1 Верификация устойчивости времени доставки по отношению к эмбеддингам узлов для всей сети

1.1 Анализ графа сети

Для того, чтобы исследовать влияние небольших изменений эмбеддингов на время доставки во всей конвейерной сети, мы можем запустить описанный ранее алгоритм верификации для каждой пары узлов графа сети [1]. Так как контроллер учитывает информацию не только о соседних узлах, но и о конечном узле, такие пары являются упорядоченными, а их количество равняется числу размещений $A_n^2 = n(n-1)$, где n — порядок графа.

Мы считаем, что отклонители хорошо обучены, а сумка обычно движется по кратчайшему пути. Так как мы имеем ограничение на изменение эмбеддингов, их изменение на кратчайшем пути будет в большей степени отражаться на времени доставки. Рассмотрим пример графа сети, представленный на рисунке 1.

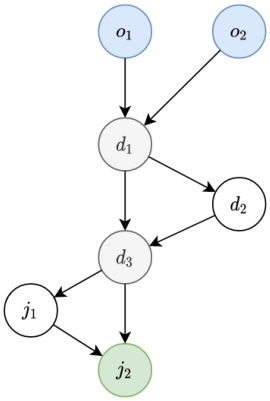

Рисунок 1 — Пример графа конвейерной сети

Задача расчета ожидаемого времени доставки параметризуется множеством нетривиальных отклонителей, лежащих на возможных путях сумки, а также конечным узлом. Для доставки из узла o_1 в узел j_2 мы можем формализовать ее следующим образом:

$$\{d_1, d_3\}_{j_2} \to t_{o_1 \to j_2}.$$

Теперь рассмотрим задачу между парой, кратчайший путь которой является суффиксом исходного кратчайшего пути:

$$\{d_3\}_{j_2} \to t_{d_1 \to j_2}.$$

Можно обратить внимание на то, что она является подзадачей исходной, а все связанные эмбеддинги могли быть уже исследованы в более полной задаче. Поэтому чтобы минимизировать количество запусков верификации, мы будем искать пару, имеющую самый длинный кратчайший путь, который не является суффиксом у ранее рассмотренных пар.

Также нам стоит учесть случаи, когда кратчайший путь между парой не является суффиксом, но задача расчета ожидаемого времени доставки в точности повторяется. Например, это можно наблюдать у пар (o_1, j_2) и (o_2, j_2).

Заметим, что не все пары вершин могут иметь нетривиальные отклонители на своем кратчайшем пути. Такие пары мы также можем не рассматривать, так как эмбеддинги узлов никак не влияют на время доставки.

1.2 Описание алгоритма

Псевдокод алгоритма минимизации количества пар для верификации приведен листинге 1.

Листинг 1 — Поиск множества пар

```
 1: function GetPairs(g)
 2:     V ← GetNodes(g)
 3:                          ▷ пара состоит из начала и конца кратчайшего пути
 4:     for (o, i) ∈ V² do
 5:         path ← FindShortestPath(g, o, i)
 6:         if ∃path ∧ d ∈ path then
 7:             paths ← path
 8:         end if
 9:     end for
10:
11:     Sort(paths)                              ▷ по возрастанию кол-ва узлов
12:     while paths ≠ ∅ do
13:         path ← Pop(paths)
14:         RemoveSuffixes(path, paths)
15:         (o, i) ← path
16:         if GetDiverters(o, i) ∉ GetDiverters(res, i) then
17:             pairs ← (o, i)
18:         end if
19:     end while
20:
21:     return pairs
22: end function
```

1. Для каждой пары узлов графа конвейерной сети находим кратчайший путь (5). Если этот путь существует и содержит хотя бы один отклонитель, то сохраняем его (6–7).

2. Извлекаем самый длинный путь и удаляем все его суффиксы из множества найденных путей (10–13). Если множество отклонителей данной пары не совпадает с каким-либо множеством отклонителей сохраненых пар, которые имеют тот же сток, то сохраняем эту пару (16–17).

Псевдокод алгоритма верификации устойчивости времени доставки для всей сети приведен в листинге 2

Листинг 2 — Верификация устойчивости времени доставки для всей сети

```
1: procedure VerifEdtRobustnessWrtEmb(g)
2:     for (o, i) ∈ GetPairs(g) do
3:                         ▷ конечный узел не должен иметь исходящих ребер
4:         m ← CreateAbsorbingMC(g, i)
5:         τ ← CreateEDTSolver(m, o)
6:         if GetNontrivDvtrs(τ) = ∅ then
7:             continue
8:         end if
9:
10:        t_⊤ ← τ()
11:        for k ∈ (0.95, 0.99, 1.01, 1.05) do
12:            t_b ← t_⊤ · k
13:            VerifEdtBoundWrtEmb(t_b)
14:        end for
15:    end for
16: end procedure
```

1. Для каждой необходимой пары по полному графу сети создаем поглощающую марковскую цепь с одним поглощающим состоянием (2–4).

2. Определяем функцию расчета ожидаемого времени доставки при помощи марковской цепи (5). Если нетривиальные отклонители отсутствуют, то отбрасываем текущую пару (6–7).

3. Рассчитываем ожидаемое время доставки и используем его в качестве границы для верификации (10–13).

1.3 Результаты выполнения

Результаты выполнения алгоритма для $\epsilon = 0.1$ и первого примера конвейерной сети приведены в таблице 1.

Таблица 1 — Результаты верификации устойчивости времени доставки для всей сети

	$t_\top$	t_b	Вериф.	Прод., с
$o_1 \to d_7$	40.10	38.10	−	0.09
		39.70	−	0.19
		40.50	+	0.78
		42.11	+	0.52
$o_1 \to d_8$	50.10	47.60	−	0.09
		49.60	−	0.11
		50.60	+	0.74
		52.61	+	0.34

$o_1 \to i_0$	50.15	47.64	−	0.09
		49.65	−	0.11
		50.65	+	0.77
		52.66	+	0.53
$o_0 \to i_0$	40.20	38.19	−	0.10
		39.80	−	0.12
		40.60	+	0.94
		42.21	+	0.66
$o_0 \to i_1$	53.10	50.45	−	0.11
		52.57	−	0.11
		53.63	+	0.76
		55.76	+	0.41
$o_1 \to i_1$	43.25	41.09	−	0.15
		42.82	−	0.17
		43.68	+	2.85
		45.41	+	1.80
$o_1 \to i_2$	53.06	50.41	−	0.12
		52.53	−	0.16
		53.59	+	4.54
		55.71	+	1.64
$o_0 \to i_2$	53.07	50.42	−	0.09
		52.54	−	0.11
		53.60	+	0.63
		55.72	+	0.28
$o_1 \to i_3$	43.10	40.95	−	0.14
		42.67	−	0.17
		43.53	+	2.27
		45.26	+	1.45
$o_0 \to i_3$	43.15	40.99	−	0.10
		42.72	−	0.11
		43.58	+	0.91
		45.31	+	0.54
$d_3 \to i_3$	43.10	40.95	−	0.10
		42.67	−	0.12
		43.53	+	0.82
		45.26	+	0.56
$o_1 \to j_0$	30.10	28.60	−	0.09
		29.80	−	0.11
		30.40	+	0.89
		31.61	+	0.52

2 Поиск состязательного примера

2.1 Формализация задачи

Требуется найти изменение с заданным ограничением в эмбеддингах, при котором ожидаемое время доставки будет максимальным:

$$\underset{\|\boldsymbol{\Delta e}\|_\infty \leq \epsilon}{\arg\max}\, \tau_{\boldsymbol{e}}(\boldsymbol{\Delta e}),$$

где $\boldsymbol{e}$ — конкатенация эмбеддингов узлов, участвующих в расчете Q-значения.

Найденное худшее время доставки является оценкой нижней грани множества значений, которые могут быть верифицированы. Приведенный алгоритм использует итеративный метод быстрого знака градиента [2].

2.2 Описание алгоритма

В листинге 3 приведен псевдокод алгоритма.

Листинг 3 — Поиск оптимального возмущения эмбеддингов

```
1: function PERTURB(e, α)
2:     for i ∈ I do
3:         if i ≠ I[−1] then
4:             Δe^(0) ← direc ∈ U_m(−ϵ, ϵ) · scale ∈ U(0, 1)
5:         else
6:             Δe^(0) ← 0^m
7:         end if
8:
9:         for j ∈ J do
10:            grad = ∇τ_e(Δe)
11:
12:            if ‖grad‖_1 = 0 then
13:                break
14:            end if
15:
16:            step = sgn grad · ϵ · α
17:            Δe^(j) = Δe^(j−1) + step
18:
19:            if ‖Δe^(j)‖_∞ > ϵ then
20:                Δe^(j) ← {v : v = clip_ϵ Δe_i}
21:            end if
22:        end for
23:
24:        t ← τ_e(Δe)
25:        if t > t_b then
26:            t_b ← t
27:            Δe_b ← Δe
28:        end if
29:    end for
30:
31:    return Δe_b
32: end function
```

1. Выполняем несколько итераций поиска возмущения (2). В начале каждой итерации определяем случайный начальный вектор возмущения из равномерного распределения, но применяем эвристику и на последней итерации оставляем вектор нулевым (3–6). Если мы просто возьмем вектор из совместного распределения, то при его большой размерности норма будет близка к своей границе: $\lim_{|\boldsymbol{e}|\to\infty} \|\boldsymbol{e}\|_\infty = \epsilon$. Чтобы это избежать, умножим вектор на случайный масштабный коэффициент.

2. Используем несколько шагов проективного градиентного спуска для улучшения начального возмущения (9). Отметим, что для удобства мы можем определить градиент следующим образом: $\nabla_{\boldsymbol{\Delta e}} \tau(\boldsymbol{e}+\boldsymbol{\Delta e}) = \nabla\tau(\boldsymbol{e}+\boldsymbol{\Delta e})\cdot 1$ (10). Если градиент получился нулевым, то прекращаем спуск (12–13).

3. Преобразовываем градиент, вычисляя его знак и масштабируя (16).

Здесь α является подбираемым параметром, определяющим скорость спуска.

4. Прибавляем преобразованный градиент к аргументу целевой функции (17). Мы прибавляем градиент, так как наша задача максимизировать время доставки.

5. Проверяем ограничение на норму вектора (19). Если оно нарушено, то выполняем проекцию вектора на границу нормы (20). Схематично эта операция изображена на рисунке 2.

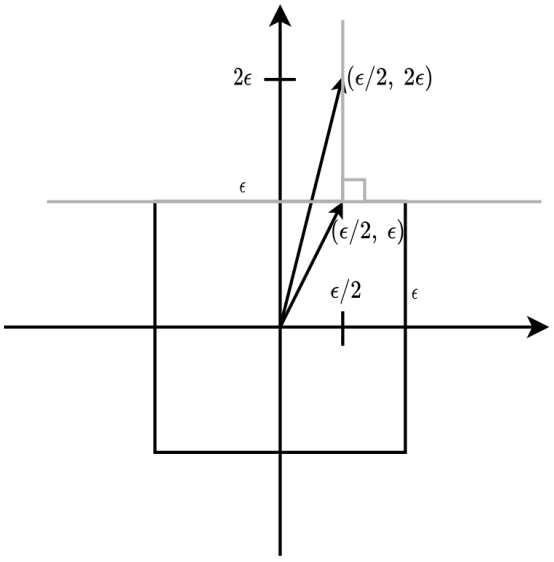

Рисунок 2 — Пример проекции вектора

Ее можно определить при помощи функции, которая выполняет коррекцию каждой компоненты:

$$\mathrm{clip}_\epsilon\, x = \begin{cases} \epsilon & \text{if } x > \epsilon \\ -\epsilon & \text{if } x < -\epsilon \\ x & \text{else} \end{cases} .$$

6. После выполнения спуска мы сохраняем полученное возмущение, если оно дает наибольшее время доставки (24–27).

2.3 Результаты выполнения

Данные в таблице 3 описывают процесс выполнения последнего градиентного спуска для пары вершин (o_1, i_3) при $\epsilon = 0.4$, $\alpha = 0.02$.

Таблица 2 — Результаты градиентного спуска

Шаг	$t_\top$	$\|\boldsymbol{\Delta e}\|_\infty$	$\|\boldsymbol{e}+\boldsymbol{\Delta e}\|_\infty$	p_2	p_3
1	43.100559	0.08590	5.20921	0.0050	0.9950
5	43.100675	0.12055	5.16921	0.0050	0.9950
10	43.101031	0.16055	5.12921	0.0050	0.9950
15	43.102261	0.20055	5.08921	0.0051	0.9950
20	43.106360	0.24055	5.04921	0.0053	0.9950
25	43.120090	0.28055	5.00921	0.0060	0.9950
30	43.168993	0.32055	5.00570	0.0084	0.9950
35	43.349849	0.36055	5.04570	0.0174	0.9949
40	44.038848	0.40000	5.08570	0.0517	0.9949
45	45.812226	0.40000	5.12570	0.1399	0.9948
50	48.289755	0.40000	5.16570	0.2631	0.9947
55	50.217126	0.40000	5.20570	0.3589	0.9946
60	50.849359	0.40000	5.22040	0.3904	0.9946
65	50.865898	0.40000	5.22040	0.3912	0.9946
70	50.882441	0.40000	5.22040	0.3920	0.9946
75	50.898993	0.40000	5.22040	0.3928	0.9946
80	50.915559	0.40000	5.22040	0.3936	0.9946
85	50.932139	0.40000	5.22040	0.3945	0.9946
90	50.948734	0.40000	5.22040	0.3953	0.9945
95	50.957471	0.40000	5.22040	0.3957	0.9945
100	50.963990	0.40000	5.22040	0.3960	0.9945

Можно заметить, что алгоритм стремится изменять эмбеддинги таким образом, чтобы отклонитель d_2 (соответствует вероятность p_2), лежащий на кратчайшем пути, стал чаще принимать неверные решения.

3 Верификация устойчивости времени доставки в динамической сети

Допустим, что нагрузка на конвейер не постоянна, а меняется в течение времени. Тогда вес ребра в графе сети будем определять не константой, а некоторым распределением. Для того, чтобы выполнить верификацию, возьмем момент времени, когда нагрузка на сеть максимальна. Преобразование графа сети выполним, как показано на рисунке 3.

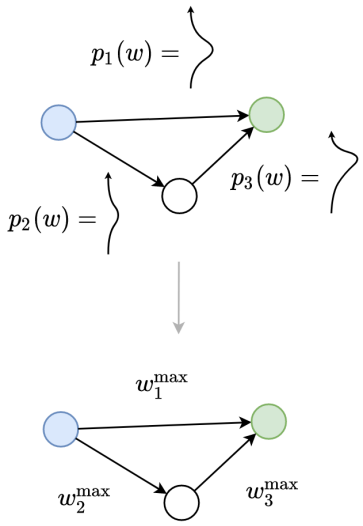

Рисунок 3 — Преобразование динамического графа

Листинг 4 — Верификация в динамическом графе

```
1:  function VerifDynNetwork(e)
2:      E ← GetEdges(g)
3:      for edge ∈ E do
4:          w ← GetWeight(edge)
5:          c ∈ X                          ▷ X имеет распределение Парето
6:          SetWeight(edge, w · c)
7:      end for
8:
9:      Relearn(e)
10:
11:     return VerifEdtBoundWrtEmb()
12: end function
```

1. Вес каждого ребра графа сети доможнаем на случайное число из распределения Парето (2–6).
2. Переобучаем эмбеддинги узлов на новой топологии графа (9).
3. Выполняем верификацию границы времени доставки (11).

Таблица 3 — Результаты верификации границы времени доставки в динамической сети

ϵ	$t_b = 44.00$	$t_b = 43.50$	$t_b = 43.12$	$t_b = 43.10$	$t_b = 43.00$
0.00	+ ()	+ ()	+ ()	+ ()	+ ()
0.01	+ ()	+ ()	+ ()	+ ()	+ ()
0.10	+ ()	+ ()	+ ()	+ ()	+ ()
0.20	+ ()	+ ()	+ ()	+ ()	+ ()
0.40	+ ()	+ ()	+ ()	+ ()	+ ()
0.80	+ ()	+ ()	+ ()	+ ()	+ ()

Список литературы

[1] Мультиагентные алгоритмы маршрутизации на основе глубоких нейронных сетей с подкреплением и их верификация. — Санкт-Петербург. — 2020. — 108 с.

[2] Alex Kurakin, Ian Goodfellow, and Samy Bengio. Adversarial examples in the physical world : technical report. // arXiv. — 2016. — URL: https://arxiv.org/abs/1607.02533