AppGym: Бейсово Дълбоко Самообучение за Автоматично Тестване на Софтуер

Firstname Surname

A thesis presented for the degree of
Doctor of Philosophy

Supervised by:
Professor Louis Fage
Captain J. Y. Cousteau

University College London, UK
January 2015

I, AUTHORNAME confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the thesis.

Абстракт

Благодарности

Съдържание

Списък на фигурите

List of tables

Речник

API	**A**pplication **P**rogramming **I**nterface
JSON	**J**ava**S**cript **O**bject **N**otation

Глава 1

Увод

Съществуващите методи не дават възможност за споделяне на наученото от друго приложение, намиране на аномалии във функционалността, бързо научаване на промени (премахване на старо знание) и оценка на несигурността при изпълнение на действие.

1.1 Структура на дисертационния труд

Глава 2 дава познания върху Дълбокото подсилено обучение и Бейсовото моделиране. **Глава 3** поставя целите и задачите на текущата работа. В **Глава 4** се създава среда за тестване на Android мобилни приложения. **Глава 5** представя модел за генериране на входни данни за тестови случаи. В **Глава 6** се представя модел за намиране на аномалии в тестови случай по подадено изображение. Цялостната система е представена в **Глава 7** заедно с емпирични сравнения спрямо други решения. Нерешени проблеми, бъдещи подобрения и дискусия се намират в последната глава.

Глава 2

Литературен обзор

2.1 Подсилено обучение

Един от основните проблеми в сферата на изкуствения интелект е взимане на поредица от решения в стохастична система. Агент, който изучава приложение, е пример за такава среда. Този проблем се състои в избора на редица от решения, които да максимизират разгледаните състояния на текущото приложение. Това е по-сложно от задачи, в които трябва да се направи само едно решение. Оценката за представянето на агента може да се даде само след много извършени стъпки. Това означава, че той може да избере неправилно действие сега и да разбере за това много по-късно, т.е. имаме *забавяне на последствията*. Допълнително, не може да наблюдаваме точното състояние на агента, поради липсата на точен модел на приложението, което тества.

Марковски вериги за вземане на решения (MDP) Моделират системи, които искаме да контролираме. Във всяка времева стъпка t, системата се намира в дадено състояние s. Например, описаният агент може да се намира на даден екран от приложението, след като е натиснал определен бутон. Системата преминава през различни състояния като резултат от действията, които сме избрали. Задачата ни е да избираме действия, които са добри и да минимизираме броя на тези, които не са. Разнообразни проблеми са моделирани чрез MDP формализма. Някои примери са системи за препоръки (Joachims et al. 1997), рутиране на мрежи (Boyan et al. 1994), управление на асансьори (Crites & Barto 1996), навигация на

роботи (Sutton & Barto 1998).

Подсиленото обучение (RL) (Sutton & Barto 1998) дава способи за решаване на проблеми, дефинирани чрез MDP формализма. RL агент взаимодейства със среда за определено време. На всяка времева стъпка t, агентът получава състояние s_t и избира действие a_t от пространство с действия A, следвайки политика $\pi(a_t|s_t)$. Политиката π определя поведението на агента. Тя дава функция за преобразуване на състояние s_t до действия a_t. Използвайки дадена политика, агентът получава скаларна награда r_t и преминава в следващо състояние s_{t+1}, което се определя от функцията за награди $R(s, a)$ и функцията, даваща вероятности за преминаване в друго състояние $P(s_{t+1}|s_t, a_t)$. Когато проблемът е дискретен, т.е. може да се разглежда като отделни епизоди, описаният процес продължава докато агентът не достигне до крайно състояние и се рестартира. Общата награда, дефинирана като:

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$$

представлява обезценена стойност с фактор $\gamma \in (0, 1]$. Агентът се опитва да максимизира очакваната стойност за такава дългосрочна награда във всяко състояние.

Функция на стойностите дава предсказана обща бъдеща награда, която измерва до колко са добри дадено състояние или двойка състояние-действие. Стойността на дадено действие $Q^\pi(s, a) = E[R_t|s_t = s, a_t = a]$ ни дава очакваната награда за избиране на дейсвие a в състояние s и следвайки политика π. Оптимална стойностна функция $Q^*(s, a)$ предоставя действие a, което максимизира стойността на наградата за дадено състояние s. Може да дефинираме функция даваща стойност на състоянията $V^\pi(s)$, както и оптималната й версия $V^*(s)$ по сходен начин.

2.1.1 Дълбоко обучение

Нека разгледаме един от най-простите статистически модели - линейната регресия (Gauss 1809; Legendre 1805). Нека е дадено множество от N входно-изходни

двойки $\{(x_1, y_1), ..., (x_n, y_n)\}$. Например, нека x да е тегло в кг, а y - височина в см на N човека. Линейната регресия прави предположението, че съществува линейна функция, която преобразува всяко $x_i \in \mathbb{R}^Q$ към $y_i \in \mathbb{R}^D$. Тогава нашият модел е линейна трансформация на входните данни:

$$f(x) = xW + b$$

където W е $Q \times D$ матрица и b е вектор от D елемента. Тогава, задачата се свежда до намиране на такива параметри W и b, които минимизират средната квадратична грешка:

$$e = \frac{1}{N} \sum_i ||y_i - (x_i W + b)||^2$$

В общия случай, връзката между x и y може да не е линейна. Тогава искаме да дефинираме нелинейна функция $f(x)$, която преобразува входните данни до изходни. За тази цел може да приложим linear basis function regression (превод?) (Bishop 2007; Gergonne 1815), където входните данни x се подават на K фиксирани скаларни нелинейни трансформации $\phi_k(x)$ за създаване на свойствен вектор $\Phi(x) = [\phi_1(x), ..., \phi_k(x)]$. Трансформациите ϕ_k наричаме базисни функции. Върху така създадения вектор се прилага линейна регресия. LBFR може да се сведе до линейна регресия, когато $\phi_k(x) := x_k$ и $K = Q$. Този тип функции се смятат за фиксирани и взаимно ортогонални. Когато тези ограничения се пропуснат говорим за *параметризирани* базисни функции.

2.1.1.1 Изкуствени невронни мрежи

Когато подредим параметризирани базисни функции в йерархия, може да говорим за изкуствени невронни мрежи. Всеки свойствен вектор в тази йерархия ще наричаме слой. Композицията от подобни слоеве води до голямата гъвкавост на тези модели. Често те постигат високи резултати на различни задачи и могат да се приложат върху реални проблеми, работещи върху терабайти от данни.

Feed-forward neural networks. Нека разгледаме модел с един *скрит слой* (Rumelhart et al. 1985). Нека x е вектор с Q елемента, представящ входните

данни. Трансформираме го с афинна трансформация до вектор с K елемента. Отбелязваме с W_1 линейната преобразуваща матрица (матрица на теглата) и с b транслацията използвана за трансформиране на x за да получим $xW_1 + b$. Върху всеки елемент на получената матрица се прилага нелинейна функция $\sigma(\cdot)$. Резултатът е т. нар. *скрит слой*, а всеки елемент се нарича *мрежова единица*. Върху резултата се прилага втора линейна трансформация с матрица на теглата W_2, която преобразува скрития слой до изходен вектор с D елемента. Имаме $Q \times K$ матрица W_1, $K \times D$ матрица W_2 и b - вектор от K елемента. Резултат от дадена невронна мрежа би бил:

$$\hat{y} = \sigma(xW_1 + b)W_2$$

при дадени входни данни x.

Когато използваме невронната мрежа за решаване на регресионна задача, може да минимизираме Евклидовата грешка:

$$e^{W_1,W_2,b}(X, Y) = \frac{1}{2N} \sum_{i=1}^{N} ||y_i - \hat{y}_i||^2$$

където $\{y_1, \ldots, y_n\}$ са N наблюдавани изходни стойности, $\{\hat{y_1}, \ldots, \hat{y_n}\}$ са изходни данни от модела, а $\{x_1, \ldots, x_n\}$ са входните данни. Предполагаме, че минимизирайки тази грешка спрямо W_1, W_2, b ще получим модел, който генерализира добре при нови данни $X_{\text{test}}, Y_{\text{test}}$.

Когато задачата е да се предскаже класът, към който x принадлежи, от множеството $\{1, \ldots, D\}$, използваме същия модел. Промяната се състой в това, че прилагаме softmax функция върху получения резултат. Тази функция ни дава нормализирани оценки за всеки клас:

$$\hat{p_i} = \frac{exp(\hat{y_i})}{\sum d' exp(\hat{y_i'})}$$

Когато вземем логаритъма от горната функция, получаваме softmax грешка:

$$e^{W_1,W_2,b}(X,Y) = -\frac{1}{N}\sum_{i=1}^{N} log(\hat{p}_{i,c_i})$$

където $c_i \in \{1,\ldots,D\}$ е наблюдаваният клас за вход i.

Описаният по-горе модел има проста структура, но може да бъде разширен за по-специализирани задачи. Този тип по-сложни модели се използват, когато задачите изискват обработка на поредици или изображения.

Convolutional Neural Networks CNN е архитектура (LeCun et al. 1989), която се използва при изображения. Задачи, които до скоро се смятаха за нерешими, имат решения посредством този тип модели (Hinton et al. 2012). Моделът е създаден чрез рекурсивно приложение на конволуции и обединяващи слоеве. Конволуционният слой е линейна трансформация, която запазва пространствена информация от входното изображение.

Recurrent neural networks (RNN) RNN е модел (Rumelhart et al. 1985; Werbos 1988), базиран на поредици от данни, който се използва за обработка на текст, обработка на видео и други (Kalchbrenner & Blunsom 2013; Sundermeyer et al. 2012). Входните данни за RNN са поредица от символи. За всяка времева стъпка t, проста невронна мрежа е приложена върху единствен символ, както и изходните данни от мрежата от предишната стъпка.

Конкретно, при дадена редица от входни данни $x = [x_1,\ldots,x_t]$ с дължина T, прост RNN модел е създаден чрез повтарящо се приложение на функция f_h. Така се генерира скрито състояние h_t за времева стъпка t:

$$h_t = f_h(x_t, h_{t-1}) = \sigma(x_t W_h + h_{t-1} U_h + b_h)$$

за някаква нелинейна функция σ. Изходните данни от модела може да бъдат дефенирани като:

$$\hat{y} = f_y(h_T) = h_T W_y + b_y$$

Съществуват и по-сложни RNN модели, като LSTM (Hochreiter & Schmidhuber 1997) и GRU (Cho et al. 2014).

2.1.2 Дълбоко подсилено обучение

Този тип методи се класифицират, когато използваме дълбоки невронни мрежи за апроксимиране на някой от компонентите на подсиленото обучение: функция на стойностите $V(s;\theta)$, политика $\pi(a|s;\theta)$ или модела за промяна на състояние и награди. Параметрите θ представляват тегла в дълбоки невронни мрежи. Когато използваме "плитки" модели, като например линейна регресия, дървета за вземане на решения и др. като апроксиматори на функция, имаме "плитко" подсилено обучение с параметри θ за съответния модел. Основната разлика между дълбокото и плиткото подсилено обучение се състой в апроксиматора на функцията, която използват. Когато се използва извън политикова апроксимация - например на нелинейни функции, може да се наблюдават нестабилност и разходимост (Tsitsiklis et al. 1997). Въпреки това, скорошната работа върху дълбоки Q-мрежи (Mnih et al. 2015) и *AlphaGo* (Silver & Hassabis 2016) стабилизират процеса на обучение и постигат много добри резултати.

Дълбокото подсилено обучение започна рязкото си развитие с работата на (Mnih et al. 2015). Преди това, RL даваше нестабилни резултати, когато се използваха нелинейни апроксиматори като невронни мрежи. Дълбоките Q мрежи (DQN) направиха няколко важни приноса: 1) стабилизиране на обучението, използвайки дълбоки невронни мрежи (Lin 1992) 2) подход за цялостно обучение без почти никакво познание за областта 3) обучаване на гъвкава невронна мрежа с еднакъв алгоритъм за изпълняване на различни задачи, например 49 Atari игри (Bellemare et al. 2013), на които се представят по-добре от всеки известен алгоритъм до момента.

2.1.2.1 Double DQN

(Van Hasselt et al. 2016) предложиха Double DQN (D-DQN) за справяне с проблема на прекалена увереност (overestimate?) на Q-learning алгоритъма. В базовият алгоритъм (както и в DQN), параметрите се обновяват според:

$$\theta_{t+1} = \theta_t + \alpha(y_t^\theta - Q(s_t, a_t; \theta_t))\Delta_{\theta_t} Q(s_t, a_t; \theta_t)$$

където

$$y_t^Q = r_{t+1} + \gamma \max_{\alpha} Q(s_{t+1}, a; \theta_t)$$

така че оператора $\max$ използва еднакви стойности, за да избере и оцени дадено действие. Като следствие от това, е по-вероятно да избере недостатъчно добри стойности. Double DQN предлага да оцени алчната политика спрямо невронна мрежа, но използва друга, за да оцени стойността ѝ. Това може да се постигне с малка промяна на DQN алгоритъма, заменяме y_t^Q с:

$$y_t^{D-DQN} = r_{t+1} + \gamma Q(s_{t+1}, \max_{\alpha} Q(s_{t+1}, a_t; \theta_t); \theta_{\bar{t}})$$

където θ_t е параметър за първата невронна мрежа, а $\theta_{\bar{t}}$ е параметър за целевата мрежа.

2.1.2.2 Асинхронни методи

(Mnih et al. 2016) предложи асинхронни методи за четири RL алгоритъма: Q-learning, SARSA, n-step Q-learning and advantage actor-critic и asynchronous advantage actor-critic (A3C). Този подход използва паралелни агенти, които използват различни политики за изучаване на средата. Асинхронните методи могат да се изпълняват върху многоядрени процесори. Те се изпълняват много по-бързо и предоставят по-бързо обучение от други известни методи.

2.2 Бейсова статистика

Избиране на следващо действие по време на създаване на тестов случай пряко зависи от увереността във взимането на правилното решение. Несигурността от избиране на действие може да бъде моделирана посредством Бейсов подход.

Нека θ е неизвестна стойност, която може да е скаларна, векторна или матрица. Методите за статистически извод (inference) могат да ни помогнат да я намерим. Класическият статистически подход третира θ като фиксирана стойност. Един-

ствената информация, която използваме за намиране на неизвестната стойност, идва от данните, с които разполагаме. Изводът се базира на резултат, получен от фунцкията на правдоподобие на θ, която свързва стойности от $p(y|\theta)$ с всяка възможност на θ, където $y = (y_1, ..., y_n)$ е вектор с наблюдавани стойности.

Бейсовият подход третира θ като случайна стойност. За достигане на извод се използва разпределението на параметри при дадени данни $p(\theta|y)$. Това разпределение се нарича апостериорно. Освен функцията на правдоподобие, Бейсовият подход включва априорно разпределение $p(\theta)$, което представя вярванията ни за θ преди да се разгледат данните.

Теоремата на Бейс дава връзка между фунцкията на правдоподобие и априорното разпределение:

$$p(\theta|y) = \frac{p(\theta|y)p(\theta)}{p(y)}$$

където:

$$p(y) = \int p(y|\theta)p(\theta)d\theta$$

Формулата на Бейс може да бъде пренаписана по следния начин:

(1)
$$p(\theta|y) \propto p(\theta|y)p(\theta)$$

тъй като $p(y)$ не зависи от θ

Когато θ е многомерна величина може да напишем уравнение (1) използвайки маргиналните апостериорни разпределения като например:

$$p(\theta_1|y) = \int p(\theta|y)d\theta_2$$

където $\theta = (\theta_1, \theta_2)$. В много случаи резултатите са многомерни и точни изводи може да бъдат направени само аналитично. Поради тази причина често се използват приближения.

2.2.1 Монте Карло алгоритми за Марковски Вериги (MCMC)

MCMC алгоритмите правят неявно интегриране като взимат извадки от апостериорното разпределение. По този начин се намират приближения на стойностите, от които се интересуваме.

В съществото си тези методи създават Марковска верига с апостериорното разпределение на параметрите като стационарно разпределение. Когато веригата е крайна и повтаряща се, стойността на θ може да бъде оценена от извадки на средни пътища. Генерираните извадки $\theta^{(t)}, t = 1, \ldots, N$ от това разпределение дават представа за целевото разпределение.

2.2.1.1 Метрополис-Хастингс алгоритъм

Този алгоритъм е предложен от Metropolis (Metropolis et al. 1953) и по-късно генерализиран от Hastings (Hastings 1970). Методът създава Марковска верига с желаното стационарно разпределение. Алгоритъмът избира кандидат стойност θ' от предварително избрано разпределение $q(\theta, \theta')$, където $\theta' \neq \theta$. Избраната стойност θ' се проверява чрез приеми-откажи метод (accept-reject step), за да се подсигури, че принадлежи на целевото разпределение.

2.2.1.2 Извадки на Гибс

Този метод, предложен от Geman и Geman (Geman & Geman 1984), често се представя като специален случай на Метрополис-Хастингс алгоритъма.

2.2.2 Извод със свободни вариационни параметри

Variational Inference (VI) методите обикновено предлагат по-добри резултати спрямо MCMC, когато времето за изпълнение е ограничено. Допълнително предимство на тези подходи е, че те са детерминирани. Систематичната грешка и дисперсията се приближават до 0 при MCMC методите, за колкото повече време бъдат оставени да се изпълняват те. Тези свойства правят MCMC алгоритмите

много ефективни на теория. В практиката обаче, времето за изпълнение и изчислителната мощ са ограничени. Това налага търсенето на по-бързо методи дори когато това намаля точността на получените резултати.

Този тип методи дефинират приближено вариационно разпределение $q_\omega(\theta)$, параметризирано от ω, с лесна за оценяване структура. Искаме приближеното разпределение да е максимално близко до това на апостериорното. За целта свеждаме задачата до оптимизационна и минимизираме Kullback-Leibler (KL) (Kullback & Leibler 1951) отклонението спрямо ω. Интуитивно, KL измерва приликата между две разпределения:

$$KL(q\omega(\theta) \,||\, p(\theta|x,y)) = \int q\omega(\theta) log \frac{q\omega(\theta)}{p(\omega|x,y)} d\omega$$

(Define x,y - dataset)

Този интеграл е дефиниран, когато $q\omega(\theta)$ е непрекъсната спрямо $p(\theta|x,y)$. Нека $q^*_\omega(\theta)$ е минимизираща точка (може да е локален минимум). Тогава KL може да ни даде приближение на апостериорното разпределение:

$$p(y^*|x^*,x,y) \approx \int p(y^*|x^*,\theta) q^*_\omega(\theta) d\theta =: q^*_\omega(y^*|x^*)$$

VI методите заменят изчисляването на интеграли с такова на производни. Това е много подобно на оптимизационните методи използвани в DL. Основната разлика се състой в това, че оптимизацията е върху разпределения, а не точкови оценки. Този подход запазва много от предимствата на Бейсовото моделиране и представя вероятностни модели, които дават оценка на несигурността в изводите си.

2.2.3 Бейсови Невронни Мрежи

Един от големите недостатъци на съществуващите архитектури на невронни мрежи е, че изводите, които получаваме, са оценки на точки. Моделите не казват до колко са сигурни в предложените резултати. Когато например един лекар получи резултат от даден модел, той трябва да знае защо и как моделът е

стигнал до него. Бейсовата статистика може да даде отговор на тези въпроси (Gal & Ghahramani 2015). Дори при модели използващи RNN, Бейсова интерпретация на задачата дава по-добри резултати от съществуващи такива (Gal & Ghahramani 2016).

Бейсови невронни мрежи, предложени в края на 80-те години (Kononenko 1989) и задълбочено изучавани по-късно (MacKay 1992; Neal 2012), предлагат вероятностна интерпретация на моделите за дълбоко обучение, като представят теглата им като вероятностни разпределения. Този тип модели са устойчиви на пренастройване (overfitting), предлагат оценки на несигурността и могат да се тренират върху малко на брой данни.

2.3 Автоматизирано тестване на ГПИ

Проверката за правилно поведение на софтуер продукт е неизменна част от създаването му. Откриване и поправяне на всички потенциални проблеми преди той да бъде доставен до крайния потребител може да се сметне за най-добър случай.

2.3.1 Автоматизирано тестване на Android приложения

Мобилните приложения също имат нужда от проверка на качеството. Поради тази причина, в последните години засилено се разглеждат начини за автоматизацията на подобен вид тестове. Много голяма част от извършената работа до момента се състои в създаване на входни данни за приложения за мобилната операционна система Android. Подходите използвани до момента, се различават по начина, по който създават входнни данни и изучават и използват евристики за приложението.

2.3.1.1 Съществуващи системи

Dynodroid (Machiry et al. 2013) е инструмент, който се базира на случайно изучаване. Предлага се и ръчен начин за въвеждане на входнни данни, когато сис-

темата е заседнала.

MobiGUITAR (Amalfitano et al. 2015) строи модел на приложението по време на тестване. За всяко ново състояние се поддържа списък с възможни действия, които се изпълняват използвайки DFS (depth first search) стратегия.

SwiftHand (Choi et al. 2013) се опитва да максимизира покритието на код за тестваното приложение. Допълнително, инструментът се старае да минимизира броя рестартирания на приложението. SwiftHand генерира единствено докосвания и скролвания.

PUMA (Hao et al. 2014) предлага генерална среда за автоматизиране на ГПИ. Инструментът предлага рамка за програмиране, в която могат да бъдат имплементирани различни стратегии за изучаване на тестваното приложение.

2.3.1.2 Покритие на код

Една от основните цели на системите за автоматизирано тестване на софтуер е да постигнат максимално покритие на програмния код. Няколко решения се опитват да постигнат това и за операционната система на Android.

BBoxTester (Zhauniarovich et al. 2015) е рамка за изготвяне на доклади относно покритието на програмния код, без той да бъде наличен. За разлика от други подобни системи, BBoxTester предлага детайлни метрики за покритието на отделни класове, методи и т.н. В основата на системата се използва друг софтуерен продукт - Emma (Roubtsov & others 2005). BBoxTester е система с отворен код, намираща се на https://github.com/zyrikby/BBoxTester. За съжаление системата е неподдържана (от 2015г.) и несъвместима с нови версии на Android.

CovDroid (Yeh & Huang 2015) е друга система за тестване посредством подход на черната кутия (black-box testing). Програмният код на продукта не е наличен. CovDroid изчислява покритието на код като инструментира кода на приложението и използва Android Debug Bridge (adb) за да наблюдава изхода от изпълнение на програмата.

ABCA (Huang et al. 2015) използва подход, много близък до този на CovDroid. Софтуерният пакет може да бъде намерен на http://cc.ee.ntu.edu.tw/~farn/tools/

abca/. По време на този обзор, страницата на инструмента не беше активна. Авторите на статията не отговориха на запитването за активен адрес за изтегляне на ABCA.

GUITracer (Molnar 2015) представя иновативен подход за визуализация на покритието на код, когато приложението е базирано на ГПИ. Основен недостатък на системата е ограничението за работа върху Java AWT, SWING или SWT рамки за изграждане на ГПИ.

GroddDroid (Abraham et al. 2015) предлага автоматично намиране и изпълнение на зловреден софтуер (malware). Системата предлага и измерване на покрит код. Софтуерът може да бъде намерен на http://kharon.gforge.inria.fr/. Програмният код е ясно документиран и лесен за употреба. Един недостатък е използването на Logcat монитора за извличане на метрики за покритие на кода. Повечето от горепосочените системи използват този подход.

2.3.1.3 Текущо състояние (State of the art?)

(Choudhary et al. 2015)

2.3.2 Проверка на качеството

- Достатъчно бързо ли е? (Model should monitor for speed exec anomalies or report just slow parts)
- Как да повторя грешката? (Provide/execute steps for reproduction)
- Има ли разлики в изходните данни? (Change in hierarchy/image screenshot)

Глава 3

Цел и задачи

Целта на настоящата дисертацаионна работа е да създаде система за автоматизирано тестване на ГПИ, която използва за входни данни само визуалния изход на тестваното приложение. За постигане на целта трябва да се изпълнят следните задачи:

- Избор на подходящи оценъчни функции и награди, които да мотивират максималното покритие на програмен код по време на тестване
- Създаване на структури, в които да се запазват поредиците от стъпки, необходими за повтаряне на тестови случай
- Създаване на модел, който генерира поредица от действия, използвани за изграждане на тестовите случаи
- Създаване на модел, който намира аномалии по време на изпълнение на програмата
- Автоматично именуване на отделни екрани и действия с цел улесняване на разбирането
- Създаване и провеждане на експерименти, които да сравнят преложения модел с вече съществуващи такива
- По подадено изображение, йерархия на изгледите и действия, да се определи кои действия са валидни върху кои елементи

Глава 4

Среда за тестване на Android приложения

Много от съществуващите системи за автоматично тестване на Android приложения се опитват да изградят решения, които взимат предвид недостатъците при тестване на приложения. Някои от трудностите повече не съществуват благодарение на напредъка на модерния компютърен хардуер, а други могат да бъдат решени много по-ефективно благодарение на новъведени инструменти за разработка за Android. Например, **SwiftHand** (Choi et al. 2013) се опитва да намали нуждата от преинсталиране на приложението върху устройството. В по-новите си версии, adb, предлага способ за изчистване на състоянието на дадено приложение, без нужда от преинсталирането му.

От особена важност за изграждане на алгоритъм в среда за подсиленото обучение е наличието на награда. Повечето от изградените системи се опитват да максимизират покритието на код. В практиката тази метрика е важна, но и недостатъчна. Фактът, че дадена част от програмния код се е изпълнила и не е предизвикала грешка в програмата не означава, че поведението на програмата е правилно (или не се е променило без това бъде желания ефект от разработчиците).

За нуждите на текущата работа и всеки желаещ да използва се предлага обща среда за тестване на Android приложения. Поради липсата на други свободни инструменти (или такива, които са използваеми). Системата е свободна за из-

ползване, с отворен код и може да бъде намерена на https://github.com/curiousily/dissertation (replace this with new repo link).

Средата се състои от два основни компонента - клиент и сървър. Сървърът работи върху Android устройството и предоставя данни за постигнатото покритие на код, изпълнение на действията, генерирани от модела и изображение за текущото състояние. Клиентът предоставя възможните действия на модела, както и комуникира със сървъра за да представи неговата функционалност.

Клиентът предоставя интерфейс към средата подобен на този на OpenAI gym (Brockman et al. 2016). Двата основни метода, които реализира са `reset()` и `step(action)`. `reset()` предоставя възможност на средата да се върне до първоначално състояние. Това се постига чрез спиране на приложението (ако то е стартирано), изтриване на данните поддържащи състоянието му, стартирането му и предоставяне на образ от екрана, както и възможните действия за състоянието. Изброената функционалност се реализира посредством adb команди и библиотеката `uiautomator` https://github.com/xiaocong/uiautomator. Методът `step(action)` изпълнява избраното действие и предоставя новото състояние на средата, заедно с получената награда и новите възможни действия. Тук също се взима решение дали текущия епизод от обучението е приключил.

Множеството от възможните действия за текущото състояние се базират на броя и видовете графични елементи в него. Всеки елемент върху който може да се извърши докосване, задържане, скролиране, влачене и т.н. се превръща в действие. Множеството от графични елементи се извлича посредством библиотеката `uiautomator`.

Наградата за всяка избрана стъпка пряко се базира на покритието на код за текущия епизод. Стойността се изменя в интервала $[0; 1.0]$ и е нарастваща. Получаването на наградата след всяка стъпка е необходимо за обучение на модела. Скоростта на изпълнение пряко влияе на общото бързодействие на системата. За намиране на текущото покритие на код и изграждане на доклад се използва `JaCoCo` (Hoffmann et al. 2009). `JaCoCo` е интегриран в инструментите за разработка на Android и се използва основно, когато е нужно покритие на код базирано на преминали тестове.

За целта на текущата работата бяха направени някои промени, които предоста-

вят възможност за извличане на необходимите данни, докато програмата се изпълнява и не е в тестова среда. Няколко подхода бяха изпробвани за изграждане на крайните доклади. Първоначално бяха използвани adb команди и генериране на доклад посредством gradle задача. Бързодействието не беше задоволително - необходими бяха около 2 секунди на съвременен мобилен компютър. Около повината от времето се губеше в генериране на доклад, който предоставя повече от необходимата информация.

Крайното решение използва комбинация от `HTTP` сървър на устройството, клиент, специализиран начин за записване на данните за покритие на код и специализиран генератор за доклади. Допълнително бързодействие се постига чрез Nailgun https://github.com/martylamb/nailgun сървър, който изпълнява генератора за доклади. Така описаните оптимизации извършват необходимата работа за около 20 милисекунди (или 0.02 секунди) на същия компютър.

Глава 5

Генериране на входни данни за тестови случаи

Глава 6

Намиране на аномалии в тестови случаи

Глава 7

Експерименти и резултати

Глава 8

Заключение

8.1 Нерешени проблеми

8.2 Бъдеща работа

8.3 Дискусия

Приложение 1: Някои важни вероятностни разпределения

Приложение 2: Фигури

Литература

Abraham, A. et al., 2015. GroddDroid: A gorilla for triggering malicious behaviors. In *Malicious and unwanted software (malware), 2015 10th international conference on*. IEEE, pp. 119–127.

Amalfitano, D. et al., 2015. MobiGUITAR: Automated model-based testing of mobile apps. *IEEE Software*, 32(5), pp.53–59.

Bellemare, M.G. et al., 2013. The arcade learning environment: An evaluation platform for general agents. *J. Artif. Intell. Res.(JAIR)*, 47, pp.253–279.

Bishop, C., 2007. Pattern recognition and machine learning (information science and statistics), 1st edn. 2006. corr. 2nd printing edn. *Springer, New York*.

Boyan, J.A., Littman, M.L. & others, 1994. Packet routing in dynamically changing networks: A reinforcement learning approach. *Advances in neural information processing systems*, pp.671–671.

Brockman, G. et al., 2016. OpenAI gym. *arXiv preprint arXiv:1606.01540*.

Cho, K. et al., 2014. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*.

Choi, W., Necula, G. & Sen, K., 2013. Guided gui testing of android apps with minimal restart and approximate learning. In *ACM sigplan notices*. ACM, pp. 623–640.

Choudhary, S.R., Gorla, A. & Orso, A., 2015. Automated test input generation for android: Are we there yet?(e). In *Automated software engineering (ase), 2015 30th ieee/acm international conference on*. IEEE, pp. 429–440.

Crites, R.H. & Barto, A.G., 1996. Improving elevator performance using reinforcement learning. *Advances in neural information processing systems*, 8.

Gal, Y. & Ghahramani, Z., 2016. A theoretically grounded application of dropout in recurrent neural networks. In *Advances in neural information processing systems*. pp. 1019–1027.

Gal, Y. & Ghahramani, Z., 2015. Dropout as a bayesian approximation: Representing model

uncertainty in deep learning. *arXiv preprint arXiv:1506.02142*, 2.

Gauss, C.F., 1809. *Theoria motus corporum coelestium in sectionibus conicis solem ambientium auctore carolo friderico gauss*, sumtibus Frid. Perthes et IH Besser.

Geman, S. & Geman, D., 1984. Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *IEEE Transactions on pattern analysis and machine intelligence*, (6), pp.721–741.

Gergonne, J., 1815. Application de la méthode des moindres quarrésa l'interpolation des suites. *Annales de Math. Pures et Appl*, 6, pp.242–252.

Hao, S. et al., 2014. Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps. In *Proceedings of the 12th annual international conference on mobile systems, applications, and services*. ACM, pp. 204–217.

Hastings, W.K., 1970. Monte carlo sampling methods using markov chains and their applications. *Biometrika*, 57(1), pp.97–109.

Hinton, G.E. et al., 2012. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*.

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

Hoffmann, M. et al., 2009. Jacoco code coverage tool. online, 2009.

Huang, S.-Y. et al., 2015. ABCA: Android black-box coverage analyzer of mobile app without source code. In *Progress in informatics and computing (pic), 2015 ieee international conference on*. IEEE, pp. 399–403.

Joachims, T. et al., 1997. Webwatcher: A tour guide for the world wide web. In *IJCAI (1)*. Citeseer, pp. 770–777.

Kalchbrenner, N. & Blunsom, P., 2013. Recurrent continuous translation models. In *EMNLP*. p. 413.

Kononenko, I., 1989. Bayesian neural networks. *Biological Cybernetics*, 61(5), pp.361–370.

Kullback, S. & Leibler, R.A., 1951. On information and sufficiency. *The annals of mathematical statistics*, 22(1), pp.79–86.

LeCun, Y. et al., 1989. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4), pp.541–551.

Legendre, A.M., 1805. *Nouvelles méthodes pour la détermination des orbites des comètes*, F. Didot.

Lin, L.-J., 1992. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8(3-4), pp.293–321.

Machiry, A., Tahiliani, R. & Naik, M., 2013. Dynodroid: An input generation system for android apps.

In *Proceedings of the 2013 9th joint meeting on foundations of software engineering*. ACM, pp. 224–234.

MacKay, D.J., 1992. A practical bayesian framework for backpropagation networks. *Neural computation*, 4(3), pp.448–472.

Metropolis, N. et al., 1953. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21(6), pp.1087–1092.

Mnih, V. et al., 2016. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*.

Mnih, V. et al., 2015. Human-level control through deep reinforcement learning. *Nature*, 518(7540), pp.529–533.

Molnar, A.-J., 2015. Live visualization of gui application code coverage with guitracer. In *Software visualization (vissoft), 2015 ieee 3rd working conference on*. IEEE, pp. 185–189.

Neal, R.M., 2012. *Bayesian learning for neural networks*, Springer Science & Business Media.

Roubtsov, V. & others, 2005. Emma: A free java code coverage tool.

Rumelhart, D.E., Hinton, G.E. & Williams, R.J., 1985. *Learning internal representations by error propagation*, DTIC Document.

Silver, D. & Hassabis, D., 2016. AlphaGo: Mastering the ancient game of go with machine learning. *Research Blog*.

Sundermeyer, M., Schlüter, R. & Ney, H., 2012. LSTM neural networks for language modeling. In *Interspeech*. pp. 194–197.

Sutton, R.S. & Barto, A.G., 1998. *Reinforcement learning: An introduction*, MIT press Cambridge.

Tsitsiklis, J.N., Van Roy, B. & others, 1997. An analysis of temporal-difference learning with function approximation. *IEEE transactions on automatic control*, 42(5), pp.674–690.

Van Hasselt, H., Guez, A. & Silver, D., 2016. Deep reinforcement learning with double q-learning. In *AAAI*. pp. 2094–2100.

Werbos, P.J., 1988. Generalization of backpropagation with application to a recurrent gas market model. *Neural networks*, 1(4), pp.339–356.

Yeh, C.-C. & Huang, S.-K., 2015. CovDroid: A black-box testing coverage system for android. In *Computer software and applications conference (compsac), 2015 ieee 39th annual*. IEEE, pp. 447–452.

Zhauniarovich, Y. et al., 2015. Towards black box testing of android apps. In *Availability, reliability and security (ares), 2015 10th international conference on*. IEEE, pp. 501–510.