Dependency-aware Form Understanding

Shaokun Zhang
MOE Key Lab of HCST,
Dept of Computer Science,
Peking University, Beijing, China
skzhang@pku.edu.cn

Yuanchun Li
Microsoft Research
Beijing, China
Yuanchun.Li@microsoft.com

Weixiang Yan
Beijing University of
Posts and Telecommunications
Beijing, China
yanweixiang@bupt.edu.cn

Yao Guo†
MOE Key Lab of HCST,
Dept of Computer Science,
Peking University, Beijing, China
yaoguo@pku.edu.cn

Xiangqun Chen
MOE Key Lab of HCST,
Dept of Computer Science,
Peking University, Beijing, China
cherry@pku.edu.cn

***Abstract*—Form understanding is an important task in many fields such as software testing, AI assistants, and improving accessibility. One key goal of understanding a complex set of forms is to identify the dependencies between form elements. However, it remains a challenge to capture the dependencies accurately due to the diversity of UI design patterns and the variety in development experiences. In this paper, we propose a deep-learning-based approach called DependEX, which integrates convolutional neural networks (CNNs) and transformers to help understand dependencies within forms. DependEX extracts semantic features from UI images using CNN-based models, captures contextual patterns using a multilayer transformer encoder module, and models dependencies between form elements using two embedding layers. We evaluate DependEX with a large-scale dataset from mobile Web applications. Experimental results show that our proposed model achieves over 92% accuracy in identifying dependencies between UI elements, which significantly outperforms other competitive methods, especially for heuristic-based methods. We also conduct case studies on automatic form filling and test case generation from natural language (NL) instructions, which demonstrates the applicability of our approach.**

***Index Terms*—Form understanding, dependencies, deep learning, CNN, automatic form filling**

I. Introduction

Form understanding is useful for various purposes in many software engineering applications [1]–[4]. For example, in automated testing for mobile applications, valid inputs are required in the form fields to explore hidden states, in order to achieve higher code coverage. In AI assistants, natural language commands are interpreted to carry out specific tasks, most of which are related to forms. It is crucial for understanding complex forms to extract dependencies between form elements. However, it is extremely challenging to model the dependencies mainly due to: 1) the differences in code behavior and programming experience between developers; 2) the diversity of its layout, style, and content.

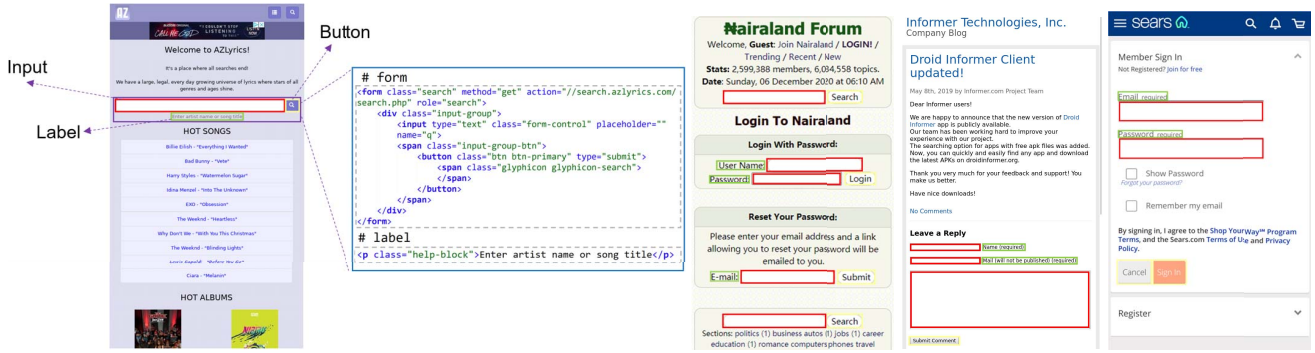
Many studies have been conducted to describe and contextualize the form inputs using semantic annotations such as *tooltip* and *aria* attributes [5]–[7]. However, such annotations are not available in many form elements, as shown in Figure 1(a). This is mainly due to the poor development behavior of programmers, which leads to a lack of descriptive annotations or incorrect attributes in elements. The scenario has been supported in more recent literature [8], [9]. In addition, several record-and-replay methods are proposed to understand the form inputs. For instance, Li *et al.* [6] associated text labels with individual UI elements by aggregating user interaction trace collected by app developers. However, this strategy with limited scalability is infeasible and inapplicable when confronted with an abundant number of applications. For these cases, it is necessary to identify the label of inputs based on visual or logical contexts.

Several attempts have been made to capture dependencies between form inputs and labels by developing heuristic rules [9]–[11]. For instance, Pasupat *et al.* [9] proposed to incorporate spatial context into the embedding-based model to understand web elements such as text boxes. More specifically, one sample was taken from all neighboring label texts of the element as its semantics. Consider the example in Figure 1(d). Suppose that our goal is to identify the label related to the first input box. The heuristic method mentioned above will fail with a 50% probability because there are adjacent elements both above and below the input box. In summary, this approach has shown limited effectiveness in capturing relationships (dependencies) between inputs and labels. It is also difficult to design a good heuristic rule due to the complexity of UI layouts, as illustrated in Figure 1.

Conventional machine learning methods have also been proposed to understand dependencies within forms [12]–[14]. However, they rely heavily on predefined features hand-engineered by domain experts. To improve the robustness and effectiveness, a more intelligent method for form understanding is clearly needed. Recently, deep learning models have been demonstrated as a powerful tool to automatically extract latent patterns from large volumes of samples [15], [16]. Especially for CNNs, they can learn hierarchical feature

†Corresponding author.

2332-6549/21/$31.00 ©2021 IEEE
DOI 10.1109/ISSRE52982.2021.00026



(a) below (b) left (c) right (d) above

Fig. 1: Examples of "label-element" dependencies in different mobile applications. Here we show that, in real applications, the descriptions (i.e., labels, shown in green boxes) of an input element (shown in red boxes) can be located in different places around the input elements.

representation from inputs in an end-to-end manner with a broad spectrum of applications in computer vision tasks [17]–[20]. More recently, He *et al.* [19] proposed a model called ResNet to understand the context of different images, achieving superior performance in object recognition. Although these models have the potential to be applied to software engineering tasks such as dependency understanding, the majority of deep-learning-based models commonly require large-scale, well-annotated training data.

To deal with these issues, we first construct a new dataset *UIE-Dependency* for form understanding, which involves a large number of GUI pages. The dataset consists of 25,140 annotated samples represented as triplets, which describe dependencies between two form elements. We focus on two representative dependency types: *label-element* and *input-action*. Figure 1 presents some examples within these raw pages, which mainly include what we refer as "label-element" dependencies, which shows the relationship of a description (i.e., label) of a given input element. The "input-action" dependency is used to describe where an action (button) is followed after all input elements were filled. Besides these two dependency types, we also include a type called "others" for evaluation purpose. Furthermore, all samples are annotated by a combination of automated and manual approaches, which we will describe in detail later.

Inspired by popular methods used in image classification and machine translation, we propose a deep-learning-based model called DependEX, which incorporates CNNs and transformers to extract dependencies between elements. More specifically, DependEX employs CNN-based models to extract semantic features of UI elements and then applies multiple transformer encoder layers to model contextual patterns. Finally, two fully connected layers are exploited to capture dependencies between form elements.

We evaluate DependEX with the aforementioned dataset collected from mobile Web applications. Experimental results indicate that our approach performs significantly better than the other competitive models in capturing dependencies between UI elements, achieving an accuracy of 92.12%. In addition, we also conduct two case studies on automatic form filling and test generation from NL instructions, which incorporate DependEX in the corresponding pipelines and demonstrate the benefits and applicability of our method.

In summary, this paper makes the following main contributions:

- We propose a new deep learning-based dependency-aware form understanding approach DependEX, which can learn better discriminative features using a multi-layer transformer encoder module as it incorporates relative position information.
- We conduct extensive experiments on a large-scale form dependency dataset collected from mobile Web applications. The experimental results demonstrate that our proposed approach outperforms state-of-the-art prediction methods.
- We demonstrate the applicability of the proposed method through two case studies: automatic form filling and test case generation from NL instructions. The results show that it can adapt to different scenarios and exhibit better performance than existing approaches.

In addition, we have constructed the first large-scale dataset of form dependencies between UI elements, which will be released to the research community[1].

II. Preliminaries

In the context of mobile applications (as well as in general software applications), a form contains a tree of elements, which are divided into two main types: *labels*, *inputs* such as checkbox, radio box, and selection lists, and *actions* such as buttons. Each input element generally corresponds to a textual description. Note that the terms "labels" and "textual descriptions" are used interchangeably in the remainder of this paper. Besides, there may be multiple actions in one

[1]https://github.com/skzhangPKU/DependEX

form, as shown in Fig. 1(d). To automatically and efficiently discover the large amount of available content hidden behind the form, it is necessary to understand its intrinsic patterns (dependencies between UI elements). Next, we will provide the definitions of relevant concepts.

Definition 1 (The Form Understanding Problem). In this work, we seek to understand the forms from the perspective of dependencies between elements. It is transformed into a typical pattern classification problem. Given a large number of annotated samples, we aim to construct a structural model $\mathcal{G}$ to learn dependencies between UI elements. The problem can be formulated as follows:

$$\mathcal{Y}_d = \mathcal{G}\left(\mathcal{U}_e, \mathcal{P}_e; \Theta\right) \tag{1}$$

where $\mathcal{U}_e$ and $\mathcal{P}_e$ denote UI elements and the corresponding view hierarchies, respectively, $\mathcal{Y}_d$ represents the dependencies between elements, and Θ stands for learnable parameters. The view hierarchy, like the one depicted in the lower left section of Fig. 2, is obtained by using a webbot.

There may exist multiple dependencies between form elements. In this paper, we primarily focus on two aspects: understanding the semantics of form inputs and determining which action to choose for interaction, which are represented as *label-element* and *input-action* dependency extraction problems, respectively. However, our work is generic and can be easily extended to many other dependencies.

Definition 2 (Label-Element Dependency Extraction). Let $\mathcal{F}_o$ be a form in the UI of a mobile Web application $\mathcal{U}$, which contains a set of input elements $\mathcal{E} = \{e_1, e_2, \cdots, e_n\}$. Let $\mathcal{D} = \{d_1, d_2, \cdots, d_m\}$ be a set of textual descriptions in $\mathcal{U}$. The goal of label-element dependency extraction is determining the set of mappings $\mathcal{M}$ between each input field e_i in $\mathcal{E}$ and the corresponding descriptive labels d_j in $\mathcal{D}$, such that:

$$\mathcal{M} = \{(e_i, d_j) \mid \bigcup_{i=1}^{n} e_i = \mathcal{E} \textit{ and } \bigcup d_j \subseteq \mathcal{D} \textit{ and } d_j \textit{ is the best semantic description of } e_i\}$$

Definition 3 (Input-Action Dependency Extraction). Let $\mathcal{A} = \{a_1, a_2, \cdots, a_p\}$ be a set of actions in form $\mathcal{F}_o$. The goal of input-action dependency extraction is to find a set of corresponding target actions $\hat{\mathcal{A}}$ given all required input fields $\mathcal{E}_* = \{e_{1^*}, e_{2^*}, \cdots, e_{q^*}\}$, such that:

$$\hat{\mathcal{A}} = \{a_k \mid \mathcal{E}_*;\ a_k \in \mathcal{A} \textit{ and } \mathcal{E}_* \subseteq \mathcal{E} \textit{ and } a_k \textit{ can result in a valid form submission if all fields in } \mathcal{E}_* \textit{ are filled}\}$$

III. Our Approach

To solve the problem formulated in the previous section, we propose DependEX, a deep learning based approach to dependency-aware form understanding. The key goal of DependEX is to develop a general approach that is able to identify (and distinguish between) different dependencies within a form, in order to help understand forms and boost the efficiency in tasks such as software testing.

Fig. 2 illustrates the overall architecture of our approach, which consists of the following functional modules: data preprocessing, semantic feature extraction, and contextual feature modeling. The preprocessing module can be split into three main steps, including augmentation, normalization, and resizing. Based on the preprocessed data described above, DependEX leverages a CNN [21] to extract semantic features of UI elements and then applies a multilayer transformer encoder module [22] that incorporates relative position information to model contextual patterns. Finally, two embedded layers are used to learn discriminative features to identify dependencies between elements.

A. Data Preprocessing

Given a UI image I, whose width, height, and channel are W, H, and C, respectively, we extract a set of n bounding boxes about the label, input, and action from the UI tree based on some of the attributes (i.e., "form_id") automatically. The set of bounding boxes is denoted as

$$\mathcal{B} = \{B_1, B_2, \ldots, B_n\} \tag{2}$$

where the bounding box B_i can be described as follows:

$$B_i = (x^i_{min}, y^i_{min}, x^i_{max}, y^i_{max}) \tag{3}$$

Here the coordinates (x^i_{min}, y^i_{min}) and $\left(x^i_{max}, y^i_{max}\right)$ stand for the top-left and bottom-right corner points of the bounding box, respectively. Note that we take the top-left corner of the image as the origin $(0, 0)$ in the coordinate system. Further, we can crop a set of region images based on these coordinates, which are denoted as

$$\mathcal{R} = \{R_1, R_2, \ldots, R_n\} \tag{4}$$

where R_i represents the rectangle area of the original image with (x^i_{min}, y^i_{min}) as its top-left coordinates and $(x^i_{max} - x^i_{min}, y^i_{max} - y^i_{min})$ as its width and height. Then, to lessen the effects of background variation, we paste the cropped region images onto a gray blank image of the same size as the original image. Next, we perform various data transformations, including augmentation (random brightness, contrast, and saturation changing), normalization, and resizing, to avoid the risk of overfitting. The procedure is described as follows:

$$I_{merge} = paste(\mathcal{R}, I_{bg}) \tag{5}$$

$$I_{trans} = transforms(I_{merge}) \tag{6}$$

where I_{merge} and I_{trans} stand for the image produced after the paste and transform operations separately, and I_{bg} stands for the gray blank image.

B. Semantic Feature Extraction

To extract the semantic features from UI images, we leverage CNN models, which have shown superior performance on image recognition tasks compared with conventional methods [23]–[27]. The main reason is that CNN-based models can capture high-level latent patterns from images via a hierarchical layer-based structure. The feature extraction process can be described as follows:

$$\mathcal{F}_m = CNN(I_{trans}; \mathcal{W}) \tag{7}$$

where $\mathcal{F}_m$ indicates the feature map extracted from the image I_{trans} and $\mathcal{W}$ are all learnable parameters of the CNN model. Our CNN model mainly involves two operations: convolution and pooling.

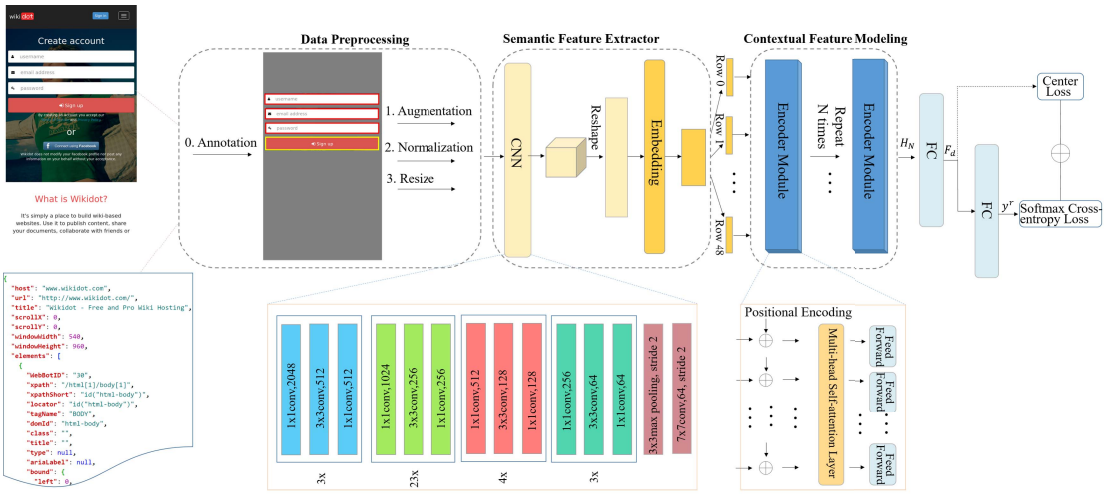


Fig. 2: The overall architecture of DependEX.

Besides, to capture position dependencies of semantic features, the feature map extracted from the CNN is reshaped and then fed into an embedding layer, which is described as follows:

$$\mathcal{F}_m^{'} = reshape(\mathcal{F}_m) \tag{8}$$

$$\mathcal{F}_e = embedding(\mathcal{F}_m^{'}) \tag{9}$$

where $\mathcal{F}_m^{'}$ represents the output produced after the $reshape$ operation, and $\mathcal{F}_e$ is the embedded semantic feature.

C. Contextual Feature Modeling

To capture the spatial context pattern of the embedded semantic features, we adopt the multilayer transformer encoder module, which is a part of the transformer architecture [22]. *Our intuition is that abstract dependencies can be extracted based on relative positional relationships between UI elements within a given context.* Besides, knowledge such as the proximity and closure between semantically correlated UI elements also provides a significant insight into the problem. There are different methods for embedding a collection, where elements are structurally correlated with each other. For instance, graph convolutional networks [28] utilize an adjacency matrix determined by the UI layout to adjust the effect of neighbors on an object. Transformers [29] can learn the relationship between objects by incorporating the relative positions. Thus we choose transformers to capture contextual patterns.

The multilayer module takes the embedded semantic feature as input, i.e.,

$$\mathcal{H}_0 = \mathcal{F}_e = \{\mathcal{F}_e^{r_1}, \mathcal{F}_e^{r_2}, \ldots, \mathcal{F}_e^{r_L}\} \tag{10}$$

where $\mathcal{F}_e^{r_i}$ indicates the i-th element of the embedded feature sequence, and L is the length of the sequence. Further, the multilayer encoder module can be phrased as follows:

$$\mathcal{H}_l = transformer\_encoder(\mathcal{H}_{l-1}), \text{ for } l = 1, \ldots, N \tag{11}$$

where $\mathcal{H}_{l-1}$ represents the input of the i-th encoder module, and N is the number of modules. For the encoder module, it contains three components: positional encoding, multi-head self-attention layer, and feed-forward layer. The positional encoding represents the relative position information of embedded semantic features, which can be integrated into the model to better learn dependencies between UI elements. The multi-head self-attention layer is used to capture local and global patterns between input vectors. The resulting output is a concatenation of attention value vectors generated by multiple heads in the same position.

D. Dependency Relation Identification

As mentioned in the previous sections, we leverage a CNN to extract embedded semantic features of UI elements and a multilayer transformer encoder module that incorporates relative position information to capture contextual patterns. To model dependency relations between UI elements, the contextual features extracted from encoder modules are fed into two fully connected layers, the first of which is followed by a batch normalization layer and a ReLU activation layer. Finally, we apply a softmax operation to the output of fully connected layers. Specifically, it can be formulated as follows:

$$\mathcal{F}_d = W_1^r \times \mathcal{H}_N + b_1^r \tag{12}$$

$$\mathcal{F}_d^{br} = relu(batch\_norm(\mathcal{F}_d)) \tag{13}$$

$$\mathcal{Y}^r = softmax(W_2^r \times \mathcal{F}_d^{br} + b_2^r) \tag{14}$$

where W_*^r and b_*^r are both trainable parameters in the model, $\mathcal{F}_d$ stands for the output of the first dense layer, and $\mathcal{Y}^r$ indicates the estimation score of the model for each category.

Joint Loss Function. Our model is optimized by leveraging a joint loss that consists of softmax cross-entropy loss and center loss. The center loss can contribute to better learn discriminative features. The joint loss is defined as follows:

$$\mathcal{L} = \mathcal{L}_s + \mu\mathcal{L}_c \tag{15}$$

where $\mathcal{L}_s$ denotes the softmax cross-entropy loss, $\mathcal{L}_c$ stands for the center loss, and μ controls the trade-off between $\mathcal{L}_s$ and $\mathcal{L}_c$.

To reduce the within-class variance of discriminative features, we introduce the center loss, which is calculated using the output feature vector of the last hidden layer. The system

assigns a global center to each category and minimizes the distance between the output feature vector and the center. However, it is impractical to calculate the center for all output features. One intuitive solution is to update parameters based on mini-batch. Specifically, the center loss is defined as follows:

$$\mathcal{L}_c = \frac{1}{N_b}\sum_{i=1}^{N_b}\|\mathcal{F}_d^i - z_{t_i^*}\|^2 \tag{16}$$

where t_i^* stands for the category to which the sample corresponding to the i-th output feature belongs, and z_k $(k \in \{1, 2, \ldots, C_n\})$ denotes the global center for the k-th category.The center vector is updated according to the following formula:

$$\frac{\partial L_c}{z_k} = \frac{1}{N_b + 1}\cdot\sum_{i=1}^{N_b}(z_{t_i^*} - \mathcal{F}_d^i)\cdot 1\{t_i^* = k\} \tag{17}$$

where $1\{\cdot\}$ denotes the indicator function. The function value is defined as 1 when the category of the i-th sample is equal to k and 0 otherwise.

IV. Evaluation

Section 4.1 presents the experimental setups including dataset, compared methods and comparing metrics, etc. Section 4.2 compares the performance of DependEX with other competitive models. Section 4.3 investigates the effect of hyperparameters on the overall performance of DependEX. Section 4.4 evaluates the usefulness of our approach with two case studies: automatic form filling and test case generation from NL instructions.

A. Experimental Setups

1) Datasets: We evaluate DependEX with a large-scale dataset (UIE-Dependency) collected from mobile Web applications by ourselves. We implement a crawler to collect raw samples from 20,000 mobile Web sites, which performs actions such as clicks on these sites in order to reach more different UI states. The raw dataset contains a total of 51,695 samples, each of which involves a UI state and a view hierarchy. A UI can contain 0 or more forms, and each form has multiple types (e.g., login, contact, registration, and search). After screening and removal of UIs without forms, the remaining samples are selected for analysis. Here, 0 forms mean that the UI does not interact with the remote server. Furthermore, we adopt a combination of automated and manual methods to label relations between form elements, which results in 25,140 annotated element dependency pairs (label-element: 8558, input-action: 5811, others: 10771). Overall, in terms of label-element and input-action dependencies, around 87% of samples are manually annotated or refined on top of the automatic labeling process. Notably, there are a large number of pairs in the "others" category, where input elements are randomly mapped to incorrect descriptions or actions. This aims to keep a balanced sample distribution. The sample size is 10771, which is fewer than the sum of 8558 and 5881, mostly because there is only one action or description in some forms. 80% of all pairs are taken as the training dataset, and the remaining part is used as a testing dataset. In addition,

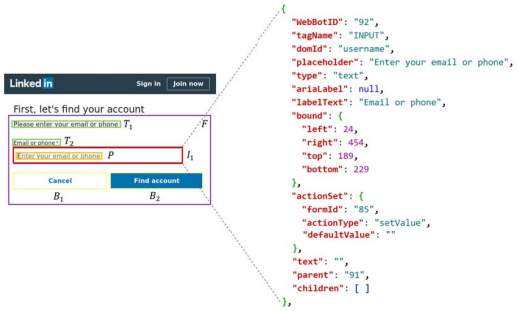


Fig. 3: An illustrative example of the annotation process. Here, the blue box indicates the form F, the green box the text descriptions T_*, the yellow box the buttons B_*, and the red box and the orange box indicate the input element I_1 and its placeholder P. The right part of the figure shows the attributes of I_1.

we also perform a series of basic preprocessing operations, including augmentation, normalization, and resizing.

The whole annotation process is summarized in Algorithm 1. To clarify the procedure of this algorithm, a representative example is presented in Fig. 3. We observe that form F consists of two text descriptions (T_1 and T_2), two buttons (B_1 and B_2), and an input element (I_1) that contains the placeholder (P). In the first phase, shown in lines 3-11, T_1 and T_2 are grouped into the set L, B_1 and B_2 into the A, and I_1 is grouped into the set I. Besides, the input I_1 itself is a semantic description due to the presence of the placeholder text. In the next phase, in lines 13-19 of Algorithm 1, the 'labelText' attribute value of the input I_1 ("Email or phone") is the same as the text of T_2, which implies a label-element dependency between T_2 and I_1. In the last phase, we manually label I_1 and B_2 as the input-action dependency since there are multiple buttons in the form (lines 21-24).

2) Compared Methods: To evaluate the effectiveness of DependEX, we compare it against the following methods:

- **RS** [9]: It randomly selects (RS) one from all neighboring descriptions of the element as its semantics.
- **SCD**: It selects the closest description (SCD) of the element in terms of distance.
- **GP** [7]: It captures dependencies between input elements and labels based on Gestalt principles (GP) of visual perception.
- **GNB** [5]: It is a Gaussian Naïve Bayes (GNB) classifier.
- **VGGNet** [23]: It is made up of a series of convolutional, pooling, and dense layers.
- **ResNet** [24]: It designs a residual structure that uses shortcut connections to mitigate the degradation of networks.
- **GoogleNet** [25]: It consists of a variety of different inception modules, which can model local characteristics at multiple scales.
- **DenseNet** [26]: It has a hierarchical structure, where each layer takes the output of all previous layers as input.

Algorithm 1: The annotation process

for *each form* $\mathcal{F}_o$ **do**
  // Phase 1
  for *each element* $e \in \mathcal{F}_o$ **do**
    if *e.hasText* **then**
      Append e to the set of label fields L;
    if *e.tagName is 'BUTTON'* **then**
      Append e to the set of actions A;
    if *e.tagName is 'INPUT'* **then**
      Append e to the set of input fields I;
      if *e.hasPlaceholder* **then**
        Append $\langle e, (e) \rangle$ to the set of dependencies between labels and input fields D_l;
  // Phase 2
  for *each input field* $e_i \in I$ **do**
    for *each label field* $e_j \in L$ **do**
      if e_j*.text equals* e_i*.labelText* **then**
        if $D_l[e_i]$*.hasElement* **then**
          Append e_j to $D_l[e_i]$;
        else
          Append $\langle e_i, (e_j) \rangle$ to D_l;
  // Phase 3
  if *A.size equals 1* **then**
    Append $\langle I, A[0] \rangle$ to the set of dependencies between input elements and actions D_i;
  else
    Manually annotate input-action dependencies;
Manually annotate unlabeled label-element dependencies;
Manually check and correct annotation errors.

3) Hardware Setup and Parameters: In this study, we conduct all pretreatment operations that include augmentation, normalization, and resizing on a PC (CPU: Intel (R) Core (TM) i7-9700F @ 3.00GHz, Memory: 16GB). The deep learning models are built and implemented in the PyTorch environment on a CUDA-enabled NVIDIA GPU (RTX 2080 Ti).

The weights of all CNN-based architectures are initialized from the corresponding ImageNet pre-trained models and fine-tuned on the UIE-Dependency dataset. For these CNN-based models, all layers except the last one are frozen, and the last layer is replaced with a randomly initialized layer that outputs the number of dependency types. VGGNet, ResNet, and DenseNet consist of 19, 101, and 201 layers, respectively. We also employ an Adam optimizer at an initial learning rate of 0.001 and a batch size of 64 to update the weights.

4) Evaluation Metrics: We assess the performance of machine-learning-based models using the standard metrics (i.e., accuracy, precision, recall, and F1 scores), which are widely used in classification tasks. The evaluation metrics are described as follows:

$$accuracy = \frac{TP + TN}{TP + FN + TN + FP} \tag{18}$$

$$precision = \frac{TP}{TP + FP} \tag{19}$$

$$recall = \frac{TP}{TP + FN} \tag{20}$$

$$F_1\text{-}score = \frac{2 \times precision \times recall}{precision + recall} \tag{21}$$

where TP, TN, FN, and FP stand for true positive, true negative, false negative, and false positive, respectively.

We evaluate the performance of heuristic methods using identification accuracy (the ratio between the number of correctly identified samples and the total number of samples). Notably, their accuracy should be compared with the recall value of classification methods on LE.

B. Overall Results

In this section, we evaluate and compare the performance of different models (RS, GP, SCD, GNB, VGGNet, ResNet, GoogleNet, DenseNet, and DependEX). Table I presents the quantitative results of these models for identifying dependencies between UI elements.

The results show that DependEX performs the best among deep-learning-based models on the dataset. Specifically, DependEX increases the mean accuracy by 15.33%, 16.05%, 19.21%, and 14.99% with respect to VGGNet, ResNet, GoogleNet, and DenseNet, respectively. The results demonstrate the benefits of incorporating transformer encoders into the model, indicating that these pure CNN architectures are less suitable for form understanding. For other evaluation metrics (precision, recall, and F1-score), it also surpasses all competitors, which implies the effectiveness and robustness of DependEX. Moreover, a higher F1-score value indicates that our model tends to exhibit better results on LE than IA[2].

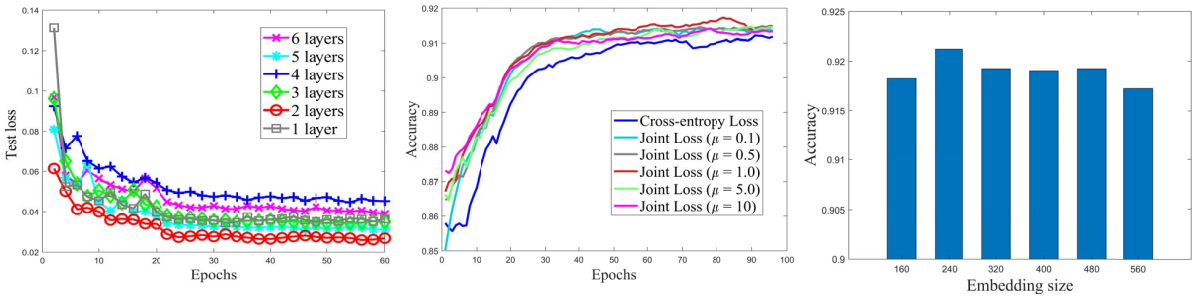
We can also observe that DependEX outperforms RS, SCD, GP, and GNB. The main reason is that the scalability of these heuristic methods and machine-learning-based methods is limited. For example, GP only considers the top and left sides of input elements while ignoring its right and bottom sides. Thus, it will fail for samples like those shown in Fig.s 1 (a) and (c). We also show an illustrative example missed by both RS and SCD but correctly identified by DependEX in Fig. 5. In this case, RS and SCD mistakenly take the 'PRICE' description as the semantics of the input element. We believe that it is difficult to design a good heuristic rule due to the complexity of UI layouts. Thus a learning-based approach makes sense because modeling dependencies between form elements ultimately requires some heuristics, which is exactly what DependEX is designed to learn.

There exhibit slight inconsistencies in the performance ranking of baseline models under different evaluation metrics. From Table 1, we observe no strong correlation between evaluation indicators. For instance, the average accuracy, precision, recall, and F1-score values of DenseNet are 77.13%, 75.42%, 75.22%, and 75.29%, respectively. Compared with VGGNet, it achieves better performance in terms of the mean accuracy

[2]F1-score considers both precision and recall, and provides an overall measure of classification performance.

TABLE I: Performance comparison of different models. Here, LE and IA indicate the two dependency classes: label-element and input-action, respectively. AVG represents the average value for all classes. *Note that RS, SCD, and GP are heuristic methods, which only consider label-element dependency extraction. Therefore, their precision, recall, and F1-score are not reported in the table.*

	Model	Accuracy (%)	Precision (%)			Recall (%)			F1-score (%)		
			LE	IA	AVG	LE	IA	AVG	LE	IA	AVG
Baselines	RS [9]	28.86	-	-	-	-	-	-	-	-	-
	SCD	46.04	-	-	-	-	-	-	-	-	-
	GP [7]	43.42	-	-	-	-	-	-	-	-	-
	GNB [5]	41.98	40.27	42.33	42.25	59.74	15.57	39.05	48.11	22.76	37.95
Pretrained CNN architectures	VGGNet [23]	76.79	70.09	67.63	75.83	82.63	64.16	75.24	75.85	65.85	75.23
	ResNet [24]	76.07	71.44	65.79	74.32	80.18	57.57	73.56	75.56	61.41	73.76
	GoogleNet [25]	72.91	72.91	61.77	71.83	79.04	59.28	71.23	72.14	60.50	71.19
	DenseNet [26]	77.13	73.46	66.55	75.42	77.39	64.16	75.22	75.37	65.33	75.29
Our proposed model	**DependEX**	**92.12**	**93.20**	**81.96**	**90.98**	**85.82**	**93.24**	**91.94**	**89.36**	**87.24**	**91.29**



(a) Test loss values with different numbers of transformer encoder layers (*lower is better*).

(b) Performance with different values of μ in the joint loss function (*higher is better*).

(c) Performance with different embedding size (*higher is better*).

Fig. 4: The results of sensitivity analysis.

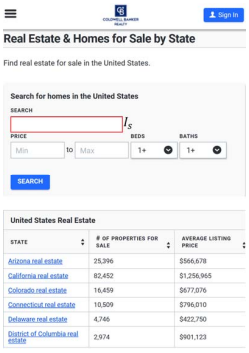


Fig. 5: One failure example in both RS and SCD. Here, the red box indicates the input element I_s.

and F1-score, but not in terms of the average precision and recall. The results also indicate that it is noteworthy that our model can perform well under all metrics.

C. Sensitivity Analysis

In the first experiment, we investigate the parameter sensitivity of N, the number of transformer encoder layers, on the overall performance of DependEX. Fig. 4 (a) shows the test loss curves of DependEX under different quantities of transformer encoder layers (1 to 6). We can see that DependEX achieves convergence when the number of epochs is about 25. Moreover, DependEX performs differently for varying numbers of encoder layers. In particular, DependEX yields the lowest test error when N is equal to 2.

We also evaluate the effect of the hyperparameter μ in the joint loss function on model performance. Fig. 4 (b) shows the overall accuracy of DependEX using the joint loss function with μ set to 0, 0.1, 0.5, 1.0, 5.0, and 10. In particular, $\mu = 0$ indicates that the objective function is equivalent to the categorical cross-entropy. For visualization purposes, we perform smoothing operations[3] on these accuracy curves. We can see that the joint loss function enables a higher classification accuracy than the cross-entropy loss function, which implies the effectiveness of the center loss in modeling dependencies between UI elements. Besides, among a set of different parameter values, DependEX with $\mu = 1.0$ performs the best, which yields an accuracy of 92.12%.

The last experiment is to explore how the embedding size affects the performance of DependEX. Fig. 4 (c) illustrates the quantitative results of DependEX with varying embedding

[3]Smoothing is widely used in the ML community to avoid showing the extreme values in a dataset.

size. The figure shows that the accuracy first rises and then falls with the increase of the embedding size. We observe that our model exhibits the best performance with an embedding size of 240. When the size exceeds more than 240, the accuracy reveals a decreasing trend overall. One potential reason is that as the model complexity increases, it is more likely to fall into a local optimum during the training stage.

D. Case Studies

In this section, we evaluate the usefulness of our approach in two case studies: *automatic form filling* and *test generation from NL instructions*.

The dataset for quantitative evaluation contains a total of 30 form entries, which are randomly sampled from invisible-web.net. The website includes about 1,000 deep Web sources across 18 top-level domains. We choose this dataset based on the following reasons: 1) The dataset is publicly available. 2) It has been used in previous research studies. For instance, Zhang *et al.* [30] proposed a hybrid model that integrates a 2P grammar and a global mechanism to capture the hidden syntax of forms using this dataset.

Automatic form filling. It helps reduce the repetitive behavior of end-users, thereby increasing work productivity. To evaluate the usefulness of our approach in automatic form filling, we conduct a comparison with the Autofill Form plugin of Mozilla Firefox[4], which serves as a baseline method. Specifically, the key-value pairs predefined in the plugin are used as input. The experiment is carried out according to the following steps. First, we employ DependEX to identify descriptive labels associated with input elements. Second, there are two different strategies for selecting default values based on labels: semantics similarity[5] or fuzzy matching, which correspond to DependEX-S and DependEX-E in Table II, respectively. Note that the plugin also adopts a fuzzy matching strategy. Third, we enter the identified values into the corresponding form fields. Fourth, we use the Firefox browser to open the same page and the plugin to fill out forms automatically. Last, we calculate the precision and recall of our approach and the plugin using the following equations.

$$precision = \frac{|Correct\ filled\ input\ fields|}{|Filled\ input\ fields|} \quad (22)$$

$$recall = \frac{|Correct\ filled\ input\ fields|}{|Input\ fields\ need\ to\ be\ filled|} \quad (23)$$

The experimental results show that our approach using a fuzzy matching strategy outperforms than the Firefox plugin in automatic form filling. More specifically, DependEX-E achieves average precision and recall of 90.32% and 69.14%, respectively, while the plugin yields that of 83.33% and 37.04%, respectively. The main reason is the lack of descriptive annotations or incorrect attributes in form elements due to poor development behaviors. For this case, the plugin will completely fail in form filling. Therefore, it is not feasible to rely solely on attribute matching like plugins.

[4]https://addons.mozilla.org/en-US/firefox/addon/autofill-forms-webextension

[5]https://github.com/JennaBellassai/phrase-similarity

Fig. 6: One failure example of the form filling plugin.

TABLE II: Comparison of our approach comparing with the corresponding baselines in two case studies: automatic form filling and test generation from NL instructions.

Task	Method	Metric	
Automatic form filling	Firefox plugin	precision	83.33%
		recall	37.04%
	DependEX-E	precision	90.32%
		recall	69.14%
	DependEX-S	precision	**94.59%**
		recall	**86.42%**
Test generation from NL instructions	LRTB	pass rate	86.67%
	DependEX	pass rate	**93.33%**

Fig. 6 illustrates one representative failure of the form filling plugin. By default, the plugin uses a fuzzy matching strategy to map the name attribute value of input elements to keywords in the predefined user profile. However, in this example, the name attribute is completely absent in input elements. In contrast, DependEX-E identifies corresponding semantic descriptions by modeling the dependencies between elements, thereby filling in input fields correctly. The results demonstrate that DependEX is useful for automatic form filling.

Another important observation is that the semantic similarity strategy performs better than fuzzy matching. Specifically, the precision and recall values of DependEX-S are 94.59% and 86.42%, which are 4.36% and 17.28% higher than that of DependEX-E, respectively. The reason is that input fields, corresponding to some labels with at least one match based on semantic similarity and no matches based on fuzzy search, can be correctly filled in. For instance, a predefined user profile contains the '*lastname*' keyword, while the description of input elements is 'surname'. In this scenario, DependEX-S exhibits better performance than DependEX-E.

Test generation from NL instructions. It refers to validating the software based on NL instructions provided by a group of freelance testers in a real-world user environment. Compared with traditional methods, it can detect software defects more quickly and affordably. Next, we investigate the validity of our approach in test generation based on test reports, which involve a set of natural language instructions. Note that here we focus primarily on mobile Web forms. For evaluation, we define a baseline model called LRTB to

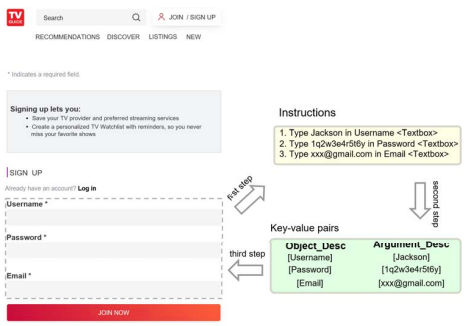


Fig. 7: The description of test instructions.

compare with our approach, which processes instructions in sequence, tests form elements from left to right and top to bottom, and randomly chooses an action for interaction. Our goal is to quantitatively assess the pass rate of given test reports for our method and LRTB.

The experiments involve five main steps. In the first step, we recruit undergraduate students to provide test instructions for a given UI. The second step is to convert instructions into key-value pairs using natural language processing (NLP) techniques (stop words removal, part-of-speech tagging, etc.). The third step is to test form elements whose labels are semantically similar to a given key and choose an action for interaction based on DependEX. The fourth step is to operate under the rules defined by LRTB. In the last step, we compute the pass rate of test reports for DependEX and LRTB using the following formula.

$$pass\_rate = \frac{|Passed\ test\ reports|}{|Test\ reports\ need\ to\ be\ executed|} \quad (24)$$

Fig. 7 shows an example of manually written test instructions. First, the recruited students provide the corresponding NL instructions for the 'Username', 'Password', and 'Email' input fields. Next, we extract the phrase pairs that describe each operation from the instructions, which includes its object and arguments (lower right of Fig. 7). Finally, we identify the form elements to be tested under the rules of DependEX or LRTB based on the UI tree and screenshots and use Appium [31] to enter the extracted pairs into the input elements and submit the form. In this case, there is a test failure for LRTB because it fills 'Jackson', '1q2w3e4r5t6y', and 'xxx@gmail.com' into the 'Search', 'Username', and 'Password' textboxes, respectively.

The experimental results indicate that our model outperforms LRTB in this application. Specifically, the overall pass rate of DependEX is 93.33%, while LRTB is 86.67%. With careful examination, we find that this can be attributed to the following reasons: 1) LRTB cannot locate the tested input elements correctly due to the presence of multiple forms. 2) There are multiple actions in a form, which may make LRTB unavailable for identifying the action that leads to a valid form submission. 3) The UI layout changes caused by the diversity of mobile devices may invalidate the rule-based LRTB method.

V. Discussions

A. Threats to Validity

Internal Threats. Hyperparameters settings are the main internal threat for all models except for RS and GP, which can affect the classification performance to a certain degree. In order to reduce the impact on the results, all parameters are set to the default values whenever possible. For cases where default configurations are not available, we select the best result in a small-scale setting for evaluation, in order to minimize this threat.

External Threats. In the second case study, we only recruited two undergraduate students to provide test reports of given forms. However, in real situations, there are a large number of freelance testers with various instruction styles. There may exist a performance bias in converting test instructions into key-value pairs. Indeed, NLP techniques have been developed for many years. We only need to select the appropriate processing steps and technology. Therefore, this threat can be significantly reduced.

B. Limitations

As mentioned in Section 4, our approach is a promising technique with outstanding performance compared with other competitive baselines. However, it also has several unresolved limitations, such as it may involve a relatively time-consuming process. For instance, given a form input, unlike recognition methods that can directly output the corresponding label, our method needs to match it against all candidate descriptions. We will consider designing a recognition-based model to capture dependencies between UI elements in our future work.

C. Performance Considerations

An important consideration is the computational complexity and the memory overhead of DependEX. In terms of GPU memory, main memory, and time, the training phase takes 2.58 GB, 2.64 GB, and 98 minutes, respectively, while the inference phase requires 2.43 GB, 2.0 GB, and 0.1658 seconds, respectively. We can see that the fast inference time could meet the real-time deployment requirements.

D. Other Potential Application Scenarios

Besides the two cases presented in this work, we expect DependEX can be applied in many a wide range of scenarios. We observe that mobile Web designs typically follow certain common patterns. For example, UI elements are arranged according to specific rules, such as from top to bottom and left to right. Due to the characteristics of these patterns, we believe DependEX can also be applied in many other cases such as:

Reference games. In this scenario, the system needs to select the corresponding reference object for a given utterance. Most existing methods [9] determine objects based on the semantic similarity of utterances and attributes. However, such attributes are not available in many objects due to the poor development behavior of programmers. This has been confirmed by more recent studies [8], [9]. It is necessary to

identify descriptive labels associated with objects, which can be extended easily based on DependEX.

Building task-oriented bots from mobile apps. The core idea of building bots is to transform the logic of user tasks into a conversation. Prior work [6] has explored the question-answer interface generation using a hybrid model, which integrates rule-based and neural network methods. However, they adopted a method of aggregating user interaction traces collected by app developers to understand the form elements during the question generation process. This is undoubtedly an expensive procedure that can be optimized with our proposed method, as our experiments indicate that DependEX performs well in identifying dependencies between UI elements.

VI. Related Work

Form understanding has been investigated extensively for many years [9], [10], [13], [30]. A variety of methods have been proposed to unlock the hidden contents of mobile applications [3], [5], [7]. Khare *et al.* [14] systematically reviewed existing methods, which fall into two general categories: rule and heuristic approaches and machine-learning-based approaches.

A. Rule-based and Heuristic Approaches

Many work have attempted to understand dependencies in forms based on a bundle of rules, including GUI design rules, hand-crafted rules, and heuristic rules [9], [32], [33]. For instance, Becce *et al.* [32] proposed to identify the descriptive information of widgets based on Gestalt principles (proximity, homogeneity, and closure). Besides these principles, Wanwarang *et al.* [7] also added an enclosure metric on the Android view hierarchy, which means that form inputs can provide a brief description of themselves, thereby visually increasing the richness of GUI. However, they are all based on the assumption that GUI designs follow a hidden pattern, which implies that all descriptions appear at the top/left of form inputs.

Moreover, a series of methods based on hand-crafted rules have been developed to understand web query interfaces [4], [10], [30], [33]–[35]. For example, He *et al.* [33] proposed a two-step method to perform interface extraction: First, the interface expression, which includes multiple basic items (elements, labels, row delimiters, etc.), was extracted from the form layout. In the second step, elements and labels were grouped into logic attributes based on expressions. Besides, Zhang *et al.* [30] formalized interface understanding as a parsing problem. More specifically, a 2P grammar, which incorporates several production rules, was developed to capture common design patterns. A global parsing mechanism was proposed to understand the semantics of interfaces systematically.

There are also some studies based on heuristic rules to interpret forms [9]–[11], [36]. Pasupat *et al.* [9] proposed to employ an embedding-based model that incorporates spatial context to understand web elements such as text boxes. More specifically, one sample was randomly selected from all neighboring label texts of the element as its semantics. Experimental results demonstrate that this strategy is not suitable for identifying associated labels of elements. Raghavan *et al.* [11] developed an application-specific crawler called HiWE, which combines visual layouts and heuristic rules, to extract content from the hidden web. The HiWE first computes the pixel distances between the form field and candidate text pieces. Then, a series of heuristic rules based on multiple factors (font size, direction, etc.) are employed to select one of these candidate labels. However, these approaches showed limited effectiveness in capturing dependencies between elements and descriptions.

B. Machine-Learning-based Approaches

In contrast to the methods mentioned above, the traditional machine learning methods can adaptively model the relationship between features hand-designed by domain experts [5], [12]–[14], [37], [38]. Several researchers attempted to employ Naïve Bayes and Decision Tree classifiers to learn dependencies between form elements and labels. For instance, Nguyen *et al.* [5] proposed a novel hierarchical classifier called LABELEX to extract form layout patterns. The LABELEX combines different kinds of features, including the string similarity between candidate descriptions and elements, spatial features, etc. However, they did not take field groups into account. In addition, Santiago *et al.* [12] developed a hybrid approach to extract implicit form constraints. More specifically, several techniques, including machine learning and natural language processing, were exploited to extract the semantics of web forms (element types and constraints). Based on the extracted semantics with constraint information, constraint solvers that combine boundary value analysis and equivalence class partitioning algorithms were applied to generate test inputs. However, the small sample size limits the scalability of these methods.

VII. Concluding Remarks

In this paper, we propose DependEX, a deep-learning-based approach to identify dependencies between UI elements within forms. Inspired by recent developments in deep learning, we design CNN-based models to extract semantic features from UI images and apply a multilayer transformer encoder module that incorporates relative positions of elements to capture contextual patterns. Moreover, two fully connected layers are employed to learn discriminative features to identify dependencies between UI elements. Extensive experiments show that our approach offers promising performance compared with other competitive models.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable feedback. This work was supported in part by the National Key Research and Development Program under Grant 2017YFB1001904 and in part by the National Natural Science Foundation of China under Grant 61772042.

REFERENCES

[1] Z. Bar-Yossef and M. Gurevich, "Random sampling from a search engine's index," *Journal of the ACM (JACM)*, vol. 55, no. 5, pp. 1–74, 2008.

[2] W. Su, H. Wu, Y. Li, J. Zhao, F. H. Lochovsky, H. Cai, and T. Huang, "Understanding query interfaces by statistical parsing," *ACM Transactions on the Web (TWEB)*, vol. 7, no. 2, pp. 1–22, 2013.

[3] S. Raghavan and H. Garcia-Molina, "Crawling the hidden web," in *VLDB*, 2001, pp. 129–138.

[4] O. Kaljuvee, O. Buyukkokten, H. Garcia-Molina, and A. Paepcke, "Efficient web form entry on pdas," in *Proceedings of the 10th international conference on World Wide Web*, 2001, pp. 663–672.

[5] H. Nguyen, T. Nguyen, and J. Freire, "Learning to extract form labels," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 684–694, 2008.

[6] T. J.-J. Li and O. Riva, "Kite: Building conversational bots from mobile apps," in *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, 2018, pp. 96–109.

[7] T. Wanwarang, N. P. Borges Jr, L. Bettscheider, and A. Zeller, "Testing apps with real-world inputs," in *Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test*, 2020, pp. 1–10.

[8] J. Chen, C. Chen, Z. Xing, X. Xu, L. Zhu, G. Li, and J. Wang, "Unblind your apps: Predicting natural-language labels for mobile gui components by deep learning," *arXiv preprint arXiv:2003.00380*, 2020.

[9] P. Pasupat, T.-S. Jiang, E. Z. Liu, K. Guu, and P. Liang, "Mapping natural language commands to web elements," *arXiv preprint arXiv:1808.09132*, 2018.

[10] W. Wu, A. Doan, C. Yu, and W. Meng, "Modeling and extracting deep-web query interfaces," in *Advances in Information and Intelligent Systems*. Springer, 2009, pp. 65–90.

[11] S. Raghavan and H. Garcia-Molina, "Crawling the hidden web," Stanford, Tech. Rep., 2000.

[12] D. Santiago, J. Phillips, P. Alt, B. Muras, T. M. King, and P. J. Clarke, "Machine learning and constraint solving for automated form testing," in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2019, pp. 217–227.

[13] T. Furche, G. Gottlob, G. Grasso, X. Guo, G. Orsi, and C. Schallhart, "Opal: automated form understanding for the deep web," in *Proceedings of the 21st international conference on World Wide Web*, 2012, pp. 829–838.

[14] R. Khare, Y. An, and I.-Y. Song, "Understanding deep web search interfaces: A survey," *ACM SIGMOD Record*, vol. 39, no. 1, pp. 33–40, 2010.

[15] M. Jain, J. C. Van Gemert, and C. G. Snoek, "What do 15,000 object categories tell us about classifying and localizing actions?" in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 46–55.

[16] G. Papandreou, T. Zhu, N. Kanazawa, A. Toshev, J. Tompson, C. Bregler, and K. Murphy, "Towards accurate multi-person pose estimation in the wild," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 4903–4911.

[17] L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille, "Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs," *IEEE transactions on pattern analysis and machine intelligence*, vol. 40, no. 4, pp. 834–848, 2017.

[18] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2014, pp. 580–587.

[19] K. He, X. Zhang, S. Ren, and J. Sun, "Identity mappings in deep residual networks," in *European conference on computer vision*. Springer, 2016, pp. 630–645.

[20] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," in *Advances in neural information processing systems*, 2015, pp. 91–99.

[21] Y. LeCun, Y. Bengio *et al.*, "Convolutional networks for images, speech, and time series," *The handbook of brain theory and neural networks*, vol. 3361, no. 10, p. 1995, 1995.

[22] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[23] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[24] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[25] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[26] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 4700–4708.

[27] A. Zeggada, F. Melgani, and Y. Bazi, "A deep learning approach to uav image multilabeling," *IEEE Geoscience and Remote Sensing Letters*, vol. 14, no. 5, pp. 694–698, 2017.

[28] M. Niepert, M. Ahmed, and K. Kutzkov, "Learning convolutional neural networks for graphs," in *International conference on machine learning*. PMLR, 2016, pp. 2014–2023.

[29] N. Parmar, A. Vaswani, J. Uszkoreit, L. Kaiser, N. Shazeer, A. Ku, and D. Tran, "Image transformer," in *International Conference on Machine Learning*. PMLR, 2018, pp. 4055–4064.

[30] Z. Zhang, B. He, and K. C.-C. Chang, "Understanding web query interfaces: Best-effort parsing with hidden syntax," in *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, 2004, pp. 107–118.

[31] Appium, "Project homepage," 2016. [Online]. Available: http://appium.io/

[32] G. Becce, L. Mariani, O. Riganelli, and M. Santoro, "Extracting widget descriptions from guis," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2012, pp. 347–361.

[33] H. He, W. Meng, C. Yu, and Z. Wu, "Automatic extraction of web search interfaces for interface schema integration," in *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, 2004, pp. 414–415.

[34] S. M. Benslimane, M. Malki, M. K. Rahmouni, and D. Benslimane, "Extracting personalised ontology from data-intensive web application: an html forms-based reverse engineering approach," *Informatica*, vol. 18, no. 4, pp. 511–534, 2007.

[35] H. He, W. Meng, Y. Lu, C. Yu, and Z. Wu, "Towards deeper understanding of the search interfaces of the deep web," *World Wide Web*, vol. 10, no. 2, pp. 133–155, 2007.

[36] E. C. Dragut, T. Kabisch, C. Yu, and U. Leser, "A hierarchical approach to model web query interfaces for web source integration," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 325–336, 2009.

[37] R. Khare and Y. An, "An empirical study on using hidden markov model for search interface segmentation," in *Proceedings of the 18th ACM conference on Information and knowledge management*, 2009, pp. 17–26.

[38] W. Wu, C. Yu, A. Doan, and W. Meng, "An interactive clustering-based approach to integrating source query interfaces on the deep web," in *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, 2004, pp. 95–106.