Learning to Estimate Search Progress Using Sequence of States

Matan Sudry, Erez Karpas

Technion — Israel Institute of Technology, Haifa, Israel
matansudry@campus.technion.ac.il, karpase@technion.ac.il

Abstract

Many problems of interest can be solved using heuristic search algorithms. When solving a heuristic search problem, we are often interested in estimating search progress, that is, how much longer until we have a solution. Previous work on search progress estimation derived formulas based on some relevant features that can be observed from the behavior of the search algorithm. In this paper, rather than manually deriving such formulas we leverage machine learning to automatically learn more accurate search progress predictors. We train a Long Short-Term Memory (LSTM) network, which takes as input sequences of nodes expanded by the search algorithm, and predicts how far along the search we are. Importantly, our approach still treats the search algorithm as a black box, and does not look into the contents of search nodes. An empirical evaluation shows our technique outperforms previous search progress estimation techniques.

Introduction

Many interesting and challenging real-world problems can be defined as search problems, where the objective is to find a path in a state space (which is often implicitly defined by a black-box successor generator) from some given initial node to a goal node. Often, we can use a heuristic function which estimates the distance from any given node to the goal to better guide this search. Many heuristic search algorithms have been developed to solve such problems, the most well-known of which are probably A^* (Hart, Nilsson, and Raphael 1968) and Greedy Best-First Search (GBFS) (Doran and Michie 1966).

Search algorithms can have unpredictable behavior, as we typically do not know the topology of the search space (if we did, perhaps we would not need to use a search algorithm). While some theoretical results exist for cases where the search problem is described symbolically (Hoffmann 2011), these results do not apply for black-box search problems. Other work has addressed estimating search effort in terms of number of expanded nodes (Lelis, Stern, and Sturtevant 2014; Belov et al. 2017) or the cost of an optimal solution (Lelis et al. 2014) – we discuss these and others in more detail later. Importantly, these approaches are all *offline* – they give their predictions before the search starts.

We are interested in predicting remaining search effort *online*, that is, in estimating how much progress the search algorithm has already made. This problem was first introduced by Thayer, Stern, and Lelis (2012), who also derived some formulas for estimating search progress based on relatively simple search features, which we explain in more detail later.

Search progress estimation is a useful tool not just for letting a user know whether they have time to make a cup of coffee before search completes, but also in making decisions about the search. For example, remaining search time estimates are used in situated temporal planning (Cashmore et al. 2018; Shperberg et al. 2021) to prioritize search nodes which have a higher probability to lead to a timely solution. Such estimates can also be used to decide when to perform a restart in a Branch-and-Bound search (Anderson et al. 2019) for solving a Mixed Integer Program.

In this paper, we focus on search progress estimation (not its above mentioned applications), and leverage recent developments in machine learning to automatically learn more complex search predictors based on a given training set. Specifically, we view the search algorithm as a black-box that emits a sequence of nodes, and train a Long Short-Term Memory (LSTM) network (Hochreiter and Schmidhuber 1997) to predict remaining search effort. An empirical evaluation on a large set of planning domains shows that our technique achieves state-of-the-art performance in search progress estimation, outperforming previous work (Thayer, Stern, and Lelis 2012).

Background

In this section we review some of the necessary background on LSTM networks and heuristic search.

Heuristic Search

As previously mentioned, many problems of interest can be modeled as state-space search problems (Bonet and Geffner 2001). A state space is formally represented as a tuple $T = \langle S, S_0, S_G, A, f, c \rangle$, where:

- S is a finite and non-empty set of states,
- $S_0 \in S$ is the initial state,
- $S_G \subseteq S$ is a non-empty set of goal states,
- $A(s) \subseteq A$ denotes the actions applicable in each state $s \in S$,

- $f : S \times A \to S$ is the state transition function, such that applying action a in state s leads to state $f(s,a)$, and
- $c(s,a)$ is the cost of performing action a in state s

Uninformed search algorithms such as Breadth-First Search (BFS) can be used to solve small state-space search problems. However, to solve larger problems, we typically also use a heuristic which estimates the distance from any given node to the goal. Formally, a heuristic is a function mapping the nodes S to some real number (or infinity to indicate a dead-end), that is $h : S \to \mathbb{R}_+ \cup \{\infty\}$.

Most search algorithms define some priority function $f(s)$ over the nodes. Typically, $f(s)$ is obtained from $h(s)$, the heuristic estimate of the cost to go from s, and $g(s)$, the cost of the best known path to s so far. For example, A^* (Hart, Nilsson, and Raphael 1968) expands the node with the lowest $f_{A^*}(s) = g(s) + h(s)$, while GBFS expands the node with the lowest $f_{GBFS}(s) = h(s)$.

Search Progress Estimation

We can now formally describe the problem we address in this paper – search progress estimation. Our definition here is adapted from previous work (Thayer, Stern, and Lelis 2012). Let $\mathbb{A}$ be a search algorithm, and $P = \langle T, h \rangle$ be a heuristic search problem, which consists of a search space T and a heuristic function h. Denote by $\mathrm{E}_{\mathbb{A}}(P)$ the total number of nodes expanded by search algorithm $\mathbb{A}$ while solving search problem P. Let $\mathrm{Rem}_{\mathbb{A}}(P, \mathrm{Gen}_{\mathbb{A}}(P))$ be the number of remaining nodes that are going to be expanded by $\mathbb{A}$ when solving P after $\mathbb{A}$ has already expanded $\mathrm{Gen}_{\mathbb{A}}(P)$ nodes, that is, $\mathrm{Rem}_{\mathbb{A}}(P, \mathrm{Gen}_{\mathbb{A}}(P)) = \mathrm{E}_{\mathbb{A}}(P) - \mathrm{Gen}_{\mathbb{A}}(P)$.

Definition 1 (Search Progress). *The search progress of algorithm $\mathbb{A}$ solving problem P after expanding $Gen_{\mathbb{A}}(P)$ nodes is:* $Prog_{\mathbb{A}}(E_{\mathbb{A}}(P)) = \dfrac{Gen_{\mathbb{A}}(P)}{Gen_{\mathbb{A}}(P) + Rem_{\mathbb{A}}(P, Gen_{\mathbb{A}}(P))}$

As Definition 1 shows, search progress is a number between 0 and 1, which indicates what fraction of the total search effort to solve the problem we have already expended. For example if we expanded node number 1,000 out of 10,000 nodes that are going to be expanded in total, the search progress will be $1,000/10,000 = 0.1$. This allows comparing search progress between problems of different sizes on the same scale.

LSTM Networks

Long Short-Term Memory (LSTM) networks (Hochreiter and Schmidhuber 1997) are a specific type of Recurrent Neural Network (Medsker and Jain 2001) – a deep learning approach which is especially suitable for learning from sequences. LSTMs were first introduced in order to solve the vanishing gradient problem in conventional Recurrent Neural Networks (Hochreiter and Schmidhuber 1997), and have been extensively used in natural language processing, e.g., (Sutskever, Vinyals, and Le 2014; Gers, Schmidhuber, and Cummins 2000). We now briefly review how LSTM networks are built, but refer the interested reader to a detailed survey (Yu et al. 2019) for more details.

An LSTM network consists of several LSTM cells, arranged in sequence. Each LSTM cell has a hidden state, and

3 gates: input, forget and output, that manage the reading, writing and memory update, respectively. The cell learns gate weights and uses them to change the output and the hidden state based on the previous input in the sequence.

The cell processes input x_t together with the hidden state h_{t-1} (which is the result of processing the previous input) by first passing them to the forget gate, which has a Sigmoid activation function. The output of the forget gate is a number between 0 and 1 for each element in the cell state C_t – the information we are passing to the next LSTM cell.

Next, the processing continues through the input gate, which has 2 activation functions, Sigmoid and tanh. The input is the same as the forget layer: x_t and h_{t-1}. The Sigmoid activation decides which values to update and the tanh activation creates the values themselves. The outputs from both of these are multiplied.

Finally, processing continues to the output gate that gets x_t and h_{t-1} as inputs and goes through a Sigmoid activation which decides what parts will be output. The output also goes through a tanh activation, which is multiplied by the output of the Sigmoid activation.

This is all captured by the following equations, which describe the operation of each LSTM cell:

- forget gate: $f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$
- input gate: $i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$
- new cell state value: $\widetilde{C_T} = tanh(W_c \cdot [h_{t-1}, x_t] + b_C)$
- update cell state: $C_T = f_t \cdot C_{t-1} + i_t \cdot \widetilde{C_T}$
- output gate: $o_t = \sigma(W_0[h_{t-1}, x_t] + b_0)$
- hidden state: $h_t = o_t \cdot tanh(C_t)$

Where W indicate learned weights and b a learned constant for different gates.

Related Work

Having described the necessary background, we can now review related work in this field. Some previous work attempted to predict search effort in specific domains, e.g., predicting the number of nodes expanded by A^* for the 15-puzzle (Breyer and Korf 2008). Other work used sampling to estimate search effort (Lelis et al. 2014; Belov et al. 2017; Hutter et al. 2014) or the cost of a solution (Lelis, Stern, and Sturtevant 2014) in a more general setting. However, as previously mentioned, these are all *offline* methods, which attempt to make a prediction before the search starts. We now review in more detail the only prior work we are aware of that addresses *online* search progress estimation (Thayer, Stern, and Lelis 2012). Several search progress estimators were proposed, which we now describe, and later use as baselines in our empirical evaluation:

Velocity-Based Search Speed Estimator (VeSP) The Velocity-Based Search Speed Estimator (VeSP) calculates the velocity of heuristic decrease during the search as:

$$V = \frac{h_0 - h_{\min}}{\mathrm{Gen}_{\mathbb{A}}(P)}$$

where $h_{\min}$ is the minimum h-value of any nodes generated by the search so far and h_0 is the heuristic value of

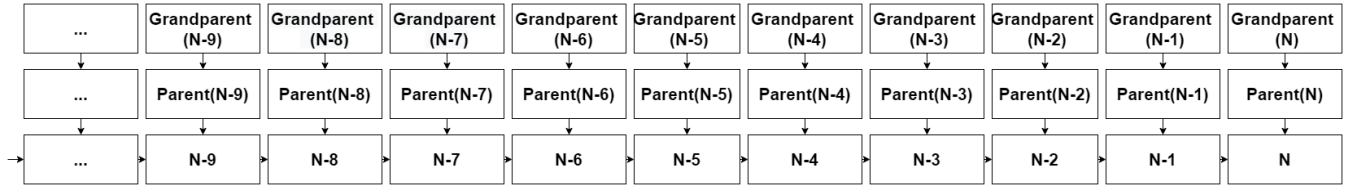| ...	Grandparent (N-9)	Grandparent (N-8)	Grandparent (N-7)	Grandparent (N-6)	Grandparent (N-5)	Grandparent (N-4)	Grandparent (N-3)	Grandparent (N-2)	Grandparent (N-1)	Grandparent (N)
...	Parent(N-9)	Parent(N-8)	Parent(N-7)	Parent(N-6)	Parent(N-5)	Parent(N-4)	Parent(N-3)	Parent(N-2)	Parent(N-1)	Parent(N)
...	N-9	N-8	N-7	N-6	N-5	N-4	N-3	N-2	N-1	N

Figure 1: Illustration of the nodes used as input for prediction.
N is the node that was expanded last, $N-1$ the node expanded before it, and so on. Parent() and Grandparent() are obtained by following the parent pointers for the indicated node.

the initial state. It then uses the velocity V and the current $h_{\min}$ to estimate the remaining search effort (number of expanded nodes)

$$SE_V = \frac{h_{\min}}{V}$$

Finally, it uses this estimate to predict search progress by using:

$$VeSP(\text{Gen}_{\mathbb{A}}(P)) = \frac{\text{Gen}_{\mathbb{A}}(P)}{\text{Gen}_{\mathbb{A}}(P) + SE_V}$$

Vacillation-Based Search Speed Estimator (VaSP) The Vacillation-Based Search Speed Estimator (VaSP) differs from VeSP by using node serial numbers to estimate the *expansion delay* (Dionne, Thayer, and Ruml 2011), that is, the average number of expansions between when a node is generated and when it is expanded, denoted $\overline{\Delta e}$.

The expansion delay is combined with $h_{\min}$ to predict remaining search effort as:

$$SE_e = \overline{\Delta e} \cdot h_{\min}$$

As in VeSP, the estimate of search progress is:

$$VaSP(\text{Gen}_{\mathbb{A}}(P)) = \frac{\text{Gen}_{\mathbb{A}}(P)}{\text{Gen}_{\mathbb{A}}(P) + SE_e}$$

Path-Based Progress Estimator (PBP) We first describe the Naive PBP (NPBP), which estimates search progress at a given node n by looking at the ratio between the cost until the current node ($g(n)$) and the total estimated cost ($g(n) + h(n)$),

$$NPBP(Exp) = \frac{g(n)}{g(n) + h(n)}$$

The main problem with NPBP is that it depends on the current node, but in many cases there are other open nodes with higher progress. Thus the PBP estimator returns the maximum NPBP estimate among the nodes expanded so far.

Distribution-Based Progress Estimator (DBP) The Distribution-Based Progress Estimator (DBP) estimates the search progress using data observed during search. It relies on counting how many nodes are expanded by the search for each value of $d(n)$ – an estimate of remaining plan *length*. We remark that in unit-cost

domains $d(n) = h(n)$, and where action costs are non-uniform, $d(n)$ can usually be estimated alongside $h(n)$. Define $c[d_i]$ as the count of how many nodes had value $d(n) = d_i$ during search. DBP fits a second degree polynomial $\hat{c}$ to c, in order to estimate progress. The DBP estimate is then defined as:

$$Prog^*(\text{Gen}_{\mathbb{A}}(P)) = \frac{\text{Gen}_{\mathbb{A}}(P)}{\sum_{i=1}^{m} \hat{c}[d_i]}$$

where m is the highest d-value with a count higher than 0.

Learning to Predict Search Progress

We now describe our technique for search progress estimation. Our technique exploits the fact that different heuristic searches can behave similarly to each other, especially when the search space, search algorithm, or heuristic are similar between the different searches. Thus, our technique uses supervised machine learning, training a search progress estimator on a set of already solved search problems. As our empirical evaluation (on a test set consisting of unseen search problems) will show, our technique is able to generalize across different problems, different domains, different search algorithms, and different heuristics.

Our technique leverages the observation that any expansion-based forward search search algorithm can be seen as a black box which generates a sequence of nodes to be expanded, which we call a *search history*. Assume once search completes, the sequence of expanded nodes will be $\langle s_1, s_2, \ldots, s_n \rangle$. After expanding m nodes, the partial search history is $\langle s_1, s_2, \ldots, s_m \rangle$, which is the input to our predictor. The correct label that we would like to predict would then be $\frac{m}{n}$.

As we are attempting to predict from a sequence of nodes of varying length, it is natural to use a Recurrent Neural Network (Medsker and Jain 2001). In the remainder of this section we describe in more detail how we represent a partial search history as input features to the network, the network structure itself, and how we train it.

Input Features: To represent a search history as input to our network, we look at the last k nodes that were expanded. However, for each of these nodes, it is also important to know something about where it came from – the path from the initial node to that node. Thus, for each of these k nodes, we also look at their ancestors m layers up. In this paper, we used used different values of k varying between 1 and

40 and fixed $m = 2$, resulting in collecting features from $k \cdot (m + 1)$ nodes, as illustrated in Figure 1.

For each of these $k \cdot (m + 1)$ nodes, we collect the following features:

- $g(n)$ – the cost of the best known path to node n.
- $h(n)$ – the heuristic estimate of the cost from n to the goal.
- $f(n)$ – the f-value for node n (which varies with the search algorithm).
- $b(n)$ – the branching factor of n, that is, the number of successors n has.
- $N(n)$ – the serial number of n, that is, how many nodes were expanded before n. We remark that this number was previously used to measure the expansion delay (Dionne, Thayer, and Ruml 2011).

Additionally, we use some global features, which capture a snapshot of the current state of the search:

- $h_0 = h(S_0)$ – the heuristic value of the initial state. This value stays constant throughout the search, but is important for estimating the cost of the solution that will be returned.
- $h_{\min}$ – the minimal h-value we have seen so far among the expanded nodes.
- $Nh_{\min}$ – the number of nodes we expanded since the last time $h_{\min}$ was updated.
- $f_{\max}$ – the maximum f value we have seen so far.

These features are used in the previous work (Thayer, Stern, and Lelis 2012), with the exception of $Nh_{\min}$, which is used to get some information about whether search seems to be stuck in a heuristic plateau – an area of the search space where the heuristic assigns similar values to all nodes, and thus heuristic guidance is not very useful. Search algorithms typically spend most of their time in heuristic plateaus, as otherwise, the heuristic would lead the search algorithm directly to the goal.

We record the value of the global features at the time each node was expanded, and use these values for each of the last k states as features. These input features can be seen as a matrix of real numbers, of dimension $5k(m + 1) + 4k$, which are used as the input (features) to our network. Note that for the first k nodes, we use feature values of 0 for the undefined nodes (i.e., nodes with a negative serial number).

Network Structure: Our network[1] consists of 15 LSTM layers, followed by a fully connected layer which reduces the dimension by half (from $15 * k$ to $15 * k/2$), then a dropout layer with 50% dropout and Rectified Linear Units (ReLU) activations, and finally another fully connected layer which outputs our prediction – a single number. The parameters here (number of LSTM layers, dimensions, dropout rate, etc.) were determined by manual tuning over preliminary data. Automatically setting these parameters is future work.

---

[1]https://github.com/matansudry/Learning_to_Estimate_Search_Progress_Using_Sequence_of_States

Domain	VaSP	VeSP	RF	OD
Arprt(2)	**26.4(10.7)**	33.7(0.6)	54.9(1.3)	29.3(0.1)
Blck(10)	29.8(7.6)	24.8(4.5)	39.8(5.7)	**24.2(4.6)**
Depot(3)	31.2(4.5)	22.5(3.4)	35.7(8.5)	**21.1(3.1)**
Elev(2)	33.1(2.7)	25.7(10.7)	36.3(4.7)	**16.9(0.1)**
Frcll(3)	26.1(4.1)	28.0(5.7)	35.8(8.6)	**20.5(5.1)**
OS(16)	32.8(5.7)	48.8(9.3)	34.7(16.5)	**21.0(3.5)**
Pegsol(4)	37.3(11.7)	42.7(12.0)	**22.7(11.1)**	23.4(6.3)
PSR(9)	38.9(9.0)	30.7(4.7)	24.9(8.6)	**23.4(4.8)**
Rover(4)	33.9(2.7)	**25.1(7.7)**	29.3(8.5)	27.3(4.5)
Sat(2)	27.5(4.5)	23.0(4.2)	23.9(10.4)	**19.9(2.3)**
Scnlzr(1)	28.2(0.0)	37.0(0.0)	**20.6(0.0)**	24.0(0.0)
Soko(19)	45.3(9.2)	31.7(7.3)	40.1(11.2)	**29.4(5.0)**
TPP(7)	35.4(13.8)	38.4(8.1)	20.7(8.7)	**19.9(4.1)**
Trns(6)	36.7(9.1)	36.0(9.6)	27.4(10.3)	**21.9(4.5)**
WW(1)	44.0	31.5(0.0)	53.5(0.0)	**18.5(0.0)**
Avgd(15)	33.4(5.9)	32.9(7.5)	33.2(9.6)	**23.3(3.7)**
Avgp(89)	36.5(3.1)	35.3(3.3)	34.0(3.6)	**24.0(5.5)**

GBFS with Lmcut

Table 1: RMSE (in percent) and standard deviation in parentheses, using the OD regime and GBFS with Lmcut. Avgp is average over RMSE in each problem, and Avgd is the average over average RMSE in each domain. Results in bold are the best performance in this row. Arprt=Airport, Blck=Blocks, Elev=Elevators, Frcll=Freecell, OS=Openstacks, Parcp=Parcpriter, PSR=Psrsmall, Sat=Satellite, Scnlzr=Scanalyzer, Soko=Sokoban, Trns=Transport and WW=Woodworking

Training: Given a planning problem, we first solve it using a planner running a given search algorithm and heuristic, and generate the true label (the correct search progress) for each node. During training, we sample 1,000 nodes uniformly at random from each problem in our dataset, in order to avoid bias to larger problems. This corresponds to the objective of optimizing the accuracy for a randomly chosen problem. If the objective had been to optimize the accuracy for a randomly chosen node, then it would be better to sample more nodes from the bigger problems. We use the node serial number to obtain the true label of search progress – for node n the true label is $N(n)/N(g_n)$, where g_n is the goal node found by the search algorithm.

We used Adam (Kingma and Ba 2014) as an optimiser with a learning rate of 0.001 and a batch size of 1024, and Mean Squared Error (MSE) as the loss function. Training is relatively fast, taking at most 12 minutes to train a search-effort predictor on any given domain.

Empirical Evaluation

Having described our technique and the previous work on search progress estimation, we can now compare them empirically. We now describe this empirical evaluation.

Benchmarks

In order to compare search progress estimates on various problems with different heuristics and different search algorithms, we chose to use planning benchmarks from past International Planning Competitions, as this allows us to im-

Domain	VeSP	DBP	OD
Arprt(6)	46.5(10.8)	44.8(8.5)	**25.1(5.3)**
Blck(20)	38.0(6.6)	43.6(8.5)	**21.8(3.3)**
Depot(19)	**25.7(3.9)**	27.2(6.8)	26.3(4.1)
Elev(27)	27.9(6.9)	31.8(5.8)	**27.3(3.1)**
Frcll(14)	**18.0(7.1)**	25.8(11.0)	20.2(4.0)
Gripper(4)	57.7(0.0)	54.9(3.7)	**24.3(3.9)**
Logistics(11)	**17.9(2.9)**	24.2(7.4)	26.7(3.9)
Miconic(5)	42.6(5.1)	30.7(5.6)	**20.6(2.9)**
OS(6)	29.4(2.5)	**23.5(2.7)**	23.8(3.0)
Parcp(8)	24.7(3.3)	26.0(4.8)	**23.6(5.9)**
Pegsol(18)	35.8(11.1)	48.8(6.6)	**26.5(3.8)**
PSR(12)	48.3(14.1)	43.9(9.8)	**25.3(4.2)**
Rover(4)	**22.4(3.9)**	35.8(7.6)	26.0(1.3)
Sat(2)	33.3(0.0)	28.9(0.0)	**24.7(0.2)**
Scnlzr(5)	37.2(14.4)	42.9(7.4)	**26.1(3.7)**
Soko(20)	31.7(5.8)	40.7(7.8)	**28.5(4.7)**
TPP(4)	25.8(4.3)	24.3(6.5)	**23.9(3.4)**
Trns(8)	28.3(6.6)	29.3(7.7)	**25.1(4.6)**
WW(8)	27.9(7.1)	45.0(12.9)	**23.2(4.8)**
Zenotravel(3)	35.3(7.3)	27.1(1.7)	**25.5(4.8)**
Avgd(20)	34.2(10.2)	36.2(9.7)	**24.8(2.2)**
Avgp(204)	33.0(11.6)	37.3(11.7)	**25.2(4.5)**

GBFS with h_{FF}

Table 2: RMSE (in percent) and standard deviation in parentheses, using the OD regime and GBFS with h_{FF}. Results in bold are the best performance in this row. See caption of Table 1 for details

plement our technique only once. Specifically, we extended Pyperplan (Alkhazraji et al. 2020) to output the features of the nodes it expands during search. This data is then used for training and testing of search progress estimators, as we describe next.

Our dataset started with planning problems from 21 IPC domains – specifically, the benchmarks which are part of the Pyperplan repository. These domains have a total of 605 planning problem instances. For each instance, we ran 4 configurations of Pyperplan, choosing one of two search algorithms (A* or GBFS) with one of two heuristics: h_{FF} (Hoffmann and Nebel 2001) or Lmcut (Helmert and Domshlak 2009). We ran the planner with a time limit of 24 hours and a memory limit of 1,000,000 expanded nodes. From the solved instances we omitted the instances which were solved using less than 1,000 expanded nodes, as these are solved so quickly that search progress estimation is useless, and it would have made sampling 1,000 nodes from each instance difficult.

Experimental Setup

We compare our technique to the techniques introduced by previous work (Thayer, Stern, and Lelis 2012), which we described in detail above: VaSP, VeSP, PBP, and DBP. For VaSP, we used a moving average over the last 200 nodes to estimate expansion delay. For DBP we used NumPy to fit the polynomial to the data at hand. We also compared to flat machine learning approaches based on the same set of features as our LSTM. Specifically, we used Random Forest (RF) (Breiman 2001), which is a state-of-the-art classifier.

All of our experiments were run on a server with 72 Intel Xeon E5-2695 CPUs (utilizing at most 18 processes in parallel (Tange et al. 2011). The deep learning was implemented in PyTorch, and run on 2 Nvidia Tesla M60 GPUs. We remark that this GPU was released more than 5 years ago and in current computers the GPU should be a more advanced version. On this hardware the inference takes 0.05ms per node and the collection time of the data is minor because the algorithms already calculate it.

As our technique is based on learning from a training set, we examine three different training/test regimes:

Other Domains (OD): In this regime, when we test on problems from some domain, we train on all problems from all *other* domains. This simulates the setting where we need to estimate search progress on a problem from a completely unknown domain.

Same Domain (SD): In this regime, when we test on problems from some domain, we train only on problems from the same domain. To have some meaningful learning, we only used domains with more than 15 instances. For each domain, we split its instances into the even- and odd-numbered problems, to obtain two sets of roughly the same size, with roughly the same distribution on problem difficulty. We then trained a predictor on one set, and evaluated its performance on the other.

Other Domains Tune Same (ODTS): In this regime, we first train a predictor based on instances from the other domains. We then take this trained model, and fine tune it on one half of the instances from the target domain (split, as before, to even and odd numbered problems). During this fine tuning we run the same training algorithm, except that the learning rate is 0.0001 and the batch size is 2048. As fine-tuning requires less data than SD, we used domains with more than 6 instances.

We measure the accuracy of each search progress estimate prediction (for a given node) by the squared difference between the prediction and the true progress (which we have, since we use only solved problems). The accuracy of a search progress estimator over a given problem is the root of the average accuracy of its predictions over all the nodes expanded by the algorithm, that is the Root mean square error (RMSE). The accuracy for a set of problems (all problems in a domain, or all problems overall) is the average RMSE across these problems (giving equal weight to each problem, regardless of the number of nodes it required expanding). We also report the average of domain averages (that is, giving the same weight to each domain, regardless of how many problems it contains).

Empirical Results

We now describe the results of our empirical evaluation, starting with comparing the baselines to our approach using the OD regime. OD is the closest to these in terms of assumptions – it does not assume any knowledge about the problem it is used on. First, Figure 2 shows the accuracy using different values of K under the OD regime for different planner configurations. Unsurprisingly, as K increases so does accuracy, and so we present the results for $K = 40$

Domain	VaSP	VeSP	DBP	RF	OD
Arprt(5)	31.3(12.3)	23.0(3.3)	**21.1(3.4)**	54.2(1.0)	29.2(0.2)
Blck(13)	**27.0(3.6)**	31.8(8.6)	53.4(1.9)	29.1(11.3)	27.5(1.7)
Depot(3)	28.9(2.3)	33.1(6.2)	45.0(3.6)	29.4(10.2)	**24.4(0.7)**
Elev(11)	26.4(3.9)	29.1(5.6)	40.4(5.8)	**25.4(5.4)**	26.0(3.3)
Frcll(1)	29.1(0.0)	**23.7(0.0)**	39.8(0.0)	24.9(0.0)	26.5(0.0)
Gripper(4)	34.3(3.2)	**21.7(1.4)**	25.6(3.6)	35.1(15.9)	26.9(3.6)
Logistics(5)	39.4(3.6)	28.7(2.8)	35.3(8.0)	41.9(6.9)	**25.7(3.3)**
OS(5)	**24.4(2.0)**	40.3(3.6)	44.1(4.4)	32.6(8.4)	27.6(3.4)
Parcp(6)	40.1(5.6)	42.2(2.8)	48.3(2.1)	36.6(13.3)	**27.6(3.8)**
Pegsol(15)	34.7(2.7)	33.5(4.6)	48.4(2.3)	28.7(6.2)	**27.5(3.9)**
PSR(15)	38.8(5.6)	31.3(3.2)	48.3(2.1)	27.2(10.2)	**24.9(4.3)**
Rover(3)	28.6(1.8)	38.3(1.9)	46.7(1.6)	**22.2(4.7)**	26.8(1.7)
Sat(2)	29.1(3.4)	27.1(4.4)	41.3(13.6)	42.3(18.9)	**24.9(1.3)**
Scnlzr(3)	29.1(3.7)	29.8(3.8)	34.0(7.1)	**22.2(12.2)**	26.7(0.2)
Soko(16)	35.1(8.3)	32.2(7.8)	45.7(7.8)	34.0(6.2)	**27.2(2.7)**
TPP(1)	34.9(0.0)	47.7(0.0)	49.7(0.0)	**20.4(0.0)**	27.2(0.0)
Trns(5)	26.5(1.8)	27.7(2.8)	40.4(5.9)	36.8(7.6)	**25.6(3.4)**
WW(5)	54.6(3.5)	46.7(3.8)	51.6(7.6)	**23.6(2.7)**	27.3(3.9)
Zenotravel(3)	28.7(4.6)	32.0(8.0)	36.0(8.3)	29.6(16.5)	**24.9(14.1)**
Avgd(19)	33.4(7.1)	33.4(7.5)	42.7(8.6)	32.4(8.5)	**26.6(1.3)**
Avgp(121)	34.0(8.2)	32.8(7.5)	44.8(9.2)	32.0(10.9)	**26.7(3.2)**

A* with Lmcut

Table 3: RMSE (in percent) and standard deviation in parentheses, using the OD regime and A* with Lmcut. Results in bold are the best performance in this row. See caption of Table 1 for details

Domain	VaSP	VeSP	PBP	RF	OD
Arprt(8)	18.8(4.6)	**16.1(6.4)**	25.4(0.8)	34.3(14.3)	25.8(5.3)
Blck(7)	36.2(0.6)	32.9(3.1)	28.7(0.1)	32.4(7.9)	**26.3(1.5)**
Depot(1)	27.4(0.0)	**19.9(0.0)**	26.3(0.0)	22.4(0.0)	21.3(0.0)
Elev(5)	31.8(1.4)	**26.0(3.9)**	27.9(0.4)	29.2(15.3)	28.0(0.8)
Frcll(2)	34.7(4.2)	31.5(12.8)	27.6(0.8)	34.(7.6)	**27.1(6.7)**
Gripper(4)	33.4(1.1)	55.2(2.8)	27.7(0.5)	**22.4(8.3)**	23.3(5.5)
Logistics(9)	31.3(3.5)	28.1(2.2)	27.4(0.5)	**22.3(7.4)**	25.2(4.6)
Miconic(4)	35.4(2.3)	24.3(2.3)	27.8(0.4)	**23.8(8.5)**	24.1(2.1)
OS(5)	31.9(2.7)	**18.5(1.0)**	27.6(0.3)	20.8(6.7)	24.8(5.3)
Parcp(5)	**20.2(2.1)**	24.1(2.1)	27.2(0.3)	30.3(4.7)	24.7(3.9)
Pegsol(15)	42.1(0.7)	34.1(10.4)	28.4(0.3)	29.2(6.2)	**25.5(2.6)**
PSR(16)	37.5(5.9)	49.2(14.0)	28.5(0.3)	**26.6(11.2)**	27.7(2.3)
Sat(2)	30.3(3.8)	30.6(4.1)	28.2(0.5)	41.7(4.3)	**26.1(3.2)**
Scnlzr(4)	35.9(1.9)	29.5(5.4)	28.4(0.6)	46.2(7.4)	**28.2(1.4)**
Soko(16)	39.6(5.3)	30.4(6.2)	**27.8(1.3)**	37.7(9.4)	28.0(3.3)
TPP(1)	31.6(0.0)	23.7(0.0)	26.9(0.0)	**11.5(0.0)**	23.6(0.0)
Trns(6)	30.2(1.4)	**25.9(7.0)**	27.8(0.4)	28.9(4.10)	26.1(4.4)
WW(5)	36.3(0.6)	30.8(2.0)	28.5(0.2)	**26.9(6.6)**	27.6(2.1)
Zenotravel(4)	33.2(3.3)	**23.9(4.5)**	27.8(0.4)	30.5(10.9)	27.3(1.5)
Avgd(19)	33.0(5.8)	30.6(9.4)	27.7(0.8)	30.0(8.0)	**25.9(1.9)**
Avgp(119)	34.8(7.3)	33.0(12.0)	27.8(1.0)	30.5(10.5)	**26.4(3.5)**

A* with h_{FF}

Table 4: RMSE (in percent) and standard deviation in parentheses, using the OD regime and A* with h_{FF}. Results in bold are the best performance in this row. See caption of Table 1 for details
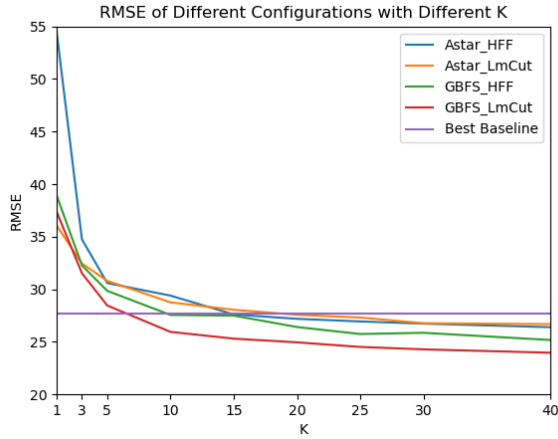
Figure 2: RMSE with Different K and Different Configurations, Best Baseline is the best RMSE from the other methods and from all the different configurations

	A*/Lmcut	A*/h_{FF}	GBFS/Lmcut	GBFS/h_{FF}
S	121	119	89	204
B	9/19	5/18	11/15	15/20
M	0.82	0.94	0.71	0.73

Table 5: Experiment 1 summary on all configurations. S=Solved problems,B=Best accuracy, M=RMSE ours/best predictor.

in the detailed tables in the remainder of this section. However, note that we outperform the previous techniques even with $K = 20$. Tables 1, 2, 3, and 4, compare our technique using the OD regime to the estimators proposed in previous work and to using the random forest classifier in all 4 configurations of heuristic and search algorithm. In each subtable, we display only the dominant algorithms, and omit those that did not get the best results in any domain. Note that some domains do not appear for some configuration – namely, those where no problems were solved.

To summarize these results more concisely, Table 5 shows in how many domains OD was the best predictor, and the overall accuracy of our OD predictor vs. the best overall predictor from the previous work. Overall, OD was the most accurate, with better RMSE compared to the best predictor. These results hold whether we use the average over RMSE in each problem (Avgp), or the average over average RMSE in each domain (Avgd). Although OD got better results than the other methods, we want to understand where it works better and where it does not. Looking more closely at the results, we observe that larger problems had higher error. Figure 3 shows a scatter plot for each planner configuration. Each point corresponds to a solved problem, with the number of expanded nodes on the x-axis, and the RMSE of OD on the y-axis. We also add a linear regression line, showing a clear correlation. additionally we checked statistical significance using Kendall's tau (Kendall 1938). The correlation we found is $tau = 0.29, 0.23, 0.49, 0.36$ for A* with
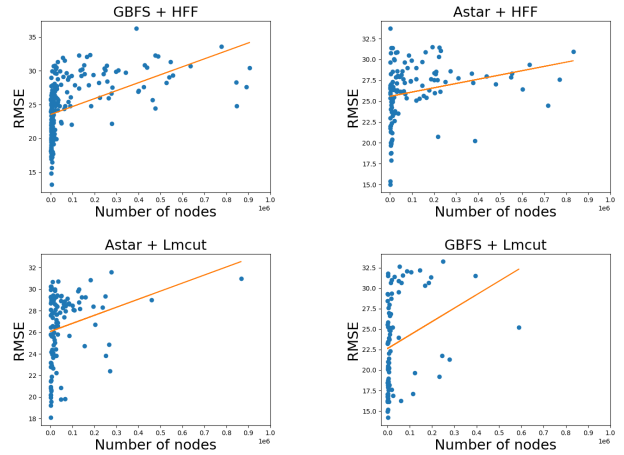

Figure 3: Correlation between number of expanded nodes and RMSE using OD in different configurations

h_{FF}, A* with Lmcut, GBFS with h_{FF} and GBFS with Lmcut respectively. In all cases, the p-value was less than 0.001, showing a statistically significant correlation between number of nodes and RMSE. We believe this is due to the fact that heuristics tends to be less accurate on the bigger (more difficult) problems, and since our search progress estimate relies on the heuristic, it is less accurate as well.

Having seen that our technique, in the OD regime, outperforms previous work, we now turn to evaluating the impact of more specific training data by comparing the OD regime to the SD and ODTS regimes. Recall that in the SD regime our network is trained only on instances from the same domain, while in ODTS it is trained on instances from all other domains, and then fine-tuned on the domain being evaluated. Also recall that these regimes can only be used on domains which have enough data (15 solved instances for SD, 6 for ODTS), and thus the comparison is on fewer domains.

Table 6 shows the results for this evaluation. In the table there are missing domains since in some configurations the number of problems the planner solved was below the required number we define for this experiment. Interestingly, ODTS outperforms SD in most cases, showing that the training data from other domains is important. ODTS also outperforms OD in most domains, as well as overall in each table, showing that domain-specific tuning does help. In some domains where OD outperforms ODTS, we observe that the split between the even and odd problems was not really balanced in terms of problem difficulty among the folds, and in some cases the number of expanded nodes in one fold was double that in the other. This could explain the poor behavior of ODTS.

Generalization

We conclude our empirical evaluation by evaluating how our predictor handles changing the search algorithm, the heuristic, or both. In this experiment, we trained our predictor (using the OD regime) on one choice of search algorithm and heuristic, and then tested its accuracy on another choice.

	Domain	A*		
		OD	SD	ODTS
Lmcut	Blck	27.5(9.3)	-	**27.3(9.8)**
	Elev	**26.0(12.8)**	-	**26.0(13.5)**
	Parcp	27.6(13.7)	-	**27.3(15.2)**
	Pegsol	27.5(14.0)	**26.9(15.1)**	27.0(21.4)
	PSR	24.9(14.7)	21.9(13.2)	**21.3(12.3)**
	Soko	27.2(12.0)	25.8(14.4)	**25.1(13.4)**
	Avgd(3)	26.6(8.1)	**25.0(11.4)**	25.8(10.6)
	Avgp(46)	26.7(12.8)	25.8(15.0)	**25.5(16.3)**
h_{FF}	Arprt	25.8(5.3)	-	**22.8(5.6)**
	Blck	26.3(1.5)	-	**24.2(2.6)**
	Logistics	25.2(4.6)	-	**25.4(5.2)**
	Pegsol	25.5(2.6)	24.9(2.7)	**24.0(2.1)**
	PSR	27.7(2.3)	**23.1(5.6)**	23.6(5.6)
	Soko	28.0(3.3)	27.5(3.3)	**24.9(4.1)**
	Trns	**26.1(4.4)**	-	27.1(4.3)
	Avgd(3)	25.9(1.9)	25.2(9.3)	**24.6(12.0)**
	Avgp(47)	26.4(3.5)	27.5(4.5)	**24.5(4.4)**
	Domain	GBFS		
		OD	SD	ODTS
Lmcut	Blck	24.2(4.6)	-	**23.9(5.6)**
	OS	21.0(3.5)	15.4(3.2)	**13.4(2.3)**
	PSR	23.4(4.8)	-	**23.0(6.5)**
	Soko	29.4(5.0)	**29.1(0.3)**	29.1(2.3)
	TPP	**19.9(4.1)**	-	24.0(9.6)
	Trns	**21.9(4.5)**	-	25.0(6.1)
	Avgd(2)	25.2(3.7)	22.2(9.9)	**21.3(5.2)**
	Avgp(35)	24.0(5.5)	24.0(2.7)	**23.8(2.7))**
h_{FF}	Arprt	25.1(5.3)	-	**22.7(5.0)**
	Blck	21.8(3.3)	**17.4(5.1)**	17.5(5.6)
	Depot	26.3(4.1)	-	**24.7(6.3)**
	Elev	27.3(3.1)	**26.3(4.4)**	26.5(5.0)
	Frcll	20.2(4.0)	-	**19.2(5.0)**
	Logistics	26.7(3.9)	-	**24.8(3.1)**
	OS	23.8(3.0)	-	**19.1(3.3)**
	Parcp	23.6(5.9)	-	**21.7(6.8)**
	Pegsol	**26.5(3.8)**	26.6(4.1)	26.8(4.9)
	PSR	**25.3(4.2)**	-	26.8(4.3)
	Soko	**28.5(4.7)**	29.0(1.4)	29.6(4.6)
	Trns	**25.1(4.6)**	-	25.7(6.8)
	WW	**23.2(4.8)**	-	23.8(6.9)
	Avgd(4)	24.8(2.2)	25.2(5.3)	**24.0(3.7)**
	Avgp(85)	25.2(4.5)	25.3(6.0)	**24.5(10.2)**

Table 6: RMSE (in percent) and standard deviation in parentheses with A*/GBFS and Lmcut/h_{FF} using all 3 regimes where Avgp is average RMSE over each problem and Avgd is the average over average RMSE in each domain. Results in bold are the best performance in this row. See caption of Table 1 for details

Test/Train	G/L	A/L	G/H	A/H
G/L	**24.0(5.4)**	28.3(3.5)	28.3(4.4)	28.2(4.2)
A/L	**26.1(4.2)**	26.7(3.2)	27.2(3.5)	27.6(3.3)
G/H	25.7(5.8)	29.3(6.0)	**25.2(4.5)**	26.8(3.5)
A/H	27.1(7.3)	29.2(5.5)	26.8(4.7)	**26.4(3.5)**

Table 7: RMSE (in percent) and standard deviation in parentheses on generalization evaluation. results in bold are the best performance in this row. A=A*, G=GBFS, H=h_{FF} and L=Lmcut

the polar opposite (different search algorithm and heuristic) still yields better accuracy than the best predictor from previous work. This shows that our predictor can generalize to a different search algorithm and heuristic without suffering too severe a decrease in performance.

Discussion and Future Work

In this paper we have described a novel approach to search progress estimation using deep learning. We have shown that our approach outperforms previous state-of-the-art approaches, and that it benefits from having access to better training data. It is also interesting to note that the previous search progress estimators (Thayer, Stern, and Lelis 2012), which were developed manually using extensive expertise in heuristic search and human creativity, are outperformed by a machine learning algorithm with very limited prior knowledge.

Besides being an interesting problem, search progress estimation can be used to make decisions about search. For example, as previously mentioned, estimates of remaining search time have been used in situated temporal planning (Cashmore et al. 2018) and in Branch-and-Bound search (Anderson et al. 2019). Other applications are in anytime search (Dionne, Thayer, and Ruml 2011) or metareasoning (Shperberg et al. 2019, 2020, 2021). Specifically, the situated temporal planner (Shperberg et al. 2021) uses VaSP, which is outperformed by our technique. Thus, we expect replacing the search progress estimator there will improve the performance of the situated temporal planner.

However, using search progress estimates to make decisions about search has a problem with self-reference – the predictions affect search decisions, which affect search time, which then affect the true search progress. We intend to address this challenge in future work by attempting to learn a series of predictors which converge to a fixed point which exhibits good performance. Finally, we remark that we employed fairly simple machine learning tools here. In future work we will examine whether we can improve performance even more by using different architectures of neural networks, or more informative features.

Acknowledgments

With 2 search algorithms and 2 heuristics, we have 4 combinations, and thus 16 settings to evaluate.

Table 7 shows the results of this cross domain evaluation. As expected, using the same configuration for training and in test leads to the best performance in almost all cases. The more interesting phenomenon is that when we change the algorithm and keep the heuristic the average RMSE increases by 1.43. When we change the heuristic and keep the algorithm, RMSE increases by 2.13, and changing both increases RMSE by 2.4. Note that, for any configuration, training on

References

Alkhazraji, Y.; Frorath, M.; Grützner, M.; Helmert, M.; Liebetraut, T.; Mattmüller, R.; Ortlieb, M.; Seipp, J.; Springenberg, T.; Stahl, P.; and Wülfing, J. 2020. Pyperplan. https://doi.org/10.5281/zenodo.3700819.

Anderson, D.; Hendel, G.; Le Bodic, P.; and Viernickel, M. 2019. Clairvoyant restarts in branch-and-bound search using online tree-size estimation. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence and Thirty-First Innovative Applications of Artificial Intelligence Conference and Ninth AAAI Symposium on Educational Advances in Artificial Intelligence*, 1427–1434.

Belov, G.; Esler, S.; Fernando, D.; Le Bodic, P.; and Nemhauser, G. L. 2017. Estimating the size of search trees by sampling with domain knowledge. In *IJCAI*, 473–479.

Bonet, B.; and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence*, 129(1-2): 5–33.

Breiman, L. 2001. Random forests. *Machine learning*, 45(1): 5–32.

Breyer, T.; and Korf, R. 2008. Recent results in analyzing the performance of heuristic search. In *Proceedings of the First International Workshop on Search in Artificial Intelligence and Robotics (held in conjunction with AAAI)*, 24–31.

Cashmore, M.; Coles, A.; Cserna, B.; Karpas, E.; Magazzeni, D.; and Ruml, W. 2018. Temporal Planning while the Clock Ticks. In *ICAPS*, 39–46.

Dionne, A. J.; Thayer, J. T.; and Ruml, W. 2011. Deadline-Aware Search Using On-Line Measures of Behavior. In *SOCS*, 39–46.

Doran, J. E.; and Michie, D. 1966. Experiments with the graph traverser program. *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences*, 294(1437): 235–259.

Gers, F. A.; Schmidhuber, J. A.; and Cummins, F. A. 2000. Learning to Forget: Continual Prediction with LSTM. *Neural Computation*, 12(10): 2451–2471.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Trans. Syst. Sci. Cybern.*, 4(2): 100–107.

Helmert, M.; and Domshlak, C. 2009. Landmarks, critical paths and abstractions: what's the difference anyway? In *Proceedings of the Nineteenth International Conference on International Conference on Automated Planning and Scheduling*, 162–169.

Hochreiter, S.; and Schmidhuber, J. 1997. Long short-term memory. *Neural computation*, 9(8): 1735–1780.

Hoffmann, J. 2011. Analyzing Search Topology Without Running Any Search: On the Connection Between Causal Graphs and h+. *J. Artif. Intell. Res.*, 41: 155–229.

Hoffmann, J.; and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14: 253–302.

Hutter, F.; Xu, L.; Hoos, H. H.; and Leyton-Brown, K. 2014. Algorithm runtime prediction: Methods & evaluation. *Artificial Intelligence*, 206: 79–111.

Kendall, M. G. 1938. A new measure of rank correlation. *Biometrika*, 30(1/2): 81–93.

Kingma, D. P.; and Ba, J. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Lelis, L. H.; Stern, R.; Felner, A.; Zilles, S.; and Holte, R. C. 2014. Predicting optimal solution cost with conditional probabilities. *Annals of Mathematics and Artificial Intelligence*, 72(3): 267–295.

Lelis, L. H.; Stern, R.; and Sturtevant, N. R. 2014. Estimating search tree size with duplicate detection. In *Seventh Annual Symposium on Combinatorial Search*.

Medsker, L. R.; and Jain, L. 2001. Recurrent neural networks. *Design and Applications*, 5.

Shperberg, S. S.; Coles, A.; Cserna, B.; Karpas, E.; Ruml, W.; and Shimony, S. E. 2019. Allocating Planning Effort When Actions Expire. In *AAAI*, 2371–2378. AAAI Press.

Shperberg, S. S.; Coles, A.; Karpas, E.; Ruml, W.; and Shimony, S. E. 2021. Situated Temporal Planning Using Deadline-aware Metareasoning. In *ICAPS*, 340–348. AAAI Press.

Shperberg, S. S.; Coles, A.; Karpas, E.; Shimony, S. E.; and Ruml, W. 2020. Trading Plan Cost for Timeliness in Situated Temporal Planning. In Bessiere, C., ed., *IJCAI*, 4176–4182. ijcai.org.

Sutskever, I.; Vinyals, O.; and Le, Q. V. 2014. Sequence to sequence learning with neural networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems-Volume 2*, 3104–3112.

Tange, O.; et al. 2011. Gnu parallel-the command-line power tool. *The USENIX Magazine*, 36(1): 42–47.

Thayer, J. T.; Stern, R.; and Lelis, L. H. 2012. Are We There Yet?-Estimating Search Progress. In *SOCS*.

Yu, Y.; Si, X.; Hu, C.; and Zhang, J. 2019. A review of recurrent neural networks: LSTM cells and network architectures. *Neural computation*, 31(7): 1235–1270.