $t(\alpha,\beta,\gamma)$ and $E(\alpha,\beta,\gamma)$ that comply with the formulations for t and E presented above are given by:

$$t(\alpha,\beta,\gamma) = \sum_{i=1}^{\alpha} t_{i,l} + I_\beta \left(2 * RTT + \frac{M_{\alpha+1} + M_{\alpha+\beta+1}}{r} + \sum_{i=\alpha+1}^{\alpha+\beta} t_{i,r} \right) + \sum_{i=\alpha+\beta+1}^{\alpha+\beta+\gamma} t_{i,l}$$

$$E(\alpha,\beta,\gamma) = P_L \sum_{i=1}^{\alpha} t_{i,l} + I_\beta \left(P_T \frac{M_{\alpha+1}}{r} + P_I \left(2 * RTT + \frac{M_{\alpha+\beta+1}}{r} + \sum_{i=\alpha+1}^{\alpha+\beta} t_{i,r} \right) \right) + P_L \sum_{i=\alpha+\beta+1}^{\alpha+\beta+\gamma} t_{i,l}$$

Here $t_{i,l}$ is the average local computation time of the i^{th} stage, $t_{i,r}$ is the average remote computation time of the i^{th} stage, M_i is the average number of bytes that must be exchanged to migrate execution at the start of the i^{th} stage, RTT is the current expected round-trip time between the local and remote devices, and r is the current expected throughput on the link between the local and remote devices. I_β is an indicator variable that is 1 if $\beta > 0$ and 0 otherwise. P_L is the CPU power-under-load, P_I is the CPU power while idling, and P_T is the wireless transmitter power [6]. In this work, we let P_L be 1.5 watts, P_I be 0.5 watts, and P_T be 1.2589 watts. These values correspond to the approximate power consumption of a Raspberry Pi Zero [7].

Notice that both t and E depend on dynamic network parameters r and RTT. The values of these parameters can be kept up-to-date by directly measuring them at the local device anytime it exchanges a message with the remote server [6]. Changes in these parameters over time will, through the changes they cause to t and E, affect the resulting utility values for each combination of α, β, and γ, resulting in dynamic offloading decisions.

b. Problem Statement

Now that we have fully fleshed out the utility model, we can state that the problem we attempt to solve is as follows:

Given: the current estimate RTT of the round-trip time, the current estimate r of the network throughput, the number of stages n in the computational task, the local computation time $t_{i,l}$ for each stage, the remote computation time $t_{i,r}$, the message size corresponding to each stage M_i, the local device's CPU power under load P_L, the local device's CPU power while idling P_I, and the transmission power P_T of the local device's wireless network card

Do: Determine, for each stage, whether the stage should be executed locally or remotely such that the total utility over the entire n-stage computational task is maximized.

c. Optimal Offloading via Exhaustive Search

One approach for finding the optimal offloading strategy is to perform an exhaustive search over the parameter space for α, β, and γ. This involves computing the utility for each valid combination of α, β, and γ and choosing the one that yields the highest total utility [6]. Unfortunately, because this operation must run on the low-power, resource-constrained local device performing an exhaustive search to find this optimum can be prohibitively expensive, especially as the number of stages in the computational task increases.

To provide a reference point for later comparisons, we make five measurements of average the exhaustive search time, in milliseconds, across 100,000 examples (generated as described below in Section II) of tasks with 10, 15, and 30 stages. These measurements were made on a HP Envy 15t-j100 Quad Edition laptop with an Intel Core i7-4700MQ CPU. (Note that this laptop is significantly more powerful than the typical mobile or embedded device that would need to perform this computation.) The observed exhaustive search times are shown in Table 2 (below).

	Exhaustive Search Time (ms)	
	Average	Standard Deviation
10-Stage Tasks	0.4181	0.0373
15-Stage Tasks	0.8991	0.1698
30-Stage Tasks	4.5144	0.6197

Table 2: Exhaustive search times for tasks with various numbers of stages

In subsequent sections, we will investigate whether machine-learning approaches can be used to learn to make sufficiently near-optimal local/remote stage assignments with significantly less computational overhead. In evaluating the effectiveness of the machine-learning approaches, we will compare the stage assignments generated by the learned model with the optimal stage assignments identified through an exhaustive search of the α, β, and γ parameter space.

d. Training and Test Data Generation

The training and test data we use to train and evaluate the machine-learning approaches is synthetically generated. For simplicity, we use an example-generation mechanism compatible with both the neural network and reinforcement learning approach we investigate. Since the neural network makes no demands of the order of the training examples, the primary constraint in doing this comes from the reinforcement learner, which requires sequences of examples from the same computational problem (that is, the same set of local computation times, remote computation times, and message sizes) with network conditions that evolve in a plausible way. Thus, the data sets are generated as follows:

For a task with n stages:

1. Select a random local computation time, in milliseconds, for each of the n stages. These times are skewed towards small computation times by sampling from a gamma distribution with shape parameter 1.5 and scale parameter 100. This reflects the intuition that most stages will be relatively fast to compute, while a handful can take significantly longer.
2. Compute corresponding remote stage computation times. To do this, we take the previously selected local computation time for each stage and divide it by a speedup factor, which is randomly chosen from a normal distribution with mean 15 and standard deviation 5.
3. Select a random message size, in bytes, for each stage (plus one additional message size for returning the computation's result in the case where the final stage is executed remotely). As with the local computation time, this is chosen from a gamma distribution to ensure that the average message has a reasonable size while occasionally allowing messages to be quite large.
4. Select a random round-trip time uniformly from the range 20 milliseconds to 700 milliseconds
5. Generate a sequence of throughput values, in bits per second. The sequence is a sine wave, vertically stretched and shifted to range between 0 and 5 Mbps, plus normally distributed white noise. This reflects the fact that instantaneous network throughputs should be expected to change over time but are unlikely to bounce dramatically up and down over any very brief period of time.
6. Create a feature vector for each value in the throughput sequence. This consists of one throughput value and all the values chosen in steps 1 – 4.
7. Using the local computation times, remote computation times, messages sizes, round-trip time, and throughput in each feature vector in the sequence, identify the local/remote stage assignments that yield the highest utility using the exhaustive search procedure described earlier and store the feature vector alongside the corresponding optimal stage assignments.
8. Repeat steps 1 – 7 many times to obtain many example sequences.

A Python script was used to generate sequences of examples of various lengths for tasks with 10, 15, and 30 stages and save them to a CSV file for later use.

III. Artificial Neural Network Approach

a. Multi-Label Classification Using Artificial Neural Networks

Artificial neural networks loosely simulate some of the connectivity and functionality of neurons found in human brains. Each neuron has several inputs (which correspond to the dendrites of a biological neuron) and a single output (which corresponds to the axon of a biological neuron). The value of the output is related non-linearly to the weighted summation of all the inputs. A network of neurons created by connecting several layers of neurons where the neurons of one layer accept as inputs the outputs of neurons in the previous layer can perform complex non-linear mappings of the inputs to first layer of neurons onto alternate spaces. Using the well-known backpropagation algorithm, neural networks can learn a mapping that fits example data.

One application of neural networks is **multi-label classification**. In multi-label classification, the classifier assigns multiple labels to a single input feature vector. For instance, the classifier might label a picture of a person as both "child" and "male" or both "adult" and "female". To do this, the network would be configured with two neurons in the output layer. The first is interpreted as a binary classification of adult/child and the second as a binary classification of male/female.

The problem of determining which stages to execute locally and which to execute remotely in computational offloading can be easily formulated as a multi-label classification problem. For a task with n stages, we configure a neural network with n outputs. The i^{th} output, for $i \in \{1,2,\dots,n\}$, is interpreted to determine as whether the i^{th} should be computed locally or remotely. The inputs to the first layer are a single feature vector consisting of:

- The throughput r
- The RTT
- n local stage computation times
- n remote stage computation times
- $n+1$ message sizes corresponding to the expected amount of data that would need to be transferred if the execution location is changed prior to each of the n stages and to return the computation's result if the final stage is computed remotely

The target output vector is the optimal stage assignments as determined by the exhaustive search procedure described in Section II.

b. Implementation

To construct and train the network, we utilize the Python scikit-learn library, which provides an implementation of basic artificial neural network functionality. We allow up to **5,000 training iterations** when training our network. Overfitting control is achieved using "**early stopping**". This involves holding aside 10% of the training set as a "validation set". The validation set is not used directly to train the network, but as the network is trained on the rest of the training set, the accuracy of the network on the validation set is periodically obtained. If the accuracy on the validation set begins to decrease, training is ceased to prevent overfitting. All networks are **fully-connected**. Network weights are updated during training using **stochastic gradient descent** and all neurons use **logsig activation** functions.

When training and testing the network, we use **5-fold cross-validation**. Because for a few of the example sequences the optimal offloading stage assignment varies little as the throughput changes, we keep all examples from the same sequence in the same fold when implementing the cross-validation to ensure that the test fold will consist entirely of fully-novel examples that do not closely resemble any of the training examples from the remaining folds. This cross-validation logic is implemented from scratch in Python. We also directly implement analysis logic to decode the neural network output and compute performance metrics.

c. Experiments and Results

We train numerous networks with various combinations of numbers of stages per task and numbers of neurons per hidden layer, and using various numbers of training examples. In total, we trained 36 networks, one for each combination of:

- 10, 15, or 30 stages per task
- 2 layers and 10, 50, 250, or 500 neurons per hidden layer
- 62,500, 250,000, or 500,000 examples

We initially experimented with varying the number of hidden layers but quickly discovered that using more than two hidden layers was providing little to no performance benefit while greatly increasing the required training time. Thus, all networks use **two hidden layers**.

We assess the performance of the trained networks via their test-set performance along the following metrics:

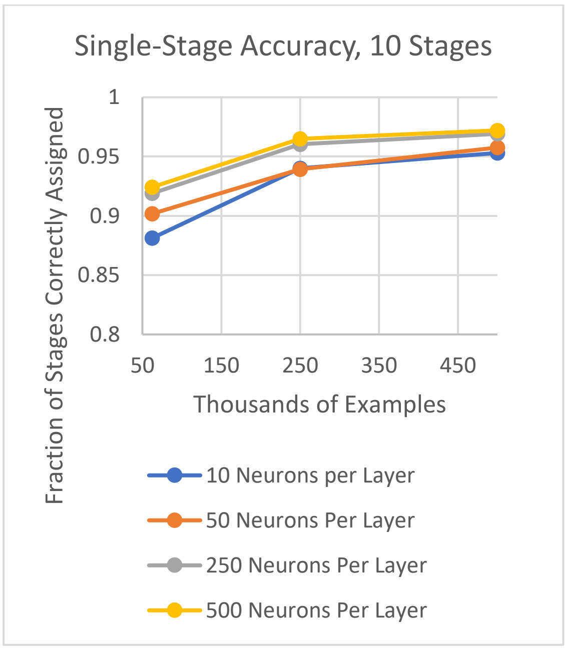
1. **Single-Stage Accuracy**: the fraction of stages for which the correct local/remote execution assignment is made
2. **All-Stages Accuracy**: the fraction of examples for which the correct local/remote execution assignments are made for all n stages
3. **Utility Loss**: the average difference between the optimal utility and the utility of the stage assignment produced by the neural network
4. The **average time** it takes the network to classify a novel example

Both the absolute measures of these performance metrics as well as their relative measures as the number of stages in the task, neurons per layer, and number of examples are varied are of interest.

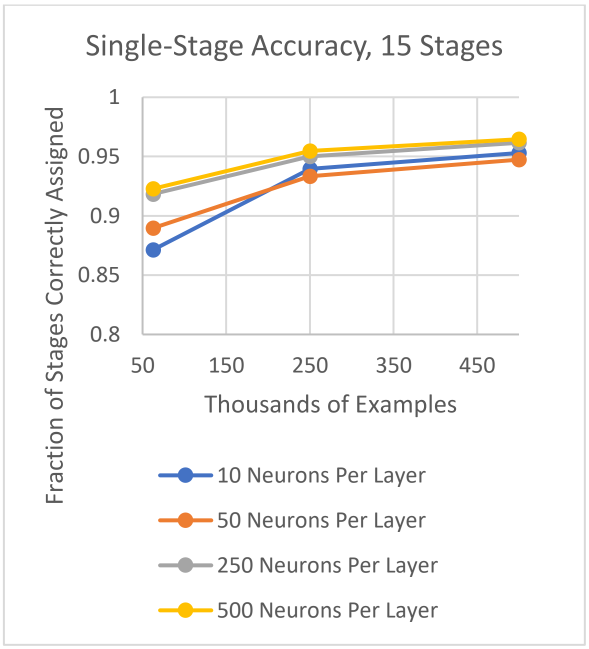
1. Single-Stage Accuracy

The average single-stage accuracy was high for all network configurations. The lowest value, 83.4%, unsurprisingly occurred for the scenario with the largest number of stages per task (30) and the fewest neurons per layer (10) and number of examples (62,500). Similarly, the highest value of 97.2% occurred with the fewest stages per task (10) and the most neurons per layer (500) and total examples (500,000).

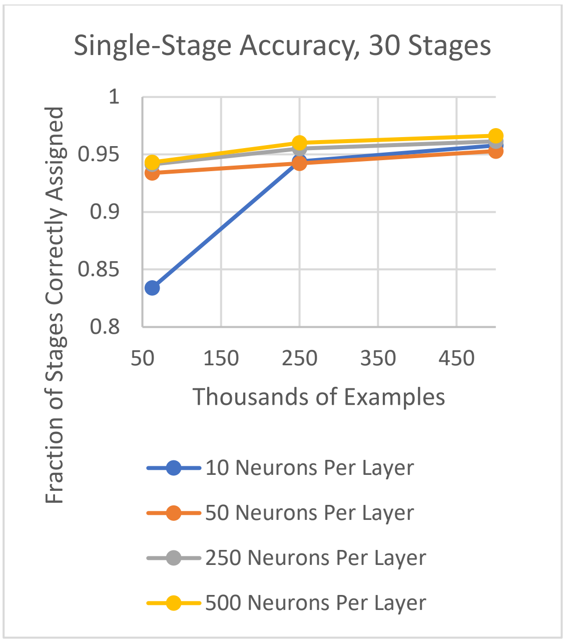
Figure 1 (below) depicts the single-stage accuracy for all 36 networks. Although the absolute differences in the accuracy of the networks trained with 500,000 examples is small, the standard deviations in the measures across different test folds is also quite small. As a result, a t-test of the stage assignment accuracy reveals that, for each number of stages per task, the 500-neruons-per-layer network is superior at the 95% confidence level to networks with 10 and 50 neurons per layer. The 500-neurons-per-layer network is also superior at the 95% confidence level to the 250-neurons-per-layer network for 30-stage tasks (but not for 10-stage tasks where $p = 0.0844$ or for 15-stage tasks where $p = 0.1264$).



(a)



(b)



(c)

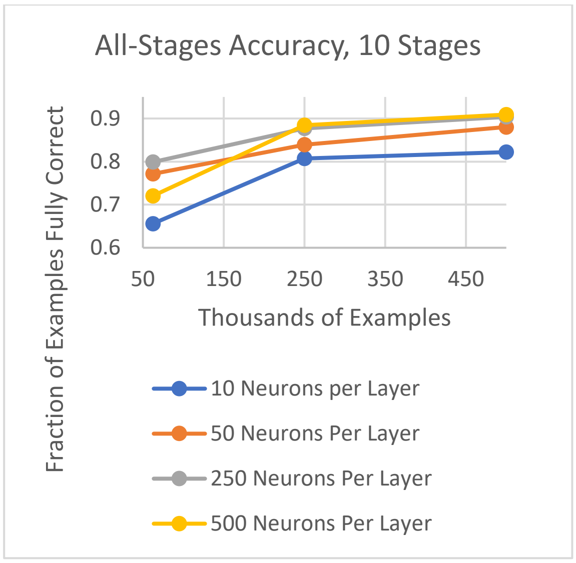
Figure 1: Learning curves showing the change in the average fraction of stages in test examples assigned the correct local/remote execution location for networks trained with various numbers of examples and numbers of neurons per layer. (a) shows the case where there are 10 stages in the computation, (b) shows the case where there are 15 stages in the computation, and (c) shows the case where there are 30 stages in the computation

2. *All-Stages Accuracy*

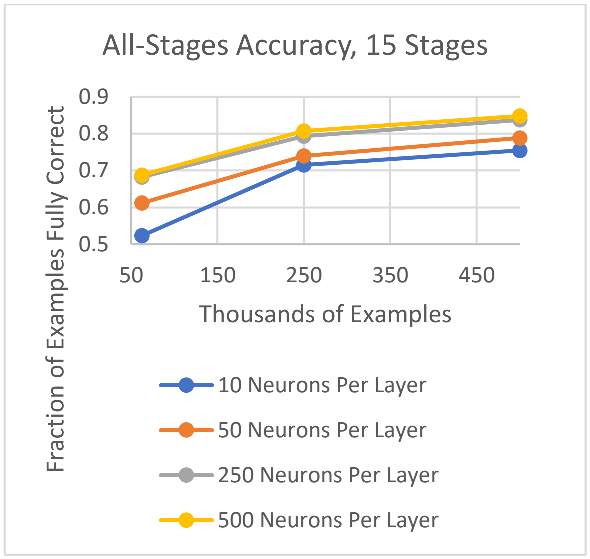
The average fraction of examples for which the execution location of all n stages is correctly assigned varies much more dramatically than the single-stage accuracy as the number of stages in the computation, the number of neurons per layer, and the number of training examples change. However, as with single-stage accuracy, the worst performance occurs in the network for 30 stages with 10 neurons per layer and 62,500 examples and the best performance occurs in the network for 10 stages with 500 neurons per layer and 500,000 examples.

Figure 2 (below) depicts the all-stages accuracy for all 36 networks. Notice that the networks with more stages struggle more to correctly classify all stage as local or remote than those with fewer stages. The best-performing network for 10-stage computations scored 90.9%, the best-performing network for 15-stage computations scored 84.8%, and the best-performing network for 30-stage computation scored only 73.9%. This is not a surprising result. We should expect that the probability of having an error in the classification of at least one of the n total stages in an example will be larger as n increases, particularly since we see from Figure 1 (above) that the assignment accuracy of a single stage varies very little as a function of the number of stages in a computational task.

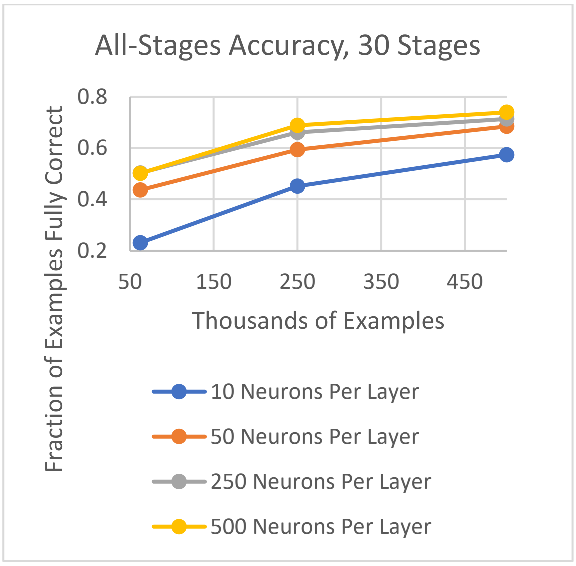
Of the networks trained on 500,000 examples, the fraction of examples where all stages are classified correctly is highest in the network with 500 neurons-per-layer for all task lengths. With two exceptions, the all-stages accuracy of these networks is statistically superior at the 95% confidence level to all other networks for tasks with the same number of stages. The two exceptions are the 250 neurons-per-layer networks for 10 and 15 stage computations (where $p = 0.1785$ for 10-stage tasks and $p = 0.0697$ for 15-stage tasks).



(a)



(b)



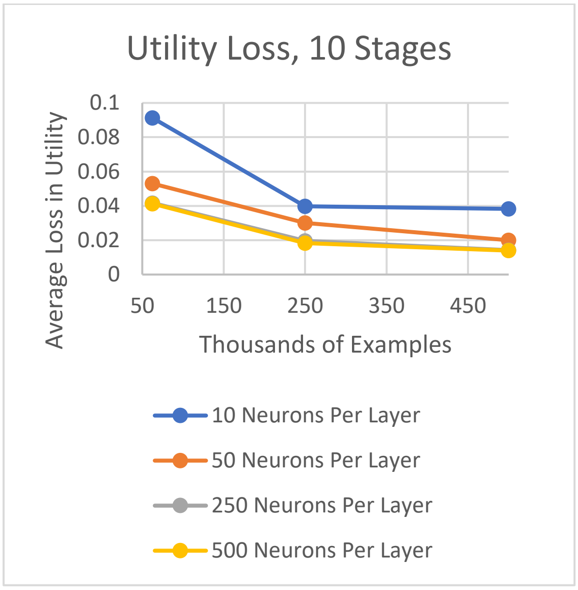
(c)

Figure 2: Learning curves showing the change in the average fraction of test examples where all n stages given the correct local/remote stage assignments for networks trained with various numbers of examples and numbers of neurons per layer. (a) shows the case where there are 10 stages in the computation, (b) shows the case where there are 15 stages in the computation, and (c) shows the case where there are 30 stages in the computation.

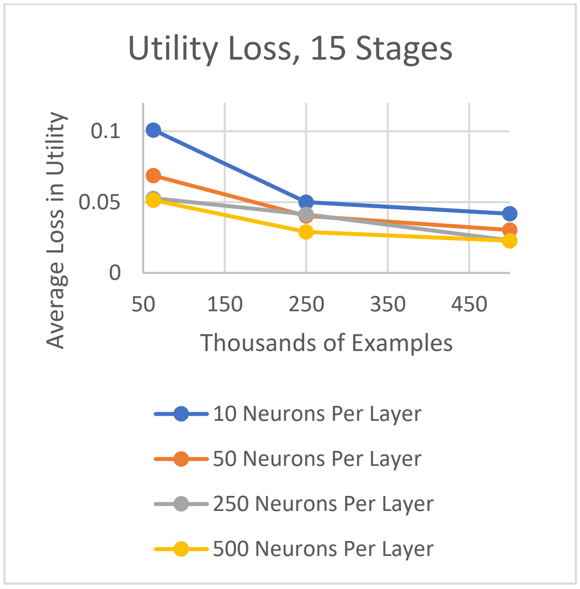
3. *Average Utility Loss*

Using the utility formulation presented in Section II, we compute the average difference between the optimal utility and the utility of the offloading assignments produced by the neural network. This allows us to evaluate how costly the network's mistakes tend to be to the system. The average value of this utility loss across all folds is shown in Figure 3 below. Notice that tasks with more stages have a higher average utility loss than those with fewer stages (for the same number of neurons per layer and number of training examples). For 10-stage tasks, the best-performing network had an average utility loss of 0.014. For 15-stage tasks, the best-performing network had a higher average utility loss of 0.023. For 30-stage tasks, the best-performing network had an even higher average utility loss of 0.034.

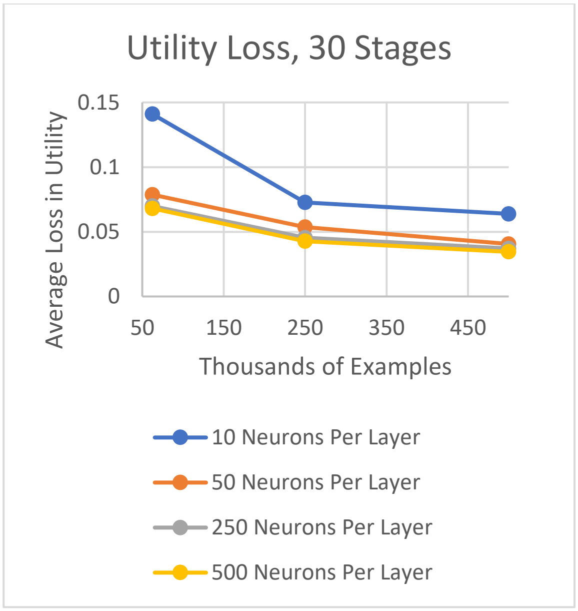
In all cases, the best-performing networks are those trained on 500,000 examples with 500 neurons per layer. With two exceptions, the utility loss of these networks is statistically superior at the 95% confidence level to all other networks for tasks with the same number of stages. The two exceptions are the 250 neurons-per-layer networks for 10 and 15 stage computations.



(a)



(b)



(c)

Figure 3: Learning curves showing the average utility loss for networks trained with various numbers of examples and numbers of neurons per layer. (a) shows the case where there are 10 stages in the computation, (b) shows the case where there are 15 stages in the computation, and (c) shows the case where there are 30 stages in the computation.

4. *Average Classification Time*

For each of the networks trained on 500,000 examples, we track the average time taken to evaluate a single test example using the multi-label neural network classifier. These measurements are shown below in Table 3. All measurements were taken on the same hardware configuration used when measuring the exhaustive search times in Table 2. All twelve averages are less than the corresponding exhaustive search time in Table 2 with an extremely high degree of statistical significance (p-values for all measures in a student's-t test are less than 0.0001).

Average Classification Times (ms)		10 Neurons per Layer	50 Neurons per Layer	250 Neurons per layer	500 Neurons per layer
10-Stage Tasks	Average	0.00124	0.00435	0.0200	0.04995
	Standard Deviation	6.372e-5	6.757e-5	0.0021	0.0032
15-Stage Tasks	Average	0.00153	0.00462	0.0224	0.04945
	Standard Deviation	3.465e-5	0.0004	0.0010	0.0062
30-Stage Tasks	Average	0.00226	0.00610	0.0248	0.02939
	Standard Deviation	0.0002	0.0003	0.0022	0.0011

Table 3: The average classification time, in milliseconds, of the multi-label neural network for various combinations of task lengths and network complexities. Averages taken across folds.

d. *Discussion*

Overall performance of the multi-label neural network approach is quite promising across all metrics. We see that it is possible to train a network where there is a greater than 95% chance of selecting the correct execution location for a particular stage, regardless of how many stages are in the overall task. The fraction examples where the correct execution location is selected for all stages decreases as the number of stages per task increases, but in the best networks this value exceeds 90% for 10-stage tasks and 73% for 30-stage tasks. Even when errors are made, they tend not to cost the system much total utility. The average utility loss increased as the number of stages increased, but even for the 30-stage task, we see that the average utility loss was only 0.034. Since the allowable range of utility values in the utility formulation from Section II is between 0 and 1, this suggests that an application with 30 or fewer stages could use a multi-label neural network classifier for execution location assignments in dynamic computational offloading and surrender, on average, no more than 3.4% of the total possible utility.

Whether errors are small enough and rare enough to allow for the use of a multi-label neural network classifier in a deployed system that conducts computational offloading system depends, obviously, on requirements unique to the specific application. However, the fact that the accuracy is generally good and the utility cost of a mistake generally small suggests that there are probably applications where this approach could be successfully utilized.

The primary benefit of utilizing this multi-label neural network classifier approach is significantly faster decisions about where each stage should be executed. Even for the most-complex, slowest-to-evaluate neural networks – those with 500 neurons per hidden layer – the average example evaluation time using the neural network is nearly an order of magnitude faster than the corresponding time required to select stage execution locations using exhaustive search.

There was a clear pattern that emerged in the tests of statistical significance conducted on single-stage accuracy, all-stages accuracy, and utility loss. For each metric, we found the networks with 500 neurons per hidden layer to perform significantly better than those with 10 or 50 neurons per layer regardless of the number stages in the task. For the 30-stage tasks, we also found that the 500 neurons-per-layer networks performed significantly better than the networks with 250 neurons per layer. However, the 500 neurons-per-layer networks did not significantly outperform the 250 neurons-per-layer networks on tasks with 10 or 15 stages. This suggests that for tasks of 15 stages or fewer, 250 neurons per layer is sufficient but that for tasks with more stages, additional neurons in each hidden layer can help improve performance.

We also see from the learning curves for single-stage accuracy and all-stages accuracy that a very large number of examples are required to train the network well. While the learning curve for single-stage accuracy is nearly flat between 250,000 and 500,000 examples, the learning curve for all-stages accuracy still has a significant positive slope. This suggest that, with additional training examples, even better network performance may be possible.

IV. Reinforcement Learning Approach

a. UCB1 Reinforcement Learning

UCB1 (upper confidence bound) learning is a reinforcement learning technique for choosing actions in bandit problems [8]. A bandit problem is a Markov Decision Process (MDP) with only a single state; from this state the system is permitted to take various actions, each of which yields a reward given by some unknown probability distribution [8]. The goal of the system is to learn which action to take to maximize the reward [8].

The approach UCB1 takes towards bandit problems is **optimism in the face of uncertainty**; the algorithm maintains upper confidence bounds for the potential reward of each action and chooses the action that with the highest upper confidence bound [8]. After taking an action, it adjusts the upper confidence bound for that action in response to the reward it received for taking it [8].

Specifically, UCB1 models the upper confidence bound on an action a as follows, where t is an integer time-step counter representing the total number of actions the system has previously made, $r_t(a)$ is the average reward the system received for taking an action a through time t, and $n_t(a)$ is the number of times the system has selected action a through time t. The reward for an action is permitted to range from $-R$ to $+R$. [8]

$$UCB(a) = r_t(a) + R\sqrt{\frac{2\log(t)}{n_t(a)}}$$

UCB1 can be applied in computational offloading to the problem of identifying the best execution location for each stage. In this formulation, the reward is the realized utility, which we must rescale to range from -1 to +1 (instead of 0 to 1). The realized utility can be computed using the equations presented in Section II. In a deployed system, the parameters needed to perform this computation can be observed directly by timing (i) the amount of time spent executing stages locally, (ii) the amount of time spent transmitting messages, and (iii) the amount of time spent waiting on remote computation. In the simulations we use to evaluate this approach, the required parameters are obtained from sequences of examples generated as described in Section II.

b. Implementation, Experiments, and Results

The UCB1 reinforcement learner was implemented directly in Python without the use of external machine learning libraries. For tasks with 10 stages, 15 stages, and 30 stages, we generated we generated 500 sequences of 2,000 examples and 10 sequences of 25,000 examples on which to test the UCB1 reinforcement learner using the example generation procedure described in Section II.

As with the neural network approach, we evaluate the performance of the UCB1 reinforcement learner using the metrics listed below.

1. **Single-Stage Accuracy**: the fraction of stages for which the correct local/remote execution assignment is chosen
2. **All-Stages Accuracy**: the fraction of examples for which the correct local/remote execution assignments are chosen for all n stages
3. **Utility Loss**: the average difference between the optimal utility and the utility of the stage assignment chosen by the reinforcement learner
4. The average **decision time** needed for the reinforcement learner to choose an action

The average values of these performance metrics for each data set are shown in Table 4 (below). All measurements were taken on the same hardware configuration used when measuring the exhaustive search times in Table 2.

UCB1		500 Example Sequences of Length 2,000		10 Example Sequences of Length 25,000	
		Mean	Standard Deviation	Mean	Standard Deviation
10-Stage Tasks	Single-Stage Accuracy	0.6950	0.0459	0.8238	0.1307
	All-Stages Accuracy	0.3132	0.1071	0.7026	0.2189
	Utility Loss	0.2099	0.1932	0.0801	0.1396
	Decision Time (ms)	0.0540	0.0138	0.0591	0.0109
15-Stage Tasks	Single-Stage Accuracy	0.6733	0.0387	0.7243	0.0546
	All-Stages Accuracy	0.1443	0.0825	0.2630	0.1409
	Utility Loss	0.2673	0.2308	0.1342	0.1724
	Decision Time (ms)	0.0962	0.0414	0.1159	0.0280
30-Stage Tasks	Single-Stage Accuracy	0.6675	0.0400	0.7748	0.0526
	All-Stages Accuracy	0.0107	0.0355	0.1665	0.1412
	Utility Loss	0.2726	0.2281	01512	0.1976
	Decision Time (ms)	0.1314	0.0496	0.3540	0.1839

Table 4: Performance of the UCB1 reinforcement learning approach

c. Discussion

There is little in these results to recommend using UCB1 reinforcement learning for assigning stage execution locations in computational offloading problems. The average values of all four performance metrics are uniformly much lower than the performance we achieved using the multi-label neural network in Section III. Furthermore, with the possible exception of the case where tasks have 10-stages and the example sequences are long, the performance measures are so uniformly bad that they are unlikely to be acceptable in any real-world deployment of dynamic computational offloading.

Just about the only positive result is that all four measures of the decision time are significantly lower than the corresponding exhaustive search times in Table 2 (in all cases, the p-value from a t-test is less than 0.0001). This suggests that the UCB1 reinforcement learner unambiguously makes offloading decisions faster than an exhaustive optimizer. However, those offloading decisions are so poor as to render this advantage irrelevant. Furthermore, the decision times of the multi-label neural network approach are superior to the UCB1 decision times.

Why does the UCB1 approach yield such poor results? I believe the primary factor is that UCB1 has no mechanism for recognizing that two similar actions are likely to yield similar rewards. Typically, the total utility of offloading the first stage of a 10-stage task and computing the remaining 9 stages locally is very likely to be somewhat similar to the total utility of offloading the first two stages and computing the remaining 8 stages locally. UCB1 treats the two cases as entirely independent, which means it needs to explore both cases to be able to reduce the upper confidence bound on both. The 25,000-example sequences have much more time to do this, which explains why the UCB1 performance on those sequence is modestly higher than on the shorter 2,000-example sequences. A reinforcement learner that models the correlations between the reward for one action and the reward for a similar action would likely see significantly improved performance.

V. Conclusions

In this paper, we explore how to efficiently make decisions about which stages of a computation should be executed locally and which should be executed remotely for systems that engage in dynamic computational offloading. We study two alternatives to the basic optimization-by-exhaustive search strategy. The first, a multi-label classifier implemented using a multi-output neural network, proved able to output very accurate execution location assignments for the stages of a task while taking significantly less time, on average, to produce these assignments than the exhaustive-search strategy. As a result, this approach seems viable in many applications of dynamic computational offloading. The second approach, a UCB1 reinforcement learner, proved to have poor overall performance. One issue that was likely a significant factor in UCB1's failure on this problem is the algorithm's inability to recognize correlations between the rewards for similar actions. One possible avenue for future work would be to apply an alternative reinforcement learning algorithm (or to modify UCB1) to recognize these correlations.

References

[1] K. Kumar, J. Liu, Y.-H. Lu and B. Bhargava, "A Survey of Computation Offloading for Mobile Systems," *Mobile Networks and Applications,* vol. 18, no. 1, pp. 129-140, 2013.

[2] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall and R. Govindan, "Odessa: Enabling Interactive Perception Applications on Mobile Devices," in *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, Bethesda, Maryland, USA, 2011.

[3] Y.-H. Kao, B. Krishnamachari, M.-R. Ra and F. Bai, "Hermes: Latency Optimal Task Assignment for Resource-constrained Mobile Computing," *IEEE Transactions on Mobile Computing,* vol. 16, no. 11, pp. 3056-3069, 2017.

[4] J. Flinn, D. Narayanan and M. Satyanarayanan, "Self-tuned remote execution for pervasive computing," in *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, Elmau, Germany, 2001.

[5] J. Flinn, S. Y. Park and M. Satyanarayanan, "Balancing Performance, Energy, and Quality in Pervasive Computing," in *Proceedings of the 22nd International Conference on Distributed Computing Systems*, Vienna, Austria, 2002.

[6] J. Veltri, *Computational Offloading for Sequentially Staged Tasks: A Dynamic Approach Demonstrated on Aerial Imagery Analysis,* M.S. thesis, Case Western Reserve University, December 2017.

[7] "Raspberry Pi 3 is Out Now! Specs, Benchmarks & More," The MagPi: The Official Raspberry Pi Magazine, [Online]. Available: https://www.raspberrypi.org/magpi/raspberry-pi-3-specs-benchmarks/. [Accessed September 2017].

[8] C. Szepesvari, "Algorithms for Reinforcement Learning," in *Sythesis Lectures on Artificial Intelligence and Machine Learning*, Morgan & Claypool Publishers, 2009.